Edícia výskumných textov
informatiky a informačných technológií

Štúdie vybraných tém programových a informačných systémov (4)

Mária Bieliková,
Pavol Návrat (editori)

Štúdie vybraných tém programových a informačných systémov 4

Pokročilé metódy navrhovania softvéru
Pokročilé metódy získavania, vyhľadávania, reprezentácie a prezentácie informácií

STU FIIT
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

Štúdie vybraných tém programových a informačných systémov (4)

Pokročilé metódy navrhovania softvéru
Pokročilé metódy získavania, vyhľadávania, reprezentácie
a prezentácie informácií

Mária Bieliková, Pavol Návrat (editori)
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave
Ilkovičova 3, 842 16 Bratislava
http://www.fiit.stuba.sk/, {bielik,navrat}@fiit.stuba.sk

Návrh grafickej úpravy: Anton Andrejko, Mária Bieliková
Technický redaktor: Anton Andrejko
Technická spolupráca: Katarína Mršková, Nikoleta Habudová
Obálka: Peter Kaminský

PUBLIKÁCIU PODPORILO ZDRUŽENIE
GRATEX IT INŠTITÚT
v rámci fondu GraFIIT www.gratex.com

Publikácia bola vydaná s čiastočnou podporou projektov Vedeckej grantovej agentúry Ministerstva školstva Slovenskej republiky a Slovenskej akadémie vied (VEGA)

– VG1/3102/06 Modely softvérových systémov v prostredí webu so sémantikou
– VG1/0508/09 Adaptívny sociálny web a jeho služby pre sprístupňovanie informácií

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

ISBN 978-80-227-3139-3

PREDHOVOR

Publikácia, ktorú dostávate do rúk, je už štvrtou v poradí v *Edícii výskumných textov informatiky a informačných technológií* na témy z oblasti programových a informačných systémov. Táto publikácia ako aj doterajšie *Štúdie* sa venujú dvom ťažiskovým okruhom. Prvým okruhom sú pokročilé metódy navrhovania softvéru. Druhým sú pokročilé metódy získavania, vyhľadávania, reprezentácie a prezentácie informácií. Voľba oboch okruhov tém nebola náhodná. Všetky tieto témy sú aktuálnymi témami súčasného výskumu v oblasti programových a informačných systémov. Ako také sú predmetom záujmu, štúdia a výskumu študentov, t.j. najmä študentov doktorandského štúdia. Oni sú nielen prvými čitateľmi *Štúdií*, vybraní doktorandi sú aj autormi jednotlivých častí v každej z publikácií.

Prvé *Štúdie* sa sústreďovali na dve ťažiskové témy. Prvou témou bola analýza návrhových vzorov, ktoré predstavujú jednu z kľúčových oblastí vyvíjajúcej sa disciplíny softvérového inžinierstva. Druhá časť obsahovala päť štúdií z vybraných tém programových a informačných systémov, ktoré diskutujú a analyzujú otvorené vedecké problémy v predmetnej oblasti aj v spojitosti so spracovaním informácií na internete.

Druhé *Štúdie* sa sústredili vo svoje prvej časti na analýzu rôznych aspektov toho, čo sa začalo nazývať webová inteligencia. V rámci druhej časti sme uviedli štyri štúdie, ktoré diskutujú a analyzujú vybrané otvorené vedecké problémy podobne ako v prvom zväzku.

Tretie *Štúdie* sa v prvej časti venuje otázkam spojeným s architektúrou softvéru. Poskytuje prehľad rôznych prístupov k navrhovaniu a k tvorbe architektúry softvéru, pričom sa zaoberá aj niektorými aplikačnými aspektami. Druhá časť uvádza štyri štúdie koncipované podobne ako v predchádzajúcich zväzkoch.

Obdobný postup ako pri doterajších *Štúdiách* sme zvolili aj pri tomto zväzku. Vznikol na základe seminárov študentov doktorandského štúdia študijného programu programové systémy v odbore softvérové inžinierstvo na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave.

