Edícia výskumných textov
informatiky a informačných technológií

Štúdie vybraných tém softvérového inžinierstva

Pokročilé metódy navrhovania programových systémov
Pokročilé metódy získavania, vyhľadávania, reprezentácie a prezentácie informácií

Kniha vznikla a bola vydaná s finančnou podporou projektu:

Projekt JPD 3 2004/1-022: Podpora vzdelávania mladých vedeckých pracovníkov s cieľom vychovať tvorivých expertov – profesionálov informatikov – pre modernú spoločnosť založenú na vedomostiach

Európsky sociálny fond pomáha rozvíjať zamestnanosť podporovaním zamestnateľnosti, obchodného ducha, rovnakých príležitostí a investovaním do ľudských zdrojov.

Mária Bieliková,
Pavol Návrat
a kol.

Štúdie vybraných tém softvérového inžinierstva

Pokročilé metódy navrhovania programových systémov
Pokročilé metódy získavania, vyhľadávania, reprezentácie a prezentácie informácií

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Edícia výskumných textov informatiky a informačných technológií

Štúdie vybraných tém softvérového inžinierstva

Pokročilé metódy navrhovania programových systémov
Pokročilé metódy získavania, vyhľadávania, reprezentácie
a prezentácie informácie

Autorský kolektív:

Mária Bieliková
Pavol Návrat
Anton Andrejko
Peter Blšták
György Frivolt
Vladimír Grlický
Jaroslav Jakubík
Matej Košík
Jaroslav Kuruc
Marián Lekavý
Vladimír Marko
Matúš Navarčík

Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave
Ilkovičova 3
842 16 Bratislava

© Mária Bieliková, Pavol Návrat a kol., 2006

Text Design & Composition: Mária Bieliková
Copy Editor: Anton Andrejko
Cover Designer: Peter Kaminský

Vydala Slovenská technická univerzita v Bratislave
vo Vydavateľstve STU, Bratislava, Vazovova 5.

ISBN

PREDHOVOR

Publikácia, ktorá sa vám dostala do rúk, vznikla na základe seminárov študentov doktorandského štúdia študijného programu programové systémy v odbore softvérové inžinierstvo. Semináre boli podporené projektom Európskych štrukturálnych fondov, ktorého hlavným cieľom je podpora vzdelávania prostredníctvom motivačných nástrojov pre doktorandov a zvyšovaním kvality vzdelávania v treťom stupni vysokoškolského štúdia v oblasti informatiky a informačných technológií.

Informatika a informačné technológie sú kľúčovým prvkom budovania modernej spoločnosti založenej na vedomostiach. Mladí talentovaní absolventi druhého stupňa vysokoškolského štúdia v oblasti informatiky alebo príbuzných oblastiach majú v súčasnosti veľké možnosti uplatnenia sa v praxi. Informačná spoločnosť však potrebuje aj špecializovaných odborníkov a vedeckých pracovníkov s ukončeným tretím stupňom vysokoškolského štúdia v študijných odboroch skupiny informatických vied, informačných a komunikačných technológií tak, aby bolo možné budovať ekonomiku založenú na najnovších vedeckých poznatkoch. S tým súvisí potreba profesionálov v oblasti uchovávania, spracúvania a prezentácie informácií v bohatej palete reprezentácií ako základného prvku informačnej spoločnosti.

S rozvojom informatiky a informačných technológií a s posunom spoločnosti k informačnej spoločnosti, resp. spoločnosti založenej na vedomostiach, vzniká potreba vychovávať odborníkov v špecializovaných oblastiach. Semináre, ktoré sa uskutočňujú na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave v rámci doktorandského štúdia a podporené projektom sa zameriavajú na oblasť programových systémov, ktorá zahŕňa najrôznejšie aspekty softvérového inžinierstva od analýzy, návrhu, implementácie a testovania až po manažment verzií programových systémov, manažment kvality a softvérových projektov a v oblasti spracovania informácií.

Našou ambíciou bolo sprístupniť záujemcom o softvérové inžinierstvo vybrané témy a tým zdieľať výsledky seminárov a tvorivého prístupu študentov k jednotlivým témam v rámci diskusií.

Výskumné texty v tejto publikácii sú vhodné aj pre študentov ďalších študijných programov v odboroch ako napr. informatika, aplikovaná informatika, informačné systémy, či umelá inteligencia a to v študijných programoch uskutočňovaných na Slovenskej technickej univerzite v Bratislave a aj na iných univerzitách.

Publikácia pozostáva z dvoch častí, v prvej (Diel 1: Návrhové vzory) sa sústreďujeme na analýzu návrhových vzorov, ktoré predstavujú jednu s kľúčových vyvíjajúcej sa disciplíny softvérového inžinierstva. Druhá časť (Diel 2: Vybrané témy programových a informačných systémov) obsahuje päť štúdií, ktoré diskutujú a analyzujú vybrané

otvorené vedecké problémy z dynamicky sa rozvíjajúcej oblasti programových systémov so špeciálnym dôrazom na programové informačné systémy aj v spojitosti s internetom.

Diel 1: Návrhové vzory

Návrhové vzory sú významným metodologickým príspevkom v rozvoji softvérového inžinierstva ako vedeckej disciplíny. V prvej polovici deväťdesiatych rokov minulého storočia došlo nielen k explicitnému zvýrazneniu pojmu softvérovej architektúry a rozbehol sa jej výskum, ale aj sa postupne stávalo bežným explicitné používanie tohto pojmu pri vývoji softvérových systémov. Ako to už býva pri vzniku nového pojmu, nebolo celkom jasné, čo znamená. Ako to už býva pri vzniku novej vedeckej disciplíny, k určeniu jej obsahu výrazne prispievajú vedecké monografie, ktoré sa mu venujú. V prípade softvérovej architektúry sa nedá obísť monografia Shaw, Garlan (Software architecture: perspectives on an emerging discipline. Prentice-Hall, Inc., 1996). Určite to nebola učebnica – na vznik učebnice bola disciplína ešte príliš nezrelá, učebnica nemohla vzniknúť. Koniec koncov, vznik učebnice predpokladá, že sa príslušná disciplína niekde na univerzite vyučuje. Avšak zaviesť niekde takúto novú univerzitnú disciplínu nie je vonkoncom jednoduché. Musí tomu predchádzať výskum v príslušnej oblasti. Je to len ilustráciou všeobecne uznávanej pravdy, že výskum a vzdelávanie sa na univerzite musia rozvíjať ruka v ruke.

V našom prípade sme naozaj začali skúmať vybrané otázky, týkajúce sa architektúry softvérových systémov ako vedeckej disciplíny, v druhej polovici deväťdesiatych rokov minulého storočia. To nám umožnilo, aby sme mohli zaviesť predmet Architektúra softvérových systémov do študijného programu na druhom stupni vysokoškolského štúdia. Samozrejme, oblasť architektúry softvérových systémov je značne široká a náš výskum, ak mal mať nádej na dosiahnutie významnejších výsledkov, sa musel zamerať na vybrané otázky. V našom prípade sme si zvolili tému návrhových vzorov, špeciálne v tom zmysle, ako ju zaviedli autori GoF (E. Gamma, R. Helm, R. Johnson, J. Vlissides.: Design Patterns: Elements of Reusable Object-Oriented Software. 1st edition, Addison-Wesley, 1995). Ide o monografiu, ktorá prelomovým spôsobom poznačila vývoj disciplíny. Dodnes nie je svojím spôsobom prekonaná. Súčasne však treba povedať, že podnietila rozsiahly výskum, ktorý smeroval okrem rozšírenia toho, čo dosiahla, aj v podstate k jej prekonaniu. Tento výskum pokračuje a bolo by len prirodzené, ak by sme sa aj my na takom výskume podieľali. Naozaj, v uplynulom období sa podarilo dosiahnuť viacero pôvodných výsledkov v tejto oblasti.

Ak teda celkom pokojne akceptujeme, že monografiu GoF, akokoľvek je míľnikom, alebo práve preto, treba prekonať, tak súčasne sme uzrozumení s tým, že prvým krokom je pochopiť základné myšlienky, ktoré priniesla. Ďaleko viac: mali by sme analyzovať základné princípy a štruktúry, ktoré zaviedla a pokiaľ možno kriticky zhodnotiť ich význam ako z hľadiska teoretického prínosu pre disciplínu architektúry softvérových systémov, tak z hľadiska praktického prínosu pre prax vyvíjania softvérových systémov.

Toto je asi hlavný dôvod, prečo považujeme štúdium návrhových vzorov za potrebné pre študentov doktorandského štúdia. Ak si osvoja pojem návrhový vzor, oboznámia sa s jeho princípom ako aj rôznymi druhmi a v rámci nich s jednotlivými typickými prípadmi, tak budú lepšie pripravení na to, aby návrhové vzory buď rozvinuli, alebo

zavrhli. Samozrejme, na to nestačí oboznámiť sa len s návrhovými vzormi. Tento text však má realistický cieľ, ktorým je kritický opis jednotlivých najdôležitejších návrhových vzorov.

Návrhové vzory sme v súlade s pôvodnou monografiou GoF začlenili do troch skupín:

- vzory vytvárania,
- štrukturálne vzory,
- vzory správania sa.

Navyše, ako odraz súčasného vývoja, sme zaradili do tohto textu aj

- vzory J2EE.

Každému vzoru venujeme opis, ktorý nie je len mechanickým prevzatím myšlienok z pôvodnej monografie GoF, ale má snahu o istú aktualizáciu a kritický pohľad.

Vytvorený text však aktualizáciu a kritický pohľad skôr naznačuje: je predsa viac na čitateľovi, študentovi, ktorý sa podujal bádať v oblasti softvérového inžinierstva, aby kritický pohľad prehĺbil, samotný pojem návrhového vzoru rozpracoval a v konečnom dôsledku hypotézu o vhodnosti návrhových vzorov na vyjadrenie niektorých vedomostí používaných vo vývoji softvérových systémov buď potvrdil alebo zavrhol.

Opis každého návrhového vzoru sleduje jednotnú štruktúru, ktorá má inšpiráciu v štruktúre pôvodného výkladu v GoF. Každý opis návrhového vzoru je výsledkom tvorivej činnosti, ku ktorej prispeli viacerí. Samotný text každého návrhového vzoru písal ten-ktorý doktorand a jeho autorský prínos treba čo najvýraznejšie zdôrazniť a oceniť. Prvé verzie opisov predniesli doktorandi na seminároch v rámci doktorandského štúdia, ktoré viedol Pavol Návrat. Na seminároch prebiehala diskusia, na ktorej sa zúčastňovala celá skupina doktorandov a ktorá v tom-ktorom prípade ovplyvnila definitívne znenie opisu. Napriek tomu považujeme za korektné, aby sme označili ako jediných autorov jednotlivých opisov doktorandov, ktorí im dali písomnú podobu.

Autori sa podieľali na jednotlivých kapitolách takto:

- Vzory vytvárania: Peter Blšták
- Štrukturálne vzory: György Frivolt, Matej Košík
- Vzory správania: Anton Andrejko
- J2EE vzory: Jaroslav Jakubík

Diel 2: Vybrané témy programových a informačných systémov

Do druhej časti zaraďujeme päť štúdií, ktoré sa venujú vybraným otvoreným vedeckým problémom, týkajúcim sa programových a informačných systémov. Ide o oblasti, v ktorých prebieha veľmi intenzívny vývoj. Programové systémy sa stávajú systémami, pôsobiacimi v čoraz rôznorodejšom prostredí, vrátane internetu. Stávajú sa súčasťou čoraz komplexnejších systémov – na jednej strane rozsiahlych informačných systémov, na druhej strane systémov, spolu určených technickou platformou, ktorou už dávno nie je len počítač v klasickom slova zmysle, ale aj najrôznejšie vnorené systémy, (tele-)komunikačné systémy, atď.

Informačné systémy sa stávajú univerzálnym modelom spôsobov vyhľadávania, získavania, sprístupňovania, uchovávania, odovzdávania, spoločného používania, prezentovania informácií. I keď sa v zásade dá na ne nazerať odhliadnuc od toho, či sú operácie a procesy podporené počítačom alebo nie, čoraz viac sa zväčšuje praktický význam informačných systémov, ktoré sú realizované pomocou programových systémov (a tie samozrejme pomocou počítačových systémov alebo iných technických systémov, zahŕňajúcich počítače). Je to najmä preto, že softvérovo podporené informačné systémy majú vďaka možnostiam, ktoré poskytuje naprogramovaný počítač, výhody, ktoré sa ručným spracovaním nedajú dosiahnuť. Toto je súčasne aj argumentom pre úzke prepojenie výskumu v oboch oblastiach – ako softvérového inžinierstva, tak aj informačných systémov.

Štúdie sú výsledkom práce doktorandov v rámci ich doktorandského štúdia. Možno nezaškodí pripomenúť, že doktorandské štúdium sa koná pod vedením školiteľa. Na každej štúdii má preto podiel aj príslušný školiteľ. Napriek tomu však považujeme za korektné, aby sme označili ako jediných autorov jednotlivých štúdií doktorandov, ktorí im dali písomnú podobu a ktorí ich predložili a úspešne obhájili ako písomnú časť svojej dizertačnej skúšky.

Autori sa podieľali na jednotlivých kapitolách takto:

- Vývoj softvéru založený na vzoroch: Vladimír Marko (školiteľ prof. Pavol Návrat)
- Formalizácia softvérových architektúr: Matúš Navarčík (školiteľ prof. Pavol Návrat)
- Multiagentové systémy: Marián Lekavý (školiteľ prof. Pavol Návrat)
- Modelovanie adaptívnych webových systémov: Jaroslav Kuruc (školiteľ: prof. Mária Bieliková)
- Prezentácia informácií a znalostí na webe so sémantikou: Vladimír Grlický (školiteľ prof. Pavol Návrat)

Dúfame, že text poslúži záujemcom o programové a informačné systémy, umožní zdieľať výsledky štúdia v tejto oblasti. Tešíme sa na prípadné odozvy, pripomienky, ktoré budú môcť prispieť k ďalším takýmto publikáciám.

Marec 2006,
Bratislava

Mária Bieliková a Pavol Návrat

OBSAH

DIEL I: NÁVRHOVÉ VZORY

1 VZORY VYTVÁRANIA 5

1.1 Abstraktná továreň 5
1.2 Staviteľ 7
1.3 Výrobná metóda 11
1.4 Prototyp 14
1.5 Unikát 16

2 ŠTRUKTURÁLNE VZORY 19

2.1 Adaptér 19
2.2 Premostenie 21
2.3 Zloženina 24
2.4 Dekoratér 27
2.5 Fasáda 29
2.6 Mušia váha 31
2.7 Zástupca 33

3 VYBRANÉ VZORY SPRÁVANIA 37

3.1 Príkaz 37
3.2 Reťaz zodpovednosti 40
3.3 Interpret 43
3.4 Iterátor 45
3.5 Sprostredkovateľ 46
3.6 Memento 49

4 J2EE VZORY 53

4.1 Klientska abstrakcia biznis služieb 53
4.2 Dáta pristupujúci objekt 55
4.3 Abstrakcia biznis služieb 57
4.4 Prenositeľ 60

DIEL II: VYBRANÉ TÉMY PROGRAMOVÝCH A INFORMAČNÝCH SYSTÉMOV

5 PATTERN-BASED SOFTWARE DEVELOPMENT 67
5.1 Pattern Systems and Languages 68
5.2 Static Properties of Design Patterns 80
5.3 Dynamic Properties of Design Patterns 89
5.4 Conclusions 92
References 93

6 FORMALIZATION IN SOFTWARE ARCHITECTURES 97
6.1 Formalization 97
6.2 Object level formalization 99
6.3 Software architecture 101
6.4 Architecture Description Languages 106
6.5 Object based approaches versus software architectures 113
6.6 Conclusion 123
6.7 Appendixes 125
References 134

7 MULTIAGENTOVÉ SYSTÉMY 137
7.1 Agent a multiagentový systém 137
7.2 Základné charakteristiky agentov 139
7.3 Organizácie agentov 144
7.4 Plánovanie v kontexte multiagentových systémov 159
Literatúra 176

8 PREZENTÁCIA INFORMÁCIÍ A ZNALOSTÍ NA WEBE SO SÉMANTIKOU 181
8.1 Ontológie 181
8.2 Web so sémantikou 184
8.3 Prezentácia informácií a znalostí na webe so sémantikou 193
8.4 Špecifikácia a generovanie prezentácie pomocou XSLT 194
8.5 Záver 200
Literatúra 203

9 MODELOVANIE ADAPTÍVNYCH WEBOVÝCH SYSTÉMOV 207
9.1 Úvod 207
9.2 Charakteristika adaptívnych hypermediálnych systémov 208
9.3 Modely adaptívnych hypermediálnych systémov 216
9.4 Záver 228
Literatúra 229

DIEL I:
NÁVRHOVÉ VZORY

Slovník pojmov

Abstract Factory	–	Abstraktná továreň
Adapter	–	Adaptér
Bridge	–	Premostenie
Builder	–	Staviteľ
Business Delegate	–	Klientska abstrakcia biznis služieb
Command	–	Príkaz
Composite	–	Zloženina
Data Access Object	–	Dáta pristupujúci objekt
Decorator	–	Dekoratér
Facade	–	Fasáda
Factory Method	–	Výrobná metóda
Flyweight	–	Mušia váha
Chain of Responsibility	–	Reťaz zodpovednosti
Interpreter	–	Interpreter
Iterator	–	Iterátor
Mediator	–	Sprostredkovateľ
Memento	–	Memento
Observer	–	Pozorovateľ
Prototype	–	Prototyp
Proxy	–	Zástupca
Session Facade	–	Abstrakcia biznis služieb
Singleton	–	Unikát
State	–	Stav
Strategy	–	Stratégia
Template Method	–	Šablónová metóda
Transfer Object	–	Prenositeľ
Visitor	–	Návštevník

Východisková literatúra

ALPERT, S. R. ET AL. (1998). The design patterns Smalltalk companion. 1st edition. Massachusetts: Addison-Wesley, pp. 444, ISBN 0-201-18462-1.

GAMMA, E. ET AL. (1995). Design Patterns. 1st edition. Massachusetts: Addison-Wesley, pp. 395, ISBN 0-201-633610-2.

GAMMA, E. AT AL. (2003). Návrh programů pomocí vzorů. 1. vyd. Praha: Grada Publishing a.s., 388 s., ISBN 80-247-0302-5.

DEEPAK, A. ET AL. (2001). Core J2EE Patterns: Best Practices and Design Strategies. 1st edition. Prentice Hall/Sun Microsystems Press, pp. 496, ISBN 0-130-64884-1.

1 VZORY VYTVÁRANIA

Návrhové vzory vytvárania zovšeobecňujú proces tvorby objektov. Zabezpečujú nezávislosť systémov (aplikácií) od toho, ako sú objekty vytvárané, komponované a reprezentované (akých typov budú konkrétne vytvorené inštancie). Odtieňujú systémy (aplikácie) od problému tvorby požadovaných objektov. Tvorba objektov aplikácie je delegovaná na iné objekty. Vzory vytvárania:

- zapuzdrujú vedomosť o tom, aké konkrétne triedy budú vytvorené a použité,
- skrývajú, ako sú inštancie vytvárané a skladané dokopy.

Aplikácia vie o konkrétnych vytvorených objektoch zväčša iba ich *interface* alebo ich abstraktnú nadtriedu. Tento fakt dáva veľkú flexibilitu vo vytváraní, t.j. čo sa vytvorí, kto to vytvorí, ako sa to vytvorí a kedy. Konfigurácia môže byť statická alebo dynamická, čiže definovaná počas behu aplikácie.

Návrhové vzory vytvárania sú v anglickej literatúre označované ako *Creational Patterns.*

1.1 Abstraktná továreň

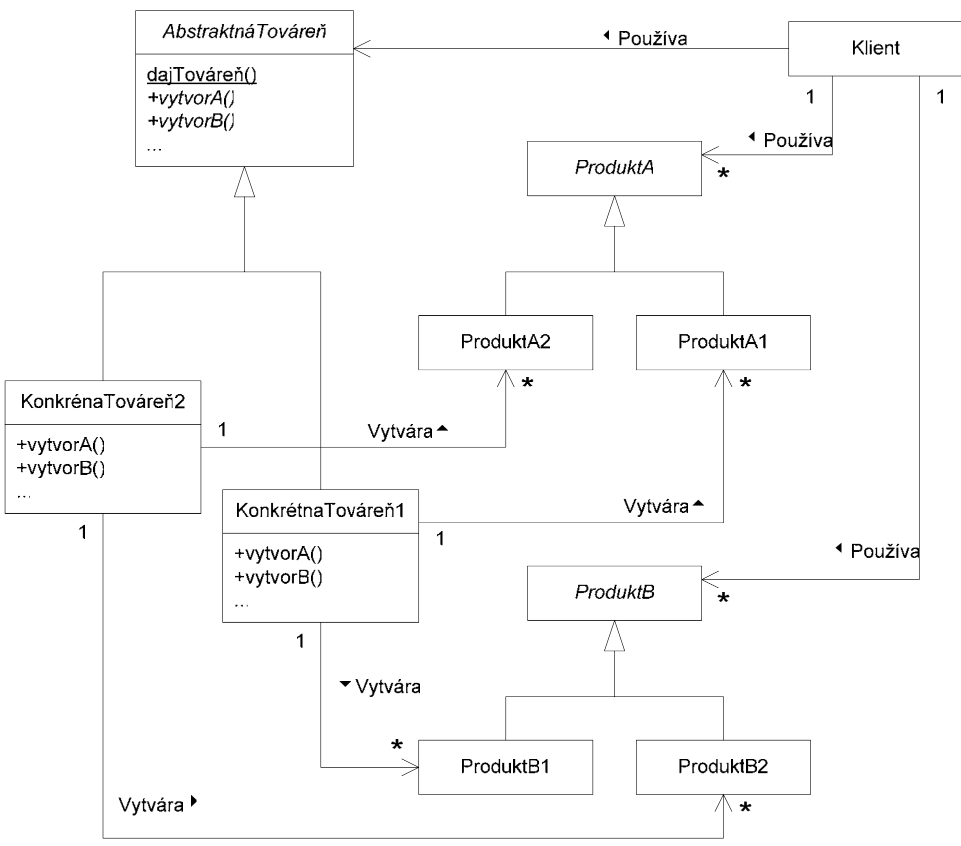
Návrhový vzor Abstraktná továreň je v anglickom jazyku známy ako *Abstract Factory*.

Účel

Definuje rozhranie pre tvorbu skupín súvisiacich alebo závislých objektov bez toho, aby boli špecifikované konkrétne triedy vytváraných objektov (produkty). Tento vzor je vhodné použiť v prípadoch, keď:

- systém má mať možnosť, aby na základe konfigurácie používal jednu zo skupín súvisiacich objektov (produktov),
- súvisiace objekty v skupine sú navrhnuté tak, aby boli použité spolu (v rámci svojej skupiny) a je potrebné zabezpečiť túto požiadavku,
- potrebujeme použiť knižnicu tried produktov, bez toho aby sme odhalili implementácie a urobili okolité aplikácie závislé od implementačných tried.

Štruktúra



Obrázok 1-1. Štruktúra vzoru Abstraktná továreň.

Súčasti

`Klient` – triedy v role klienta používajú rôzne produkty (Produkt A, B...) a využívajú ich služby. Klient pozná iba abstraktné produkty a nemá žiadnu znalosť o ich konkrétnych implementáciách.

`AbstraktnáTováreň` – definuje abstraktné metódy pre tvorbu sady súvisiacich produktov (Produkt A, B...).

`KonkrétnaTováreň1, 2` – reprezentujú konkrétne implementácie abstraktnej továrne, ktoré vracajú jednu sadu súvisiacich produkty (napr. Produkt A1, B1...).

`ProduktA, B` – abstraktné triedy alebo rozhrania, ktoré definujú základné vlastnosti vytváraných produktov v podobe svojich metód.

`ProduktA1`, A2 – konkrétne realizácie abstraktného produktu A, ktoré patria do rôznych sád produktov, ktoré sú definované Konkrétnymi továrňami (sady 1, 2...).

Dôsledky

Aplikácia je nezávislá od konkrétnych realizácií produktov, toho ako sú vytvárané, komponované a reprezentované.

Jednoduchým spôsobom sa dá vymeniť skupina súvisiacich objektov používaných aplikáciou – na jednom mieste sa vymení použitie konkrétnej implementácie Abstraktnej továrne za inú.

Vzor zabezpečuje konzistenciu použitých produktov (sú vytvárané len produkty z jednej skupiny – sady).

Podpora nových produktov je namáhavá na prácu (je potrebné modifikovať všetky podtriedy Abstraktnej továrne).

Implementácia

Implementáciu Abstraktnej továrne často realizuje návrhový vzor Výrobná metóda. To znamená, že trieda `AbstraktnáTováreň` poskytuje statickú metódu s parametrom, ktorej výsledkom je unikátna inštancia konkrétnej továrne (vzor Unikát).

Rozhranie pre tvorbu produktov, ktoré poskytuje `AbstraktnáTováreň` môže byť realizované ako:

- skupina Výrobných metód (pre každý produkt jedna),
- jedna parametrizovateľná Výrobná metóda
- alebo pomocou prototypov.

Príklad

Príkladom použitia vzoru Abstraktná továreň je AWT (*Abstract Widget Toolkit*) Java API. AWT je knižnica pre tvorbu používateľského grafického rozhrania a obsahuje sady platformovo závislých (Windows, Unix) implementácií grafických prvok, ktoré implementujú definované platformovo nezávislé rozhrania grafických prvkov. K týmto sadám existujú platformovo závislé konkrétne továrne na výrobu grafických prvkov, ktoré implementujú abstraktnú továreň `java.awt.Toolkit`.

Príbuzné vzory

Výrobná metóda – Abstraktná továreň využíva sadu Výrobných metód.

Unikát – konkrétna továreň je často realizovaná ako Unikát.

1.2 Staviteľ

V anglickej literatúre sa s týmto návrhovým vzorom môžeme stretnúť pod názvom *Builder*.

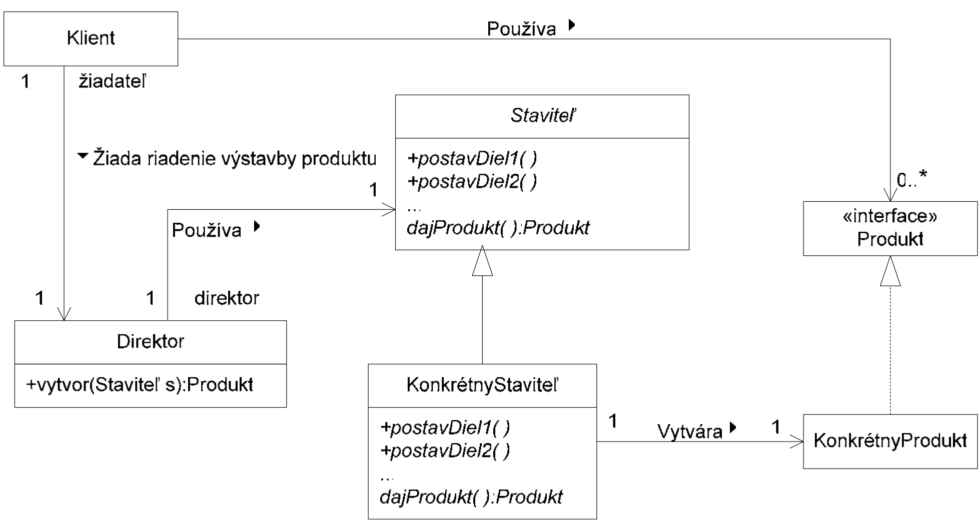
Účel

Vzor zabezpečuje oddelenie algoritmu tvorby zložitých objektov (kompozície) od vytvárania a skladania jeho jednotlivých častí tak, že ten istý proces konštrukcie produktu môže vytvoriť rôzne reprezentácie. Umožňuje znovupoužitie algoritmu pre

tvorbu viacerých podobných reprezentácií rovnakého obsahu, zväčša s iným správaním.

Vzor staviteľ sa využíva v prípadoch, keď algoritmus tvorby zložitých objektov má byť nezávislý od jednotlivých častí vytváraného objektu, príp. od toho ako sú tieto časti spájané dohromady alebo v prípadoch, keď proces tvorby musí umožniť rôzne reprezentácie vytváraného objektu (rozhrania).

Štruktúra



Obrázok 1-2. Štruktúra vzoru Staviteľ.

Súčasti

`Direktor` – trieda, ktorá definuje základný algoritmus pre vytváranie zložitých objektov (produktov). Deleguje vytváranie jednotlivých častí – dielov na Staviteľa.

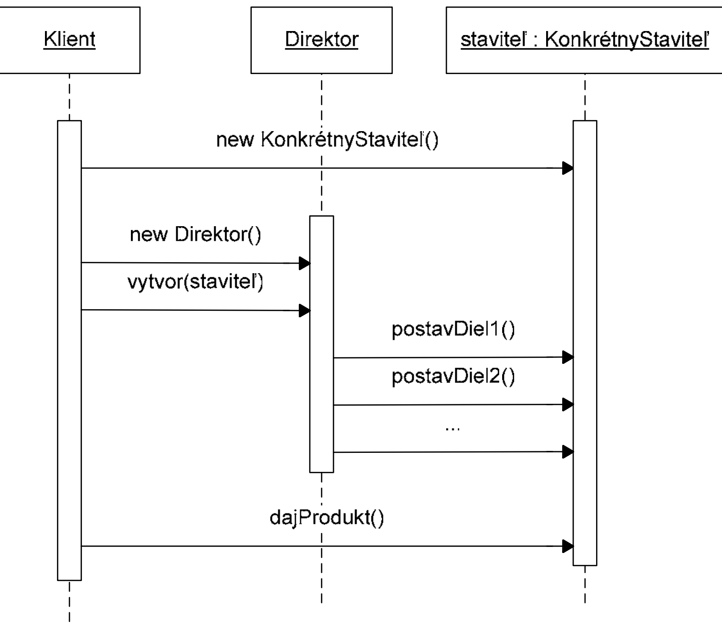
`Staviteľ` – definuje typy operácií, ktoré sú vyžadované od Staviteľa pre konkrétnu `Direktor` triedu. Ide o abstraktnú triedu alebo rozhranie.

`KonkrétnyStaviteľ` – realizuje operácie, ktoré slúžia na vytvorenie jednotlivých častí – dielov konkrétneho produktu, pričom tieto diely spolu súvisia.

`Produkt` – rozhranie, ktoré definuje správanie vytváraných objektov. Vytvárané objekty sú často zloženiny.

`KonkrétnyProdukt` – produkt, ktorý je vytvorený konkrétnym staviteľom.

`Klient` – používateľ produktov.



Obrázok 1-3. Ukážka použitia vzoru Staviteľ.

Dôsledky

Vnútorná reprezentácia vytváraných produktov sa môže líšiť.

Oddeľuje sa proces tvorby od reprezentácie (zvýšená modularita), môžeme znovupoužiť proces tvorby s rôznymi vnútornými reprezentáciami (rôzny Direktor s jedným Staviteľom alebo viac Staviteľov s jedným Direktorom).

Poskytuje jemnejšie riadenie nad procesom tvorby (tvorba krok po kroku – nie iba jeden krok).

Implementácia

Pri realizácii Staviteľa, treba dobre zvážiť jeho rozhranie (aké operácie a s akými parametrami bude mať). V jednoduchých prípadoch postačuje, keď sú vytvárané časti pripájané k čiastočne vytvorenému produktu. Vtedy si staviteľ drží vo svojej premennej čiastočný výsledok a poskytuje sadu metód na vytvorenie ďalších dielov, ktoré jednoduchým spôsobom pripája k čiastočnému výsledku.

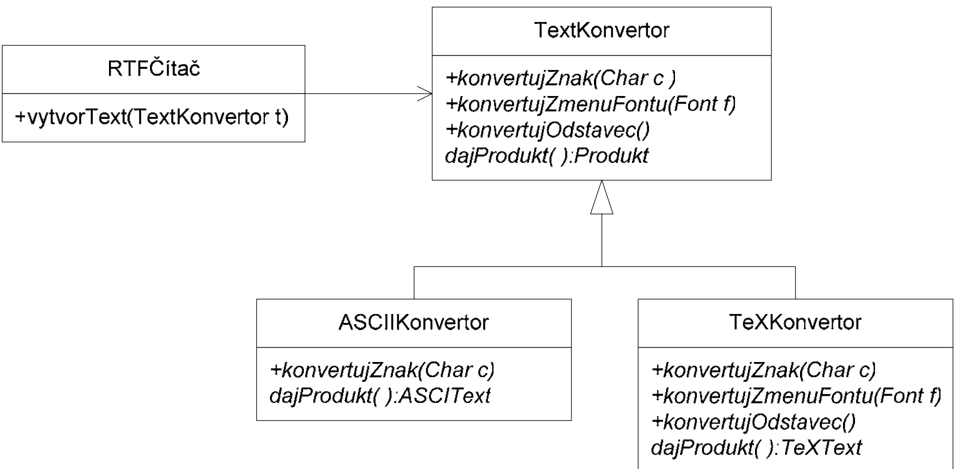
V zložitejších prípadoch, keď vytvárame hierarchicky štruktúrované objekty ako napr. reprezentáciu používateľského rozhrania (GUI), môže byť potrebné, aby Direktor dostával od Staviteľa čiastočné výsledky a posielal ich Staviteľovi ako parameter pri budovaní ďalších častí. V tomto prípade môžu mať metódy na tvorbu jednotlivých dielov vstupné parametre a výstupné hodnoty. Napr. pri vytváraní prvku formulára je potrebné poznať, v akom zloženom prvku sa vytváraný prvok nachádza – groupbox, podformulár…).

Často sa stáva, že pre vytváraný produkt sa nešpecifikuje spoločné rozhranie ani abstraktná nadtrieda. Na jednej strane, je to dané tým, že vytvárané produkty sú tak rôzne, že to nemá zmysel a často to ani nejde a na druhej strane, klient dodáva objekt konkrétneho staviteľa, a teda vie, čo bude jeho výstupom a môže použiť pretypovanie vytvoreného objektu na požadovaný typ.

V niektorých prípadoch sa využíva abstraktná implementácia triedy `Staviteľ`, ktorá definuje prázdne implementácie výrobných metód pre jednotlivé diely. Konkrétny stavitelia v tomto prípade prepíšu iba tie metódy, ktoré potrebujú pre správnu funkčnosť. Tento prístup sa využíva vtedy, keď rozhranie `Staviteľ` definuje veľké množstvo dielov, z ktorým môže byť použitých iba niekoľko.

Príklad

Na obrázku 1-4 je znázornená štruktúra príkladu požitia vzoru Staviteľ. Ide o realizáciu čítania RTF dokumentov a ich transformáciu do iných vnútorných reprezentácií pomocou konvertorov. `RTFČítač` realizuje čítanie dokumentu vo formáte RTF a volá metódy triedy `TextKonvertor`, na vytvorenie inštancií jednotlivých logických častí textu. `RTFČítač` predstavuje direktora, `TextKonvertor` reprezentuje staviteľa a `ASCIKonvertor` a `TeXKonvertor` reprezentujú konkrétnych staviteľov vzoru Staviteľ.



Obrázok 1-4. Príklad použitia vzoru Staviteľ.

Príbuzné vzory

Zloženina – výsledkom (produktom) vzoru Staviteľ je často zloženina.

1.3 Výrobná metóda

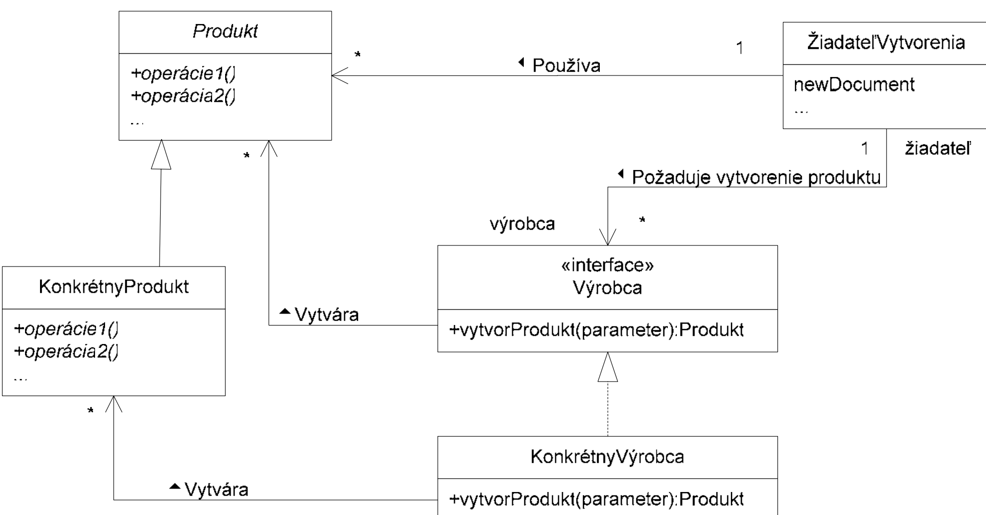
V anglickej literatúre sa na návrhový vzor odvoláva ako na *Factory Method*.

Účel

Návrhový vzor Výrobná metóda definuje rozhranie pre tvorbu objektov – produktov v podobe špeciálnej inštančnej metódy triedy, pričom rozhodnutie o tom, akú konkrétnu triedu vytvoriť, ponecháva na svoje podtriedy. Využíva sa v prípadoch, keď

- trieda nevie určiť (predvídať), aký typ objektov bude musieť vytvárať (napr. spoločná nadtrieda zabezpečujúca spoločnú funkcionalitu pre všetky typy odvodených tried, ale nepozná konkrétne objekty, s ktorými ma pracovať, resp. ktoré má vytvárať,
- trieda chce, aby jej podtriedy určili objekty, ktoré sa budú vytvárať.

Štruktúra



Obrázok 1-5. Štruktúra vzoru Výrobná metóda.

Súčasti

`Produkt` – výsledok volania výrobnej metódy – abstraktná nadtrieda vytváraných objektov (produktov) alebo rozhranie, ktoré implementujú vytvárané objekty.

`KonkrétnyProdukt` – konkrétna trieda, ktorej inštancie vytvára konkrétny výrobca.

`Výrobca` – nadtrieda, ktorá definuje výrobnú metódu, ktorá musí byť implementovaná, resp. môže byť prepísaná (*override*) konkrétnymi výrobcami (nutnosť prepísania tejto metódy závisí od toho či je abstraktná, alebo má implicitnú implementáciu).

`KonkrétnyVýrobca` – implementuje alebo prepisuje výrobnú metódu, v ktorej realizuje tvorbu objektov (produktov).

Dôsledky

Eliminuje potrebu previazania aplikačne špecifického kódu s infraštrukturálnym kódom (abstraktným, spoločným, kódom univerzálneho rámca). Pod infraštrukturálnym kódom rozumieme časť kódu, ktorá je definovaná mimo aplikačného kódu, predstavuje jadro funkcionality, ktorú chceme využiť a rozšíriť v našej aplikácií a často je dodávaná prostredníctvom knižníc. Na obrázku 1-5 predstavujú infraštrukturálny kód triedy `Výrobca` a `Produkt` a aplikačný kód zvyšné tri triedy. V kóde triedy `Výrobca` sa predpokladá použitie rozhrania `Produkt`, a teda nie je potrebné vedieť nič o aplikačnej triede `KonkrétnyProdukt`.

Umožňuje podtriedam triedy `Výrobca` predefinovať štandardnú implementáciu metódy `vytvorProdukt`. Podtriedy tak môžu jednoduchým spôsobom prispôsobiť (meniť a rozširovať) správanie nadtriedy.

Umožňuje prepojiť paralelné hierarchie tried, t.j. máme hierarchiu tried, ktorých inštancie vytvárame v aplikačnom kóde. S týmito triedami súvisia iné triedy, ktoré používame, pričom tieto triedy sú organizované v podobnej hierarchii. V princípe platí, že každá trieda z prvej hierarchie súvisí s nejakou triedou z paralelnej druhej hierarchie. Na prepojenie využijeme výrobnú metódu v prvej hierarchii, kde výsledkom jej volania bude inštancia nejakej triedy z druhej hierarchie.

Núti aplikáciu, aby vytvárala ďalšie triedy `KonkrétnyVýrobca` z nadtriedy `Výrobca` iba kvôli tomu, že chceme vytvárať inštancie ďalšej (inej) triedy `KonkrétnyProdukt`.

Implementácia

Trieda `Výrobca` môže poskytovať štandardnú implementáciu výrobnej metódy `vytvorProdukt`. Čiže výrobná metóda je alebo nie je abstraktná, a teda podtriedy ju buď musia alebo nemusia implementovať.

Výrobná metóda môže byť parametrizovateľná. Parameter bližšie určuje, aký typ inštancie má byť vytvorený, a teda výrobná metóda poskytuje sadu rôznych objektov.

Pokiaľ produkty vytvárané jedným konkrétnym výrobcom spolu súvisia, z Výrobnej metódy s parametrom sa nám stáva istá forma Abstraktnej továrne, s tým rozdielom, že Výrobná metóda vracia iba inštancie jedného typu (inštancie z jednej hierarchie tried). Tento rozdiel sa dá obísť, pokiaľ zvolíme dostatočne všeobecný návratový typ (napr. `Object` v Jave) a v klientskom kóde sa realizuje pretypovanie vytvorených objektov v závislosti od parametra, ktorý sme poslali výrobnej metóde.

Príklad

Použitie vzoru Výrobná metóda môžeme nájsť na mnohých miestach. Jedným z príkladov použitia v Java API je realizácie metódy `getContent` triedy `URLConnection`, ktorá v závislosti od obsahu poslaného cez sieťové spojenie vracia rôzny obsah. Môže ísť o text, obrázok, XML dokument a pod.

Príkladom prepojenia paralelných hierarchií tried je MVC vzor aplikovaný v prostredí editora GUI (konkrétne ide o GEF – *Graphical Editing Framework* pre IDE prostredie Eclipse). V ňom máme hierarchiu *controller* objektov (v zásade pre každý grafický

prvok jedna trieda), ktoré implementujú spoločné rozhranie `EditPart`. Každý grafický prvok má nejakú grafickú reprezentáciu, ktorá predstavuje *view*. *View* objekty sú tiež usporiadané do podobnej hierarchie a dedia zo spoločnej nadtriedy `Figure`. Rozhranie `EditPart` definuje metódu `getFigure`, ktorá reprezentuje výrobnú metódu. Implementácie tohto rozhrania vracajú inštanciu z paralelnej hierarchie `Figure`. Náčrt kódu je znázornený nižšie.

```
public abstract class Figure {
  public abstract void render(GraphicContext gc);
  ...
}

public class GroupBoxFigure extends Figure {
  public void render(GraphicContext gc) {
    ...
    gc.drawRectangle(x, y, width, height, border);
    ...
}
  ...
}

public class TextFieldFigure extends Figure {
  public void render(GraphicContext gc) { ... }
  ...
}

public interface EditPart {
  Figure getFigure();
  Object getModel();
  EditPart getParent();
  Collection getChildren();
  ...
}

public class GroupBoxPart implements EditPart {
  private GroupBoxFigure figure;
  public Figure getFigure() {
    if (figure == null) {
      figure = new GroupBoxFigure(getModel());
      ...     }
    return figure;
  }
  ...
}

public class TextFieldPart implements EditPart {
  private TextFieldFigure figure;
  public Figure getFigure() {
    if (figure == null) {
      figure = new TextFieldFigure(getModel());
      ...     }
    return figure;
  }
  ...
}
```

Príbuzné vzory

Prototyp – alternatívny spôsob vytvárania objektov ku vzoru Výrobná metóda.

Abstraktná továreň – Výrobná metóda slúži zväčša na tvorbu individuálnych inštancií produktov. Pokiaľ chceme vytvárať množiny súvisiacich objektov, vzor Abstraktná továreň je vhodnejší[1].

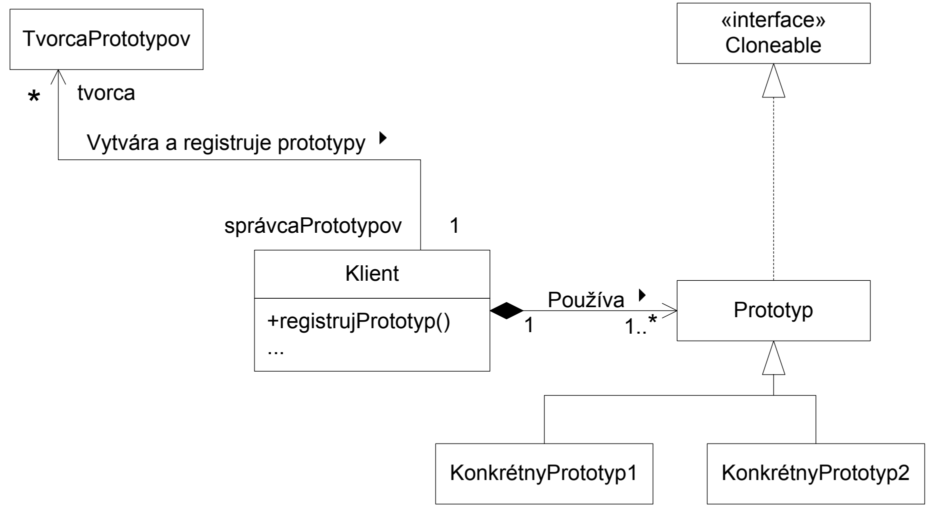
1.4 Prototyp

Účel

Vytvárané objekty sú definované dodanými prototypmi týchto objektov. Prototypy predstavujú tzv. druh objektu, pod ktorým rozumieme typ objektu (teda triedu) plus hodnoty jeho atribútov a asociácií. Z dodaných prototypov sa vytvárajú nové inštancie klonovaním – tvorbou replík (kopírovaním samých seba).

Využitie vzoru prototyp sa odporúča najmä v prípadoch, keď objekty, ktoré majú byť vytvorené majú byť špecifikované až počas behu aplikácie – *runtime*.

Štruktúra



Obrázok 1-6. Štruktúra vzoru Prototyp.

Súčasti

`Prototyp` – základný typ objektov, s ktorými pracuje daná aplikácia. Zväčša je realizovaný ako abstraktná trieda alebo rozhranie, ktoré poskytuje metódu pre vytvorenie svojej repliky. Na obrázku 1-6 je schopnosť vytvorenia repliky definovaná implementovaním rozhrania `Clonable`.

---

[1] V časti implementácia bol spomenutý spôsob použitia Výrobnej metódy s parametrom, kedy dostávame podobné správanie ako pri použití vzoru Abstraktná továreň

`KonkrétnyPrototyp1, 2...` – konkrétny typ objektov, s ktorými pracuje aplikácia.

`Klient` – časť aplikácie, ktorá spravuje prototypy, vytvára nové objekty a využíva ich.

`TvorcaPrototypov` – samostatná trieda, ktorá realizuje vytvorenie a dodanie (registrovanie) prototypov.

Dôsledky

Medzi základné výhody vzoru Prototyp patrí možnosť dynamického pridávania nových inštancií a možnosť prispôsobenia existujúcich inštancií počas behu aplikácie. Vytvárané objekty sú často realizované pomocou vzoru Zloženina (pozri kapitolu 2.3).

Skrýva pred klientom, skutočné triedy inštancií, ktoré sa budú vytvárať a používať.

Počas behu aplikácie je možné ľahko pridávať a odoberať nové produkty zaregistrovaním a odregistrovaním nových inštancií produktov:

- Možnosť meniť vytvárané inštancie menením hodnôt atribútov.
- Možnosť meniť vytvárané inštancie menením jej štruktúry. S využitím vzoru Zloženina je možné vytvárať zložité objekty, ktoré je možné znovupoužiť. Typickým príkladom použitia vzoru Prototyp je realizácia palety prvkov pre kresliaci program – napr. program na kreslenie logických obvodov. V takomto programe by sme mohli vytvoriť nový prvok palety pospájaním existujúcich prvkov do štandardnej súčiastky a pridaním tejto zloženiny do palety.

Redukuje sa nutnosť podtrieďovania triedy `Výrobca` pri použití vzoru Výrobná metóda, kde sa často vytvára hierarchia tried `Výrobca`, ktorá korešponduje s produktmi, ktoré sa vytvárajú.

Implementácia

Pokiaľ chceme, aby sa dali pridávať nové a odstraňovať pôvodné objekty, je vhodné použiť manažér prototypov – tzv. *registry* objekt. Manažér prototypov spravuje zoznam objektov, ktoré sú k dispozícii pre vytváranie a sprístupňuje ich pod nejakým kľúčom. Najčastejšie sa realizuje ako statický (globálny) alebo unikátny objekt typu `HashMap`.

Manažér prototypov môže byť realizovaný samostatným objektom alebo môže byť implementovaný ako časť triedy, ktorá využíva vytvárané produkty (druhá z uvedených možností je znázornená na obrázku 1-6).

Problémom vo vzore Prototyp môže byť klonovanie prototypov a realizovanie hlbokých kópií. Problémy sa vyskytujú najmä pri komplikovaných objektoch s cyklickými referenciami.

Príklad

Príkladom použitia sú palety kresliacich programov ako napr. CAD alebo WYSIWYG (*What You See Is What You Get*) nástroje na tvorbu GUI.

Príbuzné vzory

Zloženina – vytvárané produkty sú často realizované ako zloženiny.

Dekoratér – vytvárané produkty často využívajú vzor Dekoratér.

Výrobná metóda, Abstraktná továreň – alternatívy k vzoru Prototyp, ktoré je možné použiť, pokiaľ nepotrebujeme meniť zoznam vytváraných objektov počas behu aplikácie.

1.5 Unikát

V anglickej literatúre sa na návrhový vzor odvoláva ako na *Singleton*.

Účel

Zabezpečuje existenciu jednej inštancie nejakej triedy. Ide zväčša o inštancie, ktoré poskytujú centrálny prístup k zdrojom alebo službám a sú prístupné z dobre známeho miesta. Ide napr. o služby správy prístupových práv (*permission manager*), *JNDI lookup service*, *database connection pool manager*, *file system manager*.

Podobné správanie ako má unikátna inštancia je možné docieliť implementáciou žiadanej funkčnosti skupinou statických metód. Tento spôsob má zásadný nedostatok. Funkčnosť implementovanú pomocou statických metód nie je možné jednoduchým spôsobom zmeniť a nie je možné mať jej viaceré implementácie (názov triedy, ktorá realizuje statické metódy je zadrôtovaný v zdrojovom kóde aplikácie).

Pojem jedinej inštancie – unikát, v menej striktnom chápaní, často znamená aj to, že aplikácia využíva iba jednu inštanciu triedy. Existenciu jednej inštancie však nemusí zabezpečovať samotná trieda, t.j. trieda poskytuje aj verejné konštruktory. Čiže ide o unikát iba z pohľadu aplikácie. Tento typ unikátov nezodpovedá návrhovému vzoru Unikát.

Použitie aplikačných unikátov môžeme nájsť v servisne orientovaných informačných systémoch s viacvrstvovou architektúrou (SOA).

Štruktúra

Obrázok 1-7. Štruktúra vzoru Unikát.

Súčasti

`Unikát` – realizuje požadovanú funkcionalitu prostredníctvom svojich inštančných metód, zabezpečuje možnosť vytvorenia a prístupu k jedinej svojej inštancii pomocou privátneho konštruktora a verejnej statickej metódy na získanie tejto inštancie.

Dôsledky

Reštrikcia na vytvorenie jedinej inštancie, v réžii samotnej triedy (prípadne nadtriedy).

Ľahko je možné zmeniť unikátnu implementáciu na viac objektovú triedu (postačuje zmeniť metódu `dajInštanciu`).

Na rozdiel od statických metód je možné implementáciu unikátnej triedy ľahko nahradiť realizáciou inej triedy, ktorá má rovnaké rozhranie ako požadujeme od unikátneho objektu. Táto vlastnosť nám umožňuje jednoduchých spôsobom počas testovania častí aplikácie, ktoré využívajú náš unikát, nahradiť jeho funkcionalitu testovacou (*mock*, *stub*) implementáciou.

Je možné používať viacero typov tried (podtried). To, ktorá z implementácií sa použije, môže byť špecifikované v konfigurácii aplikácie (.properties, .xml).

Implementácia

Jednu inštanciu zabezpečuje zväčša samotná trieda tak, že neposkytuje verejný konštruktor. Na získanie inštancie slúži verejná statická metóda (metóda triedy), ktorá zabezpečuje inicializáciu jedinej inštancie. Vo väčšine prípadov sa využíva inicializácia na požiadanie (*lazy initialization*).

Príklad

Jedným z možných príkladov použitia unikátnych objektov je realizácia služby prístupových práv. Najjednoduchší spôsob implementácie je znázornený v nasledujúcom zdrojovom kóde.

```
public class PermissionManager {
  private static PermissionManager singleton;
  public static PermissionManager getInstance() {
    if (singleton == null) {
      singleton = new PermissionManager();
      // initialize
      ...
    }
    return singleton;
  }

  public boolean isPermited(String resourceID, ACLCode acl) {
    ...
  }

  // dalsie metody pre realizaziu pristupovych prav
  ...
}
```

Inou možnosťou je realizácia služieb pomocou unikátnych objektov bez reštrikcie na konštruktor. Typickým príkladom sú aplikácie využívajúce Spring framework. Spring je definovaný ako IoF (*Inversion of Controll*) a AOP (*Aspect Oriented Programming*) kontajner. Jednou z funkcií *Spring framework* je riadenie životného cyklu unikátnych objektov, čo zahŕňa tvorbu a inicializovanie unikátnych inštancií a ich poskytovanie ostatným častiam aplikácie na základe definovaného kľúča. Typ unikátnych objektov

definuje rozhranie (*interface*). Skutočný objekt, ktorý aplikácia použije môže byť ľubovoľný objekt, ktorý implementuje požadované rozhranie. Definícia a inicializácia je zapísaná v XML súbore, ktorý opisuje ako sa jednotlivé objekty vytvárajú, čím sa napĺňajú ich atribúty a pod.

Príbuzné vzory

Vzor Unikát sa často používa v spojení s ďalšími vzormi ako napr. Staviteľ, Abstraktná továreň či Prototyp.

2 ŠTRUKTURÁLNE VZORY

Časté a opakujúce sa problémy pri vývoji a implementácií softvéru vznikajú pri práci so zložitými (rekurzívnymi i nerekurzívnymi) štruktúrami. Existujú viaceré prístupy k vytváraniu a pristupovaniu k prvkom týchto štruktúr. Štruktúry ako také na jednej strane chceme čo možno najviac zjednodušiť pre klientov, na druhej strane málokedy nám to podklady z analýzy priamo dovoľujú.

Práve na riešenie vyššie opísaných problémov sú určené štrukturálne vzory. Umožňujú kontrolovaný prístup k definovaným (rekurzívnym, stromovým) i nedefinovaným často na prvý pohľad príliš zložitým štruktúram.

Štrukturálne vzory sa nezaoberajú iba samotnými štruktúrami, ktorým je venovaná väčšia pozornosť, ale i jednotlivými elementmi, ktoré sú často súčasťami zložitejších štruktúr a sú ďalej nedeliteľné.

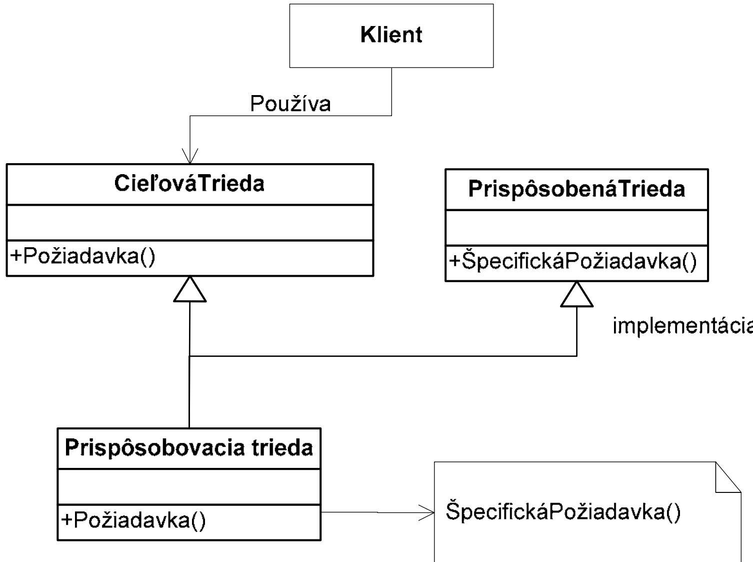
2.1 Adaptér

V anglickej literatúre sa na návrhový vzor odvoláva ako na *Adapter* alebo *Wrapper*.

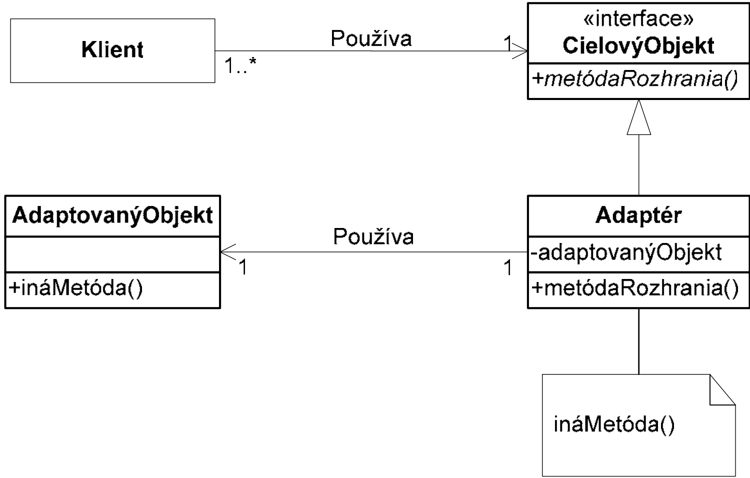
Účel

- Prispôsobenie jedného rozhrania na iné rozhranie.
- Vytvorenie jednotného rozhrania medzi rôznymi triedami slúžiacimi na podobný účel.
- Sprístupnenie funkcionalít oboch tried v rámci jedného rozhrania.
- V prípade objektového adaptéra prispôsobenie rozhrania podtried abstraktnej triedy.

Štruktúra



Obrázok 2-1. Štruktúra vzoru triedneho adaptéra.



Obrázok 2-2. Štruktúra vzoru objektového adaptéra.

Súčasti

`Klient` – využíva cieľovú triedu, resp. cieľový objekt. Nakoľko adaptér je podtrieda cieľovej podtriedy, klient vie používať inštanciu adaptéra.

`AdaptovanýObjekt`, `PrispôsobenáTrieda` – objekt/trieda, ktorého rozhranie prispôsobujeme k rozhraniu cieľovému objektu/triedy.

`CieľovýObjekt`, `CieľováTrieda` – objekt/trieda, ku ktorého rozhraniu prispôsobujeme rozhranie adaptovaného objektu/triedy.

Dôsledky

Adaptér je možné realizovať dvoma základnými prístupmi. Triedu môžeme prispôsobiť deklaráciou podtriedy (triedny adaptér) alebo delegáciou jej funkcionalít (objektový adaptér).

Triedny adaptér je menej flexibilný kvôli nemožnosti adaptovania podtried prispôsobenej triedy. Pre objektový adaptér sa podtriedy prispôsobujú delegovaním operácií do inštancie podtriedy prispôsobenej triedy.

Implementácia

V prípade jazykov s viacnásobným dedením môžeme vytvoriť adaptér dedením metód od oboch adaptovaných tried (obrázok 2-1). V tomto prípade metódy ponúkaného rozhrania zavolajú metódy zdedené od prispôsobovanej triedy. Nevýhoda tohto prístupu je nemožnosť adaptovania podtried prispôsobovanej triedy.

Na rozdiel od triedneho adaptéra objektový adaptér implementuje iba ponúkané rozhranie, metódy prispôsobovanej triedy sú dosiahnuté delegovaním (obrázok 2-2). Adaptovaný objekt sa zapamätá vo vnútornej premennej adaptéra pri jej vytváraní. Výhodou objektového adaptéra je možnosť adaptácie podtried prispôsobovanej triedy. Objektový adaptér je možné implementovať aj v jazykoch, ktoré neumožňujú viacnásobné dedenie.

Z pohľadu testovania objektový adaptér umožňuje implementáciu falošného objektu (*mock object*) s rozhraním prispôsobovanej triedy, takýmto spôsobom je možné otestovať správnosť interakcií medzi adaptérom a prispôsobovanou triedou.

Príklad

Mnoho nových verzií projektov predstaví nové API pre niektoré funkcionality softvérového produktu. Znamená to reimplementáciu časti kódu s iným rozhraním. Nakoľko nie je správne udržiavať duplikáty kódu a funkcionalít, stará implementácia kódu by sa mala odstrániť a vykonávať pomocou využitia nového API.

Príbuzné vzory

Adaptér definuje obalenie objektu alebo triedy, aby tak ponúkol iné rozhranie. Podobné obalenie sa nachádza aj v návrhovom vzore Zástupca, ktorý zakrýva niektoré vlastnosti obaleného objektu. Dekoratér namiesto zaistenia iného rozhrania, zmení (rozšíri alebo utlmí) správanie dekorovaného objektu.

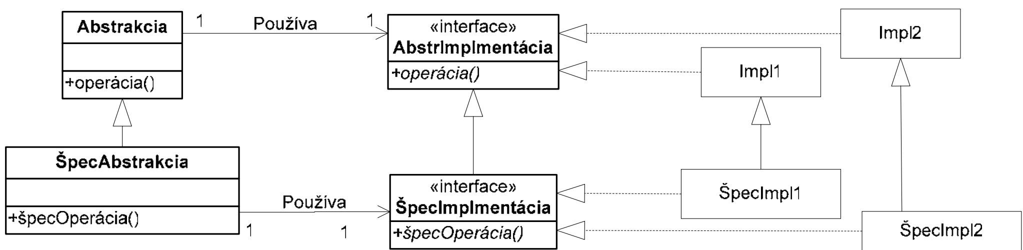
2.2 Premostenie

Návrhový vzor premostenie, po anglicky *Bridge*, slúži na rozdelenie abstraktnej a implementačnej hierarchie tried. Rozdelenie týchto hierarchií je odôvodnené tam, kde môže existovať viac možných implementácií pre tie isté funkcionality (napr. pre viaceré platformy).

Účel

- Rozdelenie abstraktnej a implementačnej hierarchie.
- Zabezpečenie ľahkej rozšíriteľnosti oboch hierarchií.
- Zakrytie implementácie pomocou abstraktnej hierarchie.

Štruktúra



Obrázok 2-3. Štruktúra vzoru Premostenie.

Súčasti

Abstraktná hierarchia – štruktúra abstraktných tried nezávislá od implementácie.

Implementačná hierarchia – definuje jednotné rozhranie na prácu s externými knižnicami. V implementačnej hierarchii sa vytvorí súbor adaptérov na prispôsobenie rozhrania knižníc na rozhranie implementačnej hierarchie.

Externé knižnice – Vonkajšie prostriedky a knižnice, ktoré chceme v kóde jednotne udržiavať. Na obrázku 2-3 sa externé knižnice používajú v triedach `Impl1`, `Impl2`,..., `ŠpecImpl1`, `ŠpecImpl2`,...

Dôsledky

Premostenie je spôsob udržiavania kódu. Dáva kódu rámec, ktorý je ľahko rozšíriteľný. Abstraktná hierarchia určuje rozhrania *front-end* objektov. Konkrétne implementácie rozhraní určujú spôsob na prístup k zariadeniam sprístupneným cez knižnice, *back-end* objekty. Každá knižnica má zvyčajne svoje vlastné, často veľmi odlišné rozhranie. Implementačná hierarchia takto tvorí súbor adaptérov, prispôsobuje *back-end* objekty k rozhraniu implementačnej hierarchie.

Implementácia

Triedy v abstraktnej hierarchii pristupujú k metódam zariadení sprístupnené cez rozhranie `Implementácia` nezávisle od toho, s akým konkrétnym zariadením a s akou konkrétnou implementáciou pracujú. Konkrétna implementácia rozhrania `Implementácia` je určená pri vytváraní triedy abstraktnej hierarchie.

```
package premostenie;
import junit.framework.*;
class Abstrakcia {
  private Implementácia implementácia;
  public Abstrakcia(Implementácia imp) {
    implementácia = imp;
  }

  // Abstrakcia používaná rôznymi front-end
  // objektami na implementáciu ich rozhrania
  public void služba1() {
    // Implementácia funkcionality pomocou
```

```
    // kombinácií back-end implementácií:
    implementácia.zariadenie1();
    implementácia.zariadenie2();
  }
  public void služba2() {
    implementácia.zariadenie2();
    implementácia.zariadenie3();
  }
  public void služba3() {
    implementácia.zariadenie1();
    implementácia.zariadenie2();
    implementácia.zariadenie4();
  }

  // Použité podtriedami Implementácie
  protected Implementácia vráťImplementáciu()
    { return implementácia; }
}

class KlientskáSlužba1 extends Abstrakcia {
  public KlientskáSlužba1(Implementácia imp)
    { super(imp); }
  public void službaA()
    { služba1(); služba2(); }
  public void službaB()
    { služba3(); }
}

class KlientskáSlužba2 extends Abstrakcia {
  public KlientskáSlužba2(Implementácia imp)
    { super(imp); }
  public void službaC() {
    služba2(); služba3(); }
  …implmentácie službaD, službaE …
}

interface Implementácia {
  // Spoločná implementácia daná v back-end
  // objektoch, každá vo vlastnej forme

  void zariadenie1(); void zariadenie2();
  void zariadenie3(); void zariadenie4();
}

class Knižnica1 {
  public void metóda1()
    { System.out.println("Kniz1.metóda1()"); }
  public void metóda2()
    { System.out.println("Kniz1.metóda2()"); }
}

class Knižnica2 {
  public void operácia1()
    { System.out.println("Kniž2.operácia1()");}
  …implmentácie operácia2, operácia3…
}

class Implementácia1 implements Implementácia {
```

```
  // Každé zariadenie je delegovné do príslušnej knižnice
  private Knižnica1 deleguj = new Knižnica1();
  public void zariadenie1() {
    System.out.println("Impl 1 - zariadenie1");
    deleguj.metóda1();
  }
  …implmentácie zariadenie2,zariadenie3, zariadenie4…
}

class Implementácia2 implements Implementácia {
  private Knižnica2 deleguj = new Knižnica2();
  …implmentácie zariadenie1,zariadenie2,
    zariadenie3, zariadenie4…
}
```

Príklad

Premostenie sa používa často v implementáciách GUI API, napríklad v Jave knižnice AWT a Swing využívajú premostenie na zabezpečenie platformovej nezávislosti. Prístup a manipulácia s dátami môžu byť vyriešené premostením, pristupovanie k rozhraniu abstraktnej hierarchie je nezávisle od dátového média, nad ktorým pracujeme.

Príbuzné vzory

Implementačná hierarchia tvorí súbor adaptérov, ktoré prispôsobujú rozhrania knižníc k rozhraniu implementačnej hierarchie.

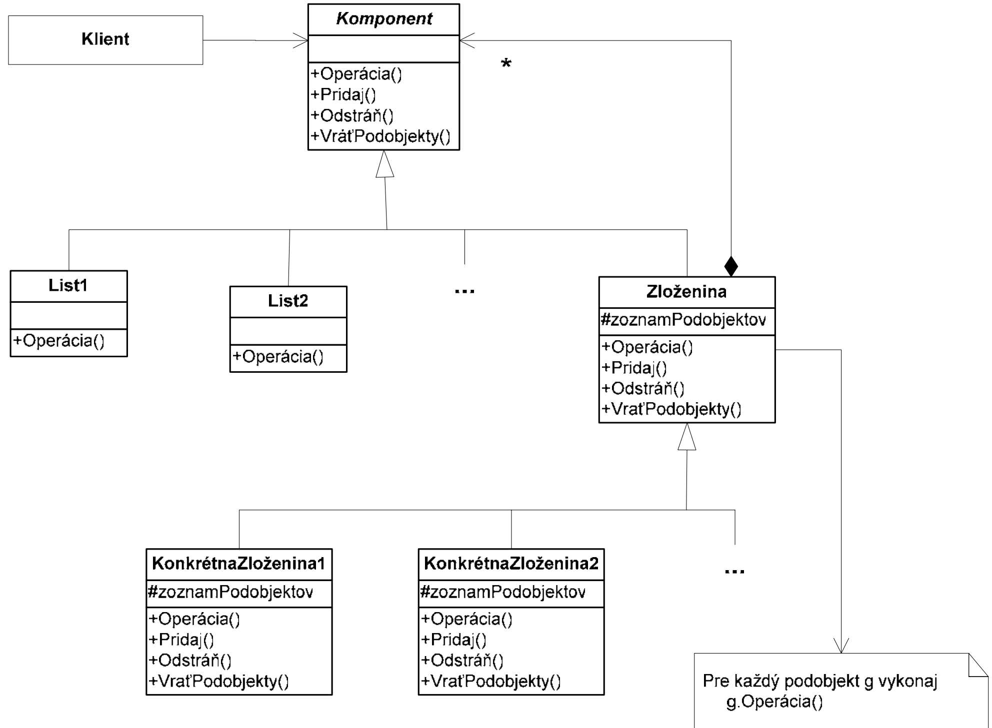
2.3 Zloženina

Na návrhový vzor zloženina sa v anglickej literatúre odvoláva ako na *Composite*. Slúži na tvorbu vnorených flexibilných štruktúr.

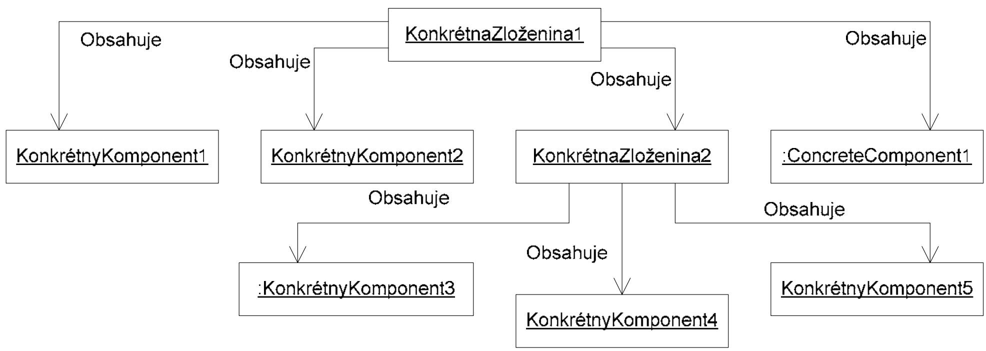
Účel

- Komponuje a súčasne zakrýva objekty obsiahnuté v komponujúcom objekte.
- Zabezpečuje kompatibilné rozhranie komponujúceho objektu a obsiahnutých objektov

Štruktúra



Obrázok 2-4. Triedna štruktúra Zloženiny.



Obrázok 2-5. Objektová štruktúra zloženín a komponentov.

Súčasti

`Komponent` – definuje spoločné rozhranie pre zloženiny a listové triedy, je to abstraktná trieda alebo iba rozhranie.

`List` – trieda, ktorej inštancie neobsahujú iné triedy.

`Zloženina` – trieda, ktorej inštancie môžu obsahovať inštancie podtried komponentov.

`Klient` – objekt pracujúci s komponentmi. Rozdiel medzi listovými triedami a zloženinami sa vníma často iba pri ich konštrukcii.

Dôsledky

Zloženina je objekt ktorý môže obsahovať objekty daného rozhrania a súčasne má rozhranie totožné s rozhraním obsiahnutých objektov. Umožňuje to vykonávať tie isté operácie so zloženinou ako listovými objektmi. Zvyčajne sú operácie delegované smerom k obsiahnutým objektom.

Zloženina sa ľahko implementuje, zjednoduší a sprehľadní kód.

Implementácia

Je viac možných spôsobov ako implementovať zloženinu. Je však možné vytvoriť zoznam vlastnosti, ktoré zloženina má spĺňať

- komponujúci objekt musí poskytovať také rozhranie ako majú komponenty,
- implementácia operácií komponentu v komponujúcom objekte deleguje danú operáciu na obsiahnuté komponenty,
- komponujúci objekt poskytuje operácie na pridanie objektov, podľa potreby aj operáciu na vymazanie objektu zo zoznamu alebo vrátenie zoznamu obsiahnutých objektov.

Príklad implementácie uvedený nižšie je jednoduchý spôsob ako implementovať zloženinu. Komponujúca trieda je podtriedou `ArrayList` a implementuje operácie rozhrania `Komponent`.

V prípade, že zloženinu implementujeme dedením, t.j. komponujúci objekt je podtriedou komponentu, potom stojí za úvahu, či operácie na manipuláciu s podobjektami neuviesť v triede komponenty ako prázdne (v prípade zavolania nevykoná nič).

Ak väčšina podtried komponentu pracuje potrebuje možnosť obsahovať iné komponenty, potom má zmysel implementovať zmysluplnú manipuláciu s obsiahnutými objektmi na úrovni komponentu. V tomto prípade listové komponenty túto možnosť utlmia.

```
package zloženina;
import java.util.*;
import junit.framework.*;

interface Komponent {
  void operácia();
}
class List implements Komponent {
  private String meno;
  public List(String meno) {
    this.meno = meno;
  }
```

```
  public String toString() {
    return meno;
  }
  public void operácia() {
    System.out.println(this);
  }
}

class Zloženina extends ArrayList implements Komponent {
  private String meno;
  public Zloženina(String meno) {
    this.meno = meno;
  }
  public String toString() {
    return meno;
  }
  public void operácia() {
    System.out.println(this);
    for(Iterator it = iterator(); it.hasNext();)
      ((Komponent)it.next()).operacia();
  }
}
```

Príklad

Zloženina sa používa, keď potrebujeme pracovať s hierarchickými štruktúrami. Zvyčajne implementácie GUI obsahujú zloženinu. Objekty vykreslené na plochu ako okná a dialógy obsahujú objekty ako tlačidlá, textové polia a iné komponenty grafického rozhrania. Súčasne však aj okno a dialóg sú obsiahnuté plochou, ktorá tak isto implementuje operácie rozhrania pre GUI objekty. Pri vykresľovaní okno deleguje operáciu vykresľovania objektom, ktoré obsahuje.

Príbuzné vzory

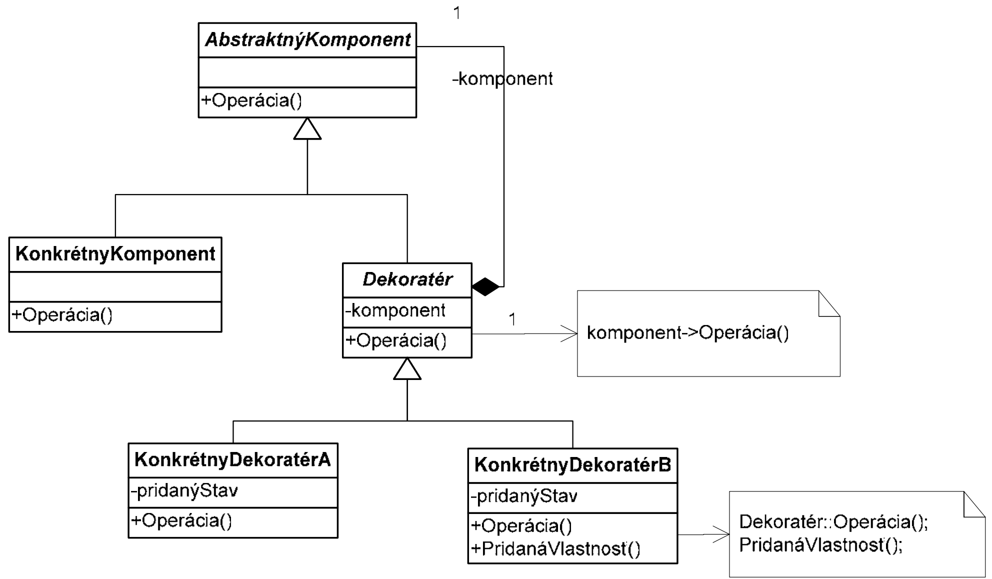
Zloženina, podobne ako návrhový vzor Fasáda, zakrýva viac objektov a umožní tak prácu s nimi cez jednoduchšie rozhranie. Objekty zakryté fasádou však nezdieľajú nutne to isté rozhranie ako je to v prípade zloženiny.

2.4 Dekoratér

Účel

- Zmení (rozšíri, obmedzí) správanie dekorovaného objektu.
- Dekoratér má to isté rozhranie ako dekorovaný objekt.
- Zmenší počet tried, pridáva flexibilitu pri určení vyžadovaného správania objektu.
- Zmena správania objektov sa vykonáva dynamicky počas behu; nie ako v prípade definovania podtriedy s rozšírením správaním.

Štruktúra



Obrázok 2-6. Návrhový vzor Dekoratér.

Súčasti

`AbstraktnýKomponent` – definuje spoločné rozhranie pre implementované konkrétne podtriedy a dekoratéry.

`KonkrétnyKomponent` – objekt vykonávajúci operácie.

`Dekoratér` – objekt meniaci správanie komponentov.

Dôsledky

Zmena správania objektu jej obalením iným objektom, dekoratérom môže významne zjednodušiť hierarchiu tried za cenu malého skomplikovania kódu. Rozšírenie alebo utlmenie nejakého špecifického správania viacerých objektov sa rieši namiesto vytvorenia viacerých podtried pomocou implementácie jednej triedy `Dekoratér`, ktorá zabezpečí zmenu správania všetkých objektov s požadovaným rozhraním.

Implementácia

Ako ilustráciu užitočnosti Dekoratéra si môžeme predstaviť implementáciu jedálneho lístka. Hosť si môže vybrať jedlo, ktoré môže rozšíriť oblohou, nápojom, atď. Ak by sme sa pokúsili implementovať všetky možné kombinácie jedál s možnými oblohami so zvlášť triedami, tak pridávaním nových prvkov do menu by sme dostali exponenciálne rastúci počet tried. Dekoratérmi však implementujeme zvlášť triedu len pre jednotlivé prvky z jedálneho lístka, čím pridaním nového prvku sa nám zvýši počet tried len o jeden. Kombinácie z jedálneho lístka vyriešime postupným obaľovaním pomocou dekoratérov.

Dekoratéry môžu vytvárať kaskádu, a tak viac zjemniť obalený objekt. Aktívne používanie príliš dlhých kaskád dekoratérov však môže spôsobiť neprehľadný a neefektívny kód.

Príklad

V GUI knižniciach AWT a Swing sa pomocou dekoratérov rieši obalenie okien rámikom. Rozšírenie objektu zapisujúce/čítajúce zo súboru alebo z iného zdroja v prechodnej pamäti sa často rieši pomocou dekoratérov (napr. `Stream`, resp. `Reader` a `Writer` v Jave).

Príbuzné vzory

Podobne ako Dekoratér aj objektový Adaptér obaľuje objekt, rozdiel je však v tom, že Dekoratér ponúka také isté rozhranie ako obalený objekt, kým Adaptér prispôsobuje rozhranie obaleného objektu.

2.5 Fasáda

Účel

Poskytuje jednotné rozhranie k súboru viacerých rozhraní nejakého podsystému. Fasáda poskytuje vysoko-úrovňové rozhranie, pomocou ktorého je možné jednoduchšie používať nejaký zložitý podsystém.

Vďaka tomu, že rozdelíme veľký systém na viac menších podsystémov, sme schopní redukovať komplexnosť celku. Každý z jednotlivých podsystémov by mal realizovať nejakú jednoduchú funkciu. Zložitejšie celky sa potom vytvárajú hierarchickým spájaním jednoduchších častí. Tieto jednotlivé súčiastky sa snažíme spájať tak, aby boli pokiaľ možno čo najviac flexibilné. To znamená, že žiadna súčiastka by nemala narábať s vnútornosťami inej súčiastky priamo, a tým byť zbytočne závislá od jej implementačných detailov. Jednou z možností, ako toto dosiahnuť, je použitie objektu `Fasáda`, ktorý poskytuje jednoduché rozhranie k zložitej funkcionalite, ktorú poskytuje nejaký podsystém. Vďaka tomu, že pristupujeme k službám tohto podsystému cez fasádu, nemusíme sa starať o jeho implementačné podrobnosti.

Použitie

Fasádu je výhodné použiť v prípadoch, keď

- chceme poskytnúť jednoduché rozhranie k pomerne zložitému systému,
- klienti nejakého podsystému príliš závisia na jeho implementácii,
- potrebujeme rozvrstviť podsystémy, použijeme fasády na definovanie vstupných bodov do jednotlivých úrovní. Keď jedna vrstva používa inú vrstvu, vďaka komunikácii cez fasádu bude jasné, ako jednotlivé vrstvy spolu komunikujú.

Súčasti

`Fasáda`

- oddeľuje klientov od implementačných podrobností zložitého podsystému,

- vie o tom, ktorému podsystému je potrebné delegovať jednotlivé druhy klientov.

Triedy podieľajúce sa na implementácii podsystémov

- implementujú funkcionalitu jednotlivých podsystémov,
- zvládajú prácu, ktorú od nich žiada fasáda,
- nevedia nič o tom, kto presne ich používa (fasáda), nedržia referencie priamo na svojich klientov.

Spolupráca

Klienti komunikujú s podsystémom tak, že posielajú správy fasáde, ktorá tieto správy preposiela príslušnému podsystému. Skutočnú prácu vykonávajú samotné podsystémy. Fasáda robí len preposielanie správ. Prípadne ich transformáciu pred preposlaním.

Klienti, ktorí používajú fasádu, nemajú prístup k samotným podsystémom, ktoré ona zapúzdruje.

Dôsledky

Vzor fasáda má nasledujúce výhody:

- Umožňuje klientom, aby sa nemuseli zaťažovať zložitými implementačnými detailmi jednotlivých podsystémov. Tým pádom fasáda redukuje nevyhnutný počet objektov, s ktorými musia klienti komunikovať, vďaka čomu je používanie týchto podsystémov jednoduchšie.
- Podporuje tvorbu voľných (namiesto tesných) spojení medzi klientom a subsystémom. Vďaka tomu, že toto spojenie je voľné, môžeme jednoduchšie zamieňať (vylepšovať) subsystémy bez toho, aby sme museli zakaždým prispôsobovať ich klientov. Fasáda eliminuje zložité a cyklické závislosti medzi objektmi. Vďaka tomu je možné klienta a subsystém implementovať oddelene.
- Nezabraňuje klientom priamo používať triedy subsystému, keď je to naozaj nevyhnutné. To, či fasádu použijete alebo nie, je na vás. Môžte si zvoliť pohodlie (prístup cez fasádu) alebo všeobecnosť (priamy prístup k objektom, ktoré reprezentujú príslušný subsystém).

Príklad

Fasáda predstavuje užitočnú techniku, ktorá sa v praxi aj často využíva. Fasáda je jedným z mechanizmov abstrakcie. Teda, umožňuje nám zmierňovať nepríjemné problémy vznikajúce pri práci s komplikovanými systémami. Napríklad rôzne vygenerované lexikálne analyzátory sa používajú jednoducho tak, že

```
myParser parse: 'crude text'
```

a nemusíme vôbec vedieť o tom, ako presne tento objekt (syntaktický analyzátor) vznikol a ako presne operuje so surovým textom. Syntaktický analyzátor mohol navyše poskytnúť niekto iný a vďaka jeho jednoduchému rozhraniu ho vieme bez problémov používať.

Iným príkladom je trieda `Speaker` v jazyku Squeak.

Ukážka použitia:

```
Speaker man say: 'Say something!'.
```

V tomto prípade objekt `Speaker` je fasádou k zložitej funkcionalite realizujúcej syntézu ľudskej reči (angličtiny) zo zadaného textového.

Príbuzné vzory

Abstraktná továreň môže byť použitá na to, aby sme pomocou nej vytvárali rôzne implementácie subsystému s viac-menej tou istou funkcionalitou nezávisle od toho, o aký konkrétny subsystém sa jedná. Abstraktná továreň môže byť použitá ako alternatíva k fasáde za tým účelom, aby sme skryli platformovo špecifické triedy.

Sprostredkovateľ je podobný fasáde v tom, že tvorí abstrakciu k už existujúcim triedam. Cieľom sprostredkovateľa ale je vytvoriť abstrakciu pre komunikáciu medzi kolegami tvoriacimi v určitom zmysle jeden podsystém. Kolegovia vedia o sprostredkovateľovi, komunikujú s ním namiesto toho, aby komunikovali priamo s ďalšími kolegami. Fasáda, na druhej strane, len tvorí abstrakciu rozhrania nejakého subsystému, a tým ho robí jednoduchšie použiteľným. Nedefinuje novú funkcionalitu a samotné triedy subsystému o fasáde nevedia nič, ani ju priamo nepoužívajú.

Jeden subsystém má obyčajne jedinú fasádu (toho istého typu). Kvôli tomu je fasáda často realizovaná návrhovým vzorom Unikát.

2.6 Mušia váha

Účel

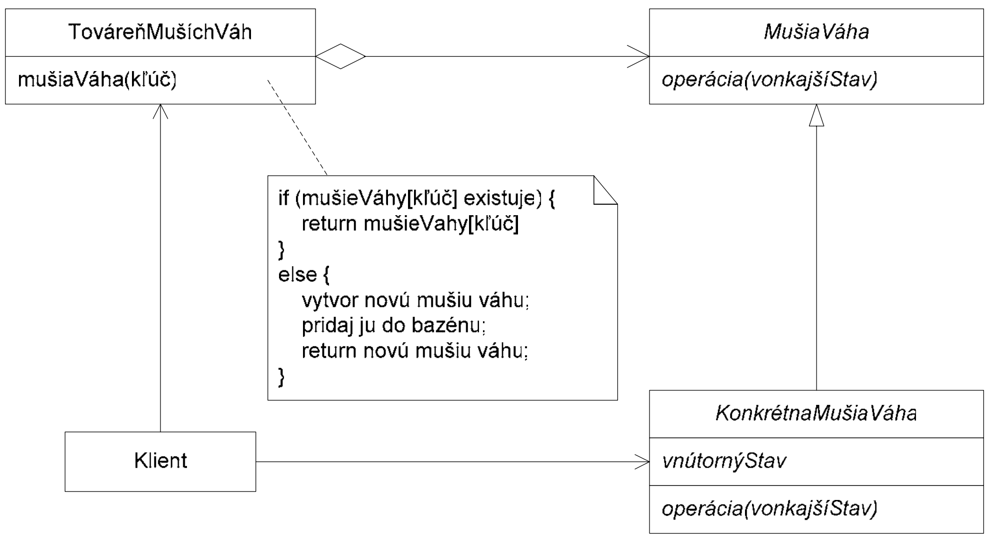
Používa zdieľanie na efektívnu reprezentáciu veľkého množstva jemnozrnných objektov.

Pri vývoji niektorých aplikácii môže byť výhodné, keď sa používajú objekty na reprezentáciu všetkých možných aspektov príslušnej aplikácie. Naivný prístup však môže niekedy spôsobiť to, že príslušný model sveta bude zbytočne pamäťovo náročný.

Zoberme si napríklad nejaký bežný objektovo-orientovaný editor dokumentov. Tieto väčšinou pomocou objektov reprezentujú rôzne zabudované elementy, ako tabuľky a obrázky. Avšak obyčajne už nepoužívajú objekty na reprezentovanie samotných znakov dokumente, aj keby to mohlo byť flexibilnejšie a čistejšie aj na takejto mikroúrovni. Keby sme aj znaky reprezentovali pomocou objektov, tak by sme s nimi mohli narábať podobne ako aj s inými vizualizovateľnými objektmi z pohľadu kreslenia a formátovania dokumentu. Zmenou toho, ako sa jednotlivé znaky vykresľujú, by sme mohli podporovať rôzne znakové sady a nemuseli by sme pritom zvlášť zasahovať do iných častí aplikácie.

Nevýhodou hore uvedeného prístupu je jeho pamäťová náročnosť. Aj stredne veľké dokumenty obsahujú stovky tisíc znakov. Tieto by zabrali veľa pamäte a mohli by spôsobiť, že aplikácia by nebola prakticky použiteľná pre bežné dokumenty, lebo by sa nezmestili do fyzickej pamäte. Návrhový vzor Mušia váha ukazuje, ako je možné ostať pri reprezentácii jemnozrnných objektov bez toho, aby sme za to museli draho zaplatiť pamäťou.

Štruktúra



Súčasti

`MušiaVáha` (Glyf) – deklaruje rozhranie, pomocou ktorého mušie váhy operujú nad vonkajším stavom.

`KonkrétnaMušiaVáha` (Znak) – implementuje rozhranie `MušiaVáha` a pridáva ukladanie vnútorného stavu, ak je niečo také v tomto prípade potrebné. Inštancie triedy `KonkrétnaMušiaVáha` musia byť zdieľateľné, t.j. celý stav uložený v tomto objekte musí byť vnútorný, t.j. nesmie byť závislý na kontexte, v ktorom sa príslušná konkrétna mušia váha používa.

`TováreňMušíchVáh`

- vytvára a spravuje objekty muších váh,
- zabezpečuje, aby boli mušie váhy správnym spôsobom zdieľané. Keď klient požaduje mušiu váhu, Továreň muších váh poskytne príslušnú existujúcu mušiu váhu. Ak ešte neexistuje, tak vytvorí novú mušiu váhu s príslušným vnútorným stavom, zaregistruje ju v bazéne všetkých muších váh a poskytne ju klientovi na používanie.

`Klient`

- drží referencie na tie mušie váhy, ktoré potrebuje,
- dopočítava alebo pamätá si vonkajší stav jednotlivých muších váh, ktoré drží.

Použitie

Použitie tohto vzoru znižuje prehľadnosť zdrojového kódu za cenu zvýšenia efektivity jeho vykonania. Situácie, v ktorých je možné použiť tento vzor, sú zriedkavé. Nie je jasné, či sa použil tento vzor aj pre iné prípady než len optimalizovanie reprezentácie znakov ako objektov. V jazyku Smalltalk sú triedy `CharacterSet` a `Character`,

ktoré predstavujú použitie tohto návrhového vzoru. Tento vzor by bolo možné použiť aj pre iné podobné príklady, kde je konečný počet mušších váh (tak ako je to v prípade znakov), ktoré môžu byť odkazované z nejakej továrne mušších váh (v hore uvedenom príklade ním je trieda `CharacterSet`).

Pri rozhodovaní o tom, či použiť tento návrhový vzor alebo nie (a ostať pri naivnom prístupe) treba zvážiť kardinalitu množiny muších váh a réžiu spojenú s ich manažovaním. Je otázne, nakoľko by bolo použitie tohto vzoru výhodné, keď je muších váh potenciálne nekonečne veľa – továreň muších váh by mohla rásť donekonečna a tým pádom stráca svoj pôvodný zmysel – optimalizácia pamäťových nárokov.

Tento vzor sa relatívne málo používa.

Príbuzné vzory

Návrhový vzor `Mušia váha` je často kombinovaný s návrhovým vzorom `Zloženina`. Za účelom implementovania štruktúry s logickou hierarchiou (ako orientovaný acyklický graf, ktorého listy predstavujú zdieľané objekty).

Často je výhodné implementovať návrhový vzor `Stav` a `Stratégiu` ako mušie váhy.

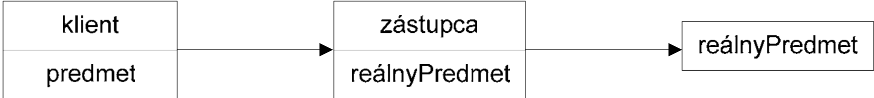
2.7 Zástupca

Účel

Poskytuje náhradný objekt, pomocou ktorého sa pristupuje k inému objektu.

Jedným z dôvodov na to, aby sme nahradili priame používanie originálneho objektu prístupom k nemu pomocou zástupcu, môže byť réžia spojená s jeho vytváraním. Zoberme si napríklad editor dokumentov, ktorý v sebe môže obsahovať obrázky. Ich uloženie do pamäte môže byť časovo aj pamäťovo náročné. Navyše, často nepotrebujeme, aby bol každý obrázok v nejakom dokumente uložený v pamäti. Len tie, ktoré má používateľ možnosť v danom okamihu vidieť na obrazovke. Jednotlivé náročné objekty sa načítajú do pamäte podľa potreby.

Štruktúra



Obrázok 2-7. Štruktúra vzoru Zástupca.

Súčasti

`Klient` – v tomto prípade komunikuje s reálnym predmetom len nepriamo, cez zástupcu.

`Zástupca`:

- udržuje odkaz umožňujúci klientovi pracovať s reálnym predmetom,

- poskytuje identické rozhranie k reálnemu predmetu ako má samotný reálny subjekt,
- riadi prístup alebo iným spôsobom upravuje rozhranie reálneho subjektu.
- Zástupcovia vzdialeného objektu sú zodpovední za zakódovanie požiadavky a jej argumentov a poslanie zakódovanej požiadavky reálnemu predmetu v inom adresovom priestore.
- `Skorozástupca` môže dočasne uchovávať niektoré údaje o reálnom predmete tak, aby k nemu nebolo potrebné pristupovať tak často.
- Ochranný zástupca kontroluje, či volajúci má prístupové práva nevyhnutné na vykonanie požiadavky.

`Reálny predmet` – definuje reálne údaje, ku ktorým sa pristupuje cez zástupcu.

Použitie

Zástupca je použiteľný kedykoľvek je potrebný pružnejší odkaz na iné objekty než obyčajný ukazovateľ.

1. Zástupca vzdialeného objektu poskytuje lokálneho predstaviteľa objektu, ktorý je umiestnený v inom adresovom priestore.
2. Virtuálny zástupca predstavuje odľahčený variant pôvodného náročného objektu. Náročný objekt sa skutočne načíta do pamäte až keď je to naozaj nevyhnutné.
3. Ochranný zástupca obmedzuje prístupové práva k už existujúcemu objektu. Pôvodný objekt mohol poskytovať viac služieb. Klienti pristupujúci k nemu cez ochranného zástupcu budú môcť používať len tie služby, ktoré im zástupca poskytne.
4. Bystrý odkaz je náhrada za obyčajný ukazovateľ, ktorý robí ešte ďalšie úkony, keď ho niekto použije. Jeho typické použitia sú:
   a. Počítanie odkazov na skutočný objekt za tým účelom, aby sme ho mohli automaticky uvoľniť, keď počet odkazov klesol na nulu.
   b. Načítanie trvalého objektu do hlavnej pamäte, keď sa k nemu prvýkrát pristupuje.
   c. Skontrolovanie toho, či je daný skutočný objekt zamknutý pred tým, než sa k nemu pristúpi.

Dôsledky

Návrhový vzor Zástupca spôsobuje, že sa medzi klienta a reálny subjekt vloží ďalšia vrstva. Takáto ďalšia vrstva sa dá využiť mnohými spôsobmi podľa toho, o aký druh zástupcu ide:

1. Vzdialený zástupca môže pred klientom skrývať fakt, že komunikuje s predmetom v inom adresovom priestore.
2. `Skorozástupca` môže vykonávať rôzne optimalizujúce činnosti ako napríklad vytvorenie objektu na požiadanie.
3. Oba druhy ochranných zástupcov ako aj bystrej referencie umožňujú robiť rôzne činnosti súvisiace s údržbou chodu systému.

Existuje aj iný druh optimalizácie, ktorú môže zástupca pred klientom skrývať. Tento mechanizmus sa volá skopíruj-pri-zápise (*copy-on-write*) a súvisí s požiadavkou na vytvorenie objektu. Skopírovanie veľkého a komplikovaného objektu môže byť časovo a priestorovo náročná operácia. Keď sa kópia nikdy nemodifikuje, tak nie je dôvod zaplatiť za kópiu takú cenu. Keď použijeme zástupcu, ktorý odloží kopírovací proces, tak máme zaistené, že cenu za skopírovanie objektu zaplatíme, len keď je to naozaj nevyhnutné.

Aby mohol mechanizmus skopíruj-pri-zápise fungovať, musíme počítať odkazy na subjekt. Skopírovanie zástupcu neurobí nič iné, len zvýši počet referencií na predmet a ako výsledok kopírovania sa vráti ten istý zástupca. Len v prípade, že sa klient pokúsi modifikovať predmet, tak zástupca vytvorí úplnú a samostatnú kópiu predmetu, začne ukazovať na túto novú kópiu a na tejto novej kópii vykoná požadovanú operáciu, ktorá predmet modifikuje. Vtedy tiež zníži počet referencií na pôvodný predmet. Mechanizmus skopíruj-pri-zápise môže značne redukovať cenu kopírovania ťažkých predmetov.

Príbuzné vzory

Adaptér – poskytuje odlišné rozhranie k objektu, ktorý prispôsobuje. Zástupca, na rozdiel od Adaptéra, poskytuje rovnaké rozhranie k pôvodnému objektu. Zástupca tiež môže zúžiť rozhranie poskytované pôvodným objektom z bezpečnostných dôvodov.

Dekoratér – aj keď implementácia Dekoratéra je podobná ako pri Zástupcovi, ich účel je odlišný. Dekoratér rozširuje rozhranie pôvodného objektu. Zástupca ho poskytuje nezmenené alebo obmedzené.

3 VYBRANÉ VZORY SPRÁVANIA

Čo v skutočnosti rozumieme pod pojmom správanie v objektovo orientovanom svete obzvlášť? Aké typy správania nám definujú takto pomenované vzory? Pokiaľ nechceme alebo nepotrebujeme odpoveď v kontexte konkrétneho vzoru môžeme povedať, že ide o definíciu všeobecného postupu, ktorý je konkretizovaný inštanciou samotného vzoru.

Vzory správania resp. ich charakteristická skupina nám umožňujú alternovať nad konkrétnym postupom v čase behu programu. Nie je potrebné určiť konkrétny prístup už v čase kompilácie, konkrétne postupy alebo správania sa dodržiavaním vzorom definovaných rozhraní stávajú vzájomne zameniteľné a z vonkajšieho pohľadu ekvivalentné.

3.1 Príkaz

Účel

Vzor zapuzdrí žiadosť na vykonanie operácie do objektu a tým umožní parametrizáciu klientov pomocou rôznych žiadostí.

Nástroje pre používateľské rozhrania obsahujú objekty (napr. tlačidla), ktoré vykonávajú žiadosť na základe vstupu od používateľa. Súpravy nástrojov však nemôžu implementovať žiadosť explicitne v tlačidle, pretože len aplikácia používajúca súpravu nástrojov vie, čo sa má s daným objektom vykonať. Vzor Príkaz umožňuje objektom súpravy nástrojov vystaviť žiadosť na nešpecifikované objekty tým, že žiadosť zmení do objektu. Toto oddelenie objektu vyvolávajúceho operáciu od objektu, ktorý vie, ako operáciu vykonať, poskytuje veľkú tvárnosť pri navrhovaní používateľských rozhraní.

Podstatou tohto vzoru je abstraktná trieda `Príkaz`, ktorá deklaruje rozhranie pre vykonávanie operácií. Konkrétne podtriedy triedy `Príkaz` špecifikujú dvojice príjemca-akcia tak, že ukladajú príjemcu ako inštančnú premennú a implementujú operáciu `Vykonaj` pre vyvolanie žiadosti.

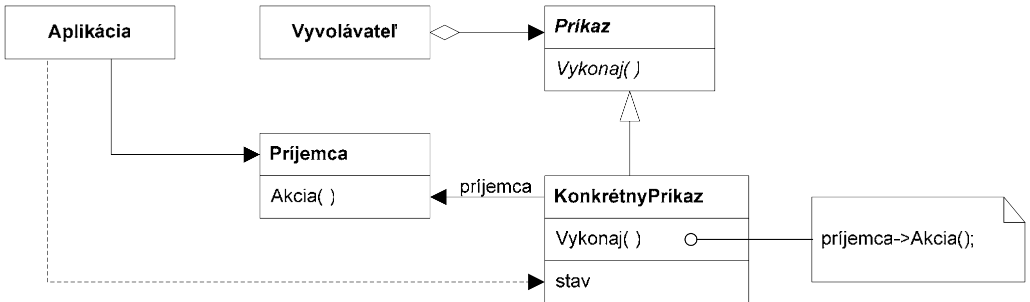
Vzor je použiteľný, ak chceme:

- parametrizovať objekty pomocou akcie, ktorú máme vykonať,
- žiadosti radiť do radu a vykonávať v rôznych okamihoch,
- podporovať funkcie Späť a Znovu (rozhraniu `Príkaz` je potrebné pridať operáciu `Odvolaj`, ktorá odvolá vplyvy prechádzajúceho volania `Vykonaj`),

- podporovať protokolovanie zmien, aby ich bolo možné znovu aplikovať v prípade havárie systému,
- štruktúrovať systém prostredníctvom vysokoúrovňových operácií postavených na primitívnych základoch. Takáto štruktúra je bežná v informačných systémoch podporujúcich transakcie.

Použitie vzoru ukážeme na jednoduchom príklade vypínača, ktorý riadi spínanie ventilátora a osvetlenia. Ventilátor aj osvetlenie majú rozdielne rozhrania, čo znamená, že vypínač musí mať rozhranie nezávislé na príjemcovi, alebo nevie, aké je rozhranie príjemcu. Potrebujeme teda parametrizovať každý z vypínačov prislúchajúcim príkazom. Je zrejmé, že osvetlenie aj ventilátor budú používať iný príkaz.

Štruktúra



Obrázok 3-1. Štruktúra vzoru Príkaz.

Súčasti

`Príkaz` – deklaruje rozhranie k vykonaniu operácie.

`KonkrétnyPríkaz` (Rozsvietiť osvetlenie, Spustiť ventiláciu)

- definuje väzbu medzi objektom `Príjemca` a akciou,
- implementuje `Vykonaj` vyvolaním odpovedajúcej operácie, či operácií na objekte `Príjemca`.

`Aplikácia` – vytvára objekt `KonkrétnyPríkaz` a nastavuje jeho príjemcu.

`Vyvolávateľ` (vypínač) dáva pokyn príkazu, aby žiadosť vykonal.

`Príjemca` (osvetlenie, ventilátor) – vie, ako vykonať operácie asociované s výkonom žiadosti. Ako `Príjemca` môže slúžiť ľubovoľná trieda.

Dôsledky

Vzor odpája objekt, ktorý operáciu vyvoláva od objektu, ktorý pozná spôsob jej vykonania. Objekty `Príkaz` Je možné rozširovať a zaobchádzať s nimi ako s inými objektmi. Príkazy je možné skladať do zloženého príkazu. Je možné jednoducho pridávať nové triedy `Príkaz`, pretože sa nemusia meniť existujúce triedy.

Implementácia

Príkaz môže mať rôzny rozsah schopností. Na jednej strane iba definuje väzbu medzi príjemcom a akciami, ktoré žiadosť vykonávajú. Na druhej strane implementuje všetko sám a nič nedeleguje na príjemcu. Doplnením operácie `Odvolaj` môže podporovať funkcie Späť a Znovu.

Príklad

```
public interface Command {
  public abstract void execute();
}

class Switch {
  private Command UpCommand, DownCommand;
  public Switch(Command Up, Command Down) {
    UpCommand = Up;
    DownCommand = Down;
  }
  void flipUp() {
    //vyvolavatel zavola konkretny prikaz, ktory vykona
    //prikaz na prijemcovi
  UpCommand.execute();
  }
  void flipDown() {
    DownCommand.execute();
  }
}

//spinanie osvetlenia
class LightOnCommand implements Command {
  private Light myLight;
  public LightOnCommand (Light L) {
    myLight = L;
  }
  public void execute() {
    myLight.turnOn();
  }
}

//spinanie ventilatora
class FanOnCommand implements Command {
  private Fan myFan;
  public FanOnCommand(Fan F) {
    myFan = F;
  }
  public void execute() {
    myFan.startRotate();
  }
}

public class TestCommand {
  public static void main(String[] args) {
    Light  testLight = new Light( );
    LightOnCommand testLOC = new LightOnCommand(testLight);
    LightOffCommand testLFC = new LightOffCommand(testLight);
    Switch testSwitch = new Switch( testLOC,testLFC);
    testSwitch.flipUp( );
```

```
        testSwitch.flipDown( );
        Fan testFan = new Fan( );
        FanOnCommand foc = new FanOnCommand(testFan);
        FanOffCommand ffc = new FanOffCommand(testFan);
        Switch ts = new Switch( foc,ffc);
        ts.flipUp( );
        ts.flipDown( );
}
```

Príklad 3-1. Ukážka implementácie vzoru Príkaz.[2]

Príbuzné vzory

Pomocou vzoru Zloženina je možné implementovať zložený príkaz. Vzor Memento môže udržiavať stav, ktorý príkaz vyžaduje k odvolaniu svojho účinku. Príkaz, ktorý sa musí skopírovať predtým, než sa umiestni do zoznamu histórie, sa správa ako Prototyp.

3.2 Reťaz zodpovednosti

Účel

Vzor sa vyhýba spojeniu odosielateľa žiadosti s príjemcom tak, že umožní spracovať žiadosť viac ako jednému objektu. Objekty príjemcov zreťazí a žiadosť odovzdáva pomocou vytvorenej reťaze, kým ju nejaký objekt nespracuje.

Prvý objekt v reťazi príjme žiadosť a spracuje ju, alebo ju odovzdá ďalšiemu kandidátovi v reťazi, ktorý urobí to isté. Keďže objekt, ktorý zadal žiadosť, nemá žiadnu explicitnú znalosť o tom, kto žiadosť spracuje, hovoríme, že žiadosť má implicitného príjemcu. Ideou tohto vzoru je odpojiť odosielateľa a príjemcu tak, že sa dá možnosť spracovať žiadosť viacerým objektom.

Vzor je použiteľný, ak:

- žiadosť môže spracovať viac ako jeden objekt a spracovateľ nie je vopred známy; spracovateľ sa má určiť automaticky,
- žiadosť chceme vystaviť jednému z viacerých objektov bez toho, aby sme explicitne určovali príjemcu,
- množina objektov, ktoré môžu žiadosť spracovať, má byť určená dynamicky.

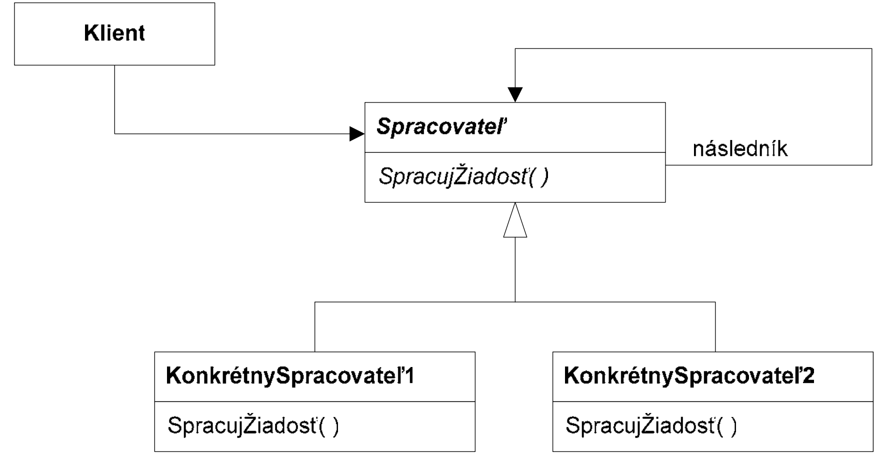
Použitie reťaze zodpovednosti môžeme nájsť pri realizácii filtra elektronickej pošty. Spracovateľ, ktorý je na začiatku reťaze príjme novú požiadavku (v tomto prípade novú správu). Ak je správa vyhodnotená ako nevyžiadaná (*spam*), nepokračuje ďalej v reťazi. Naopak dôveryhodné správy pokračujú k ďalšiemu spracovateľovi, ktorým môže byť iný filter. Posledný spracovateľ v reťazi môže správu, potom čo prešla filtrami, uložiť.

---

[2] Zdroj: `www.javaworld.com`

Štruktúra

Potrebujeme zaistiť, aby pri odovzdávaní žiadosti ostali príjemcovia implicitní. To zabezpečíme tak, že každý objekt v reťazi zdieľa spoločné rozhranie pre spracovanie žiadosti a pre prístup k jeho následníkovi v reťazi.



Obrázok 3-2. Štruktúra vzoru Reťaz zodpovednosti.

Súčasti

`Spracovateľ`

- definuje rozhranie k spracovaniu žiadosti,
- (voliteľné) implementuje prepojenie na následníka.

`KonkrétnySpracovateľ`

- spracováva žiadosti, za ktoré je zodpovedný,
- môže pristupovať k svojmu následníkovi,
- ak dokáže žiadosť spracovať, tak ju spracuje; inak ju posunie svojmu následníkovi.

`Klient` – inicializuje žiadosť na objekt `KonkrétnySpracovateľ` v reťazi.

Dôsledky

Vzor oslobodzuje objekt od znalosti, ktorý iný objekt žiadosť spracováva. Ani odosielateľ ani príjemca o sebe explicitne nevedia. Namiesto toho, aby objekty udržiavali odkazy na všetkých potenciálnych príjemcov, udržujú odkaz len na následníka.

Zodpovednosť za spracovanie žiadosti je možné pridať alebo zmeniť za behu pridaním objektu do reťaze alebo inou úpravou, čím je pridaná tvárnosť pri priraďovaní povinností objektom.

Keďže žiadosť nemá explicitného príjemcu, nie je zaručené jej spracovanie. Žiadosť môže prejsť celou reťazou bez spracovania.

Implementácia

Vzor potrebuje mať vytvorenú reťaz objektov. Pri vytváraní reťaze sa môžu využiť existujúce prepojenia, alebo sa môžu definovať nové. Ak neexistujú vopred dané odkazy pre definovanie reťaze, musíme ich zaviesť sami. V tomto prípade trieda `Spracovateľ` definuje nie len rozhranie pre žiadosti, ale aj udržuje následníka. Žiadosť vstupujúca do reťaze môže byť:

- pevne naprogramovaná – môže odovzdávať len pevnú sadu žiadostí,
- vyjadrená funkciou, ktorej parameter je kód žiadosti,
- odovzdávaná pomocou zvláštnych objektov, ktoré budú k sebe pribaľovať parametre žiadosti,
- automaticky odovzdávaná (napr. v jazyku *Smalltalk)*.

Príklad

```
public abstract class Handler {
  ...
  public void handleRequest(SomeRequestObject sro) {
    if(successor != null)
      successor.handleRequest(sro);
  }
}

public class SpamFilter extends Handler {
  public void handleRequest(SomeRequestObject mailMessage) {
    if(isSpam(mailMessage)) {
      //Ak je email nevyziadany, vykonaj suvisiacu operaciu
    }
    else {
      super.handleRequest(mailMessage);
      //Ak je doveryhodny, posun ho dalsiemu filtru
    }
  }
}
```

Príklad 3-2. Implementácia filtra nevyžiadanej pošty.[3]

Príbuzné vzory

Reťaz zodpovednosti sa často aplikuje v spojení so vzorom Zloženina. V tejto situácii sa môže rodič komponentu správať ako jeho následník.

[3] Zdroj: `www.javaworld.com`

3.3 Interpret

Účel

Vzor Interpret je zvyčajne opísaný z hľadiska interpretácie gramatík. V danom jazyku definuje vyjadrenie gramatiky vrátane interpreta, ktorý vyjadrenie použije na interpretáciu viet.

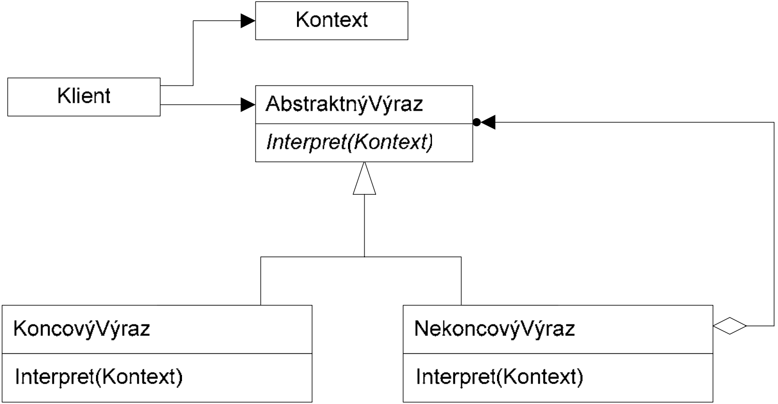
Ak sa nejaký problém vyskytuje častejšie, môže byť užitočné vyjadriť inštanciu problému ako vetu v jednoduchom jazyku. Vzor opisuje, ako definovať gramatiku pre jednoduché jazyky, vyjadrovať vety v jazyku a interpretovať ich.

Vzor je použiteľný, ak:

- existuje jazyk k interpretácii a je možné v ňom vyjadriť vety v podobe abstraktných syntaktických stromov,
- je gramatika jednoduchá, keďže v zložitejších gramatikách sa triedna hierarchia stáva príliš veľkou a ťažko spravovateľnou,
- účinnosť nepredstavuje kritický faktor, pretože najúčinnejšie interpretery nie sú spravidla implementované k priamej interpretácii analyzačných stromov, ale k ich úvodnému prevodu do iného tvaru.

Ako príklad môžeme uviesť regulárne výrazy, kde vzor opisuje, ako definovať gramatiku pre regulárne výrazy, vyjadriť určitý regulárny výraz a interpretovať ho. Každý regulárny výraz definovaný touto gramatikou je vyjadrený abstraktným syntaktickým stromom. Pre tieto regulárne výrazy môžeme vytvoriť interpreta definovaním operácie `Interpret` v každej podtriede triedy `AbstraktnýVýraz`. Jeho argumentom je kontext, v ktorom sa výraz prekladá.

Štruktúra



Obrázok 3-3. Štruktúra vzoru Interpret.

Súčasti

`Kontext` – obsahuje informácie, ktoré sú pre interpreta globálne.

`Klient`

- vytvára (obdrží) abstraktný syntaktický strom vyjadrujúci vetu v jazyku definovanom pomocou gramatiky,
- volá operáciu `Interpret`.

`AbstraktnýVýraz` – deklaruje abstraktnú operáciu `Interpret` spoločnú pre všetky uzly v abstraktnom syntaktickom strome.

`KoncovýVýraz`

- implementuje operáciu `Interpret` asociovanú s koncovými symbolmi.
- Pre každý koncový symbol vo vete je nevyhnutná jedna inštancia.

`NekoncovýVýraz`

- Pre každé pravidlo gramatiky $R::=R_1R_2...R_n$ je potrebná jedna takáto trieda.
- Implementuje operáciu `Interpret` pre nekoncové symboly gramatiky.
- Udržuje inštančné premenné typu `AbstraktnýVýraz` pre každý symbol R_1 až R_n.

Dôsledky

Vzor používa triedy k vyjadreniu pravidiel gramatiky. Použitím dedičnosti môžeme gramatiku meniť alebo rozširovať. Triedy, ktoré definujú uzly v abstraktnom syntaktickom strome majú podobné implementácie, a preto ich generovanie môžeme zautomatizovať. Ak zadefinujeme na výrazových triedach nové operácie, získame nový spôsob interpretácie výrazov.

Pre každé pravidlo gramatiky je potrebné vytvoriť najmenej jednu triedu, preto je komplikované udržiavať gramatiky obsahujúce veľké množstvo pravidiel.

Implementácia

Vzor nevysvetľuje, ako vytvoriť abstraktný syntaktický strom. Operáciu `Interpret` nie je potrebné definovať vo výrazových triedach, ale ak bežne vytvárame nového interpreta, je výhodnejšie použiť vzor Návštevník. Ak gramatika obsahuje veľa koncových symbolov, môže využiť zdieľanie jednej kópie (vzor Mušia váha).

Príbuzné vzory

Interpret môže spolupracovať so vzormi Zloženina, Mušia váha, Iterátor a Návštevník. Abstraktný syntaktický strom je inštanciou vzoru Zloženina, vzor Mušia váha má uplatnenie pri zdieľaní koncových symbolov a vzor Návštevník je možné použiť k udržiavaniu správania v každom uzle abstraktného syntaktického stromu v jednej triede.

3.4 Iterátor

Účel

Poskytuje spôsob sekvenčného prístupu k prvkom zoskupeného objektu bez toho, aby sa odhalilo jeho vnútorné vyjadrenie.

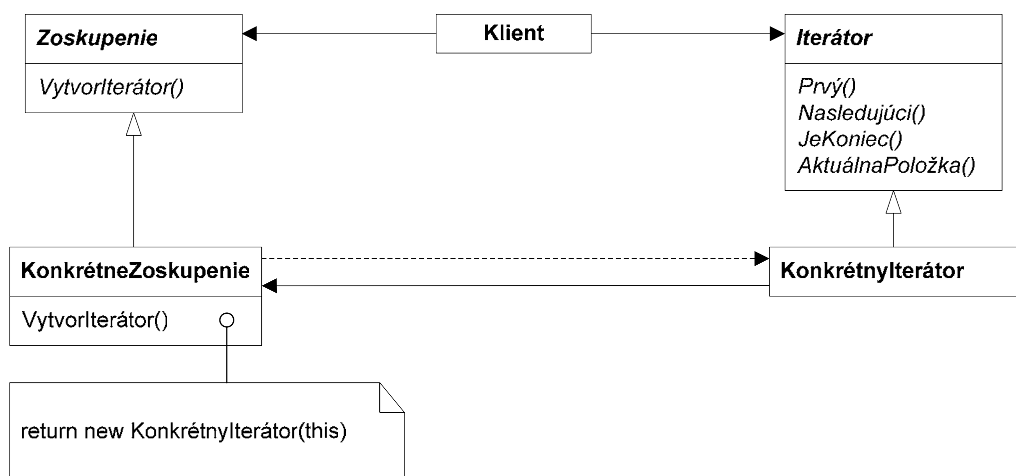
Navyše umožňuje doplniť podporu o viaceré spôsoby prechádzania. Podstatou je odobratie zodpovednosti za prístup a prechádzanie z objektu zoznamu do objektu iterátora. Iterátor a zoznam sú spolu spojení, a teda klient musí vedieť, že sa jedná prechádzaný zoznam a nie len o nejakú zoskupenú štruktúru.

Vzor je použiteľný, ak

- potrebujeme pristupovať k obsahu zoskupeného objektu bez toho, aby sme odhalili jeho vnútorné vyjadrenie,
- podporujeme viac spôsobov prechádzania zoskupených objektov,
- poskytujeme jednotné rozhranie k prechádzaniu rôznych zoskupených štruktúr.

Vzor Iterátor patrí medzi veľmi často používané vzory. Iterátory sa bežne vyskytujú v objektovo-orientovaných systémoch. Väčšina knižníc tried ponúka iterátory v najrôznejších tvaroch.

Štruktúra



Obrázok 3-4. Štruktúra vzoru Iterátor.

Súčasti

`Iterátor` – definuje rozhranie pre prístup a prechádzanie prvkov.

`KonkrétnyIterátor`

- implementuje rozhranie `Iterátor`,
- sleduje aktuálnu pozíciu pri prechádzaní zoskupenia a môže určiť nasledujúci objekt prechádzania.

`Zoskupenie` – definuje rozhranie k tvorbe objektu `Iterator`.

`KonkrétneZoskupenie` – implementuje rozhranie k tvorbe objektu `Iterátor` a vracia inštancie riadnej triedy `KonkrétnyIterátor`.

Dôsledky

Vzor podporuje variácie pri prechádzaní zoskupení. Iterátory uľahčujú zmeny algoritmu prechádzania náhradou inštancie iterátora za inú. Pre podporu nových druhov prechádzania definujeme nové podtriedy triedy `Iterátor`. Každý iterátor sleduje vlastný stav prechádzania, a preto môže v zoskupení prebiehať súčasne viac prechádzaní. Iterátor zjednodušuje rozhranie `Zoskupenie`, keďže rozhranie na prechádzanie je umiestnené v iterátore.

Implementácia

Iterácie môžu byť riadené iterátorom (interný iterátor) alebo klientom (externý iterátor), ktorý ho používa. Externé iterátory sú tvárnejšie. Pri externom iterátore musí klient explicitne požiadať o ďalší prvok a posun pri prechádzaní. Interný iterátor aplikuje operáciu od klienta na každý prvok zoskupenia. Pri implementácii iterátora je potrebné myslieť na jeho robustnosť, aby po pridávaní alebo odstránení prvkov zo zoskupenia nedošlo pri prechádzaní napr. k vynechaniu prvku. Ďalšie rozšírenie iterátora o operáciu `Predchadzajuci` umožní prechod na predchádzajúci prvok.

Príbuzné vzory

Iterátory sa často aplikujú na rekurzívne štruktúry, ktoré sa podobajú vzoru Zloženina. Pozícia v štruktúre môže dosiahnuť niekoľko úrovní. Externý iterátor si musí cestu skladbou ukladať, aby sledoval aktuálny objekt. Výrobná metóda sa používa pri tvorbe inštancií podtried triedy `Iterátor` pri polymorfných iterátoroch. K zachyteniu stavu iterácie je možné použiť vzor Memento.

3.5 Sprostredkovateľ

Účel

Definuje objekt, ktorý zapuzdruje interakciu sady objektov. Sprostredkovateľ uprednostňuje voľné spojenie tak, že bráni objektom v explicitných vzájomných odkazoch a umožňuje ich interakciu meniť nezávisle.

Objektovo orientovaný návrh podporuje distribúciu správania medzi objektmi. Výsledkom toho rozdelenia môže byť objektová štruktúra s mnohými prepojeniami medzi objektmi – v najnepriaznivejšom prípade nakoniec každý objekt vie o všetkých ostatných.

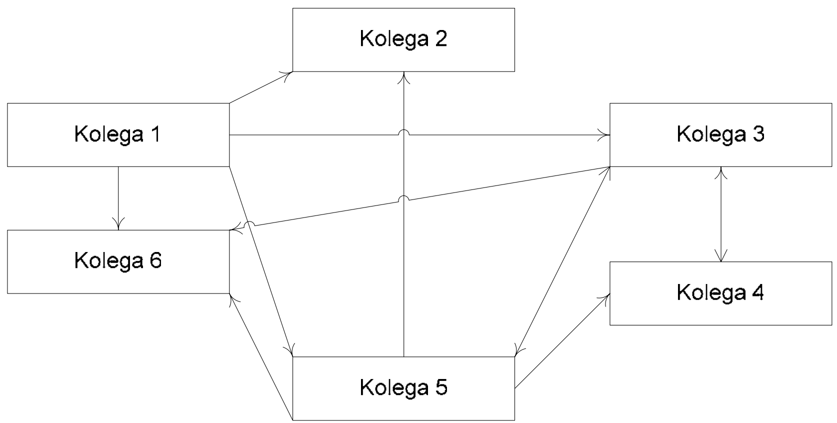
Aj keď rozdelenie systému do mnohých objektov vo všeobecnosti zdokonaľuje znovupoužitie, príliš veľký nárast vzájomných spojení má tendenciu ho zredukovať. Mnoho vzájomných spojení znižuje pravdepodobnosť, že objekt môže fungovať bez podpory ostatných – systém sa správa ako keby bol monolitický. Naviac môže byť obtiažne meniť správanie systému akýmkoľvek výraznejším spôsobom, lebo toto správanie je distribuované medzi mnohé objekty. Výsledkom toho je, že sme nútení definovať mnoho podtried za účelom modifikácie správania systému.

Ako príklad môžeme uvažovať o implementácii dialógových okien v grafickom používateľskom rozhraní. Dialóg používa okno k vyjadreniu kolekcie pomôcok, napríklad tlačidiel, ponúk a vstupných polí. Medzi pomôckami dialógu často existujú závislosti. Napr. tlačidlo je deaktivované, keď je dané vstupné pole prázdne. Výber položky z viacerých možností, nazývaných jednoducho zoznam, môže zmeniť obsah vstupného poľa. Naopak, zadanie textu do vstupného poľa môže automaticky vybrať jednu alebo viac odpovedajúcich položiek v zozname. Keď sa text zobrazí vo vstupnom poli, môžu sa aktivovať ďalšie tlačidla umožňujúce používateľovi pracovať s textom, napríklad meniť alebo odstraňovať vec, na ktorú ukazuje.

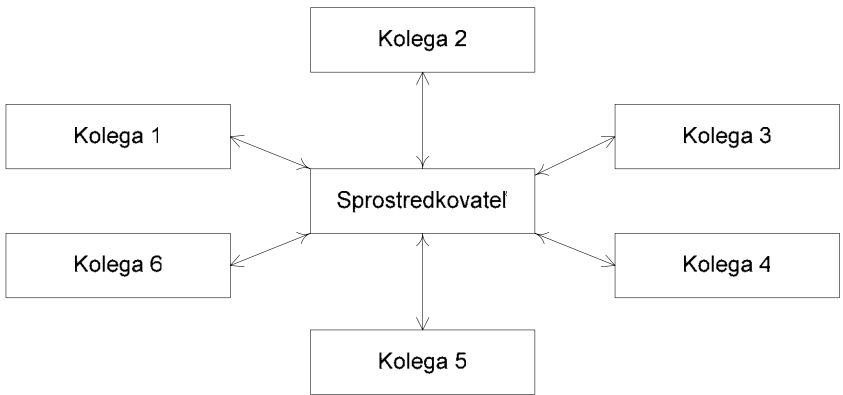
Rôzne dialógy majú rôzne závislosti medzi pomôckami. Takže aj keď dialógy zobrazujú rovnaké druhy pomôcok, nemôžu jednoducho znovupoužiť jednoduché pomôckové triedy; musia sa upraviť, aby odrážali špecifické správanie dialógu. Ich individuálna úprava pomocou tvorby podtried je prácna, pretože sa týka mnohých podtried.

Týmto problémom sa môžeme vyhnúť zapúzdrením kolektívneho správania do oddeleného objektu: sprostredkovateľa. Sprostredkovateľ je zodpovedný za riadenie a koordinovanie interakcií skupiny objektov.

Štruktúra



Obrázok 3-5. Ad hoc štruktúra.



Obrázok 3-6. Štruktúra so sprostredkovateľom.

Použitie

Vzor Sprostredkovateľ používame v týchto situáciách:

- Sada objektov komunikuje dobre definovaným, ale komplexným spôsobom. Výsledné vzájomné závislosti sú neštruktúrované a obtiažne pochopiteľné.
- Znovupoužitie objektu je obtiažne, pretože sa odkazuje a komunikuje s mnohými inými objektmi.
- Správanie, ktoré je distribuované do niekoľkých tried, by malo byť upraviteľné bez toho, aby bolo potrebné vytvárať mnoho podtried.

Súčasti

Sprostredkovateľ – definuje rozhranie pre komunikáciu kolegov.

Konkrétny sprostredkovateľ

- implementuje kooperatívne chovanie tým, že koordinuje kolegov,
- pozná všetkých kolegov tvoriacich jeden tým.

Kolega

- Každá trieda kolegu pozná svojho sprostredkovateľa.
- Každý kolega komunikuje so svojimi kolegami cez sprostredkovateľa vtedy, keď by inak komunikoval s iným kolegom priamo.

Spolupráca

Kolegovia posielajú a prijímajú žiadosti od sprostredkovateľa. Sprostredkovateľ implementuje kooperatívne správanie pomocou smerovania žiadostí medzi príslušnými kolegami.

Dôsledky

Vzor Sprostredkovateľ má tieto výhody a nevýhody:

- Obmedzuje nutnosť tvorby nových a nových podtried. Sprostredkovateľ sústreďuje správanie, ktoré by inak bolo rozdelené medzi niekoľko objektov. Zmenu špecifického správania celku je možné realizovať na jedinom mieste – zmenou implementácie konkrétneho sprostredkovateľa.
- Oddeľuje kolegov. Sprostredkovateľ uľahčuje to, aby boli kolegovia spojení voľne. Triedy kolegov a sprostredkovateľov je možné nezávisle meniť a znovupoužiť.
- Zjednodušuje objektový protokol. Sprostredkovateľ nahrádza interakcie typu „n ku m“ na interakcie typu „jedna ku n“ – medzi sebou a svojimi kolegami. Vzťahy typu „jedna ku n“ sú ľahšie pochopiteľné a rozšíriteľné.
- Zabstraktňuje spôsob spolupráce objektov. Vytvorenie sprostredkovateľa ako nezávislého konceptu a jeho zapúzdrenie do objektu nám umožňuje zamerať sa na to, ako objekty reagujú okrem svojho individuálneho správania. To môže pomôcť pri vyjasnení interakcií objektov v systéme.
- Centralizuje riadenie. Vzor Sprostredkovateľ mení komplexnosť interakcií za zložitosť v sprostredkovateľovi. Pretože sprostredkovateľ zapuzdruje protokoly,

môže sa stať komplexnejší než jednotlivý kolega. To môže zo sprostredkovateľa samotného urobiť monolit, ktorý sa neudržuje jednoducho.

Príbuzné vzory

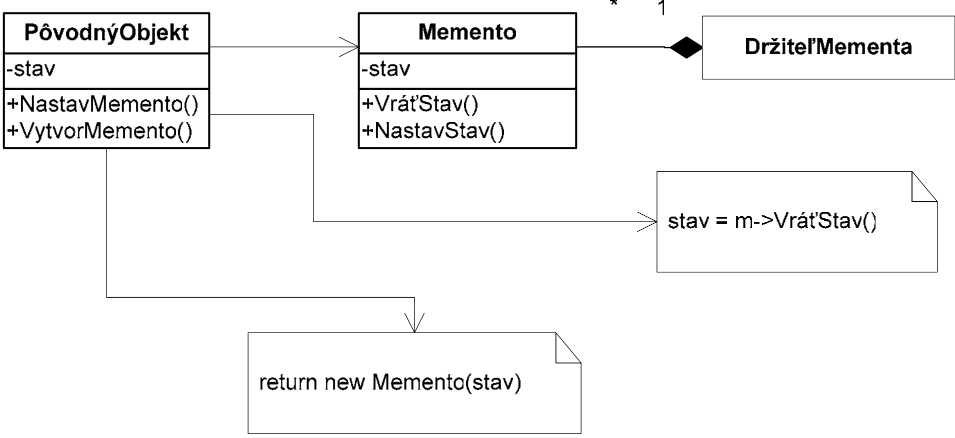
Fasáda sa líši od sprostredkovateľa tým, že zabstraktnením subsystému poskytuje vhodnejšie rozhranie. Protokol fasády je jednosmerný: objekty Fasáda sa odvolávajú na systémové triedy, ale už nie naopak. Na rozdiel od toho Sprostredkovateľ aktivuje kooperatívne správanie, ktoré spolupracujúce objekty neposkytujú či poskytovať nemôžu a protokol je viacsmerný. Kolegovia môžu so sprostredkovateľom komunikovať pomocou vzoru Pozorovateľ.

3.6 Memento

Účel

- Slúži na zapamätanie stavu objektu, aby sa neskôr mohol na základe mementa obnoviť.
- Stav objektu (memento) je uchovaný objektom, ktorý požiadal o uchovanie stavu daného objektu.
- Memento je pasívny objekt.

Štruktúra



Obrázok 3-7. Návrhový vzor Memento.

Súčasti

`PôvodnýObjekt` – objekt, ktorého stav si zapamätáme v memente.

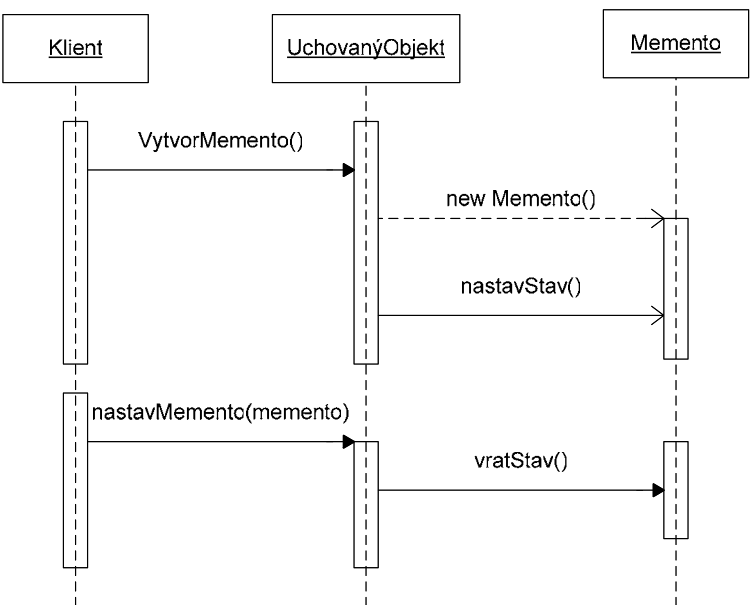
`Memento` – pasívny objekt, v ktorom je uchovaný stav pôvodného objektu.

`DržiteľMementa` – objekt, ktorý si vyžiadal vygenerovanie mementa a drží ho, aby niekedy mohol vrátiť objekt do pôvodného stavu.

Dôsledky

Serializácia stavu objektu je potrebná v prípadoch, keď potrebujeme obnoviť stav objektu v budúcnosti, napríklad v prípade načítania uložených objektov na bezpečné médium alebo na obnovu predchádzajúceho stavu pri odvolaní posledných vykonaných operácií (*undo*).

Memento je pasívny objekt, využije sa iba pri vytvorení a pri obnove stavu uchovávaného objektu. Memento drží objekt, ktorý jeho vytvorenie vyžiadal.



Obrázok 3-8. Sekvenčný diagram interakcií medzi klientom, uchovaným objektom a mementom.

Implementácia

Zapamätanie stavu môže byt v niektorých prípadoch zložité alebo pamäťovo náročné. Konkrétna implementácia mementa závisí od prípadu použitia. Ak objekt deklaruje operácie a súčasne aj inverzné operácie, potom memento môže byť sekvencia operácií, ktoré vrátia objekt do stavu, v ktorom bolo memento vyžiadané. V tomto prípade pri vykonaní operácie objekt pridá do mementa inverznú operáciu. Uchovanie sekvencie inverzných operácií sa môže implementovať návrhovým vzorom Príkaz. Pri ďalšej požiadavke sa vytvorí ďalšie memento s prázdnym zoznamom inverzných operácií. K predchádzajúcemu mementu sa určí referenciou nové memento. Takto sa vytvorí kaskáda objektov, ktorá obsahuje zoznamy operácií na vrátenie uchovávaného objektu do požadovaného stavu.

Príklad

Editor vi si zapamätáva každú vykonanú zmenu v texte. Pri zavolaní príkazu *:undo* zvráti zmenu podľa týchto uložených dát. Dokáže vykonať obnovu aj druhým smerom: pri nestabilite systému z pôvodného súboru a zo zaznamenaných údajov vie obnoviť

text do stavu, v ktorom nastal systémový výpadok. V tomto kontexte súbor na zaznamenanie vykonaných textových operácií je memento.

Operácia *undo* v prostredí na kreslenie diagramov má zabezpečiť, že zmenené škatuľky sa dostanú do pôvodnej polohy a súčasne referencie medzi nimi sa obnovia podľa zapamätaného stavu. Kvôli tomu sa vykoná memento nielen pre objekty škatuliek, ale aj pre riešenie referencií (*constraint solver*).

Príbuzné vzory

V memente sa často uchovávajú objekty návrhové vzoru Príkaz. Vykonaním ich inverzných príkazov sa objekt dostane do pôvodného stavu. Na memento sa niekedy odvoláva ako na externalizáciu, alebo serializáciu stavu objektu.

4 J2EE VZORY

Sú J2EE vzory naozaj návrhovými vzormi? Nájsť jednoznačnú odpoveď asi nebude také ľahké. Svojim formálnym a štruktúrovaným opisom by sa dali bez problémov zaradiť do tejto kategórie vzorov. Na druhej strane sú J2EE len akousi špecifickou aplikáciou niektorého z návrhových vzorov opísaných v (Gamma, 1995). J2EE vzory ponúkajú komplexnejšie riešenia ako samotné návrhové vzory, ide o celkový prístup k vývoju viacvrstvových a distribuovaných aplikácií implementovaných v jazyku JAVA. Ako prostriedky pre realizáciu J2EE vzory využívajú dobre definované návrhové vzory.

V nasledujúcich podkapitolách sa uvádzame vzory vybrané z pätnástich základných J2EE vzorov.

4.1 Klientska abstrakcia biznis služieb

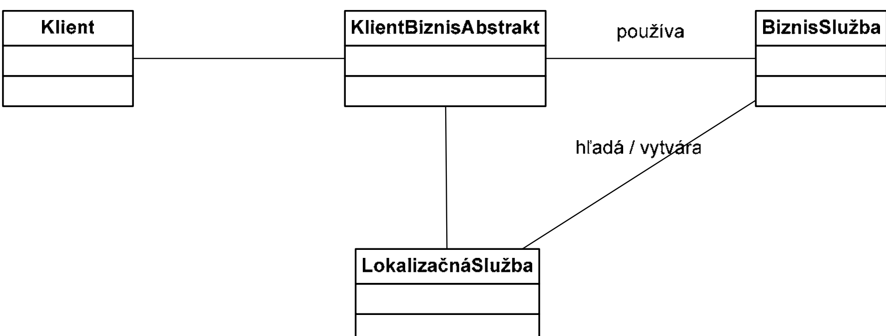
U viacvrstvových aplikácií je potrebné medzi iným oddeliť klientsku (prezentačnú) vrstvu od biznis vrstvy. Je teda potrebné vyčleniť isté rozhrania z biznis logiky a tieto istou formou poskytnúť rôznorodým klientom.

Účel

Vzor Klientska abstrakcia biznis služieb, v angličtine známy ako *Business Delegate*, je určený pre oddelenie biznis logiky od vyšších vrstiev a poskytnutie služieb rôznym klientom alebo prezentačnej vrstve. Vzor sa využíva pri často sa meniacich požiadavkách na biznis logiku, pretože definuje rozhrania pre vyššie vrstvy, ktoré by mali byť odolné voči zmenám biznis logiky.

Vzor umožňuje implementáciu klientskej vyrovnávacej pamäti pre minimalizáciu volaní vzdialených metód, čím redukuje sieťové zaťaženie.

Štruktúra



Obrázok 4-1. Vzťahy medzi triedami vo vzore Klientska abstrakcia biznis služieb.

Súčasti

`Klient` – príklad klienta pristupujúceho (využívajúceho) k biznis službám. Týmto klientom môže byť i vyššia vrstva aplikácie, napr. prezentačná vrstva.

`KlientBiznisAbstrakt` – samotná klientska abstrakcia biznis služby. Kontroluje a chráni biznis vrstvu pred klientmi. Pre lokalizáciu samotnej služby v zložitej spleti biznis logiky môže, ale nemusí, využívať služby `LokalizačnáSlužba`.

`BiznisSlužba` – konkrétna biznis služba, resp. systém biznis služieb, ktoré majú byť sprístupnené klientom.

`LokalizačnáSlužba` – nepovinná súčasť vzoru umožňujúca oddeliť a centralizovať mechanizmy lokalizácie biznis služieb. Využíva sa pri distribuovanej realizácií biznis logiky.

Dôsledky

- umožňuje implementovať klientsku vyrovnávaciu pamäť,
- umožňuje prístup rôznym typom klientov k biznis službám,
- oddeľuje biznis logiku od jej použitia vo vyšších vrstvách aplikácie ako i u iných druhoch klientov,
- umožňuje realizáciu zmien biznis logiky bez nutnosti zmeny klientov, resp. vyšších vrstiev aplikácie,
- skrýva implementačné detaily biznis logiky,
- v prípade potreby dokáže transformovať výnimky biznis vrstvy na výnimky pre prezentačnú vrstvu.

Implementácia

Delegovaný zástupca – vyjadruje rozhranie, ktoré pre klientov sprístupňuje dôležité biznis služby. Pri tomto prístupe existuje istá paralela so vzorom Zástupca.

Delegovaný adaptér – využíva sa k pripájaniu B2B služieb, služieb založených na J2EE architektúre. Prístup je realizovaný využitím vzoru Adaptér.

Príklad

```
public class ResourceDelegate
{
  // vzdialeny dokaz na zdroj
  private ResourceSession session;

  // biznis sluzba
  private static final Class homeClazz =
  corepatterns.apps.psa.ejb.ResourceSessionHome.class;

  // konstruktor, najde sluzbu a pripoji sa k nej vytvorenim
  // instancie
  public ResourceDelegate() throws ResourceException {
    try
```

```
    {
      ResourceSessionHome home = (ResourceSessionHome)
      ServiceLocator.getInstance().getHome("Resource",
      homeClazz);
      session = home.create();
    }
    catch(Exception ex)
    {
      // preloz vynimku do formatu vyssej vrstvy
      throw new GUIException(...);
    }
  }

  // biznis metoda spristupnena klientom
  // spolu s konverziou vynimiek do formatu vyssej vrstvy
  public SkladovaPolozka zaskladni(SkladovaPolozka sklPolozka)
  throws ResourceException
  {
    try
    {
      return session.zaskladni(sklPolozka);
    }
    catch (Exception ex)
    {
      // preloz vynimku do formatu vyssej vrstvy
      throw new GUIException(...);
    }
  }
  // vsetky ostantne biznis metody pre spristupnenie klientom
  ...
}
```

Príklad 4-1. Ukážka implementácie KlientBiznisAbstrakt ako delegovaného zástupcu.

Príbuzné vzory

Vzor sa využíva v spojení so vzorom Zástupca alebo Adaptér.

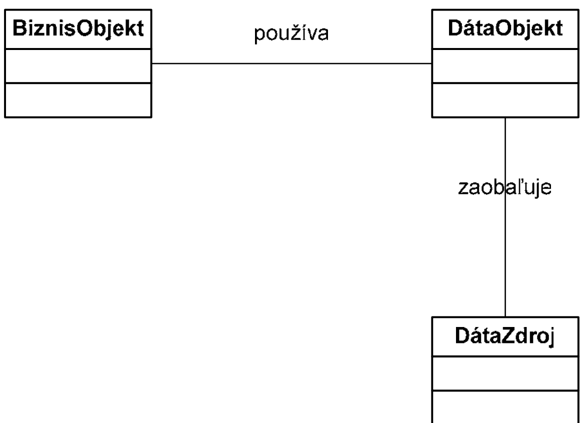
4.2 Dáta pristupujúci objekt

V súčasnosti je jedným z dôležitých rozhodnutí pri vytváraní informačných systémov práve výber vhodného dátového úložiska. Existujú rôzni výrobcovia, rôzne druhy a rôzne implementácie úložísk. Rozhodnutie o výbere konkrétneho úložiska môže, ale nemusí mať dopad na celú aplikáciu.

Účel

Návrhový vzor Dáta pristupujúci objekt, v angličtine známy ako *Data Access Object*, umožňuje voľnú zmenu dátového zdroja, bez nutnosti veľkých zmien v rôznych častiach a vrstvách aplikácie. Vzor zjednocuje prístup k rôznym dátovým zdrojom, napr. RDBMS, OODBMS, XML, LDAP a skrýva implementáciu konkrétneho prístupu pred inými vrstvami aplikácie.

Štruktúra



Obrázok 4-2. Vzťahy medzi objektmi vo vzore Dáta pristupujúci objekt.

Súčasti

`BiznisObjekt` – je v tomto prípade samotný klient. `BiznisObjekt` potrebuje k svojej činnosti perzistentné dáta, ktoré získava cez `DátaObjekt`.

`DátaObjekt` –reprezentuje `DátaZdroj`. `DátaObjekt` môže byť realizovaný rôznymi prístupmi, no myšlienka zaobaliť `DátaZdroj` zostáva vo všetkých prípadoch zachovaná.

`DátaZdroj` – predstavuje dátové úložisko, ktorého údaje je potrebné sprístupniť klientovi.

Dôsledky

- `BiznisObjekt` môže používať dátový zdroj bez nutnosti jeho poznania,
- implementačné detaily spojené s prístupom k dátovému zdroju sú ukryté v objekte `DátaObjekt` alebo v implementácií nástroja tretej strany,
- vzor Dáta pristupujúci objekt nám umožňuje jednoduchú zmenu zdroja údajov bez potreby zmien v ostatných vrstvách zložitej aplikácie,
- kód spojený s prístupom k dátovému zdroju sme izolovali v objekte `DátaObjekt`, čím sme odľahčili `BiznisObjekt`,
- v jedinej vrstve sme centralizovali prístup k dátovému zdroju, no pridaním ďalšej vrstvy sme zvýšili komplexnosť aplikácie a znížili jej flexibilitu.

Implementácia

Prístup automatického generovania objektov `DátaObjekt` využíva možnosť použiť nástroje tretích strán, ktoré realizujú a zaobaľujú realizáciu mapovania medzi objektmi `DátaObjekt` a `DátaZdroj`. Takéto nástroje poskytujú rôzne komfortné rozhrania pre generovanie zdrojových kódov `DátaObjekt`, pre dopredné inžinierstvo (z tried `DátaObjekt` vygenerujú DDL skripty pre vytvorenie štruktúry zdroja dát), pre spätné inžinierstvo (z existujúcich dátových zdrojov generujú `DátaObjekt`).

Vybrané nástroje tohto typu podporujú rôzne moduly pre grafické modelovanie dátovej štruktúry.

Továreň pre `DátaObjekt` spája použitie preberaného vzoru so vzorom Abstraktná továreň. Továreň sa využíva ako centrum, ktoré je informované o aktuálne používanom dátovom zdroji a podľa toho dokáže vrátiť adekvátny `DátaObjekt`. Táto implementácia sa často využíva u špeciálnych typov aplikácií, ktoré počas svojho behu menia používané zdroje dát.

Príklad

```
// implemetnácia s využitím abstraktnej továrne
public abstract class DAOFactory {

  // zoznam podporovaných zdrojov
  public static final int CLOUDSCAPE = 1;
  public static final int ORACLE = 2;
  public static final int SYBASE = 3;
  ...
  // metódy pre prístupk rôznym dátam
  // konkrétna továreň bude tieto metódy implementovať
  public abstract CustomerDAO getCustomerDAO();
  public abstract AccountDAO getAccountDAO();
  public abstract OrderDAO getOrderDAO();
  ...

  public static DAOFactory getDAOFactory(int whichFactory) {
    switch (whichFactory) {
      case CLOUDSCAPE:
        return new CloudscapeDAOFactory();
      case ORACLE :
        return new OracleDAOFactory();
      case SYBASE :
        return new SybaseDAOFactory();
      ...
      default :
        return null;
    }  }  }
```

Príklad 4-2. Ukážka implementácie DátaObjekt cez Abstraktnú továreň.

Príbuzné vzory

Jeden z prístupov k implementácií využíva vzor Abstraktná továreň.

4.3 Abstrakcia biznis služieb

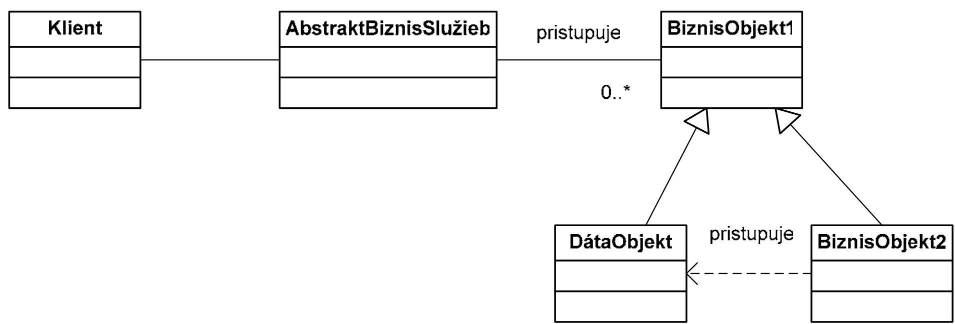
Biznis vrstva môže byť vytváraná formou elementárnych činností, ktorých spojením vznikne aplikačná logika. Aplikačná logika resp. akýkoľvek klient potom spája elementárne činnosti do procesov podľa vlastnej logiky.

Účel

Vzor Abstrakcia biznis služieb, v angličtine známy ako *Session Facade*, slúži podobne ako predchádzajúci vzor na oddelenie klienta od biznis služieb, pričom za klienta môže byť považovaná i vyššia vrstva aplikácie.

Využíva sa v prípade, že aplikačná logika alebo klientska aplikácia vyžaduje volanie biznis vrstvy. Vzor podobne ako predchádzajúce vzory znižuje nároky na vyťaženie siete.

Štruktúra



Obrázok 4-3. Vzťahy medzi triedami vo vzore Abstrakcia biznis služieb.

Súčasti

`Klient` – iná aplikácia alebo vyššia vrstva aplikácie vyžadujúca biznis služby.

`AbstraktBiznisSlužieb` – manažuje skupinu biznis objektov a prístupov k nim, čím vytvára biznis služby (resp. aplikačnú logiku). Zaobaľuje štruktúru biznis vrstvy od vyšších vrstiev aplikácie. Realizuje prístupový bod k biznis logike pre rôzne druhy aplikácií. Stará sa o manažment vytvárania, používania a zrušenia potrebných súčastí pre zaistenie biznis logiky.

`BiznisObjekt1`, `BiznisObjekt2`, `DátaObjekt` – podobne ako `DátaObjekt` a `BiznisObjekt2` predstavuje príklad zložitej siete biznis objektov, ktoré kooperáciou implementovanou v `AbstractBiznisSlužieb` vytvárajú poskytované služby.

`LokátorSlužieb` – nie je znázornený v štruktúre, pretože je nepovinným komponentom vo vzore. Jeho služby môže využívať `AbstraktBiznisSlužieb` pre lokalizáciu a vytvorenie potrebných biznis objektov podobným spôsobom ako v prípade vzoru Klientska abstrakcia biznis služieb trieda `KlientBiznisAbstrakt`.

Dôsledky

- Vzor môže reprezentovať kontrolnú vrstvu medzi klientom a biznis vrstvou,
- pri jednoduchých aplikáciách ide iba o implementáciu vzoru Zástupca nižších vrstiev v aplikácií,

- redukuje zviazanosť a zvyšuje manažovateľnosť, použitie vzoru znižuje závislosť klienta od biznis vrstvy,
- zvyšuje výkonnosť a redukuje použitie jemnozrnných metód (metódy vykonávajúce čiastočné činnosti s minimálnymi vstupmi, produkujúce malé výsledky),
- centralizuje bezpečnostný a transakčný mechanizmus.

Implementácia

Bezstavová implementácia – sa používa pri tzv. nekonverzačných biznis procesoch, kedy nie je potrebné medzi volaniami jednotlivých metód uchovávať stav zmenený predchádzajúcim volaním inej z metód. Biznis proces je realizovaný volaním jedinej metódy.

Stavová implementácia – používa sa pri tzv. konverzačných biznis procesov, ktoré sú charakterizované volaním viacerých jemnozrnných metód a takouto kooperáciou vytvára biznis proces. Stavy medzi volaniami metód sú uchovávané vo vnútri triedy `AbstractBiznisSlužieb`.

Príklad

```
// metóda zaobalujúca workflow konkrétneho biznis procesu
public void assignResourceToProject(int numHours)
throws PSAException
{
  try {
    if ((projectEntity == null) || (resourceEntity == null))
      {
        // facada nie je pripojena k jednotlivým entitám
        throw new PSAException(...);
      }

    ResourceTO resourceTO = resourceEntity.getResourceData();
    ProjectTO projectTO = projectEntity.getProjectData();
    projectEntity.addResource(resourceTO);
    // vytvorenie novej väzby pre projekt
    CommitmentTO commitment = new CommitmentTO( ...);
    // pridanie vazby do projektovej entity
    projectEntity.addCommitment(commitment);
  }
  catch(...)
  {
    // ošetrenie výnimiek
  }
}

// proxy metoda pre ziskanie PrenositelObjekt-u
public ProjectCTO getProjectDetailsData()
throws PSAException
{
  try {
    ProjectTOAHome projectTOAHome = (ProjectTOAHome)
    ServiceLocator.getInstance().getHome("ProjectTOA",
      ProjectTOAHome.class);
    // Transfer Object Assembler session bean
```

```
    ProjectTOA projectTOA = projectTOAHome.create(...);
    return projectTOA.getData(...);
  }
  catch (...)
  {
    // osetrenie vynimiek
  }
}
```

Príklad 4-3. Ukážka implementácie AbstraktBiznisSlužieb bezstavových metód.

Príbuzné vzory

V niektorých prípadoch sa vzor používa ako Zástupca resp. Fasáda.

4.4 Prenositeľ

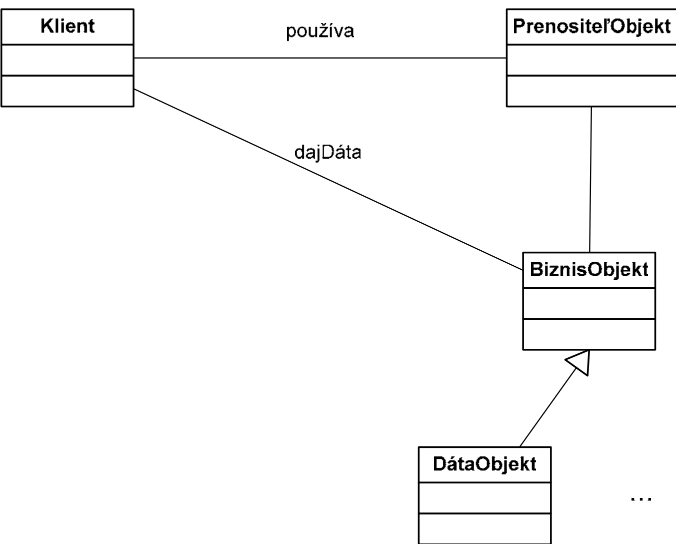
Podľa viacerých štatistík aplikácie typu informačných systémov niekoľkonásobne viackrát čítajú údaje ako ich modifikujú alebo zapisujú. Súčasná architektúra JVM a viacvrstvových aplikácií vyžaduje pri každom sprístupnení hodnoty klientovi prechod cez všetky vrstvy aplikácie, siete i JVM (aj v prípade že všetky súčasti aplikácie bežia na rovnakom stroji). Takéto volania vzdialených metód spôsobujú oneskorenie reakcií aplikácie.

Často je výhodnejšie získať všetky potrebné hodnoty volaním jedinej metódy a až u klienta pracovať s konkrétnymi hodnotami.

Účel

Návrhový vzor Prenositeľ, v angličtine známy ako *Transfer Object*, rieši situáciu, kedy je potrebné časté čítanie (rovnakých) údajov bez potreby opakovaného volania vzdialených metód. Vzor umožňuje prenos súvisiacich údajov vo forme jediného objektu (s odkazmi na hodnoty jednotlivých atribútov) ako výsledku volania jedinej metódy.

Štruktúra



Obrázok 4-4. Vzťahy medzi triedami vo vzore Prenositeľ.

Súčasti

`Klient` môže predstavovať celú klientsku aplikáciu (resp. aplikačnú logiku), ktorá vyžaduje čítať resp. modifikovať údaje. `Klient` žiada `BiznisObjekt` o potrebné údaje.

`BiznisObjekt` predstavuje poskytovateľa údajov (resp. prístupový bod pre poskytovanie údajov) klientskym aplikáciám. `BiznisObjekt` na požiadanie vytvára `PrenositeľObjekt`, ktorý vracia objektu `Klient` naplnený konkrétnymi hodnotami.

`PrenositeľObjekt` je vytváraný objektom `BiznisObjekt` na požiadanie klienta. `PrenositeľObjekt` je obrazom skutočných dát.

`DátaObjekt` je naplnený konkrétnymi údajmi, napr. využitím vzoru Dáta pristupujúci objekt.

Dôsledky

- Väčšie množstvo prenesených dát na menší počet volaní vzdialených metód,
- redukuje sieťové zaťaženia v prípade potreby čítania viacerých údajov z jedného `PrenositeľObjekt`, no na druhej strane v prípade potreby jedinej hodnoty sa vždy prenáša celý `PrenositeľObjekt,`
- vybrané z implementácií tohto vzoru redukujú opakovanie zdrojového kódu,
- v prípade možnosti modifikácie `PrenositeľObjekt` je potrebné v `BiznisObjekt` implementovať metódu zisťovania zmenených atribútov,
- v prípade možnosti modifikácie `PrenositeľObjekt` je potrebné riešiť viacnásobný prístup, transakcie a vzájomné vylučovanie.

Implementácia

PrenositeľObjekt môže byť implementovaný cez verejne prístupné atribúty alebo v prípade potreby obmedzenia prístupu k jednotlivým atribútom použijú sa súkromné atribúty a metódy pre sprístupnenie resp. nastavenie hodnôt jednotlivých atribútov. Najčastejšie používané prístupy k realizácií tohto vzoru sú:

- PrenositeľObjekt s možnosťou modifikácie dát – pri tomto prístupe musí byť implementovaná porovnávacia metóda pre identifikáciu zmenených atribútov, taktiež musí byť implementovaný mechanizmus pre riešenie konfliktov pri n klientoch.
- Viacnásobný PrenositeľObjekt – najčastejšie sa používa tento vzor v kombinácií s Dáta pristupujúcim objektom pri zachovaní pomeru DátaObjekt : BiznisObjekt : PrenositeľObjekt 1:1:1. V špeciálnych prípadoch sa dá použiť modifikácia tohto prístupu, kedy BiznisObjekt podľa volanej metódy (resp. podľa vstupných parametrov) vracia jeden z *n* PrenositeľObjekt.
- PrenositeľObjekt rodičom BiznisObjekt – pri tomto prístupe BiznisObjekt dedí od PrenositeľObjekt všetky get/set metódy, čím redukuje zbytočne sa opakujúci zdrojový kód v oboch triedach.