Informatika a informačné technológie sú kľúčovým prvkom budovania modernej spoločnosti „založenej na vedomostiach", ako je dnes módne vravieť. Mladí talentovaní absolventi druhého stupňa vysokoškolského štúdia v oblasti informatiky alebo príbuzných oblastiach majú v súčasnosti veľké možnosti uplatnenia sa v praxi. Súčasná 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. V širšom kontexte ide o rozvoj spoločnosti (ak chcete, založenej na vedomostiach), nielen ekonomiky, schopnej vyrovnávať sa so zložitými výzvami, ktoré pred ňou stoja. 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 sa posilňuje potreba odborníkov v špecializovaných oblastiach, schopných samostatne riešiť otvorené problémy, ktoré nemajú doteraz známe riešenia. Práve doktorandi sa na takúto úlohu pripravujú svojím doktorandským štúdiom. Z iného pohľadu ide totiž o výskum, ktorý je podstatnou náplňou ich štúdia. Jedným z prejavov fungujúcej výskumnej činnosti na pracovisku je seminár. Semináre, ktoré sa uskutočňujú na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave v rámci doktorandského štúdia sa zameriavajú na rôzne oblasti programových a informačných systémov. V prvom zväzku *Štúdií* sme podchytili seminár venovaný návrhovým vzorom a v druhom seminár venovaný webovej inteligencii. V treťom sa seminár sústreďoval na podstatu softvérovej architektúry. V tomto zväzku sme zostali pri téme softvérových systémov. Zvolili sme pohľad, v ktorom je architektúra síce významným, ale len jedným z viacerých podstatných pojmov. Pohľad, ktorý sme v tejto knihe podrobili skúmaniu, nazerá na tvorbu softvérových systémov cez prizmu softvérových paradigiem.

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 dielov. V prvom (Diel 1: Softvérové paradigmy) sa sústreďujeme na prevládajúce spôsoby opisu a navrhovania softvérových systémov na rôznych úrovniach abstrakcie. Druhý (Diel 2: Vybrané témy programových a informačných systémov) obsahuje sedem 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: Softvérové paradigmy

Čo sú to softvérové paradigmy? Jedna cesta, ktorá by mohla viesť k odpovedi, ide cez podrobnejšie preskúmanie pojmu paradigma. Pojem paradigma nadobudol moderný (súčasný) obsah najmä vďaka práci T. Kuhna, v ktorej sa zaoberal paradigmou ako výsledkom vedeckej revolúcie, meniacej náhľad vedeckej komunity príslušnej oblasti na problémy a metódy riešenia prelomovým spôsobom.

Uvádzame tento význam slova paradigma, pretože sa jednoducho v žiadnom vedeckom pojednaní, postavenom na pojme paradigmy, nedá dosť dobre obísť. To však neznamená, že to je jediný relevantný pohľad. Jednotlivé oblasti poznania často používajú pojem paradigma v zmysle, ktorý je do značnej miery odlišný, i keď možno nie priamo protirečivý. Väčšinou sa pojem paradigma v špeciálnych oblastiach poznania používa v omnoho špeciálnejšom zmysle, než ako sa chápe Kuhnovská paradigma. Tu už paradigma nie je nutne výsledkom vedeckej revolúcie či prevratu. Paradigma tiež nie je (jediným) prevládajúcim náhľadom na metódy riešenia problémov príslušnej oblasti. Pokojne môžeme hovoriť o paradigmách v množnom čísle, nakoľko sa v komunite uznáva viacero dosť špecifických schém riešenia problémov.