Príklad

```
// PrenositelObjekt pre prenos udajov spojenych s Projektom
public class ProjectTO implements java.io.Serializable {
  public String projectId;
  public String projectName;
  public String managerId;
  public String customerId;
  public Date startDate;
  public Date endDate;
  public boolean started;
  public boolean completed;
  public boolean accepted;
  public Date acceptedDate;
  public String projectDescription;
  public String projectStatus;
}

public class ProjectBO implements EntityBean
{
  ...
  // metóda pre vytvorenie PrenositelObjekt-u a
  // skopírovanie potrebných dát
  private ProjectTO createProjectTO() {
    ProjectTO proj = new ProjectTO();
    proj.projectId = projectId;
    proj.projectName = projectName;
    proj.managerId = managerId;
    proj.startDate = startDate;
    proj.endDate = endDate;
    proj.customerId = customerId;
    proj.projectDescription = projectDescription;
    proj.projectStatus = projectStatus;
```

```
      proj.started = started;
      proj.completed = completed;
      proj.accepted = accepted;
      proj.closed = closed;
      return proj;
    }
    ...
}
```

Príklad 4-4. Ukážka implementácie PrenositeľObjekt cez verejné atribúty.

Príbuzné vzory

Často sa používa v spojení so vzorom Dáta pristupujúci objekt.

DIEL II:
VYBRANÉ TÉMY PROGRAMOVÝCH A INFORMAČNÝCH SYSTÉMOV

5 PATTERN-BASED SOFTWARE DEVELOPMENT

For a long time, computer scientists were looking for a silver bullet which would solve the problem of reuse of existing knowledge to the benefit of future software with aim to build the more quickly, efficiently and reducing inherent risks. Moreover, software engineering, as one of the engineering disciplines, is displaying the reuse as one of the banners in the battle against software complexity as source of major risks in software development in order to achieve better time to market and improved quality products.

Object-oriented analysis and design apparently made us to prevail on many fronts fighting the software complexity. However there are several misunderstandings linked to the use of object-oriented approaches. The concept of a class as such is only marginally useful for the purpose of reuse and was initially invented to support encapsulation and logical decomposition of a solution. The reason is that a class itself is very rarely able to do something useful, merely there are whole conglomerates of classes bound with various types of relations which are able to solve specific problem. In order to make reuse of such clusters possible, the solution must be abstracted to allow creation of parametrized of instances -- the components. Components feature clearly carved interface shielding the internal working from the rest of the software system and vice versa. Then, proper specification and documentation of an interface is the way how to make the benefits of the component available to others. However not all knowledge can be abstracted and extracted in this manner. Certain solutions cannot be confined to single component; these may consist of collaborating components specific to application domain[4], while the way they collaborate may be independent and usable in variation of its context. Such solutions may be abstracted in the form of pattern.

In order to allow the pattern to be reapplied, one must capture its characteristics abstracted from concrete context. Then originating from such abstract-context description it is possible to specialize the pattern again to solve the task in another concrete context. The task of abstracting is intellectual challenge for few human designers who are able to highlight viable and successful solutions for the ones who come after. However selecting and applying already described pattern may be also

[4] A set of technical and business constructs that provides immediate context for software development.

difficult and error-prone, not to mention the process of transforming objectionable solution into satisfactory solution. While most patterns are described in expressive and well human-readable form, the descriptions are also imprecise and ambiguous at some occasions. Misunderstandings and unwanted shortcuts during applying patterns also cannot be ruled out as this is also creative process.

To eliminate these risks it is necessary to bring the use languages of patterns to the same level of understanding and ease of use as working with classes and components. This process includes extending existing CASE tools and development environments to embrace the development with patterns just as development at lower conceptual levels. Such progress depends on devising extensible and expressive pattern description system to effectively capture the knowledge about patterns and allow reapplying it.

5.1 Pattern Systems and Languages

It is well known truth that for software developers it is beneficial to think in high level abstraction instead of low level programming terms to achieve reasonably efficient communication and well understanding of a problem being solved (Schmidt, 1995). Many solutions need to be communicated at the different level than the level of specification of component interfaces.

For example, we need to describe generalized solutions to the problem of representing trees as part-whole hierarchies. This problem cannot be flexibly solved by plugging in any component. We need to describe how the components are joined together and how they collaborate to accomplish the task. This encompasses the cases when there is need to reuse the idea of how to accomplish task using component collaboration.

Imagine the benefits of having captured such knowledge in reusable form and the advantage of being able to reapply it anytime you need to represent data and accompanied operations in a tree-like structure.

This is where the concept of a pattern comes handy. The pattern can be defined as abstracted and generalized solution to the recurring problem, which can be applied over and over to similar problems. In other words, it can be characterized as a reuse and conceptualization of "a glue" that holds components together and form the essence of a solution. The advantage of reuse of this "glue" over reuse of components within application design has been recognized and reported in Brown et al (1998), Chapter 2 on design reuse.

As this work is not intended to question and argue over benefits of the pattern conept, we will just summarize benefits of this mode of reuse (adapted from Shalloway et al, 2001, Chapter 5):

- Reuse successful solutions. By reusing successful solutions, one can prevent negative surprises and avoid "reinventing the wheel".
- Establish common terminology. Communication and teamwork require a common base of vocabulary and a common viewpoint of the problem. Design patterns provide a common point of reference during the analysis and design phase of a project.
- Manage complexity. Patterns allow establishing higher level view of the problem and/or solution. This helps to avoid dealing with details too early in development.

During recent decades large number of such solutions have been identified and eventually described. Consequently whole pattern languages, in Alexandrian[5] sense, were formed. The pattern language can be viewed as an alphabet which enables to express and describe solutions using higher level concepts than common software design or implementation level (e.g. classes). However, it is not always reasonable to construct software only from patterns, because such pattern language would need to introduce very fine grained letters to couple low level design concepts to make expressing application specific details (Schmidt, 1995).

Many catalogs of patterns at a different scope and view emerged from which the most remarkable are the following: Gamma et al. (1995), Shu et al. (1999), Schmidt et al. (2000), Eventhelix (2005), Cunningham (1996); Fowler (1996), Marinescu (2002), Douglass (2002), Alur et al. (2003), Brown et al. (1998); Fowler et al. (2002b). In addition to pattern catalogs there have been written considerable amount of literature on the issues connected to the construction and application of the patterns.

Because the intricacies consideration and reasoning of pattern usage is out of scope of this document, we will omit further details in this chapter and go on to describing design pattern lifecycle, selected kinds of pattern systems and their application to improve understanding of the main topic of design pattern characterization before continuing to following chapters.

5.1.1 Pattern Lifecycle

The lifecycle of a pattern is show in Figure 5-1. Three main activities may be observed during this lifecycle. Following paragraphs characterize each of the stages.

Discovery

Patterns originate from experience; they are discovered rather than invented (Fowler, 1996). An "author of design pattern" actually doesn't "create" the pattern himself, but merely recognizes common properties of pattern instances usually contained in several successful software systems.

As the idea behind each pattern is highly abstract and general there may be considerable variance in what is considered to be a instance of the same pattern. There are exceptions from the above rule as the software frameworks may or may not stem from existing experience, but most of the time they do. More on the issue of software frameworks may be found in Section 5.1.2.

The result of the discovery of a pattern is abstracted in the form of pattern template as described by Marko, 2004a. In order to specify a template, it is necessary to identify and name following entities:

- Participants.Which members are working together to perform goals of the patterns. E.g. these may be classes, components, and etc.
- Collaboration. How do the members achieve goals? What is the nature of their cooperations, communication, flow of control and data, and/or structure?

[5] Named after Christopher Alexander, an architect who pioneered the area by describing pattern language for planning towns and construction of buildings within the towns.

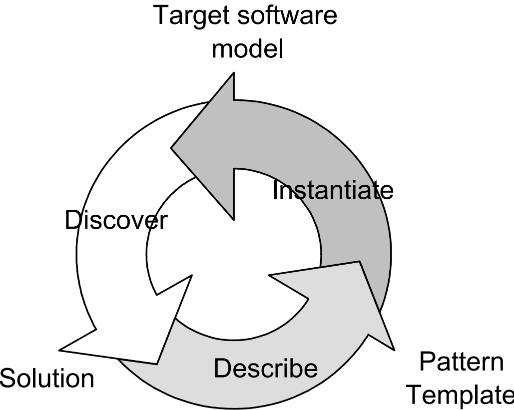


Figure 5-1. Pattern lifecycle.

Description

For the discovered solutions of general problem to be usable and reusable, these have to be abstracted, generalized and described. While abstraction and generalization are highly creative activities, which heavily depend on the nature of a pattern, description is meant as a characterization of pattern properties in a structured way to allow reapplying captured knowledge.

During the process the author may define additional constraints or degrees of freedom. For example, she may constrain number of occurrences of particular participant to a sensible range.

A *pattern template* is a result of the description step. The template may be of various forms, ranging from narrative description to machine readable model. Figure 5-2 shows the relation of discovery and description activities. It is obvious that initial identification of solution and early description resulting in (human readable) pattern catalog entry (carried out by Author A) may be further refined into another characterization, which may be more precise in some sense (Author B).

Instantiation

While prior steps were more or less one-time activities, the instantiation may occur over and over. The input of instantiation process is a pattern template and the result is its *instance* adapted to solve concrete occurrence of problem in concrete context. It may take form of concrete participants which carry responsibilities induced from their membership within the pattern and from their application specific task. More on weaving pattern-specific and application-specific responsibilities can be found in Section 5.2.

The result of instantiation – the pattern enriched software model can be consequently the source of patterns discovered in the future, thus closing the loop of the lifecycle.

Yacoub et al. (2004) offers another view of pattern lifecycle. The approach in the book focus on gathering and polishing of knowledge, while the above cycle focus on basic activities and artifacts during the whole cycle. However the cycles can be mapped to each other.

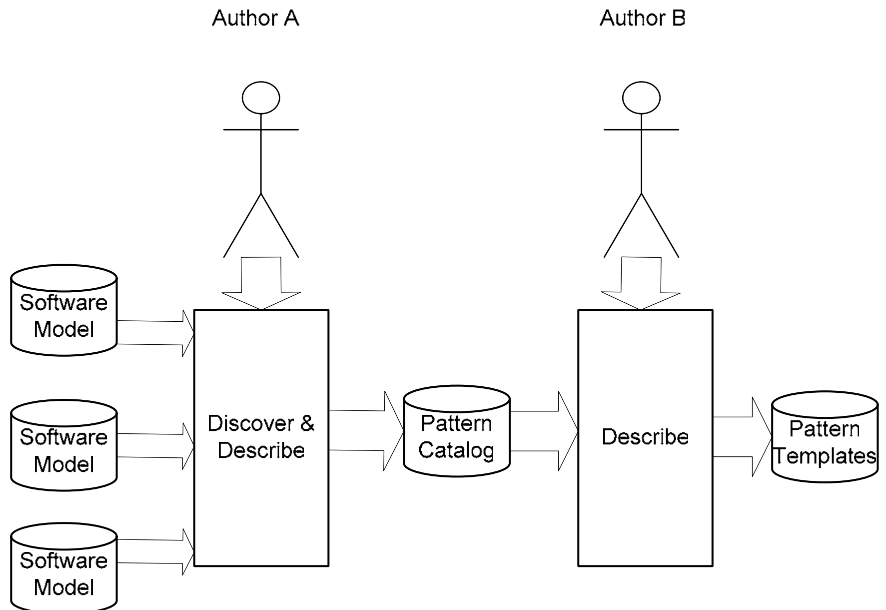


Figure 5-2. Discovery and description of a pattern.

It consists of three steps:

- Mining. Experienced authors discover and document the pattern. Results in preliminary documentation. Maps to Discovery.
- Polishing. Evaluating and improving the pattern by experienced practitioners and researchers. Results in reusable version of the pattern. Maps to Discovery/Description.
- Reuse. Pattern users look for pattern in the pattern catalog and instantiate it in their practical applications. Maps to Instantiation.

5.1.2 Characteristics of Selected Pattern Systems

A single pattern's aim is to solve a distinct problem. However, there are known several ways of combining of patterns. We consider the most prominent of them are described by Völter et al., 2002:

- Composite patterns. The patterns assembled from smaller parts, which are again known patterns. It is important, that the new pattern solves distinct problem and not just the combination of problems solved by its component patterns.
- A family of patterns. A set of solutions that solve the same general problem. Each pattern solves the problem specifically or resolves different set of forces.
- A collection or system of patterns. Consists of several pattern solving problems from the same domain or area.
- A pattern language. The most complex form of combining patterns. There is a language-wide goal and the purpose of the language is to guide the user to its goal.

Another important feature is the generative nature of the language, which in turn requires defining the context for each character (a pattern) in which it may occur.

It is clear, that the above categories are in no way disjunctive. Most of the time we refer to the term pattern system because we are most often interested in a set of pattern at the similar level in terms of problem and/or solution domain, because such groups tend to have similar properties and characterization.

The common character of some pattern systems is schematically displayed in Figure 5-3. Frameworks and architectural styles are overlapping because framework often implement particular architectural style (see Section 5.1.1). Most of the frameworks are built from design patterns.

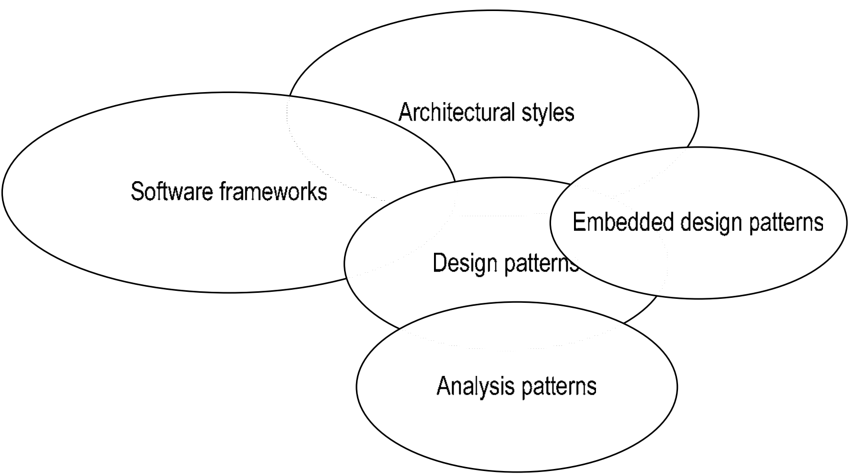


Figure 5-3. Pattern systems.

A linkage between patterns is crucial for a set of pattern to become a pattern system. Even if each pattern of a system may be useful in isolation, the whole language is even more important because it integrates solutions in particular problem areas (Schmidt, 2000).

The notions of pattern systems and pattern languages differ mostly in the fact, that systems not necessarily carry the generative feature and therefore there are additional constraints employed when constructing patterns of particular language. These constraints are described in deeper detail in Völter et al (2002) and Salingaros (2000) where can be the similar meaning of pattern within domain of (building) architecture and computing also examined.

To establish a link to the concepts defined in previous section, it can be shown that pattern systems are differing in the type of participants (Table 5-1). Most of the kinds of participants can be modeled using the most abstract ones (classes), but at times it is beneficial to consider their real kind, especially if the code or configuration (i.e. generating of Enterprise Java Beans and J2EE deployment descriptors) generation is involved.

All of the pattern system and languages mentioned in this section have connections to software development. However the idea of pattern language, originating from the domain of architecture, has spread virtually into every area of knowledge.

Pattern system	Participants
Design patterns (Gamma et al., 1995)	Classes, Objects
Analysis patterns (Fowler et al., 1996)	Data entities
Embedded design patterns (Eventhelix, 2005)	Modules, Tasks, Classes, Source modules
Server Component Patterns (Völter, 2002)	Server-side components
Design patterns (Marinescu, 2002)	Enterprise Beans, Classes

Table 5-1. Participants and collaborations of pattern systems.

Architectural Styles

Architectural styles can be viewed as a system of patterns which suggest the preferred way of building successful software architecture considering particular context and forces. As there is no consensus on what software architecture is, we will have to define this notion. One of the most prominent definitions denote a set of system structures of which each describes the system from certain viewpoint (Bass et al., 2003). It can be added to this paraphrase, that we have in mind, the structures that landmark the blueprint of a system and the most fundamental working of the system. In other words, results of most of the hard to change decisions are reflected in the architecture to capture what is unlikely to change and should accommodate the possible change of other parts. Therefore employing architectural style requires strategic thinking and experience in order to avoid negative surprises later.

Having the notion of software architecture defined, we can proceed to the concept of an architectural pattern is which is according to Bass et al. (2003) determined by:

- A set of element types (such as a data repository or a component that computes a mathematical function).
- A topological layout of the elements indicating their interrelationships.
- A set of semantic constraints (e.g., filters in a pipe-and-filter style are pure data transducers -- they incrementally transform their input stream into an output stream, but do not control either upstream or downstream elements).
- A set of interaction mechanisms (e.g., subroutine call, event-subscriber, blackboard) that determine how the elements coordinate through the allowed topology.

Architectural style reflects the above definition and often forms the overall concept of the system being developed as a grand motif. There are cases, when multiple styles are stacked on each other to provide synergetic effect. The experience from using Apache Cocoon[6] publishing framework for Web application development (Bielik et al., 2004)

[6] The Apache Cocoon Project, `http://cocoon.apache.org/`

shows that such composition is a benefit in terms of clarity of architecture and allowing per partes approach to meeting design requirements. Apache Cocoon framework provides filters-and-pipes architectural style allowing processing of data to be published in consecutive steps determined by pipeline configuration. The Apache Avalon[7] component framework provides underlying flexible component service by employing Dependency injection pattern (Fowler, 2004) – a variant or supplement of Inversion of control pattern which forms the basis of most object-oriented frameworks. The basic idea behind Dependency injection pattern is to decouple component implementation and their communication configuration.

Software Frameworks

Object and component frameworks represent a kind of object-oriented systems -- a piece of software which is prepared to be extended. It is intentionally designed to capture shared knowledge from certain domain and to allow its extension (specialization) to accommodate the needs of anticipated applications. The framework implements general processing and provides the points which may be extended with applications specific components. The extensible points are often referred as hotspots. Most of the frameworks are composed from one or more patterns of smaller scope, each solving a partial problem or providing supporting mechanism. Using patterns to construct a framework allows better understanding of how the framework functions and interacts with its context as well.

The object oriented software framework design typically imposes inversed control flow. There is a contrast to conventional software architecture, where the main application constitutes application specific aspects, while reusable library modules are being invoked by the main application (Figure 5-4a). The inversion of control causes, that the reusable framework manages the control flow, while application specific modules are being invoked (Figure 5-4b).

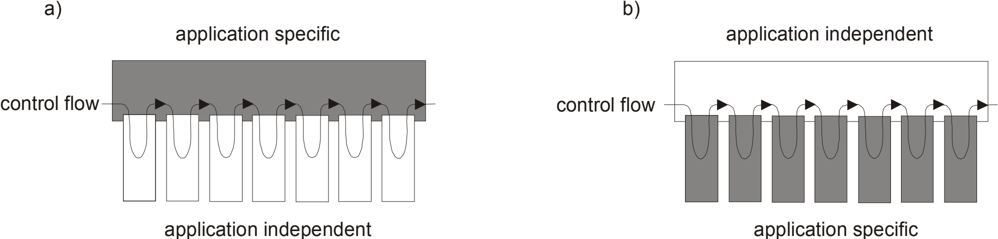


Figure 5-4. Inversion of Control framework.

Using framework architectures allows better separation of concerns and configuration management and even enables the dynamic reconfiguration. Frameworks often realize architectural styles in a domain specific manner.

Design Patterns

This pattern system is the topic of one of the now classic monograph, Gamma et al. (1995). The concept of design pattern is widely popularized and gained considerable

[7] Avalon/Excalibur, Inversion of control container, `http://excalibur.apache.org/`

popularity among researchers and developer community because of its benefits for developing software system using object-oriented paradigm.

Patterns of this system represent reusable bits of design, basically focused on the level of static structure. Gamma, et al., has suggested three categories of design patterns according to purpose (with typical members in parentheses):

- Creational. Concern object creation (Abstract Factory, Builder).
- Structural. Represent composition of objects and classes (Adapter, Composite).
- Behavioral. Deal with object creation (Observer, Command)

Most of the design patterns, even ones devised later, can be categorized in the mentioned way. This organization is rather rough, but gives initial clue of the purpose of each pattern.

This view is supported by the presentation in most catalogs, which relies on UML class models of sample pattern instances. Design patterns foster tactical thinking, to use the pattern at will, where relevant problem is to be solved.

Analysis Patterns

Analysis patterns (Fowler, 1996) are groups of concepts that represent a common construction structures in business modeling. It may be relevant to only one domain, or it may span multiple domains. Analysis patterns are usually more complex (in terms of number of distinguished participants) that patterns of systems mentioned so far. The complex nature of the patterns somewhat limit reusability, to compensate for this downside, it is possible to provide management of features of the pattern to eliminate unwanted concepts from it (Filkorn et al., 2005).

Important fact about analysis pattern is that while design patterns or architectural styles use vocabulary of the solution domain, analysis patterns use the problem domain vocabulary. Therefore analysis patterns are often employed when designing domain models for information systems and catalogs are also constructed with that in mind.

5.1.3 Representing pattern through its lifecycle

The relation of pattern system to the pattern catalog in general is a one-to-one relation. For purpose of this section, the term pattern catalog refers to the human readable catalog. However in broader terms, any description of a pattern system might be regarded as a catalog. The description of a pattern in a pattern catalog can be taken as an initialization for characterization of the pattern. Most of the catalogs employ similar structure describing a context, a problem and a solution. The description is intended to be used to educate a developer or to serve as a point of reference for solving development issues. We refer to such a description as a *characterization*. It should reflect flexibility, generality and abstractness of a pattern (Smolarova et al., 2000).

On the other hand, such presentation and content is not directly usable for machine processing, it is far from becoming a model of the pattern. The principal issue is the lack of generative nature of such characterization in a sense, that a model is a generator of potential configurations of systems; the possible systems are its extent, or values (OMG, 2003a).

We have to distinguish between pattern template and pattern instance, as well as between pattern instance and pattern instance runtime representation. Figure 5-5 depicts the relationships between mentioned conceptual levels.

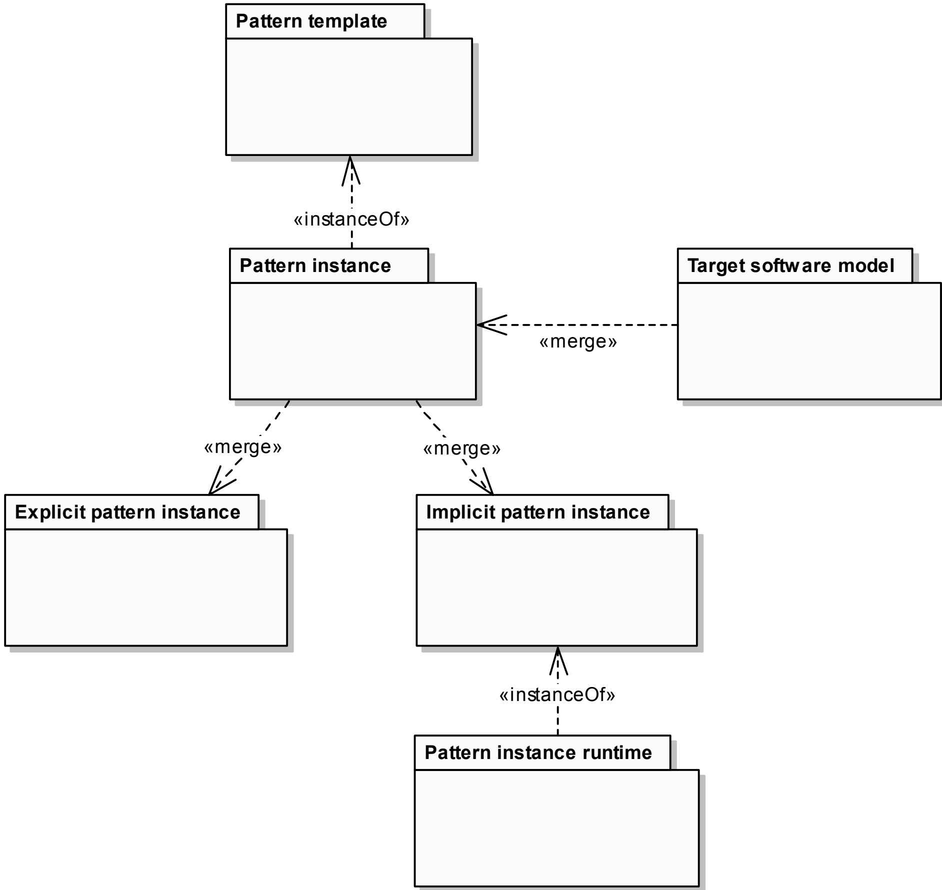


Figure 5-5. Representation of the pattern at the stages of its lifecycle.

If we take for example the design patterns as an example, the mapping to the features of a pattern belonging to this system would be as follows:

- Pattern template – descriptions of abstract participants.
- Pattern instances – concrete classes, operations, attributes participating in a pattern and pattern-induced relations (implicit instance) and additional descriptive data about an instance (explicit instance)
- Pattern instance runtime – concrete instantiated objects carrying responsibilities resulting from their participation in the pattern instance and pattern-induced run-time relations

Target software model, which represents a software model to be enriched by pattern instances merges Pattern instances package (semantics of PackageMerge relationship

of the UML Package model can be found in OMG (2003b). Instantiations are denoted by *instanceOf* relationships.

Each pattern features distinct properties which define its form through its lifecycle. These properties can be divided into two main categories:

- Static properties. Describe the leitmotif of a design pattern. Which elements it consists of, how they are composed and what responsibilities are they supposed to carry.
- Dynamic properties. Describes when and how a pattern may be used. Relations and collaborations among patterns and/or pattern instances. May include instructions for its suggested applications.

Finding proper set of properties to describe the pattern for particular application may lead to articulation of a model for describing pattern, which enables to make the pattern both better human understandable and able to be machine-processed. Therefore we have included an overview of multiple methods for capturing static and dynamic properties of patterns. Eventually descriptions of certain properties may be grouped to form views which describe a pattern taking specific concern into account.

Software architecture community within computing science and software engineering came close to consensus on the fact that software architecture modeling requires multiple concurrent viewpoints. While by far not only approach, this achievement reflected into construction of software processes that promote structured approach to modeling software using agreed on viewpoints. We can take so called *4+1 model* (Kruchten, 1995) as an example of such approach.

Such view based approach is a key one in the UML: It provides the basis for the UML notion of a model. However, it is not a formal construct in the UML. Instead, it acts as an informal concept that helps situate the role and responsibilities of the modeler. In the UML, a model is a complete representation of a system from a particular viewpoint (that is, an aggregation of a set of views from specific perspectives). At the same time, systems are logically composed of many nearly independent models, representing many different viewpoints; and they can be physically composed of many independent subsystems, each of which can also be treated as a system for modeling purposes (Evitts, 2000).

However the viewpoints are not selected blindly, they rather reflect major concerns of architecting software and distribution of responsibilities among members of development teams. An artifact earned by viewing the model from certain viewpoint is called a *view*. Such concept, when used properly, allows modularizing knowledge captured by a model to improve manageability of a development process. Separation of concerns and distribution of responsibilities should be taken into account also when designing views to a pattern model.

5.1.4 Applications

It is obvious, that the selection of views required for modeling of a pattern depends on intended application, i.e. the views required for instantiation will include the dynamics of the pattern, while the view intended for pattern recognition might not include the dynamics. Under the term *application* we understand the usage of a pattern description to the benefit of engineering a software system and reuse of knowledge.

Pattern Instantiation

The instantiation is a process of specializing pattern into the form adapted to solve concrete problem in concrete context. It should be regarded as a process, input of which is consisting of pattern templates (Figure 5-6) and as a result there are explicit and implicit information about pattern instance embedded into *target model* (referring to the model of the software which is being developed using patterns, be it a design model or actual implementation according to Marko, 2004b). The process usually requires assistance of human developer.

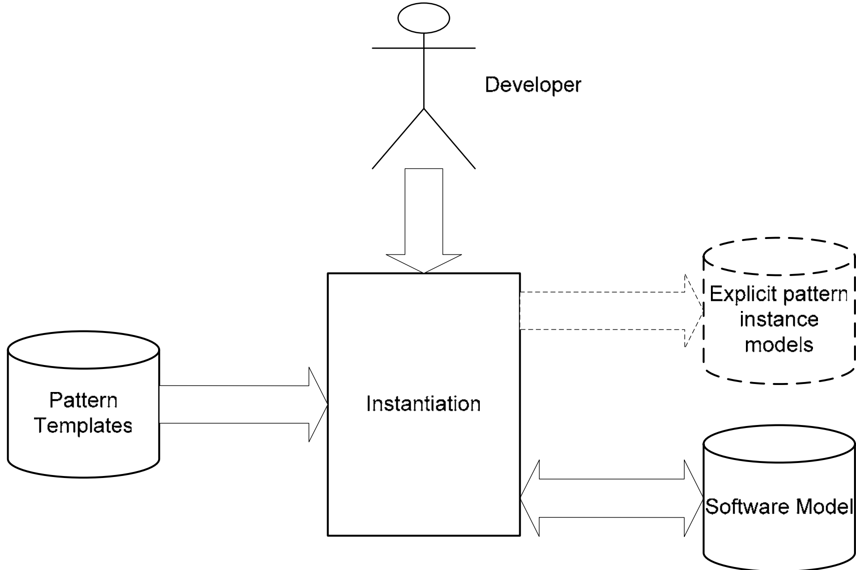


Figure 5-6. Pattern instantiation conceptual schema.

CASE tool support for instantiation is considered to be the prime application of most approaches referred in this work. The instantiation process is a complex activity which in general cannot be performed in one step because besides the pattern template itself, also the target model needs adaptation to merge successfully with the pattern. The modifications have to be performed in different places within the model, however the tool may assist to developer to make correct decisions or even perform some of the adaptations automatically. One of the approaches is to design a process which features 3 basic properties (Hautamaki, 2002; Marko, 2004b):

- Incremental. Desired instance may be too complex to make impossible the consideration all facets of the problem at once as well as to adapt it and the target model to successfully merge. Moreover, the requirement of incremental process is implied by the need to evolve the pattern instance as developer proceeds with the work and/or accepts new requirements.
- Iterative. Most activities of software engineering are iterative, thus it is natural to go back to improve design, to fix mistakes or to solve several problems simultaneously.
- Interactive. The interactivity involves multiple interactions with designer during formation of pattern instance. Moreover, the process should also be able to suggest possible steps of instantiation.

The prime issues linked to this application include supporting the developer in making correct choices of the pattern (ranging to helping identify need for using the pattern), then to manage the process of instantiation to keep resulting instance consistent with the template and maintaining information of the instances in the target model.

Pattern Instance Visualization

Pattern instances, unlike components are dispersed inside the software in a sense, that different participants of particular instances may be part of completely different logical units and the instances may be even overlapping. The instance is consisting of implicit and explicit compound.

The basic problem is, how to make dispersed instance visible and make the developer aware of its existence. In UML, there are efforts to explicitly introduce a linkage between participants in form of pattern collaboration (represented using collaboration element of the language). Other solutions include automatic generation of exclusive subviews which denote selected single instance (Marko, 2004a). The prime problems for this application include brief yet expressive visual representation of an instance within target model.

The incremental instantiation is linked to the visualization in a sense, that it requires incremental updating of layout diagrams (i.e. user placed elements stay in place, newly generated elements are placed automatically in the appropriate location). This is especially true for implicit pattern instance diagrams (e.g. class model). Unfortunately most of current renowned CASE tools fall short of being able to satisfy the need for reasonable automatic layouting of design elements.

Pattern-based Documenting and Pattern Recovery

Even the brightest and most elegant frameworks won't be used unless they are adequately documented. There are estimates that a professional programmer needs 6 -- 9 months to become proficient with larger framework with exponential increase with the complexity of software. Patterns represent higher level concepts that help to understand complex structures because they are suitable to express a kind of collaboration. It is beneficial to embed this knowledge into the framework to establish pattern-driven O-O understanding (Booch94 et al, 1994).

Figure 5-7 shows schema of the pattern recovery process. There is a model of the software system on the input of the process, and it results in explicit information about pattern instances. The process may or may not involve the developer.

An approach described in Keller et al (1999) is based on combination of manual and semiautomatic tools supporting design pattern recovery. The recovery interface is complemented by visualization of the software design and highlighting pattern members already identified as participants with certain pattern instance.

Another approach (Zhang et al., 2004) focuses on extending partial mapping of elements or modules at different levels of abstraction (namely high level design level to implementation structure) to a more complete mapping. The aim of this approach is to accommodate mapping of the design to the implementation model, which may not necessarily be 1:1 mapping. This is achieved by solving a simultaneous set of equations constraints on the mapping imposed by high level model, while the solution represents a set of design entities to which an implementation entity may be mapped.

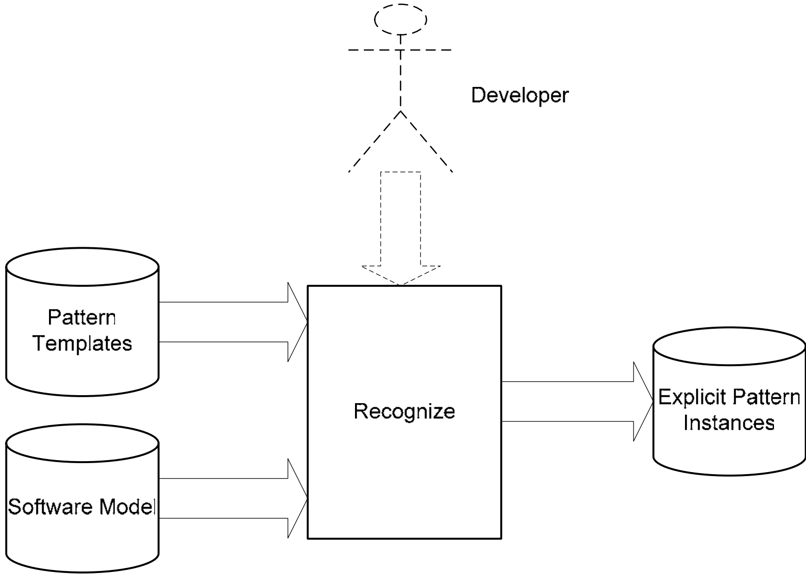


Figure 5-7. Pattern recognition conceptual schema.

Manual recovery of pattern within large system has proven to be infeasible. Therefore automated approaches have been developed. They may be based on matching of graph structures or using graph grammars to accompany generative nature of the pattern model. Fully automatic approaches are suitable only for limited set of patterns because of ambiguity of other patterns' nature and variance of possible instances, which are consequently difficult to detect. The issues often take form of tendency to misses and/or false hits.

Semiautomatic architecture recovery which uses not only static structure of the software in the form of source code written in high level programming language, but also the dynamic information about runtime performance is described in (Wu et al., 2004). Trace data are collected from the running program in order to discover dependencies and collaborations between program elements and modules.

Pattern recovery technique, according to Heuzeroth (2003), takes advantage of abstract syntax tree (a structure produced by common compilers). Then it is possible to look up possible candidates for patterns in the nodes of the tree in the form of tuples of pattern member suspects. However a set of the candidates is usually too large to be enough to compose recovered patterns. Therefore dynamic analysis is employed which includes tracing the effects of execution of candidates, the tracing evaluates constraints given by the type of relation among participants of a pattern (multiplicity relation). Candidates violating constraints are then removed from the set of candidates.

5.2 Static Properties of Design Patterns

Static properties describe a pattern from the static aspect. The pattern is viewed as a template, which defines the structure of its members, their relations and means, how the pattern instance can be mapped to the template. The static aspect should be viewed as the template being invariant across its instantiations and while it may be partly

parametrized for purpose of particular instantiation, changes of an already created instance are disregarded. In addition do the template, there is also the static form of representation of pattern instances.

The challenge of characterization of static properties lies in capturing the variant and invariant part of the pattern. Most of the time we will be using system of design patterns as a reference pattern system.

This chapter presents brief survey of advances in describing selected static properties.

5.2.1 Pattern Template

A pattern template represents an abstract and general view of a pattern. It has these principal functions:

- defines the participants in a pattern by constraining their types,
- constrains valid structure of the instance,
- defines views in which the pattern instance is modeled.

The commonly used method for definition of types of participants is role modeling. Valid structure is characterized by employing relationships among roles and the template is complemented by means of specifying the structure and behavior of future instances.

Role modeling

Role modeling is widely adopted (Kim et al., 2003; Marko, 2004, Hakala et al., 2000) technique to achieve separation of those concerns that are specific to pattern and those specific to its context. Role (based) modeling can be compared to class (based) modeling as show in Riehle et al. (1998). A role model is the description of a (possibly infinite) set of object collaborations using role types. It focuses on a single purpose of object collaboration: a role model does not try to encompass all possible aspects of a given object collaboration. In a role model, each role type specifies the behavior of one particular object with respect to the model's purpose.

A role model is, much like a traditional class diagram, a constraining specification for a set of valid runtime object collaborations. The difference between a role model based specification and a class-based specification stems from the modeling concepts involved. With role models, objects in an object collaboration, which conforms to a role model, are known to behave as specified by the role types of the roles they are playing. However, a single role type specifies only part of an object's overall behavior. In contrast, using class diagrams, objects in collaboration are known to be of particular classes. Such an assertion is more restrictive, because a class already combines several different collaboration aspects (role types) and does not distinguish the different purposes of the various collaboration arrangements.

Role modeling allows layering of pattern specific aspects within a software model being developed into overlapping planes tied to software model elements implementing responsibilities induced from participation in a pattern instance. Application specific and pattern specific concerns are interconnected by means of roles (Noda et al. 2001). The role in a design pattern can be characterized as a point where such responsibility

can be cast to actual model element (Hakala et al., 2000). Within scope of role modeling, the model element which has been cast will be refered to as a *participant*.

However, any number of roles may be mapped to single model element. An example of such casting depicts Figure 5-8. There can be seen horizontal a) and b) planes containing fragments of instances of Composite and Iterator patterns (Gamma, 1995) respectively. These planes contain roles of that particular pattern. The plane at the bottom, marked with c), is the plane containing concrete classes playing roles.

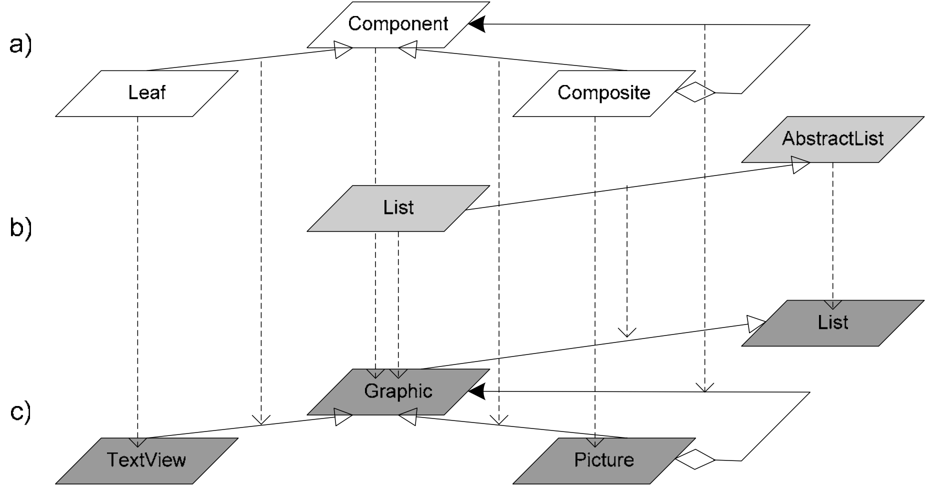


Figure 5-8. Mapping roles to model elements.

Role to model element bindings are depicted using vertical dashed lines. A role is always of specific *type*, which refers to type of an element which can be bound to that role. In this example we have limited our view only to classes as well as certain relations. Later in this section, we illustrate how this view can be extended to include also features of classes such as operations and attributes.

Pattern descriptions published in a pattern catalog often introduce other types of responsibilities in addition to the class-type ones. Frequent cases are operations or attributes that play a role within a design pattern. Therefore modeling approach should also consider those additional role types.

Another important purpose of a role concept is that a role forms the most basic constraint for casting model elements by not allowing casting different type of element as prescribed by role type.

Representing role systems using UML

The pattern system can be expressed using a language which allows specifying the pattern templates. However it must be taken into account, that there is also instancing aspect in working with patterns. It means, that pattern template has to be a description (or even a generator) of its instances. One of such languages which became industrial standards is Unified Modeling Language. It is possible to take advantage of its infrastructure to employ it to describe the new concept. However, the semantics has to be extended as well. The mechanisms, which facilitates extending of the UML is called metamodeling.

When dealing with meta-layers to define languages there are generally three model layers that always has to be taken into account:

- the language specification, or the metamodel,
- the user specification, or the model, and
- objects of the model.

This structure can be applied recursively many times so that we get a possibly infinite number of meta-layers; what is a metamodel in one case can be a model in another case, and this is what happens with UML and Meta Object Facility (MOF). UML is a language specification (metamodel) from which users can define their own models. Similarly, MOF is also a language specification (metamodel) from which users can define their own models. From the perspective of MOF, however, UML is viewed as a user (i.e., the members of the OMG that have developed the language) specification that is based on MOF as a language specification. In the four-layer metamodel hierarchy, MOF is commonly referred to as a meta-metamodel, even though strictly speaking it is a metamodel.

The basis of most of the UML based pattern description systems is a role concept. This concept, intuitively defined in previous sections, needs to be characterized in terms of UML. However, the notion of role in the object-oriented community is strictly based on objects, and does not allow the use of the word "role" in any other place where the context is not object-based. Characteristics of a general role concept have been defined in Kim et al. (2003) as follows:

- A role has structural and behavioral properties. Properties of a role may be inherited from other roles in the hierarchy. Instances that play a role can also play the superrole of the role.
- A role defines a subset of instances of a type.
- Access to a role is restricted by context.

Specializations of the general role inherit the above characteristics, but may also have their own characteristics. An object role is a specialization of the general role in that instances are objects (e.g., instances of classes). Object roles have additional characteristics that are runtime specific. Theoretically, the general role can be used to define roles at any level (e.g., metametamodel level).

Kim et al. (2003) defines a role in terms of UML infrastructure as it can be seen in Figure 5-9. It is obvious, that MyRole, defining constraints to the participants of type Class, represents *model* role of a pattern template, while RoleA represents object role in that model. Vertical bar symbol is used to distinguish a role. In general, roles may be derived from any metaclass (e.g. Association, Classifier, Operation, etc.).

An extension of this approach which hybridizes structural and behavioral description is proposed in Song et al. (2002).

UML profile for Enterprise Distributed Object Architecture (EDOC) specification also brings a way to describe patterns in context of UML. More on this topic can be found in Section 5.3.2.

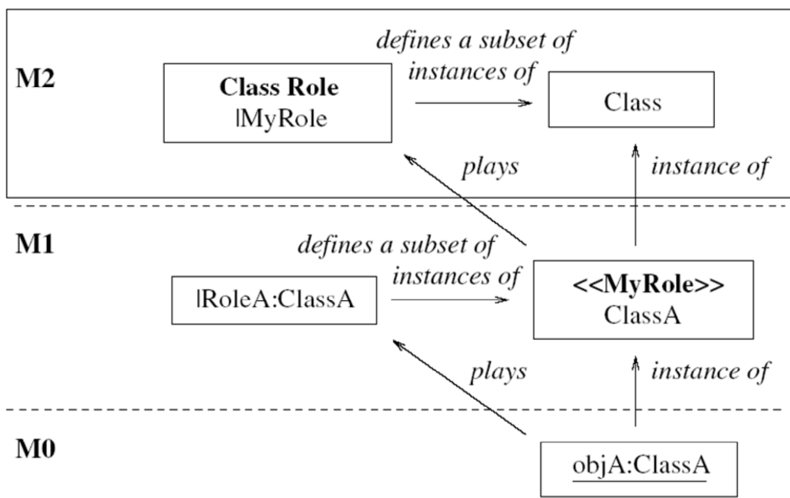


Figure 5-9. Relationship between model role and UML infrastructure.

Pattern structure and behavior

An instance of a pattern represents specialized form of a panel which has been obtained from pattern template through instantiation process. Many of known approaches mix representation of pattern template and its instance by treating them as whole. This is especially obvious, when the pattern is described using a static class model and there is a claim: "this is a pattern". Many examples of such stance can be found in pattern catalogs. In fact, usually there is an example of pattern instance's static class model provided. This information is a *structural* template for an instance.

But the pattern is not characterized exclusively by its static structure, also behavioral information is a part of characterization (for example: see description of Command pattern in Gamma et al., 1995 and note the interaction diagram (OMG, 2003a) in Collaborations section). Code snippets, interaction and statecharts are often the source of *behavioral* information. This compound of pattern description is often omitted despite it would be really useful if it was possible to instantiate also this behavioral information.

Pattern templates are then formed as structures of model roles. An example of Visitor pattern description according to Kim et al. (2003) using this system can be seen in Figure 5-10. Types of the roles are shown in bold inside each class element and role name is prefixed by vertical bar. Note that in the figure, there are also roles of generalization and association present.

Different methods of representing pattern structure include pattern definition graph as introduced by Hakala et al. (2000). The approach has been realized for supporting specialization of software frameworks using specialization patterns, but have been adapted for instantiating design patterns in Marko (2004a). The pattern structure is represented as a pattern graph, which defines dependencies between roles. However roles represented in such way are missing instance level semantics yet, therefore they need to be extended with additional constraints which describe actual program constructs such as class and method declarations (Hautamaki, 2002).

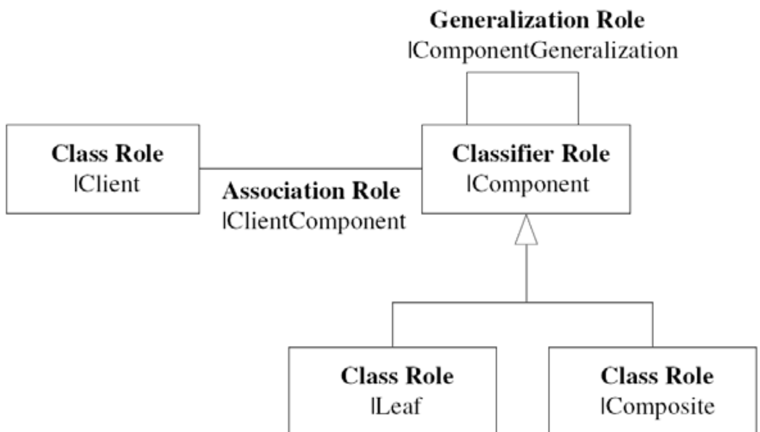


Figure 5-10. Composite pattern structure description (Kim et al., 2003).

Extension described in Marko (2004a) uses dual pattern template representation for describing its structure to separate different concerns. First of them is the the static structure of participants expressed as a class model. See an example of Prototype pattern definition in Figure 5-11a. This structure constrains type of participants and relationships among them, and is complemented by *role graph* Figure 5-11b) which constraints number of casts of each role.

The rationale behind separation of the two compounds is that they represent completely different concepts. One of them is dealing with pattern instance participants and the other is defining roles and their relationships. It allows avoiding duplication of model views which are already available. Even the prototype software tool which implements this method is created in a way that it reuses pattern structure models generated by standard CASE tools.

The template states, that the ConcretePrototype participant generalizes Prototype participant class and there is a Client that references Prototype. As generalization relation is transitive, any subclass of Prototype may be cast into role of ConcretePrototype. On the other hand, in Kim et al. (2003) one should declare explicit reflexive generalization relation in the pattern role model to achieve the same effect.

In reality relationships among roles may not necessarily one-to-one map to design model constructs. This is one of the crucial properties distinguishing pattern leitmotifs from generic class templates and frameworks. For example a relationship between two classifiers may be rendered as generalization or realization depending on the type of the participants in an instance. This property has been disregarded in two latter approaches. In Kim et al. (2003) the generalization/realization combination had to be modeled using two separate roles: of generalization and of realization, running in parallel. In Marko (2004a) this is partially facilitated by introducing the *compatibility matrix* which enabled or disabled certain roles to be cast with the same participant at once. This solves the situation for example of *Observer* (structure in Figure 5-12) pattern, when we do need only single Observer class. In such situation the compatibility matrix enabled to cast Observer and ConcreteObserver roles to the same participant.

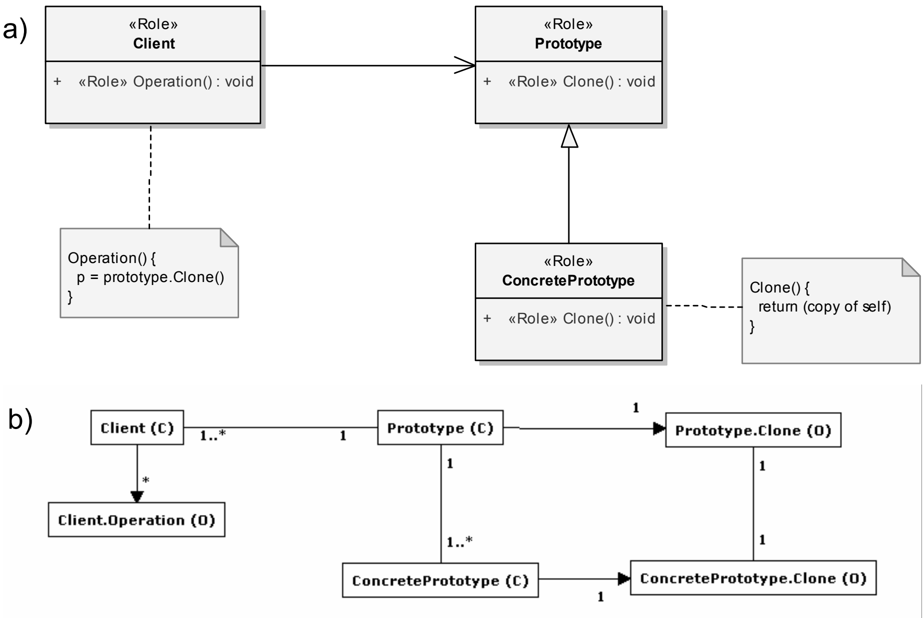


Figure 5-11. Pattern template of Prototype pattern (from Marko (2004a))

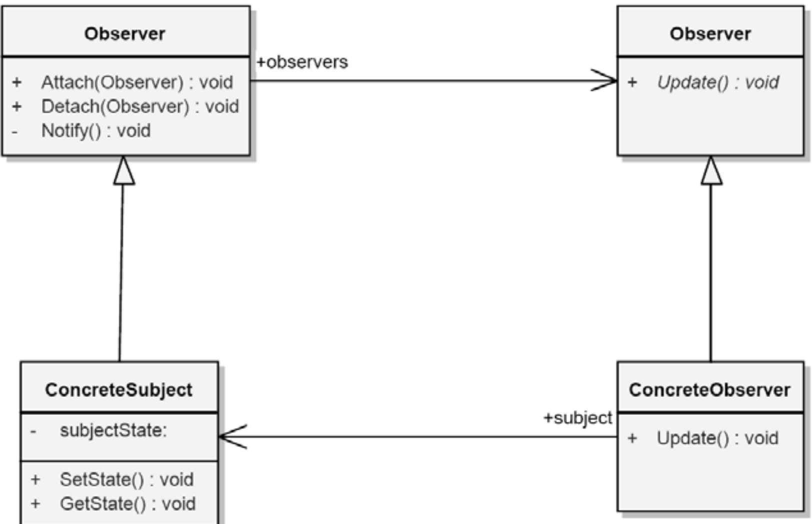


Figure 5-12. Pattern structure of Observer pattern.

Other approach to this problem is suggested in Mak et al. (2004) which proposes using abstract relationships among roles. These relationships denote nature of abstract relations among pattern members which can be eventually *reified* into concrete relations. This approach allows greater flexibility of resulting structures of pattern instances. proposes a set of abstract relationship along with corresponding reifications summarized in Table 5-2.

Abstract relationship	Description	Reifications
Realize	Defines that the source role (Class) provides implementation to the behavioral feature of the target role (Class/Interface).	Realization, Generalization or no explicit relationship
Implement	Defines that the source role (Method) implements the specification given by the target role (Operation).	Polymorphism or the implementation of operations
Invoke	Defines that the triggering of the source role (Operation/Method) will lead to the invocation of the target role (Operation/Method).	None

Table 5-2. Abstract relationships (Mak et al, 2004).

Participant multiplicity

We have intentionally avoided the question of role multiplicity in previous sections. Multiplicity of a role denotes how many times each role may be cast. It is defined as a specification of the range of allowable cardinalities that a set of participants may assume. Generally, we can categorize approach to two groups: one of them relying on absolute and the other relying on relative role multiplicity. Absolute multiplicity (Hautamäki, 2002; Kim et al, 2003) bounds number of casts in scope of an instance, while relative multiplicity (Marko, 2004a; Meijler et al., 1997) bounds number of casts of a given role in relation to the instance of other role. While essentially a multiplicity is a (possibly infinite) subset of the non-negative integers, in practice, only few of absolute multiplicity values are being used and are shown in Table 5-3 (Hautamäki, 2002).

Multiplicity	Description
1	Mandatory. Exactly one participant
0..1	Optional. Zero or one participant (optional participant)
*	Many. Any number of partcipant (including zero)
1..*	At least one participant

Table 5-3. Absolute multiplicities.

Absolute multiplicities are simple to implement in a sense that the algorithms that evaluate conformance of an instance to the template are very simple, because it is enough to count the instances of each type and compare counted value to the multiplicity range. However it is not possible to express some relationship using absolute multiplicity alone. Let's take a Adapter design pattern as an example (Gamma et al., 1995).

According to pattern description in a correct instance of the pattern, there must be:

1. exactly one Target,
2. any number of Adaptees, and
3. as many Adapters as Adaptees

The 1:1 relationship between Adaptees and Adapters cannot be expressed using absolute multiplicity, but can be described easily using relative multiplicity. Approach presented in Mapelsden (2004) does even further to solve problem of matching transitive multiplicity dependencies. Transitive dependency means that number of casts of a group of roles must be equal. It manages to express such dependencies employing so called *dimensions*. The same dimension can be associated with different participants in a pattern and this specifies not only that these participants can have some multiple numbers of objects associated with them but that number of objects is the same for both participants. Dimensions are often indicated by presence of dependencies is static class model of a pattern instance. Approach is very useful especially in conjunction with patterns like *Abstract Factory*, where there are multiple overlapping dimensions:

- Role CreateProductA()--Role AbstractProduct
- Role ConcreteFactory--Product

5.2.2 Pattern Instance

Pattern instance represents pattern in its specialized form for concrete context. As was mention earlier, pattern instance can be represented implicitly and/or explicitly. When talking about implicit instance, we are dealing with an instance described in a language of target software model. This information is useful for further processing of a model which is aware of only target model language. An example of implicit information would be specialized class structure embedded in a static class model of a developed system.

In contrary, explicit instance is information which resides outside of target model. In Marko (2004a) author describes a method of keeping information about grouping of pattern participants outside of target software model in a form of *instance graph*. Nodes of this graph represents participant cast into role. The benefit of this special compound lies in possibility to capture extra information in comparison with implicit description. In Marko (2004b), Hakala (2000) it is used to keep momentary multiplicity relations among pattern roles which were already cast with participants.

Conformance of an instance

A challenging issue of pattern instantiation is conformance of pattern instance to the pattern template. It is important for pattern recognition as well as it enables to select candidates for pattern instances from investigated software model.

Basically conformance is defined as a homomorphism of a pattern structure in software model. However additional constraints of role types must be taken into account, so that only conformant participant may be cast into particular role.

The way how conformance is evaluated greatly depends on how the template structure is defined. Conformance can be evaluated according to multiplicity of role and types of roles.

5.3 Dynamic Properties of Design Patterns

Dynamic properties describe how a pattern evolves and interacts. Because the software model representation changes during its lifecycle it is subject to the change at every stage. This document refers to the changes using the term evolution as a parallel to the evolution of living beings. The main argument justifying this metaphor is that the each pattern just as living being adapts gradually to its environment (i.e. abstract and general pattern adapts to the concrete solution in concrete context). By investigation of the nature of evolution of a pattern, three dimensions of evolution have been recognized.

Lifecycle dimension involving the software project lifecycle.

Ontogenetic dimension involving the evolution of an instance.

Phylogenetic dimension involving the evolution of a pattern template.

The names of the dimensions are borrowed to serve as metaphors from theory of evolution of living beings. Ontogenesis and phylogenesis denotes development of an individual and a species respectively. Such division helps to distinguish between development of pattern instance and the development of pattern itself. Lifecycle dimension refers to the lifecycle of software project. These dimensions are orthogonal and the pattern solution may move along any of the dimension. Other parallels to the evolution of living species can be seen, such as the fact that species that are phylogenetically specialized to the particular environment become vulnerable and unable to survive in an altered environment. In terms of patterns, overspecified pattern becomes too tied to given context and inapplicable to even slightly altered context. To elaborate on pattern evolution, each of the subsections of this chapter deals with one of the dimensions. Unlike Section 5.2, great part of the problems remarked in this section was not addressed in context of pattern characterization, yet. Therefore we add more ideas that need to be investigated in the future.

5.3.1 Ontogenetic Dimension

Ontogenetic dimension of pattern evolution describes the evolution of a pattern instance. There are two purposes of pattern instance evolution in ontogenetic dimension:

- Pattern instance evolves to meet the intent of the pattern in concrete context, or
- Unsatisfactory solution evolves into pattern-based solution.

To explain the former purpose, it is necessary to mention how the pattern is instantiated. One of possible approaches involves adding pattern structure to the target model all at once. The instantiation should be viewed as a process, not an atomic action. It evolves in time, however there are constraints. Some are implied by static structure of a pattern, but some should be added to affect the dynamics of the instantiation process or simply hint the developer (composition of patterns, unidirectional guides, annotations, hints etc.).

Pattern template driven evolution

One of the methods of specifying evolution of a pattern instance is to use multiplicity to generate potential future states of pattern instance. It is accomplished by generating *role instances* constrained by multiplicity. Each dependency of the components is

annotated by relative multiplicity. A pattern instance evolves as the instantiation proceeds in a sequence of steps. In each step, a developer may instantiate single role by casting existing element into it or letting the system to generate a conforming target model element.

Relative multiplicities are employed not only to control number of possible instances of particular role, but also to generate future refinements. The instantiation engine generates potential future pattern instance by taking an instance from prior step and adding all potential role instances, which are allowed by relative multiplicity constraint (see Figure 5-11). Engine is driven by dependencies of pattern template. For each instantiated role generates potential instances of roles which depend on it according to template. The benefit of this approach lies in the fact, that it ensures conformance to the pattern template after each step, thus make it impossible to create invalid instance (in violation of multiplicity constraints). The disadvantage is that the multiplicity itself is not enough to ensure that transitive relationships are conforming at all times. The potential improvement of this method could be by replacing relative multiplicities with dimension based multiplicities.

Generation of program elements based in a steps described by graph grammar is described in Hakala et al., 2001. In this case potential refinements of pattern instance are generating by applying graph grammar.

Developer guidance

It is important to guide the developer through the process of instantiation, to make her aware of possibilities of further refinement and to prevent mistakes by forbidding nonconforming instances. Both approaches mentioned in previous section generate a list of possible actions, with which they prompt the developer. This ensures that developer may create only instances that are conforming.

Model transformation

There are several approaches that are candidates to the mean of description of transformation of unsatisfying model to the conforming one. One of the widely used transformation technique is called *refactoring*.

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure (Fowler et al., 2002a). It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In general, refactoring is used to improve the code or design of the system in a limited scope without endangering rest of the system. Similarly to patterns, catalogs of refactorings (transformation that do not harm rest of the software system) has been created (Fowler et al., 2002a; Wake, 2003), each of them addressing single shortcoming.

Systematic approach to refactoring involves model transformations. It is applicable when there are clearly defined sets of source models, target model and transformation models. Source and Target model set are characterized by role models and represent the unwanted and wanted pattern-based solution respectively (Song et al., 2002). Transformation set consists of transformations, each of them accomplish transformation goal. Even transformations between semantically similar patterns can be accomplished using model transformations. The transformations can be described precisely and later applied automatically. Refactorings can be described also using

formal model (Tip et al., 2003). The example involves specifying *Extract interface* refactoring as knows from CASE tools.

Metamodeling method of applying transformations (Judson et al., 2003) adds additional metamodeling layer M2' to the M2 layer of UML metamodel which enables to metamodeling of transformations. The M1' level is an extension of the UML model level (M1) that supports representation of model transformations. A model transformation at the M1' level takes a source UML model and transforms it to a target UML model. This approach employ transformation schema which represents the structure of target model and transformation constraint which takes form of relationships which must hold between source model element and target model element.

Composition

Ontogenetic dimension defines pattern composition as composing of patterns in form of pattern instance. In other words this composition view describes whether roles of a pattern can be cast with certain participant simultaneously. Riehle et al. (1997) proposes following constraints to specify allowed role combination:

- An object playing role A also always plays role B in the same collaboration. Thus role A implies role B.
- An object playing role A never plays role B in the same collaboration. Thus, role A prohibits role B.
- Two roles A and B arbitrarily mix and match ("don’t care"). Thus, role A may or may not go with Role B; nothing can be said.

Allowed and disallowed combinations are described as a compatibility matrix. In Marko (2004b) the compatibility is constrained in scope of single instance and the constraint is defined inside pattern template so it can be used to control the instantiation process.

The pattern composition is meant to foster pattern base design and development instead using patterns in isolation to stitch application specific components together. Each pattern in isolation only focuses on itself. This makes it harder to recognize pattern inter-relationships and solve more complex system architecture problems effectively. In contrast, a pattern-based design can fulfill a software system's requirements more successfully by integrating the patterns consistently and synergistically (Schmidt et al., 2000).

5.3.2 Phylogenetic Dimension

The pattern itself may evolve and interact with other patterns to form new patterns. Some patterns are composites.

Above mentioned approaches were based on role offers, but there is also another approach of customizing the pattern to meet developer's needs.

Composition

Composition at the phylogenesis level means composing of the pattern templates at the meta level in such way, that one pattern becomes part of another pattern. Gamma et al. (1995) proposes *Pattern Map* which describes potential compositions of pattern of the language. The relationships of patterns have character of suggestions, how patterns

could collaborate and it may be the origin of an effort to describe pattern composition in more detail.

UML profile for Enterprise Distributed Object Architecture (EDOC) specification is designed to provide standard means, Business Function Object Patterns (BFOP), for expressing object models using UML package notation together with the mechanisms for applying patterns that are required to describe models. BFOP is a set of object patterns laid out in a hierarchical multi-layer structure, which represents enterprise system architecture (OMG, 2004b).

Ram et al. (2004) investigates approach to pattern compositions. Two types of pattern interactions are recognized:

- Uses interaction. A pattern uses another patten *uses* to solve its subproblem.
- Combines interaction. A pattern *combines* with another pattern to achieve mixed behavior.

These interactions are used to describe a set of hybrid (composite) patterns which solve higher level problems than composed patterns. Gamma et al. (1995) suggests similar meaningful compositions in the form of pattern map, which describes possible relationships between patterns.

5.3.3 Lifecycle Dimension

The instance of the design pattern may be introduced in various state of abstraction. It is closely related to software development lifecycle. More abstract (informal) concepts mature into concrete formal state. From certain point, there is homomorphic mapping between elements of development artifacts (class models, sources) downstream of the lifecycle (e.g. design level class may map 1:1 to implementation class model). However the level of detailed ness and formalization rises as it approaches implementation. Pattern instances can be also transformed (mapped) through the lifecycle.

In current practice, patterns that are instantiated at any point of the software lifecycle, pass only implicit pattern instance to subsequent stages of the cycle. Explicit pattern instance if any doesn't pass to later stages. For example patterns are instantiated in the design model of the software and code is generated from participating classes. But code generator, virtually, takes only the implicit pattern information into account -- valuable explicit information created with CASE tool is lost and/or incomprehensible for code generator. It would be beneficial to have persistent pattern instance through the software lifecycle (e.g. it could enhance code generation). However standardize carrier of the pattern instance data has to be employed. XML Metadata Interchange standard (OMG 2004a) belongs to the most prospective because of its ties to UML and support by today's CASE tools.

5.4 Conclusions

In this work we have characterized pattern systems, pattern languages and their applications as well as approaches that characterize important properties of the patterns. In concluding section we briefly summarize shortcomings of these approaches and plans for further work.

It is obvious, that pattern modeling technology and tool supported pattern reuse is yet to achieve maturity level as presented in Shaw et al. (2001). There are signs, that it is currently placed somewhere near 3rd or 4th level of the scale. It means that, fundamental research has already taken place and basic concepts have been formulated. There are also concrete applications as well, which can be used to solve real problems. There is an established consensus on benefits and principles of use of design patterns; however the consensus on appropriate modeling techniques is yet to be achieved. We have identified following issues with current way of describing patterns:

- inadequate description of design patterns, allowed by a fact, that patterns are often described using only single or few viewpoints, but modeling is usually limited to static class structure,
- lack of separation of concerns within pattern models, and
- item nonexistence of common communication standards to allow knowledge interchange and cooperation of complementary tools

References

ALUR, D. – CRUPI, J.– MALKS, D. (2003). *Core J2EE$_{TM}$ Patterns: Best Practices and Design Strategies*. Second. Prentice Hall PTR.

APPLETON, B. (2000). *Patterns and Software: Essential Concepts and Terminology*. February, http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

BASS, L. – CLEMENTS, P. – KAZMAN, R. (2003). *Software Architecture in Practice*. Second. Addison-Wesley.

BIELIKOVÁ, M. – KURUC, J. – MARKO, V. (2004). Entry into Virtual University Space through Web-Based e-Application. In: *Proc of 3nd Int. Conference on Emerging Telecommunications Technologies and Applications, ICETA 2004*, Elfa, pp. 403–410.

BOOCH, G. (1994). Designing an Application Framework. In: *Dr. Dobb's Journal* 19 (1994), February, Nr. 2, S. 24.

BROWN, W.J. – MALVEAU, R.C. – MCCORMICK, H.W., ET AL. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons.

CUNNINGHAM, W. (1996) *Portland Pattern Repository*. Available at http://c2.com/ppr/. June.

DOUGLASS, B.P. (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. First. Addison Wesley.

EVENTHELIX.COM (2005) *Embedded Design Pattern Catalog*. Available at http://www.eventhelix.com/RealtimeMantra/PatternCatalog/.

EVITTS, P. (2000). *A UML Pattern Language*. First. New Riders Publishing.

FILKORN, R. – NÁVRAT, P. (2005). An Approach for Integrating Analysis Patterns and Feature Diagrams into Model Driven Architecture. In: *Proceeding of SOFSEM 2005 Vol*. 3381, Springer, pp. 367–370.

FOWLER, M. (1996). *Analysis Patterns: Reusable Object Models*. Addison Wesley.

FOWLER, M. (2004). *Inversion of Control Containers and the Dependency Injection Pattern*. Available at http://martinfowler.com/articles/injection.html.

FOWLER, M. – BECK, K. – BRANT, J. ET AL. (2002a). *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

FOWLER, M. – RICE, D. – FOEMMEL, M. ET AL. (2002b). *Patterns of Enterprise Application Architecture*. Addison Wesley.

GAMMA, E. – HELM, R. – JOHNSON, R. – VLISSIDES, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

HAKALA, M. (2000). Task-Based Tool Support for Framework Specialization. In: *Proceedings of OOPSLA'00 Workshop on Methods and Tools for Framework Development and Specialization*.

HAKALA, M. – HAUTAMÄKI, J. – KOSKIMIES, K. ET AL. (2001). Generating application development environments for Java frameworks. In: *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering GCSE'2001 Vol*. 2186, Springer Verlag, 2001, pp. 163–176.

HAUTAMÄKI, J. (2002). *Task-Driven Framework Specialization – Goal-Oriented Approach, Licentiate thesis, Report A-2002-9*. Department of Computer and Information Sciences, University of Tampere.

HEUZEROTH, D. – HOLL, T. – HÖGSTRÖM, G. ET AL. (2003). Automatic Design Pattern Detection. In: *11th Internatinal Workshop on Program Comprehension, co-located with 25th International Conference on Software Engineering*, IEEE, May 2003.

HUSTON, V. (2002). *Design Patterns*. Available at http://home.earthlink.net/huston2/dp/patterns.html, Februar 2002.

JUDSON, S.R. – CARVER, D.L. – FRANCE, R.B. (2003). A Metamodeling Approach to Model Transformation. In: *Proc. of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '03)*, ACM, October 2003, pp. 326–327

KELLER, R.K. – SCHAUER, R. – ROBITAILLE, S. ET AL. (1999). Pattern-Based Reverse-Engineering of Design Components. In: *Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press, pp. 226–235.

KIM, D.-K. – FRANCE, R. – GHOSH, S. ET AL. (2003). A Role-Based Metamodeling Approach to Specifying Design Patterns. In: *Proc. of the 27th Annual International Computer Software and Applications Conference (COMPSAC '03)*, pp. 452–459.

KRUCHTEN, P. (1995). The 4+1 View Model of Architecture. In: *IEEE Softw.* 12 (1995), No. 6, pp. 42–50.

MAK, J.K.H. – CHOY, C.S.T. – LUN, D.P.K. (2004). Precise Modeling of Design Patterns in UML. In: *Proc. of the 26th International Conference on Software Engineering (ICSE'04)*, pp. 252–261.

MAPELSDEN, D. – HOSKING, J. – GRUNDY, J. (2004). Design Pattern Modelling and Instantiation using DPML. In: Noble, J. (Hrsg.) ; Potter, J. (Hrsg.): *Proc of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)* Bd. 10, Australian Computing Society, pp. 3–11.

MARINESCU, F. (2002). *EJB Design Patterns: Advanced Patterns, Processes and Idioms*. John Wiley & Sons.

MARKO, V. (2004a) Supporting Design Pattern Instantiation in Software Design. In: *Proc. Of the 7th Int. Conference Information Systems Implementation and Modelling ISIM 2004*, MARQ, pp. 85–92.

MARKO, V. (2004b). Template Based, Designer Driven Design Pattern Instantiation Support. In: *Proc.of 8th East-European Conference on Advances in Databases and Information Systems (ADBIS 2004) Vol*. 3255, Benczur, A.; Demetrovicz, J.; Gottlob, G., editors, Springer, pp. 144–158.

MARKO, V. (2005). Describing Structural Properties of Object-Oriented Design Patterns. In: *Communications of SOFSEM 2005*.

MEIJLER, T.D. – DEMEYER, S. – ENGEL, R. (1997). Making Design Patterns Explicit in FACE - A Framework Adaptive Composition Environment. In: *Proceedings of the Sixth European Software Engineering Conference(ESEC/FSE 97)*, Jazayeri, M.; Schauer, H., editors, Springer-Verlag, pp. 94–110.

NODA, N. – KISHI, T. (2001). Design Pattern Concerns for Software Evolution. In: *Proc. Of IWPSE '01*, ACM.

OBJECT MANAGEMENT GROUP. (2003). *UML 2.0 OCL Specification, Version 2.0, Adopted Specification*.October 2003.

OBJECT MANAGEMENT GROUP. (2003a). *UML 2.0 Superstructure Specification, Version 2.0*. August 2003.

OBJECT MANAGEMENT GROUP. (2003b). *Unified Modeling Language (UML) Specification: Infrastructure, Version 2.0, Adopted Specification*. December 2003.

OBJECT MANAGEMENT GROUP. (2004a). *Meta Object Facility (MOF) 2.0 XMI Mapping Specification, Adopted Specification, Version 1.0*. June 2004.

OBJECT MANAGEMENT GROUP. (2004b). *UML Profile for Patterns Specification, Version 1.0*. February 2004.

RAM, D.J. – REDDY, P.J.K. – RAJASREE, M.S. (2004). Pattern Hybridization: Breeding New Designs Out of Pattern Interactions. In: *ACM Software Engineering Notes* 29 (2004), May, No. 4.

RIEHLE, D. (1997). Composite Design Patterns. In: *Proc. of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, ACM Press.

RIEHLE, D. – GROSS, T. (1998). Role Model Based Framework Design and Integration. In: *Proc. of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, ACM Press, 1998, pp. 117–133.

RUMBAUGH, J. – JACOBSON, I. – BOOCH, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.

SALINGAROS, N.A.(2000). The Structure of Pattern Languages. In: *Architectural Research Quarterly* 4, pp. 149–161.

SCHMIDT, D. – STAL, M. – ROHNERT, H. ET AL. (2000). *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2*. John Wiley & Sons.

SCHMIDT, D.C. (1995). Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software. In: *Communications of ACM, Special Issue on Object-Oriented Experiences* 38 (1995), October, No. 10, pp. 1–10.

SHALLOWAY, A. – TROTT, J.R. (2001). *Design Patterns Explained: A New Perspective On Object Oriented Design*. Addison-Wesley.

SHAW, M. (2001). The coming-of-age of software architecture research. In: *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society.

SHU, S. – NORRIE, D.H. (1999). Patterns for Adaptive Multi-Agent Systems in Intelligent Manufacturing. In: *Proc. of the 2nd International Workshop on Intelligent Manufacturing Systems*, pp. 67–74.

SMOLÁROVÁ, M. – NÁVRAT, P. (2000). Reuse with Design Patterns: Towards Pattern-Based Design. In: *Proc. Software: Theory and Practice*, Publishing House of Electronics Industry, pp. 232–235.

SONG, E. – FRANCE, R.B. – KIM, D.-K. (2002). Using Roles for Pattern-Based Model Refactoring. In: *Proceedings of the Workshop on Critical Systems Development with UML CSDUML'02*, pp. 181–188.

TIP, F. – KIEZUN, A. – BÄUMER, D. (2003). Refactoring for Generalization using Type Constraints. In: *Proc. of Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA '03*, ACM.

VÖLTER, M. – SCHMID, A. – WOLFF, E. (2002) *Server Component Patterns: Component Infrastructures Illustrated with EJB*. John Wiley & Sons.

WAKE, W.C. (2003). *Refactoring Workbook*. Addison Wesley.

WU, L. – SAHRAOUI, H. – VALTCHEV, P. (2004) Program comprehension with dynamic recovery of code collaboration patterns and roles. In: *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, pp. 56–67.

YACOUB, S.M. – AMMAR, H.H. (2003). *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software*. Addison-Wesley.

ZHANG, X. – YOUNG, M. – LASSETER, J.H.E.F. (2004). Refining code-design mapping with flow analysis. In: *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, ACM Press, pp. 231–240.

6 FORMALIZATION IN SOFTWARE ARCHITECTURES

Object based systems mature and require major changes and refinements. To model these changes, we use formalisms appropriate to object level. Current research in object level is limited to particular object structures; dynamics of changes structures one-to-other are being studied. Object structures are analogy to software architectures, which are well known thanks formal languages called ADLs. We provide a survey of current state of software architecture formalization and possibilities of transfer of architectures-based knowledge to object level. The most appropriate way seems to be in using UML in combination with OCL, which is de facto current industrial standard considered a formal language.

6.1 Formalization

6.1.1 Motivation

Although it could sound a little bit funny, the question "why to formalize" must be placed. The *why* must be answered before we get to any research. The formalization is believed to be a way of abstract modeling of a given problem domain. Anyway, browsing articles upon formalization, there is a lot of lines dealing with concrete formalisms but a few of them tries to answer why they formalize anything. For software architecture, (Allen, 1997) stated:

Evidently, what is needed is a more rigorous basis for describing software architectures. At the very least, we should be able to say precisely what is intended meaning of a box-and-line description of some system. More ambitiously, we should be able to check that the overall description is consistent in the sense that the parts fit together appropriately. More ambitiously still, we would like a complete theory of architectural description that allows us to reason about the behavior of a system as a whole.

The given quotation provides three points of formalization. Although it is not a definition of formalization, it clearly states three reasons for formalization in software architectures.

6.1.2 Formal and semiformal methods

As we are looking for rigorous notations, we must decide what are properties and features of formal methods. If we do so, we can decide upon what are so-called semiformal methods. However, this is not discussed and every author believes that there is general consensus upon the division. Consequently, some methods are shifting in general belief from semiformal to formal as no criteria are stated. Great example is UML (Unified Modeling Language).

No doubt that divison of methods is depending on purpose of formalization. Formalization is closely connected to abstraction. Abstraction leads to omitting of *unimportant* details. This fact brings more complications to the methods' division.

The *important* details and *for what* (clearly other words for "omitting unimportant" and "why") are easily found in ADLs (Architecture Description Languages, see Section 6.4). By (Clements, 1996), "ADLs result from a linguistic approach to the formal representation of architectures, and as such they address the shortcomings of informal representations". However, a deeper look at the ADLs will ensure us that each of them uses its own definitions of notions taking into account only chosen details. And they attempt to reason in some given areas (although general, without a given domain, but just about a chosen set of features and properties).

Formal methods

What is formal, indeed? Formalization is a notion referring to arranging something according to fixed structure (Procter, 1995). Simply, form is fixed structure, formalization is a process of setting something to a form. Preferably, formal structure should be fixed in syntax and semantics. Hence formalization is a process of providing given object in a form that has the same meaning every-time (under the same conditions). This is a reason why generally describing architecture by line-and-box scratches is not formal as well as describing architecture by simple text using spoken English.

Formal methods are usually based on mathematics. As so, algebra (e.g. Object Algebras) and logic (e.g. π-calculus) are used. These methods clearly define semantics. ADLs are believed to be formal as they provide the notation of architecture that is understandable the same way every-time. These languages have fixed syntax and (usually) defined semantics, preferably using denotations (formal interpretation of meaning in terms of given language and its vocabulary).

However, denotations for complex notations as ADLs are so hard to declare that the most of ADLs has barely defined some operational semantics and vocabulary. At this point, we come to UML (Unified Modeling Language, (OMG, 2003a) which is often considered a semi-formal notation. UML has strictly defined syntax, operational semantics and stable, fixed vocabulary of terms. In addition, profile of the language (the part of the UML belonging to implied extension mechanism of UML) defines strictly logical boundaries of stereotypes (semantic and syntactical identifiers of the language), basically using OCL (Object Constraint Language, (UML, 2003) with strict denotation semantics. As such, admitting that ADLs are formal languages and belong to formal methods, UML is ultimately a formal language. This fact took a long time software engineering community to be accepted and till now, there exists minor part of it that considers UML to be semiformal rather than informal.

Semiformal methods

Semi-formality is a hardly explainable concept. Even often used, there is no definition of the concept. The best description of it could be "weaken formality". To weaken a concept refers to relaxing assumptions of formality. What is omitted in *semi*formality?

Let us get back to UML; UML has been considered by many authors as semiformal notation. The reasons for that were in little knowledge of meta structures of the language defining syntax and semantics, and naturally in fact that majority of tools "supporting" the language were simply paint-brushes providing predefined pictures.

To close the discussion on formalization, we use a quotation from (Allen, 1997). Allen scratched the notion of *informal* description; in addition he introduced current state in search for formal methods in software architectures descriptions.

We take as our starting point a view of architectural description inspired by informal descriptions: a software architecture can be defined as a collection of computational components together with a collection of connectors, which describe the interactions between the components. While this abstraction ignores some important aspects of architectural description (such as hierarchical decomposition, assignment of computations to processors, and global synchronization and scheduling), it provides a convenient starting point for discussing architectural description.

6.2 Object level formalization

The level of object refers to a view to a software at the level of simple structures. In fact, this is very detailed view; the simplicity and natural evolution of this level is based on formalisms (mathematics, logics). Depending on paradigm, the object level refers to the lowest level of software engineering. It covers methods and attributes of objects, executive code, and others. This view can be considered as a view to code level where the code is conceptually organized.

Objects are studied in two simple ways: the object in whole, and the object as composition of code and data. The compositional view is better understood; it uses results gained in studies to code level, and the code itself can be partially direct mapped to some formalisms. Formal methods used at this level are formal languages, functions (algebraic; functionals of recursive nature), etc. We don't want to analyze these methods.

Considering the object in whole, well suited formalisms has been introduced. The common denominator of these methods lays in descriptions of object structure - a hierarchy that objects are organized in. The formalisms of this type are usually based on algebraic structures embedded with logics. Special family of these methods is covered by Object Algebras. On the edge of them, there are specialized formal languages adapted to object structures, e.g. Object-Z. The most popular approach is the use of Unified Modeling Language embedded with Object Constraint Language. Some researches even admit usage of programming languages - mainly their declarative parts. We believe, that algebraic approach is the most useful thanks its level of understandability accompanied by their virtue to cover reasoning upon the structure.

6.2.1 Dynamic versus static

Formal descriptions of object structures could be classified as either static or dynamic. This classification refers to language facility to model system dynamics and evolution. Dynamics is considered from the point of changes to a system; the dynamics refers to changes of the (object) structure. From the point of view of ADLs, dynamics refers to dynamics of configurations (see Section 6.4.1).

Majority of languages are rather static. Static description brings better understanding of structure and allows some reasoning about communication within the structure, delegation, and other important aspects of objects. However, static languages lack ability to cover changes of object hierarchy; therefore they are useless in reasoning about changes of structure in time. Even OCL is a static language, as we demonstrate in 6.7.

Dynamic languages are the point of interest of research in the area of object structure description. There is a great number of methods able to study computations in meaning of executive code dynamics, but there is a little knowledge about dynamics of object structures. We have already tried to suggest an instance of object algebra - a language describing object structure, 6.2.2. However, the notation seems to be static, now, and hence of little use in case of dynamical changes description. Because of resemblance between object structures (as a system) and software architectures, the study of dynamics describing Architecture Description Languages (ADLs; Section 6.4) should be a good motivation for object structure behavioral descriptions. Alternatively, ADLs could be rigged to suit the needs of object structure dynamics.

6.2.2 Static description

We have studied static object structure description notation based upon semantical model of OCL (Navarčik, 2005). The language strictly follows the features of object structures in meanings of object-oriented paradigm.

Object structure is signed as O_i; the indexing allows distinguish among different structures. Indexed structures may be versions of structure after changes (structure refinement, modifications, two structure integration, etc.); this feature however is not facility of dynamics modeling, yet. This is the task for further research.

Object structure O_i is the mathematical structure $O_i = \{CLASS_i, ATT_i, OP_i, ASSOC_i, \prec_i\}$ but we prefer definition without member data and operations of the classes as $O_i = \{CLASS_i, ASSOC_i, \prec_i\}$ where $CLASS_i$ is a set of classes $\{c_{i1}, c_{i2}, ..., c_{ix}\}$, where x is arbitrary number with cardinality of set of classes; $ASSOC_i$ is a set of associations between classes in O_i; $\prec_i$ is a partial ordering on $CLASS_i$ reflecting the generalization hierarchy of classes (generalization and specialization by superclasses and subclasses); ATT_i or $ATT_{c_{ij}}$ is a set of operation signatures for functions mapping an object of class structure O_i or class c_{ij}, respectively, to an associated class member data value; OP_i or $OP_{c_{ij}}$ is a set of signatures for class member functions of all O_i and class c_{ij}, respectively.

Then we encapsulate properties (attributes and methods) to the class c as c_{ij} = ($ATT_{c_{ij}}$, $OP_{c_{ij}}$). The full definition is tuple containing also class associations and

hierarchy in $c_{ij} = (ATT_{c_{ij}}, OP_{c_{ij}}, ASSOC_{c_{ij}}, \prec_{c_{ij}})$.

Formally, $ASSOC_i = \{as \mid \text{as} \mapsto \langle c_{i1},...c_{in}\rangle \mid c_{ij} \in CLASS_i, \text{for } \forall \text{j} \in \{1,...,n\}\}$. The set is given by finite set of names and a function associates: *associates*: $ASSOC \to CLASS^+$ as $as \mapsto \langle c_1,...,c_n\rangle$ with $n \geq 2$; mostly the association is binary, $associates(as) = \langle c_{i_k1}, c_{i_k2}\rangle$ where $k_1, k_2 \in 1,...,x$.

$ATT_i = \{a : t_{c_{ij}} \to \text{t} \mid c_{ij} \in CLASS_i, \text{for } \forall \text{j} \in \{1,...,n\}\}$ or $ATT_{c_{ij}} = \{a : t_{c_{ij}} \to \text{t} \mid c_{ij} \in CLASS_i\}$, where t is any primitive type in environment (formally, the type is unimportant and the only primitive type in pure object environment could be the Object), $t_{c_{ij}}$ is class c_{ij} as a type in system.

$OP_i = \{\omega : t_{c_{ij}} \times t_1 \times t_2 \times ... \times t_n \to \text{t} \mid c_{ij} \in CLASS_i, \text{for } \forall \text{j} \in \{1,...,n\}\}$ or $OP_{c_{ij}} = \{\omega : t_{c_{ij}} \times t_1 \times t_2 \times ... \times t_m \to \text{t} \mid c_{ij} \in CLASS_i\}$, where t, t_l stands for types, $t_{c_{ij}}$ is a c_{ij} class subtype.

Aggregation and containment are modeled through the roles and multiplicities. Classes can appear in more association playing different roles. Role names are defined by a function roles, roles: $ASSOC \to N^+$, where $N \subseteq A^+$ are non-empty names over alphabet A, roles: $as \to < r_1, ..., r_n >$, with $n \geq 2$. Mostly, $roles(as) = < r_1, r_2 >$.

The number of links between objects of associated classes is specified by function (relation) multiplicities, multiplicities: $ASSOC \to I$ as $as \to < M_1, ..., M_n >$, $n \geq 2$. The function navends gives the set of role names navigable from a class c over an association as. Function navends is projection $CLASS \times ASSOC \to \mathcal{P}(N)$ described as $(c, as) \to \{ r \mid associates(as) = < c_1, ..., c_n > \wedge roles(as) = < r_1, ..., r_n > \wedge \exists i, j \in \{1, ..., n\}: (i \neq j \wedge c_i = c \wedge r_i = r)\}$.

6.3 Software architecture

Section 6.2 has studied object level as a level of view to a software systems. The higher level of view to a system is by software architectures. Study to software architectures and their formalizations may lead us to some important results that could be applicable in dynamics modeling in object structure formalisms. The object structure is a special case of and narrowed view to software architecture. As we will try to show, object based approach has a strong influence to software architectures and we would like to enforce the backwards influence, too.

Software architecture is notion widely used in software engineering. This concept is generally accepted and used although there is no universal definition of software architecture. This key notion has more than 90 different meaningful definitions (Clements, 2002). The (Clements, 2002) provide their own definition of the software architecture:

A software architecture for a system is the structure or structures of the system, which consist of elements, their externally visible properties, and the relationship among them.

To illustrate inconsistency and disagreement upon what architecture is, we provide a definition of architecture used by (ROP, 1998). The definition is accepted and approved by IEEE. It says:

[Software architecture is] The highest level concept of a system in its environment. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces.

The definition is discussed e.g. in (Fowler, 2003a). As an important point of the discussion, the fact that "there is no highest level concept of a system" is emphasized. Anyway, any definition of software architecture is discussable in this way.

6.3.1 Trinity of architecture elements

We will accept the definition provided by (Clements, 2002) as quoted above. The definition, in other words, describes the trinity of architecture. Architecture of software system can be modeled using three basic concepts:

- components,
- connectors,
- configurations.

This taxonomy is widely accepted. However, for practical purposes, the configuration is omitted. In such a case, configuration comes as a part of connectors. Connectors turn to a complex glue interconnecting components. (Clements, 2002) identified this view to software architecture as component and connector view style, C&C style. This view style is so typical for industrial usage of architecture that even in research configuration, as the third concept of software architecture, is often out of scope.

Components

By (Clements, 2002) components are defined as:

Components are the principal computational elements and data stores that execute in a system.

This definition shows that components are prime elements of architecture. Anyway, component can be as small as single procedure or as large as entire application (Medvidovic, 2000). Component requires its own execution space and data space. Both can be shared with other elements. In general, when modeling components, they should be named and have a type. There is no common type taxonomy. Example of type could be *filter*, which requires some input data at input channel, performs a transformation and puts result of transformation to output channel interfaces.

Component has interfaces often called *ports*. An interface is a specific point of potential interaction of a component with its environment (Clements, 2002). The notion of port is used to emphasize the runtime nature of it hence port is a kind of interface. Anyway, there is no general consensus upon the proper notion, as shown in Section 6.4.1, following (Medvidovic, 2000). Different authors and research have their own understanding of concepts of port and interface; thus interface of one author is a port by definition of the other author. Anyway, except the UML usage, we will use port and

interface interchangeably. Let us show now the difference between port and interface as defined in UML 2.0.

Each notion of component, interface and port are included in UML 2.0 specification. As such, UML 2.0 brings own interpretation of notions. Let us compare both interface and port to find a difference. By (OMG, 2003a), the port is:

A port is a structural feature of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier. A port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment.

To comparison, interface is much more complex notion:

An interface is a kind of classifier that represents a declaration of a set of coherent public features and obligations. In a sense, an interface specifies a kind of contract which must be fulfilled by any instance of a classifier that realizes the interface. The obligations that may be associated with an interface are in the form of various kinds of constraints (such as pre- and postconditions) or protocol specifications, which may impose ordering restrictions on interactions through the interface. Since interfaces are declarations, they are not directly instantiable. Instead, an interface specification is realized by an instance of a classifier, such as a class, which means that it presents a public facade that conforms to the interface specification. Note that a given classifier may realize more than one interface and that an interface may be realized by a number of different classifiers.

First point that appears is port is a structural feature of a classifier and interface is kind of classifier. Hence port is narrow notion of interface. In addition, even by UML 2.0, port is of runtime nature. We will try to study interfaces in connection with notion of connector.

Connectors

By (Clements, 2002) connectors are:

A connector is a runtime pathway of interaction between two or more components.

By (Medvidovic, 2000) connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Simple kinds of connectors are procedure calls (between objects, client and servers), asynchronous messages, multicasts, etc. (Clements, 2002). Connectors can be classified in types, though types are rarely used. Good example of connector is (type of) pipe - a connector with asynchronous order preserving data stream.

As Medvidovic and Taylor remarked, connectors can not correspond to compilation units in an implemented system. They may be implemented as separately compilable message routing devices, but may also manifest themselves as shared variables, table entries, buffers, instructions to a linker, dynamic data structures, sequences of procedure calls embedded in code, initialization parameters, etc.

Now, in connection with UML 2.0, interface can be considered in some cases to be a part of connector. Anyway, connector is partially covered by "assembly connector" in UML 2.0 (usually called simply “assembly”):

An assembly connector is a connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port.

Till components are the most natural and the most expected compound of the architecture trinity, connectors may be considered to be and extra overhead. However, connectors are those points of match; they capture communication, interchange in between components. Connectors make components älive". For better understanding of the notion we will study an example of architecture description language called Wright; the study will hopefully help us to clear the notion.

Configurations

Components are elements of an architecture, connectors define protocols or the way how components can communicate. The trinity is completed by defining the way the components are interconnected via connectors to fulfill requirements put on software. By (Medvidovic, 2000), architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether appropriate components are connected, their interfaces match, connectors enable proper communication, and their combined semantics result in desired behavior.

In concert with models of components and connectors, descriptions of configurations enable assessment of concurrent and distributed aspects of an architecture, e.g., potential for deadlocks and starvation; performance, reliability, security, and so on. Descriptions of configurations also allow analysis of architectures for adherence to design heuristics (e.g., direct communication links between components hamper evolvability of an architecture) and architectural style constraints (e.g., direct communication links between components are disallowed).

Configurations complete the trinity of software architecture. If components and connectors are enough to make the *system* "alive", why do we need configuration? Components and connectors make no assumptions or normatives upon final constitution of system. Matching a components through connectors is not enough to reason about the architecture and the system. The configuration tells what-to-what is connected, makes all the checks and allows software architect to be architect, not just an oracle. We will come to the notion at example of ADL - Wright, in 6.4.2.

6.3.2 Software architecture views, viewtypes and styles

(Clements, 2002) introduced another way to software architectures. Previous research was, in general, oriented to modeling architectures. It has been resulting in wide range of approaches, each identifying important details. But simple questions, e.g. what is architecture, is blackboard a pattern or architecture or style, etc., were hard to answer. As (Clements, 2002) decided to write a book a book about documenting software architectures, they had to answer those simple, fundamental questions.

The new way is based on an idea that we need to document architecture used in practical applications. Documentation enforced systematization in software architecture knowledge. The passive (documenting) goal brought a static and systematic view on architecture. Note that this approach does not explain software

architecture the same way as trinity; this approach is a systematic view to architectures and their descriptions.

Views

View is defined as a representation of a set of system elements and the relationships associated with them". View is a perspective look at the modeled reality. For practitioners, a view is, for instance a chosen type of diagram over a model (software system) capturing a concrete part of it. In words of UML, one can choose a appropriate diagram to represent behavior, static structure, etc. of given system (or the part of it). The notion of view is fundamental in documenting software architecture. Anyway, it is fundamental in documenting *any* system. And not even in documenting. (Clements, 2002) gives a great example to explain views using an analogy to bird wing:

There is no single rendition of a bird wing. Instead, there are many: feathers, skeleton, circulation, muscular views, and many others. Which of these views is architecture of the wing? None of them. Which views convey the architecture? All of them.

Different views are used for different looks at the modeled architecture. For instance, a layered view tells about the system portability, a deployment view will let us reason about system reliability and performance, etc. The notion of view is a part of methods and methodologies. A good example is Krutchen's “4+1” approach[8] (Kruchten, 1995) to architecture, now a foundation of e.g. Rational Unified Process. These views are:

- logical view – for capturing behavioral requirements of the system, the services provided to users; these view leads to object decomposition, and additionally can lead to functional analysis; identifies mechanisms common to whole system,
- process view – studies threads of control (of elements identified using logical view), processes constituting the system; it checks concurrency, distribution, system integrity, fault tolerance,
- development view – searches for the organization of software modules; enables reasoning about reuse, portability, security,
- physical view – focuses on system requirements: system availability, reliability, performance, scalability, etc.; it maps various elements identified in the other three views (networks of control, processes, tasks, objects) onto processing nodes,
- the "+1" stands for subset of scenarios (use case instances) to show cooperation of elements of the 4 views; this view is redundant

Viewtypes

(Clements, 2002) suggested three types of views, simply viewtypes, that they use in their book to document software architectures. Anyway, viewtype is defined as a definition of “the element types and relationships types used to describe the architecture of a software system from a particular perspective”.

---

[8] The "4+1" approach is now implemented in many professional software engineering tools. For instance, basic profile used by Enterprise Architect assumes the "4+1" approach with extension to custom view.

Viewtypes identified by (Clements, 2002) are:

- the module viewtype – principal structure with modules as elements,
- the Component and connector (C&C) viewtype – runtime execution units and potential interactions among them,
- the allocation viewtype – mapping components and connectors to elements of environment.

Viewtypes do not limit views. View types only capture the fact, that software architects must think about the system in three ways: how the system is structured in implementation units, how is structured in runtime, what are behavioral elements and their interactions, and how the system relates to non-software structures in its environment. Viewtype captures the perspective that must be considered by architect.

Styles

Studies of software architectures have shown that even in different systems some recurring forms are widely observed. This forms are known as styles. Styles (architectural styles) are defined as ä specialization of elements and relation types, together with a set of constraints on how they can be used". A style, in other words, defines a family of architectures that satisfy the constraints. Example of a style is client-server or pipe-and-filter. This is important detail because in praxis architectural styles are often considered to be architectures. Style brings important constraints to a system, impairs important properties and simplifies reasoning upon a system. Anyway, style itself is useless as it can not describe the system on its own; style can bind (important) properties of the system, only.

No system is built from one single style. Styles are combined to provide the best properties. The resulting system is a amalgamation of styles (Clements, 2002).

What is the final superposition of views, viewtypes and styles? The highest level is a perspective - choosing a viewtype we are going to consider the system. The middle is a style - which brings constraints to a system. Finally, the lowest is a view - providing final "window" to a system.

6.4 Architecture Description Languages

Architecture description languages are a class of languages defined by (Clements, 2002) as follows:

An architecture description language (ADL) is a language (graphical, textual, or both) for describing a software system in terms of its architectural elements and the relationships among them.

The example of ADL is Wright, see Section 6.4.2, special case on edge of ADLs is ACME, described in Section 6.4.3. Brief survey of other particular ADLs is available in appendix Section 6.7.2.

There is no surprise that definition merely corresponds to definition of software architecture, described in Section 6.3. Anyway, this definition explicitly involves graphical languages into ADLs. Although pioneers of ADLs support graphical representation, it was a long time considered only an aid to textual representation.

There has always been demand for more readable and usable tools for industrial usage.

6.4.1 Features of ADLs

As we have already seen in Section 6.3 at software architecture, there must be a taxonomy of features provided and supported by ADLs to fulfill intention of description.

(Medvidovic, 2000) introduced his own framework for ADL classification. The purpose of the framework is to classify ADLs depending on virtue they provide. The classification framework from Medvidovic and Taylor must be hence considered as ideal (maximal) set of features expected as those provided by ADLs.

The framework is a perfect summary for understanding ADLs. The architecture modeling features identified by Medvidovic and Taylor for ADLs are (simplified in descriptions, partially cited):

- **Components**
  - Interface – a set of interaction points between component and the external world. The interface specifies services (messages, operations, and variables) a component provides. In order to support reasoning about a component and the architecture that includes it, ADLs may also provide facilities for specifying component needs, i.e., services required of other components in the architecture. An interface thus defines computational commitments a component can make and constraints on its usage.
  - Types – abstractions that encapsulate functionality into reusable blocks. A component type may be parameterized and reused.
  - Semantics – high-level model of a component behavior. Such a model is needed to perform analysis, enforce architectural constraints, and ensure consistent mappings of architectures from one level of abstraction to another. It expresses application-level functionality.
  - Constraints – a property of or assertion about a system or one of its parts, the violation of which will render the system unacceptable (or less desirable) to one or more stake holders. This feature ensures adherence to intended component uses, enforces usage boundaries, and establish dependencies among internal parts of a component, constraints on them must be specified.
  - Evolution – (informally defined) the modification of (a subset of) a component properties, e.g., interface, behavior, or implementation. ADLs can ensure systematic evolution by employing techniques such as sub-typing of component types and refinement of component features.
  - Non-functional properties – e.g. safety, security, performance, portability; early specification of such properties (at the architectural level) is needed to enable simulation of runtime behavior, perform analysis, enforce constraints, map component implementations to processors, and aid in project management.
- **Connectors**
  - Interface – a set of interaction points between the connector and the components and other connectors attached to it. It enables proper connectivity of components and their interaction in an architecture and, thereby, reasoning about architectural configurations.

- Types – abstractions that encapsulate component communication, coordination, and mediation decisions. Architecture-level interactions may be characterized by complex protocols. Reusable protocols (in architecture and across architectures) are modeled as types either as extensible type systems, defined in terms of interaction protocols, or as built-in, enumerated types, based on particular implementation mechanisms.
- Semantics – high-level model of a connector behavior. They entail specifications of (computation-independent) interaction protocols.
- Constraints – ensure adherence to intended interaction protocols, establish intraconnector dependencies[9], and enforce usage boundaries.
- Evolution – the modification of (a subset of) its properties, e.g., interface, semantics, or constraints on the two consecutive versions or modifications of the connector
- Non-functional properties – represent (additional) requirements for correct connector implementation.

▪ **Architectural configurations**

- Understandability – One role of software architecture is to serve as an early communication conduit for different stakeholders in a project and facilitate understanding of (families of) systems at a high level of abstraction. Thus to model structural (topological) information with simple and understandable syntax is required.
- Compositionality – (hierarchical composition) a mechanism that allows architectures to describe software systems at different levels of detail. Complex structure and behavior may be explicitly represented or they may be abstracted away into a single component or connector. Situations may also arise in which an entire architecture becomes a single component in another, larger architecture. Such abstraction mechanisms should be provided as part of an ADLs modeling capabilities.
- Refinement and traceability – ability to correct and consistent refinement of architectures into executable systems and traceability of changes across levels of architectural refinement.
- Heterogenity – ability to facilitate development of large-scale systems, preferably with preexisting components and connectors of varying granularity, possibly specified in different formal modeling languages and implemented in different programming languages, with varying operating system requirements, and supporting different communication protocols.
- Scalability – possibility to cope with the issues of software complexity and size; support of specification and development of large-scale systems that are likely to grow further.

[9] Over again, the framework covers maximal expectable set of features which could be ideally provided by ADLs. Not every listed feature must be supported but modeling of fundamental elements: components, interfaces of components, connectors, configurations.

- Evolution – the ability to cope with software evolution, shifts of concepts, updating requirements; support at the level of components and connectors with features for their incremental addition, removal, replacement, and reconnection in a configuration.
- Dynamism – feature of modifying the architecture and enacting those modifications in the system while the system is executing. To support architecture-based run-time evolution, ADLs need to provide specific features for modeling dynamic changes and techniques for effecting them in the running system.
- Constraints – dependencies in a configuration; complementing those specific ones to individual components and connectors; so-called global constraints. Often referring to connector and component constraints.
- Non-functional properties – system-level, rather than individual component or connector properties.

Note that items written in bold are considered essential features: components, interfaces of components, connectors and configurations. These items can be considered as essential minimal set of features of ADLs.

In addition Medvidovic and Taylor complete ADL's profile within their framework by "Tool support" item. Although ADLs have been created at academical spheres, since the very beginning the goal of any research to ADLs has been including creation of supporting tool. Support features for ADLs complete the approximation of ADLs' features. Those covers active specification, multiple views, analysis, refinement, implementation generation and dynamism.

Active specification refers to tool's ability to reduce cognitive load put on architect. In other words, the tool should provide guidance through specification, preferably by pro-active suggests of further steps and by pruning useless courses in specification. Multiple views reflect the fact, that complex *over*view to architecture is Baffin and there is a need to filter unimportant out. View is a simplified look at the architecture (or model in general) that copes with only those details important for the current stakeholder (Clements, 2002). Analysis is ability to evaluate the properties of such systems upstream, at an architectural level. It deals especially with bottom-up reasoning. Refinement provides ability to improve architecture by refining details at different levels of view. Implementation generation gives the real automation to the process of software development. As such, any ADL tool should be able to produce more or less complex real code solution, usually at the level of skeleton. There is a question how far and deep should ADL's code generation go as component or connector codes are traditionally specified in design-time tools. Dynamism reflects expected ADL's feature of architectural configuration level. This enforces requirement on tool to support changes, enforce correctness of changes and probably code generated by tool should provide vital testing of dynamically changed specifications.

6.4.2 Example of ADL: Wright

Wright is a general purpose architecture description language. Wright (Allen, 1998a) has been intended to be ADL capable to model any style hence it does not bound modeling to any particular style. In other words, Wright does not enforce any rules of particular style (compared to e.g. C2). (Medvidovic, 2002) notes that Wright keeps

some limitations on topology of architecture: Wright disallows two connectors from being directly attached to one another.

The language is based on theory by C. A. R. Hoare of communicating sequential processes (CSPs) introduced in 1985 (Hoare, 1985). CSP theory is widely used especially in modeling of distributed systems . Wright assumes that any system is a system of finite set of finite-state sequential processes that by communication create a complex system (in fact, the assumption of finiteness uses a subset of CSP, only). The assumption of CSP nature simplifies the task of modeling architecture and "imports" theoretical results.

Wright explicitly distinguishes connectors from components. Allen and Garlan consider this departure to be Wright's most significant point. They argue that *...it simplifies the language. Moreover, if there is only one form of abstraction in the language (namely, that used to define computational elements), the mapping to a semantic base (such as CSP or, in the case of Rapide, Posets) is simpler.*

In addition, component and connectors approach supports reuse.

Communicating sequential processes

Allen and Garlan emphasized connectors and reasoning upon their protocols by CSPs. They put three reasons for using CSP as formalism:

- ability to capture certain critical properties of architectural connection - the power of CSP semantics allows to model inter-component communication and detect mismatched assumptions (the communication failures); CSP provides external (deterministic) and internal (nondeterministic) operators which allow to identify action and reaction responsibilities in system
- simple form of composition - the parallel operator enables to reason about behavior of components without reasoning about their composition; composition thanks the operator will respect assumptions established about the parts
- pragmatic concern for specification checking automated tools

CSP processes are not meant as processes that must be directly implemented (as for example UNIX processes, threads in Java, etc.); processes are logical entities that build the software architecture specification - consequently, their final implementation can be complete different from processes, e.g. as functions or procedures. CSP notation relevant to Wright is the following:

- process (`P`) communication and events (`e`):
  - `e?x` - event e with input value x
  - `e!x` - event e with output value x
  - `STOP` - the stop event (final state)
  - √ - success event
- α`P` - alphabet of P; the set of events with which a process `P` can communicate
- `e → P` - prefixing; `A = e → P` is a process that engages event `e` and then becomes process `P`

- P □ Q - external choice; alternative; process A = P □ Q behaves like P or Q upon environment[10] choice
- P ⊓ Q - internal choice; decision; process A = P ⊓ Q behaves like P or Q by (nondeterministic) decision of process itself
- processes are named, the namespace is finite although names can be associated with a (recursive) process expression
- $P_1 \parallel P_2$ - parallel composition of processes:
    - processes P_1 and P_2 may interact by jointly (synchronously) engaging events that belong to both αP_1 and αP_2, too (alphabet intersection)
    - direct usage of parallel composition is not allowed in definitions of ports, roles, glue processes

In addition, Wright defines for connectors a special process called glue. Glue is a central process of protocol that "glues" roles, their cooperation. Other important symbol introduced in Wright is § - defined as $\S \stackrel{def}{=} \surd \rightarrow STOP$ - stands for successfully terminating process (in CSP the process is called SKIP).

Example in Wright

Wright enables both static and dynamic description of the system; thanks CSPs the specifications of port protocols and connector behavior. 6.4.1 gives an simple example of client-server system (example adapted and refined from (Allen, 1998a).

```
System SimpleClientServer
  component Server =
    port Provide = (request!x → result?y → Provide) ⊓ §
    spec Compute = ¯(internalComputation?x → Provide.request!x
 → Provide.result?y → internalPostprocess!y
 → internalComputation) ⊓ §
  component Client =
    port Request = (invoke?x → return!y → Request) □ §
    spec Compute = ¯(internalComputation?x → Request.invoke!x
 → Request.return?y → internalPostprocess!y
 → internalComputation) ⊓ §
  connector CSConnector =
    role Requestor = (request!x → result?y → Requestor) ⊓ §
    role Provider = (invoke?x → return!y → Provider) □ §
    glue = ¯ (Requestor.request?x → Provider.invoke!x
 → Provider.return?y → Requestor.result!y → glue) □ §

Instances
  c: Client
  s: Server
  csc: CSConnector
```

[10] Environment refers to the other processes that interact with the process

```
Attachments
  s.provide as csc.Provider
  c.request as csc.Requestor
end SimpleClientServer
```

Example 6-1. Simple Client-server system denoted in (static) Wright.

The description of the system, named SimpleClientServer, consists of three parts. The first one specifies components (Server, Client) and connectors (CSConnector) of the system. Components are described by their ports and specification of their function. The example shows only one port per component; Wright does not limit the number of component's ports. Connectors are described by roles they expect and the glue process. The glue process specifies the cooperation of roles; it "coordinates the behavior of the two roles by indicating how the events of the roles work together" (Allen, 1998a).

The second part of the description defines instances of components and connectors in the system. While first part has defined specifications and behavior in general, the second part defines real usages of components and connectors. Naturally, the number of instances is not limited by the language but specification restrictions that invalidate resulting system.

The third part defines attachments of instances: which components are connected to which connectors as which roles. Note that attachments are allowed in pairs component – connector.

Defined system can be checked for validity by semantics of CSP and their rules. The process should be suitable for the role process in such a way that the rest of the connector interaction cannot detect that the role process has been replaced by the port process. Allen and Garlan used CSPs' refinement relationship to check suitability.

In CSP, process P is a triple (A, F, D), where A is the alphabet of P; F is a set of "failures"; and D is a set of "divergences". The failures of a process (F) is a set of pairs (trace, set of events); process can "refuse" participation in these events after executing the trace. The divergences of a process (D) is a set of traces of P after which P can behave "chaotically"; essentially, with arbitrary behavior.

By definition, process P = $((A_p, F_p, D_p))$ is refined by process Q = (A_q, F_q, D_q), P ► Q, if

1. P = Q,
2. $F_p \subseteq F_q$
3. $D_p \subseteq D_q$.

For checking the suitability of attachments, the property $\alpha P = \alpha Q$ is the most important. This means that refinement P ► Q requires that Q respects all interaction obligations of P within the environment. Note that if P permits an internal choice among alternatives, Q is allowed to constrain the choices.

Notes upon Wright

The language of Wright is still considered for adopting CSP as a language strong in connector analysis, especially in deadlock freedom of the system. However, Wright is a language that faces the same story as many of ADLs: it was the point of interest for couple years and nowadays it is out of main stream. We will discuss this later.

Although Wright has dispositions of deadlock prediction, strong dynamics studies through CSPs and well-formed theory behind, its authors are still looking for making the Wright stronger. The most interesting part of the research is introducing dynamics into Wright (Allen, 1998b). However, the work introduced some new concepts and brought some new notations.

6.4.3 Architecture interchange language: ACME

ACME is an example of language that follows ADLs. ACME can be considered a *meta* ADL, as the language is designed and developed for architecture interchange. The language is based upon the following core ontology for architectural representation:

- components,
- connectors,
- systems, or configurations of components and connectors,
- ports, or points of interaction with a component,
- roles, or points of interaction with a connector,
- representations, used to model hierarchical compositions, and
- rep-maps, which map a composite component or connector's internal architecture to elements of its external interface.

Any other aspect of architectural description is represented with property lists. In other words, nothing but ontology is considered by ACME to be architecture core.

ACME has a special position among ADLs: it represents the least common denominator of existing ADLs, rather than a definition of an ADL (Medvidovic, 2000).

ACME brings uninterpreted annotations of elements. See appendix Section 6.7.3 for details. ACME supports components as implementation independent components; interfaces are named ports. ACME is typed language with extensible type system (hierarchical), supporting parameterization. ACME has no semantics corresponding to its purpose - interchange language; consequently constraining the system is enabled via interfaces, only. Connectors are described explicitly by listing; connector interfaces are called roles and connectors are described via protocols (compare to Wright, 6.4.2). Configuration is described by attachments.

There is a chance that ACME could be replaced (in connection with UML mapped ADLs) by XML Metadata Interchange (XMI, (OMG, 2003b); for detailed discussion see Section 6.5.2). However, the main difference is in fact, that XMI is dedicated to UML while ACME should be mediator among many languages. Anyway, there is a number of critiques to ACME that some languages (e.g. C2) are pretty incompatible with ACME and hence mediation cannot be applied.

6.5 Object based approaches versus software architectures

Section 6.2 discusses object level of software; the level in meaning of detail view to software. Interest in object environment comes from the fact that although current

software research main stream is oriented to post-object paradigms, object based systems still play very important role in software development. Post-object approaches, however, build up on object level as object approach does on his predecessors (functional programming paradigm, procedural approach, ...). Object based systems matured to age when their deep modifications are required; these tends are represented by refactoring (Fowler, 2002), refinements studies (Hnatkowska, 2004), etc. The modification corresponds to dynamics as described in Section 6.2.1.

On the other hand, software architectures, that are discussed in Section 6.3, have different origin; their basic purpose is to support software development and its consecutive activities at the large-scale, high-level abstraction. This view to software architectures enabled deep studies to aspects of dynamics, inter alia. The question that should come to mind is: what is the relation between object structures and their formalizations and software architectures and ADLs? Additionally, object oriented approaches resulted into some important tools, languages, methods, e.g. Unified Modeling Language, Object Constraint Language or XML Metamodel Interchange (described in Section 6.5.2). Can these products of object oriented approaches adapted to software architectures and ADLs? In case of positive answer, we could be able to interchange results of both researches in object level and software architectures.

6.5.1 Unified Modeling Language

Unified Modeling Language is "family of graphical notations, backed by single metamodel, that help in describing and designing software systems, particularly software systems built using the object-oriented (OO) style" (Fowler, 2003b). ÜML provides a variety of useful capabilities to the software designer, including multiple, interrelated design views, a semiformal semantics expressed as a UML meta model, and an associated language for expressing formal logic constraints on design elements" (Medvidovic, 2002). UML, currently valid version 2.0, is de facto industrial standard. It has been proposed and developed under the Rational corporation; authors of the language came both from industry, academy and research as well. The language is considered to be informal graphical notation and hence for a long time it has been ignored by researchers at architecture level. The popularity of the language now attracts researchers to reviews of their languages and results; the trend of approximations to UML is obvious. A lot of critique has been put on the language; however, now the positive critique is the prevailing and it seems that UML is becoming the language of interest even to researchers.

ADL authors often use a reference to graphical notations as box-and-line notations. Naturally, box-and-line notations were not codified in a sense-making ways, without defined semantics. Ambiguous notations and the lack of their compilation support by computers belong to main reasons for line oriented notation used by ADLs. Although UML is still not accepted as formal language, UML can not be ever considered to be informal notation.

We must argue that UML is *unified* and *modeling* and *language*.

This means:

- unified – the language is result of unifications (consensus) among different previous notations,

- modeling – the goal of language is to support modeling, not simply describing the system in a static way,
- language – UML is not simple another graphical notation, it is a language with defined alphabet and syntactical rules.

UML is reflective language: the language is universal enough to describe itself. There is a general four-layer metamodeling architecture defined in reflection of UML. This layered architecture enables flexible definition and distinctions among concepts of model, class, profile; it enabled UML to define strict syntax described in UML and introduces built-in extensional mechanism of UML.

The meta-meta model layer defines a language for specifying the meta model layer. The meta model layer provides legal specifications in a given modeling language (the UML meta model defines legal UML specifications). The model layer carries models of specific software systems (UML profile). The user objects layer is used to construct specific instances of a given model. The model and meta model layers are the most relevant for modeling software architectures in UML (Medvidovic, 2002).

UML is very flexible language. It defines involves mechanism usable to enrich language without changing the language itself without modifying language structures. Roughly speaking, extension mechanism is a way how create new "words" of the language that can be described in terms of the language and have clear semantics and syntax. Extension mechanism has delivered UML beyond object-oriented world. The extension mechanism consists of the following:

- constraints – place added semantic restrictions on model elements; they are applicable in numerous cases, e.g. type constraints on class attribute values, constraints on the construction of associations between classes, and so on,
- tagged values – allow associations between attributes and model elements,
- stereotypes – bring groups of constraints and tagged values to be given descriptive names, signed as <<>>. Applied to model elements, they effectively create a new yet restricted form of meta class for constructing models. The semantic effect is as if the constraints and tagged values were attached directly to those elements[11].
- profiles – predefined sets of stereotypes, tagged values, constraints, and icons to support modeling in specific domains

Object Constraint Language

The power of UML for practical purposes was gained by a constraint specification language - Object Constraint Language. OCL enriches UML in a way that UML models can be handled by embedded reasoning upon particular elements of the model,

---

[11] (UML2003): *A stereotype is, in effect, a new class of metamodel element that is introduced at modeling time. It represents a subclass of an existing metamodel element with the same form (attributes and relationships) but with a different intent. Generally a stereotype represents a usage distinction. A stereotyped element may have additional constraints on it from the base metamodel class. It may also have required tagged values that add information needed by elements with the stereotype. It is expected that code generators and other tools will treat stereotyped elements specially.*

the model itself, or even the system in whole. However, tools for UML based modeling are not OCL enabled in a way to use OCL as formalism for reasoning.

The (UML, 2003) defines not just the language; it provides abstract and concrete syntax as well as detailed semantics down to denotation semantics of constraints. The power of OCL is based on simple but useful and understandable of predicate logic and set theory to constructions compliant with structures modeled by UML. OCL is typed language with a set of simple and composite types. Simple types are e.g. real, char, etc; composite are sets, bags (multisets) and collections (ordered multisets). Types are featured by standard operations (+, -, ...; union, intersection, ...) and some extra operations (select, collect, etc). Some special operators are introduced (OCLIsKindOf, OCLIsVoid, ...). Important feature is navigability across the model using roles (in the meaning of the UML).

The language is quite intuitive to read. OCL can be used for constraint declaration in the model; anyway, it can be used for other purposes, for instance reasoning about the model and comparison of different models. Examples of the OCL language usage can be found in Section 6.5.1 in declaration of constraints on stereotypes.

ADLs and UML

ADLs are compared to other languages. There exists a trend to specify what the ADL exactly is and how it differs from other languages (see Section 6.4). The UML attacks ADLs with its popularity, relative simplicity and expressiveness. This is the reason why repeatedly appear studies approximating and mapping ADLs to UML. What is the real position of UML to ADLs?

UML is a general purpose modeling language. Anyway, it fulfills non-trivial set of properties expected from ADLs (compare e.g. to Medvidovic's framework, see Section 6.4.1)[12]. Often, UML is on the list of ADLs (including some other languages probably not ADLs, e.g. Demeter[13] and ACME; e.g. refer to ADL page at SEI CMU[14]).

In (Garlan, 2000a), Garlan and Kompanek discuss upon the position of UML and ADLs in software development. Software engineering transforms requirements of stakeholders to resulting software application and in between requirements and code, there is analysis and design phase (A&D). A&D is supported by specialized languages; among these belong both ADLs and UML. In case of architecture, at least one of them is expected to be used. Anyway, the analogical problem is solved in documenting software architectures. The scenarios of choice are (depicted in Figure 6-1):

- Requirements, ADL, Code - is a way omitting UML; ADL used in this fashion must be able to fulfill all needs of analyst and designer; this way is very unlike to be used; it is suitable for integration projects
- Requirements, UML, Code - a way omitting ADL; UML is general purpose modeling language and could be suitable for *any* A&D job; fairly, although UML

---

[12] In some details, the UML is more ADL than some ADLs

[13] Demeter is a language older than ADLs, a long time considered to be ADL. It was created at Xerox PARC as a language for adaptive software systems specification and now it is considered to be the adaptive object-oriented programming base method.

[14] Architecture Description Languages: http://www.sei.cmu.edu/architecture/adl.html

is based upon object-oriented principles, it is suitable for any other methodology and method; this way is often used in software development; however, the generality of UML often allows A&D to tangle down to a spaghetti design a long time before even a line of code is written

- Requirements, ADL, UML, Code - is the longest and the most complex way; this is probable future of ADLs; ADLs are capable to capture architecture details, to provide specific reasoning upon architectures and keeps even large systems at the architecture level clean and correct; transformation to UML allows practitioners to model details without risk of complete messing up the system.

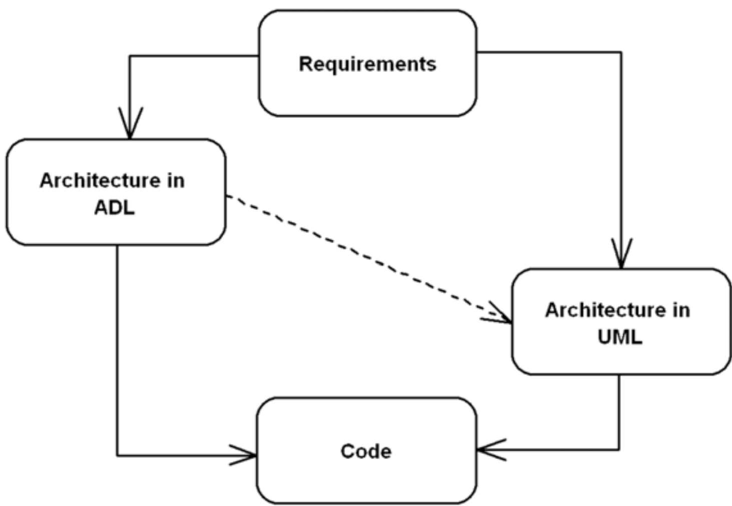


Figure 6-1. ADL and UML between requirements and code. Adopted with minor changes from (Garlan, 2000b).

By (Garlan, 2000b):

Using a more general modeling language such as UML has the advantages of providing a notation that practitioners are more likely to be familiar with, and providing a more direct link to object-oriented implementations and development tools. But general-purpose object languages suffer from the problem that the object conceptual vocabulary may not be ideally suited for representing architectural concepts, and there are likely to be fewer opportunities for automated analysis of architectural properties.

Transformations of ADLs to UML

There were some studies trying to provide transformations from ADL to UML. Transforming ADL to UML may shed more light on relation between ADL and UML. The transformation may bring, consequently, knowledge of ADLs to those practitioners who prefer UML.

We will start with (Garlan, 2000b). Garlan studied direct notation possibilities of UML to capture architecture (components, connectors and configurations). He identified five of them (examples are depicted in Appendix 6.7.5):

- types as classes, instances as objects - using a class diagram

- types as stereotypes, instances as classes - using a class diagram
- types as classes, instances as classes - using a class diagram
- components; types as stereotypes, instances as components - using a component diagram
- subsystems; subsystem is modeled by package, hence instances are packages - using component or deployment diagrams

Garlan found these mappings to be somewhat semantics mismatch between UML encoding and ADLs. He argues that the major problem of mismatch lies in vocabulary that UML's uses: the vocabulary (semantics of UML) is designed for object-oriented approach and does not fit *his* ADLs vocabulary.

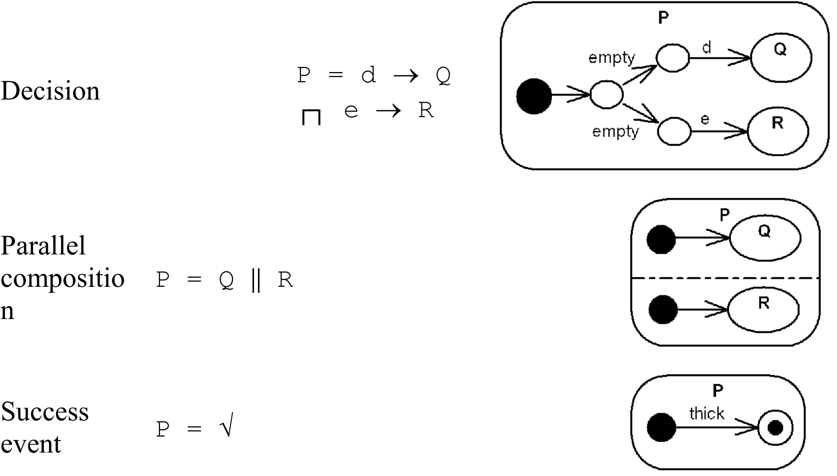
Without major changes when compared to (Garlan, 2000b), Kompanek provided an update of the mapping in (Ivers, 2004). He maps UML 2.0 features that were not considered by Garlan. UML 2.0 brought new concepts to UML, probably the most important concept from the point of view of ADLs is assembly connector (example is given in Figure 6-7, Appendix 6.7.5). Although the UML acceptance by ADL researchers grows, even Kompanek's report uses UML as documenting language, not modeling.

The other and more promising way of transformations has been proposed by Medvidovic (Medvidovic, 2002). Medvidovic studied extension mechanisms of UML to enrich models in a way that overbridges discrepancies identified by previous studies and that won't change the UML. The major goal has been to find a mechanism capable to enrich existing UML in a way that is suitable for UML compliant tools without enriching meta model and meta meta-model of the UML. Using extension mechanism of UML, he delivered semantics of chosen ADLs to UML. We will show this strategy in 6.5.1.

Wright and UML

In (Medvidovic, 2002) Medvidovic suggested usage of UML extension mechanism to enable Wright representation in UML. The Wright's CSPs capturing behavioral substance of modeled system can be represented successfully by state charts of UML and Medvidovic suggested transformation of CSP operations to state machine.

CSP Concept	CSP Notation	FSA transitions
Prefixing	$P = a \rightarrow Q$	
Alternative	$P = b \rightarrow Q$ $\square\ c \rightarrow R$	



Decision	P = d → Q ⊓ e → R	
Parallel composition	P = Q ‖ R	
Success event	P = √	

Table 6-1. CSP to state transitions of FSA.

Medvidovic suggested stereotypes to map constructions of Wright into state charts; state chart diagrams in UML model state-machines (final state automaton, FSA). Since state-machines are well-known and their theory is successfully applied in many cases, Medvidovic's method is fairly a guide for notation for software designers. Mapping of basic CSP concepts, that are adapted by Wright, to FSA is shown in Table 6-1.

Stereotypes are counterpart of UML extension mechanism. Practitioners are familiar with the notion of stereotype, hence this approach is very promising. Stereotype definition is usually based on OCL expression and tagged values.

Let us follow Medvidovic's stereotypes. Medvidovic uses tagged values and OCL to build up needed stereotypes. For behavioral specifications in Wright:

- WSMTransition for instances of meta class Transition
  - a tagged value of two cases:
    ```
    WSMtransitionType : enum { event, action }
    ```
  - "event" (`e?x`) transition is a call event only:
    ```
    self.WSMtransitionType = event implies
    (self.event->notEmpty
    and self.event.oclIsKindOf(CallEvent)
    and self.action->isEmpty)
    ```
  - "action" (`e!x`) transition is a null event and action:
    ```
    self.WSMtransitionType = action implies
    (self.event->isEmpty and self.action->size = 1)
    ```
- WrightState for instances of meta class State
  - all transitions in composite state of Wright must be WMTransitions:
    ```
    self.oclIsKindOf(CompositeState) implies
    self.transition->forAll(t - t.stereotype =
    WSMTransition)
    ```
  - restrictions to state of Wright are applied recursively downwards:
    ```
    self.oclIsKindOf(CompositeState) implies
    ```

```
(self.oclAsType(CompositeState).state ->
forAll(s - s.stereotype = WrightState))
```

- WrightStateMachine for instances of meta class StateMachine
  - WrightStateMachine consists of one composite (top) state.
  - all transition in state machine of Wright must be WSMTransitions:
    ```
    self.top.oclIsKindOf(CompositeState) implies
    self.top.transition ->
    forAll(t - t.stereotype = WSMTransition)
    ```
  - all nested states must be of WrightState:
    ```
    self.top.oclIsKindOf(CompositeState) implies
    (self.top.oclAsType(CompositeState).state ->
    forAll(s - s.stereotype = WrightState))
    ```

Except the behavioral descriptions, Wright supports component and connector interfaces (ports and roles). Medvidovic suggested usage of explicit modeling of protocols connected to interfaces by WrightStateMachine elements. Hence for interfaces in Wright:

- WrightOperation for instances of meta class Operation
  - operations in Wright are parameter-free; parameters are implicit in CSP:
    ```
    self.parameter -> isEmpty
    ```
- WrightInterface for instances of meta class Interface
  - interface is tagged value port or role:
    ```
    WrightInterfaceType : enum {port, role}
    ```
  - all operation in interface are WrightOperation:
    ```
    self.operation ->
    forAll(o - o.stereotype = WrightOperation)
    ```
  - interface is associated to exactly one state machine:
    ```
    self.stateMachine -> size = 1 and self.stateMachine ->
    forAll(sm - sm.stereotype = WrightStateMachine)
    ```
  - the machine of interface is a machine a operations of interface; each operation is associated with event call on the transitions of the machine:
    ```
    self.stateMachine.transition ->
    forAll(t - (t.event.oclIsKindOf(CallEvent)) implies
    self.operation ->
    exists(o - o = t.event.operation))
    ```

The last part of Wright that must be modeled are connectors. Connector is a set of roles describing behavior of components connected and a special process denoted glue that "glues" interaction of roles. Medvidovic suggested to use UML meta class Class to model Wright connectors which provide multiple interfaces (roles) and participate with other classes (components). Connector state is compound of its internal states thus it may have no direct attributes. For Wright's connectors:

- WrightGlue for instances of meta class Operation

- glue consists of single machine:
```
self.stateMachine -> size = 1 and self.stateMachine ->
forAll(sm - sm.stereotype = WrightStateMachine)
```

- WrightConnector for instances of meta class Class

  - connector implements at least one interface that is a role:
```
self.interface -> size >= 1 and self.interface ->
forAll(i - i.stereotype = WrightInterface and
i.WrightInterfaceType = role)
```

  - connector contains a single glue:
```
self.operation -> size = 1 and self.operation ->
forAll(o - o.stereotype = WrightGlue)
```

  - trigger events of the glue are operations with no data or with input data belonging to different interface elements:
```
self.operation.stateMachine.transition ->
forAll(t - (t.event.oclIsKindOf(CallEvent)) implies
self.interface.operation ->
exists(o - o = t.event.operation))
```

  - similar to the recent constraint is constraint on output data belonging to different interface elements, which are actions of the glue

  - The semantics of a Wright connector can be described as the parallel interaction of its glue and roles:
```
let glueop = self.operation ->
select(o - o.stereotype = WrightGlue) in
self.stateMachine -> size = 1 and self.stateMachine ->
forAll(sm - sm.top.oclIsKindOf(CompositeState) implies
(sm.top.isConcurrent = true and sm.top.state ->
size = 1 + self.interface -> size and sm.top.state ->
exists (gs gs = glueop.stateMachine.top) and
self.interface -> forAll(i - sm.top.state ->
exists (rs - rs = i.stateMachine.top)))
```

  - a connector must have at least one instance in running system:
```
self.allInstances->size >= 1
```

Components in Wright are modeled by a set of ports describing interface of component, and a computation. Medvidovic suggested WrightComponent stereotype which is simple analogy of WrightConnector.

Finally, Wright specifies configuration using attachments. Attachments can be covered using component and connectors; hence for attachments:

- WrightAttachment for instances of meta class Association

  - attachments connect two elements only:
```
self.associationEnd->size = 2
```

  - attached are connector and component:
```
let ends = self.associationEnd in
ends[1].multiplicity.min = 1 and
ends[1].multiplicity.max = 1 and
ends[2].multiplicity.min = 1 and
```

```
ends[2].multiplicity.max = 1 and
((ends[1].interface.stereotype = WrightInterface and
ends[1].interface.WrightInterfaceType = port and
ends[2].interface.stereotype = WrightInterface and
ends[2].interface.WrightInterfaceType = role) or
(ends[2].interface.stereotype = WrightInterface and
ends[2].interface.WrightInterfaceType = port and
ends[1].interface.stereotype = WrightInterface and
ends[1].interface.WrightInterfaceType = role)
```

- WrightArchitecture for instances of meta class Model
  - all classes of the model must be Wright elements:
    ```
    self.modelElement ->
    select(me - me.oclIsKindOf(Class)) ->
    forAll(c - (c.stereotype = WrightComponent or
    c.stereotype = WrightConnector)
    ```
  - all associations of the model must be Wright elements:
    ```
    self.modelElement ->
    select(me - me.oclIsKindOf(Association)) ->
    forAll(a - a.stereotype = WrightAttachment)
    ```
  - each port connects with exactly one role and vice versa:
    ```
    let comps = self.modelElement ->
    select(e - e.stereotype = WrightInterface) in
    comps.associationEnd->size <= 1
    ```
  - Wright connectors and components participate in Wright associations, only; analogy to last two constraints in WrightConnector

Although Wright is not a complicated language, its mapping to UML is pretty complex at level of profiling UML. Almost every constraint in stereotypes is simple. So final checker of UML which is able to use the profile for Wright; the checker will be forced a simple task in consistency checking. Finally, appendix Section 6.7.4 gives an example of partial usage of the mapping.

6.5.2 XML Metadata Interchange

In context of ACME project (6.4.3), the man involved in industrial praxis must be asking a question why we need another architecture interchange. The another in this case refers to XML Metadata Interchange (XMI) which is rather format than language.

XMI is XML-based format (exactly, XMI is XML instance with implied semantics) designed to UML. Accepting the fact that architecture can be mapped to UML arises the question why to use ACME. We will try to find an answer.

The XMI is format based upon Meta Object Facility (MOF). MOF is formally a technology by OMG for defining metadata and representing it as CORBA objects. Here, metadata refer to general term of data that describes information. The MOF supports any kind of metadata that can be described using Object Modeling techniques. This metadata may describe any aspect of a system and the information it contains, and may describe it to any level of detail and rigor depending on the metadata requirements (OMG, 2002). The term of model is, in general, used to denote a description of something, typically something in the real world. In MOF, model is any collection of

metadata that is related in a way of common implied semantics, abstract syntax or metadata related to other.

MOF can be related to UML meta levels, in fact directly by one-one mapping as MOF defines levels:

- M3 - meta-metamodel, the MOF model
- M2 - metamodel (meta-metadata), UML metamodel level
- M1 - model (metadata), UML model
- M0 - data

XMI as language captures through MOF models, primarily in UML. The XMI is designed for OO models interchange.

The XMI is useful for UML model interchange. As it can capture full information upon UML model, the XMI format can be used for model interchange. Although, till mapping of architectures, especially ADLs, to UML is still considerably one-way, the XMI format can be applied to any domain that is MOF mappable.

The difference between ACME and XMI is enormous: ACME is language accompanied by the abilities of formal checking and reasoning upon the architecture and model. ACME is specialized language that, in co-operation with other ADLs, can be used to share, interchange architectures. It enables many useful operations upon the model. On the other hand, XMI is simple interchange format, rather encoding than language.

6.6 Conclusion

Within a couple pages we have browsed current knowledge in areas of software architecture emphasizing the formal approach to. The notion of formalization, although not explicitly defined by computer scientists, is the key to unambiguous reasoning on software architectures. The influence of such attitude covers not just architecture as research topic; it impacts consequently practical methods of e.g. costly software integration, large scaled systems design, development and testing, etc. Software architectures, if appropriately formalized and "armed" with suitable reasoning methods, can be the key for maintenance problems of large systems - this is a way that current trends in real software systems evolves; the larger the system, the higher costs and the more space for errors, bugs and wrong decisions.

6.6.1 State-of-art

The lowest level, that was of our particular interest, was the object structure. Studying object structures emphasizes key notions of architecture; as object structure is a special *micro*architecture. There is no valid proof of this theorem; anyway, it gives further inspirations and questions to software architectures. Object structures are intensively studied using different approaches. The most used one is a family of object algebras. The research of them is under double attack: one is designing of new - hopefully more optimal to some criteria - algebra languages, the other is un-stopping stream of critiques on existing languages. The object structure can be studied on itself as a static structure. Studies of communication and relation inside the structure should not be

considered as dynamical description; this is covered by lower approaches to code analysis. The dynamical point of view refers to behavioral features and qualities that appear when the structure is being transformed to other by some operations. Such dynamics is the one remaining software architectures and is interesting in consistency checking, integration, refactoring, refinement process, white-box testing, etc. Current formalisms on object structures provide very little support to such dynamics and should be the point of further research.

Object structures and their graphical notation led to designing a Unified Modeling Languages. The UML has matured to its current version 2.0 and provides much more than has been influenced by object structures; implicit object orientation is not "the must" of language usage, the UML holds the "unified modeling" facilities and hence must be considered as general modeling language. The UML has a strictly defined syntax, it has a considerably well defined semantics. Accompanied by Object Constraint Language and using built-in extension mechanism, the UML can be customized in legal way (both syntax and semantics) to many applications. This is the position when we should ask whether UML can be the language for software architectures.

Software architectures has been studied for more than 15 years intensively and their research still continues. Software architectures are the probable key to future of software engineering. They provide a higher level of abstraction omitting details of lower levels; the notions used here are components, connectors and configurations. Practicians of industry use these notions in many different ways; the styles of architectures are those often (*mis*)used as everybody are conscious about client-and-server, pipe-and-filter, blackboard, etc. On the other hand, the usage and *real* understanding of these "architectures" leads to many discrepancies and mismatches. There has been a need for common, formally defined vocabulary: architectural style, architecture, view, viewtype. And there has been a need for formal descriptions of systems. That descriptions should provide consistency checking, reasoning on architecture, modeling, dynamics of changing systems. This is a task for Architecture Description Languages.

ADLs appeared in very natural way when software systems scales attained limits of maintainability by lover levels of detail. A good motivation can be found in legacy systems integration. ADLs are considered a formal languages that describe software architecture and often provide a reasoning virtues. We have studied ADL Wright that is based on Hoare's theory of communicating sequential processes that allows strong reasoning on architecture, checks consistency of the system, enables detecting deadlock possibilities of such systems, etc. Wright is an example of ADL that is capable to reason on behavioral features of the modeled system. The other language we decided to get in touch with has been ACME. ACME is not typical ADL as it conveys architecture interchange facilities among other ADLs. However, the ACME is not universal ADLs mediator as ADLs differ in so many presumptions, constraints, obligations, that there can not be proposed language that freely interchanges architectures among all ADLs. ACME cover wide representative set of ADLs. The question that appears at ACME is whether we can not use industrial OMG standard XML Metamodel Interchange.

ADLs have origin different from the origin of UML; even the main purpose of languages differs. However, recent research in area of software architectures tends to studies on mapping ADLs to UML. The reason for such mapping is not the proof of the

differences between UML and ADLs; the aim is to use existing tools supporting UML, bring ADL virtue to UML community. As a side effect, mapping ADLs to UML improves understanding what ADL is, what is it good for and what are differences in among them; consequently, the mappings provide alternative architecture interchange mechanism. However, UML mappings are still too "raw" to fulfill those expectations.

UML on itself is not ADL. Anyway, UML can be profiled using its built-in extension mechanism to support approaches advocated by particular ADLs. UML on its own supports architecture modeling, even in multiple ways. The difference between ADLs and UML stands on dynamics modeling and reasoning on it. UML provides mechanisms to capture and design architecture, UML can even make a syntactical matches. However, UML, for now, can not support further reasoning on architecture. On the other hand, UML is suitable for heterogeneous components, can abstract from details of communication and concurrently can model it. UML provides features of modeling that have never been even in a scope of ADLs; UML is therefore a potential "glue" that can be used as a common formalism in process of software development including the views of architectures.

6.6.2 Open problems

The brief study to software architectures formalization touched borders and edges of the area. Let us pin out those of particular interests:

- object structure dynamics - is an area that is still not sufficiently studied; formal studies to dynamics can support and deepen the possibilities of dynamical modifications of object structure; object-oriented community puts a lot of effort to structure refinements, describing patterns (of any kind), refactoring, etc.
- current mapping of ADLs to UML are rather documenting and does not use all possibilities of UML - mappings to UML have been strongly studied at level of UML 1.1 - 1.4, partially 2.0 with a little interest dedicated to other diagrams but class and state chart; well studied approach includes stereotyping, as we have shown; anyway, there is still a gap that has not been properly studied; in addition, we believe that UML is applicable to simple reasoning about the system and as so, UML can be used as "fully qualified" ADL
- dynamics of architectures - even more to object structures, ADLs provide scanty possibilities to model higher dynamics on architectures, e.g. combining two systems into a one (although it can be done in a way of abstraction)

This survey of software architectures formalization introduced a way that ADLs reason on architectures. These results could be used in object structure descriptions to describe dynamics of those systems; as a side effect, better understanding and possibilities to architecture dynamics can be gained.

6.7 Appendixes

6.7.1 OCL: a static language

OCL is a static language in the meaning that has been proposed in Section 6.2.1. This means that OCL is unable to describe dynamic change of object structure. Following example comes from (Navarčik, 2005) where the OCL is used as alternative way of

proof upon theorem about special refinement to object structure. The quoted study copies with improving of existing object structure using derived subclass for rarely used attribute (study 4.1).

Long time used systems often turn in discrepancy in model and practical usage of the system. We won't discuss the reason why this happens. Anyway, discrepancies lead to inefficiencies of many types. One of them is bad memory or space management. Within this case, we will follow Figure 6-7.

Let $K \in < 0, 1 >$ is given ratio, e.g. 0.7. K is the ratio of unacceptable memory usage efficiency. When memory usage ratio gets above the K value (the ratio of unused but allocated memory to total allocated memory) the ineffective state must be fixed up. In OCL description:

```
context c:Class inv:
(c.allInstances()->select(ax.oclIsUndefined())->size()) /
(c.allInstances()->size()) > K
```

To fix the inefficiency, let us derive subclass `Cx` bearing the rarely used attribute `ax`. Then:

```
context c1:Cx inv:
(c1.allInstances()->select(ax.oclIsUndefined())->
  size()) = 0
```

The solution is based upon Fowler's refactoring rules (Fowler, 2002) 1.16 and 1.15.

Lemma. Let $m = |\{c_i \in \text{oid}(Class)\text{: } c_i.ax = \text{NULL }\}|$, and let $n = |\{c_i \in \text{oid}(Class)\text{: } c_i.ax \neq \text{NULL }\}|$. Then direct spatial effectivity of spatial consumption (memory) is $[m/(m+n)]$. Unused memory ratio is then $1 - [m/(m+n)]$.

Theorem. Let oid($Class$) = $\{c_1, c_2, ..., c_n\}$ are instances of class *Class*. Let each instance $c_i \in$ oid($Class$) contains attribute *ax*. Let for relevant majority of instances the *ax* value is unspecified. Then, considering the spatial consumption, more effective structure can be obtained by deriving $Cx < Class$ subclass, moving the ax from Class to *Cx* and instantiating all objects with specified ax value as instances of *Cx*.

Proof. We won't provide the proof of lemma. The proof of theorem is driven by the solution. Both structures are able to capture the same modeled reality. Initial situation is described as follows:

```
context c:Class inv:
let
  m: Integer = c.allInstances() ->
     select(ax.oclIsUndefined()) -> size()
  n: Integer = c.allInstances() -> size()
in
  if n>0 then m/n > K
```

Refactoring the structure, for the final model holds:

```
context c:Class, cx: Cx inv:
let
```

```
  m: Integer = cx.allInstances() ->
     select(ax.oclIsUndefined()) -> size()
  n: Integer = cx.allInstances() -> size()
  o: Integer = c.allInstances() -> size()
in
  m = 0
and
  if n>0 then (n-m)/n = 1
and
  if o>=(1/K) then n/o < K
  -- this binds to former state with floor(1/K) undefined
```

Resulting structure is less space (memory) consuming as far as the ax attribute, formally unused, has been moved down to subclass and all instances of the subclass fill-up the attribute.

QED

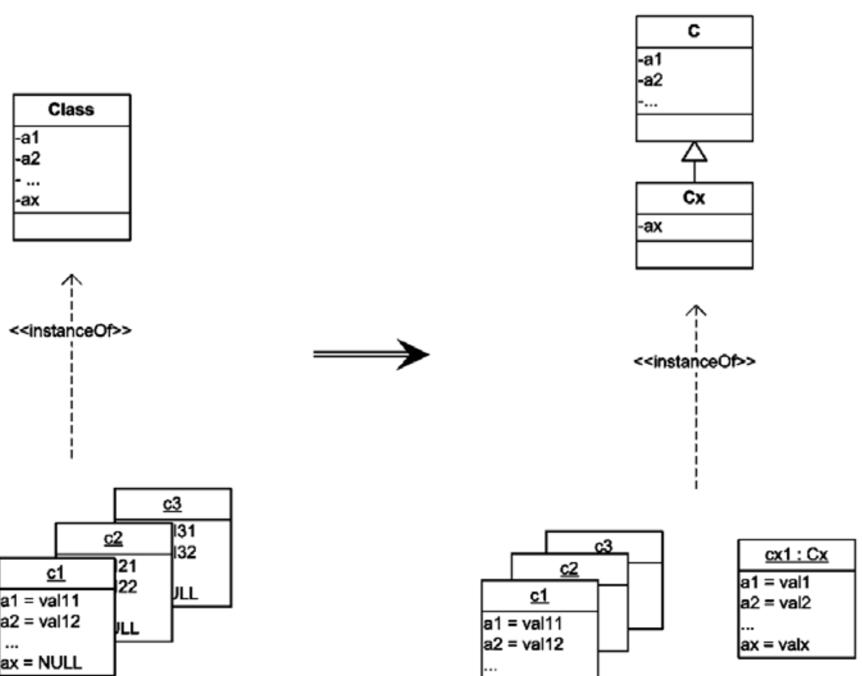


Figure 6-2. Derived subclass for rarely used attribute.

The proof uses OCL notation. Anyway, OCL can be replaced by proof written in mathematical fashion. Alternative proof is:

Proof. In oid(*Class*) = $\{c_1, c_2, ..., c_n\}$, the number of allocations for ax attribute is n. The majority of them is unused, let us denote the number as m, $n \geq m > 0$. Derive the subclass $Cx < Class$ and move the *ax* attribute to it such that each instance cx_i has ax: $t_c \neq$ NULL. Then only $n - m$ objects (instances of *Cx*) allocates the attribute. Hence resulting structure is more effective in memory consumption as $n - m < n$ for $m > 0$.

QED

6.7.2 Survey of other ADLs

The ADLs are (more or less) formal languages used to represent software architecture. As architecture becomes a dominating theme in large system development and acquisition, methods for unambiguously specifying an architecture become indispensable (Clements, 1996). The number of ADLs is still growing. We have already shown Wright as a multi-purpose ADL. Brief survey, based mainly upon (Medvidovic, 2000):

Aesop supports specification of architectures in specific styles; extremely supports styles; distinguishes input and output ports; behavior preserving sub-typing; connector semantics optionally based on Wright; ports may be attached to roles and vice versa; graphical visualization

Demeter probably the oldest language; almost ADL converted to method called adaptive object-oriented method

C2 architectures of highly-distributed, evolvable, and dynamic systems; semantics based on 1st order logic; heterogeneous sub-typing; connector semantics partially specified via message filters; graphical visualization

Darwin designed for highly-distributed message-passing systems whose dynamism is guided by strict formal underpinnings; supports parameterization; includes in-line specification and graphical notation; π-calculus based semantics; does not explicitly model connectors - uses bindings instead

MetaH specialized language for design of real-time avionics software; implementation restricting components; emphasizes non-functional properties as real-time schedulability, reliability, security analysis; provides also graphical notation

Rapide modeling and simulation of the dynamic behavior described by an architecture; allows architectural designs to be simulated; has tools for analysis of simulation results; interfaces called constituents; supports parameterization; semantics upon partially ordered event sets (posets); structural sub-typing; textual detailed notation; graphical notation, too

SADL provides a formal basis for architectures refinement across levels of detail; with no semantics; refinements supported via pattern maps; connector semantics based upon axioms in the constraint language; refinement maps enable correct refinements across styles

UniCon glue centered language for interconnection of heterogeneous components and connectors using common interaction protocols; interfaces are players; predefined set of types; semantics on event traces; schedulability analysis; implementation constraining components; supports only predefined connector and component types, supports component wrappers; players attached to roles and vice versa, only

Weaves data-flow architectures, characterized by high-volume of data and real-time requirements on its processing; implementation constraining; semantics based upon partial ordering over input and output objects; connectors are transport services; graphical specification

xArch XML based ADL distinguishing run-time and design-time; uses current state of XML defining its schemas; does not expect own tools - simply uses off-the-shelf XML tools;

6.7.3 ACME

Example 6-2 (adopted from (Garlan, 2000b) shows a simple client-server system example in ACME. To authors of ACME belong both David Garlan and Robert Allen. As ACME is a language dedicated to architecture interchange, one should compare the example with the alike system written in Wright, 6.4.2.

```
System simple_cs = {
    Component client = { Port sendRequest }
    Component server = { Port receiveRequest }
    Connector rpc = { Roles {caller, callee} }
  Attachments : {
    client.sendRequest to rpc.caller ;
    server.receiveRequest to rpc.callee }
}
```

Example 6-2. Simple client-server system in ACME.

6.7.4 Wright to UML

(Medvidovic, 2002) suggested mapping of Wright to UML by state machines. The mapping uses UML extension mechanism by defining a set of stereotypes. Resulting profile is capable to capture structures of Wright by UML. The approach is described in Section 6.5.1.

Example 6-3 shows a simple chunk of code in Wright that defines connector of pipe type usable in architectural style of filter-and-pipe. Using Medvidovic's approach, the connector can be modeled in UML by single state-chart, that is depicted in Figure 6-3.

```
connector  Pipe =
  role Writer = write → Writer  ⊓ close → √
  role Reader =
let ExitOnly = close → √
in let DoRead = (read → Reader  □ readeof → ExitOnly)
in DoRead  ⊓ ExitOnly
  glue = let ReadOnly = Reader.read → ReadOnly  □
    Reader.readeof → Reader.close → √   □ Reader.close → √
    in let WriteOnly = Writer.write → WriteOnly  □ Writer.close
→ √
    in Writer.write → glue  □ Reader.read → glue  □

Writer.close →
      Reader.ReadOnly  □ Reader.close → Writer.WriteOnly
```

Example 6-3. Wright connector specification (pipe). Adopted from (Medvidovic, 2002). Resulting UML state chart is depicted in Figure 6-3.

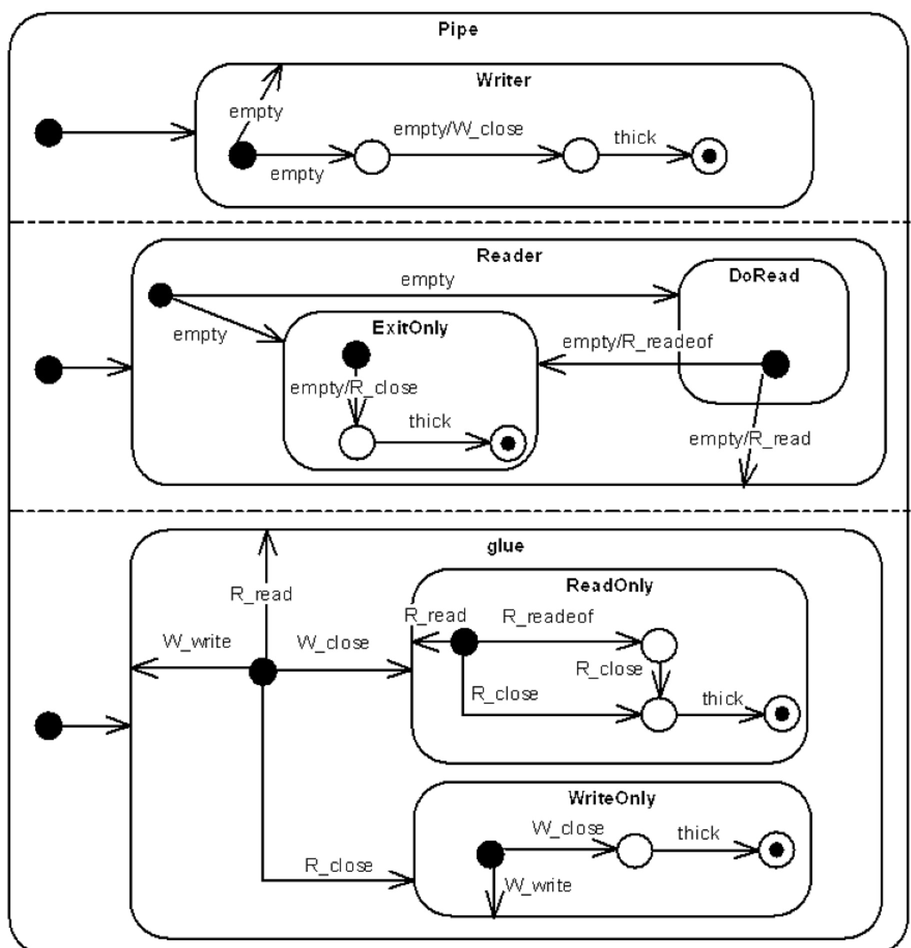


Figure 6-3. UML state chart capturing connector specification from Example 6-3. Stereotypes are omitted..

6.7.5 UML architecture notation

In Section 6.5.1 we have sketched some methods of mapping architecture to UML. This appendix section provides 4 of 5 principal approaches studied by David Garlan (Garlan, 2000a). The same ideas can be found with minor refinements in (Clements, 2002).

As an simple example, we adapted Garlan's model: the model of simple pipe-and-filter system that provides a specialized information maintenance process. The example is based upon a simple Unix script calling complex utilities; it is general architecture example and can be applied in much larger scale, indeed.

All figures within this appendix are adapted from (Garlan, 2000a) but prior notice. The diagrams may seem at first look similar; there is a strong semantical difference from both the point of UML and architecture description. We omitted the case of subsystem modeling as it is well-covered by component approach.

All figures compound of two parts (divided by “swimline”). The top parts of the pictures provides a style key; the bottoms depict a system example.

Types as classes, instances as objects

Mapping architecture this way seems to be very natural. However, limitations on classes and objects put this approach on the border of interest. Classes describe the conceptual vocabulary of a system just as component and connector types form the conceptual vocabulary of an architectural description in a particular style. Additionally, the relation between classes and instances is similar to the relationship between architectural types and their instances (Garlan, 2000a).

In this approach, Garlan suggested stereotypes usage. These stereotypes are not clearly declared and can not be considered as a part of language profile. The example of approach is shown in Figure 6-4.

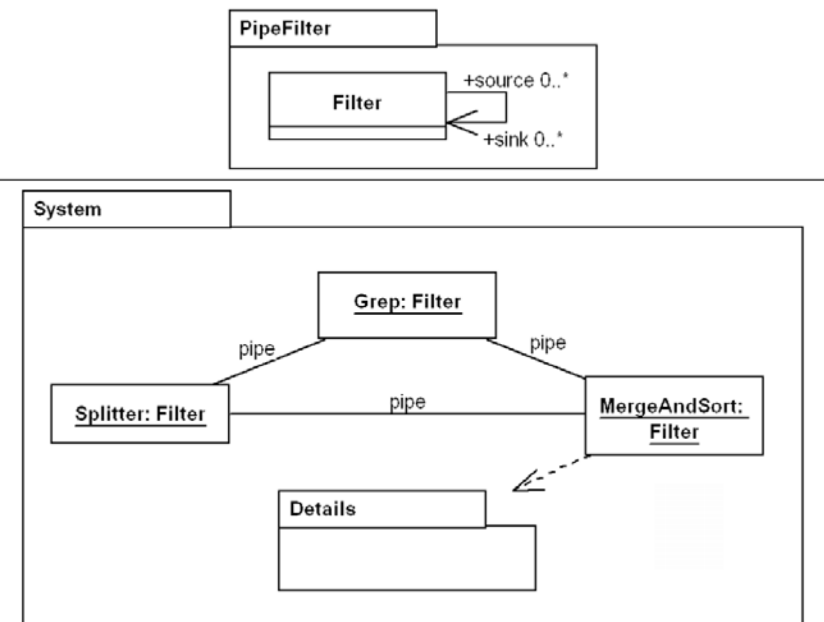


Figure 6-4. Modeling architecture strategy based on classes and objects.

Types as stereotypes, instances as classes

Profiling UML by stereotypes is very promising strategy. This way has been chosen by e.g. (Medvidovic, 2002), too; see Section 6.5.1. A component instance is represented as a class with a stereotype. Using this approach, architectural concepts become distinct from the built-in UML concepts, and in principal, a UML-based modeling environment can be extended to support the visualization and analysis of new architectural types within a style and enforce design constraints captured in OCL. The example of this solution is shown in Figure 6-5.

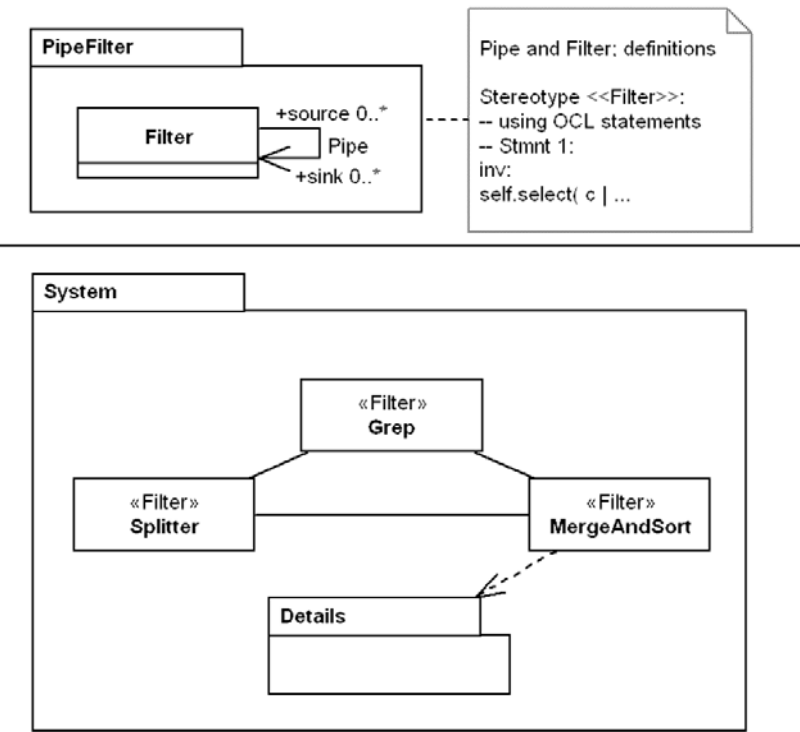


Figure 6-5. Modeling architecture strategy based on stereotypes and classes.