Takto nejako budeme chápať paradigmu aj my v tejto knihe. Podobne ju chápe S. H. Kaisler vo svojej monografii Softvérové paradigmy, ktorá bola hlavnou odporúčanou literatúrou pre študentov doktorandského seminára. Kaisler sa zaoberá softvérovými paradigmami, členiac ich zhruba podľa granularity častí softvéru, ktorých sa týkajú. Základné členenie, podľa ktorého budeme postupovať, je členenie na triedy návrhových vzorov, softvérových architektúr a rámcov.

Návrhové vzory sú dnes už známou schémou vyjadrenia návrhárskej skúsenosti v oblasti navrhovania softvérových systémov. Budeme sa zaoberať nielen softvérovými vzormi, ako sú Unikát, Abstraktná továreň, Obaľovač, ale aj vzormi pre navrhovanie rozhrania človek-počítač. Okrem toho sa vzory začínajú používať aj v iných doménach.

Ďalšou dôležitou úrovňou členenia softvéru, pre ktorú máme ustálené postupy navrhovania, sú softvérové súčiastky. Špeciálnou skupinou sú súčiastky pre distribuované spracovanie. No a tiež sa budeme venovať paradigmám samotných softvérových architektúr. Tu sa rozlišujú najmä podľa toho, či ide o systém, založený na spracovaní toku dát alebo systém, ktorého štrukturovanie je dané klasickou schémou volaní alebo systém, štrukturovaný ako hierarchia virtuálnych strojov. Neobídeme ani architektúry softvérových systémov súbežného spracovania. Do tohto kontextu dnes už neodmysliteľne patria rámce. Venovať a budeme nielen klasickým rámcom pre grafické rozhranie človek-počítač, ale aj napr. vývojovým rámcom.

Práve toto boli hlavné dôvody, pre ktoré sme sa rozhodli zamerať doktorandský seminár v akademickom roku 2008/09 na softvérové paradigmy v takom chápaní, v akom ho prezentuje uvedená knižka. Vybrané kapitoly sa stali základom pre referáty, ktoré boli úvodmi pre seminárne diskusie. Seminár v rámci doktorandského štúdia viedol Pavol Návrat. Doktorandi, ktorí referáty predniesli, dopracovali ich textovú podobu potom do výsledného tvaru, ktorý máme možnosť čítať v tomto zväzku.

Každá kapitola je tak výsledkom tvorivej činnosti, ku ktorej prispeli viacerí. Samotný text každej časti v rámci jednotlivých kapitol vždy ten-ktorý doktorand. Ako autor vychádzal nielen z uvedenej monografie, ale aj z iných literárnych prameňov, ktoré preštudoval. 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.

Náš výber tém zo softvérových paradigiem, ktorý sme zaradili do seminára (a teda aj do tejto knižky), možno rozčleniť do štyroch okruhov (kapitol tejto publikácie): návrhové vzory (6 tém), softvérové súčiastky (7 tém), architektúry softvéru (8 tém) a rámce (3 témy). Autori sa podieľali na jednotlivých kapitolách takto:

- *Návrhové vzory*
  - Prehľad návrhových vzorov: Nikoleta Habudová
  - Unikát: Ivan Kišac
  - Abstraktná továreň: Tomáš Kuzár
  - Obaľovač: Pavol Mederly
  - Pozorovateľ: Marián Šimko
  - Rozhranie človek-počítač: Jozef Tvarožek

- *Softvérové súčiastky*
  - Softvérové súčiastky a ich modely: Nikoleta Habudová
  - Distribuované súčiastky: Tomáš Kuzár
  - Súčiastky založené na udalostiach: Ivan Kapustík
  - CORBA: Pavol Mederly
  - JavaBeans: Marián Šimko
  - System Object Model: Jozef Tvarožek
  - Softvérové inžinierstvo založené na súčiastkach: Ivan Kapustík
- *Architektúry softvéru*
  - Prehľad architektúr softvéru: Ivan Kišac
  - Systémy riadené tokom údajov: Tomáš Kuzár
  - Dátovody a filtre: Pavol Mederly
  - Volanie a návrat: Jozef Tvarožek
  - Systémy nezávislých súčiastok: Ivan Kapustík
  - Úložisko: Nikoleta Habudová
  - Architektúry súbežného softvéru: Ivan Kišac
  - Výzvy softvérovej architektúry: Tomáš Kuzár
- *Rámce*
  - Základné koncepty rámcov: Ivan Kišac
  - Rámce GUI: Marián Šimko
  - Vývojové rámce: Marián Šimko

Naše *Štúdie* sú dielom, ktoré sa líši od Kaislerovej monografie. Taký bol náš úmysel, Kaislerova monografia bola cennou inšpiráciou a východiskovým zdrojom vedomostí. Sú to dve rozdielne diela. Naše *Štúdie* nemôžu Kaislerovu monografiu nahradiť. Práve naopak, všetkým záujemcom ju vrelo odporúčame.

Diel 2: Vybrané témy programových a informačných systémov

Do druhej časti zaraďujeme osem 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 a pod.

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. Oni im dali písomnú podobu, predložili a aj úspešne obhájili ako písomnú časť svojej dizertačnej skúšky. Autori sa podieľali na jednotlivých kapitolách takto:

- *Znovupoužitie návrhových vzorov na úrovni modelu: Ľubomír Majtás*
  (školiteľ prof. Pavol Návrat)
- *Semantic Web Services: Peter Bartalos*
  (školiteľ: prof. Mária Bieliková)
- *Sprístupňovanie informácií pomocou grafov: Ján Suchal*
  (školiteľ prof. Pavol Návrat)
- *User Modeling for Personalized Web Based Systems: Michal Barla*
  (školiteľ prof. Mária Bieliková)
- *Personalized Collaboration: Jozef Tvarožek*
  (školiteľ: prof. Mária Bieliková)
- *Semantic-based Navigation in Open Spaces: Michal Tvarožek*
  (školiteľ: prof. Mária Bieliková)
- *Získavanie informácií z webu metódami inšpirovanými sociálnym hmyzom: Anna Bou Ezzeddine* (školiteľ prof. Pavol Návrat)

Dúfame, že táto knižka poslúži záujemcom o poznanie programových a informačných systémov. Umožňuje spoločne využiť výsledky štúdia v tejto oblasti. Tešíme sa na prípadné odozvy alebo pripomienky.

August 2009,
Bratislava

Mária Bieliková a Pavol Návrat

OBSAH

DIEL I: SOFTVÉROVÉ PARADIGMY

DIEL I

SOFTVÉROVÉ PARADIGMY

1
NÁVRHOVÉ VZORY

Nikoleta Habudová, Ivan Kišac, Tomáš Kuzár,
Pavol Mederly, Marián Šimko, Jozef Tvarožek

Návrhové vzory sú významným pojmom softvérového inžinierstva, ktorý so sebou prináša objektovo-orientovaná komunita. Návrhový vzor predstavuje abstraktné riešenie problému softvérového návrhu. Prvá zmienka o návrhových vzoroch sa objavila v práci Christophera Alexandra, ale taktiež pochádza z úspešného vzoru Model-Pohľad-Ovládač (angl. *Model-View-Controller, MVC*), ktorý sa využil v jazyku Smalltalk-80 pre návrh grafického rozhrania (angl. *Grafical User Interface,GUI*). Dnes existuje niekoľko kategórií návrhových vzorov naprieč mnohých technických a netechnických disciplín.

Hoci vzory sa v iných odboroch používajú už stovky rokov, v rámci vedy o počítačoch sa im venuje pozornosť iba v poslednom desaťročí. V roku 1987 Cunningham a Beck použili nápad Christophera Alexandra a vyvinuli malý vzorový objektovo-orientovaný programovací jazyk pre Smalltalk-80. Neskôr iní autori postupne zhromažďovali a katalogizovali narastajúci zoznam návrhových vzorov. Nakoniec v roku 1995 skupina autorov známa pod názvom Gang of Four (GoF: Gamma, Helm, Johnson a Vlissides) publikovala svoju knihu návrhových vzorov.