Types as classes, instances as classes

The other class-based approach is to represent component types as (*meta*)classes and component instances as classes. By representing both component types and instances as classes, we have the full set of UML features to describe both component types and instances. We can also capture patterns at both the type (as part of a description of an architectural style) and instance level, supporting the description of a dynamic architecture whose structure evolves at run-time (Garlan, 2000a). The example of approach is depicted in Figure 6-6.

Components

Although (Garlan, 2000a) consider this strategy to be a little bit confusing and obviously insists on modeling using class diagrams, recent changes to UML introducing UML 2.0 make this strategy pretty suitable and understandable. Existing UML 2.0 vocabulary better covers notions of architectures; the key to suitability is in assembly relation. The assembly relation is analogy to attachments in Wright or ACME.

Anyway, Garlan stated that UML includes a "component" modeling element which is used (in praxis) to describe implementation artifacts of a system and their deployment. A component diagram is often used to depict the topology of a system at a high level of granularity and plays a similar function, although at the implementation level, as an architectural description of a system.

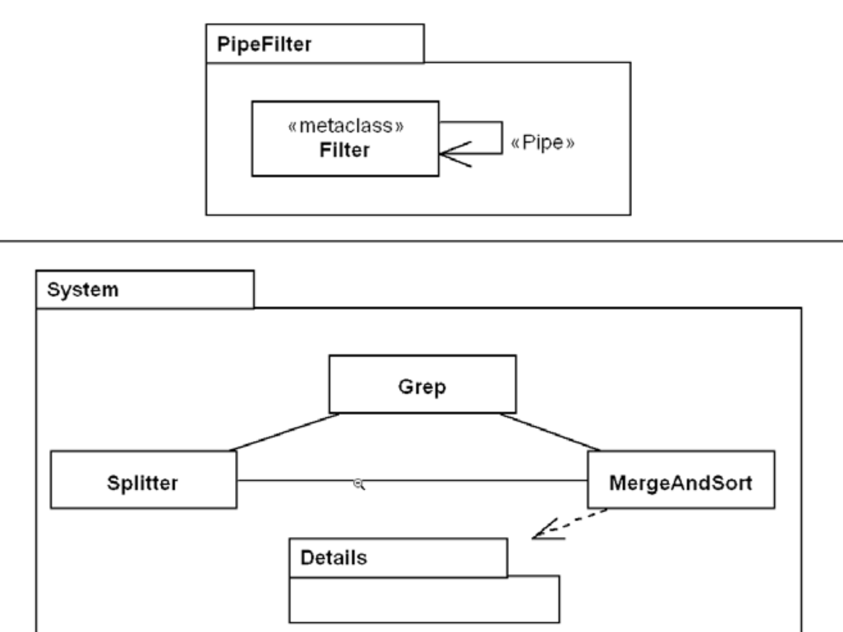
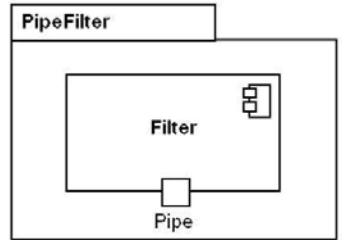


Figure 6-6. Modeling architecture strategy based on (meta)classes and classes.

The Figure 6-7 is strongly modified to suit to and to use UML 2.0.



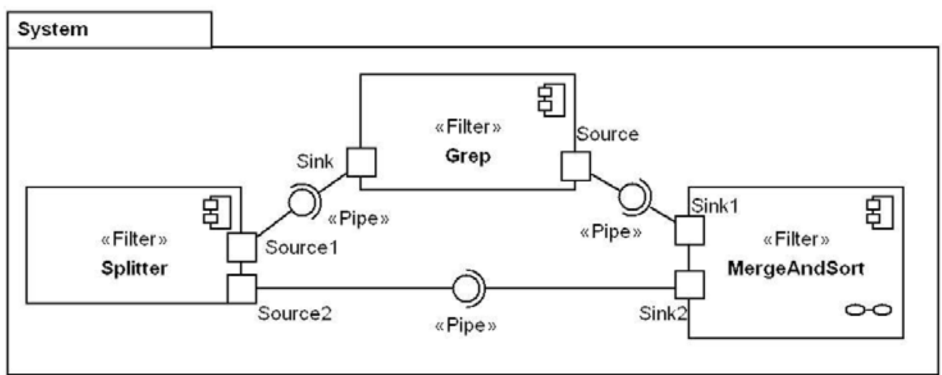


Figure 6-7. Modeling architecture strategy based on components.

References

Allen, R. – Garlan, D. (1997) A Formal Basis for Architectural Connection. In: *ACM Transactions on Software Engineering and Methodology*, vol. 6, No. 3, July 1997, pp. 213—249

Allen, R. – Garlan, D. (1998a) A Formal Basis for Architectural Connection. In: *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3, July 1997.

Allen, R. – Douence, R. – Garlan, D. (1998b) Specifying and Analyzing Dynamic Software Architectures. In: *Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering*, Lisabon, Portugal. March 1998.

Clements, P. (1996) A Survey of Architecture Description Languages. In: *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD '96)*, IEEE.

Clements, P., et al. (2002) Documenting Software Architectures. Views and beyond. Addison-Wesley, Boston, MA, USA. ISBN 0-20270372-6

Dashofy, E.M. – van der Hoek, A. – Taylor, R.N. (2001) A Highly-Extensible, XML-Based Architecture Description Language. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA.01)*. IEEE.

Dini, P. – Belkhelladi, A. – Melo, W.L. (1997) Formalizing Software Architectures: An Industrial Experience. In: *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. Springer-Verlag, Zurich, Switzerland.

Fowler, M. (2003a) Who needs an architect? In: *IEEE Software*, 2003/03, pp. 2—4.

Fowler, M. (2003b) UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd edition. Addison-Wesley.

Garlan, D. – Kompanek, A.J. (2000a) Reconciling the Needs of Architectural Description with Object-Modeling Notations. In: *Proceedings of the Third International Conference on the Unified Modeling Language – << UML >> 2000*, October, 2000, York, UK.

Garlan, D. – Monroe, R.T. – Wile, D. (2000b) Acme: Architectural Description of Component-Based Systems. In: *Foundations of Component-Based Systems*. Cambridge University Press, pp. 47—68.

Ivers, et al. (2004) Documenting Component and Connector Views with UML 2.0. Technical report CMU/SEI-2004-TR-008, ESC-TR-2004-008. Carnegie Mellon, SEI, 2004. Available at: http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tr008.pdf (August 2005)

Lichtner, K. – Alencar, P. – Cowan, D. (2000) A Framework for Software Architecture Verification. In: *Australian Software Engineering Conference 2000*, pp. 149—158

Lopes, A. – Wermelinger, M. – Fiadeiro, J.L. (2003) Higher-Order Architectural Connectors. In: *ACM Transactions on Software Engineering and Methodology*, Vol. 12, No. 1, January 2003, pp. 64—104.

MEDVIDOVIC, N. – TAYLOR, R.N. (2000) A Classification and Comparison Framework for Software Architecture Description Languages. In: *IEEE Transactions on Software Engineering*, vol. 26, No. 1, January 2000

MEDVIDOVIC, N. – ROSENBLUM, D.S. – REDMILES, D.F. – ROBBINS, J.E. (2002) Modeling Software Architectures in the Unified Modeling Language. In: *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No. 1, January 2002, pp. 2—57.

NAVARČIK, M. – POLÁŠEK, I. (2005) Object Model Notation. In: *Proc. of 8th Intl. Conference on Information Systems Implementation and Modeling (ISIM '05)*, Ostrava, Czech Rep., April 2005, pp. 211—218.

OBJECT MANAGEMENT GROUP (2002) OMG XML Metadata Interchange (XMI) Specification. Ver. 1.2, January 2002

OBJECT MANAGEMENT GROUP (2003a) UML 2.0 Superstructure Specification: Final Adopted Specification. 2003. Available at: http://www.omg.org/docs/ptc/03-08-02.pdf (August 2005)

OBJECT MANAGEMENT GROUP (2003b) XML Metadata Interchange (XMI) Specification. Ver. 2.0, May 2003

PROCTER, P., ET AL. (1995) Cambridge International Dictionary of English. Cambridge University Press, Cambridge, United Kingdom. ISBN 0-521-48421-9

ROP (1998) Rational Unified Process 5.1. Rational Software Corporation, 1998. Hypertext commercial document.

SHAW, M. – GARLAN, D. (1996) Software Architectures. Prespectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, NJ, USA. ISBN 0-13-182957-2

UML 2.0 OCL Specification OMG Adopted Specification ptc/03-10-14, October, 2003

ZHANG, B. – DING, K. – LI, J. (2002) An XML-message Based Architecture Description Language and Architectural Mismatch Checking. In: *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC.01)*, IEEE.

Referred sources

FOWLER, M., ET AL.(2002) Refactoring: Improving the Design of Existing Code, Addison Wesley.

GARLAN, D. (2001) Software Architecture. In: *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., USA.

HNATKOWSKA, B., ET AL. (2004) Structural Refinement of Class Diagrams. In: *Proc. of 7th Intl. Conference on Information Systems Implementation and Modeling ISIM '04*, Ostrava, April 2004.

HOARE, C.A.R. (1985) Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs, NJ.

KRUCHTEN, P. (1995) The 4+1 View Model of Architecture. In: *IEEE Software 12*, vol. 6, November 1995. pp. 42—50.

7 MULTIAGENTOVÉ SYSTÉMY

Trend postupného distribuovania výpočtov neobišiel ani umelú inteligenciu. Myšlienka jedného superinteligentného centrálneho systému, ktorý samostatne rieši úlohy, získala konkurenta už viac než pred tretinou storočia. Približne v tejto dobe sa začali objavovať systémy založené na spolupráci viacerých expertov na riešení spoločnej úlohy. Dodnes pribudlo mnoho ďalších spôsobov ako v umelej inteligencii využiť princíp distribuovaných výpočtov a vykonávania, a to pre dosiahnutie kvantitatívnych, ale aj kvalitatívnych zlepšení.

Distribuované výpočty a vykonávanie v multiagentových systémoch však so sebou prinášajú aj rad problémov. Je potrebné určiť, ako problém rozdeliť na jednoduchšie časti. Je potrebné zabezpečiť prenesenie jednotlivých podproblémov na miesto ich riešenia a nakoniec zozbierať a spojiť čiastkové výsledky. Je tiež potrebné koordinovať činnosť jednotlivých agentov tam kde sú ich činnosti od seba vzájomne závislé a synchronizovať prístup k spoločným zdrojom.

Napriek tomu, že multiagentové systémy ako vedná disciplína sú staršie ako štvrť storočia, definície jednotlivých konceptov v rôznych zdrojoch nie sú jednotné a často ani jednoznačné. Vymedzenie jednotlivých konceptov uvedených v tejto kapitole vychádza predovšetkým z monografií (Ferber, 1999; Návrat, 2002; Mařík, 1993; Mařík, 2003a; Mařík, 2003b), ale aj z ďalších zdrojov a v neposlednej miere z diskusií s kolegami ktorým by som chcel touto cestou poďakovať.

7.1 Agent a multiagentový systém

Práca pojednávajúca o multiagentových systémoch by pravdepodobne nebola kompletná bez uvedenia definície pojmov „agent“ a „multiagentový systém“. Tieto pojmy sú v literatúre často definované pomerne vágne a definície v jednotlivých zdrojoch sa často líšia. Na druhej strane, v rôznych definíciách je možné nájsť spoločné črty, o ktoré je možné oprieť sa. V tejto práci vychádzam z definícií, ktoré vo svojej monografii použil Ferber (Ferber, 1999).

Multiagentový systém (MAS) je systém pozostávajúci z

- prostredia (P),
- množiny objektov (O), ktoré sú umiestnené v prostredí P,
- množiny agentov (A), ktoré predstavujú aktívnu časť objektov ($A \subseteq O$),

- množiny vzťahov (V), ktoré spájajú jednotlivé objekty (a teda aj agenty)
- množiny operácií (Op), ktoré môžu agenty použiť na zmenu stavu objektov a prostredia,
- zákonov prostredia (Z), ktoré určujú reakciu prostredia na pokus o vykonanie niektorej z operácií Op, prípadne zmeny v prostredí bez zásahu agentov.

Teda: MAS = (P, O, A, V, Op, Z)

Prostredie (P) môže mať veľa rôznych podôb. Často je to prostredie s euklidovským priestorom, v ktorom sú umiestnené jednotlivé objekty. V prípade robotických systémov je prostredím reálny svet alebo jeho časť – stavový priestor, v ktorom sa môže robot pohybovať. Prostredie však môže vyzerať aj inak, napríklad to môže byť priestor riešení istého problému, prípadne môže byť prostredie totožné s prázdnou množinou ($P=\emptyset$).

Objekty (O) majú spravidla svoje umiestnenie v prostredí, čiže v každom časovom okamihu je možné priradiť objektu pozíciu v prostredí. Objekty môžu byť pasívne alebo aktívne. Pasívne objekty nevykonávajú zmeny v prostredí, naopak aktívne objekty – agenty (A), majú možnosť aktívne zasahovať do prostredia a ovplyvňovať stav iných objektov použitím dostupných operátorov (Op). Na objekty (aktívne aj pasívne) pôsobia rôzne vonkajšie vplyvy, ktoré menia ich stav, môžu ich vytvárať a rušiť. Vonkajšie vplyvy prichádzajú od agentov (ako dôsledok aplikovania operácií) alebo od prostredia na základe zákonov prostredia (Z).

Zákony prostredia (Z) reprezentujú prostredie, v ktorom sa pokúšame riešiť problémy pomocou MAS. V multirobotických systémoch sú to fyzikálne zákony, v prípade systému pre riadenie toku dokumentov môže zákony prostredia určovať fungovanie lokálnej siete, po ktorej sa dokumenty prenášajú. V prostrediach s diskrétnou zmenou stavu možno zákony prostredia vyjadriť vo forme zobrazenia aktuálneho stavu prostredia a objektov (S) a aktivovaných operácií ($OpA \in Op$) na budúci stav prostredia (S') $Z: (S, OpA^{*}) \rightarrow S'$. Všeobecnejším prípadom sú prostredia so spojitou zmenou stavu, v ktorých k zmene nedochádza v diskrétnych okamihoch ale počas časových intervalov. Typickým príkladom takéhoto prostredia je reálne prostredie. V takýchto prostrediach si nevystačíme iba so zobrazením aktuálneho stavu na stav nasledujúci, ale musíme hľadať iný spôsob ako opísať zákony prostredia, napríklad diferenciálne rovnice. Sú dokonca situácie, kedy nedokážeme presne určiť nasledujúci stav prostredia. Dôvod môže byť nepredvídateľnosť (nedeterminizmus) prostredia a jeho odozvy na aplikovanie operácií alebo jednoducho nemožnosť zistenia budúceho stavu. Príkladom pre prostredie kde nie je možné presne určiť budúci stav skôr než nastane je vzájomné gravitačné pôsobenie 3 a viac telies (problém 3 telies, problém n telies (Poincaré, 1892)), ktorý je analytickým spôsobom neriešiteľný (hoci je riešiteľný numericky, ale s obmedzenou presnosťou).

Všeobecná definícia agenta môže vyzerať takto. Agent:

- je fyzický alebo virtuálny celok,
- je schopný konať v prostredí,
- je schopný vnímať svoje prostredie,
- má iba čiastočnú (alebo žiadnu) reprezentáciu svojho prostredia,

- je autonómny,
- má vlastné zdroje.

Na to aby sme mohli niečo považovať za agenta, musí splniť všetky body tejto definície. Agent musí tvoriť jeden celok, ktorý je jasne oddeliteľný od svojho prostredia, od ostatných agentov a iných objektov. To, či je agent fyzickým celkom umiestneným v reálnom svete (napríklad robot) alebo je virtuálnym celkom bez fyzickej existencie (softvérový agent) z pohľadu definície nehrá úlohu.

Agent musí byť schopný konať, teda musí dokázať riadeným spôsobom ovplyvňovať svoje prostredie, objekty a agenty v prostredí, vrátane samého seba. Príkladom akcií ktoré môže agent v prostredí vykonávať je vlastný presun, vytváranie, modifikácia a rušenie objektov, zmena celkového stavu prostredia. Za konanie sa vo všeobecnosti považuje aj komunikácia medzi agentmi. Prostredníctvom komunikácie dokáže agent priamo ovplyvňovať vnútorný stav ostatných agentov.

Agent musí dokázať vnímať svoje prostredie, či už priamo rozpoznávaním objektov a stavu prostredia alebo iba na základe jednoduchých vnemov ako je napríklad intenzita svetla či iného signálu v okolí senzora.

Na základe vnemov si agent môže vytvárať vnútornú reprezentáciu prostredia, či už na úrovni symbolov alebo na subsymbolickej úrovni. Dôležité je, že agent nemôže mať úplnú reprezentáciu, ale má reprezentáciu iba časti prostredia, ktorá dokonca pre správne fungovanie agenta ani nemusí byť správna a presná. „Vševediaci“ agent, ktorý má úplnú reprezentáciu prostredia, prestáva byť agentom, a stáva sa súčasťou prostredia.

Pravdepodobne najdôležitejšou vlastnosťou agenta je autonómnosť. Táto vlastnosť odlišuje agenta od podobných, pasívnych, konceptov ako sú objekty, moduly či procedúry. Autonómnosť znamená, že agent nie je priamo ovládaný z vonku (iným agentom alebo človekom). Dalo by sa povedať, že to čo poháňa agenta v jeho činnosti je agent sám.

Posledný bod definície, nutnosť vlastniť zdroje, je istým spôsobom spojený s autonómnosťou. Nejde totiž iba o „typické“ zdroje ako je vlastníctvo rôznych objektov a informácií. Každý agent musí vlastniť minimálne zdroje, ktoré potrebuje pre svoju základnú činnosť, ako je energia, výpočtový čas, prípadne reálny čas – čas ktorý má k dispozícii na splnenie úlohy.

Pre zhrnutie: Agent je autonómny celok, ktorý vníma svoje prostredie, môže si vytvárať jeho čiastočnú reprezentáciu a riadeným spôsobom ho ovplyvňuje, pri čom používa zdroje, ktoré vlastní.

7.2 Základné charakteristiky agentov

Podľa definície z kapitoly 7.1, agent je autonómny celok, ktorý vníma svoje prostredie, môže si vytvárať jeho čiastočnú reprezentáciu a riadeným spôsobom ho ovplyvňuje, pri čom používa zdroje, ktoré vlastní.

Napriek tomu, že definícia pomerne jasne vymedzuje hranicu medzi pojmom agent a ostatnými, príbuznými, pojmami (objekt, modul, hráč), ponecháva stále veľký priestor, v rámci ktorého sa môžeme pohybovať. V skutočnosti, ak sa pozrieme na

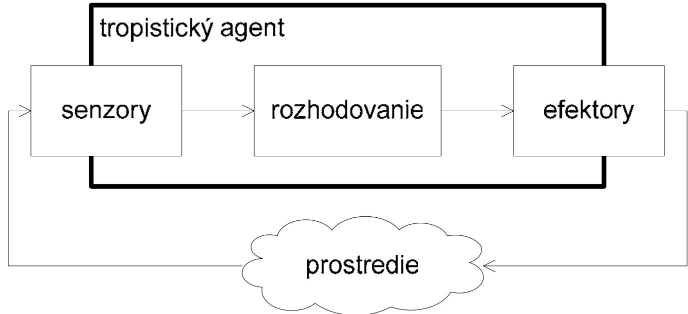
existujúce multiagentové systémy, nájdeme obrovské množstvo podôb aké môžu agenty mať. Medzi najjednoduchšími agentmi, ktorí tvoria bunky bunkových automatov (Gerhardt, 1995) a pohybujú sa na hranici pojmu agent a medzi zložitými agentmi vykonávajúcimi komplexné výpočty a symbolické odvodzovanie je priepastný rozdiel. Napriek tomu je možné všetky tieto autonómne jednotky združiť pod pojmom „agent“.

V tejto kapitole sa pokúsim poskytnúť prehľad základných vlastností, ktoré charakterizujú agenty. Na základe týchto vlastností a ich kombinácií je možné pomerne dobre a jednoznačne odlíšiť základné skupiny, typy, agentov. Rozdelenie, ktoré je tu poskytnuté, nie je vzhľadom na veľký rozsah oblasti multiagentových systémov úplné. Uvedené vlastnosti a kategórie vždy neopisujú oddelené triedy agentov a multiagentových systémov, ale sú často iba orientačnými bodmi vymedzujúcimi hraničné prípady, medzi ktorými je možné voľne sa pohybovať. Vlastnosti agentov a multiagentových systémov, ktoré sa nevzťahujú priamo na agenta ako jednotlivca ale na celú organizáciu agentov, sú oddelené v ďalšej kapitole.

7.2.1 Schopnosť pamätať si

Na základe schopnosti pamätať si minulosť a uchovávať informácie vo svojom vnútri rozdeľujeme agenty na 2 základné skupiny: histeretické agenty ktoré túto schopnosť majú a tropistické agenty bez schopnosti pamätať si.

Tropistický agent, agent bez pamäte, čisto reaktívny agent



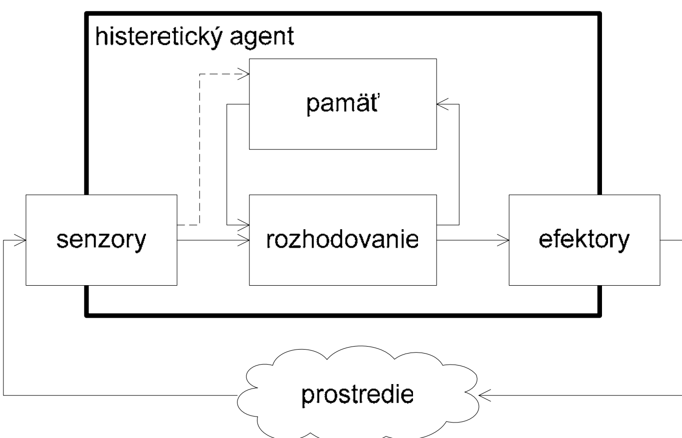
Obrázok 7-1. Štruktúra tropistického agenta (agenta bez pamäte). Rozhodovanie tropistického agenta priamo mapuje vnemy na akcie.

Tropistický agent nemá vnútornú pamäť, či už symbolickú alebo subsymbolickú. Agent bez pamäte teda nemá žiadny vnútorný stav, pomocou ktorého by dokázal ukladať informácie. Koná vždy iba na základe aktuálneho vnemu (obrázok 7-1). Tropistický agent teda iba reaguje na aktuálny stav prostredia, bez uvažovania predchádzajúcich stavov prostredia. Zvykne sa tiež označovať ako čisto reaktívny agent.

Za princípom tropistických agentov stojí myšlienka, že najlepšou a najpresnejšou reprezentáciou prostredia je prostredie samotné (Brooks, 1990). Agent si nevyhnutne nepotrebuje ukladať informácie vo svojom vnútri, stačí mu pozorovať prostredie a na základe vnemu (teda aktuálneho stavu prostredia) zvoliť vhodnú akciu. Jediný spôsob, akým dokáže tropistický agent uchovávať informácie, je uchovávať ich priamo v prostredí, prostredníctvom zmien ktoré agent v prostredí robí.

Mohlo by sa zdať, že použitie takéhoto jednoduchého agenta bez možnosti pamätať si predchádzajúce stavy prostredia a uchovávať si svoj vlastný stav bude značne obmedzené. Tento predpoklad však nie je celkom správny. Aj s takýmito obmedzeniami je možné vytvoriť systémy, ktoré vykazujú veľmi komplexné správanie. Príkladom môže byť robot Herbert (Brooks, 1988), ktorý dokáže zbierať prázdne plechovky, zatiaľ čo sa autonómne pohybuje a vyhýba sa prekážkam. Iným príkladom sú simulácie kolónií hmyzu, ktorý sa dokáže orientovať v prostredí na základe ferómónových značiek, ktoré jedinci zanechávajú v prostredí, teda používajú prostredie ako svoju pamäť. Napriek tomu však agenty bez pamäte svoje obmedzenia majú.

Histeretický agent, agent s pamäťou



Obrázok 7-2. Štruktúra histeretického agenta (agenta s pamäťou). Rozhodovanie histeretického agenta okrem vnemov využíva aj pamäť.

Napriek možnostiam ktoré poskytujú agenty bez pamäte, niektoré vlastnosti dokážeme dosiahnuť iba s použitím vnútornej pamäte a nahrádzať vnútornú pamäť značkami v prostredí nemusí byť výhodné. Pamäť je napríklad nevyhnutné použiť pre získavanie skúseností (učenie sa), komplexné výpočty a odvodzovanie alebo v prípade že je prostredie málo pozorovateľné a potrebujeme sa rozhodovať na základe postupnosti viacerých vnemov.

Histeretický agent má vnútorný stav, je schopný pamätať si, či už na subsymbolickej alebo symbolickej úrovni. Agent je schopný pamätať si svoj stav a aj predchádzajúce stavy, rovnako ako si môže zapamätať predchádzajúce vnemy a históriu stavov prostredia. Rozhodnutie agenta nie je potom robené len na základe aktuálneho vnemu prostredia, ale aj na základe vnemov predošlých (obrázok 7-2).

7.2.2 Vnútorná reprezentácia a spôsob rozhodovania

Na základe spôsobu rozhodovania je možné rozdeliť agenty na 2 základné skupiny: reaktívne agenty a kognitívne agenty. Vo všeobecnosti sú tieto dva pojmy chápané a definované veľmi vágne. Reaktívne agenty sú vo všeobecnosti chápané ako jednoduché, niekedy až primitívne, ktoré „iba“ reagujú na podnety zo svojho okolia. Naopak, kognitívnemu agentu sa prisudzuje schopnosť „rozmýšľať“, teda vykonávať

symbolické odvodzovanie, plánovať svoju činnosť, dohodnúť sa na ďalšom postupe s inými agentmi, prípadne do istej miery predpovedať budúcnosť.

Kde teda môžeme nakresliť pomyseľnú hranicu medzi reaktívnymi a kognitívnymi agentmi? Vžitá definícia reaktívneho agenta: „Reaktívny agent je ten, ktorý iba reaguje na aktuálnu situáciu," je pomerne nevhodná. Uvedomme si napríklad, že aj plánovo uvažujúci agent zostavením plánu svojej ďalšej činnosti tiež iba reaguje na svoju aktuálnu situáciu. To by znamenalo, že podľa tejto definície sú všetky agenty reaktívne. Jednou možnosťou je stotožniť pojmy „tropistický" a „reaktívny". Reaktívny agent by potom bol ten, ktorý reaguje iba na aktuálnu situáciu (rozdiel oproti predchádzajúcej definícii je v posunutí slova „iba"). Tropistické agenty sa skutočne zvyknú označovať pojmom „čisto reaktívne". Toto by však znamenalo že napríklad jednoduchý agent, ktorý má iba 2 vnútorné stavy (povedzme zapnutý/vypnutý), by sme museli považovať za kognitívny, prípadne by sme museli nájsť ďalší pojem, ktorý by vyplnil vzniknutú medzeru medzi pojmami „reaktívny" a „kognitívny".

Prirodzená hranica, ktorú je možné použiť na oddelenie reaktívnych a kognitívnych agentov, je spôsob rozhodovania o ďalšom postupe agenta. V prípade že sa agent rozhoduje na symbolickej úrovni, môžeme ho označiť za kognitívneho. Ak sa rozhodovanie odohráva na subsymbolickej úrovni, môžeme agenta nazvať reaktívnym. Namiesto pojmu „reaktívny agent" je však v tomto prípade vhodnejšie použiť pojem „subkognitívny agent".

Subkognitívny agent, reaktívny agent

Rozhodovanie subkognitívneho agenta sa deje na subsymbolickej úrovni, čiže na určenie akcie, ktorú agent vykoná nie sú použité postupy, ktoré manipulujú so symbolmi. Príkladom je rozhodovanie na základe jednoduchých pravidiel mapujúcich vstupy agenta na výstupy, neurónové siete alebo stavový stroj (deterministický alebo nedeterministický).

Subkognitívny agent nemá symbolickú reprezentáciu prostredia, svojho stavu ani stavu ostatných objektov a agentov v systéme. Všetky informácie týkajúce sa vnútorného stavu agenta a jeho reprezentácie prostredia sú na subsymbolickej úrovni, napríklad vnútorný stav rekurentnej neurónovej siete. Subkognitívny agent môže mať prostredie reprezentované aj symbolmi, ale tieto symboly ďalej spracováva subsymbolickými metódami, teda jeho vnútornú reprezentáciu je možné považovať za subsymbolickú keďže sa s významom symbolov nepracuje.

Oproti prístupom „klasickej" umelej inteligencie založenej na symbolickom odvodzovaní a symbolickej reprezentácii sveta dnes stojí pomerne rozsiahla reaktívna škola, ktorej základné myšlienky vyjadril Brooks (Brooks, 1990; Brooks, 1991a; Brooks, 1991b) nasledovne:

- Inteligentné správanie je možné dosiahnuť bez explicitnej reprezentácie akú navrhuje symbolická umelá inteligencia,
- Inteligentné správanie je možné dosiahnuť bez explicitného abstraktného odvodzovania akú navrhuje symbolická umelá inteligencia,
- Inteligencia je emergentná vlastnosť komplexných systémov istého typu.

Skutočne, veľa úspešných systémov je založených práve na subsymbolickom prístupe. Výhodou subsymbolického prístupu je väčšinou menšia náročnosť na výpočtové

prostriedky a pamäť, často aj podstatne väčšia rýchlosť odozvy na vonkajší podnet alebo zmenu. V dynamicky sa meniacich a zle zanalyzovaných prostrediach je veľmi ťažké zostaviť plán činnosti na dlhší čas dopredu a subsymbolické prístupy tu často dosahujú lepšie výsledky a väčšiu robustnosť pri reakcii na neočakávané situácie. Na druhej strane ak napríklad chceme sledovať (a pochopiť) proces rozhodovania, alebo potrebujeme vytvoriť presný a optimálny plán činnosti (v dobre zanalyzovanej doméne), je lepšie sa spoľahnúť na symbolické prístupy.

Čo sa týka výsledkov symbolického a subsymbolického prístupu, sú tieto často porovnateľné, respektíve sú oblasti aplikácie kde dosahujú lepšie výsledky subsymbolické prístupy a sú oblasti kde je výhodnejšie využiť symbolický prístup.

Špeciálny prípad reaktívneho agenta je čisto reaktívny agent, teda tropistický agent, agent bez pamäte. Keďže nemá žiadnu pamäť, nemá ani žiadnu reprezentáciu svojho prostredia a objektov v ňom. (Podrobnejšie o tropistických agentoch v kapitole 7.2.1)

Kognitívny agent

Kognitívne agenty sú založené na myšlienkach „klasickej" umelej inteligencie. Na rozdiel od subkognitívnych agentov má kognitívny agent k dispozícii explicitnú symbolickú vnútornú reprezentáciu vonkajšieho sveta a objektov v ňom. Nad touto reprezentáciou potom kognitívny agent dokáže vykonávať symbolické odvodzovanie a manipuláciu so symbolmi, či už s použitím odvodzovania nad rôznymi logikami (výroková, predikátová, temporálna, ...), rôznych foriem plánovania, sémantických sietí alebo iných prístupov.

Medzi kognitívnymi agentmi je možné ďalej vyčleniť skupinu plánovo uvažujúcich agentov. Plánovo uvažujúci agent využíva niektorý z postupov plánovania (niektoré sú opísané v kapitole 7.4). Má schopnosť zvažovať alternatívy ďalšieho postupu a vytvárať plány, na základe ktorých potom koná po určité časové obdobie. Typická činnosť plánovo uvažujúceho agenta pozostáva z troch opakujúcich sa základných krokov: Vyhodnoť aktuálnu situáciu, vytvor plán ďalšieho postupu, vykonaj plán. V prípade multiagentového systému môže byť súčasťou vytvorenia plánu aj komunikácia a vyjednávanie s ďalšími agentmi. Veľa systémov tiež umožňuje priebežne prehodnocovať a modifikovať vykonávaný plán na základe nových informácií a zmien v prostredí.

Plánovo uvažujúcim agentom sa tiež pripisuje čiastočná schopnosť predvídať budúcnosť. Skutočne, pri vytváraní plánu ďalšieho postupu je potrebné aspoň do určitej miery predvídať budúce správanie prostredia, objektov a agentov v ňom. Rovnako, vytvorením plánu agent deklaruje ako by podľa neho mala vyzerať budúcnosť na základe myšlienky, že najlepší spôsob ako predvídať budúcnosť je vytvárať ju.

Sociálny agent

Sociálny agent je špeciálny prípad kognitívneho agenta rozšíreného o explicitnú reprezentáciu ostatných agentov. Okrem reprezentácie prostredia má k dispozícii aj reprezentácie ostatných agentov, a to nie len reprezentáciu agentov ako objektov, ale aj explicitne vyjadrené modely ich správania. Sociálny agent teda môže usudzovať a vytvárať predpoklady aj o správaní a možnom budúcom stave ostatných agentoch, čo mu umožní efektívnejšie vykonávať svoju činnosť. V prípade homogénnych agentov (agentov s rovnakou štruktúrou) je možné predpovedať správanie iného agenta

jednoducho tak, že sa na vstup rozhodovacieho mechanizmu (spoločného pre všetky agenty) privedie predpokladaný stav tohto agenta. V prípade heterogénnych agentov je potrebné vytvoriť model, ktorý aproximuje správanie druhého agenta.

7.2.3 Vyjadrenie cieľa

Každý agent má nejaký cieľ. Bez vlastného cieľa by nesplĺňal jednu zo základných požiadaviek definície agenta: autonómnosť. Rozdiel je však v tom akým spôsobom je cieľ vyjadrený, či je neoddeliteľnou súčasťou agenta alebo pochádza z prostredia a je dosahovaný iba reakciami agenta na podnety z prostredia. V praxi je niekedy ťažké rozlíšiť tieto dve situácie, pretože väčšinou časť cieľov prichádza z prostredia a časť má svoj pôvod vo vnútri agenta.

Telenomický agent, agent so zámerom

Telenomický agent má vo svojom vnútri explicitne vyjadrený cieľ, ktorý chce dosiahnuť, či už priamo ako súčasť bázy znalostí alebo podmienkami na cieľový stav, do ktorého sa agent snaží dostať. Cieľ môže byť súčasťou agenta od jeho vzniku, ale agent môže svoje ciele získať aj počas svojej existencie napríklad ako výsledok dekompozície hlavného cieľa na podciele alebo komunikáciou s iným agentom. Dôležitá charakteristika je aj perzistencia, trvácnosť cieľov. Cieľ môže pretrvávať až do času keď je splnený alebo sa ukáže jeho nesplniteľnosť. Cohen a Levesque (Cohen, 1986) takéhoto agenta, ktorý nedokáže zmeniť svoj cieľ, nazývajú fanatický. Na druhej strane je niekedy vhodné prehodnotiť svoje ciele s možnosťou zvoliť alternatívny cieľ, ktorý môže priniesť menší zisk za cenu menšieho rizika.

Kognitívny telenomický agent, teda agent ktorý pracuje na symbolickej úrovni na základe explicitne vyjadreného cieľa, sa nazýva agent so zámerom alebo racionálny agent. Takýto agent sa snaží nájsť „rozumné" spôsoby ako dosiahnuť svoj cieľ. Do tejto skupiny patrí väčšina kognitívnych agentov.

V prípade subkognitívneho (reaktívneho) agenta je cieľ vyjadrený ako motivačný mechanizmus, ktorý núti agenta k splneniu jeho úlohy. Takéhoto agenta môžeme nazvať agent riadený pudmi (angl. *drive-based agent*).

Agent riadený reflexami

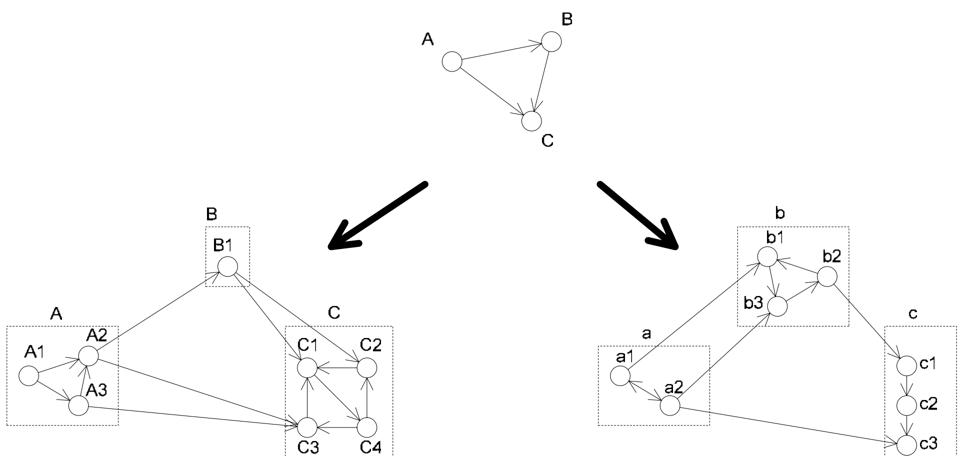
Agent riadený reflexami nemá vlastný explicitný cieľ, jeho činnosť je daná reakciami na aktuálny stav agenta a prostredia alebo reakciami na podnety z prostredia a od ostatných agentov. Ciele jednotlivých agentov sú v tomto prípade plne určené ich prostredím alebo štruktúrou multiagentového systému.

7.3 Organizácie agentov

Ako bolo spomenuté skôr (kapitola 7.1), multiagentový systém pozostáva z prostredia (P), množiny objektov (O), množiny agentov (A), množiny vzťahov medzi objektmi (V), množiny operácií (Op) a zákonov prostredia (Z). Predchádzajúca kapitola sa venovala hlavne vlastnostiam individuálnych agentov. Cieľom tejto kapitoly je zamerať sa na vzťahy medzi agentmi a na ich štruktúru. Vzájomné vzťahy a ich štruktúra sú tým, čo odlišuje multiagentový systém od skupiny agentov.

Analýza vzťahov medzi agentmi čerpá mnohé poznatky zo sociológie a naopak, sociológia bola obohatená poznatkami získanými štúdiom multiagentových systémov. Z oboru sociológie pochádza aj definícia jedného z hlavných pojmov, ktorými sa zaoberajú multiagentové systémy: Organizácia je usporiadanie vzťahov medzi komponentmi a jedincami, ktoré vytvára jeden celok (systém) s vlastnosťami, ktoré sa nevyskytujú na úrovni komponentov a jedincov (Ferber, 1999). Existujú aj iné definície organizácie, napríklad Romelaer definuje organizáciu ako množinu ľudí, ktorí medzi sebou majú pravidelné a predvídateľné vzťahy (Romelaer, 2002). Mintzberg definuje organizáciu ako spôsob rozdeľovania a koordinácie práce medzi organizačné jednotky (časti organizácie) (Mintzberg, 1979). Vo všetkých týchto definíciách hrajú hlavnú úlohu jedinci (či už ľudia, agenty alebo ich zoskupenia) a vzájomné vzťahy medzi nimi.

Organizácia je tvorená vzťahmi medzi jej časťami. Zároveň však organizácia umožňuje vzťahom vznikať a existovať. Reálne organizácie, či už tie, ktorými sa zaoberá sociológia alebo tie zložené z agentov, sú v drvivej väčšine dynamické. Vzťahy v organizácii vznikajú, zanikajú a menia svoj charakter, do organizácie prichádzajú noví jedinci, pohybujú sa v nej a odchádzajú. Na pozadí týchto premien však ostáva nezmenená statická časť organizácie – jej štruktúra.



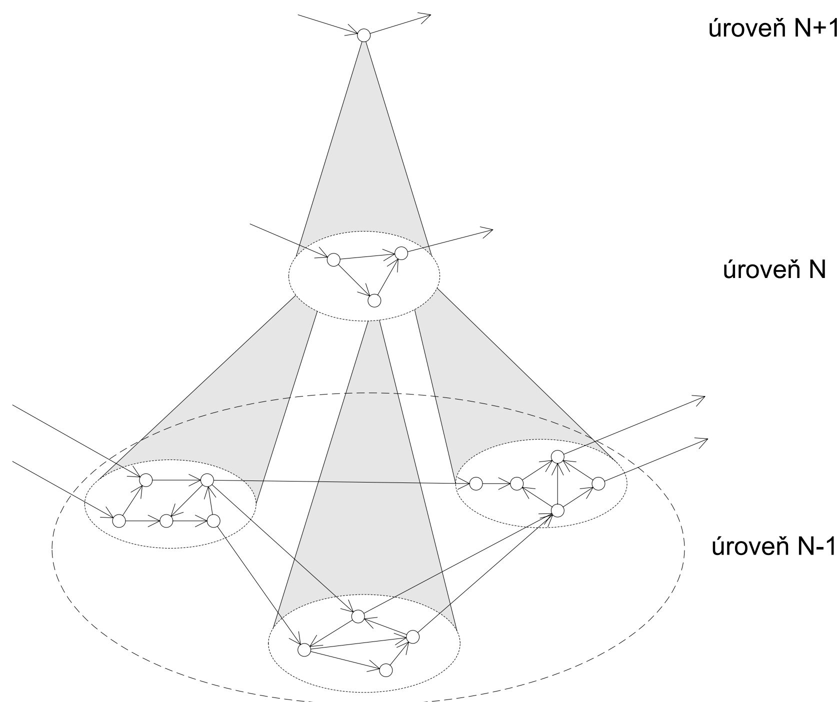
Obrázok 7-3: Príklad organizačnej štruktúry (hore) a dvoch konkrétnych organizácií (dole), ktoré sú jej inštanciami.

Štruktúra organizácie charakterizuje spôsob, akým sú jednotlivé časti organizácie poprepájané, určuje hlavné časti organizácie a obmedzuje vzťahy medzi nimi. Zároveň vytvára rámec, v ktorom môžu vznikať konkrétne organizácie ako inštancie danej organizačnej štruktúry. Príklad organizačnej štruktúry s dvomi konkrétnymi organizáciami, ktoré sú jej inštanciou je na obrázku 7-3.

7.3.1 Holarchia – organizácia organizácií

Pojem organizácie môžeme aplikovať na dvoch rôznych úrovniach: na úrovni multiagentového systému, ktorý je organizáciou agentov, ale aj na úrovni agenta, ktorý predstavuje organizáciu svojich častí. Vzniká tu akási dualita, ktorá umožňuje pozerať sa na každý systém zároveň ako na zoskupenie menších jednotiek, jeho častí, ale tiež ako na časť zoskupenia na vyššej hierarchickej úrovni (obrázok 7-4) Táto dualita

umožňuje chápať systémy ako hierarchické usporiadanie subsystémov na stále nižšej úrovni. Príkladom môže byť ľudská spoločnosť, ktorá sa skladá z rôznych spoločenstiev ľudí, tie sú zložené z jednotlivých ľudí, ľudia z telesných orgánov, orgány z buniek, bunky z molekúl, molekuly z atómov a subatomárnych častíc. Jednotky na každej úrovni sú samostatné systémy, ktoré sú ovplyvňované interakciami s ostatnými systémami na svojej úrovni a zároveň aj interakciami svojich subsystémov.



Obrázok 7-4. Príklad organizácie s viacerými úrovňami. Časti organizácie na úrovni N sú súčasne organizáciami na úrovni N-1. Organizácia na úrovni N je samostatnou jednotkou tvoriacou časť organizácie na úrovni N+1.

Na zvýraznenie tejto duality sa používa pojem „holon“ (Köstler, 1967), ktorý vznikol zložením gréckych slov „holos“ (celok) a „on“ (časť). Holon je systém, ktorý je sám o sebe celok, ale zároveň tvorí časť väčšieho celku. Hierarchia zložená z holonov sa potom nazýva holarchia alebo holonická organizácia. Pojmy holon a holarchia sa používajú hlavne v oblasti riadenia priemyselných procesov, koncepty stojace za týmito pojmami sa však používajú aj v iných aplikáciách multiagentových systémov (často pod inými názvami ako napríklad viacúrovňová organizácia).

Koncept holarchie (vnorených organizácií) je možné veľmi úspešne a prirodzeným spôsobom využiť na vytváranie zložitých systémov na základe skladania elementárnych častí do stále komplexnejších celkov alebo počas dekompozície systému na menšie funkčné celky. Keďže je holon uzavretý celok, umožňuje využitie holarchie väčšiu flexibilitu v riešení úloh, pretože spôsob akým holon rieši svoju úlohu je uzavretý v jeho vnútri a môže byť nahradený ekvivalentným postupom s inými špecifickými vlastnosťami bez zásadného ovplyvnenia zvyšku organizácie.

7.3.2 Dimenzie organizácie

Funkcie organizácie môžeme rozdeliť na základe 5 dimenzií (Ferber, 1999):

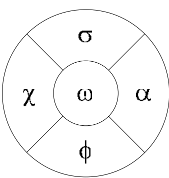
Fyzická dimenzia (ϕ) sa zaoberá vnútornou štruktúrou organizácie, jej implementáciou a personálnymi zdrojmi. Personálne zdroje v tomto prípade tvoria jednotlivé súčasti organizácie. Fyzická dimenzia určuje, aké sú súčasti organizácie a aké sú vzťahy medzi nimi.

Sociálna dimenzia (σ) určuje miesto a úlohu organizácie v nadradenej organizácii. Sociálna dimenzia zabezpečuje to, aby bola organizácia schopná reagovať na potreby organizácie na vyššej úrovni a zabezpečiť tak fungovanie organizácie vyššej úrovne, do ktorej patrí.

Dimenzia vzťahov (α) zabezpečuje interakciu organizácie s inými organizáciami na tej istej úrovni. Zaoberá sa komunikáciou a koordináciou organizácií a vzájomným správaním jednotlivých organizácií na tej istej úrovni.

Dimenzia prostredia(χ) je spojená so schopnosťou organizácie vnímať svoje prostredie a konať v ňom. Táto dimenzia zahŕňa všetky schopnosti organizácie ovplyvňovať prostredie a vykonávať v ňom svoju činnosť.

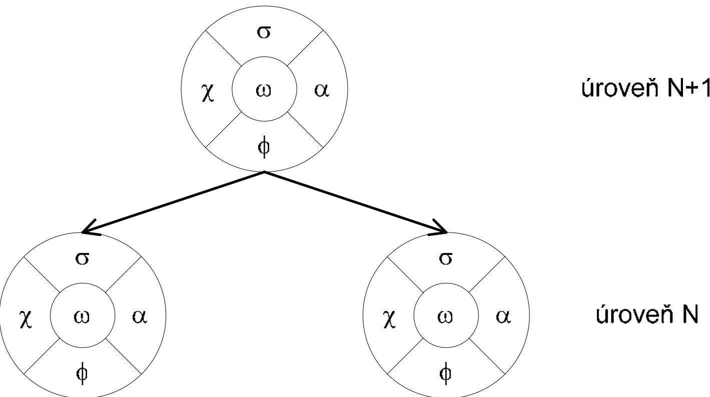
Osobná dimenzia (ω) súvisí s organizáciou ako samostatným celkom, predstavuje reprezentáciu samej seba, schopnosť učiť sa a získavať znalosti, zachovávanie vlastnej štruktúry či schopnosť meniť seba samú. Je tiež zodpovedná za motiváciu organizácie – jej ciele a snahu o sebazáchovu.



Obrázok 7-5. Päť rozmerov organizácie: fyzická dimenzia (ϕ), sociálna dimenzia (σ), dimenzia vzťahov (α), dimenzia prostredia(χ) a osobná dimenzia (ω).

Obrázok 7-5 zobrazuje spomenutých päť dimenzií organizácie usporiadaných pozdĺž dvoch osí. Horizontálna os predstavuje interakciu s organizáciami a prostredím na tej istej úrovni. Vertikálna os predstavuje na jednej strane vzťah organizácie k svojim komponentom a na druhej strane vzťah k organizácii na vyššej úrovni, ktorej súčasť tvorí. Uprostred je osobná dimenzia, ktorá predstavuje samotnú organizáciu. Keď skúmame organizáciu na viacerých úrovniach, zistíme, že fyzická dimenzia na vyššej úrovni korešponduje so sociálnou dimenziou na úrovni nižšej (obrázok 7-6). To, čo pre organizáciu predstavuje štruktúra jej častí, zároveň pre jednotlivé časti predstavuje sociálnu štruktúru.

Hoci týchto 5 dimenzií organizácie slúži skôr ako podklad pre analýzu organizácie a jej funkcií, modifikácie tohto modelu sa používajú priamo ako základ štruktúry niektorých organizácií, príkladom sú organizácie založené na štruktúre structure-in-5 (Kolp, 2001).



Obrázok 7-6: Fyzická dimenzia organizácie na vyššej úrovni je priamo spojená so sociálnou dimenziou organizácií na nižšej úrovni.

7.3.3 Vzťahy medzi agentmi

Vzťahy a ich štruktúra sú jedným z dvoch základných prvkov každej organizácie (druhým sú entity, ktoré organizáciu tvoria a medzi ktorými tieto vzťahy sú). Táto kapitola sa pokúsi dať odpoveď na otázku prečo vlastne vzťahy vznikajú a aké sú ich základné typy z pohľadu štruktúry organizácie.

Dôvody pre vznik vzťahov

Konkrétne dôvody pre interakciu agentov (komponentov organizácie) môžu byť rôzne, vždy však súvisia s cieľmi agentov, vzťahom agentov k zdrojom, ktoré majú k dispozícii a so schopnosťami agentov. Kombinácia týchto troch aspektov potom určí, aké budú základné typy vzťahov medzi agentmi.

Kompatibilita cieľov. Asi najdôležitejším dôvodom interakcie agentov sú ciele, ktoré sa snažia dosiahnuť. Ciele agentov môžu byť v 2 základných vzťahoch: Môžu byť kompatibilné alebo nekompatibilné. To, že sú ciele nekompatibilné, znamená, že splnenie jedného cieľa má za následok nesplniteľnosť druhého cieľa. Formálne: splnený(cieľ1) $\Rightarrow \neg$splnený(cieľ2) (z toho zároveň vyplýva že platí splnený(cieľ2) $\Rightarrow \neg$splnený(cieľ1)). V prípade, že sú ciele dvoch agentov navzájom nekompatibilné, nemôže medzi nimi vzniknúť spolupráca a hovoríme že ich vzťah je antagonistický. Naopak ak sú ciele kompatibilné, teda môžu byť splnené oba súčasne, môže medzi agentmi vzniknúť spolupráca, ktorá vedie k splneniu oboch cieľov. Na základe kompatibility cieľov sa môžu v multiagentovom systéme vytvárať skupiny spolupracujúcich agentov, ktoré majú antagonistický vzťah k iným skupinám agentov.

Dostatok zdrojov. Na dosiahnutie svojich cieľov agenty potrebujú využívať zdroje, či už sú to materiálne zdroje ako suroviny a nástroje alebo nehmotné zdroje ako informácie, služby, priestor alebo výpočtový čas. V prípade, že má agent dostatok vlastných zdrojov pre dosiahnutie svojich cieľov, je situácia pomerne jednoduchá. Problém nastáva, keď agent nemá dostatok zdrojov alebo je nútený deliť sa o zdroje s ostatnými. V prípade nedostatku zdrojov musí agent interagovať s ostatnými agentmi, aby potrebné zdroje získal alebo zabezpečil vykonanie činnosti, na ktorú nemá dosť zdrojov, iným agentom. Ďalším problémom je prístup viacerých agentov k zdroju, ktorý môže naraz vlastniť iba jeden agent, prípadne nepriaznivé ovplyvňovanie sa akcií,

ktoré agenty so zdrojom vykonávajú. V takomto prípade je potrebné prístup agentov ku zdroju koordinovať.

Dostatok schopností. V prípade, že agent nemá dostatok schopností na dosiahnutie svojho cieľa, musí na jeho dosiahnutí spolupracovať s inými agentmi. Ide napríklad o využitie služieb experta na riešenie určitej úlohy alebo o dosiahnutie kvalitatívnych výsledkov použitím kvantitatívnych metód (skupina agentov má schopnosti, ktoré samostatne nemá žiadny z nich).

Statické a dynamické vzťahy

Vzťahy medzi agentmi môžu byť statické, nemenné počas celej doby existencie organizácie, alebo môžu mať určitý stupeň voľnosti, v rámci ktorého sa môžu dynamicky meniť.

Na základe možnosti meniť vzťahy v organizácii rozoznávame:

- Pevné vzťahy. Nemenia sa, ostávajú zachované počas celého života organizácie, a to na úrovni štruktúry organizácie aj konkrétnej organizácie. Vzťahy v organizácii sú v tomto prípade presne zadefinované a nemôžu vznikať nové, zanikať alebo meniť svoj charakter. Organizácia s pevnými vzťahmi je na hranici medzi multiagentovým systémom a systémom so spolupracujúcimi modulmi.
- Premenlivé vzťahy. Vzťahy medzi časťami organizácie sa môžu meniť, avšak iba na úrovni konkrétnej organizácie. Zostáva pri tom zachovaná statická organizačná štruktúra, v rámci ktorej môžu jednotlivé vzťahy v konkrétnej organizácii vznikať. Príkladom je hierarchická štruktúra, v ktorej môžu na jednotlivých úrovniach hierarchie pribúdať a ubúdať agenty, avšak stále zostáva zachovaná hierarchická štruktúra.
- Evolučné vzťahy. Meniť sa môže aj samotná organizačná štruktúra. Príkladom môže byť hierarchická štruktúra, ktorá sa postupným pridávaným vzťahov mimo hierarchie môže zmeniť na voľnú štruktúru bez pôvodných hierarchických vzťahov.

Podriadenosť agentov

Z hľadiska vzájomnej podriadenosti agentov existuje veľké množstvo rôznych stupňov a spôsobov jej realizácie. Agent je inému agentu podriadený v prípade, že jeho správanie môže byť úmyselne ovplyvňované nadradeným agentom. Stupeň ovplyvňovania však môže byť rôzny.

Najprísnejším vzťahom podriadenosti je statické podriadenie. V takomto prípade nadradený agent môže dávať príkazy podriadenému a podriadený agent nemôže odmietnuť vykonanie takéhoto príkazu. Takéto vzťahy sa používajú na vytváranie prísne hierarchických štruktúr riadenia vojenského typu, môžu sa však vyskytnúť aj v inej než hierarchickej štruktúre.

Na druhej strane stoja organizácie s rovnocennými vzťahmi, kde nie je žiadny agent podriadený inému a na výslednom rozhodnutí sa podieľajú všetci rovnakou mierou. Agenty v tomto prípade môžu využívať vzťah dynamického podriadenia, kedy agent môže požiadať iného agenta o vykonanie nejakej akcie, požiadaný agent ale môže jej vykonanie bez akejkoľvek penalizácie odmietnuť.

7.3.4 Rozdelenie schopností medzi agenty

Veľmi dôležitým faktorom pri návrhu multiagentového systému je rozhodnutie, aké schopnosti pridelíme jednotlivým agentom pre riešenie ich úloh a koľké agenty budú schopné vykonávať konkrétnu úlohu. Je dôležité zabezpečiť aby agenty v systéme mali spolu dostatočné schopnosti aby dokázali plniť svoje úlohy, je však na návrhárovi, aby tieto schopnosti rozdelil medzi jednotlivé agenty. Spôsob rozdelenia schopností medzi agenty môžeme opísať dvoma parametrami: stupňom špecializácie a stupňom redundancie.

Stupeň špecializácie

Stupeň špecializácie agenta je tým väčší, čím menej má agent schopností v porovnaní s množinou všetkých schopností potrebných na riešenie problému.

Agent, ktorý má schopnosti na vykonávanie akejkoľvek potrebnej činnosti, sa nazýva univerzálny alebo totipotentný agent. Spoločenstvo univerzálnych agentov môže riešiť úlohy dvomi spôsobmi. Prvý spôsob je nezávislé riešenie úloh individuálnymi agentmi. V tomto prípade ide o kvantitatívnu spoluprácu, kedy sa kvantitatívnym spôsobom dosahujú kvantitatívne výsledky. Druhý spôsob spolupráce univerzálnych agentov je založený na spolupráci za účelom dosiahnuť kvalitatívnu výhodu, teda za účelom dosiahnutia riešenia zložitého problému ktorý je nad rámec síl individuálnych agentov.

Použitie univerzálnych agentov nemusí byť vždy výhodné. Konštrukcia a prevádzkovanie univerzálnych agentov môže byť nákladná činnosť. Na druhej strane špecializovaný agent môže veľakrát svoju činnosť vykonávať oveľa efektívnejšie, keďže nemá prebytočné súčasti a väčšinou si vystačí s jednoduchším riadením. Špecializovaný agent nedokáže vykonávať všetky úlohy rovnako efektívne, prípadne dokáže vykonávať iba niektoré úlohy. Špeciálnym prípadom je hyper-špecializovaný agent, ktorý dokáže vykonávať práve jednu úlohu.

Redundancia

Redundancia je tým väčšia, čím väčší je počet agentov s určitou schopnosťou v porovnaní s počtom agentov s touto schopnosťou, ktorých je nevyhnutne treba na splnenie danej úlohy.

V systéme bez redundancie má každý agent presne stanovené úlohy, ktoré vykonáva a každú úlohu vykonáva práve jeden agent. V prípade, že ide o univerzálneho (totipotentného) agenta, systém sa redukuje na jediného agenta schopného vykonávať všetky úlohy. Takýto systém môže byť jednoduchý z pohľadu riadenia, pretože nie je potrebné rozhodovať, ktorý agent bude vykonávať určitú úlohu. Na druhej strane môže mať takýto systém problémy s výkonnosťou v prípade že nastávajú situácie, kedy príde úloha a agent, ktorý ju má riešiť, práve rieši inú úlohu. Systémy bez redundancie majú tiež nižšiu spoľahlivosť, pretože nefunkčného agenta nemôže nahradiť iný.

V systéme s vysokou redundanciou má každú schopnosť (presnejšie schopnosť vykonávať určitú úlohu) veľké množstvo agentov, ktorí sú vzájomne zastupiteľní. Vysoká redundancia zvyšuje spoľahlivosť systému a môže zvýšiť výkonnosť systému. Na druhej strane je nutné vytvoriť mechanizmus rozdeľovania úloh medzi agenty s rovnakými schopnosťami. Takisto, náklady na duplikáciu systémov v jednotlivých agentoch môžu byť v niektorých prípadoch veľké.

7.3.5 Rozdeľovanie úloh / činností

Ak má multiagentový systém (alebo vo všeobecnosti akákoľvek organizácia) úspešne plniť svoju úlohu, je potrebné zabezpečiť efektívne rozdeľovanie úloh medzi jednotlivé agenty (časti organizácie). Efektivita rozdeľovania úloh sa skladá z dvoch aspektov, jednak je to efektivita vykonávaných činností po rozdelení úloh a ďalej tiež efektivita samotného procesu rozdeľovania úloh.

Rozdeľovanie úloh môže byť vykonané statickým spôsobom, vtedy úlohy prideľuje tvorca systému ešte pred začiatkom činnosti systému. Takýto spôsob prideľovania úloh je pomerne neflexibilný a často nedovoľuje systému dostatočne reagovať na zmenenú situáciu, prípadne sa vysporiadať s neočakávanými situáciami. Na druhej strane statické pridelenie úloh odstraňuje potrebu mechanizmu prideľovania úloh počas chodu systému, čo môže zefektívniť riešenie dobre zadefinovaných úloh. Prideľovanie úloh tvorcom systému so sebou prináša aj ďalší aspekt, a to veľkú závislosť výsledku na schopnostiach tvorcu systému vhodne rozdeliť úlohy, a to aj s ohľadom na neštandardné situácie.

Dynamické rozdeľovanie úloh sa deje počas behu systému, väčšinou v momente, keď sa objaví nová úloha, ktorú treba riešiť, alebo sa objaví potreba prerozdeliť úlohy na základe zmenenej situácie, napríklad pri poruche agenta.

Táto kapitola obsahuje základné typy štruktúry organizácie vzhľadom na rozdeľovanie úloh medzi jednotlivé časti organizácie. Tieto spôsoby rozdeľovania úloh sú určené hlavne pre dynamické rozdeľovanie úloh. Je tiež možné kombinovať tieto techniky so statickým spôsobom prideľovania úloh, napríklad pri mechanistickej štruktúre nemusia byť (a veľa krát skutočne nie sú) štandardy výsledkom činnosti agentov ale sú vopred dané tvorcom systému. Terminológia použitá v tejto kapitole pochádza z oblasti riadenia technologických procesov ako ju zadefinoval Romelaer (Romelaer, 2002).

Jednoduchá štruktúra

Najjednoduchší spôsob rozdeľovania úloh je ten, že všetky agenty (časti systému) sú úplne riadené nadradenými agentmi. Znamená to, že agent dostane príkaz od nadradeného agenta a nemôže odmietnuť jeho splnenie. Ide teda o prísnu podriadenosť agentov.

Typické použitie jednoduchej štruktúry rozdeľovania úloh je hierarchia vojenského typu, kde k rozdeľovaniu úloh dochádza postupným delegovaním úloh agentom na nižšom stupni hierarchie. Použitie tohto princípu však nie je obmedzené iba na hierarchie, používa sa prakticky vo všetkých typoch organizácií, kde je možné vo vzťahu jednoznačne odlíšiť zadávateľa úlohy od toho kto ju vykonáva.

Mechanistická štruktúra, štandardizácia

Všetky zložky organizácie sa riadia štandardizovanými postupmi a metódami práce. Pre množiny podobných úloh sú určené štandardné postupy, ktoré sa používajú na ich riešenie. Štandardy môžu tiež obsahovať spôsob rozdelenia práce medzi agenty alebo všeobecné pravidlá správania sa agentov v určitých situáciách, ako napríklad protokol pre vyhnutie sa kolízii robotov (Zeghal, 1993) alebo spôsob oznamovania neštandardných situácií ostatným agentom (Dellarocas, 1999).

Štandardy môžu byť vytvorené tvorcom systému a počas behu systému nemenné. V prípade potreby však môžu byť štandardy upravované ako reakcia na neočakávanú

situáciu. V tomto prípade existujú špecializovaní agenti alebo organizácie, ktorí sú zodpovední za modifikáciu štandardov. Takýmto spôsobom sa zvyšuje flexibilita organizácie a umožňuje sa zvyšovanie efektivity organizácie zavádzaním nových štandardných postupov.

Štruktúra založená na kompetenciách

Každý agent / skupina agentov má zodpovednosť za určitú oblasť činností. V prípade, že príde nová úloha, je jasne určený agent, ktorý je zodpovedný za jej riešenie. Tento agent môže samozrejme časť riešenia delegovať na iného agenta, avšak znova na agenta, ktorý je za riešenie tejto čiastkovej úlohy zodpovedný. Rozdelenie zodpovedností závisí od konkrétneho problému, môže byť vytvorené napríklad na základe schopností agentov alebo geografického umiestnenia úlohy.

Rozdelenie zodpovedností medzi agenty môže byť určené tvorcom systému pred jeho spustením, alebo môže vzniknúť dynamicky explicitným pridelením zodpovedností alebo postupným prispôsobovaním agentov. Typickým príkladom dynamického rozdeľovania zodpovedností je systém Manta (Drogoul, 1995), v ktorom sa mravce s univerzálnymi schopnosťami postupne špecializujú na jednu z úloh ako je hľadanie potravy alebo starostlivosť o larvy.

Štruktúra založená na výsledkoch

V organizácii sú určené iba základné ciele, ktoré je potrebné dosiahnuť. Je určená štruktúra výsledkov aké majú jednotlivé agenty dodávať, avšak nie spôsob ich dosiahnutia. Každý agent má vo svojej kompetencii spôsob (teda aj svoje čiastkové ciele a úlohy), akým dosiahne požadovaný výsledok.

Takáto štruktúra je typická pre zabezpečovanie výsledkov kvantitatívnou metódou, tú istú úlohu rieši veľké množstvo agentov a kvalita výsledku je spravidla úmerná počtu agentov, ktoré sa na jej riešení podieľa. Ďalšie použitie štruktúry založenej na výsledkoch je riešenie veľkého počtu úloh s rovnakou štruktúrou výsledku. V takomto prípade každý agent dostáva iné úlohy a generuje výsledky s pevne danou štruktúrou. Príkladom môže byť predaj výrobkov alebo služieb, kedy má každý agent za úlohu predať čo najviac výrobkov. Činnosť jednotlivých predajcov je potom pomerne nezávislá a o spôsobe maximalizácie svojho predaja rozhoduje každý agent individuálne.

Adhokracia

Adhokracia znamená rozdeľovanie úloh metódou ad-hoc. Štruktúra a rozdelenie úloh sú vytvárané znovu pre každý problém, ktorý treba riešiť. Typické použitie adhokracie je pri nedostatočne štruktúrovaných problémoch alebo problémoch, pri ktorých nevieme dopredu určiť spôsob ich riešenia. Často sa pri prideľovaní úloh využíva mechanizmus trhu (napríklad „contract net“ (Smith, 1980)).

Pri riešení konkrétneho problému sa vytvorí ad-hoc štruktúra rozdelenia úloh. Po viacnásobnom riešení podobného problému podobným spôsobom je možné štruktúru spolupráce vytvorenú spôsobom ad-hoc ustáliť, teda preniesť do štandardov alebo rozdelenia zodpovedností.

7.3.6 Modely organizácií

Všeobecne platí, že čo problém, to iná štruktúra organizácie. Pri vytváraní štruktúry vzťahov jednotlivých komponentov organizácie je potrebné zohľadniť všetky špecifiká konkrétneho problému, ktorý daná organizácia rieši. Tieto špecifiká môžu byť veľmi odlišné, čo sa samozrejme premietne aj do rôznorodosti výsledných organizácií. Vytvárať štruktúru organizácie pre každý nový problém môže byť ale veľmi prácne a väčšinou je vhodnejšie použiť jeden z existujúcich modelov organizácie.

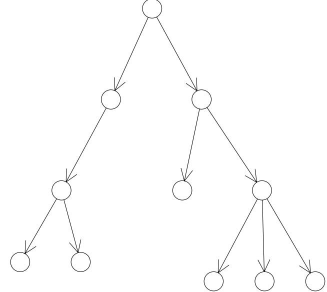
Modely organizácií nevznikli za účelom pokrytia všetkých možných štruktúr organizácií. Naopak, často vznikli zovšeobecnením a formalizáciou konkrétnych organizácií použitých v praxi. Modely organizácií môžu slúžiť pre zrýchlenie vytvárania organizácií agentov tým, že vlastnosti štruktúry jednotlivých modelov sú pomerne dobre preskúmané a experimentálne overené, čím sa výrazne znižuje riziko zvolenia nesprávnej štruktúry organizácie. Ďalšou výhodou použitia jedného zo štandardných modelov je možnosť znovupoužitia návrhu alebo dokonca časti implementácie existujúceho systému založeného na tom istom modeli.

Táto kapitola obsahuje základné modely organizácií z hľadiska štruktúry vzájomného prepojenia jej komponentov prostredníctvom interakcií týchto komponentov. Modely uvedené v tejto kapitole pochopiteľne nepokrývajú všetky možné organizácie, avšak ich modifikáciami a vzájomnou kombináciou je možné pokryť veľkú časť problémov riešených mltiagentovými systémami.

Medzi modely organizácií sa zvykne zaraďovať aj holarchia (kapitola 0). Holarchiu však môžeme chápať skôr ako metamodel, v rámci ktorého používame ostatné modely. Zatiaľ čo ostatné modely sa obmedzujú na organizačnú štruktúru na jedinej úrovni (vzťahy medzi celkom a jeho časťami), holarchia predstavuje hierarchické usporiadanie takýchto jednoúrovňových štruktúr. Holarchia umožňuje jednoduchým a prehľadným spôsobom kombinovať jednotlivé modely na rôznych úrovniach organizácie zatiaľ čo tieto úrovne jednoznačne oddeľuje.

Hierarchia

Hierarchia je tvorená vzťahmi vzájomnej podriadenosti agentov, pričom každý agent má práve jedného nadriadeného. Hierarchická štruktúra tak tvorí strom, ktorého vrcholy sú agenty, listy predstavujú agenty na najnižšej úrovni, agent na najvyššej úrovni hierarchie tvorí koreň stromu. Príklad hierarchie je na obrázku 7-7.



Obrázok 7-7. Príklad hierarchickej štruktúry organizácie.

S hierarchickou štruktúrou je často spájaná absolútna, statická podriadenosť agentov. V tomto prípade ide o hierarchiu vojenského typu, kde sú nižšie postavené agenty plne podriadené agentom stojacim v hierarchii o stupeň nižšie a vzniká tak hierarchická príkazová štruktúra. Na druhej strane hierarchia nemusí slúžiť na vyjadrenie podriadenosti jej častí. Hierarchia môže napríklad spájať agenty s podobným účelom v jednej časti stromu hierarchie, čím sa skracuje komunikačná vzdialenosť podobných agentov. Hierarchia tiež môže zodpovedať funkčnej dekompozícii riešeného problému, v tomto prípade je zabezpečená malá vzdialenosť agentov spolupracujúcich na tom istom podprobléme.

Hierarchia v jej čistej podobe má výhodu v tom, že je presne a jednoznačne zadefinovaná a v prípade niektorých problémov (napríklad ak je vhodné riešiť problém dekompozíciou) môže byť veľmi efektívna. Rovnako sa aj veľmi zjednodušuje komunikačný protokol medzi agentmi v hierarchii, pretože medzi každými dvoma agentmi existuje v hierarchii práve jedno spojenie. Komunikácia však môže predstavovať aj istý problém, pretože komunikácia často prechádza viacerými spojeniami a môže predstavovať neúmerné zaťaženie častí systému. Priama komunikácia medzi jednotkami na tej istej hierarchickej úrovni nie je možná vôbec, vždy musí ísť hierarchiou smerom nahor až po spoločnú nadriadenú jednotku a potom znova absolvovať rovnako dlhú cestu smerom nadol.

Niektoré nedostatky čisto hierarchických vzťahov je možné odstrániť zavedením vzťahov mimo štruktúry hierarchie. Takéto vzťahy môžu vzniknúť napríklad tak, že pri prvej komunikácii sa vytvorí spojenie medzi agentmi s využitím cesty hierarchiou, avšak ďalšia komunikácia už prebieha priamo.

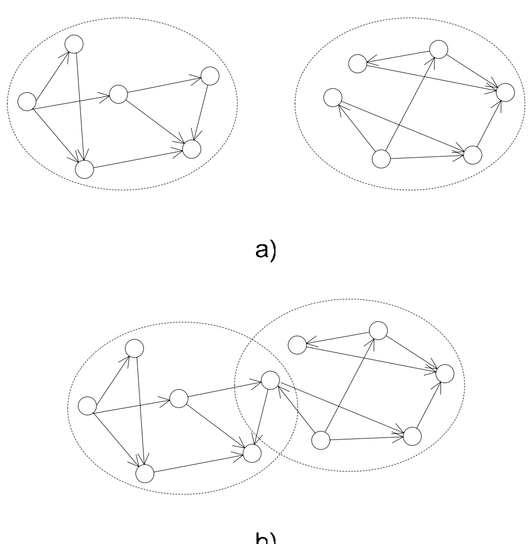
Príklad prístupu využívajúceho hierarchiu je klonovanie agentov (Ishida, 1992). Keď agent nedokáže sám riešiť nejakú úlohu, vytvorí svoju kópiu – klon. Tomuto klonu potom prenechá časť riešenia svojej úlohy. Ak je medzi pôvodným agentom a jeho klonom vzťah podriadenosti, vzniká takto hierarchická organizácia. Zároveň týmto mechanizmom vzniká hierarchická dekompozícia úlohy.

Skupiny agentov

Agenty sa môžu rozdeliť do viacerých skupín, ktoré vznikajú za určitým účelom. Účelom vzniku skupiny môže byť napríklad vzájomná spolupráca agentov. Štruktúra vo vnútri skupín a aj medzi skupinami nemusí byť pevná, môže byť vytvorená podľa ľubovoľného modelu. Napríklad skupiny medzi sebou môžu mať voľnú organizáciu, zatiaľ čo agenty v jednej skupine môžu mať hierarchickú štruktúru. Príklad organizácie založenej na skupine agentov je na obrázku 7-8.

Účel vytvorenia skupiny agentov môže byť rôzny, na základe tohto účelu je možné rozlíšiť niekoľko rôznych alternatív tohto modelu:

Spojenectvá. Sú vytvárané za účelom ľahšieho dosiahnutia osobného cieľa každého z agentov. Agenty sa na základe svojich vlastných cieľov rozhodnú spolupracovať. Toto rozhodnutie je podmienené kompatibilitou cieľov jednotlivých agentov, teda splnenie cieľov jednotlivých agentov tvoriacich spojenectvo by nemalo spôsobiť nesplniteľnosť cieľov iných agentov v spojenectve. Výhodou spojenectiev je to, že agenty môžu koordinovať svoje akcie, zdieľať navzájom svoje výsledky a postupovať spoločne, čím dokážu lepšie využiť spoločné zdroje a efektívnejšie dosiahnuť svoje ciele. Spojenectvá sa využívajú pre automatizované prideľovanie úloh agentom (či už na základe vyjednávania agentov (Sims, 2003) alebo centrálnym rozdelením agentov do spoločenstiev (Shehory, 1998)).



Obrázok 7-8. Príklad organizácie agentov do skupín. Skupiny môžu byť disjunktné (a) alebo sa môžu prekrývať (b).

Tímy. Sú vytvárané za účelom dosiahnutia spoločného, tímového, cieľa. Na rozdiel od spojenectva majú všetky agenty v tíme spoločný tímový cieľ, na ktorého dosiahnutí sa spoločne podieľajú. Keďže v tíme všetky agenty sledujú ten istý cieľ, rozširujú sa oproti spojenectvu možnosti spolupráce agentov. Príkladom systému založeného na tímoch agentov je STEAM (Tambe, 1997) použitý na plánovanie činnosti tímu (bližšie kapitola 7.4.4). Iným príkladom je systém pre riadenie elektrickej siete založený na agentoch umiestnených v uzloch elektrickej siete so spoločným cieľom zabezpečiť bezporuchovú dodávku elektrickej energie (Jennings, 1995).

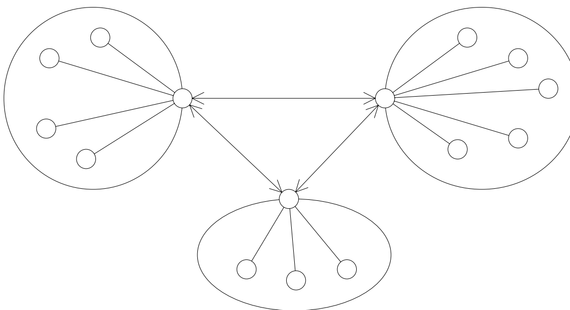
Združenia. Zatiaľ čo spojenectvá a tímy vznikajú len po dobu riešenia jediného problému, združenia sú vytvárané na základe spoločných záujmov a vlastností agentov na dlhšiu dobu. Združenie vzniká na základe predpokladu, že jeho členovia budú mať počas dlhšieho obdobia podobné ciele, na dosiahnutí ktorých môžu spolupracovať Takémuto združeniu sa zvykne hovoriť aj klan (Griffiths, 2003). Inou možnosťou je združenie agentov so schopnosťami, ktoré sa vzájomne dopĺňajú a sú potrebné pre riešenie typických úloh, ktoré združené agenty riešia. Príkladom združenia môže byť firma na výrobu elektronických súčiastok. Účelom združenia je zjednodušiť a zefektívniť spoluprácu na cieľoch a úlohách, ktoré prídu v budúcnosti tým, že sa skupina spolupracujúcich agentov vytvorí skôr alebo sa aspoň obmedzí množina agentov, z ktorej sa budú spolupracovníci vyberať. Združenia boli úspešne použité napríklad na zefektívnenie činnosti trhu (Brooks, 2002) tým, že sa predávajúce a kupujúce agenty s podobnými preferenciami združujú, čím sa v budúcnosti zrýchli nájdenie vhodných párov predávajúcich a kupujúcich.

Spoločenstvá. Spoločenstvo (alebo tiež mechanistická štruktúra (Romelaer, 2002)) vzniká na základe prijatia spoločných noriem. Vstupom do spoločenstva sa agent zaväzuje, že bude tieto spoločné normy v najväčšej možnej miere dodržiavať. Spoločenstvo (agenty v spoločenstve) jedná vždy na základe spoločných noriem. Tieto normy sa môžu týkať rôznych oblastí činnosti agentov, od štandardných postupov riešenia problémov až po spôsob vzájomného zdieľania informácií. Porušenie noriem

môže prinášať znevýhodnenie agentov v ďalších interakciách. Príkladom môže byť oblasť elektronického trhu, kde porušenie pravidiel vedie k znevýhodneniu agenta pri ďalších obchodoch (Dellarocas, 1999).

Federácie

Federácia sa podobá skupine agentov, ale s tým rozdielom, že vo federácii je vždy jeden agent (zástupca), ktorý komunikuje s ostatnými federáciami prostredníctvom ich zástupcov. Ostatné agenty vo federácii komunikujú s agentmi mimo ich federáciu iba prostredníctvom svojho zástupcu. Príklad federácie je na obrázku 7-9.



Obrázok 7-9. Príklad federácie. Federácia komunikuje s ostatnými federáciami prostredníctvom jediného agenta.

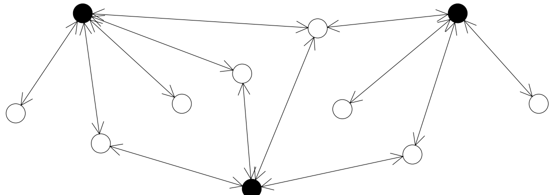
Zástupca tu pôsobí ako rozhranie federácie smerom von. Toto umožňuje zavedenie bezpečnostných mechanizmov, ktoré oddeľujú agenty od okolitého sveta a bránia ich pred neželanými zásahmi zvonku. Zástupca spracováva všetky požiadavky z iných častí organizácie a tie, ktoré sú identifikované ako potenciálne škodlivé, môže zablokovať. Príkladom použitia federácií je systém Metamorph (Maturana, 1999) pre riadenie výroby. Federácie tu združujú agenty spolupracujúce na jednej úlohe. Prístup k zdrojom sa potom pre celú federáciu riadi jednotne.

Zástupca federácie môže tiež pôsobiť vo funkcii makléra (angl. *broker*) (Sycara, 1997), ktorý ponúka schopnosti svojich agentov ostatným federáciám a naopak zabezpečuje služby, ktoré požadujú agenty z vnútra federácie, od iných federácií.

Prepojenie agentov vo vnútri federácie a prepojenie federácií navzájom môže byť vytvorené podľa ľubovoľného modelu, pre štruktúru vo vnútri federácie sa často používa plytká hierarchia, v ktorej sú všetky agenty federácie priamo podriadené zástupcovi.

Trhy

Organizácie vytvorené na princípe trhov sú tvorené dvomi typmi agentov: predávajúcimi a kupujúcimi. Kupujúci požadujú (alebo podávajú ponuky) určité služby, zdroje alebo informácie. Opačnú stranu vzťahov tvoria predávajúci, ktorí tieto služby, zdroje a informácie poskytujú kupujúcim. Predávajúci sú zodpovední za spracovávanie ponúk a výberu najlepšej ponuky, ktorú uspokoja. Predávajúci môžu tiež pôsobiť v roli sprostredkovateľa, ktorý prijíma od agentov tovar (služby, zdroje, ...), ktorý potom poskytuje kupujúcim. Príklad trhu je na obrázku 7-10. V niektorých prípadoch môže ten istý agent súčasne zastávať úlohu kupujúceho aj predávajúceho.



Obrázok 7-10. Príklad trhu. Biele krúžky predstavujú kupujúcich, čierne krúžky predstavujú predávajúcich alebo sprostredkovateľov.

Vzťahy v organizáciách tohto typu vlastne predstavujú vzťahy výrobca-spotrebiteľ a dobre zodpovedajú vzťahom v klasickej trhovej ekonomike. Výskum v tejto oblasti teda môže čerpať z výskumu v oblasti ľudských ekonomických systémov.

Použitie tohto modelu organizácie sa neobmedzuje iba na systémy založené na ponúkaní tovaru za nejaký druh kompenzácie (napríklad internetový obchod (Chavez, 1996)). Je možné tento model použiť aj pre iné problémy, typický príklad sú systémy pre rozdeľovanie zdrojov alebo úloh. Agenty v tomto prípade ponúkajú svoje schopnosti vykonávať určitú činnosť a na základe kvality týchto ponúk sa vyberajú agenty najvhodnejšie pre jednotlivé úlohy. Príkladom je protokol „contract net“ (Smith, 1980).

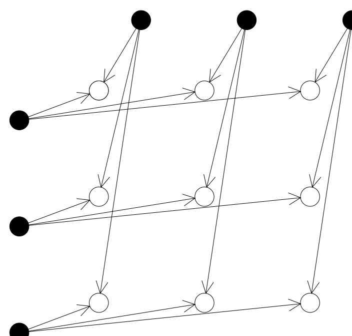
Viacnásobné hierarchie

Niekedy jednoduchá hierarchická štruktúra nie je postačujúca, a to ani v prípade, ak povolíme aj mimohierarchické vzťahy. Viacnásobná hierarchia umožňuje, aby bol agent súčasťou viacerých hierarchií naraz. Tieto hierarchie sú samostatné a prepojené sú iba prostredníctvom agentov, ktoré sa nachádzajú v oboch hierarchiách. Tento prístup je možné zvoliť, ak sú schopnosti agenta súčasne potrebné pre riešenie viacerých úloh. Príklad takejto organizácie je na obrázku 7-11.

Viacnásobné hierarchie sú známe aj pod názvom maticové organizácie, pretože vzťahy v organizácii tvoria maticu, ktorej jednotlivé rozmery zodpovedajú jednotlivým hierarchiám.

Viacnásobné hierarchie sa v „reálnom svete“ veľmi úspešne využívajú vo výrobných podnikoch, kde jedna hierarchia zodpovedá „klasickému“ systému oddelení (manažment, finančné oddelenie, vývoj, výroba, ...) a zároveň existuje druhá hierarchia, ktorá zohľadňuje jednotlivé produktové rady, ktoré podnik vyrába. Každý zamestnanec má takto viac priamych nadriadených, ktorí sa navzájom dopĺňajú. Jeden nadriadený sa stará o zabezpečenie chodu podniku ako organizácie a druhý o činnosti spojené s výrobou a predajom konkrétneho produktu.

Príkladom využitia viacnásobnej hierarchie v multiagentových systémoch je riešenie prideľovania zdrojov v distribuovanej senzorickej sieti (Horling, 2003). Každý senzor je samostatný agent, ktorý dostáva príkazy od viacerých agentov zodpovedných za preskúmavanie určitej časti okolia alebo za sledovanie známych objektov. Na základe príkazov sa potom senzor rozhoduje ktorú časť svojho okolia bude sledovať.



Obrázok 7-11. Príklad viacnásobnej hierarchie. Agent môže byť naraz súčasťou viacerých hierarchií.

Horizontálne moduly / organizácia s pevnou štruktúrou

Hierarchické organizácie sú v skutočnosti iba jednou podmnožinou organizácií založených na horizontálnych moduloch. Jednotlivé súčasti (moduly) takejto organizácie majú presne stanovený spôsob spolupráce a typy väzieb s inými modulmi. Organizácia horizontálnych modulov určuje pevné väzby medzi modulmi, pričom všetku moduly sú na tej istej úrovni abstrakcie.

Takýto systém stojí na hranici multiagentových systémov, keďže sa zavedením pevnej štruktúry výrazne obmedzuje autonómnosť jednotlivých modulov. Moduly v tomto modeli majú spravidla presne stanovenú funkciu a rozhrania na iné moduly. Konkrétna podoba takejto organizácie je veľmi závislá od konkrétneho problému, ktorý má riešiť. Organizácie s pevnou štruktúrou sa väčšinou používajú na tvorbu jednotlivých agentov, menej často na tvorbu multiagentových systémov. Príkladom systémov s horizontálnymi modulmi sú systémy s modulmi navrhnutými na základe funkčného modelu (Návrat, 2002) alebo bunkové automaty (Gerhardt, 1995).

Voľná organizácia, organizácia bez štruktúry

Voľná organizácia nie je nijako (explicitne) štruktúrovaná, interakcie medzi agentmi sú náhodné alebo na základe princípu, ktorý nevychádza zo štruktúry organizácie, napríklad priestorového umiestnenia (hoci priestorové umiestnenie môže byť niekedy považované za istý druh združenia).

Agenty sledujú vlastné ciele, vzájomne väčšinou nespolupracujú, explicitná spolupráca by totiž znamenala vytvorenie spojenectva, tímu alebo združenia. Spolupráca medzi agentmi je väčšinou iba na úrovni koordinácie akcií, prípadne vzniká emergenciou. Príkladom voľnej organizácie môžu byť simulované ekológie, kde k interakcii jedincov dochádza na základe ich fyzickej blízkosti.

Hybridné organizácie

Uvedené modely organizácií nedokážu pokryť všetky aspekty možných problémových oblastí. Často preto vznikajú organizácie, ktoré využívajú prvky viacerých modelov. Vznikajú tak hybridné organizácie, ktoré majú časť vlastností modelov, ktorých kombináciou vznikli.

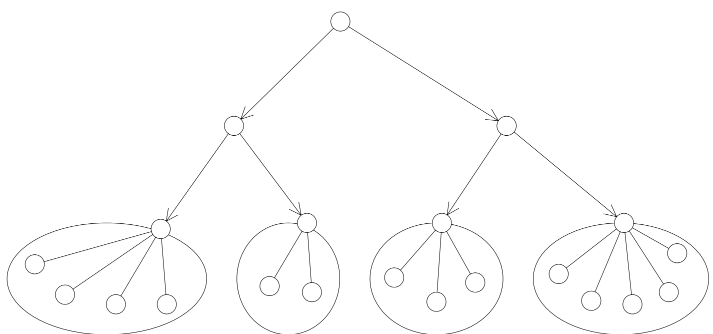
Príkladom je hierarchia s prepojeniami mimo hierarchie, ktorá vznikne kombináciou „klasickej“ hierarchie a voľnej organizácie (napríklad systém STEAM (Tambe, 1997)).

Mimohierarchické vzťahy môžu vznikať aj na základe iného modelu než je voľná organizácia, v hierarchii sa napríklad môžu vytvárať združenia na základe predpokladu častej interakcie pri riešení problémov. Iným príkladom sú spoločenstvá riadiace sa normami, v ktorých ale existujú aj ďalšie organizačné štruktúry (Dellarocas, 1999).

Zložené organizácie

Zložené organizácie používajú rôzne modely organizácie na rôznych úrovniach abstrakcie. Kombinujú tak výhody (ale samozrejme aj nevýhody) rôznych modelov organizácie. Najprirodzenejším spôsobom vytvorenia zloženej organizácie je použitie princípu holonickej organizácie. Vnútorná štruktúra holonu je vytvorená podľa niektorého modelu. Navonok sa však tvári ako uzavretý celok, ktorý je súčasťou holonu nadradeného. Nadradený holon môže byť vytvorený podľa ľubovoľného iného modelu.

Príkladom zloženej organizácie je hierarchia federácií (obrázok 7-12). Takúto štruktúru využíva napríklad distribuovaná sieť senzorov (Yadgar, 2003), kde sú jednotlivé senzory združené do federácií na základe ich fyzického umiestnenia. Agent zastupujúci federáciu potom predáva získané nespracované informácie agentom na vyššej hierarchickej úrovni na ďalšie spracovanie.



Obrázok 7-12. Príklad zloženej organizácie, ktorá má na vyššej úrovni hierarchickú štruktúru, na nižšej úrovni štruktúru federácií.

7.4 Plánovanie v kontexte multiagentových systémov

Táto kapitola sa pokúsi poskytnúť prehľad základných metód plánovania a niektorých ich modifikácií s prihliadnutím na ich možné použitie v multiagentových systémoch.

Problém plánovania je jedným z najstarších problémov umelej inteligencie. Problém plánovania možno vyjadriť ako hľadanie činností (či už zoradených sekvenčne alebo časovo sa prekrývajúcich), ktoré pre dané zadanie spôsobia jeho vyriešenie. Navyše, klasická umelá inteligencia na plánovanie kladie ďalšie požiadavky, a to racionalitu a zdôvodniteľnosť rozhodnutí a výberu činností, ktoré sa budú vykonávať.

Problém vytvárania plánov nie je triviálny ani v prípade, že vykonávanie plánu nie je distribuované a plán sa vykonáva sekvenčne. Nájdenie optimálneho plánu (optimálnej postupnosti krokov, ktoré treba vykonať) je vo všeobecnosti nerozhodnuteľný

problém[15] (dôkazy o nerozhodnuteľnosti niektorých problémov spadajúcich do oblasti plánovania možno nájsť napríklad v (Madani, 2003)). Zložitosť plánovania však môžeme podstatne znížiť, ak do plánovacieho procesu zavedieme obmedzenia alebo ak vlastníme nejakú dodatočnú informáciu o probléme.

Veľmi častým zjednodušením používaným takmer všetkými plánovacími algoritmami je diskretizácia času. Stav systému je v priebehu času považovaný za nemenný a môže sa meniť iba v diskrétnych časových okamihoch, ktoré môžu ale nemusia byť od seba rovnako vzdialené. Aj po časovej diskretizácii však plánovanie ostáva vo všeobecnosti nerozhodnuteľným problémom a je potrebné hľadať ďalšie obmedzenia. Napríklad plánovanie obmedzené jazykom STRIPS (kap. 7.4.3) má zložitosť už „iba“ v triede PSPACE [16]. Ďalšie obmedzenia jazyka STRIPS dokážu znížiť zložitosť na polynomiálnu[17] (Bylander, 1994), avšak každé zavedené obmedzenie výrazne znižuje triedu problémov, ktoré je daný algoritmus schopný riešiť. Ďalšou možnosťou, ako zefektívniť proces plánovania, je využiť doménové znalosti a namiesto všeobecného algoritmu plánovania použiť špeciálny systém pre danú doménu alebo skupiny domén. Môžeme tiež zvoliť k zložitosti iný prístup a namiesto znižovania maximálnej zložitosti znižovať zložitosť priemernú. V praxi to znamená hľadanie postupov, ktoré sú nie vždy optimálne alebo dokonca nie je zaručená ani ich správnosť alebo konečnosť, avšak sú dostatočne rýchle a presné pre väčšinu prípadov.

Plánovanie pre multiagentové systémy musí počítať s viacerými špecifikami oproti plánovaniu pre samostatné agenty. Ide hlavne o problémy, ktoré nastávajú pri paralelnom vykonávaní akcií, pretože akcie sa môžu navzájom ovplyvňovať. Výsledok dvoch rôznych akcií vykonaných v tom istom čase je často iný než „súčet“ výsledkov týchto akcií vykonaných samostatne. Ak napríklad dva agenty vykonajú akciu, vďaka ktorej sa posunú tak, že sa ich polohy prekryjú, dôjde ku kolízii. Aby sme mohli modelovať takúto kolíziu, musíme pri modelovaní akcií vziať do úvahy aj vzájomné pôsobenie súčasne vykonávaných akcií. Ďalším špecifikom multiagentových systémov je to, že jednotlivé časti systému, agenty, sú samostatné entity schopné samostatného rozhodovania. Prítomnosť ďalších samostatných agentov zhoršuje predpovedateľnosť správania sa prostredia a predpovedateľnosť výsledkov vlastných akcií.

7.4.1 Typy plánovania pre multi-agentové systémy

Samotný proces plánovania v multiagentovom systéme pozostáva z troch hlavných krokov (Ferber, 1999):

1. Vytvorenie plánov

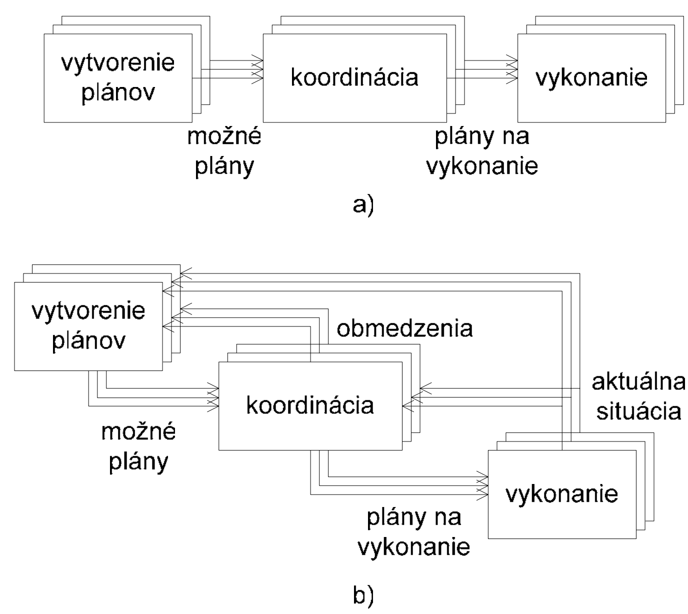
---

[15] Zjednodušene, problém je nerozhodnuteľný ak nie je možné v obmedzenom (konečnom) čase nájsť algoritmickým spôsobom jeho riešenie. Nerozhodnuteľný je ten problém, ktorý nemožno riešiť pomocou Turingovho stroja v konečnom počte krokov, čo reálne znamená že nie je riešiteľný počítačom.

[16] PSPACE[17] problémy, alebo tiež problémy s polynomiálnou pamäťovou zložitosťou. Nekladie sa žiadne obmedzenie na časovú zložitosť. Všeobecne sa predpokladá že časová zložitosť je minimálne exponenciálna, avšak neexistuje zatiaľ dôkaz tohto predpokladu.

[17] Veľmi podrobný a pravidelne aktualizovaný prehľad tried zložitosti a ich vzájomných vzťahov možno nájsť napríklad na internetovej adrese http://www.complexityzoo.com/ (Aaronson, 2005).

2. Koordinácia plánov
3. Vykonanie plánov



Obrázok 7-13. Základné tri kroky plánovania v multiagentovom systéme: vytvorenie plánov, koordinácia, vykonanie. Každý z krokov môže vykonávať viacero agentov. Tieto kroky môžu byť vykonané sekvenčne (a) alebo sa môžu časovo prekrývať (b).

Tieto úlohy sú rozdelené medzi jednotlivé agenty, pričom agent sa môže zúčastniť viacerých z nich. To, ako sú tieto úlohy rozdelené, závisí od konkrétneho systému a možností jednotlivých agentov. Prvý krok, vytváranie plánov, zabezpečí vytvorenie postupností krokov, ktoré majú agenty vykonať pre dosiahnutie ich cieľov. Táto postupnosť nemusí byť úplne usporiadaná, môže v nej byť ponechaná určitá voľnosť. Ďalším krokom je koordinácia plánov pre jednotlivé agenty. Treba zabezpečiť, aby sa akcie vykonávané agentmi navzájom nepriaznivo neovplyvňovali, a tiež zabezpečiť správne vykonanie akcií, ktoré na sebe navzájom závisia. Keď sú plány skoordinované, nastáva ich vykonanie.

Jednotlivé fázy plánovania môžu nasledovať po sebe (ako na obrázku 7-13 a)) alebo sa môžu časovo prekrývať (priebežné plánovanie, obrázok 7-13 b)). Priebežné plánovanie má výhodu v tom, že sa dokáže prispôsobiť prípadným zmenám a nepredvídaným udalostiam. Pri priebežnom plánovaní sa počas vytvárania plánov a ich koordinácie plány vykonávajú. Plány sa neustále spresňujú na základe spresnených informácií o aktuálnej situácii. Vykonávanie plánov je monitorované a v prípade detekcie zlyhania plánu alebo odchýlky od zamýšľaného vykonávania sa plán pozmení.

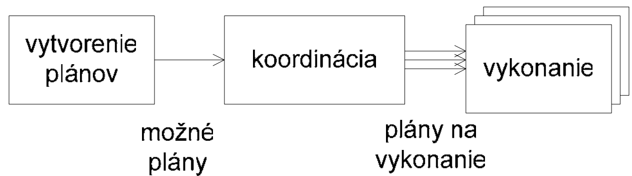
Podľa toho či jednotlivé kroky plánovania vykonáva jeden agent alebo viacero, rozlišujeme v multiagentových systémoch nasledovné typy plánovania:

- Centralizované plánovanie pre viacero agentov

- Centralizovaná koordinácia čiastkových plánov
- Distribuovaná koordinácia čiastkových plánov
- Individuálne plánovanie

Centralizované plánovanie pre viacero agentov

Centralizované plánovanie znamená, že plán pre činnosť všetkých agentov vzniká na jednom mieste. Schopnosť plánovať má takto iba jeden agent, ktorý vytvára plán pre všetky ostatné agenty. Tento agent-plánovač je spravidla zodpovedný aj za koordináciu vykonávania plánov a úloha ostatných agentov je redukovaná na vykonávanie plánu.



Obrázok 7-14. Centralizované plánovanie pre viacero agentov. Vytváranie plánov aj ich koordináciu vykonáva jeden agent.

Centralizované vytváranie plánov prebieha podobne ako vytváranie plánov pre individuálne agenty. Typická schéma vytvárania centralizovaného plánu môže vyzerať nasledovne:

1. Vytvorenie čiastočne usporiadaného plánu.
2. Nájdenie krokov plánu, ktoré sú navzájom nezávislé a môžu byť vykonané paralelne.
3. Rozdelenie úloh (častí plánu) medzi jednotlivé agenty.

Čiastočne usporiadaný plán obsahuje kroky potrebné pre dosiahnutie cieľa (cieľov) agentov spolu s obmedzeniami na vykonávanie týchto krokov. Obmedzenia sú kladené predovšetkým na časovú následnosť jednotlivých krokov plánu a zabezpečujú, aby sa súčasne nevykonávali vzájomne sa vylučujúce akcie. Obmedzenia tiež zabezpečujú aby sa pred každou akciou vykonali akcie, ktoré zabezpečujú jej predpodmienku a zároveň aby medzi vykonaním akcie a akcie zaručujúcej splnenie jej predpodmienky nedošlo k vykonaniu inej akcie, ktorá by platnosť predpodmienky zrušila. Obmedzenia sa tiež kladú na zdroje potrebné pre vykonanie jednotlivých akcií. V tejto fáze je vhodné klásť čo najmenej obmedzení aby bolo možné plán čo najjednoduchšie paralelizovať.

V čiastočne usporiadanom pláne (ak nemá príliš veľa obmedzení) je možné nájsť viacero nezávislých častí, ktoré môžu byť vykonávané paralelne, a teda rozdelené medzi viacero agentov. Je tiež potrebné nájsť miesta v pláne, kde je potrebné paralelne vykonávané akcie synchronizovať a zabezpečiť tak správnosť paralelného vykonávania plánu.

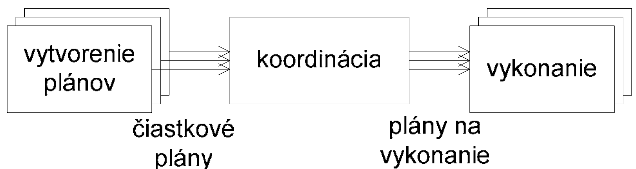
Po identifikovaní paralelných sekvencií plánu nasleduje rozdelenie častí plánu medzi jednotlivé agenty a určenie miest v pláne kde je potrebná synchronizácia. Jednotlivé paralelné sekvencie plánu identifikované v predchádzajúcom kroku sú rozdelené medzi jednotlivé agenty s ohľadom na ich špecifické vlastnosti a schopnosti. Je dobré dbať na

to, aby boli jednotlivé časti plánu rozdelené medzi agenty pokiaľ možno rovnomerne, aby sa minimalizoval čas vykonávania plánu a aby sa minimalizoval čas čakania agentov na dokončenie akcií inými agentmi. V prípade, že je vzájomná synchronizácia agentov nákladná, je vhodné spraviť rozdelenie tak, aby sa niektoré body synchronizácie nachádzali celé v častiach plánu pridelených tomu istému agentu. Pri rozdeľovaní úloh je samozrejme možné zohľadniť aj ďalšie kritériá, ako je rôzna cena vykonania akcií jednotlivými agentmi.

Centralizované plánovanie sa hodí pre systémy, kde existuje agent, ktorý má dostatočný prehľad o celkovej situácii a dokáže dostatočne presne modelovať správanie celého systému. Musí tiež existovať dostatočne spoľahlivý a rýchly komunikačný kanál, po ktorom plánujúci agent odovzdá čiastkové plány ostatným agentom.

Centralizovaná koordinácia čiastkových plánov

Centralizácia sa môže obmedziť iba na koordináciu čiastkových plánov. Každý agent je zodpovedný za vytvorenie svojho plánu, či už tento plán sleduje splnenie vlastných cieľov alebo prispieva k splneniu spoločného cieľa. Tieto čiastkové plány jednotlivé agenty pošlú centrálnemu koordinátorovi. Koordinátor potom v čiastkových plánoch nájde konflikty a navzájom sa doplňujúce časti a spojí na základe nich čiastkové plány do jedného celkového plánu.



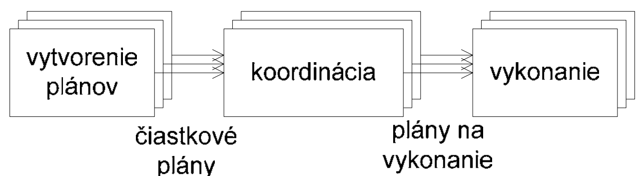
Obrázok 7-15. Centralizovaná koordinácia čiastkových plánov. Za koordináciu čiastkových plánov jednotlivých agentov je zodpovedný jeden agent.

Koordinátor identifikuje vzájomne závislé časti plánov jednotlivých agentov a nájde synchronizáciu potrebnú pre správne vykonanie jednotlivých čiastkových plánov. Môže tiež preusporiadať časti plánov, aby minimalizoval negatívne vzájomné vzťahy medzi akciami a naopak maximalizoval pozitívne účinky vzájomných vzťahov akcií.

Distribuovaná koordinácia čiastkových plánov

Pri distribuovanom plánovaní neexistuje žiadny centrálny agent, ktorý by vytváral plán pre celý multiagentový systém, či už od základu alebo na základe čiastkových plánov od ostatných agentov. Pri distribuovanom plánovaní každý agent vytvára svoj vlastný plán činnosti. Je potrebné aby agent identifikoval závislosti svojho plánu na plánoch ostatných agentov. Možné pozitívne závislosti je potrebné čo najviac využiť a zároveň odstrániť negatívne závislosti.

Tento problém nie je jednoduchý, keďže pri distribuovanej koordinácii čiastkových plánov nemusí existovať miesto, na ktorom je reprezentovaný úplný plán pre všetky agenty. Úplný plán môže existovať, avšak jeho časti môžu byť distribuované medzi jednotlivými agentmi.



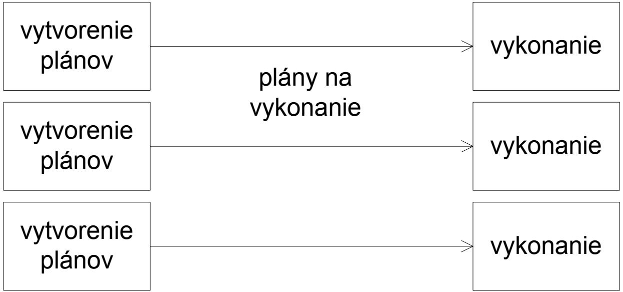
Obrázok 7-16: Distribuovaná koordinácia čiastkových plánov. Vytváranie plánov aj ich koordinácia prebieha distribuovane.

K distribuovanej koordinácii existujú dva základné prístupy. Prvou možnosťou je, že agenty oznamujú ostatným agentom svoje úmysly (svoje plány) a zároveň sa snažia prispôsobiť svoje plány úmyslom ostatných agentov. Po viacerých výmenách informácií o svojich úmysloch by mali plány jednotlivých agentov skonvergovať do stavu, kedy tvoria jeden konzistentný plán. Tento prístup využíva napríklad model PGP (angl. *Partial Global Planning*) (Durfee, 1991), kedy si agenty vymieňajú svoje čiastkové plány zbavené zbytočných podrobností a na základe svojho plánu a informácií od ostatných sa každý agent snaží vytvoriť kombinovaný globálny plán.

Druhý prístup je ten, že sa agenty nepokúšajú vytvoriť jeden globálny plán (hoci aj s distribuovanou reprezentáciou) Každý agent má svoj vlastný plán a iba ak identifikuje jeho závislosť na iných agentoch, nastáva koordinácia. Napríklad ak agent pre vykonanie nejakej akcie potrebuje pomoc iného agenta, požiada ho o spoluprácu.

Individuálne plány

Jednou z možností ako obísť problémy pri koordinácii plánov jednotlivých agentov je neriešiť ju. V niektorých prípadoch, ak sú ciele a akcie jednotlivých agentov dostatočne nezávislé, je možné zvoliť prístup individuálnych plánov. Každý agent si vytvorí vlastný plán postupu, ktorý sa potom snaží čo najpresnejšie vykonať. Pri vytváraní plánu je možné zohľadniť predpokladané správanie ostatných agentov, k explicitnej synchronizácii plánov ale nedochádza.



Obrázok 7-17. Individuálne plánovanie. Vykonávajú sa plány bez vzájomnej koordinácie.

7.4.2 Spôsob vytvárania plánov

V predchádzajúcej kapitole bola uvedená základná postupnosť krokov plánovania, ktorá sa skladá z vytvorenia individuálnych plánov, ich koordinácie a vykonania. Túto postupnosť je možné ďalej rozšíriť o činnosti, ktoré predchádzajú samotné vytváranie plánov a môžu vytvorenie plánu zjednodušiť.

Výsledná postupnosť krokov potom vyzerá nasledovne (deWeerdt, 2005):

1. Globálna dekompozícia úloh. Globálne ciele a úlohy sa postupne rozkladajú na jednoduchšie podúlohy, ktoré je možné prideliť jednotlivým agentom.
2. Pridelenie úloh. Identifikované podúlohy sa pridelia jednotlivým agentom.
3. Koordinácia pred plánovaním. Vytvoria sa pravidlá a obmedzenia pre jednotlivých agentov, ktoré majú za úlohu zabrániť vzniku nekompatibilných plánov.
4. Individuálne plánovanie. Každý agent vytvorí svoj individuálny plán, ktorý má dosiahnuť jeho ciele.
5. Koordinácia po plánovaní. Individuálne plány agentov sa skoordinujú aby sa vylúčili konflikty a zabezpečila spolupráca agentov pri akciách kde je to potrebné.
6. Vykonanie plánov. Agenty vykonajú svoje plány a spoja výsledky svojich podúloh.

Nie všetky prístupy k plánovaniu využívajú všetky kroky tejto postupnosti. V najjednoduchšom prípade je možné obmedziť sa na kroky 4. a 6., teda vytvorenie plánov a ich vykonanie. Aj v tomto prípade je však potrebné rozdeliť úlohy medzi jednotlivé agenty (krok 2.). Toto rozdelenie môže byť automatizované alebo vykonané ručne tvorcom multiagentového systému.

V skutočnosti, každý zo spomenutých šiestich krokov plánovania môže byť vykonaný ručne tvorcom systému, snáď až na vykonanie. (V prípade že je vykonávanie tiež prenechané človeku, dostávame sa na hranicu medzi multiagentovými systémami a oblasťou manažmentu a riadenia.) Niektoré činnosti, ako je dekompozícia cieľov na jednoduchšie podciele (krok 1., globálna dekompozícia úloh) alebo vytvorenie globálnych obmedzení a sociálnych noriem (krok 3., koordinácia pred plánovaním) sú v niektorých prípadoch pre stroj náročné. Napríklad systémy PGP (Durfee, 1991) (a jeho zovšeobecnenie GPGP (Lesser, 2004)) prípadne STEAM (Tambe, 1997), ktoré je možné zaradiť medzi vedúce v oblasti distribuovaného plánovania, sa stále opierajú o ručnú dekompozíciu úloh.

Jednotlivé kroky plánovania pochopiteľne nemusia nasledovať sekvenčne za sebou, ale môžu sa časovo prekrývať, čím sa umožní spätná väzba medzi jednotlivými krokmi. Napríklad ak sa pri vytváraní individuálnych plánov (krok 4.) zistí, že niektorý agent nedokáže splniť svoj cieľ, môže nastať prerozdelenie úloh (návrat ku kroku 2.). Rovnako ak sa počas vykonávania plánu (krok 6.) vyskytne nejaká neočakávaná udalosť, je potrebné vrátiť sa k predchádzajúcim krokom a upraviť plány alebo vytvoriť úplne nové.

K plánovaniu existuje v súčasnosti veľké množstvo prístupov, avšak je väčšinou možné ich zaradiť do troch hlavných skupín:

- Plánovanie založené na modeli STRIPS jeho rozšíreniach.
- Plánovanie založené na hierarchickej dekompozícii úloh a cieľov.
- Ručné vytváranie plánov.

7.4.3 STRIPS a jeho rozšírenia

Prístup, ktorý asi najviac ovplyvnil oblasť plánovania v umelej inteligencii, je použitý v systéme STRIPS (Fikes, 1971). Prístupy, ktorých základ možno nájsť práve v systéme STRIPS, sa stále úspešne využívajú. STRIPS je založený na situačnom počte.

Situačný počet vyjadruje stav sveta prostredníctvom konjunkcie literálov predikátovej logiky. Každý literál vyjadruje jednu podmienku na stav sveta a všetky podmienky opisujúce určitý stav sveta platia súčasne. Stav sveta môže byť napríklad:

S_1 = {pozícia(robot, miestnosť$_1$), dvere(miestnosť$_1$, miestnosť$_2$), ¬otvorené(miestnosť$_1$, miestnosť$_2$)}

Tento príklad ukazuje situáciu, kedy je robot v miestnosti 1 a z miestnosti 1 do miestnosti 2 vedú dvere, ktoré sú zatvorené.

Konjunkcia literálov opisujúcich určitý stav sveta zároveň určuje platnosť daného stavu. Stav sveta je platný iba vtedy, keď je konjunkcia literálov splniteľná, v opačnom prípade sa daný stav považuje za nekorektný a nemal by nastať. Napríklad stav S2 = {otvorené(miestnosť$_1$, miestnosť$_2$), ¬otvorené(miestnosť$_1$, miestnosť$_2$)} je zjavne neplatný, rovnako ako je neplatný stav S3 = {pozícia(robot, miestnosť$_1$), pozícia(robot, miestnosť$_2$)}, hoci na identifikáciu stavu S3 ako neplatného by sme museli pridať axióm, že robot nesmie byť v dvoch miestnostiach naraz. V „základnej“ verzii STRIPS musia korektnosť stavu zabezpečiť správne zadefinované operátory, ktoré stav modifikujú. V niektorých rozšíreniach tohto modelu sa do stavu sveta pridávajú ďalšie podmienky, ktoré nevyjadrujú konkrétny stav, ale slúžia pre obmedzenie množiny platných stavov. Tieto dodatočné podmienky potom predstavujú „prírodné zákony“ daného sveta.

Základná verzia STRIPS považuje svet za uzavretý, čo znamená, že ak sa v opise stavu sveta nevyskytuje niektorý literál, považujeme ho automaticky za neplatný. V príklade stavu S1 by sme potom vynechali literál ¬otvorené(miestnosť$_1$, miestnosť$_2$), ktorého platnosť by vyplývala z neprítomnosti literálu otvorené(miestnosť$_1$, miestnosť$_2$). Pri práci s uzavretým svetom si preto vystačíme iba s atomickými (jednoduchými) formulami, teda literálmi, ktoré neobsahujú negácie.

Ak chceme dokázať modelovať aj čiastočne neznámy stav sveta, je vhodné považovať svet za otvorený. V otvorenom svete neprítomnosť literálu neznamená automaticky platnosť jeho negácie a negácie literálov je potrebné vo vyjadrení stavu sveta explicitne uvádzať. V otvorenom svete teda vyjadrenie stavu $S_n = \{L_1, L_2, ..., L_m\}$ nepredstavuje jeden konkrétny stav, ale množinu všetkých stavov, v ktorých platia literály L_1 až L_m a ostatné literály majú ľubovoľnú hodnotu.

Stav sveta je modifikovaný pomocou akcií, ktoré sa nad týmto stavom vykonávajú. Akcie sú v STRIPS reprezentované pomocou operátorov. Každý operátor je vyjadrený ako usporiadaná trojica:

$$operátor = (pre, del, adds)$$

kde *pre*, *del* a *adds* sú množiny literálov. Množina *pre* obsahuje predpodmienku operátora, teda podmienky na stav sveta, v ktorom je možné operátor aplikovať. Pred vykonaním operátora je vždy nutné overiť platnosť jeho predpodmienky v súčasnom stave sveta. Množiny literálov *del* a *adds* obsahujú literály, ktoré sa po vykonaní operátora odstránia, respektíve pridajú do aktuálneho sveta. Modifikácia stavu sveta

vykonaním operátora je teda obmedzená pridávaním literálov z množiny *adds* a odstraňovaním literálov z množiny *del*.

Vykonanie operátora je možné formálne vyjadriť ako zobrazenie:

$$exec : Op \times \Sigma \rightarrow \Sigma$$

$$exec(op, \sigma) = \sigma'$$

kde *Op* je množina operátorov, Σ je množina prípustných stavov sveta, $op \in Op$ je operátor, σ je stav sveta pred vykonaním operátora a σ' je stav sveta po vykonaní operátora *op*.

Príklad STRIPS operátora môže vyzerať nasledovne:

```
operator PresuňSa(X, M1, M2)
  pre: pozícia(X, M1), dvere(M1, M2), otvorené(M1, M2)
  del: pozícia(X, M1)
  adds: pozícia(X, M2)
end
```

Operátor PresuňSa slúži na presunutie robota z jednej miestnosti do druhej, pričom medzi miestnosťami musia existovať otvorené dvere.

Stavy spolu s operátormi je možné stotožniť s reprezentáciou stavového stroja, kde operátory reprezentujú prechody medzi stavmi. Operátor môže naraz predstavovať viacero prechodov medzi stavmi, predstavuje všetky prechody, ktoré začínajú v stavoch, ktoré spĺňajú predpodmienku *pre* a končia v stavoch, ktoré vzniknú zo začiatočných stavov prechodu aplikovaním množín *del* a *adds*.

Plánovanie potom spočíva v hľadaní postupnosti operátorov, ktorých postupnou aplikáciou dokážeme modifikovať počiatočný stav sveta na požadovaný koncový stav, prípadne jeden z množiny koncových stavov. Takáto postupnosť operátorov sa nazýva plán.

Plány môžeme vytvárať dvomi spôsobmi: Môžeme prehľadávať stavový priestor problému alebo priestor možných plánov. Pri prehľadávaní stavového priestoru v ňom hľadáme cestu vedúcu od počiatočného stavu S_0 k niektorému z cieľových stavov S_f. Každý krok takejto cesty predstavuje aplikáciu operátora na stav S_{i-1} čím sa presunieme do stavu S_i. Platí teda: $S_i = exec(op_i, S_{i-1})$ a postupnosť operátorov $(op_1, op_2, ..., op_n)$ tvorí plán pre dosiahnutie stavu S_n zo stavu S_0.

Prehľadávanie stavového priestoru môžeme začať z počiatočného stavu a postupne hľadať stavy, do ktorých sa dokážeme aplikáciou jednotlivých operátorov dostať. Ak sa nám podarí dostať do niektorého z cieľových stavov, našli sme plán. Tento postup sa nazýva progresívne plánovanie. Iná možnosť je začať z cieľového stavu a hľadať stavy, z ktorých sa môžeme do koncového stavu dostať aplikovaním operátorov. Prehľadávanie končí, ak sa dostaneme do počiatočného stavu. Tento postup sa nazýva regresívne plánovanie. Výber medzi progresívnym a regresívnym plánovaním by mal závisieť od faktoru vetvenia pri doprednom alebo spätnom prehľadávaní, teda od toho, či je v priemere viac operátorov, ktoré môžeme aplikovať v danom stave alebo operátorov, ktoré vedú k dosiahnutiu určitého stavu. Účelom je samozrejme minimalizovať faktor vetvenia. Možná je aj kombinácia oboch prístupov: časť prehľadávania sa vykoná progresívne, časť regresívne a potom sa nájde prienik nájdených stavov.

Príkladom prístupu založeného na prehľadávaní stavového priestoru do hĺbky je Graphplan (Blum, 1997), ktorý je založený na vytváraní orientovaného grafu možných stavov, z ktorých sa dá dosiahnuť cieľový stav. Uzly grafu na nepárnych úrovniach (koreňom grafu počínajúc) predstavujú literály, ktoré ohraničujú aktuálny stav. Každá nepárna úroveň grafu predstavuje reprezentáciu stavu sveta, z ktorej je možné dosiahnuť cieľový stav. Uzly na párnych úrovniach grafu predstavujú akcie, ktoré je možné medzi danými stavmi vykonať. Väzby medzi jednotlivými uzlami grafu môžu byť vždy len medzi dvomi susednými úrovňami. Väzba vedúca od literálu k akcii vyjadruje predpodmienku akcie, väzba smerujúca od akcie k literálu na ďalšej úrovni vyjadruje účinky akcie (spôsobenie platnosti alebo neplatnosti literálu).

Pre menšie problémy, v ktorých nie je počet možných stavov príliš veľký, je dokonca možné zostrojiť celý stavový automat obsahujúci stavy zodpovedajúce všetkým možným stavom sveta a prechody medzi nimi, ktoré zodpovedajú jednotlivým operátorom. Takýto stavový automat je vlastne orientovaným grafom a je možné aplikovať na neho grafové algoritmy na nájdenie najkratšej cesty z počiatočného stavu do cieľového. Táto cesta je zároveň najkratším plánom.

Iná možnosť je prehľadávanie priestoru plánov, teda medzi všetkými možnými plánmi hľadáme práve ten, ktorý zmení počiatočný stav na cieľový. Toto prehľadávanie je možné robiť rôzne, medzi algoritmy na prehľadávanie priestoru plánov patrí napríklad čiastočne usporadúvajúce plánovanie, je však možné využiť aj subsymbolické prístupy, konkrétne genetické programovanie.

Algoritmus čiastočne usporadúvajúceho plánovania je založený na hľadaní plánov v podobe množiny krokov a množiny usporiadaní medzi krokmi. Ak je medzi krokmi vzťah usporiadania ($S_i \prec S_j$), znamená to že krok S_i predchádza pri vykonávaní plánu krok S_j. Relácia usporiadania je tranzitívna, čo znamená, že ak platí $S_i \prec S_k$ a zároveň $S_k \prec S_j$, potom musí platiť aj $S_i \prec S_j$.

Pri hľadaní plánu sa vychádza z plánu, ktorý neobsahuje žiadne usporiadania a obsahuje jediný krok, ktorého predpodmienkou je cieľový stav. Pre každý krok, ktorý už je v pláne, sa potom do plánu pridajú kroky, ktoré vedú k splneniu jeho predpodmienky. Ak je krok S_i pridaný pre splnenie predpodmienky kroku S_j, potom sa pridá aj usporiadanie $S_i \prec S_j$. Ak niektorý pridávaný krok S_k spôsobí zrušenie podmienky, ktorú krok S_i zabezpečuje pre predpodmienku kroku S_j, je potrebné tomu zabrániť. Väčšinou sa to rieši pridaním jedného z obmedzení $S_k \prec S_i$ alebo $S_j \prec S_k$, čím sa krok S_k presunie pred alebo za dvojicu krokov S_i a S_j. V prípade, že v pláne existujú obmedzenia, z ktorých dokážeme tranzitívnym uzáverom odvodiť $S_i \prec S_k$ a zároveň $S_k \prec S_j$, nie je možné krok S_k do plánu pridať a je potrebné nájsť iný krok alebo zmeniť aktuálny plán tak, aby niektoré z obmedzení, ktoré bránia pridaniu kroku S_k zaniklo. Podrobnejší opis algoritmu čiastočne usporadúvajúceho plánovania je možné nájsť napríklad v (Návrat, 2002).

Pred vykonaním čiastočne usporiadaného plánu je potrebné plán linearizovať, teda z čiastočného usporiadania akcií vytvoriť úplné usporiadanie. Čiastočne usporiadaný plán je tiež možné využiť na rozdelenie úloh medzi agenty v multiagentovom systéme. Usporiadania krokov plánu môžu povoliť vykonávanie niektorých krokov paralelne.

Napríklad ak sú dané usporiadania $S_i \prec S_k$, $S_j \prec S_k$, je možné kroky S_i a S_j vykonať súčasne, teda ich môžu vykonávať dva rôzne agenty. Vykonanie kroku S_k musí nasledovať až po dokončení krokov S_i a S_j a je potrebné zabezpečiť to vzájomnou koordináciou agentov.

Pri vytváraní dlhších plánov je niekedy vhodné eliminovať niektoré detaily pred tým, než sa vyriešia dôležitejšie úlohy. Toto je možné spraviť napríklad ohodnotením dôležitosti podmienok, ktoré treba dosiahnuť. Tento prístup používa systém ABSTRIPS (Sacerdoti, 1974). Pri vytváraní plánu sa najprv uvažujú najdôležitejšie podmienky. Plán vytvorený na základe najdôležitejších podmienok potom slúži ako základ plánu vytváraného po pridaní menej dôležitých podmienok. Postupné pridávanie podmienok pokračuje až kým nie je vytvorený kompletný plán.

STRIPS a jeho reprezentácia má niekoľko závažných obmedzení, ktoré sa snažia odstrániť jeho rôzne modifikácie a rozšírenia. Hlavné obmedzenia sú nasledovné:

- Statickosť prostredia. Stav prostredia je považovaný za nemenný a môžu ho ovplyvňovať iba akcie agentov. Ak chceme modelovať zmeny prostredia, ktoré sa dejú bez zásahu agentov, je potrebné pridať „agent prostredie", ktorý by dané zmeny vykonával.
- Diskrétnosť akcií. Účinky akcií sú aplikované v diskrétnych časových okamihoch a nie je možné jednoduchým spôsobom modelovať účinok, ktorý plynule mení stav prostredia. Napríklad nie je možné modelovať rovnomerný pohyb telesa priestorom, pri ktorom teleso plynule mení svoju pozíciu.
- Sekvenčné aplikovanie akcií. Model STRIPS nedovoľuje súčasnú aplikáciu viacerých akcií, akcie sú vykonávané sekvenčne. Je to nedostatok hlavne pre multiagentové systémy. Táto vlastnosť spôsobuje, že nie je možné modelovať súčasné pôsobenie viacerých akcií.
- Expresivita jazyka STRIPS. Jazyk STRIPS ponúka iba jednoduché vyjadrovacie schopnosti, ktoré sú niekedy až príliš obmedzujúce. Napríklad neponúka kvantifikátory alebo možnosť hierarchizovať akcie, neponúka ani možnosť priamo vyjadriť trvanie jednotlivých akcií. Zápis niektorých plánovacích problémov môže byť preto pomerne krkolomný a neprehľadný. Niektoré rozšírenia expresivity ako kvantifikátory, podmienečné efekty akcií alebo negatívne predpodmienky ponúka napríklad jazyk ADL (angl. *Action Description Language*) (Pednault, 1986).

Akcia ako vplyv

Ak chceme modelovať vzájomné interakcie medzi akciami vykonávanými jednotlivými agentmi v tom istom čase, môžeme to spraviť zmenou konceptu akcie. Namiesto toho, aby bola akcia vyjadrená jej účinkom, môžeme akciu vyjadriť ako vplyv na prostredie (Ferber, 1996). Skutočný účinok na prostredie sa potom určí ako výsledok vplyvov všetkých súčasne vykonávaných akcií na základe pravidiel, ktoré je možné považovať za prírodné zákony daného prostredia.

Pôvodný model vykonania operátora sa potom zmení z:

$$exec : Op \times \Sigma \rightarrow \Sigma$$

$$exec(op, \sigma) = \sigma'$$

na nový model:

$$exec : Op \times \Sigma \rightarrow \Gamma$$

$$exec(op, \sigma) = \gamma$$

Reakcia prostredia na vplyvy je potom vyjadrená nasledovne:

$$react : \Sigma \times \Gamma^{*} \rightarrow \Sigma$$

$$react\left(\sigma, \bigcup_{i} exec(op_{i}, \sigma) \right) = \sigma'$$

kde *Op* je množina operátorov, Σ je množina prípustných stavov sveta, Γ je množina možných vplyvov, $op \in Op$ je operátor, σ je stav sveta pred vykonaním operátora a γ je vplyv operátora *op*. σ' je stav sveta po aplikovaní vplyvov všetkých súčasne vykonaných akcií.

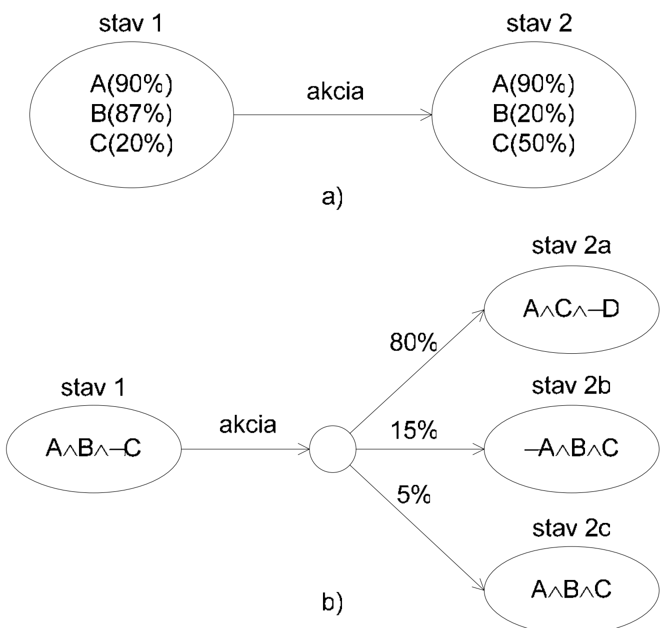
Takýto model akcií nevyžaduje, aby boli v operátoroch zahrnuté všetky ich účinky na prostredie, účinky sa presunú do zákonov prostredia. Na druhej strane zákony prostredia musia zahrnovať korektnú reakciu prostredia na všetky možné kombinácie vplyvov. Výhodou tohto modelu je, že jasne rozlišuje to, čo vykonáva agent od toho, čo sa deje v prostredí. Nevýhodou však je, že rovnako ako pôvodný model akcií pracuje v diskrétnych časových kvantách.

Plánovanie s neurčitosťou

Veľa domén je charakteristických neurčitosťou. Neurčitosť môže vyplývať z neúplných znalostí o prostredí alebo zo stochastického správania prostredia a objektov v ňom. Rovnako nie je možné vždy odhadnúť správanie ostatných agentov. Neurčitosť môže byť použitá aj na zjednodušené modelovanie zložitých javov za účelom zrýchlenia výpočtu alebo ak nemáme k dispozícii úplný model. Príkladom je predpoveď počasia, ktorá nám povie, s akou pravdepodobnosťou bude pršať. Neurčitosť do plánovania môžeme zaviesť dvomi spôsobmi: neurčitý stav sveta alebo neurčitý výsledok akcie.

Neurčitý stav sveta znamená, že jednotlivé tvrdenia o stave sveta nemajú absolútnu platnosť, ale sú platné iba s určitou pravdepodobnosťou. Výsledky jednotlivých akcií potom nenastavujú jednotlivé tvrdenia o stave sveta na hodnoty „pravda“ alebo „nepravda“, ale nastavujú pravdepodobnosť platnosti jednotlivých tvrdení (obrázok 7-18a). Výsledok akcie však nemusí iba priamo nastaviť hodnotu pravdepodobnosti, môže ju zvyšovať alebo znižovať oproti pôvodnej hodnote, a to na základe rôznych štatistických modelov. Predpodmienky akcií musia tiež zohľadňovať pravdepodobnosť, väčšinou je ako predpodmienka vyžadovaná určitá minimálna alebo maximálna pravdepodobnosť platnosti nejakého tvrdenia.

Neurčitosť sa môže vzťahovať aj na počiatočný stav sveta. V tomto prípade nevieme, v akom stave sa svet nachádza na začiatku plánovania, prípadne to vieme iba s určitou pravdepodobnosťou. Tento stav je možné jednoducho modelovať tak, že pred prvú akciu plánu vložíme akciu, ktorej výsledkom bude požadovaná pravdepodobnostná distribúcia stavov.



Obrázok 7-18. Porovnanie efektu akcie v prístupoch založených na neurčitom stave sveta (a) a neurčitom výsledku akcie (b).

Druhým spôsobom ako zohľadniť v plánovaní neurčitosť je zavedenie akcií s neurčitým výsledkom. Akcia má v tomto prípade viac možných výsledkov, ktoré nastanú s určitou pravdepodobnosťou (obrázok 7-18b). Tento spôsob vyjadrenia neurčitosti na rozdiel od predchádzajúceho nepracuje s pravdepodobnosťami na úrovni jednotlivých tvrdení o stave sveta ale s pravdepodobnosťami jednotlivých stavov sveta. Predpodmienky aj výsledky akcií sú vyjadrené rovnako ako pri plánovaní bez neurčitosti, s tým rozdielom, že výsledkov akcií je viac a majú pridelené pravdepodobnosti. Prístup založený na neurčitom výsledku akcií sa často využíva pri plánovaní s alternatívami, ale je možné ho využiť aj pri hľadaní dostatočne spoľahlivého plánu bez alternatív.

Porovnanie prístupov založených na neurčitosti stavu sveta a neurčitosti výsledku akcie je možné vidieť na obrázku 7-18. Tieto dva prístupy je možné aj kombinovať, akcia môže mať viacero možných výsledkov, pričom výsledkom akcie môže byť nastavenie pravdepodobností jednotlivých tvrdení o stave sveta.

Účelom plánovania s neurčitosťou je nájsť postupnosť akcií, ktoré dosiahnu cieľový stav s pravdepodobnosťou väčšou než je určená hranica, prípadne sa pravdepodobnosťou čo najviac priblížiť k stanovenej hranici. Podrobnejší prehľad aktuálneho stavu je možné nájsť napríklad v (Boutilier, 2003).

Markovovský rozhodovací proces

Medzi prístupy k plánovaniu s alternatívami patrí aj plánovanie pomocou markovovského rozhodovacieho procesu (MDP) (Bellman, 1961; Puterman, 1994). Ide tu vlastne iba o inú reprezentáciu toho istého problému. Plánovanie pomocou MDP je založené na vytvorení kompletnej reprezentácie stavového priestoru problému. Je vytvorený stavový stroj, ktorý obsahuje jeden stav pre každý možný stav prostredia.

Vykonávané akcie sú reprezentované ako vstup stavového stroja. Možné prechody medzi stavmi prostredia sú zároveň prechodmi medzi stavmi stroja a sú reprezentované prechodovou funkciou stroja. Prechodová funkcia stroja mapuje stav a akciu na ďalší stav, prípadne množinu stavov s určením pravdepodobnosti.

Rovnako ako prístupy spomenuté v predchádzajúcej podkapitole však má MDP problém s paralelizáciou plánov keďže sa nepredpokladá súčasné vykonávanie viacerých akcií a dĺžka trvania každej akcie je práve jedno časové kvantum.

V doménach, kde existuje veľké množstvo stavov, môže byť využitie tohto prístupu k plánovaniu neefektívne. Doména s n nezávislými stavovými premennými, ktoré môžu nadobúdať 2 hodnoty (pravda, nepravda), má 2^n rôznych stavov. Naopak, prístupy založené na operátoroch modifikujúcich stav prostredia ťažia z faktu, že operátor môže vyjadrovať viacero prechodov medzi stavmi a operátor spravidla ovplyvní iba časť stavových premenných. Predpodmienka ani výsledok operátora nemusí obsahovať všetky stavové premenné a namiesto konkrétnych stavov určujú množinu stavov.

Plánovanie s alternatívami

Pri vytváraní plánu často nemáme všetky informácie potrebné pre výber vhodnej alternatívy plánu. Toto vyplýva z nepredvídateľného správania prostredia alebo objektov (a agentov) v ňom. To či sa prostredie skutočne správa náhodne alebo iba nedokážeme odhadnúť jeho správanie v tomto prípade nie je dôležité. Odpovede na tento problém sú dve: priebežné plánovanie a plánovanie s alternatívami. Priebežné plánovanie znamená, že sa aktuálny plán nahradí novým vždy, keď sa zmenia predpoklady, na ktorých bol pôvodný plán postavený. Pre tento nový plán potom musí znova prebehnúť koordinácia.

Niekedy je vhodnejšie vytvoriť jeden plán, ktorý zároveň pokrýva viacero alternatív. Zatiaľ čo pri vytváraní lineárneho plánu sme sa museli vždy rozhodnúť ako bude plán pokračovať, pri plánovaní s alternatívami sa v mieste rozhodnutia plán rozdelí na viacero alternatívnych plánov. Z alternatívnych plánov sa potom pri vykonávaní plánu vyberie ten, ktorý zodpovedá aktuálnej situácii.

Rozdelenie plánu na viac alternatív nastáva buď na základe aktuálneho stavu sveta alebo na základe výsledku vykonanej akcie. Vetvenie na základe aktuálneho stavu sveta sa zvykne realizovať vložením senzorickej akcie, ktorá má viac možných výsledkov zodpovedajúcich jednotlivým možným stavom sveta alebo skupinám stavov. Pri vetvení na základe výsledku akcie sa vytvára alternatívna vetva plánu pre každý možný výsledok akcie.

Pri súčasných prístupoch k plánovaniu s alternatívami nastáva problém s veľkým faktorom vetvenia. Napríklad v príklade uvedenom autormi systému ZANDER (Majercik, 2003) so 4 možnými akciami a 2 krokmi vedúcimi k cieľu (s možnosťou paralelného vykonávania akcií) bolo potrebné vygenerovať binárny strom so 128 listami, všeobecne je zložitosť 2^{ak}, kde a je počet zvažovaných akcií v jednom kroku a k je minimálny počet krokov potrebných na dosiahnutie cieľa. Ak by sme vylúčili súčasné vykonávanie viacerých akcií, zmenšila by sa zložitosť na a^k. Zložitosť algoritmu je teda exponenciálna, a to nielen maximálna zložitosť, ale aj priemerná a minimálna.

Problémom pri využití doposiaľ existujúcich plánovačov s alternatívami v multiagentových systémoch je, že vytvárajú úplne usporiadané plány nevhodné pre paralelizáciu a tiež to, že pracujú v diskrétnom čase a každá akcia trvá práve jedno časové kvantum.

Senzorické akcie

Niekedy, predovšetkým v dynamicky sa meniacom prostredí, je potrebné do plánu zakomponovať aj rozhodovanie na základe aktuálnej situácie. Toto umožňuje napríklad zavedenie senzorických akcií. Senzorická akcia spôsobí zistenie aktuálnej hodnoty určitej časti stavu prostredia. V prípade, že uvažujeme pravdepodobnosť, môže senzorická akcia zvýšiť pravdepodobnosť, že túto hodnotu poznáme. Na základe možných výsledkov senzorickej akcie sa potom plán rozdelí na niekoľko alternatívnych vetiev, každá z týchto vetiev bude riešiť situáciu pre jeden z možných výsledkov senzorickej akcie. Koncept senzorických akcií je typický pre plánovanie s alternatívami (Napríklad už spomenutý systém ZANDER (Majercik, 2003)), v prípade plánovania bez neurčitosti sú senzorické akcie jediný spôsob ako docieliť vznik alternatív v pláne (napríklad Sensory Graphplan (Weld, 1998)).

Užitočnosť senzorických akcií je možné ukázať na jednoduchom príklade. Nasledujúca rovnica

$$\exists x \mid y = rnd(\ ) : P\left(\left(x \vee \overline{y}\right) \wedge \left(\overline{x} \vee y\right) = true\right) = 0{,}5$$

vyjadruje fakt, že pre zvolenú hodnotu premennej x a náhodnú hodnotu premennej y je pravdepodobnosť pravdivej hodnoty formule $\left(x \vee \overline{y}\right) \wedge \left(\overline{x} \vee y\right)$ rovná 50%. Avšak situácia sa zmení ak vymeníme poradie premenných:

$$y = rnd(\ ) \mid \exists x : P\left(\left(x \vee \overline{y}\right) \wedge \left(\overline{x} \vee y\right) = true\right) = 1{,}0$$

Pre náhodne vygenerovanú premennú y je vždy možné zvoliť hodnotu premennej x tak, aby bola formula $\left(x \vee \overline{y}\right) \wedge \left(\overline{x} \vee y\right)$ pravdivá. Predpokladom však je, že hodnotu premennej x volíme na základe zistenia aktuálnej hodnoty premennej y. Ak by sme vytvárali plán, ktorého úlohou je dosiahnuť pravdivosť uvedenej formule zvolením hodnoty premennej x, vložením senzorickej akcie na zistenie hodnoty premennej y a rozdelením plánu na 2 paralelné vetvy na základe výsledku senzorickej akcie je možné zvýšiť pravdepodobnosť úspechu plánu z 50% na 100%.

7.4.4 Hierarchická dekompozícia

Všetky prístupy v tejto kategórii využívajú ručnú dekompozíciu akcií na jednoduchšie, čo uľahčuje plánovaču prácu pri hľadaní vhodnej postupnosti krokov plánu. Plánovač totiž dostáva dodatočnú informáciu, ktorá obmedzuje množinu akcií, ktorých výber pripadá do úvahy, čím sa zrýchľuje proces vytvárania plánu.

Základ tohto riešenia bol položený už samotnými autormi systému STRIPS zavedením makrooperátorov. Makrooperátory boli vo vnútri zložené z jednoduchších operátorov na nižšej hierarchickej úrovni. Na vyššej hierarchickej úrovni však makrooperátory vystupujú ako jednoduché operátory, čím sa zrýchľuje vytváranie plánov na tejto úrovni.

Vylepšením tohto prístupu je HTN (angl. *Hierarchical Task Network*) (Sacerdoti, 1977). Základnom tohto prístupu je koncept úloh. Úlohy môžu byť zložené alebo primitívne. Pre každú zloženú úlohu sú zadefinované podúlohy, ktoré treba splniť pre splnenie nadradenej úlohy. Jednotlivé úlohy medzi sebou môžu mať rôzne vzťahy. Môže ísť o alternatívne úlohy, z ktorých stačí pre splnenie nadradenej úlohy splniť jednu, môže byť vyžadované splnenie všetkých podúloh a môžu byť určené aj ďalšie obmedzenia, napríklad na poradie vykonávania podúloh.

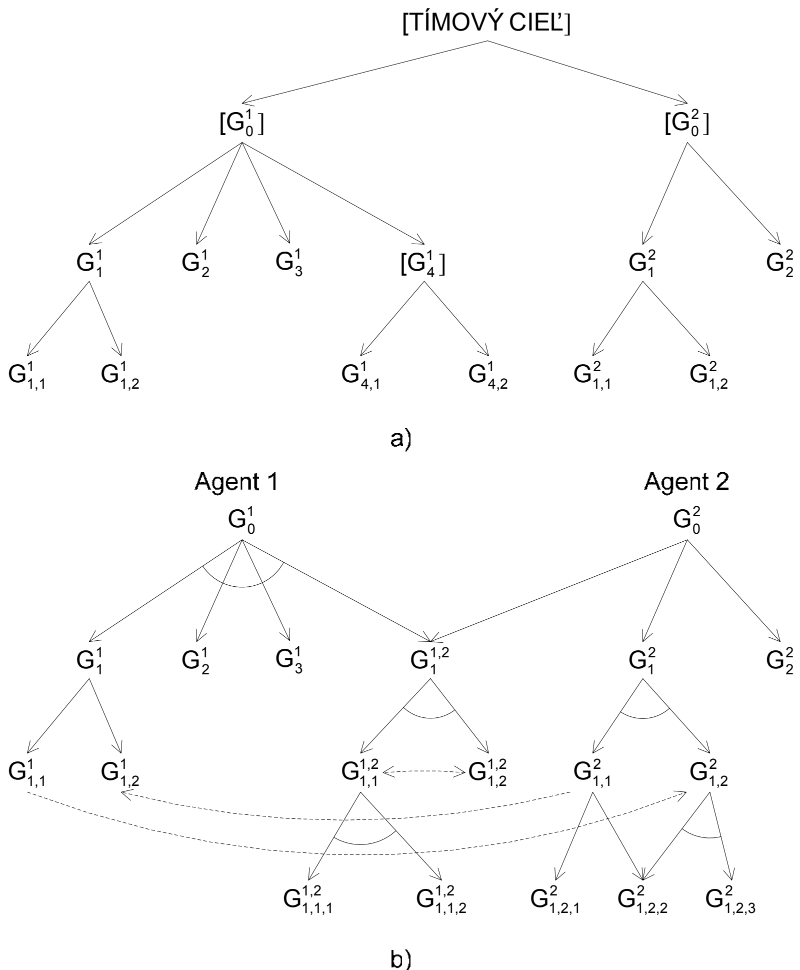
V ďalších rozšíreniach, ktoré zavádzajú systémy založené na HTN, sú vzťahy (obmedzenia) medzi úlohami ďalej doplnené o modelovanie vzťahov k zdrojom, ktoré sú pre jednotlivé úlohy použité. Týmto je zabezpečené že sa akcie používajúce ten istý zdroj nevykonajú naraz a že pred začatím úlohy používajúcej určitý zdroj je vykonaná úloha, ktorá tento zdroj vytvorí (ak nie je daný zdroj k dispozícii od začiatku).

Pri vytváraní plánu sa začína od cieľov, ktoré treba splniť. Pre každý z týchto cieľov je vytvorená úloha. Každá zložená úloha sa postupne rozkladá na jednoduchšie až na úroveň primitívnych úloh. Na základe obmedzení medzi úlohami sa z primitívnych úloh vytvorí čiastočne usporiadaný plán.

Základný algoritmus HTN je nasledovný:

1. Na základe cieľov vlož do plánu (P) hlavné úlohy
2. Ak P obsahuje iba primitívne úlohy, skonči.
3. Vyber jednu zloženú úlohu t z plánu P
4. Nahraď úlohu t jej podúlohami
5. Nájdi interakcie medzi úlohami v P a urči spôsob ich riešenia. Ak to nie je možné, skonči s chybou.
6. Prejdi na krok 2.

Prvým rozšírením HTN pre multiagentové systémy bol systém NOAH (Corkill, 1979). V tomto rozšírení každý agent vytvára svoj vlastný plán, má však k dispozícii špeciálne úlohy, ktoré reprezentujú plány iných agentov a tiež špeciálne úlohy pre synchronizáciu agentov. Časti stromu dekompozície, na základe ktorého sa vytvárajú plány, môžu byť známe len niektorým agentom, avšak celý strom musí v systéme byť, hoci aj rozdelený medzi agenty. Každý agent si uchováva čiastočnú reprezentáciu plánov ostatných agentov a na základe nej dokáže určiť kompatibilné miesta plánov, kde je možná spolupráca.



Obrázok 7-19. Príklad hierarchickej dekompozície systému STEAM (a) a PGP (b). Zatiaľ čo v systéme STEAM existuje jediný strom dekompozície, v PGP má každý agent iný cieľ a teda aj iný strom dekompozície. V diagramoch sú vynechané závislosti na zdrojoch, v strome pre STEAM (a) sú hranatými zátvorkami označené tímové úlohy (úlohy pre viac agentov). V strome pre PGP (b) sú zobrazené závislosti medzi úlohami.

Koordináciu agentov ďalej rozširuje systém PGP (angl. *Partial Global Planning*), ktorý zavádza organizáciu údajov o plánoch a cieľoch agentov do troch úrovní. Na najnižšej úrovni sú individuálne plány jednotlivých agentov (nazývané „lokálne plány"), ktoré obsahujú všetky podrobnosti potrebné na ich vykonanie. O úroveň vyššie sa nachádzajú takzvané „plánovacie uzly" (v tejto terminológii je možné stotožniť ich s agentmi), ktoré obsahujú zjednodušené individuálne plány zbavené zbytočných detailov Na najvyššej úrovni sú potom „čiastkové globálne plány" obsahujúce informácie o stave globálnych úloh. Sú tu vyjadrené ciele agenta, aktuálne úlohy, ktoré rieši a informácie o spôsobe ich riešenia, informácie o vzájomnej interakcii agentov. Tieto informácie má každý agent o sebe, ale komunikáciou získava tieto informácie aj o ostatných agentoch. Čiastkové globálne plány sú používané pre

koordináciu individuálnych činností medzi agentmi. V ideálnom stave sú čiastkové globálne plány jednotlivých agentov kompatibilné a spolu tvoria distribuovanú reprezentáciu celkového globálneho plánu. V prípade že agent detekuje, že správanie iného agenta nezodpovedá informáciám, ktoré o ňom má vo svojom čiastkovom globálnom pláne, zmení svoj plán tak, že v ňom ďalej dotyčný agent neuvažuje. Tým sa minimalizuje riziko vykonávania nekompatibilných plánov.

Prístup SharedPlan (Grosz, 1996) ďalej zavádza mechanizmy pre koordináciu činnosti jednotlivých agentov prostredníctvom vyjednávania o rozdelení úloh medzi agenty. Vyjednávanie je založené na posielaní požiadaviek agentom, ktorí sú schopní vykonať podciele jednotlivých cieľov. Agent môže mať „čiastočný úmysel" zúčastniť sa na vykonávaní jednotlivých plánov. Ako príklad systému založeného na prístupe SharedPlan možno uviesť STEAM (Tambe, 1997).

Prístupy založené na HTN možno rozdeliť na dve skupiny: prístupy s cieľmi agentov a prístupy s tímovým cieľom. Prístupy s tímovým cieľom (napríklad STEAM) vychádzajú zo spoločného tímového cieľa, ktorý je rozdelený na jednotlivé podúlohy a tieto podúlohy sa rozdeľujú medzi jednotlivé agenty. Prístupy so samostatnými cieľmi agentov (napríklad PGP) hľadajú spoločné časti hierarchií úloh jednotlivých agentov, agenty majú vlastné ciele, ktoré sa snažia dosiahnuť. Spolupráca v PGP vzniká keď majú stromy dekompozície jednotlivých agentov spoločné alebo vzájomne závislé časti. Príklady hierarchií úloh systémov PGP a STEAM je možné vidieť na obrázku 7-19.

Hoci je množina plánov, ktoré sa takto dajú vytvoriť prístupmi založenými na HTN často považovaná za väčšiu, než poskytujú systémy založené na operátoroch typu STRIPS (Erol, 1994), je to skôr teoretická výhoda. Ak totiž neobmedzíme dĺžku generovaného plánu, generovanie plánu v HTN sa stane nerozhodnuteľným problémom – nie je možné vytvoriť algoritmus, ktorý pre všetky prípady rozhodne, či plán existuje. V prípade, že dĺžku generovaného plánu obmedzíme, je možné dokázať, že obidva prístupy (operátory typu STRIPS a HTN) dokážu generovať tú istú množinu plánov.

Faktom ale ostáva, že aj keď je množina plánov vyjadriteľná pomocou HTN rovnaká ako množina plánov vyjadriteľná operátormi typu STRIPS, hierarchické vzťahy akcií sa prirodzenejšie zapisujú v prístupoch HTN od neho odvodených. HTN je teda možné považovať za spôsob ako jednoduchším spôsobom dodať plánovaču dodatočné informácie o hierarchických vzťahoch jednotlivých akcií, ktoré umožnia zjednodušiť vytváranie plánov.

Literatúra

AARONSON, S. – KUPERBERG, G (2005) The Complexity Zoo, www.complexityzoo.com (26. august 2005)

BELLMAN R.E. (1961) Adaptive Control Processes: A Guided Tour. Princeton University Press, Princeton, New Jersey

BLUM, A. – FURST, M. (1997) Fast Planning Through Planning Graph Analysis, Artificial Intelligence 90, Elsevier Science Publishers Ltd., Essex, UK, pp. 281—300.

BOUTILIER, C. – DEAN, T. – KOENIG, S. (2003) Artificial Intelligence 147, (AI2003), Special issue on planning with uncertainty and incomplete information. Elsevier Science Publishers Ltd., Essex, UK.

BROOKS, R.A. – CONNELL, J.H. – NING, P. (1988) HERBERT: A second generation mobile robot, AI memo 1016, MIT, Cambridge, USA

BROOKS, R.A. (1990) Elephants don't play chess. In: *Designing Autonomous Agents*, Maes, P., editor, The MIT Press: Cambridge, MA, pp. 3—15

BROOKS, R.A. (1991a) Intelligence without reason. In: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI–91)*, Morgan Kaufmann, Sydney, Australia, pp. 569—595.

BROOKS, R.A. (1991b) Intelligence without representation. In: *Artificial Intelligence 47*, Elsevier Science Publishers Ltd., Essex, UK pp. 139—159.

BROOKS, C.H. – DURFEE, E.H. (2002) Congregating and market formation. In Proceedings of the first international joint conference on Autonomous agents and multiagent systems, pp. 96—103.

BYLANDER, T. (1994) The computational complexity of propositional STRIPS planning. In: *Artificial Intelligence 69*, Elsevier Science Publishers Ltd., Essex, UK, pp. 161—204.

CHAVEZ, A. – MAES, P. (1996) Kasbah An agent marketplace for buying and selling goods. In: *First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96)*, London, UK, pp. 75—90.

COHEN, P.R. – LEVESQUE, H.J. (1986) Persistence, Intention, and Commitment. In: *Timberline Workshop on Planning and Practical Reasoning*. Los Altos, Calif.: Morgan Kaufman Publishers, Inc.

CORKILL, D.D. (1979) Hierarchical planning in a distributed environment. In: *IJCAI-79*

DELLAROCAS, C. – KLEIN, M. (1999) Civil agent societies: Tools for inventing open agent-mediated electronic marketplaces. In: *Agent Mediated Electronic Commerce (IJCAI Workshop)*, pp. 24—39.

DE WEERDT, M. – TER MORS, A. – WITTEVEEN, C. (2005) Multi-agent Planning: An introduction to planning and coordination. In: *Handout of the European Agent Summer School*, pp. 1—32.

DROGOUL, A. – CORBARA, B. – LALANDE, S. (1995) MANTA: New Experimental Results on the Emergence of (Artificial) Ant Societies. In: *Artificial Societies: the computer simulation social life*, Gilbert, N., Conte, R., editors, University College of London Press, London, UK, pp. 190—211.

DURFEE, E.H. – LESSER, V.R. (1991) Partial Global Planning: A Coordination Framework for Distributed Hypothesis Formation. In: *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, num. 5, pp. 1167—1183.

EROL, K. – NAU, D. – HENDLER, J. (1994) HTN planning: Complexity and expressivity. In: *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, AAAI Press, Menlo Park, USA.

FERBER, J. – MÜLLER, J.P. (1996) Influences and Reaction: A Model of Situated Multiagent Systems. In: *Proceedings of the Second International Conference on Multi-agent Systems*, AAAI Press, pp. 72—79.

FERBER, J. (1999) Multi-Agent Systems: An introduction to Distributed Artificial Intelligence. Addison Wesley Longmann, Essex England, ISBN 0-201-36048-9

FIKES, R.E. – NILSSON, N.J. (1971) STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, Artificial Intelligence 2, Elsevier Science Publishers Ltd., Essex, UK, pp. 189—208.

GERHARDT, M. – SCHUSTER, H. (1995) Das digitale Universum – Zelluläre Automaten als Modell der Natur, Vieweg, Braunschweig

GRIFFITHS, N. (2003) Supporting cooperation through clans. In: *Cybernetic Intelligence, Challenges and Advances – Proceedings of the 2nd IEEE Systems, Man and Cybernetics*, UK & RI Chapter Conference.

GROSZ, B. – KRAUS, S. (1996) Collaborative Plans for Complex Group Action, In Artificial Intelligence 86(2), Elsevier Science Publishers Ltd., Essex, UK, pp. 269—357.

HORLING, B. – MAILLER, R. – SHEN, J. – VINCENT, R. – LESSER V. (2003) Using Autonomy, Organizational Design and Negotiation in a Distributed Sensor Network. In: *Distributed Sensor Networks: A multiagent perspective*, V. Lesser, C. Ortiz, and M. Tambe, editors, Kluwer Academic Publishers, pp. 139—183.

HORLING, B. – LESSER, V.R. (2005) A Survey of Multi-Agent Organizational Paradigms, *Knowledge Engineering Review* 2005. To appear.

ISHIDA, T. – GASSER, L. – YOKOO, M. (1992) Organization selfdesign of distributed production systems. In: *IEEE Transactions on Knowledge and Data Engineering*, 4(2): pp. 123—134

JENNINGS, N. (1995) Controlling cooperative problem solving in industrial multi-agent systems. Artificial Intelligence 75(2), pp. 195—240.

KÖSTLER, A. (1967) The Ghost In The Machine. Hutchinson Publishing Group, Arkana, London.

KOLP, M. – GIORGINI, P. – MYLOPOULOS, J. (2001) A Goal-Based Organizational Perspective on Multi-Agent Architectures. In: *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, Meyer, J.J., Tambe, M., editors, pp. 146—158.

LESSER, V.R., ET AL.(2004) Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. In: *Autonomous Agents and Multi-Agent Systems*, vol. 9, num. 1, Kluwer Academic Publishers, pp. 87—143.

MADANI, O. – HANKS, S. – CONDON, A. (2003) On the undecidability of probabilistic planning and related stochastic optimization problems. In: *Artificial Intelligence 147, special issue on planning with uncertainty and incomplete information*, Elsevier Science Publishers Ltd., Essex, UK.

MAJERCIK, S.M. – LITTMAN, M.L. (2003) Contingent planning under uncertainty via stochastic satisfiability. In: *Artificial Intelligence 147, special issue on planning with uncertainty and incomplete information*, Elsevier Science Publishers Ltd., Essex, UK.

MAŘÍK, V. – ŠTĚPÁNKOVÁ, O. – LAŽANSKÝ, J., A KOL. (1993) Umělá inteligence 1, Academia, Praha. ISBN 80-200-0496-3, ISBN 80-200-0502-1

MAŘÍK, V. – ŠTĚPÁNKOVÁ, O. – LAŽANSKÝ, J., A KOL. (2003a) Umělá inteligence 3, Academia, Praha. ISBN 80-200-0472-6, ISBN 80-200-0502-1

MAŘÍK, V. – ŠTĚPÁNKOVÁ, O. – LAŽANSKÝ, J., A KOL. (2003b) Umělá inteligence 4, Academia, Praha. ISBN 80-200-1044-0, ISBN 80-200-0502-1

MATURANA, F. – SHEN, W. – NORRIE, D. (1999) Metamorph: An adaptive agent-based architecture for intelligent manufacturing. In: *International Journal of Production Research*, 37(10), pp. 2159—2174.

MINTZBERG, H. (1979) The Structuring of Organisations. McGraw Hill.

NÁVRAT, P., A KOL. (2002) Umelá inteligencia, Vydavateľstvo STU, Bratislava, Slovensko. ISBN 80-227-1645-6

PEDNAULT, E.P.D. (1986) Formulating multiagent, dynamic-world problems in the classical planning framework. In: *Reasoning about Actions and Plans, Proceedings of the 1986 Workshop*, Georgeff, M., Lansky, A. L., editors, Timberline, USA, pp. 47—82.

POINCARE, J.H. (1892-99) Les méthods nouvelles de la mécanique céleste, Gauthier-Villars, Paris, 1892 vol. I , 1893 vol. II, 1899 vol. III.

PUTERMAN, M.L. (1994) Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York.

ROMELAER, P. (2002) Organization: A Diagnosis Method. Technical Report 78, University Paris IX Dauphine, Crepa Laboratory.

SACERDOTI, E.D. (1974) Planning in a hierarchy of abstraction spaces. In: *Artificial intelligence 5*. Elsevier Science Publishers Ltd., Essex, UK

SACERDOTI, E.D. (1977) A Structure for Plans and Behavior, Elsevier-North Holland.

SHEHORY, O. – KRAUS, S. (1998) Methods for task allocation via agent coalition formation. In: *Artificial Intelligence 101(1-2)*, Elsevier Science Publishers Ltd., Essex, UK, pp. 165—200.

SIMS, M. – GOLDMAN, C. – LESSER, V.R. (2003) Self-Organization through Bottom-up Coalition Formation. In: *Proceedings of Second International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2003)*, Melbourne, AUS, pp. 867—874

SMITH, R.G. (1980) The contract net protocol: High-level communication and control in a distributed problem solver. In: *IEEE Transctions on Computers*, 29(12), pp. 1104—1113.

SYCARA, K. – DECKER, K. – WILLIAMSON, M. (1997) Middle-agents for the internet. In: *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pp. 578—583.

TAMBE, M. (1997) Towards flexible teamwork. Journal of Artificial Intelligence Research, vol. 7, pp. 83—124.

WELD, D.S. – ANDERSON, C.R. – SMITH, D.E. (1998) Extending Graphplan to Handle Uncertainty & Sensing Actions. In: *Proceedings of AAAI '98*.