Typy vzorov

Navrhovanie systémov je veľmi náročný problém. Možno ho zjednodušiť znovupoužitím existujúcich návrhov. Pri existujúcich návrhoch treba mať k dispozícii nielen opis, ktorý určuje „čo“ sa navrhuje, ale aj „prečo“. Práve to zdôvodnenie návrhu je rozhodujúce, či daný návrh je aplikovateľný resp. prispôsobiteľný. Existuje niekoľko spôsobov členenia vzorov. Jednou z možností je klasifikácia podľa Copliena na:

- *Generatívne* – špecifikovaním softvérovej architektúry pomáhajú navrhovať a budovať systém. Sú normatívne a aktívne. Majú pozitívny vplyv na štruktúru systému, ktorý sa buduje.
- *Negeneratívne* – opisné a pasívne, zabudované v systéme, nemusí byť za nimi žiadna logika.

Ďalšia možnosť členenia vzorov je založená na úrovni abstrakcie. Riehle a Züllighoven (1996) navrhujú tri typy softvérových vzorov:

- *konceptuálne* – môžu predstavovať komplexný návrh pre celú aplikáciu alebo podsystém,
- *návrhové* – poskytujú riešenia pre všeobecné problémy návrhu v príslušnom kontexte,
- *programovacie* – poskytujú znovu použiteľné súčiastky ako napr. prepojené zoznamy a transformačné tabuľky.

Klasifikáciou návrhových vzorov sa zaoberali GoF. Sadu vzorov, ktorú možno aplikovať v určitej oblasti charakterizujeme ako vzorový priestor (angl. *pattern space*). Doteraz bolo navrhnutých niekoľko vzorových priestorov. Tabuľka 1-1 uvádza prehľad vzorových priestorov ako ich pôvodne prezentovali GoF (Gamma, 1995).

Tabuľka 1-1. Prehľad návrhových vzorov podľa GoF.

Účel	**Návrhový vzor**	
Vzory vytvárania	Abstract Factory	Abstraktná továreň
	Builder	Staviteľ
	Factory Method	Výrobná metóda
	Prototype	Prototyp
	Singleton	Unikát
Štrukturálne vzory	Adapter	Adaptér
	Bridge	Premostenie
	Composite	Zloženina
	Decorator	Dekoratér
	Facade	Fasáda
	Flyweight	Mušia váha
	Proxy	Zástupca
Vzory správania	Chain of Responsibility	Reťaz zodpovednosti
	Command	Príkaz
	Interpreter	Interpreter
	Iterator	Iterátor
	Mediator	Sprostredkovateľ
	Memento	Memento
	Observer	Pozorovateľ
	State	Stav
	Strategy	Stratégia
	Template Method	Šablónová metóda
	Visitor	Návštevník

Vzory GoF sú rozdelené do troch kategórií: vzory vytvárania (angl. *creational patterns*), štrukturálne vzory (angl. *structural patterns*) a vzory správania (angl. *behavioral patterns*).

Vzory vytvárania sa zameriavajú na vytváranie objektov, typicky delegovaním tejto činnosti na vhodný objekt. Štrukturálne vzory sa zaoberajú spôsobmi, akými sú triedy a objekty spájané za účelom vytvárania zložitejších štruktúr. Vzory správania sa zaoberajú rozdelením zodpovednosti medzi objektami a opisom ich komunikácie vedúcej k spolupráci pri vykonávaní úlohy, ktorú nedokáže vykonať len jeden samotný objekt.

Z 23 vzorov sú 4 vzory vzormi tried (Factory Method, Adapter, Interpreter, Template Method), kým ostatných 19 vzorov je objektovými vzormi. Vzory tried sa sústreďujú na statický vzťah medzi triedou a jej podtriedami, zatiaľ čo objektové vzory sa zaoberajú dynamickými vzťahmi, ktoré možno spravovať počas vykonávania programu (angl. *runtime*).

Opis vzoru

Existuje niekoľko spôsobov ako opísať vzory. Autori GoF navrhujú štyri základné charakteristiky pre opis vzorov:

- *názov* – vystihuje základnú charakteristiku vzoru,
- *problém* – opisuje typický problém, na ktorý sa vzor aplikuje,
- *riešenie* – špecifikuje ako riešiť problém vrátane opisu elementov, vzťahov medzi sebou navzájom a zodpovednosti voči ostatným elementom,
- *dôsledky* – poukazujú na to, kedy a ako možno vzor použiť, príp. sa uvádzajú výhody a nevýhody aplikácie vzoru.

Taktiež boli navrhnuté prídavné charakteristiky, ktoré môžu zlepšiť pochopenie vzoru:

- *aplikovateľnosť* – predstavuje obmedzenia, kedy vzor zodpovedá reálnej situácii,
- *príklady* – príklady aktuálneho použitia vzoru pri riešení reálnych problémov,
- *ukážkový kód* – implementácia vzoru v danom kontexte,
- *racionálny význam* – formou stručného vysvetlenia, prečo bolo riešenie aplikované a pre daný problém užitočné,
- *súvisiace vzory* – vzory, ktoré možno použiť súčasne, pričom sa uvedie daný kontext a obmedzenia.

Opis návrhových vzorov má byť nezávislý od programovacieho jazyka alebo implementačných detailov, pretože návrhový vzor je vlastne šablónou, podľa ktorej možno riešenie aplikovať v rôznych situáciách.

Ako nájsť návrhový vzor

Základnou podmienkou pre „objavenie“ návrhového vzoru je, aby programátor riešil daný problém sám a pritom si robil poznámky o tom, ako jednotlivé problémy riešil. S pribúdajúcimi skúsenosťami zistíme, že ide stále o tie isté typy problémov. Vždy, keď sa daný alebo podobný problém vyskytne, poznačíme si v čom je odlišný, v čom je rovnaký a ako sme ho v danom prípade vyriešili. Eventuálne vytvoríme štandardný spôsob na riešenie takéhoto problému, ktorý budeme trvalo používať. Tak vzniká návrhový vzor.

1.1 Unikát

Unikát je vzor vytvárania známy väčšinou pod anglickým názvom *Singleton*, t.j. niečo jedinečné.

1.1.1 Účel

Ako už z názvu tohto návrhového vzoru vyplýva, pri jeho použití pôjde o určitú jedinečnosť. Vzor Unikát zabezpečuje v prvom rade existenciu najviac jednej inštancie triedy,

ktorá bola podľa tohoto vzoru navrhnutá. Príkladom takejto inštancie je (v zvyčajnom stave) napr. prezident krajiny, t.j. maximálne jedna osoba.

Vzor Unikát sa používa najmä pri implementácii tried riadiacich objektov aplikácie, resp. objektov, ktoré budú v aplikácii jediné svojho druhu, čiže každý takýto objekt bude jedinou inštanciou svojej triedy. Takmer každá aplikácia pracuje napr. s jedinou myšou, klávesnicou a pod., kde každé z týchto zariadení môže byť reprezentované jedným objektom. Nehardvérovými príkladmi môže byť napr. súbor denníka aplikácie – *log*, prípadne manažér okien aplikácie a pod. Použitie vzoru Unikát rieši aj problém s prístupom k takýmto objektom. Napomáha v jednoduchosti a možnosti globálneho prístupu k objektu vďaka použitiu statických metód.