YADGAR, O. – KRAUS, S. – ORTIZ, C. (2003) Scaling up distributed sensor networks: cooperative large-scale mobile-agent organizations. In: *Distributed Sensor Networks: a multiagent perspective*, Lesser, V., Ortiz, C., Tambe, M., editors, Kluwer publisching, pp. 185—218.

ZEGHAL, K. – FERBER, J. – ERCEAU, J. (1993) Symmetrical, transitive and recursive force: A representation of interaction and commitments. In: *IJCAI workshop on coordinated autonomous robots*, Chambery.

8 PREZENTÁCIA INFORMÁCIÍ A ZNALOSTÍ NA WEBE SO SÉMANTIKOU

Spolu s rýchlym vývojom nových technológií v oblasti informatiky a informačných technológií zákonite vznikajú nové slová, slovné spojenia alebo už existujúce pojmy nadobúdajú ďalší význam. Dôvodom je samozrejme snaha tvorcov týchto technológií o ich čo najpriliehavejšie pomenovanie. Nové pomenovania pre technológie prichádzajú mnohokrát rýchlejšie, ako pre ne možno nájsť najlepší (najpriliehavejší) slovenský ekvivalent. Možno povedať, že na webe so sémantikou tieto tvrdenia platia dvojnásobne, preto niektoré pojmy sú preložené tak, aby čo najlepšie „zapasovali" do predkladaného textu.

Pojmy uvádzané v tomto dokumente sme zvolili nasledovne: pre anglický výraz „web" je použitý rovnaký výraz, teda *web*, ktorý sa postupom času etabloval aj v slovenčine. Anglické „query" prekladáme ako *dopyt*. Ako prídavné meno od slova web používame slovo *webový* (podľa odporúčania Jazykovedného ústavu Ľudovíta Štúra SAV). Slovné spojenie „semantic web" v tejto práci prekladáme ako *web so sémantikou*. Sme si však vedomí, že nami vybrané preklady anglických termínov (najmä slovné spojenie „web so sémantikou") nemusia byť tými najvýstižnejšími alebo úplne správnymi, preto uvádzame aspoň pár najčastejšie sa vyskytujúcich alternatívnych názvov, ktoré v tejto práci použité nie sú ale možno ich nájsť inde: sémantický web, web s významom, pavučina s významom, rozumný web.

8.1 Ontológie

Termín „ontológia" sa používa mnoho rokov. Od začiatku 90-tych rokov sa ontológie stali populárnou témou pre skúmanie v oblasti znalostného inžinierstva, spracovania prirodzeného jazyka a znalostnej reprezentácie.

Úlohou ontológií je reprezentovať znalosti takým spôsobom, aby bolo možné komunikovať medzi rôznymi strojmi a medzi strojmi a ľuďmi na úrovni znalostí, na rozdiel od súčasnosti, kedy vzájomná komunikácia prebieha na úrovni údajov.[18]

Táto kapitola sa snaží priblížiť ontológie a prestaviť ich najmä tak, ako ich v priebehu času chápali „informatici", ako sa menila ich definícia, aké typy ontológií poznáme, resp. ako ich možno a podľa akých kritérií rozdeliť.

8.1.1 Definícia ontológií

Pojem *ontológia* bol vypožičaný z filozofie, kde je ontológia definovaná ako „odvetvie metafyziky zaoberajúce sa identifikáciou druhov vecí (v ich najvšeobecnejšom ponímaní), ktoré skutočne existujú. Preto *ontologické záväzky* z filozofického pohľadu obsahujú ako explicitné tvrdenia, tak aj implicitné predpoklady o existencii entít, substancií alebo bytostí jednotlivých druhov."

Neches a kol. (1991) navrhli jednu z prvých definícií ontológie v nasledovnom znení:

„Ontológia definuje základné pojmy a vzťahy obsahujúce slovník oblasti záujmu, ako aj pravidlá pre kombináciu pojmov a vzťahov, ktoré definujú rozšírenia tohto slovníka."

Túto definíciu jej autori navrhli v snahe predstaviť si výpočtové prostredie, kde znalostné systémy, nástroje umelej inteligencie a „klasický" softvér budú navzájom interagovať na úrovni znalostí a prostriedkom ich komunikácie budú zdieľané ontológie, ktoré tvoria rozhrania (alebo rámce) k bázam znalostí.

V roku 1993 prichádza Gruber s jeho definíciou ontológie, v ktorej uvádza jej vzťah s konceptom – pojmom používaným vo formálnych znalostných systémoch. Podstata formálne reprezentovanej znalosti o doméne všeobecne pozostáva z objektov a vzťahov medzi objektmi (t.j. univerzum diskurzu) a je založená na konceptualizácii domény. *Konceptualizácia* je teda chápaná ako „abstraktný, zjednodušený pohľad na svet", ktorý je formálne reprezentovaný. Vychádzajúc z predchádzajúcich vysvetlení pojmov Gruber tvrdí, že:

„ontológia je explicitná špecifikácia konceptualizácie."

Na explicitnú reprezentáciu univerza diskurzu je okrem už uvedených pojmov potrebný aj *slovník*. Hlavnou výhodou Gruberovej definície je, že je definovaná ako *explicitná*, tzn. je verejne prístupná, nie implicitne včlenená do bázy znalostí. Preto môžeme oddeliť úroveň symbolov aplikácie, na ktorej sú reprezentované interné algoritmy, od úrovne znalostí, kde sú definované komunikačné protokoly. Gruberova definícia je založená na predpoklade, že každý systém, ktorý začleňuje formálne reprezentované znalosti, je explicitne alebo implicitne viazaný na konceptualizáciu.

Borst a kol. (1997) ponúka prepracovanú Gruberovu definíciu:

„Ontológie sú definované ako formálna špecifikácia zdieľanej konceptualizácie."

Studer, Benjamins a Fensel (1998) spájajú obidve definície dovtedy používané v literatúre a ponúkajú nasledujúcu definíciu:

---

[18]Rozdiel medzi dátami, informáciami a znalosťami je vysvetlený napríklad v (Sklenák et al, 2001).

„Ontológie sú definované ako explicitná formálna špecifikácia zdieľanej konceptualizácie.“

Fensel (2000) hľadá rozdiely medzi ontológiami a databázami, a nachádza nasledujúce:

- Jazyk pre definovanie ontológií je syntakticky a sémanticky bohatší ako bežné databázové prístupy.
- Informácia, ktorá je opísaná ontológiou, pozostáva z pološtruktúrovaného textu v prirodzenom jazyku, nejde o tabuľkové údaje.
- Ontológia musí byť zdieľanou terminológiou založenou na konsenze, pretože je použitá pre zdieľanie a výmenu informácií.
- Ontológia poskytuje doménovú teóriu a nie štruktúru kontajneru údajov.

V predchádzajúcich odsekoch sme uviedli niekoľko viac či menej odlišných definícií ontológie, no všetky uvedené definície majú spoločné dve vlastnosti: *formálnosť* a *konsenzus*. Všetky definície kladú dôraz na reprezentáciu znalostí konsenzuálnym spôsobom, prinajmenšom medzi špecifikovanou skupinou tak, aby bolo zdieľanie znalostí vôbec možné a implementovateľné.

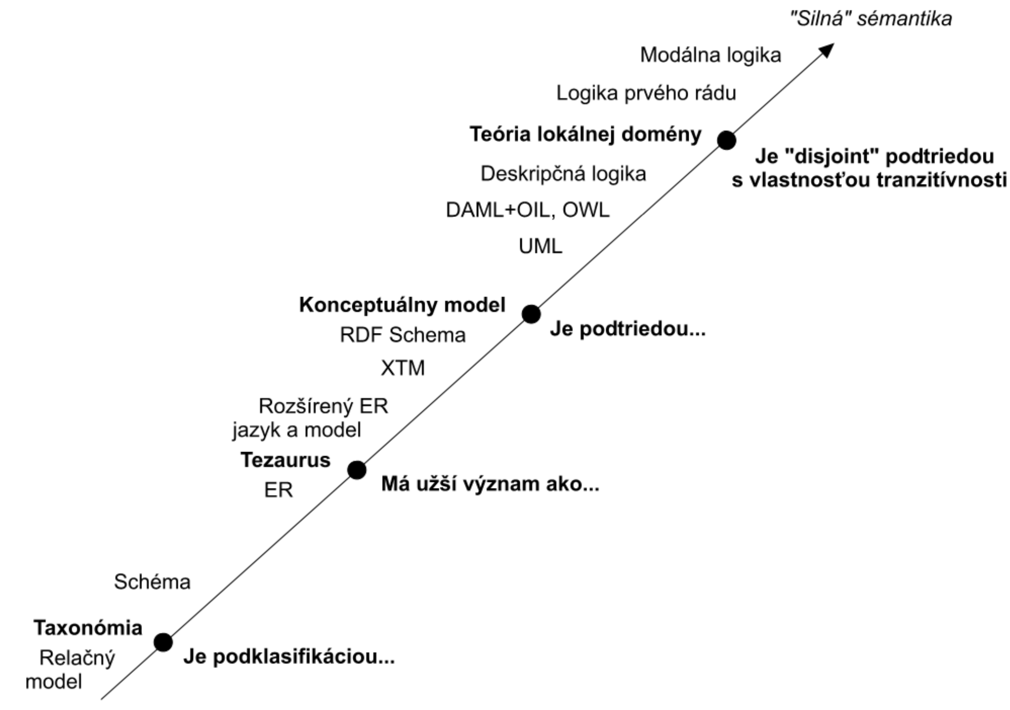
8.1.2 Typy ontológií

Ak sa na ontológie pozeráme z hľadiska predmetu konceptualizácie (Ushold a Gruninger, 1996), identifikujeme päť typov:

1. *Doménová* ontológia tvorí konceptuálny základ znalostí z vecnej domény.
2. *Úlohová* ontológia.
3. *Generická* ontológia sa vyznačuje širokým, až univerzálnym záberom, no v žiadnej z vecných oblastí nejde do takej hĺbky, aby bola dostatočným konceptuálnym základom pre bázy znalostí reálnych aplikácií.
4. *Aplikačná* ontológia je súčasťou konkrétnej aplikácie, a zväčša obsahuje doménovú aj úlohovú zložku.
5. *Reprezentačná* ontológia definuje jazyk pre reprezentáciu znalostí. Ak sú týmito znalosťami samotné ontológie, danú ontológiu nazývame *meta-ontológiou*.

8.1.3 Spektrum ontológií

Cieľom spektra ontológií, prvýkrát uvedeného McGuinnessovou (2002) a zobrazeného na obrázku 8-1 (upravené podľa Dacontu, Obrsta a Smitha, 2003), je vytvoriť rámec pre vzájomné porovnávanie sémantickej bohatosti klasifikačných a znalostných modelov z rôznych konceptuálnych paradigiem, používaných na reprezentáciu, klasifikáciu a „zjednoznačnenie“ sémantiky v rámci alebo medzi konkrétnymi doménami predmetnej veci (témy). Vľavo dole sa nachádza jeden z pólov sémantickej bohatosti – „slabá sémantika“, smerom doprava a nahor sa bohatosť zvyšuje. Na „slabšej“ strane môžeme vyjadriť len jednoduchý význam, na „silnejšej“ strane je možné vyjadriť význam ľubovoľne komplexný. Z uvedeného vyplýva, že to, čo je známe ako ontológia, môže siahať od jednoduchej *taxonómie* (znalosti s minimálnou hierarchickou štruktúrou alebo štruktúrou typu predok/potomok), cez *tezaurus* (slová a synonymá), *konceptuálny model* (s oveľa komplexnejšími znalosťami), až po *logickú teóriu* (s veľmi bohatými, komplexnými, konzistentnými a zmysluplnými znalosťami).



Obrázok 8-1. Spektrum ontológií (podľa Dacontu, Obrsta a Smitha, 2003).

Prínos ontológií nie je v spracovaní znalostí, ale v poskytnutí všeobecných a zdieľaných poznatkov o nejakej oblasti, kde takého poznatky možno prenášať rovnako medzi ľuďmi ako aj strojmi (aplikáciami), v odhaľovaní medzier a zlepšovaní prenosu skrytých znalostí.

8.1.4 Štruktúra ontológií

Podľa Dacontu, Obrsta a Smitha (2003) ontológia vo všeobecnosti obsahuje:

- *Triedy* (všeobecné veci) v mnohých doménach záujmu
- *Inštancie* (konkrétne veci)
- *Vzťahy medzi týmito vecami*
- *Vlastnosti* (a hodnoty vlastností) týchto vecí
- *Funkcie a procesy zahŕňajúce tieto veci*
- *Obmedzenia na a pravidlá obsahujúce tieto veci*

8.2 Web so sémantikou

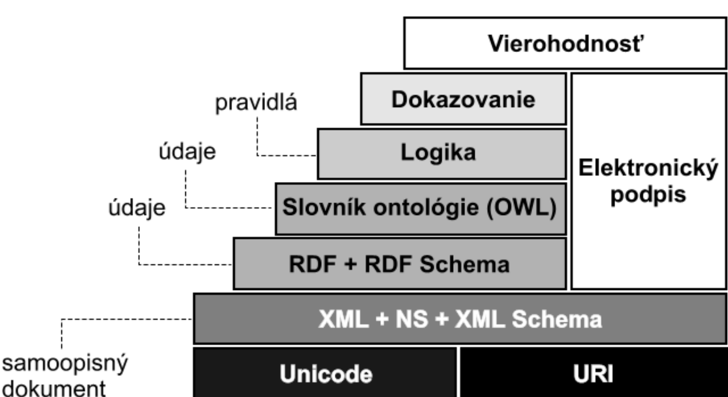
8.2.1 Vízia webu so sémantikou

Tim Berners-Lee, známy ako tvorca webu, má víziu jeho budúcnosti, ktorú nazýva „The Semantic Web“ (Berners-Lee, Hendler, Lassila, 2001), tzn. web so sémantikou.

Na webe so sémantikou budú informácie poskytované v strojovo-zrozumiteľnej podobe. V súčasnosti sa väčšina informácií na webe nachádza v prirodzenom jazyku a môžu im porozumieť len ľudia. A hoci na poli rozpoznávania a spracovania prirodzeného jazyka či extrakcie a integrácie informácií sme zaznamenali istý pokrok, ešte stále je tu mnoho otvorených problémov, ktoré je potrebné vyriešiť. Jedným zo spôsobov ako prinútiť stroje porozumieť publikovaným informáciám je formálne špecifikovať znalosti a na ich špecifikáciu použiť ontológie, ktoré sme opísali v predchádzajúcej kapitole.

Na obrázku 8-2 sú znázornené vrstvy webu so sémantikou tak, ako ich predstavil Berners-Lee v (Berners-Lee, Hendler, Lassila, 2001). Dve spodné vrstvy schémy, pozostávajúce z *Unicode+URI* a *XML+NS+XML Schema*, tvoria syntaktický základ pre jazyky webu so sémantikou. *Unicode* zabezpečuje základnú schému pre kódovanie znakov a je používané jazykom XML. Odporúčanie URI (Berners-Lee, 2005) poskytuje prostriedok na jedinečnú identifikáciu a adresáciu dokumentov na webe, no pomocou URI možno vo všeobecnosti identifikovať ľubovoľný zdroj (viac o identifikácii zdrojov možno nájsť v časti 8.2.2).

Vyšším vrstvám (Logika, Dokazovanie, Vierohodnosť, Elektronický podpis) sa v tejto práci venovať nebudeme, pretože žiadna z nich ešte neprešla procesom štandardizácie a ich bližšie opísanie by výrazne zväčšilo rozsah tejto práce.



Obrázok 8-2. Vrstvy webu so sémantikou (podľa Berners-Lee, Handler, Lassila, 2001).

8.2.2 Identifikácia zdrojov

Intuitívne pod pojmom zdroj rozumieme všetko, čomu môžeme priradiť identitu, pričom identita nemusí priamo implikovať prístupnosť zdroja či jeho umiestnenie. Fielding vo svojej dizertačnej práci (Fielding, 2000) považuje *zdroj* za kľúčovú abstrakciu informácie v architektúre moderného webu. Zdrojom môže byť ľubovoľná informácia, ktorú môžeme pomenovať. Ako príklad zdroja môžeme uviesť dokument, obrázok, kolekciu iných zdrojov, ľubovoľný nevirtuálny objekt (napr. osoba, kniha, monitor, stôl) a iné. Inými slovami, zdroj je *konceptuálne mapovanie do množiny entít.*

Identifikátor je objekt, ktorý slúži ako referencia na niečo, čo má identitu (napr. zdroj). Ak je zdroj prístupný, identifikátor možno použiť na „dereferencovanie“ zdroja. Príkladom identifikátora môže byť priezvisko, ISBN, cesta k súboru.

URI

Pridaním uniformity k identifikátorom, tj. zavedením prísnej syntaxe, ktorej sa všetky identifikátory musia podriadiť, dostávame URI (angl. *Uniform Resource Identifier*), ktoré je odporúčaním konzorcia W3. *Uniformita* ako jedna z vlastností URI je priamo dedená aj všetkými ďalšími spôsobmi adresácie a lokalizácie zdrojov (URN, URL, URC).

URL

URL (angl. *Uniform Resource Locator*) predstavuje pravdepodobne najznámejšiu a najrozšírenejšiu formu URI. URL je štandardizovaný spôsob lokalizácie zdrojov na Internete, zavedený spolu so službou WWW. Pomocou URL môžeme presne lokalizovať ktorýkoľvek dokument v hyperpriestore Internetu.

Vo všeobecnosti môžeme URL vyjadriť nasledujúcim spôsobom:

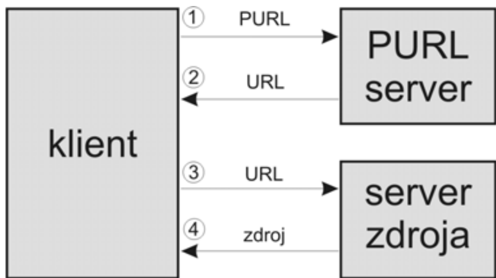
```
schema://<prihlasovacie-meno>:<heslo>@
         <adresa-pocitaca>:<port>/<cesta>;
         <parametre>?<dopyt>#<fragment>
```

URN

URN (angl. *Uniform Resource Name*) je určené pre zdroje, ktoré nemajú URL, ale je možné ich identifikovať v rámci konkrétneho menného priestoru, napr. ISBN, ISSN, DOI. V súčasnosti prebiehajú práce na štandardizácii URN.

PURL

Kým sa nezavedie a neakceptuje technológia URN, organizácia OCLC (Online Computer Library Center) vyvinula pomenovávaciu a rozhodovaciu službu pre všeobecné zdroje na Internete. Pomenovania, ktoré možno považovať za „perzistentné“ URL odkazy, môžeme použiť v dokumentoch, na webových stránkach a internetových katalógových systémoch.



Obrázok 8-3. Princíp získania URL pomocou PURL.

Z funkčnej stránky PURL je URL. Avšak, namiesto ukazovania priamo na umiestnenie požadovaného zdroja, PURL ukazuje na prechodnú rozhodovaciu službu. Táto služba tvorí akúsi medzistanicu medzi klientom a zdrojom. Rozhodovacia služba asociuje adresu PURL so skutočnou URL adresou zdroja a túto URL adresu vracia naspäť klientu. Klient môže následne dokončiť URL transakciu normálnym spôsobom. Inak povedané, ide o štandardné HTTP presmerovanie (pozri obrázok 8-3). Treba však

poznamenať, že princípom práce PURL servera nie je samotné presmerovanie, ale *nepriame smerovanie*.

PURL sa skladá z troch častí: protokol (PR), adresa rozhodovacej služby (ARS) a pomenovanie (PO). Nasledujúce príklady používajú rovnaký prístupový protokol (HTTP) na spojenie sa s PURL rozhodovacou službou (purl.oclc.org), aby sa dostali k rôznym menám:

```
http://purl.oclc.org/grlicky/home
http://purl.oclc.org/grlicky/research/publications
http://purl.oclc.org/OCLC/PURL/FAQ
----   ------------- -----------------------------
PR        ARS                   PO
```

URC

URC (angl. *Uniform Resource Characteristics;* URI Working Group, 1993) je iniciatívou organizácie IEEE. Pod skratkou URC, t.j. charakteristikou uniformných zdrojov, rozumieme balík URN, URL a ďalších identifikátorov zdroja. Cieľom URC je poskytnúť dostatočné množstvo informácií, ktoré by vytvorilo jednotný identifikačný blok pre konkrétny zdroj.

URC je zložená z dvojíc atribútov a ich hodnôt, ktorými možno opísať ľubovoľné URI (napr. autorstvo, vydavateľov, autorské práva atď.). Pretože URC doteraz ešte neprešlo štandardizáciou, príklad 8-1 ukazuje jeden z možných spôsobov reprezentácie charakteristiky zdroja.

```
<urc>
  <urn>urn:fiit.stuba.sk:people:grlicky:publications:
       msc-thesis-2003</urn>

<author>Grlicky, Vladimir</author>

<author type="email">grlicky@fiit.stuba.sk</author>
  <title>
    Information integration fromdisparate Web sources</title>
  <subject scheme="abstract">
    The main goal of this thesis was to design and evaluate the
    system...
  </subject>
  <instance>
    <coverage>Extended Abstract</coverage>
    <url>http://www.fiit.stuba.sk/~grlicky/contest/student2003/
         grlicky-abstract.doc</url>
</instance>

<instance>
<coverage>Fulltext</coverage>
    <url>http://www.fiit.stuba.sk/~grlicky/contest/student2003/
         grlicky-thesis.doc</url>
</instance>
</urc>
```

Príklad 8-1. Časť zdrojového kódu URC.

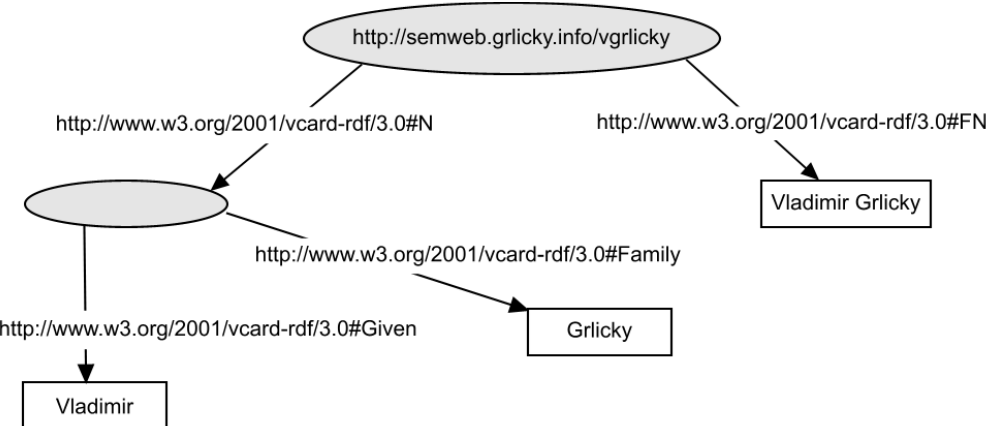
Diskusia

V tejto podkapitole sme relatívne rozsiahlu časť venovali PURL. Adaptácia URN naráža na množstvo problémov nie tak technického ako sociálneho charakteru (neexistujúce dohody na medzinárodnej úrovni ako klasifikovať objekty *reálneho* sveta). Výhodou používania PURL v súčasnosti je, že nejde o proprietárnu technológiu, tzn. ktokoľvek si môže vytvoriť a začať používať vlastný rozhodovaciu službu založenú na mechanizme nepriameho smerovania. Podobný mechanizmus môžeme nájsť aj vo webových službách vo forme protokolu UDDI (angl. *Universal Description, Discovery and Integration*). K adaptácii URN môže prispieť aj postupné presadzovanie sa technológií webu so sémantikou (ontológie v prostredí webu so sémantikou, prechod od slabo štruktúrovanej podoby webu k silnejšie štruktúrovanej), no pokiaľ nedôjde ku štandardizácii a masovému rozšíreniu URN na webe, alternatívne mechanizmy ako PURL budú stále existovať a vypĺňať priestor do budúcna vyhradený URN.

8.2.3 RDF

RDF (angl. *Resource Description Framework*) je spôsob opisu zdrojov, pomocou ktorého je možné publikovať informácie o rôznych zdrojoch v prostredí webu. Je určený najmä na poskytovanie metainformácií (napr. titul, autor a dátum modifikácie webovej stránky, informácie týkajúce sa licenčných podmienok a autorstva webových dokumentov, a pod.) o webových zdrojoch. RDF je ale použiteľné aj na poskytovanie informácií o objektoch, ktoré sú na webe nejakým spôsobom identifikovateľné, ale nie sú prostredníctvom webu priamo prístupné.

Dátový model RDF je založený na lingvisticky inšpirovanej konštrukcii *subjekt – predikát – objekt*, ktorá sa spolu označuje ako tvrdenie alebo *výrok* (angl. statement). Univerzálnym prvkom RDF je *zdroj* (angl. resource), na ktorého identifikáciu sa používa URI. Zdroj môže vystupovať v úlohe subjektu a aj v úlohe objektu. Predikáty odpovedajú sledovaným vlastnostiam subjektov, objekty zastupujú hodnoty týchto vlastností. Objektom môže byť okrem zdroja tiež *literál*, t.j. primitívna dátová hodnota.



Obrázok 8-4. Reprezentácia RDF modelu pomocou orientovaného grafu s pomenovanými hranami.

Dátový model RDF nešpecifikuje konkrétnu reprezentáciu – v prípade grafickej reprezentácie to môže byť *graf*, presnejšie orientovaný graf s pomenovanými hranami

(subjekty a objekty ako uzly, predikáty ako hrany), a ide o textovú reprezentáciu, možno použiť napr. RDF/XML (Beckett, 2004), NTriples (Beckett, 2001), N3 (Berners-Lee, 2001). Zápis tvrdení RDF modelu možno nájsť na príkladoch 8-2, 8-3 a 8-4. U textových reprezentácií hovoríme o tzv. *serializácii*, čo je presne špecifikovaný spôsob usporiadania jednotlivých prvkov výrokov RDF za seba. Základná syntax odporúčaná organizáciou W3C je založená na jazyku XML a oficiálne sa označuje RDF/XML. V jej prípade sa tvrdenia RDF serializujú do elementov a atribútov XML.

Dôležitou vlastnosťou modelu RDF je tzv. „reifikácia“. Pojem reifikácia pochádza z latinského „res“ – „vec“, tzn. „zvecnenie“ a používa sa na transformáciu tvrdenia daného jazyka na atomický objekt. Jej podstatou v prípade RDF je možnosť chápať celé tvrdenie ako zdroj, a prostredníctvom ďalších výrokov o ňom vypovedať. Reifikáciu možno uplatniť najmä pri usudzovaní o vierohodnosti jednotlivých tvrdení na základe dôvery v zdroj, ktorý ich „vyriekol“. Takto vytvorené „metavýroky“ sa dostávajú na rovnakú úroveň ako základné tvrdenia a nemusíme vytvárať špeciálne syntaktické konštrukcie. Na druhej strane však reifikácia dosť významne komplikuje formálne vlastnosti RDF modelu.

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#">
  <rdf:Description
    rdf:about="http://semweb.grlicky.info/vgrlicky">
    <vcard:N rdf:parseType="Resource">
      <vcard:Given>Vladimir</vcard:Given>
      <vcard:Family>Grlicky</vcard:Family>
    </vcard:N>
    <vcard:FN>Vladimir Grlicky</vcard:FN>
  </rdf:Description>
</rdf:RDF>
```

Príklad 8-2. Formálne vyjadrenie vzťahov trojice v notácii RDF/XML.

```
_:BN01 <http://www.w3.org/2001/vcard-rdf/3.0#Given> „Vladimir" .
<http://semweb.grlicky.info/vgrlicky>
<http://www.w3.org/2001/vcard-rdf/3.0#N> _:BN01 .
_:BN01 <http://www.w3.org/2001/vcard-rdf/3.0#Family> „Grlicky" .
<http://semweb.grlicky.info/vgrlicky>
<http://www.w3.org/2001/vcard-rdf/3.0#FN> „Vladimir Grlicky" .
```

Príklad 8-3. Formálne vyjadrenie vzťahov trojice v notácii NTriples.

```
@prefix : <#> .
<http://semweb.grlicky.info/vgrlicky>
<http://www.w3.org/2001/vcard-rdf/3.0#FN>
„Vladmir Grlicky" ;
<http://www.w3.org/2001/vcard-rdf/3.0#N>
[<http://www.w3.org/2001/vcard-rdf/3.0#Family> „Grlicky" ;
<http://www.w3.org/2001/vcard-rdf/3.0#Given> „Vladimir"] .
```

Príklad 8-4. Formálne vyjadrenie vzťahov trojice v notácii N3.

8.2.4 RDF Schéma

RDF poskytuje spôsob zápisu jednoduchých tvrdení o zdrojoch pomocou pomenovaných vlastností a ich hodnôt. Používatelia však potrebujú nástroj na definovanie nových slovníkov (angl. *vocabularies*), ktoré by bolo možné v daných tvrdeniach využiť. Dôvod používania slovníkov je, aby vytvorené tvrdenia v špecifickej oblasti mali rovnaký základ a boli jednoznačné. V slovníku budú špecifikované jednotlivé zdroje, triedy zdrojov a ich vlastnosti.

Samotné RDF neposkytuje prostriedky na definíciu takýchto tried a vlastností. Na tento účel vzniklo rozšírenie RDF s názvom *RDF Vocabulary Description Language*, alebo tiež *RDF Schema* (RDFS). Menný priestor schémy RDF je jednoznačne identifikovaný URI referenciou v tvare `http://www.w3.org/2000/01/rdf-schema#` a používa prefix `rdfs`.

RDFS neposkytuje aplikačno-špecifické slovníky, ale prostredníctvom nej je možné popísať príslušné triedy a vlastnosti a ich vzájomný vzťah. Podobne ako v RDF modeli je možné tvrdenia v RDFS zobraziť prostredníctvom grafu. Okrem opisu tried RDFS poskytuje nástroj na opis vlastností, ktoré dané triedy charakterizujú – na tieto účely sa využíva RDF trieda `rdf:Property` a RDFS vlastnosti `rdfs:domain`, `rdfs:range` a `rdfs:subPropertyOf`. Všetky vlastnosti v RDF sú zároveň inštanciami triedy `rdf:Property`.

RDFS ponúka tiež prostriedok, pomocou ktorého je možné určiť vzťah jednotlivých tried a vlastností (na to sa využívajú konštrukcie `rdfs:domain` a `rdfs:range`). Element `rdfs:range` indikuje, že hodnoty danej vlastnosti sú inštanciami špecifikovanej triedy, napr:

```
fiit:Osoba   rdf:type   rdfs:Class .
fiit:Student rdf:type   fiit:Osoba .
fiit:autor   rdf:type   rdf:Property .
fiit:autor   rdfs:range fiit:Osoba .
```

Element `rdfs:domain` indikuje, že daná vlastnosť prislúcha k špecifikovanej triede, napr:

```
fiit:Dokument rdf:type    rdfs:Class .
fiit:autor    rdf:type    rdf:Property .
fiit:autor    rdfs:domain fiit:Dokument .
```

8.2.5 OWL

Jazyk *OWL* (angl. *Web Ontology Language*; McGuinness a van Harmelen, 2004) je značkovací jazyk navrhnutý ako novonastupujúci štandard pre reprezentáciu ontológií na webe. Rozširuje slovník *RDF Schema* o ďalšie elementy súvisiace s triedami a ich vlastnosťami. Predchodcom jazyka OWL bol jazyk DAML+OIL. Vytváraním tried, definovaním vzťahov medzi nimi a opisom vlastností prvkov týchto tried je možné v jazyku OWL modelovať určitý systém znalostí o vybranej oblasti záujmu.

Ontológia v jazyku OWL je zapísaná v dokumente, na začiatku ktorého je uvedený zoznam slovníkov (priestorov mien). Tie jednoznačne definujú značky používané

v celom dokumente. Deklarácia priestorov mien je uzavretá v rámci značky `<rdf:RDF>`, napr:

```
<rdf:RDF
xmlns:rdf  = „http://www.w3.org/1999/02/22-rdf-syntax-ns#�โ
xmlns:rdfs = „http://www.w3.org/2000/01/rdf-schema#‟
xmlns:xsd  = „http://www.w3.org/2001/XMLSchema#‟
xmlns:owl  = „http://www.w3.org/2002/07/owl#‟
xmlns:fiit = „http://www.fiit.stuba.sk/ontology/0.1/‟>
```

Ukončovacia značka `</rdf:RDF>` bude uvedená na konci dokumentu za definíciou ontológie. Za deklaráciou priestoru mien sa nachádza hlavička ontológie:

```
<owl:Ontology rdf:about="">
<rdfs:comment>Ontológia FIIT STU v Bratislave</rdfs:comment>
  <owl:priorVersion
    rdf:resource="http://www.fiit.stuba.sk/ontology/0.1/"/>
  <owl:imports
    rdf:resource="http://www.fiit.stuba.sk/ontology/
                  student/0.5/"/>
  <rdfs:label>FIIT Ontology</rdfs:label>
</owl:Ontology>
```

Atribút `rdf:about` označuje zdroj, ktorý je v dokumente opísaný. V prípade, že je jeho hodnota prázdny reťazec, zdrojom je aktuálny dokument (tj. dokument opisujúci danú ontológiu). Atribút `owl:comment` poskytuje možnosť zadefinovať k danej ontológii poznámku. Prostredníctvom atribútu `owl:priorVersion` je poskytnutá informácia o ontológii, ktorá prechádzala aktuálne definovanej ontológii (informáciu o aktuálnej verzii je možné zadať prostredníctvom atribútu `owl:versionInfo`).[19] Atribút `owl:imports` vkladá do dokumentu inú, už existujúcu, ontológiu. Rozdiel medzi použitím tohto atribútu a deklaráciou menných priestorov je, že deklarácia menných priestorov iba jednoznačne definuje použiteľné značky, zatiaľ čo `owl:imports` vkladá do dokumentu už existujúce štruktúry a vzťahy.

Triedy

Základom každej ontológie sú *triedy*. Inštancie triedy sa v jazyku OWL nazývajú *indivíduá* (angl. individuals). V OWL existujú dve preddefinované triedy: `owl:Thing` a `owl:Nothing`. Element `owl:Thing` predstavuje množinu všetkých inštancií, ktoré sa v ontológii vyskytujú (každá trieda definovanej ontológie je podtriedou `owl:Thing`). Element `owl:Nothing` reprezentuje prázdnu množinu (tzn. je podtriedou každej triedy v danej ontológii). Novú triedu je možné zadefinovať nasledujúcim zápisom:

```
<owl:Class rdf:ID="Student"/>
```

Jediné, čo táto definícia hovorí je, že existuje trieda s daným identifikátorom, nehovorí nič o jej vlastnostiach a vzťahu k iným triedam. Triedy je možné definovať aj iným

[19] Viac informácií o mechanizme tvorby a záznamu verzií OWL ontológií je možné nájsť na adrese http://www.w3.org/TR/2004/REC-owl-guide-20040210/#OntologyVersioning.

spôsobom prostredníctvom identifikátora, napr. vymenovaním jej prvkov, zjednotením, prienikom, atď. Vymenovanie prvkov triedy sa realizuje prostredníctvom použitia elementu `owl:oneOf` a je možné ju zapísať napríklad takýmto spôsobom (stupne vysokoškolského štúdia):

```
<owl:Class>
<owl:oneOf rdf:parseType="Collection">
<owl:Thing rdf:about="#Bc"/>
<owl:Thing rdf:about="#Ing"/>
<owl:Thing rdf:about="#PhD"/>
</owl:oneOf>
</owl:Class>
```

Prostredníctvom konštrukcie `rdfs:subClassOf` je možné definovať podtriedy existujúcej triedy:

```
<owl:Class rdf:ID="Student">
<rdfs:subClassOf rdf:resource="Osoba"/>
</owl:Class>
```

Element `rdfs:label` poskytuje používateľsky zrozumiteľnejší názov špecifikovanej triedy. Prostredníctvom atribútu `xml:lang` je možné evidovať názov vo viacerých jazykoch. OWL poskytuje na opis vzájomných vzťahov medzi triedami ďalšie dva elementy: `owl:equivalentClass` a `owl:disjointWith`.[20]

Vlastnosti

Kým triedy a indivíduá vytvárajú akúsi štruktúru ontológie, vlastnosti vnášajú do celej koncepcie isté tvrdenia o triedach a ich prvkoch. Vlastnosti predstavujú v OWL binárne relácie buď medzi dvoma objektmi alebo medzi objektom a hodnotou údajového typu. Nové vlastnosti sa definujú podobne ako triedy, s tým rozdielom, že je použitý element `owl:ObjectProperty`:

```
<owl:ObjectProperty rdf:ID="maStudijnehoReferenta">
<rdfs:domain rdf:resource="#Student"/>
<rdfs:range rdf:resource="#StudijnyReferent"/>
</owl:ObjectProperty>
```

Element `owl:ObjectProperty` sa používa na definovanie tzv. objektových vlastností. V OWL existuje ešte iný druh vlastností – vlastnosti dátových typov (používa sa element `owl:DataTypeProperty`).

Definičný obor špecifikovanej vlastnosti a jej obor hodnôt je možné určiť pomocou konštrukcie `rdfs:domain`, resp. `rdfs:range` (ich použitie bolo uvedené vyššie v texte). Element `rdfs:subPropertyOf` definuje, že vlastnosť je podmnožinou inej vlastnosti. Presnejšie povedané, množina prvkov spĺňajúcich danú vlastnosť je podmnožinou množiny prvkov spĺňajúcich vlastnosť inú.

[20] Viac informácií o ich použití je možné nájsť na adrese http://www.w3c.org/TR/owl-ref/#ClassAxioms.

Ekvivalenciu a inverziu medzi dvoma vlastnosťami je možné vyjadriť pomocou elementov `owl:equivalentProperty`, resp. `owl:inverseOf`. V OWL je tiež možné nastaviť globálne obmedzenia kardinality vlastností, a tiež vzájomné logické charakteristiky medzi vlastnosťami ako sú tranzitívnosť a symetria.

8.3 Prezentácia informácií a znalostí na webe so sémantikou

S rýchlym rozvojom a zmenou obsahu, ktorý sa na webe nachádza, sa menia aj jazyky, ktoré sú prostriedkom na zobrazenie tohto obsahu. Publikovanie informácií pre globálne distribuovanie si vyžaduje zrozumiteľný jazyk, ktorému budú rozumieť potenciálne všetky počítače. Pre tento účel sa používajú značkovacie jazyky, ktorých spoločným predkom je jazyk SGML (angl. *Standard Generalized Markup Language*). V prostredí webu sa najčastejšie používa jazyk HTML (angl. *HyperText Markup Language*). Autorom poskytuje prostriedky pre:

- publikovanie „online" dokumentov obsahujúcich hlavičky, text, tabuľky, zoznamy a pod.,
- prepojenie informácií prostredníctvom hypertextových odkazov, a tým vytváranie informačných sietí,
- návrh formulárov pre spoluprácu so vzdialenými službami, pre použitie pri vyhľadávaní informácií, spracovaní informácií a pod.,
- rozšírenie textových dokumentov o ďalšie médiá vložením zvuku, videa a iných druhov médií.

Ak sa pri analýze zdrojov znalostí detailnejšie zameriame na zdroje na Internete, primárnym sprostredkovateľom znalostí sú *webové sídla*. Hoci na Internete možno nájsť niekoľko definícií webového sídla, vo všeobecnosti budeme pod týmto pojmom rozumieť kolekciu webových stránok takto zoskupených na základe ich spolupatričnosti ku konkrétnej téme, problémovej oblasti, spoločnosti, organizácii či osobe. Alternatívne tiež môžeme povedať, že primárnym zdrojom znalostí sú konkrétne webové stránky zoskupené do webových sídiel podľa kritérií uvedených v definícii vyššie.

Z hľadiska prístupu webových sídiel k poskytovaniu informácií tretím stranám, ktoré chcú tieto informácie ďalej spracovať, môžeme sídla rozdeliť na pasívne a aktívne.

„Pasívne" webové sídla. V súčasnosti prevažnú väčšinu sídiel vo webe možno pokladať za pasívne, nakoľko okrem poskytovania obsahu zakódovaného v primárnom formáte, ktorým je vo webe jazyk HTML, neposkytujú inú formu sprostredkovania informácií, a teda neumožňujú potenciálnym integrátorom informácií získať tieto informácie iným spôsobom ako vytvorením obaľovačov webových stránok.

„Aktívne" webové sídla. Ide o sídla, ktoré buď (1) vytvárajú alternatívne kanály pre prístup k dátam primárne poskytovaných vo forme HTML, alebo (2) obohacujú poskytovaný obsah pridaním tzv. „sémantických" značiek, príp. metadát, ktoré sú vnorené priamo do webových stránok. Do prvej kategórie možno zaradiť webové služby (Haas, 2002), kanály založené na jednej zo špecifikácií RSS alebo Atom (Nottingham a Sayre, 2004), intenzívne využívané hlavne prevádzkovateľmi

internetových podob tlačených periodík či vo webových verziách verejne prístupných osobných denníkov, tzv. blogoch. Do kategórie (2) patria technológie ako SHOE alebo GRDDL. SHOE (angl. *Simple HTML Ontology Extensions*; Luke a Heflin, 2002) je jazyk pre reprezentáciu znalostí založený na jazyku HTML (presnejšie ide o rozšírenie HTML), ktorý už nie je v súčasnosti vyvíjaný, no ako jeden z prvých predstavil myšlienku zahrnutia znalostí (nielen metadát) priamo do webovej stránky. Jeho nevýhodou však bolo príliš úzke prepojenie opísaných znalostí na konkrétny zdroj, t.j. webovú stránku. GRDDL (angl. *Gleaning Resource Descriptions from Dialects of Languages*), bližšie opísaný v časti 8.4.1, je mechanizmus pre zakódovanie RDF výrokov (trojíc v tvare subjekt – predikát – objekt) do HTML a XML dokumentov a ich extrakciu pomocou XSLT transformácií a podobných technológií. Je vyvíjaný pod hlavičkou konzorcia W3C s dôrazom na zautomatizovanie extrakcie dopredu označených údajov priamo v (X)HTML dokumente a ich následné spracovanie už na úrovni RDF modelu vo formáte RDF/XML.

V tejto časti sa sústredíme na techniky pre špecifikáciu a generovanie prezentácií, ponúkame vybrané spôsoby prezentovania informácií a znalostí v priestore webu so sémantikou. Tieto vychádzajú z jazykov pre reprezentáciu ontológií, ktoré uvádzame v kapitolách 8.2.3, 8.2.4, resp. 8.2.5.

8.4 Špecifikácia a generovanie prezentácie pomocou XSLT

Technológia XSLT (angl. *Extensible Stylesheet Language Transformations*; Holzner, 2002) umožňuje priamo manipuláciu s obsahom dokumentov XML, umožňuje transformáciu XML dokumentov na dokumenty iného formátu (napr. HTML, RTF, XSL-FO a iné), umožňuje extrahovať údaje z XML dokumentov obdobným spôsobom ako je to v prípade použitia SQL z databázy.

XSLT je v skutočnosti súčasťou širšej špecifikácie – špecifikácie jazyka XSL (angl. *Extensible Stylesheet Language*). Podstatou XSL je definovanie formátu (vzhľadu) dokumentov. Druhou časťou špecifikácie XSL sú objekty XSL-FO (angl. *XSL Formatting Objects*), ktoré sa používajú k určeniu toho, ako majú byť údaje z XML dokumentov reprezentované.

XSLT je štandardizované riešenie na oddelenie prezentačnej úrovne od aplikačnej. Tvorba prezentačných výstupov (dokumentov) môže byť oddelená od aplikačného kódu. Údaje, ktoré sa prostredníctvom XSLT transformujú do prezentačných výstupov, sú vo formáte XML. XSLT sa používa na transformáciu XML dokumentov pomocou XSL štýlov na iné, napr. (X)HTML dokumenty, XSL-FO sa zvyčajne používa k transformácii XML dokumentov na PDF (angl. *Portable Data Format*) dokumenty.

Technológie založené na XML a XSLT umožňujú vizualizáciu XML dokumentov a zobrazenie informácií pre koncových používateľov. XSL štruktúry môžu byť definované podľa ontológie, ktorá pokryje všetky možné štruktúry výstupných dát. Tento spôsob poskytuje plnú kontrolu nad štruktúrou všetkých možných výstupných dát produkovaných systémom. XSLT sú teda šablóny, podľa ktorých sa transformujú dáta z XML dokumentov zväčša do prezentačnej formy. Šablóna XSLT ako aj transformované vstupné údaje XML musia byť po syntaktickej stránke platné XML dokumenty.

Príkladom generovania prezentácie pomocou XSLT je projekt *Hera*, ktorý predstavili Frasincar a kol. (Frasincar et al., 2001; Vdovjak et al., 2003).

Obrázok 8-5. Architektúra webového informačného systému Hera.

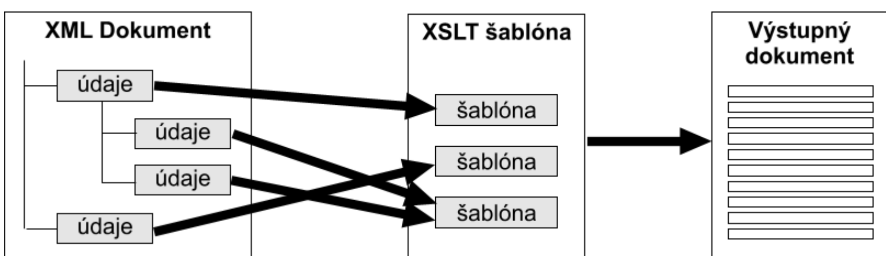
Pri generovaní prezentácie sa získané informácie a znalosti transformujú do hypermediálnej prezentácie, ktorá je vhodná pre danú platformu pre aktuálne preferencie používateľa. Navigačný aspekt hypermediálnej prezentácie je opísaný v aplikačnom modeli. Aplikačný model tvoria entity, vlastnosti entít a vzťahy medzi nimi (agregácia a navigácia), ktoré spolu definujú ontológiu. Každá entita má definovaných viacero vlastností. Aplikačný model sa používa na opis logickej úrovne hypermediálneho aspektu aplikácie. Potom generovanie prezentácie (Frasincar et al., 2001) uvedenej na obrázku 8-5 pozostáva z týchto krokov:

- *Získavanie údajov a ich vyčistenie* – údaje sa získavajú z úložiska údajov a sú prispôsobované špecifickým formátom inštancií. Pre každú aplikáciu existuje aplikačný model s opisom vzťahov medzi segmentmi nad uvažovaným doménovým modelom.
- *Logická transformácia – generovanie inštancie aplikačného modelu* – inštancia aplikačného modelu je vytvorená so získanými údajmi. Prispôsobenie (založené napr. na profile používateľa) sa tiež realizuje v tomto kroku.
- *Transformácia prezentácie* – generuje prezentáciu zo súhrnných údajov o segmente (inštancie aplikačného modelu vytvorené z Logickej transformácie, napr. vo formáte RDF/XML) na kód interpretovateľný prehliadačom (napr. HTML).

Metódy transformácií XML na (X)HTML

Pri transformácii XML dokumentov pomocou XSLT sú najčastejšie používanými metódy PUSH a PULL, resp. ich kombinácia (používaná najmä v prípade reťazenia transformácií). XSLT kód sa v obidvoch prípadoch nijako zásadne nelíši. Metódy sa odlišujú najmä v tom, či je riadiacim prvkom výstupu XSLT šablóna alebo XML dokument. Ak je celý XML dokument „pretlačený" cez pripravené šablóny, ide o PUSH metódu. V tomto prípade sú všetky elementy nachádzajúce sa vo vstupnom XML dokumente formátované do prezentačného dokumentu (obrázok 8-6).

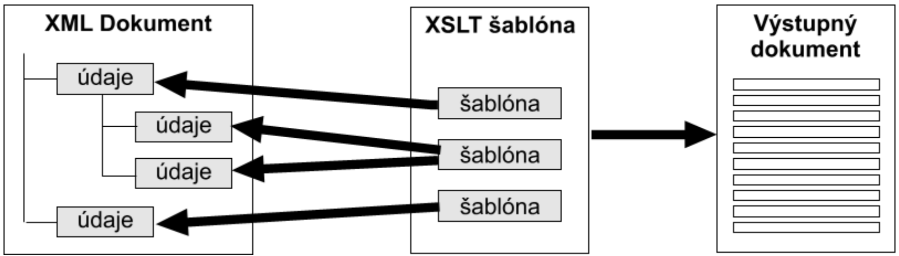
Pri PUSH metóde sa zvyčajne obsah výstupu určuje už pri tvorbe XML dokumentu. Ak chceme napríklad zobraziť údaje obsiahnuté v databáze s určitým obmedzením, zadefinujeme obmedzujúce podmienky už pri SQL dopyte do databázy, a tak dostaneme potrebný XML dokument, ktorý len „pretlačíme" cez pripravenú šablónu XSLT.



Obrázok 8-6. PUSH metóda používaná pri XSLT transformáciách.

V prípade PULL metódy šablóna XSLT „vyťahuje" z referenčného XML dokumentu len tie údaje, ktoré potrebuje, takže XML slúži len ako dátová štruktúra. Schematicky je metóda PULL zobrazená na obrázku 8-7.

Pri PULL metóde by sme celú databázu jednoducho pretransformovali do jedného dokumentu, z ktorého by XSLT šablóna „vyťahovala" potrebné údaje podľa svojich požiadaviek, teda rozhodovanie o obsahu výstupu by riadila práve XSLT šablóna.



Obrázok 8-7. PULL metóda používaná pri XSLT transformáciách.

V príklade 8-5 je uvedená časť šablóny pre transformáciu dát z XML dát do (X)HTML. Šablóna je použiteľná hlavne pre PUSH metódu, pri ktorej transformuje všetky časti XML dokumentu s elementmi `WfInstance`. V rámci tejto šablóny sa volá ďalšia šablóna s názvom `creator/Employee`, ktorá musí byť v XSLT dokumente uvedená, aby mohla byť použitá. V tejto šablóne je tiež uvedený cyklus `for-each`, ktorý je použitý na prechod všetkých elementov nachádzajúcich sa pod elementom `domain`.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<table border="0" width="100%" cellpadding="3">
<tr>
<td class="resource">

<xsl:apply-templates select="WfInstance"/>
</td>
</tr>
</table>
</xsl:template>
...

<xsl:template match="WfInstance">
Process Instance<br/>
<strong>Title:</strong> <xsl:value-of select="title"/>
<br/><br/>
<strong>Domain Information:</strong><br/>
<xsl:for-each select="domain/*">
<xsl:value-of select="name()"/>:
<xsl:value-of select="title"/><br/>
</xsl:for-each>
<br/>
<strong>Creator:</strong>

<xsl:apply-templates select="creator/Employee"/>
</xsl:template>
...
```

Príklad 8-5. Časť XSLT šablóny pre transformáciu dát z XML do (X)HTML.

```
<WfInstance ID="wfi">

<domain>
<Shape ID="shape365">
<title>Pedonale</title>
</Shape>
</domain>

<title>Repair of TL on Main Square</title>

<domain>
<District ID="Area_District557">
<title>Val Polcevera</title>
</District>
</domain>

<creator>
<Employee ID="emplGGuliano">
<lastname>giuliano</lastname>
<firstname>G.</firstname>
<title>Designer</title>
<email>giuliano@portatilelino.priv.softeco.it</email>
```

```
<wfRole>
<WfRole ID="roleDesign">
<title>Designer</title>
</WfRole>
</wfRole>
</Employee>
</creator>

<domain>
<Applicant ID="other_applicant452">
<title>School Director</title>
</Applicant>
</domain>

</WfInstance>
```

Príklad 8-6. XML dokument pripravený na transformáciu s XSLT šablónou.

Dôležitou súčasťou XSLT je jazyk *XPath*, ktorého odporúčanie je zvlášť opísané organizáciou W3C (Clark a DeRose, 1999). XPath je však možné použiť aj samostatne, resp. môže byť použitý aj v iných podobných mechanizmoch spracovávajúcich XML dokumenty.

```
<xsl:value-of select="sum(//book/@price) div count(//book)"/>
```

Príklad 8-7. Dopytovanie sa pomocou XPath v rámci XSLT šablóny.

Jazyk XPath slúži na vyhľadávanie a dopytovanie sa v štruktúre spracovávaného XML dokumentu, pričom vôbec nenarúša jeho štruktúru. Výsledok vyhľadávania tohto dopytu môže byť následne transformovaný príslušnou šablónou XSLT.

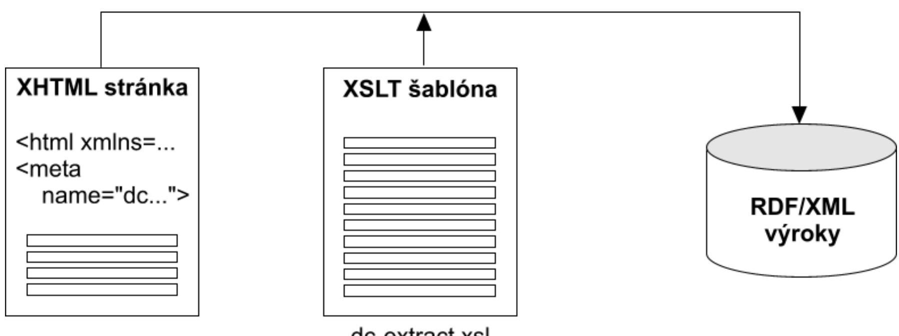
Procesory XSLT sa používajú k tomu, aby bolo možné použiť jazyk XSLT k transformácii dokumentov napr. v RDF/XML na (X)HTML (Holzner, 2002). Jazyk XSLT je možné využívať:

- *s XSLT procesormi.* Najpoužívanejšie sú XSLT procesory implementované v jazyku Java, ale dostupné a rozšírené sú aj XSLT procesory implementované v iných jazykoch ako Python, C++, Perl a pod.
- *v rámci klientskeho programu.* Príkladom klientskeho programu je webový prehliadač podporujúci XSLT transformácie.
- *na serveri.* Program na strane servera môže používať XSLT šablónu a XSLT procesory k automatickej transformácii dokumentu, a následne k odosielaniu výsledkov na klientsku stranu. XSLT procesory na serverovej strane sú dostupné aj pre rôzne „middleware“ jazyky ako Java, PHP, Perl, ASP.

8.4.1 Prezentácia informácií a ich extrakcia pomocou GRDDL

Ďalšou technikou prezentácie informácií opísaných pomocou ontológií je mechanizmus GRDDL (angl. *Gleaning Resource Descriptions from Dialects of Languages*; Hazael-Massieux a Connoly, 2004). GRDDL je primárne určený na

automatizované „vyťahovanie“ opisov zdrojov z dokumentov zapísaných pomocou jazykov rodiny XML (najmä z (X)HTML dokumentov).



Obrázok 8-8. Princíp automatizovaného zberu informácií pomocou GRDDL.

Princíp extrakcie informácií je schematicky naznačený na obrázku 8-8. Vstupom do procesu extrakcie je vo všeobecnosti XML dokument, najčastejšie je to však (X)HTML stránka obsahujúca metadáta (napr. Dublin Core [DC], Creative Commons [CC], Friend-Of-A-Friend [FOAF], GeoURL), ktoré sa nachádzajú buď v hlavičke dokumentu alebo priamo v jeho tele. XHTML dokument pripravený pre extrakciu metadát, resp. informácií ktoré sú v dokumente určené metadátami konkrétnej ontológie (a patriaci pod „aktívne“ webové sídlo), je zobrazený v príklade 8-8.

```
<html xmlns="http://www.w3.org/1999/xhtml">

<head profile="http://www.w3.org/...">

<meta http-equiv="Content-Type"
content="text/html; charset=windows-1250" />

<title>Domovska stranka Jozefa Mrkvicku
[príklad RDF v XHTML]</title>

<link rel="schema.DC" href="http://purl.org/dc" />
<link rel="meta" type="application/rdf+xml" title="FOAF"
href="http://www.w3.org/2000/06/webdata/xslt?
xslfile=http://www.firma.sk/~mrkvicka/foaf.xsl;
xmlfile=http://www.firma.sk/~mrkvicka/index.html" />
<!-- Dublin Core -->

<meta name="DC.Title" xml:lang="en" lang="en"
content="Domovska stranka Jozefa Mrkvicku" />
<meta name="DC.Creator" content="Jozef Mrkvicka" />
<meta name="DC.Description"
xml:lang="en" lang="en" content="..." />
<meta name="DC.Publisher"
content="Firma.sk - http://www.firma.sk" />

<link rel="transformation" href="FOAF.xsl" />
<link rel="transformation" href="CC.xsl" />
<link rel="transformation" href="GeoURL.xsl" />
</head>
```

```
<body>
<p>Ahoj, volam sa <span class="foaf-name">Jozef Mrkvicka</span>
   a pracujem vo firme
   <a href="http://www.firma.sk" rel="foaf-work"
   >Firma, s.r.o.</a> Mozete ma kontaktovat mailom na
   <a ref="mailto:jozef.mrkvicka@firma.sk"
   >jozef.mrkvicka@firma.sk</a> alebo ziskat viac informacii
   na mojej <a href="http://www.firma.sk/~mrkvicka/"
rel="foaf-home">domovskej stranke</a>.
</p>
</body>

</html>
```

Príklad 8-8. Časť zdrojového kódu XHTML dokumentu obsahujúceho metadáta a odkazy na XSLT šablóny pre ich extrakciu pomocou GRDDL.

Hoci je GRDDL vyvíjaný pod hlavičkou konzorcia W3, v súčasnosti ho možno považovať len za experiment alebo snahu ukázať, že automatizácia získavania informácií z existujúcich webových dokumentov je možná. Jej predpokladom je však obohatenie existujúcich (X)HTML stránok o „sémantické“ značky, t.j. značky, ktoré budú označeným dátam pridávať význam, čím ich transformujú na informácie.

8.5 Záver

Zariadenia, ktoré ukladajú na web údaje v strojovo-zrozumiteľnej podobe, majú v súčasnosti pre mnohé komunity vysokú prioritu. Web môže dosiahnuť svoj plný potenciál len vtedy, ak sa stane miestom, kde sa môžu spracovávať a zdieľať údaje, a to jednak automatickými nástrojmi ako aj ľuďmi. Kvôli škálovateľnosti webu budú musieť mať budúce programy schopnosť zdieľať a spracovávať údaje dokonca aj vtedy, ak by boli programy vyvinuté úplne nezávisle. Web so sémantikou je víziou, je to myšlienka mať údaje na webe definované a prepojené takým spôsobom, že sa budú dať použiť nielen na zobrazenie, ale aj na automatizáciu, integráciu a ich opätovné použitie rôznymi aplikáciami.

V tejto časti sa budeme snažiť identifikovať problémy, s ktorými sa stretáva web so sémantikou, a následne poukážeme na problémy týkajúce sa špeciálne prezentácie informácií a znalostí v priestore „nového“ webu so sémantikou.

8.5.1 Otvorené problémy

Pôvodné predstavy o rovnako rýchlej penetrácii webu so sémantikou medzi „obyčajných“ používateľov tak, ako tomu bolo v prípade súčasného webu, sa s veľkou pravdepodobnosťou nenaplnia. Prečo je tomu tak, je uvedené v nasledujúcich odsekoch, v ktorých sa snažíme pozerať na web so sémantikou cez kľúčové vlastnosti súčasného webu.

Za dôvody úspechu „súčasného“ webu môžeme považovať (spracované podľa Hausteina a Pleumanna, 2002a):

- *Jednoduchosť.* Jazyk HTML je ľahko pochopiteľný a rýchlo zapísateľný aj bez pomoci sofistikovaného textového editora. Aj začiatočníci môžu bez väčšej

námahy navrhnúť jednoduché webové stránky, uložiť ich do hierarchickej adresárovej štruktúry na strane webového servera, a tým zverejniť požadované údaje na webe.

- *Okamžitá spätná odozva.* Potom, ako je stránka vytvorená, výsledok je možné ihneď zobraziť a vizuálne skontrolovať v ľubovoľnom webovom prehliadači. Týmto má tvorca stránky zabezpečenú okamžitú spätnú odozvu na svoju prácu.
- *Doplňujúce výhody.* Hoci ich pôvodným zámerom je prezentovanie informácií iným ľuďom, HTML stránky môžu byť využité aj ako prostriedok diskusií alebo dokumentácie pre ľudí participujúcich na projekte alebo pre ich osobné použitie. Táto vlastnosť webu sa stala ďalšou pridanou hodnotou, ktorú používatelia získali jeho používaním, a tým sa táto web stal pre nich ešte zaujímavejšou službou, ktorú Internet poskytuje.
- *Nízka kritická „masa".* Web, ako snaha o zdieľanie informácií v sieťovom prostredí, potreboval minimálny (ale zároveň dostatočne veľký) počet účastníkov, ktorí by zvýšili záujem u „obyčajných" ľudí a presvedčili ich, aby ho začali používať. Keďže web bol prvý systém svojho druhu, tzn. dovtedy neexistoval žiadny podobný systém, ktorý by mu mohol konkurovať, tento kritický počet nadšencov potrebný na „prerazenie" bol relatívne malý.

Ak vyššie uvedené body porovnáme s webom so sémantikou v jeho súčasnej podobe, zistíme, že väčšina z nich nie je splnená:

- Jednoduchosť je splnená len čiastočne. Zmes konštrukcií jazykov RDF, DAML+OIL (a v súčasnosti aj OWL) je plne pochopiteľná len ľuďmi, ktorí majú dostatočný prehľad v umelej inteligencii, matematickej logike alebo príbuzných odvetviach. Začiatočníci budú schopní používať len základné koncepty RDF, a preto môžu mať problémy vidieť skutočné výhody webu so sémantikou.
- Okamžitá spätná odozva nie je splnená. Nanešťastie, v súčasnosti neexistuje klientsky softvér pre web so sémantikou, ktorý by urobil dojem na používateľov vhodnou prezentáciou bázy RDF faktov. Možno však argumentovať, že takýto typ softvéru nie je potrebný, pretože klientmi webu so sémantikou sú hlavne programy, nie ľudia.
- Nie sú tu žiadne prídavné výhody, aspoň nie evidentné pre „obyčajných" koncových používateľov. Kým HTML stránky, ktoré sú čitateľné ľuďmi a primárne navrhované pre iných ľudí, mohli byť použité aj pre osobné účely, toto neplatí pre RDF fakty, ktorých dôvodom použitia je hlavne ich strojové spracovanie.
- Kritický počet „nadšencov" je značne vyšší. V súčasnosti tu máme existujúci (a možno povedať, že fungujúci) systém, pôvodný web, a väčšina ľudí dnes pri vyhľadávaní informácií, ktoré tento web obsahuje, používa internetové vyhľadávače (napr. Google[21]) a pri špecifikovaní dopytu využíva metódu „hrubej sily" týmito vyhľadávačmi implementovanú. Preto je oveľa ťažšie presvedčiť ľudí, aby začali používať iný systém, aj keď je tento systém rozšírením už existujúceho.

[21] Google, `http://www.google.com`

Aj na základe vyššie uvedených poznatkov sme otvorené problémy webu so sémantikou rozdelili takto:

- evolučný verzus revolučný prístup k rozširovaniu webu so sémantikou,
- pomalá štandardizácia technológií webu so sémantikou,
- chýbajúce „úspešné“ aplikácie založené na technológiách webu so sémantikou,
- náročná adaptácia existujúcich štandardov.

8.5.2 Evolučný verzus revolučný prístup k rozširovaniu webu so sémantikou

Ide o všeobecný problém súvisiaci, resp. zastrešujúci všetky nasledujúce problémy webu so sémantikou.

Etzioni et al. (2002) vyjadruje názor, že v prvotných fázach rozširovania webu so sémantikou by sa nemala vyžadovať sémantická interoperabilita, tzn. vytváranie globálne záväzných ontológií. Skôr by malo dochádzať k vytváraniu lokálnych „webov so sémantikou“ univerzít či firiem založených na proprietárnom sémantickom značkovaní, až dodatočne sa príslušné lokálne ontológie budú na seba mapovať, a tým prepájať do väčšieho celku.

Haustein a Pleumann (2002b) tvrdia, že informácie by mali byť udržiavané a prezentované vo formáte HTML. Jednoduché pridávanie RDF metadát by však viedlo k duplicite informácií a riziku ich nekonzistencie. Preto ako čiastočné riešenie navrhujú udržiavanie informačného obsahu v XML s pomocou jednoduchých ontológií, pričom jeho prezentáciu zabezpečujú XSL šablóny za intenzívnej podpory softvérových nástrojov. Dôležité však je, že výstupom celého procesu sú HTML stánky a RDF metadáta vznikajú automaticky ako vedľajší produkt, ktorý používateľov nestojí žiadne úsilie.

8.5.3 Pomalá štandardizácia technológií webu so sémantikou

Štandardizované sú len nižšie vrstvy webu so sémantikou (tzn. XML, URI, Unicode, RDF(S), a v súčasnosti už aj OWL), ktoré tvoria tzv. povinný technologický základ webu so sémantikou, skutočné výhody oproti pôvodnému webu však sľubujú až vrstvy nachádzajúce sa nad ontológiami – logika s odvodzovacími pravidlami, dokazovanie pravdivosti, dôveryhodnosť tvrdení – tieto sú však len v ranných štádiách vývoja.

8.5.4 Chýbajúce „úspešné“ aplikácie založené na technológiách webu so sémantikou

Vo vývoji Internetu, či webu ako jednej z jeho služieb, možno nájsť niekoľko aplikácií, ktoré sa stali míľnikmi v jeho histórii, a ktoré výrazne ovplyvnili pohľad ľudí na túto celosvetovú sieť (v anglicky hovoriacich krajinách skrátene nazývané „killer apps“). Sem môžeme zaradiť webový prehliadač Mosaic, ktorého ešte úspešnejším nástupcom sa stal Netscape Communicator, Altavista – webový vyhľadávač vyvinutý firmou Digital, či v súčasnosti populárny vyhľadávač Google a mnohé ďalšie. Preto aj na webe so sémantikou budeme musieť pravdepodobne počkať na príchod aplikácie či aplikácií, ktoré dokážu svojou jednoduchosťou či kvalitou osloviť vyššie spomínané kritické množstvo ľudí, a tým ich nepriamo prinútiť využívať nové vlastnosti webu obohateného o sémantiku.

Literatúra

BECKETT, D. (2001) N-Triples : W3C RDF Core WG Internal Working Draft. Dostupné na: <http://www.w3.org/2001/sw/RDFCore/ntriples/>.

BECKETT, D. (2004) RDF/XML Syntax Specification (Revised) : W3C Recommendation 10 February 2004. Dostupné na: <http://www.w3.org/TR/rdf-syntax-grammar/>.

BERNERS-LEE, T. – HENDLER, J. – LASSILA, O. (2001) The Semantic Web. Scientific American.

BERNERS-LEE, T. (2001) An RDF Language for the Semantic Web : Notation 3. Dostupné na: <http://www.w3.org/DesignIssues/Notation3.html>.

BERNERS-LEE, T. ET AL. (2005) URI Generic Syntax (RFC 3986). Dostupné na: <http://www.ietf.org/rfc/rfc3986.txt>.

BORST, P. – AKKERMANS, H. – TOP, J. (1997) Engineering Ontologies. International Journal of Human-Computer Studies (Special Issue on Using Ontologies in KBS Development), (46):365–406.

BRAY, T. ET AL. (2004) Extensible Markup Language (XML) 1.0 (3. vydanie) : W3C Recommendation 04 February 2004. Dostupné na: <http://www.w3.org/TR/REC-xml/>.

BRICKLEY, D. (2004) RDF Vocabulary Description Language 1.0: RDF Schema : W3C Recommendation 10 February 2004. Dostupné na: <http://www.w3.org/TR/rdf-schema/>.

CLARK, J. – DEROSE, S. (1999) XML Path Language (XPath) Version 1.0 : W3C Recommendation 16 November 1999. Dostupné na: <http://www.w3.org/TR/xpath>.

DACONTA, M.C. – OBRST, L.J. – SMITH, K.T. (2003) The Semantic Web : A Guide to the Future of XML, Web Services, and Knowledge Management. Wiley : Indianapolis, Indiana, ISBN 0-471-43257-1.

DZBOR, M. – DOMINGUE, J.B. – MOTTA, E. (2003) Magpie – Towards a Semantic Web Browser. In: *Proc. of the 2nd Intl. Semantic Web Conf.*

ETZIONI, O. ET AL. (2002) An Evolutionary Approach to the Semantic Web. In: *Collected Posters, First International Semantic Web Conference (ISWC 2002).*

FENSEL, D. (2000) Ontologies : A Silver Bullet for Knowledge Management and Electronic Commerce. Berlin : Springer.

FIELDING, R.T. (2000) Architectural Styles and the Design of Networkbased Software Architectures. Dizertačná práca, University of California, Irvine. Dostupné na: <http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>.

FRASINCAR, F. – HOUBEN, G.J. (2001) XML-based Web Presentation Generation. Dostupné na: <http://wwwis.win.tue.nl/~hera/papers/webnet2001/wn.pdf>.

GRUBER, T.R. (1993) Toward Principles for the Design of Ontologies Used for Knowledge Sharing. Technická správa KSL-93-04, Stanford, CA : Knowledge

Systems Laboratory, Stanford University. Dostupné na: <ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-93-04.ps.gz>.

HAAS, H. (2002) Web Services Activity. Dostupné na: <http://www.w3.org/2002/ws/>.

HAUSTEIN, S. – PLEUMANN, J. (2002a) Easing Participation in the Semantic Web. In: *International Workshop on the Semantic Web 2002*. Dostupné na: <http://www-ai.cs.uni-dortmund.de/haustein_pleumann_2002a.pdf?self=$Document_1110175650638&part=data>.

HAUSTEIN, S. – PLEUMANN, J. (2002b) Is Participation in the Semantic Web Too Difficult? In: *The Semantic Web – First International Semantic Web Conference 2002*, Horrocks, I. – Hendler, J., editors, LNCS, s. 448. Heidelberg: Springer. Dostupné na: <http://www-ai.cs.uni-dortmund.de/haustein_pleumann_2002b. pdf? self = $Document_1110175650652&part = data>.

HAZAEL-MASSIEUX, D. – CONNOLLY.D (2004) Gleaning Resource Descriptions from Dialects of Languages (GRDDL) : W3C Coordination Group Note 13 April 2004. Dostupné na: <http://www.w3.org/TR/grddl/>.

HOLZNER, S. (2002) XSLT – Příručka internetového vývojáre. Computer Press. ISBN 80-7226-600-4.

LUKE, S. – HEFLIN, J. (2000) SHOE 1.01 Specification. Dostupné na: <http: //www.cs.umd.edu/projects/plus/SHOE/spec.html/>.

MCGUINNESS, D.L. – VAN HARMELEN, F. (2004) OWL Web Ontology Language Overview : W3C Recommendation 10 February 2004. Dostupné na: <http://www.w3.org/TR/owl-features/>.

MCGUINNESS, D.L. (2002) Ontologies Come of Age. MIT Press.

MILLER, E. – SWICK, R. – BRICKLEY, D. (2004) Resource Description Framework (RDF). Dostupné na: <http://www.w3.org/RDF/>.

NECHES, R. ET AL. (1991) Enabling Technology for Knowledge Sharing. AI Magazine, (12):36–56.

NIRENBURG, S. – RASKIN, V. (2004) Ontological Semantics. MIT Press, 440 s. ISBN 0-262-14086-1.

NOTTINGHAM, M. – SAYRE, R. (2004) The Atom Syndication Format : draft-ietf-atompub-format-03. IETF Working Group. Dostupné na: <http://www.ietf.org/internet-drafts/draft-ietf-atompub-format-03.txt>.

NOY, N.F. – MCGUINNESS, D.L. (2001) Ontology Development 101 : A Guide to Creating Your First Ontology. Technická správa KSL-01-05, Stanford, CA : Stanford Knowledge Systems Laboratory, Stanford University. Dostupné na: <http://www.ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness.pdf>.

QUAN, D. – KARGER, D.R. (2004) How to Make a Semantic Web Browser. In: *13th World Wide Web Conference*.

SKLENÁK, V. ET AL. (2001) Data, informace, znalosti a Internet. Praha : C. H. Beck, 1. vydanie, 507 s. ISBN 80-7179-409-0.

STUDER, R. – BENJAMINS, V.R. – FENSEL, D. (1998) Knowledge Engineering : Principles and methods. Data and Knowledge Engineering, 25(1–2):161–197.

SVÁTEK, V. (2002) Ontologie a WWW. DATAKON 2002, s. 1-35. Dostupné na: <http://nb.vse.cz/~svatel/onto-www.pdf>.

THE UNICODE CONSORTIUM (2003) The Unicode Standard, Version 4.0.1. Reading, MA : Addison-Wesley. ISBN 0-321-18578-1. Dostupné na: <http://www.unicode.org/versions/Unicode4.0.1/>.

URI WORKING GROUP (1993) URN to URC Resolution Scenario. Dostupné na: <http://www.mitra.biz/uri/draft-ietf-uri-urn2urc-00.txt>.

USHOLD, M. – GRUNINGER, M. (1996) Ontologies : Principles, Methods and Applications. The Knowledge Engineering Review, 11(2):93–136.

VDOVJAK, R. ET AL. (2003) Engineering Semantic Web Information Systems in HERA. Journal of Web Engineering, 0(0).

9 MODELOVANIE ADAPTÍVNYCH WEBOVÝCH SYSTÉMOV

9.1 Úvod

V súčasnej dobe význam a využitie webu neustále stúpa. Možnosti jeho využitia sú veľmi široké, pričom stále sa objavujú nové. Internet sa používa ako zdroj informácií, ako prostriedok pracovnej činnosti, ako nástroj pre výučbu a pod.

Statická štruktúra informácií na webe, ktorej úlohou je poskytovanie informácií je už dávno prekonaná. Vzniká čoraz viac webových softvérových systémov, ktoré sú zložitejšie ako kedykoľvek predtým. Z pohľadu použitia týchto systémov sa preto čoraz viac dostáva do popredia potreba obohatenia informačného priestoru heterogénnych zdrojov, nad ktorým spomínané systémy pracujú, o prvky prispôsobovania sa používateľovi a/alebo prostrediu, v ktorom používateľ pracuje. Cieľom je prezentovať používateľovi personalizované informácie, teda podľa možnosti len také, ktoré sú pre používateľa relevantné a takým spôsobom, ktorý danému používateľovi čo najviac vyhovuje.

Keďže ide mnohokrát o zložité softvérové systémy, problém nastáva aj z pohľadu tvorby takýchto systémov. Obsah, ktorý prezentujú je bohatý, heterogénny a navyše značne premenlivý. Potreba prispôsobovania sa ukazuje ako kľúčová, preto je potrebné na návrh týchto systémov klásť veľký dôraz. Dobre navrhnúť štruktúru informačného priestoru, charakteristiky, ktoré budú zdrojom prispôsobovania, ale hlavne samotný spôsob prispôsobovania tak, aby prispôsobovanie nebolo kontraproduktívne, nie je jednoduché. Potrebné sú vhodné metódy modelovania podobných systémov. V tejto práci sa budeme zaoberať problematikou modelovania adaptívnych hypermediálnych systémov, ktoré sú podmnožinou webových softvérových systémov.

V kapitole 9.2 uvedieme podrobnú charakteristiku adaptívnych hypermediálnych systémov. Uvedieme ciele, ktoré si spomínané systémy kladú. Rozoberieme tiež prínosy použitia týchto systémov, rovnako ako riziká spojené s ich používaním. Opíšeme princíp prispôsobovania a tiež úrovne, na ktorých má zmysel realizovať prispôsobovanie. Súčasťou kapitoly bude aj prehľad existujúcich metód a techník prispôsobovania.

V kapitole 9.3 najprv charakterizujeme najznámejšie referenčné modely pre modelovanie adaptívnych hypermediálnych systémov. Ďalej uvedieme spôsoby vytvárania uvedených modelov v existujúcich hypermediálnych systémoch.

Na záver zhrnieme otvorené problémy spojené s tvorbou adaptívnych hypermediálnych systémov a načrtneme možnosti ďalšieho výskumu.

9.2 Charakteristika adaptívnych hypermediálnych systémov

Adaptívne hypermediálne systémy (AHS) sú hypermediálne systémy, ktoré sledujú charakteristiky používateľa, uchovávajú ich v modeli používateľa a tento využívajú ako zdroj pre prispôsobenie rozhrania medzi systémom a používateľom (Brusilovsky, 1996). Spájajú teda výhody klasických adaptívnych systémov založených na modeli používateľa s výhodami hypermediálnych systémov. Možno ich označiť aj ako hypermediálne systémy rozšírené o možnosti prispôsobovania na základe analýzy správania sa používateľa.

Z uvedeného vyplýva, že každý adaptívny hypermediálny systém musí spĺňať nasledovné kritériá:

- musí to byť hypermediálny/hypertextový systém, umožňujúci navigáciu v hyperpriestore definovanom nad určitou aplikačnou doménou,
- musí obsahovať model používateľa, v ktorom uchováva charakteristiky používateľa a
- musí obsahovať mechanizmus prispôsobovania, ktorý je schopný dynamicky meniť prezentované informácie na základe stavu modelu používateľa.

Úlohou modelu používateľa je udržiavať sledované charakteristiky používateľa, ktoré neskôr vplývajú na spôsob prispôsobovania. Charakteristikami, ktoré možno sledovať, sú napr. znalosti o používateľovi, jeho preferencie, záujmy, úlohy alebo ciele. Sledované charakteristiky sú zvyčajne podskupinou toho, čo sa dá na používateľovi a jeho správaní počas používania systému sledovať. Obmedzujú sa iba na tie, ktoré sa pri prispôsobovaní využijú.

Keď hovoríme o prispôsobovaní, treba vymedziť rozdiel medzi adaptabilnými a adaptívnymi systémami:

- Oba typy systémov spracujú nad modelom používateľa.
- *Adaptabilný systém* dovoľuje používateľovi konfigurovať systém zmenou parametrov, na základe ktorých systém zmení svoje správanie. O tom, kedy sa zmení model používateľa však rozhoduje samotný používateľ. Je to teda systém, ktorý je prispôsobiteľný.
- *Adaptívny systém* je systém, ktorý sa prispôsobuje autonómne. To znamená, že sám automaticky sleduje správanie používateľa, zaznamenáva si zistené skutočnosti v modeli používateľa a na základe tohto modelu automaticky prispôsobuje svoje správanie. Medzi sledované skutočnosti patria napr. akcie používateľa, jeho odpovede na otázky, ktoré mu kladie systém, prípadne ďalšie informácie, ktoré o sebe používateľ poskytne.

9.2.1 Ciele AHS

Jedným z hlavných cieľov AH systémov je zvýšenie dostupnosti informácií pre používateľa. Keďže dostupné informácie sú roztrúsené v hyperpriestore, vzniká riziko straty orientácie používateľa a nájdenie hľadanej informácie nemusí byť vždy jednoduché. Úlohou AH systému je preto pomoc pri sprostredkovaní hľadanej informácie. Tento cieľ sa dosahuje prispôsobovaním obsahu a funkcionality systému v závislosti od úrovne znalostí o používateľovi.

Aby bol systém schopný prispôsobovať svoje rozhranie jednotlivým používateľom, je potrebné, aby o nich získal pokiaľ možno čo najviac informácií počas ich práce so systémom. Získavanie informácií však musí byť do maximálnej možnej miery autonómne, inak by používatelia prestali mať záujem používať daný systém, pretože by ich neustále získavanie spätnej väzby mohlo otravovať.

Výhodou systému, ktorý prispôsobuje rozhranie potrebám svojich používateľov je dostupnosť pre širšiu skupinu používateľov, pretože sa systém tvári, akoby bol vyvíjaný práve pre toho ktorého používateľa.

Cieľom je tiež zjednodušenie práce s AH systémom (napríklad zníženie počtu potrebných kliknutí na dosiahnutie hľadanej informácie), čo môže motivovať používateľov používať takýto systém.

9.2.2 Prínosy AHS

AH systémy zlepšujú interakciu medzi používateľom a systémom. Zvyšujú relevantnosť prezentovaného informačného obsahu, zároveň znižujú čas potrebný na jeho získanie a celkovo tak zefektívňujú prácu so systémom.

Je známych viacero oblastí, kde má použitie AH systémov význam:

- *Výučbové systémy*. Ide o oblasť, v ktorej je použitie AH systémov najrozšírenejšie. Ich použitie môže výrazným spôsobom podporiť proces učenia sa vďaka schopnosti prispôsobiť študijné tempo rozdielnym potrebám študentov. Menej zdatnému študentovi môže napr. ponúknuť obšírnejší výklad preberanej látky, kým študenta, ktorý danú problematiku už zvládol, nemusí zaťažovať zbytočnými podrobnosťami.
- *Adaptívne vyhľadávače*. Aj v tejto oblasti má ich použitie veľký význam. Bežné vyhľadávače vôbec neberú do úvahy používateľa, ktorý dotaz na vyhľadávanie zadáva. Každému používateľovi poskytnú rovnaké výsledky, no ako je zrejmé, keď viacero používateľov zadá ten istý dotaz, nemusí to nutne znamenať, že hľadajú tú istú vec. Ukazuje sa preto potreba rozlišovania kontextu, v ktorom je dotaz zadávaný. Riešením by mohlo byť napr. sledovanie, na aké oblasti sa daný používateľ zameriava, pričom tento fakt by bol zohľadnený pri poskytovaní výsledkov vyhľadávania.
- *Online informačné systémy*. Môže ísť napr. o veľké informačné portály, ktoré slúžia ako báza znalostí v nejakej korporácii. Zamestnanci, ktorí tento systém používajú sa zrejme špecializujú iba na určitú oblasť a je preto vhodné, aby pre nich systém dokázal filtrovať len relevantné informácie.

- *Online help systémy*. Ich význam je v zohľadnení situácie, v ktorej používateľ žiada o pomoc. Na základe vyhodnotenia kontextu sú takéto systémy schopné poskytnúť relevantnejšiu pomoc.
- *Osobné asistenty*. Ide o systémy bežiace na pozadí, ktoré počas toho ako sa používateľ pohybuje v rámci hyperpriestoru, sú schopné zvýrazňovať časti prezentovaných informácií, ktoré na základe skúmania záujmov používateľa (alebo prípadne prednastavenia určitého profilu) označia ako relevantné.

AH systém umožňuje prispôsobovanie prezentovaného obsahu, čím zvyšuje adresnosť prezentovaných informácií. Zároveň prispieva k sprehľadneniu navigácie a umožňuje tiež vhodne kombinovať navigáciu riadenú používateľom, ako aj vedenie používateľa systémom. Celkovo možno povedať, že AH systém pomáha riešiť problémy, akými sú (Bieliková, 2003):

- *zablúdenie v informačnom priestore/dezorientácia*, ide o pomerne bežný jav vzhľadom na skutočnosť, že web predstavuje otvorený priestor navzájom rôzne prepojených uzlov,
- *kognitívne preťaženie*, používateľ je zaplavený, preťažený a zmätený rôznymi možnosťami navigácie,
- *nesúvislý tok informácií*, pri neriadenej navigácii sa používateľ často nevyhne návšteve rôznych, navzájom nie veľmi súvisiacich častí informačného priestoru, dôsledkom čoho je napr. strata koncentrácie.

Najčastejšie sa AH systémy používajú v uzavretom informačnom priestore. Dôsledkom je dobre známa štruktúra informácií, vďaka čomu sa dajú podrobne špecifikovať znalosti o prispôsobovaní. Výskum prebieha aj v oblasti otvoreného informačného priestoru, kde je situácia komplikovanejšia. V tejto oblasti nie je z princípu možné nadefinovať úplný graf reprezentujúci štruktúru informácii, čo sťažuje definíciu znalostí o prispôsobovaní. Sú preto potrebné iné metódy prispôsobovania.

9.2.3 Riziká spojené s používaním AHS

Systém tým, že redukuje a upravuje možnosti navigácie, do určitej miery preberá kontrolu nad používateľom, čo je v rozpore s pôvodnou koncepciou hypertextu, kde má používateľ úplnú kontrolu nad tým, ktoré odkazy nasleduje a ku ktorým stránkam pristupuje.

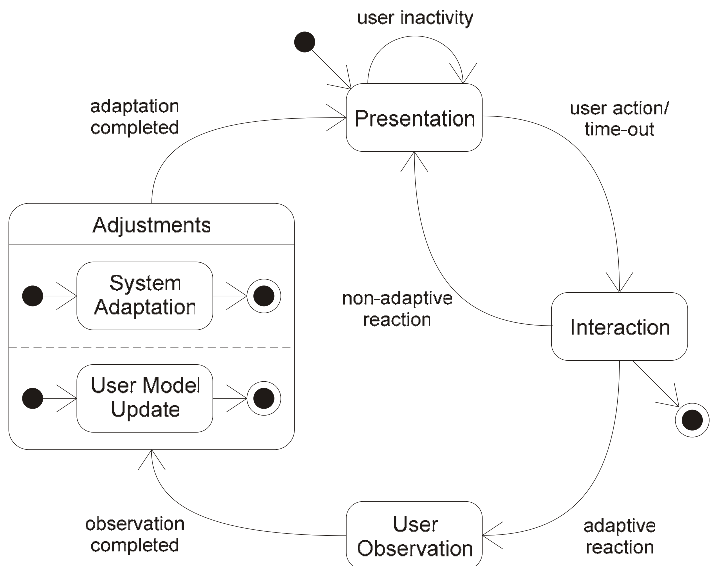
Používatelia považujú adaptívne systémy často ako obmedzujúce, nepredvídateľné a mätúce. Adaptívne systémy preto musíme navrhovať tak, aby použité metódy a techniky prispôsobovania boli použité správnym spôsobom, aby ich použitie bolo prínosom, aby nemiatli používateľa a aby ich používateľ dokázal využiť.

Jedným z problémov je nestálosť prezentácie, teda fakt, že v dvoch rôznych momentoch môže byť tá istá informácia prezentovaná rôzne. To sťažuje orientáciu používateľa. Najväčším rizikom je zmena spôsobu prezentácie informácií, ako aj prispôsobovanie možností navigácie. Jedno z riešení tohto problému poskytuje systém SmexWeb (Albrecht, 2000), v ktorom sa zaznamenáva história zmien modelu používateľa. Spôsob prezentácie informácií sa tak počas jedného sedenia, kedy používateľ pracuje so systémom, nemení.

Ďalším problémom je súkromie používateľa. Aby sa systém dokázal používateľovi prispôsobiť, je potrebné sledovať a zaznamenávať jeho prácu so systémom. Tento fakt vyvoláva debatu o tom, či je to etické. Používatelia sa obávajú možnosti zneužitia takto získaných informácií. Odborníci tvrdia, že postupom času si používatelia zvyknú a nebudú to brať ako zásah do ich súkromia, ale skôr ako niečo, z čoho budú profitovať, pretože systém im poskytne rýchlejší prístup k relevantným informáciám.

9.2.4 Životný cyklus prispôsobovania

Počas toho, ako používateľ pracuje so systémom sa tento nachádza vo viacerých stavoch v závislosti od toho, o aký druh interakcie ide. Tieto stavy opísali Jungmann a Paradies (1997), neskôr boli mierne modifikované (Koch, 2000). Uvedené sú na obrázku 9-1.



Obrázok 9-1. Životný cyklus prispôsobovania.

V prípade interakcie používateľa, ktorá je spätá s adaptívnymi vlastnosťami systému, ide postupne o prechod medzi štyrmi stavmi:

- *Prezentácia* (*Presentation*). V tomto stave sa systém štandardne nachádza potom, ako používateľovi poskytne zvolenú časť informačného priestoru. Pri ich prezentácii berie do úvahy znalosti, ktoré o danom používateľovi má. Tie využije pri rozhodovaní, v akej forme, prípadne ktorú variantu obsahu bude prezentovať a tiež pri generovaní možností ďalšej navigácie v systéme.
- *Interakcia* (*Interaction*). Systém sa rozhoduje ako reagovať na akciu používateľa.
- *Sledovanie používateľa* (*User Observation*). V tomto stave systém analyzuje informácie získané o používateľovi na základe vykonanej interakcie.
- *Prispôsobovanie* (*Adjustments*). Tento stav v sebe zahŕňa dva paralelné podstavy: aktualizáciu modelu používateľa a prispôsobenie používateľského rozhrania.

1. *Aktualizácia modelu používateľa* (*User Model Update*). Na základe získaných znalostí o používateľovi systém aktualizuje model používateľa, kde zaznamená zistené skutočnosti.
2. *Prispôsobenie používateľského rozhrania* (*System Adaptation*). Systém určí, ktoré informácie a akým spôsobom bude prezentovať používateľovi a taktiež ponúkne ďalšie možnosti navigácie v systéme. To všetko na základe aktuálneho stavu modelu používateľa.

Pokiaľ ide o interakciu používateľa, ktorá nie je spätá s adaptívnymi vlastnosťami systému, ide o prechod medzi dvoma základnými stavmi: prezentácia a interakcia. Pri prezentácii zvolenej časti informačného priestoru sa prispôsobovanie nerealizuje.

9.2.5 Metódy a techniky prispôsobovania

Základnými aspektmi hypermediálnych systémov je obsah, forma a štruktúra informácií. Z hľadiska možností prispôsobovania je potrebné tieto aspekty posudzovať osobitne. Význam jednotlivých aspektov je nasledovný:

- *Obsah*. Prezentovaný obsah pozostáva z informačných fragmentov, ktoré tvoria množinu dostupných informácií. Informačnému fragmentu zodpovedá koncept, ktorý reprezentuje príslušnú časť aplikačnej domény. Výsledkom združenia elementárnych konceptov do zložených konceptov a vyjadrením vzťahov medzi nimi je graf vyjadrujúci štruktúru informácií v aplikačnej doméne. Zároveň je určený vzťah medzi fragmentmi informačného priestoru.
- *Štruktúra*. Určuje spôsob organizácie obsahu vo forme stránok hypertextu. Mapuje koncepty na stránky hyperpriestoru a určuje možnosti navigácie medzi stránkami. Spomínané mapovanie aplikačnej domény na hyperpriestor väčšinou nie je jedna k jednej.
- *Prezentácia*. Určuje formu, akou je obsah prezentovaný. Ide napr. o spôsob rozmiestnenia prvkov na stránke, veľkosť fontu, farby a pod.

Prispôsobovanie je možné na všetkých troch úrovniach. Systém môže vybrať iba vhodné informačné fragmenty a tie zobraziť na stránke, môže ich vhodne preusporiadať, zvoliť vhodnú formu prezentácie rovnako ako môže prispôsobiť odkazy na stránke, môže niektoré z nich nezobraziť, prípadne vygenerovať iné. Brusilovsky rozlišuje dve úrovne prispôsobovania (Brusilovsky, 1996):

- *adaptívna prezentácia*, ktorá predstavuje prispôsobovanie formy a obsahu,
- *adaptívna navigácia*, ktorá predstavuje prispôsobovanie možností navigácie.

Patterno a Mancini oddeľujú prispôsobovanie obsahu od prispôsobovania prezentácie a hovoria o troch úrovniach prispôsobovania (Patterno, 1999):

- *prispôsobovanie obsahu*,
- *prispôsobovanie navigácie*,
- *prispôsobovanie prezentácie*.

Aj keď výskum v oblasti adaptívnych hypermédií trvá relatívne krátko (približne 10 rokov), za ten čas sa vykryštalizovali metódy a techniky, pomocou ktorých je možné dosiahnuť prispôsobovanie. Brusilovsky (1996) definuje adaptívnu metódu ako abstrakciu adaptívnych techník. Adaptívna technika je pritom definovaná spôsobom

reprezentácie v modeli používateľa a mechanizmom, ktorý zabezpečí samotné prispôsobovanie. Adaptívna metóda môže byť realizovaná pomocou jednej alebo viacerých techník, rovnako ako technika môže byť použitá na realizáciu viacerých metód.

V nasledujúcich kapitolách uvedieme metódy a techniky pre prispôsobovanie obsahu, navigácie a prezentácie. Pôvodne ich navrhol Brusilovsky v roku 1996 (Brusilovsky, 1996) a neskôr ich doplnil (Brusilovsky, 2001). Rozlišuje metódy a techniky pre adaptívnu prezentáciu a adaptívnu navigáciu. My ich uvádzame v súlade s členením uvedeným vyššie.

Prispôsobovanie obsahu

Cieľom metód pre prispôsobovanie obsahu je podpora používateľov s rôznym stupňom znalosti aplikačnej domény, na základe ktorého je možné vhodne kombinovať, ktorá časť informačného priestoru sa zobrazí, prípadne ktorý jej variant, s akou úrovňou podrobností a pod.

Identifikované boli nasledujúce metódy prispôsobovania obsahu:

- *Podmienený obsah*. Najčastejšie používaná metóda. Ide o zobrazovanie len relevantných častí obsahu (ostatné sa skrývajú) na základe úrovne znalostí používateľa, jeho cieľov, záujmov a preferencií.
- *Alternatívny obsah*. Ide o zobrazovanie jednej z alternatív obsahu, pričom to, ktorá alternatíva je zobrazená je opäť určené na základe modelu používateľa.
- *Usporadúvanie obsahu*. Jednotlivé časti obsahu sú zoradené podľa ich relevantnosti, pričom najskôr sa zobrazujú najrelevantnejšie časti a ako posledné najmenej relevantné časti obsahu.

Uvedené metódy je možné realizovať pomocou nasledujúcich techník prispôsobovania obsahu:

- *Vkladanie/odstraňovanie fragmentov*. Informačné fragmenty, ktoré nie sú "vhodné" sa nezobrazujú.
- *Rozťahovací text (strečtext)*. Ide o možnosť zobrazenia textu v skrátenej a aj v plnej (roztiahnutej) podobe, pričom formu zobrazenia jednotlivých fragmentov určuje adaptívny systém.
- *Alternatívy stránok*. Existuje viacero alternatív tej istej stránky, pričom sa zobrazí ten najvhodnejší variant.
- *Alternatívy fragmentov*. Oproti technike alternatív stránok ide o úroveň nižšie, a síce že aj samotné stránky sú poskladané z fragmentov, ktoré môžu mať viacero variant. Môžu sa líšiť po obsahovej stránke, alebo spôsobom prezentácie. Prezentuje sa "vhodný" variant.
- *Usporadúvanie fragmentov*. Prezentované fragmenty sa usporiadajú podľa "vhodnosti".

Metóda podmieneného obsahu je realizovateľná pomocou ľubovoľnej z techník: vkladanie/odstraňovanie fragmentov, rozťahovací text, resp. alternatívy stránok. Metóda alternatívneho obsahu je realizovateľná jednou z techník: alternatívy stránok,

resp. alternatívy fragmentov. Metóda usporadúvanie obsahu je realizovateľná jedine technikou usporadúvania fragmentov.

Prispôsobovanie navigácie

Cieľom metód pre prispôsobovanie navigácie je sprehľadnenie možností navigácie a usmernenie používateľa aby nenasledoval odkazy nesúvisiace s jeho cieľmi. Pomáhajú tiež používateľovi orientovať sa v zložitom hyperpriestore.

Identifikované boli nasledujúce metódy prispôsobovania navigácie:

- *Globálne vedenie*. Ide o metódu, ktorej cieľom je pomoc pri hľadaní najkratšej cesty v hyperpriestore vedúcej k hľadanej informácii.
- *Lokálne vedenie*. Cieľom tejto metódy je naviesť používateľa na najvhodnejšiu stránku logicky nasledujúcu po aktuálne zobrazenej stránke.
- *Podpora globálnej orientácie*. Ide o metódu, ktorej cieľom je zvýšiť orientáciu používateľa v rámci celého hyperpriestoru pomocou informácie o tom, v ktorej časti hyperspriestoru sa práve nachádza.
- *Podpora lokálnej orientácie*. Táto metóda informuje používateľa o možnostiach aktuálne dostupnej navigácie, čím mu pomáha pri jeho rozhodovaní, ktorý z odkazov nasleduje.
- *Personalizované pohľady*. Ide o metódu, ktorej cieľom je prispôsobovať navigáciu v rámci hyperpriestoru s ohľadom na jeho záujmy a ciele.

Uvedené metódy je možné realizovať pomocou nasledujúcich techník prispôsobovania navigácie:

- *Priame vedenie*. Systém vedie používateľa v informačnom priestore, t.j. vyberá najvhodnejšie koncepty a fragmenty im priradené. Často sa realizuje pomocou tlačidla "Ďalej".
- *Usporadúvanie odkazov*. Odkazy na ďalšie koncepty sa usporiadajú podľa vhodnosti, možno ich kategorizovať do niekoľkých skupín a usporiadať v rámci jednotlivých skupín.
- *Skrývanie odkazov*. Odkazy, ktoré vedú k neodporúčaným informáciám sa skryjú. Skrývanie možno realizovať niekoľkými formami: odkaz sa nezobrazí (zobrazí sa iba text odkazu), odkaz sa blokuje (spôsob prezentácie závisí od kombinácie nezobrazenia odkazu a anotácie odkazu) alebo odkaz sa zruší z prezentácie.
- *Anotácia odkazov*. Systém označuje "vhodné" odkazy. Napr. systém AHA! (De Bra, 1998) rozlišuje tri druhy odkazov *odporúčané/neutrálne/neodporúčané* (*good/neutral/bad*), ktoré aj farebne rozlišuje. Aj štandardné prehliadače rozlišujú odkazy, ktoré smerujú na navštívené a nenavštívené stránky.
- *Pasívna navigácia*. Ide o implicitné odkazy, ktoré systém využíva po rozpoznaní určitého vzoru správania sa používateľa; používateľ je neaktívny a navigáciu zabezpečuje systém (možno využiť aj pre tlačidlá "Späť" a "Ďalej").
- *Generovanie odkazov*. Systém dynamicky generuje nové odkazy (napr. objavuje súvislosti medzi jednotlivými konceptmi).

- *Adaptácia máp*. Systém na základe modelu používateľa dynamicky vytvára mapu domény (grafická prezentácia navigácie).

Okrem toho existujú aj tzv. sociálne navigačné techniky, ktoré pracujú na princípe zoskupovania používateľov do skupín s podobnými záujmami:

- *Kolaboratívne a obsahovo orientované filtrovanie*. Táto technika sleduje, ku ktorým z dostupných dokumentov používateľ pristupuje a zisťuje ich explicitné ohodnotenie používateľom. Na základe podobnosti dokumentov potom dokáže určiť relevantnosť informácií pre používateľov z rovnakej skupinu používateľov. Vyžaduje si však informáciu o predchádzajúcich použitiach danej množiny dokumentov väčším počtom používateľov. Na vytváranie skupín používateľov s podobnými záujmami používa rozličné algoritmy.
- *Sociálna adaptívna navigácia*. Táto technika využíva princípy kolaboratívneho a obsahovo orientovaného filtrovania. Vychádza z predchádzajúcich interakcií používateľov so systémom. Na ohodnocovanie dokumentov používa napr. kľúčové slová. Sledovaním veľkej skupiny používateľov sa vytvárajú kolektívne znalosti a prispôsobovanie navigácie potom prebieha na základe týchto kolektívnych znalostí. Presnosť tejto techniky pri odporúčaní odkazov má ďaleko od presnosti klasických techník, no výhodou je, že je použiteľná v otvorenom informačnom priestore.

Metóda globálneho vedenia je realizovateľná ľubovoľnou z techník: priame vedenie, usporadúvanie odkazov, generovanie odkazov, resp. pasívna navigácia. Metóda lokálneho vedenia je realizovateľná ľubovoľnou z uvedených techník. Metóda podpory globálnej orientácie je realizovateľná technikou anotácie odkazov, resp. skrývaním odkazov. Rovnako aj metóda lokálnej orientácie, kde ešte navyše pribúda možnosť použitia techniky usporadúvania odkazov. Na generovanie personalizovaných pohľadov sa využíva väčšina z uvedených techník.

Prispôsobovanie prezentácie

Pri metódach prispôsobovania prezentácie ide na rozdiel od modifikovania obsahu (ako je to v prípade metód prispôsobovania obsahu) o prispôsobovanie formy, ktorá je zvolená pre prezentáciu zvoleného obsahu. Často sa však uskutočňujú práve spolu s metódami prispôsobovania obsahu.

Medzi metódy prispôsobovania prezentácie patrí:

- *Viacjazyčnosť*. Cieľom je prispôsobovanie jazyka, v ktorom sú informácie prezentované.
- *Alternatívy prezentácie*. Zobrazovaný obsah môže byť zobrazený rôznymi štýlmi, prvky na stránke môžu byť rozmiestnené rôzne.

Uvedené metódy je možné realizovať pomocou techník pre prispôsobovania obsahu (okrem techniky rozťahovací text), ku ktorým pribúda ešte technika:

- *Alternatívy štýlov*. Existujú viaceré varianty štýlov zobrazovania obsahu (napr. farba pozadia, typ a veľkosť písma, médium a pod.).

9.2.6 Problémy spojené s tvorbou AHS

Tvorba AH systémov nie je triviálny proces a prináša so sebou mnoho problémov. Je to zložitý a časovo náročný proces, ktorého najzložitejšou časťou je vytvorenie adaptívnej funkcionality systému.

Oproti návrhu hyperpriestoru v klasických hypermediálnych systémoch, ktorý spočíva vo viac-menej priamočiarej definícii uzlov a hrán hyperpriestoru, je v prípade adaptívnych hypermédií situácia zložitejšia. Pri návrhu hyperpriestoru je potrebné brať do úvahy ciele, ktoré chceme prispôsobovaním dosiahnuť, a podľa toho vhodne navrhnúť koncepty a vzťahy medzi nimi. Navrhnutý graf konceptov je navyše potrebné vhodne namapovať na informačný priestor. Táto úloha je z hľadiska prispôsobovania kľúčová.

Kritickou časťou návrhu AH systému je aj návrh modelu používateľa. Spočíva v definovaní charakteristík, ktoré bude systém sledovať a patrične využívať. Čím prepracovanejší je návrh modelu používateľa, tým väčšie možnosti sa otvárajú pri jeho zužitkovaní počas samotného prispôsobovania. Rovnako výber metód a techník, ktoré zabezpečia prispôsobovanie, musí byť premyslený. Prispôsobovanie treba navrhnúť tak, aby bolo v prvom rade prínosom pre používateľa. Tu zasahujú aj iné vedné disciplíny, napr. *Human Computer Interface*.

Uvedené problémy môže aspoň čiastočne zmierniť existencia dobre definovaných metód modelovania. V nasledujúcej kapitole preto uvedieme prehľad referenčných modelov AH systémov a rozoberieme, akým spôsobom sa tieto modely navrhujú v existujúcich systémoch.

Menej závažným problémom, ktorý je spôsobený skôr tým, že výskum v tejto oblasti trvá relatívne krátko, je nedostatok podporných prostriedkov pre návrh a implementáciu AH systémov. V súčasnosti podobné systémy tvoria väčšinou výskumníci, ktorí sú patrične technicky zdatní. Jednou z priorít je preto aj vytvorenie vhodných modelovacích nástrojov, ktoré by vývoj AH systémov podstatne zjednodušili, čo by viedlo k väčšiemu rozšíreniu týchto systémov. Ich vytvorenie by nemalo predstavovať vážnejší problém.

9.3 Modely adaptívnych hypermediálnych systémov

9.3.1 Referenčné modely

Hypermediálne systémy zvyčajne zdieľajú spoločné prvky architektúry. Abstrakciu architektúry možno opísať referenčným modelom. Na ich špecifikáciu sa používajú rôzne techniky opisu. Jednou z možností je formálna špecifikácia pomocou špecifikačného jazyka s definovanou syntaxou a sémantikou jednotlivých prvkov jazyka, ako aj odvodzovacími pravidlami. Ďalšou z možností je opis pomocou semiformálnych techník, napríklad pomocou diagramov alebo iných štruktúrovaných dát. Najintuitívnejším spôsobom špecifikácie je neformálny opis, najčastejšie pomocou prirodzeného jazyka. Formálne techniky vyjadrujú špecifikáciu zvyčajne najpresnejšie.

V praxi sa mnohokrát využíva kombinácia viacerých techník špecifikácie. Existuje niekoľko referenčných modelov hypermediálnych (hypertextových) systémov. Dajú sa rozdeliť do dvoch skupín, podľa spôsobu špecifikácie modelov:

- neformálne, resp. semiformálne modely – do tejto skupiny patria napríklad referenčné modely:
  - *Hypertext Abstract Machine* (HAM; Campbell, 1988), vôbec prvý referenčný model navrhnutý a neformálne opísaný v roku 1988,
  - *Amsterdam Hypermedia Model* (AHM; Hardmann, 1994),
  - *Adaptive Hypermedia Application Model* (AHAM; De Bra, 1999),
- formálne modely – do tejto skupiny patria napríklad referenčné modely:
  - *Trellis Model* (Furuta, 1990), opísaný pomocou Petriho sietí,
  - *Dexter Hypertext Reference Model* (Halasz, 1990), pôvodne opísaný v jazyku Z, neskôr opísaný pomocou jazyka Object-Z (van Ossenbruggen, 1995),
  - *Munich Reference Model for Adaptive Hypermedia Applications* (Koch, 2002), špecifikácia Dexterovského referenčného modelu pomocou UML-OCL,
  - *Dortmund Family of Formal Models* (Tochtermann, 1996), špecifikácia založená na VDM (Vienna Development Model).

Uvedené modely boli navrhnuté s rôznym zámerom. Kým napríklad Dexterovský model definuje slovník pojmov, HAM opisuje architektonický pohľad na systém a Trellis formálne špecifikuje hypertexty. V nasledujúcich kapitolách uvedieme charakteristiku najvýznamnejších referenčných modelov.

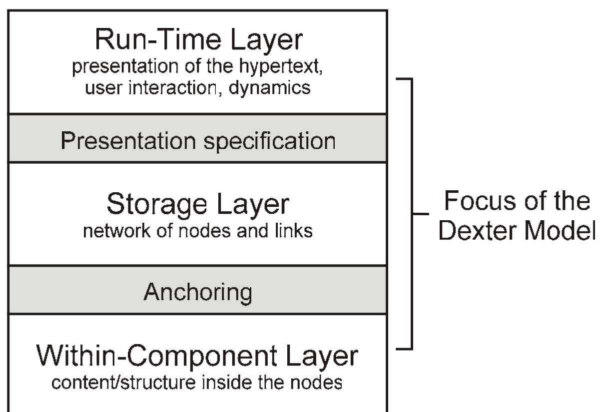
Dexterovský referenčný model

Nepochybne najvýznamnejším referenčným modelom je Dexterovský referenčný model (*Dexter Hypertext Reference Model*), ktorý bol vytvorený na pracovnom stretnutí v Dexter Inn, Sunapee, New Hampshire a bol nazvaný podľa miesta svojho vzniku (Halasz, 1990; Halasz, 1994). Cieľom bolo vytvoriť spoločný jazyk pre ľudí, ktorí sa angažujú vo vývoji hypermédií a vytvoriť abstraktný pohľad na hypermédiá na základe existujúcich aplikácií. Z Dexterovského modelu vychádzajú viaceré ďalšie modely, ktoré ho rôznym spôsobom rozširujú.

Dexterovský model definuje tri vrstvy (pozri obrázok 9-2; Halasz, 1994):

- *Run-Time Layer*, ktorá obsahuje opis prezentácie uzlov a odkazov, zodpovedá za interakciu s používateľom, získanie odozvy od používateľa a manažment sedení,
- *Storage Layer*, ktorá uchováva a sprístupňuje informácie o štruktúre hypermediálneho obsahu a
- *Within-Component Layer*, ktorá obsahuje informačný obsah spolu so štruktúrou hypermediálnych uzlov.

Dexterovský model formalizuje vrstvu *Storage Layer* spolu s rozhraniami (*Presentation Specification*, *Anchoring*) a sčasti vrstvu *Run-Time Layer*. Vrstva *Within-Component Layer* je špecifická pre každú hypermediálnu aplikáciu.



Obrázok 9-2. Vrstvy Dexterovského referenčného modelu.

Hlavným cieľom tohto referenčného modelu je opis siete uzlov a odkazov medzi nimi, nazývanej tiež štruktúra hyperpriestoru, ktorú definuje stredná vrstva. S uzlami sa v tejto vrstve pracuje ako so všeobecnými skladmi dát. Hyperpriestor je definovaný ako konečná množina komponentov, kde komponent môže byť buď atóm (nedeliteľná jednotka), odkaz, alebo zložený komponent, pozostávajúci z atómov alebo ďalších zložených komponentov. Uzol hyperpriestoru je reprezentovaný ako atóm. Špecifikácia komponentu zahŕňa:

- *identifikátor*, ktorý jednoznačne identifikuje daný komponent v rámci celého hyperpriestoru, označovaný tiež `UID`,
- *zoznam atribútov*, ktoré charakterizujú vlastnosti daného komponentu,
- *zoznam kotiev*, pomocou ktorých je možné vyjadriť prepojenie daného komponentu s ďalšími komponentmi,
- *špecifikáciu prezentácie*, ktorá predstavuje rozhranie komponentu s vrstvou *Run-Time Layer*,
- *špecifikáciu obsahu*, ktorá predstavuje rozhranie komponentu s vrstvou *Within-Component Layer*.

Ako sme uviedli vyššie, odkaz sa modeluje ako špeciálny typ komponentu, ktorý má okrem vyššie uvedených súčastí navyše definovaný zoznam najmenej dvoch špecifikátorov určujúcich komponenty, ktoré daný odkaz prepája. Definícia špecifikátora zahŕňa:

- *špecifikáciu komponentu*, ktorá určuje, s ktorým komponentom je daný špecifikátor previazaný,
- *identifikátor kotvy*, ktorý jednoznačne identifikuje kotvu previazanú so špecifikátorom,
- *smer odkazu*, teda či je špecifikátor zdrojom odkazu, cieľom, oboma alebo ani jedným,
- *špecifikáciu prezentácie*, ktorá predstavuje rozhranie s vrstvou *Run-Time Layer*.

Pomocou komponentov je možné nadefinovať ľubovoľnú štruktúru hyperpriestoru. Kľúčovou je existencia mechanizmu kotvenia odkazov. Každá kotva má priradený

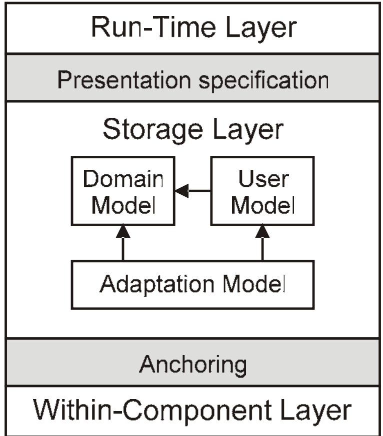
jedinečný identifikátor v rámci daného konceptu, označovaný tiež `AnchorID` a taktiež rozhranie s vrstvou *Within-Component Layer*, ktoré zabezpečuje adresovanie v rámci obsahu prislúchajúceho komponentu. Akákoľvek časť obsahu hyperpriestoru je tak jednoznačne adresovateľná pomocou dvojice `UID-AnchorID`.

AHAM model

AHAM model (*Adaptive Hypermedia Architecture Model;* De Bra, 1999) je rozšírením Dexterovského referenčného modelu o prvky prispôsobovania. Pôvodne bol navrhnutý pre výučbové adaptívne hypermédiá, neskôr bol zovšeobecnený na univerzálny referenčný model pre adaptívne hypermédia. AHAM opisuje AH systém najmä z pohľadu dát.

Pôvodnú strednú vrstvu Dexterovského referenčného modelu predstavuje:

- *model aplikačnej domény* (*Domain Model*), ktorý opisuje štruktúru obsahu hypermediálneho systému a vzťahy medzi jednotlivými komponentmi (v zmysle navigácie).



Obrázok 9-3. Vrstvy referenčného modelu AHAM.

Okrem toho referenčný model AHAM zavádza (pozri obrázok 9-3):

- *model používateľa* (*User Model*), ktorého úlohou je udržiavať hodnoty atribútov konceptov pre jednotlivých používateľov, aby tieto mohol využívať ako zdroj prispôsobovania. Tento model sa tiež označuje ako prekryvný model (*Overlay Model*). Súčasťou definície atribútov konceptov je aj definícia ich prednastavených hodnôt, ktoré sa používajú pri inicializácii modelu používateľa,
- *model prispôsobovania* (*Adaptation Model*), ktorý obsahuje pravidlá prispôsobovania. Pravidlá určujú jednak spôsob aktualizácie modelu používateľa počas práce používateľa so systémom a tiež aj spôsob prispôsobovania prezentácie, kde sa ako zdroj prispôsobovania využíva model používateľa. Tento model bol pôvodne pomenovaný ako model vyučovania (*Teaching Model*) a pravidlá, ktoré obsahoval sa nazývali pedagogické pravidlá, no neskôr bol zovšeobecnený.

Šípky medzi modulmi znázorňujú závislosti. Model aplikačnej domény nijako nezávisí od modelu používateľa, ani od modelu prispôsobovania. Model používateľa závisí od

modelu aplikačnej domény, pretože jeho úlohou je udržiavať okrem iných aj hodnoty atribútov konceptov. Model prispôsobovania obsahuje znalosti o prispôsobovaní, ktoré ako zdroj prispôsobovania používajú atribúty konceptov aplikačnej domény, rovnako ako aj iné atribúty modelu používateľa.

Aby bolo prispôsobovanie realizovateľné, je potrebné, aby AH systém obsahoval mechanizmus prispôsobovania, ktorý dokáže na základe stavu modelu používateľa vyhodnotiť definované pravidlá prispôsobovania a zabezpečiť prispôsobovanie obsahu, navigácie a/alebo formy prezentácie.

Tento model bol opísaný semiformálne formou definícií v prirodzenom jazyku. Mnohé z jeho čŕt sú podobné s Mníchovským referenčným modelom, ktorý charakterizujeme v nasledujúcej kapitole.

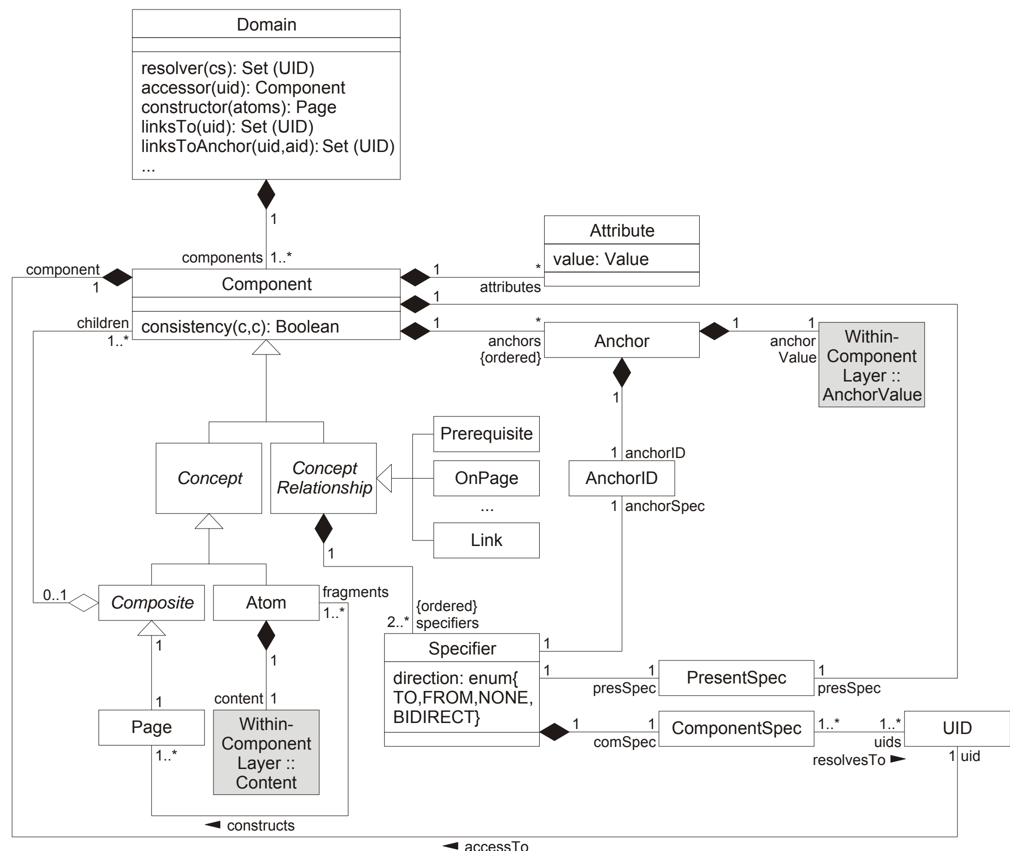
Mníchovský referenčný model pre AHS

Mníchovský referenčný model pre adaptívne hypermediálne systémy (*Munich Reference Model for Adaptive Hypermedia Systems;* Koch, 2002) vychádza z Dexterovského modelu, ktorý rovnako ako AHAM rozširuje o prvky prispôsobovania. Jeho špecifikácia je založená na objektovo-orientovanom prístupe s využitím jazyka UML (*Unified Modelling Language;* OMG, 2003) – vizuálnej reprezentácie diagramami ako aj formálnej špecifikácie v jazyku OCL (*Object Constraint Language*).

Rovnako ako AHAM model rozširuje strednú vrstvu Dexterovského referenčného modelu o model používateľa a model prispôsobovania. Štruktúra hyperpriestoru je opísaná modelom aplikačnej domény. Obrázok 9-4 (Koch, 2002) znázorňuje navrhnutý metamodel aplikačnej domény vo forme diagramu tried.

Doména (`Domain`) je definovaná pomocou sady komponentov (`Component`), ktoré majú jedinečný identifikátor (`UID`). Komponent môže byť buď koncept (`Concept`) alebo konceptuálna väzba (`Concept Relationship`). Koncept môže byť zložený (`Composite`) alebo elementárny (`Atom`). Konceptuálne väzby slúžia na vyjadrenie vzťahov medzi konceptmi. Modelujeme nimi napr. odkazy (`Link`) ale aj iné typy vzťahov ako napr. prerekvizita (`Prerequisite`). Konceptuálne väzby definujú špecifikátory (`Specifier`) nepriamo pomocou kotiev. Väzby môžme modelovať medzi dvomi a viacerými konceptmi, pričom je možné určiť aj ich smer (atribút `direction` triedy `Specifier`). Kotva (`Anchor`) má definovaný identifikátor (`AnchorID`) a hodnotu (`AnchorValue`). Hodnota kotvy, podobne ako informačný obsah združený s konceptom je definovaný vrstvou *Within-Component Layer*. Elementárnemu konceptu prislúcha informačný fragment, zloženému konceptu stránka, ktorú tvorí viacero informačných fragmentov. K jednotlivým komponentom možno definovať premenlivý počet atribútov (`Attribute`). Atribúty majú uplatnenie pri adaptívnej prezentácii, ale aj pri vyhľadávaní informácií, klasifikácii komponentov a pod. Každý komponent povinne obsahuje aj špecifikáciu prezentácie komponentu (`PresentSpec`).

Ako možno vidieť, model podporuje oddelenú reprezentáciu konceptov, vzťahov medzi konceptmi a informačnými fragmentmi (obsahom elementárnych konceptov). Uvedený model neumožňuje explicitnú reprezentáciu variantov (na úrovni konceptov alebo na úrovni informačných fragmentov). Varianty možno reprezentovať pomocou špeciálneho typu konceptuálnej väzby na úrovni elementárnych konceptov, ale aj zložených konceptov.

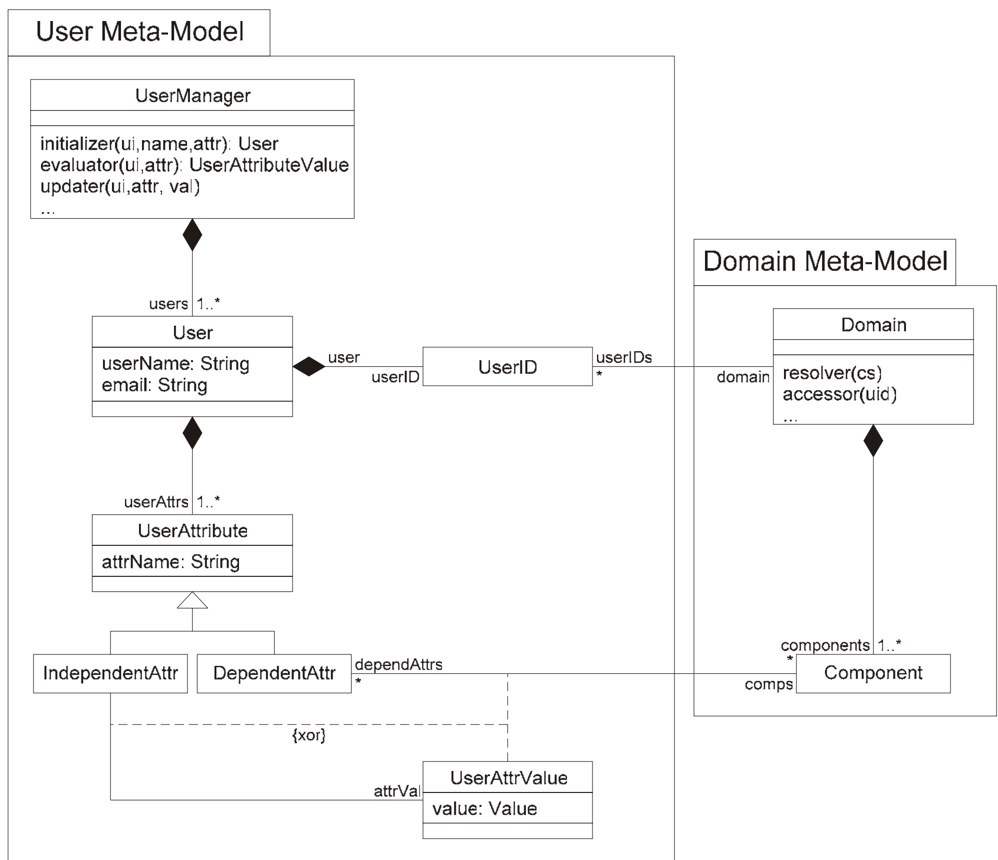


Obrázok 9-4. Metamodel aplikačnej domény Mníchovského referenčného modelu.

Metamodel používateľa, uvedený na obrázku 9-5 (Koch, 2002) vo forme diagramu tried, znázorňuje štruktúru individuálnych modelov používateľa. Model používateľa je pre každého používateľa jedinečný a obsahuje hodnoty atribútov, ktoré slúžia ako zdroj prispôsobovania.

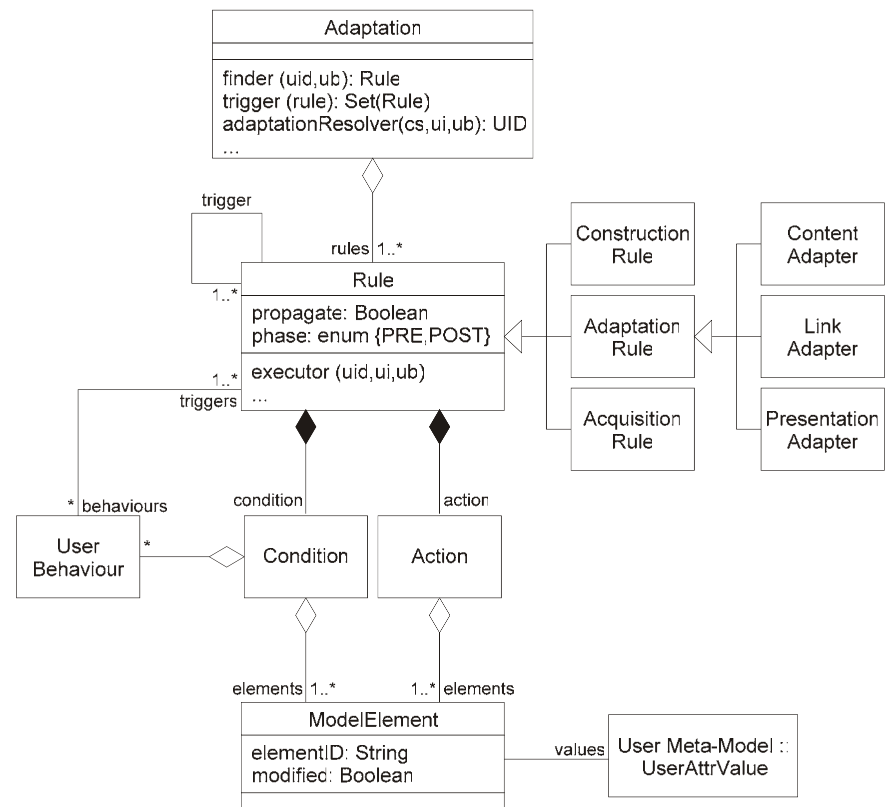
Z hľadiska použitia modelu používateľa je potrebný mechanizmus inicializácie modelu, spôsob aktualizácie modelu, ako aj vyhodnocovanie modelu za účelom prispôsobovania. Sybsystém pre prácu s modelom používateľa je reprezentovaný triedou `UserManager`. Pracuje nad množinou používateľov, pričom každý používateľ (`User`) má v rámci domény priradený jedinečný identifikátor (`UserID`). Model používateľa udržiava pre jednotlivé atribúty (`UserAttribute`) ich hodnoty (`UserAttrValue`). Atribút pritom môže byť doménovo špecifický (`DependentAttr`), kedy je združený s patričným komponentom z modelu aplikačnej domény, alebo nezávislý od domény (`IndependentAttr`).

Pre prispôsobovanie je okrem modelu používateľa nutný aj model prispôsobovania. Obrázok 9-6 (Koch, 2002) znázorňuje navrhnutý metamodel prispôsobovania vo forme diagramu tried.



Obrázok 9-5. Metamodel používateľa Mníchovského referenčného modelu.

Subsystém pre prispôsobovanie (`Adaptation`) obsahuje definíciu pravidiel prispôsobovania (`Rule`). Každé pravidlo má definovanú podmienku (`Condition`) a akciu, ktorá je pri splnení predchádzajúcej podmienky vykonaná. Obe sú definované nad určitým elementom modelu prispôsobovania (`ModelElement`), ktorý je previazaný s atribútmi z modelu používateľa (`User Meta-Model :: UserAttrValue`). Akcia, ktorá bude vykonaná súvisí s typom pravidla. Môže ísť o pravidlo, ktoré definuje spôsob aktualizácie modelu používateľa (`Acquisition Rule`), o pravidlo, ktoré definuje spôsob konštrukcie prezentovaných informácií (`Construction Rule`) alebo o pravidlo, ktoré určuje spôsob, akým sú informácie prezentované (`Adaptation Rule`). V poslednom prípade môže ísť o pravidlo určujúce prispôsobovanie prezentovaného obsahu (`Content Adapter`), prispôsobovanie navigácie (`Link Adapter`) alebo prispôsobovanie formy prezentácie (`Presentation Adapter`). Súčasťou vyhodnotenia pravidla môže byť aj vyhodnotenie ďalších pravidiel, napríklad pri zmene hodnoty atribútu v modeli používateľa môžu byť automaticky vyhodnotené aj pravidlá združené s atribútom, ktorého hodnota sa mení. Súčasťou definície pravidla je preto aj príznak, či sa majú vyhodnocovať prípadné združené pravidlá (atribút `propagate` triedy `Rule`). Pravidlo sa môže vyhodnotiť pred alebo až po prispôsobení prezentovaných informácií (atribút `phase`).



Obrázok 9-6. Metamodel prispôsobovania Mníchovského referenčného modelu.

V predchádzajúcich kapitolách sme charakterizovali najznámejšie referenčné modely adaptívnych hypermediálnych systémov. V nasledujúcich kapitolách uvedieme stručné porovnanie spôsobov vytvárania spomínaných modelov v existujúcich hypermediálnych systémoch.

9.3.2 Model aplikačnej domény

Na modelovanie statickej štruktúry konceptov sa v existujúcich metódach používajú zvyčajne semiformálne techniky ako diagramy tried, resp. entitno-relačné (E-R) diagramy. Modelovanie tak spočíva v definícii konceptov ako tried/objektov a ich atribútov/vlastností. Koncepty sú navzájom poprepájané niekoľkými typmi vzťahov (asociácia, generalizácia/špecializácia, kompozícia). Jednou z metód, ktorá je založená na využitý klasických E-R diagramov je *Relationship Management Metodology* (RMM; Isakowitz, 1995). Ďalšími sú napr. *Hypermedia Design Method* (HDM; Garzotto, 1993), ktorá využíva modifikované E-R diagramy, alebo *Web Modelling Language* (WebML; Ceri, 2002). Z metód, ktoré sú založené na použití diagramov tried možno spomenúť napr. *Object-Oriented HDM* (OOHDM; Schwabe, 1998), *UML based Web Engineering* (UWE; Hennicker, 2000) alebo W2000 (Baresi, 2001). Podrobný prehľad notácií používaných na modelovanie aplikačnej domény s využitím semiformálnych techník je uvedený aj v (Dolog, 2002). Spôsob definovania modelu aplikačnej domény v systéme AHA! (De Bra, 1998) je uvedený v (Bieliková, 2004).

Okrem semiformálnych techník sa na modelovanie aplikačnej domény používajú metódy založené na grafoch. Jedným z prvých v tejto oblasti bol systém *NoteCards* (Halasz, 2001), kde modelovanie aplikačnej domény spočíva v definícii tzv. NoteCards, obsahujúcich modelované informácie. Tieto sú začlenené do kontajnerov, pričom kontajnery môžu obsahovať ďalšie kontajnery. Celkovo tak vytvárajú orientovaný acyklický graf.

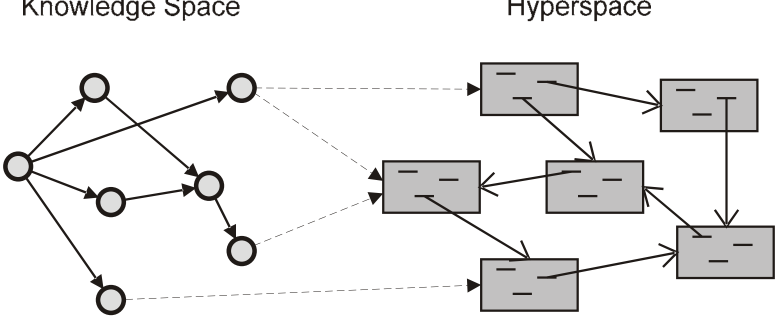
Spomínané metódy modelovania aplikačnej domény pracujú nad uzavretým informačným priestorom. To znamená, že autor, ktorý vytvára model dopredu pozná obsah informačného priestoru. Na základe toho je schopný navrhnúť všetky koncepty a vzťahy medzi nimi. Relatívne novou oblasťou je prispôsobovanie v otvorenom informačnom priestore. V tomto prípade už nie je možné dopredu definovať celý obsah aplikačnej domény a preto sú potrebné mechanizmy, ako opísať význam modelovaných dát tak, aby bol mechanizmus zabezpečujúci prispôsobovanie schopný rozpoznať, o aké informácie ide a vedel ich patrične prispôsobiť.

Metódy, ktoré si dávajú za cieľ dosiahnutie prispôsobovania v otvorenom informačnom priestore sú metódy založené na technológii webu so sémantikou. Informácie z aplikačnej domény sú reprezentované v známej ontológii pomocou definovaných štandardov ako RDF(S) (W3C, 2004a; W3C, 2004b), prípadne OWL (W3C, 2004c). Jedným z prvých príkladov použitia tohto prístupu sú napr. projekty *Hera* (Frasincar, 2001) alebo *Personal Reader* (Dolog, 2004). Táto oblasť je relatívne nová a poskytuje široké možnosti ďalšieho výskumu.

9.3.3 Model navigácie

Model navigácie stavia na modeli aplikačnej domény. Nad známou štruktúrou konceptov definuje jednotlivé stránky hyperpriestoru a možnosti navigácie medzi. Konceptom priraďuje zodpovedajúce fragmenty informačného priestoru. Mapovanie medzi stránkami a konceptmi nemusí byť (a často ani nie je) jedna k jednej. Situáciu znázorňuje obrázok 9-7 (Brusilovsky, 2003).

Uvedený obrázok naznačuje mapovanie medzi priestorom znalostí (*Knowledge Space*) reprezentovaným grafom konceptov a hyperpriestorom (*Hyperspace*) pozostávajúcim zo stránok obsahujúcich fragmenty informačného priestoru.



Obrázok 9-7. Mapovanie medzi konceptmi a informačnými fragmentmi.

Z pohľadu modelovania navigácie môžme hovoriť o dvoch aspektoch. Prvým je statická štruktúra navigácie, teda spôsob, ako sú navrhnuté jednotlivé stránky hyper-priestoru a ako sú navzájom poprepájané. Opisujú sa tak možnosti prechodov medzi

jednotlivými stránkami z pohľadu používateľa. V dnešných hypermediálnych systémoch však už iba tento statický aspekt navigácie nestačí. Dnes už majú hypermediálne systémy mnohokrát charakter webových systémov, kde akcia vyvolaná prechodom medzi dvoma stránkami spôsobuje reakciu systému, ktorá sa najtypickejšie odzrkadlí v zmene stavu systému. Je preto potrebné modelovať aj behaviorálny aspekt navigácie.

Štrukturálne techniky pre modelovanie navigácie sú použité napríklad v OOHDM (Schwabe, 1998), kde definícia modelu navigácie spočíva v definícii tzv. navigačných tried a navigačných kontextov. Navigačné triedy tvoria mapovanie nad modelom aplikačnej domény, pričom koncepty rozširujú o ďalšie atribúty potrebné z hľadiska navigácie. Navigačné kontexty združujú navigačné triedy do navigačných celkov ako sú napríklad indexy, priame vedenie a pod. Vo WebML (Ceri, 2002) je model navigácie tvorený pohľadmi (*SiteView*), ktoré obsahujú definície jednotlivých stránok hyperpriestoru aj s ich obsahom. Tieto môžu byť zoskupené do oblastí stránok (*PageArea*) a navzájom poprepájané kontextovými, resp. nekontextovými odkazmi. Kontextové odkazy sú nositeľmi informácie potrebnej pre zobrazenie cieľovej stránky, kým nekontextové odkazy takúto informáciu nenesú. Súčasťou modelu navigácie je aj definícia operácií, ktorých úlohou je zmena stavu systému. Podobnými štrukturálnymi technikami sa definuje model navigácie aj v UWE (Hennicker, 2000), RMM (Isakowitz, 1995) alebo HDM (Garzotto, 1993). Podrobný prehľad notácií používaných na modelovanie navigácie s využitím štrukturálnych techník je uvedený aj v (Dolog, 2002).

Behaviorálny aspekt navigácie sa najčastejšie modeluje pomocou stavových diagramov, resp. diagramov činností, prípadne Petriho sietí. Prístup k modelovaniu navigácie pomocou binárnych Petriho sietí je opísaný v (Stotts, 1989). *Extended Hyperdocument Model Based on Statecharts* (XHMBS; Paulo, 1999) rozšíril Harelove stavové stroje (Harel, 1987) pre použitie v oblasti modelovania hypermédií. Tento prístup integruje štrukturálne vlastnosti stavových strojov (AND a OR kompozícia stavov) o behaviorálne vlastnosti (udalosti a prechody).

Náš príspevok k rozšíreniu štrukturálneho spôsobu modelovania navigácie vo WebML o behaviorálny aspekt je uvedený v (Kuruc, 2004). Tento prístup, ktorý je založený na transformácii statického hypertextového modelu do modelu reprezentovaného stavovými strojmi, vyjadruje dynamický pohľad na navrhovaný hyperpriestor.

9.3.4 Model kontextu

Základným zdrojom pre prispôsobovanie sú charakteristiky používateľa, resp. prostredia, v ktorom používateľ pracuje. Charakteristikou používateľa môže byť napríklad znalosť doménovej oblasti nad ktorou pracuje AH systém, jeho ciele, záujmy, preferencie a pod. Všeobecne sú charakteristiky používateľa vyjadrené modelom používateľa. Pokiaľ pri prispôsobovaní berieme do úvahy aj prostredie, v ktorom daný používateľ pracuje, hovoríme o modeli prostredia. Obidve dimenzie prispôsobovania sa spoločne označujú ako model kontextu (Bieliková, 2003).

Tak ako to naznačujú referenčné modely uvedené v kapitole 9.3.1, z hľadiska modelovania ide o dvojice atribút-hodnota. Atribút môže byť buď nezávislý od domény, kedy reprezentuje všeobecnú charakteristiku používateľa, alebo doménovo závislý, kedy je združený s konkrétnym konceptom z modelu aplikačnej domény. V prvom prípade to môže byť napr. preferencia jazyka, v ktorom používateľ

uprednostňuje prezentovanie informácií. V druhom prípade môže napr. vyjadrovať znalosť daného konceptu.

Z hľadiska AH systému je potrebné zabezpečiť nasledovné druhy činností:

- inicializáciu modelu používateľa/prostredia,
- získavanie údajov o používateľovi/prostredí,
- spracovanie získaných údajov a následná aktualizácia modelu používateľa/prostredia,
- použitie modelu používateľa/prostredia pri prispôsobovaní.

Inicializácia modelu používateľa môže byť realizovaná priradením prednastavených hodnôt atribútov, alebo vstupným otestovaním používateľa ešte predtým, ako začne so systémom pracovať. Prednastavené hodnoty atribútov sú väčšinou súčasťou definície modelu používateľa.

Získavanie údajov o používateľovi/prostredí sa môže diať automaticky, napr. sledovaním jeho činností, alebo manuálne, formou dotazníkov. Sledovanie činností môže zahŕňať sledovanie, na ktoré odkazy používateľ kliká, ktoré položky v menu najčastejšie volí, ako dlho trávi čas na jednotlivých stránkach a pod. Použitie dotazníkov je vhodné napr. v prostredí výučbových systémom, kedy sa po prebraní určitého študijného materiálu môžu preskúšať študentove vedomosti a na základe nich sa môže aktualizovať model používateľa.

Spracovanie získaných údajov a následná aktualizácia modelu používateľa/prostredia rovnako ako použitie modelu používateľa/prostredia prebiehajú automaticky a sú definované pravidlami v modeli prispôsobovania.

Model používateľa

Model používateľa sa najčastejšie realizuje dvoma spôsobmi – pomocou stereotypov alebo prekryvným modelom. *Stereotypy* neumožňujú personalizáciu. Prispôsobovanie sa realizuje na základe príslušnosti používateľa k niektorej zo skupín používateľov. V rámci jednej skupiny používateľov je spôsob prispôsobovania jednotný. Výhodou však je nenáročnosť tejto metódy, pretože si nevyžaduje udržiavanie inštancie modelu používateľa pre každého používateľa.

Prekryvný model (*Overlay Model*) naopak predpokladá inštanciu modelu používateľa pre každého používateľa. Týmto spôsobom je AH systém schopný k atribútom definovaným v modeli aplikačnej domény udržiavať ich hodnoty, čo umožňuje prispôsobovanie sa systému jednotlivcom. Typickým príkladom v oblasti výučbových systémov je udržiavanie informácie znalosti študenta o jednotlivých konceptoch. Použitie tohto prístupu má však aj svoje obmedzenia. Problémom prekryvného modelu je potreba jeho inicializácie.

Aj z toho dôvodu sa mnohokrát používa kombinácia spomínaných prístupov, kedy je prekryvný model inicializovaný na základe príslušnosti daného používateľa do niektorej zo skupín používateľov.

Spôsob definície atribútov konceptov, ktorých hodnoty sa ukladajú v modeli používateľa v systéme AHA! je uvedený v (Bieliková, 2004).

Model prostredia

Použitie modelu prostredia nie je v existujúcich AH systémoch bežné. Pri prispôsobovaní prezentácie pritom môže byť dôležitým faktorom prostredie, v ktorom používateľ pracuje. Používateľovi, ktorý si informácie prezerá za počítačom s veľkou farebnou obrazovkou, by sa asi mali zobraziť inak, ako používateľovi, ktorý surfuje cez mobilný telefón. Patria tu však aj iné charakteristiky, ako napr. preferovaný jazyk prezentácie, časové a priestorové súvislosti, sociálne aspekty, ale aj kvalita pripojenia, typ prehliadača a pod. Prispôsobovanie môže zabezpečiť aj výber vhodného média pre prezentáciu. Nepočujúcemu namiesto zvuku systém poskytne obrazovú alebo textovú verziu informácie, používateľovi s nízkym rozlíšením displeja systém ponúkne redukovanejšiu verziu stránky.

Model prostredia môže byť realizovaný formou atribútov nezávislých od domény v absolútnych alebo relatívnych hodnotách, napríklad čas (konkrétny dátum ako absolútna hodnota, označenie dňa vo forme „druhý deň piateho týždňa aktuálneho roku“ ako relatívna hodnota) alebo poloha (konkrétne súradnice ako absolútna hodnota, označenie miesta vo forme „v blízkosti objektu“ ako relatívna hodnota).

Jedným zo systémov, ktorý pri prispôsobovaní uvažuje model prostredia, konkrétne model času, je systém adaptívnej elektronickej nástenky opísaný v (Bieliková, 2004a). Tento systém pri prezentácii informácií, ktoré sprístupňuje formou nástenky berie do úvahy časové obdobie, v ktorom sú informácie prezentované a vyberá len relevantné z nich. Systém obsahuje aj podporu relatívnych hodnôt atribútov, kde je čas platnosti informácie daný formou týždňa semestra.

9.3.5 Model prispôsobovania

Okrem vhodne navrhnutého modelu aplikačnej domény a mechanizmu, ktorý zabezpečí sledovanie charakteristík vplývajúcich na prispôsobovanie musí AH systém obsahovať aj aktívny komponent, ktorý realizuje samotné prispôsobovanie. Vďaka tomuto mechanizmu môže systém vhodne upravovať svoje rozhranie tak, akoby bol vytvorený práve pre daného používateľa s ohľadom na jeho znalosti, ciele, preferencie a pod. V závislosti od typu systému to môže mať prínos napr. v ponúkaní relevantnejších výsledkov hľadania (v prípade vyhľadávačov), v odporúčaní študijného materiálu (v prípade výučbových systémov), v odporúčaní informácií súvisiacich s prezentovanými informáciami (v prípade online informačných systémov) a pod. Význam, ciele a spôsoby prispôsobovania sme podrobne uviedli v kapitole 9.2.

Pri prispôsobovaní je rozumné oddeliť špecifikáciu prispôsobovania (znalosti o prispôsobovaní) od mechanizmu prispôsobovania (interpretácia znalostí o prispôsobovaní). Toto rozdelenie nám umožní pri existencii metaznalostí o prispôsobovaní modifikovať samotnú bázu znalostí o prispôsobovaní.

Z pohľadu modelovania znalostí o prispôsobovaní ide vo väčšine existujúcich AH systémov (De Bra, 1998; De Bra, 1999; Koch, 2002) o reprezentáciu vo forme pravidiel. Zvyčajne ide o pravidlový formalizmus s priamym reťazením, kde je výsledkom vyhodnotenia pravidiel zoznam akcií, ktoré sa vykonajú. Pravidlá sa líšia v závislosti od typu akcií, ktoré spúšťajú:

- pravidlá, ktoré určujú spôsob konštrukcie prezentovaných informácií, resp. určujú vhodnosť jednotlivých konceptov, napr. na základe znalosti problematiky,

- pravidlá, ktoré určujú spôsob aktualizácie modelu kontextu, napr. na základe sledovania činností používateľa alebo vyhodnotením kontextu, v ktorom používateľ pracuje,
- pravidlá, ktoré ovplyvňujú spôsob prezentácie informácií. Tieto pravidlá sa ďalej delia na:
  - pravidlá, ktoré určujú výber vhodných informačných fragmentov,
  - pravidlá, ktoré určujú spôsob odporúčania odkazov, napr. formou ich anotácie, usporiadania a pod.,
  - pravidlá, ktoré určujú formu prezentácie, napr. veľkosť fontu, vhodné médium a pod.

Spôsob definície pravidiel prispôsobovania v systéme AHA! je uvedený v (Bieliková, 2004).

Okrem pravidlového formalizmu je však možné aj použitie iných metód, napr. metód umelej inteligencie. Zdrojom pre prispôsobovanie je model kontextu, preto miera prispôsobovania závisí od miery rozpracovania tohto modelu. Čím viac charakteristík model kontextu zohľadňuje, tým širšie možnosti prispôsobovania sa naskytajú.

Znalosti o prispôsobovaní sa rozlišujú na znalosti všeobecné pre celú aplikačnú doménu a znalosti vzťahujúce sa k špecifickým konceptom alebo skupinám konceptov. Príkladom všeobecného pravidla je pravidlo (bežne používané v súčasných webových prehliadačoch), ktoré v závislosti od toho, či už používateľ nasledoval odkaz na stránke tento prezentuje inak, ako keď ešte spomínaný odkaz používateľ nenavštívil. Príkladom pravidla vzťahujúceho sa k špecifickému konceptu je pravidlo, ktoré určí spôsob prezentácie informačného fragmentu, ktorý prislúcha danému konceptu, v závislosti od znalosti tohto konceptu.

V poslednej dobe sa zaujímavou oblasťou stáva tzv. sociálna adaptívna navigácia (Brusilovsky, 2004). Ide o skúmanie správania sa členov rovnakej komunity, ktoré sa neskôr využíva pre potreby prispôsobovania. Táto metóda je založená na predpoklade, že odkazy, ktoré nasleduje člen určitej komunity sú relevantné aj pre iných členov tejto komunity. Pri prispôsobovaní odkazov sa teda berú do úvahy predchádzajúce rozhodnutia členov komunity a uprednostňované sú tie odkazy, ktoré už predtým členovia komunity nasledovali, a teda označili ako relevantné. Význam tejto metódy rastie najmä v prostredí otvoreného informačného priestoru, kde je viac ako potrebné vhodné filtrovanie možností navigácie z dôvodu rizika straty v informačnom priestore.

9.4 Záver

V tejto kapitole sme sa venovali problematike modelovania adaptívnych hypermediálnych systémov, ako podskupiny adaptívnych webových softvérových systémov. Uviedli sme podrobnú charakteristiku AH systémov. Rozobrali sme ciele, ktoré sa pri tvorbe takýchto systémov sledujú. Uviedli sme možné prínosy a riziká ich použitia, ako aj známe oblasti ich využitia. Podrobne sme charakterizovali možnosti prispôsobovania vo forme prehľadu existujúcich metód a techník prispôsobovania obsahu, navigácie a prezentácie Súčasťou práce je aj zhrnutie problémov spojených s tvorbou AH systémov, ktorých čiastočným riešením je existencia vhodných metód

modelovania. V ďalšej časti práce sme sa preto venovali problematike modelovania AH systémov.

Počas štúdia problematiky sme identifikovali niekoľko otvorených problémov a možností ich riešenia.

Väčšina existujúcich AH systémov pracuje nad uzavretým informačným priestorom. Dôsledkom je dobre známa štruktúra informácií, vďaka čomu sa dajú podrobne špecifikovať znalosti o prispôsobovaní. V prostredí otvoreného informačného priestoru je však definovanie znalostí o prispôsobovaní vážnym problémom. Základným predpokladom prispôsobovania je totiž poznanie významu informácií, ktoré chceme prispôsobovať. V opačnom prípade nie sme schopní určiť, ako sa majú prispôsobiť. Problém otvoreného informačného priestoru je v tom, že nie je dopredu známa jeho štruktúra a význam a preto sú potrebné iné mechanizmy definovania modelu aplikačnej domény. Vhodným riešením je *využitie technológií webu so sémantikou*. Táto technológia poskytuje univerzálne prostriedky na explicitnú špecifikáciu vzťahov medzi časťami informačného priestoru. Existujú jazyky na zápis týchto vzťahov (RDF(S), OWL), ako aj prostriedky umožňujúce usudzovanie a objavovanie nových súvislostí (RDQL), čo môže byť výhodné práve z pohľadu prispôsobovania. Táto oblasť sa v poslednom období intenzívne rozvíja a sľubuje široké možnosti uplatnenia. Použitie technológie webu so sémantikou má význam aj pri *zdieľaní modelu kontextu* medzi viacerými AH systémami. Riešením je reprezentácia modelu kontextu vo vhodnej ontológii, vďaka čomu je explicitne daný význam jednotlivých atribútov modelu.

Literatúra

ALBRECHT, F. – KOCH, N. – TILLER, T. (2000). SmexWeb: An adaptive web-based hypermedia teaching system. *Journal of Interactive Learning Research, Special Issue on Intelligent Systems/Tools in Training and Life-Long Learning*, vol. 11, no. 3-4, pp. 367–388.

BARESI, L. – GARZOTTO, F. – PAOLINI, P. (2001). Extending UML for modelling web applications. In: *Proc. of the 34th Hawaii International Conference on System Sciences*, HICCS-34. IEEE Computer Society.

BIELIKOVÁ M. (2003). Presentation of adaptive hypermedia on the web. In: *Proc. of DATAKON 2003*. L. Popelínský (Ed.), Brno, Czech Republic, pp. 72–91.

BIELIKOVÁ, M. – HABALA, R. (2004). University course support by web-based adaptive e-board. In: *Proc. of ICETA 2004*. F. Jakab, L. Samuelis, I. Sivý, M. Bučko (Eds.), Košice, Slovak Republic, pp. 395–402.

BIELIKOVÁ, M. – KURUC, J. (2004). Tvorba obsahu v adaptívnych hypermédiách pre e-vzdelávanie. In: *Proc. of Technologie pro e-vzdělávání*. B. Mannová, P. Šaloun, and M. Bieliková (Eds.), Praha, Czech Republic, pp. 11–21.

BRUSILOVSKY. P. (1996). Methods and techniques of adaptive hypermedia. *User Modeling and User-Adapted Interaction*, vol. 6, no. 2-3, pp. 87–129.

BRUSILOVSKY, P. (2001). Adaptive hypermedia. User Modeling and User-Adapted Interaction, vol. 11, no. 1-2, pp. 87–110.

BRUSILOVSKY, P. (2003). Developing adaptive educational hypermedia systems: From design models to authoring tools. In: *Authoring Tools for Advanced Technology Learning Environments*. T. Murray, S. Blessing, and S. Ainsworth (Eds.), Kluwer Academic Publishers.

BRUSILOVSKY, P. – CAVAN, G. – FARZAN, R. (2004). Social adaptive navigation support for open corpus electronic textbooks. In: *Proc. of AH'2004 – International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*. W. Nejdl and P. de Bra (Eds.), Eindhoven, The Netherlands, Springer Verlag, LNCS 3137,pp. 24–33.

CAMPBELL, B. – GOODMAN, J.M. (1988). HAM: A general purpose hypertext abstract machine. *Communication of the ACM*, vol. 31, no. 7, pp. 856–861.

CERI, S. – FRATERNALI, P. – BONGIO, A. – BRAMBILLA, M. – COMAI, S. – MATERA, M. (2002). Designing Data-Intensive Web Applications. Morgan Kaufmann Publishers.

DE BRA, P. – CALVI, L. (1998). AHA: A generic adaptive hypermedia system. In: *Proc. of the 2nd Workshop on Adaptive Hypertext and Hypermedia*, Pittsburg, USA, pp. 5–12.

DE BRA, P. – HOUBEN, G.-J. – WU, H. (1999). AHAM: A Dexter-based reference model for adaptive hypermedia. In: *Proc. of the tenth ACM Conference on Hypertext and hypermedia: returning to our diverse roots*, Darmstadt, Germany, pp. 147–156.

DOLOG, P. – BIELIKOVÁ, M. (2002). Hypermedia systems modelling framework. *Computing and Infromatics*, vol. 21, no. 3, pp. 221–239.

DOLOG, P. – HENZE, N. – NEJDL, W. – SINTEK, M. (2004). The Personal reader: Personalizing and enriching learning resources using semantic web technologies. In: *Proc. of AH'2004 – International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*. W. Nejdl and P. de Bra (Eds.), Eindhoven, The Netherlands, Springer Verlag, LNCS 3137, pp. 85–94.

FRASINCAR, F. – HOUBEN, G.-J. (2001). XML-based automatic web presentation generation. In: *Proc. of World Conference on the WWW and Internet*, WebNet 2001, Orlando, Florida, AACE, pp. 372–377.

FURUTA, R. – STOTTS, D.P. (1990). The Trellis hypertext reference model. In: *Proc. of NIST Hypertext Standardization Workshop*, pp. 83–93.

GARZOTTO, F. – PAOLINI, P. (1993). HDM: A model based approach to hypertext application design. *ACM Transactions on Information Systems*, vol. 11, no. 1, pp. 1–26.

HALASZ, F.G. – SCHWARTZ, M. (1990). The Dexter hypertext reference model. In: *Proc. of NIST Hypertext Standardization Workshop*, pp. 95–133.

HALASZ, F.G. – SCHWARTZ, M. (1994). The Dexter hypertext reference model. *Communications of the ACM*, vol. 37, no. 2, pp. 30–39.

HALASZ, F.G. (2001). Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *ACM Journal of Computer Documentation*, vol. 25, no. 3, pp. 71–87.

HARDMAN, L. – BULTERMAN, D. – VAN ROSSUM, G. (1994). The Amsterdam hypermedia model: Adding time and context to the Dexter model. *Communications of the ACM*, vol. 37, no. 2, pp. 50–62.

HAREL, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274.

HENNICKER, R. – KOCH, N. (2000). A UML-based methodology for hypermedia design. In: *Proc. of the Unified Modeling Language Conference*, UML'2000. A. Evans, S. Kent, and B. Selic (Eds.), York, UK, Springer Verlag, LNCS 1939, pp. 410–424.

ISAKOWITZ, T. – STOHR, E.A. – BALASUBRAMANIAN, P. (1995). RMM: A methodology for structured hypermedia design. *Communications of the ACM*, vol. 38, no. 8, pp. 34–44.

JUNGMANN, M. – PARADIES, T. (1997). Adaptive hypertext in complex knowledge domains. In: *Proc. of the Flexible Hypertext Workshop (Hypertext'97).*

KOCH, N.P. (2000). Software engineering for adaptive hypermedia systems: Reference model, modeling techniques and development process. PhD. Thesis, Ludwig-Maximilians-Universität München.

KOCH, N. – WIRSING, M. (2002). The Munich reference model for adaptive hypermedia applications. In: *Proc. of Internation Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*. P. de Bra, P. Brusilovsky, and R. Conejo (Eds.), Springer Verlag, LNCS 2347, Malaga, Spain, pp. 213–222.

KURUC, J. – DOLOG, P. – BIELIKOVÁ, M. (2004). Prototyping navigation in web-based information systems using WebML. In: *Proc. of DATAKON 2004*. K. Ježek (Ed.), Brno, Czech Republic, pp. 253–262.

OBJECT MANAGEMENT GROUP (2003). Unified Modeling Language (UML) Specification: Infrastructure, Version 2.0, Adopted Specification, Dec. 2003.

PATERNO, F. – MANCINI, C. (1999). Designing web interfaces adaptable to different types of use. In: *Proc. of the Workshop Museums and the Web*.

PAULO, F.B. – TURINE, M.A.S. - DE OLIVEIRA, M.C.F. – MASIERO, P.C. (1998). XHMBS: A formal model to support hypermedia specification. In: *Proc. of the 9th ACM Conference on Hypertext*, Hypertext'98, Pittsburgh, USA, pp. 161–170.

SCHWABE, D. – ROSSI, G. (1998). An object-oriented approach to web-based application design. *Theory and Practise of Object Systems* (TAPOS), Special Issue on the Internet, vol. 4, no. 4, pp. 207–225.

STOTTS, D.P. – FURUTA, R. (1989). Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems*, vol. 7, no. 1, pp. 3–29.

TOCHTERMANN, K. – DITTRICH, G. (1996). The Dortmund family of hypermedia models. *Journal of Universal Computer Science*, vol. 2, no. 1, pp. 34–55.

VAN OSSENBRUGGEN, J. – ELIËNS, A. (1995). The Dexter hypertext reference model in Object-Z. Technical report, Vrije Universiteit, Faculty of Mathematics and Computer Sciences, The Netherlands.

WORLD WIDE WEB CONSORTIUM (2004). RDF/XML Syntax Specification (Revised), W3C Recommendation, Feb. 2004.

WORLD WIDE WEB CONSORTIUM (2004). RDF Vocabulary Description Language 1.0: RDF Schema, Feb. 2004.

WORLD WIDE WEB CONSORTIUM (2004). OWL Web Ontology Language Reference, W3C Recommendation, Feb. 2004.

EDÍCIA VÝSKUMNÝCH TEXTOV
INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Mária Bieliková, Pavol Návrat a kol.

ŠTÚDIE VYBRANÝCH TÉM SOFTVÉROVÉHO INŽINIERSTVA

Pokročilé metódy navrhovania programových systémov
Pokročilé metódy získavania, vyhľadávania, reprezentácie
a prezentácie informácií

1. vydanie
Tlač Vydavateľstvo STU v Bratislave

238 strán

2006

ISBN