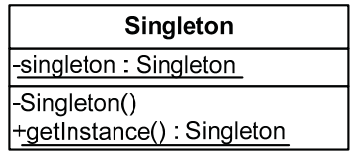
Statické metódy triedy implementovanej na základe vzoru Unikát zabezpečujú aj vytváranie konkrétnych objektov (v skutočnosti jediného objektu) tejto triedy. Tak trieda samotná zaisťuje existenciu práve jednej svojej inštancie.

Použitím Unikátu môžme napríklad zápis do denníku aplikácie realizovať jednoducho vyžiadaním inštancie denníka a zápisom do nej. Či ide o prvý zápis a treba inštanciu a popritom aj reálny súbor vytvoriť, vyhodnocuje samotná trieda a na základe toho vráti existujúcu alebo vytvorí novú inštanciu. Unikát je teda charakterizovaný:

- najviac jednou inštanciou svojej triedy,
- globálnym bodom prístupu k inštancii.

1.1.2 Štruktúra

Unikát je návrhový vzor, ktorý dáva požadované, preň charakteristické vlastnosti, konkrétnej triede, preto štrukturálne neovplyvňuje a nezahŕňa v sebe iné triedy, ale aplikuje sa iba na želanú triedu (resp. viaceré želané triedy v aplikácii).



Obrázok 1-1. Unikát (Singleton) – štruktúra vzoru.

1.1.3 Súčasti

`Unikát` (angl. *Singleton*) – samotná vzorová trieda reprezentujúca Unikát. Zabezpečuje všetku funkcionalitu s ohľadom na vytváranie jedinej inštancie tejto triedy (seba samej) prostredníctvom svojej statickej metódy.

1.1.4 Dôsledky

Unikát je vhodné použiť, ak:

- vlastníctvo inštancie sa nedá rozumne prideliť,
- pomalá (neskorá) inicializácia nie je nežiaduca – keďže sa inštancia vytvára až pri prvom vyžiadaní,
- globálny prístup nie je inak poskytovaný.

Použitie Unikátu so sebou prináša niekoľko dôsledkov:

- vzor Unikát umožňuje ľahký prístup k jedinečnej inštancii triedy pomocou statickej metódy a statického atribútu reprezentujúceho inštanciu triedy,
- správa vytvárania a sprístupňovania inštancie je zapuzdrená v jednej triede, čo umožňuje ľahšiu manipuláciu s ňou a dáva možnosť jej zamieňania pri dodržaní stanoveného rozhrania,
- vytváranie inštancie a jedinečnosť zabezpečuje trieda sama.

Pri testovaní s použitím vzoru Unikát, napr. pomocou testov jednotiek (angl. *UnitTests*), treba pamätať na to, že jednotlivé metódy pri testovaní nevytvárajú vždy nový objekt, ale pri získavaní nového dostanú ten, ktorý sa použil v minulom teste. Toto treba brať na vedomie a podľa toho formulovať testy.

1.1.5 Implementácia

Základom implementácie je ukrytie konštruktora ako súkromnej metódy triedy. Taktiež treba ukryť kopírovací konštruktor aj operátor priradenia. Tým sa znemožní nekontrolované vytváranie inštancií používateľom.

Na získanie inštancie musí používateľ volať statickú metódu `getInstance`, ktorá umožňuje kontrolu nad vytváraním inštancií, pretože iba ona volá konštruktor triedy. Táto metóda najskôr skontroluje, či už bola vytvorená inštancia, ktorá je v tom prípade reprezentovaná ako súkromný atribút triedy. Ak táto inštancia neexistuje, vytvorí sa, uloží a metóda vráti referenciu na ňu. Inak sa priamo vracia táto referencia.

Tento prístup predstavuje pomalú inicializáciu (angl. *lazy initialization*), pretože inštancia sa vytvára až pri prvej požiadavke na jej získanie.

1.1.6 Príklad

Príkladom využitia aplikácie môže byť aj spomínaný vlastný denník aplikácie (angl. *log*). Schematický náčrt jeho implementácie poskytuje príklad 1-1.

```
public class MyLog {
  private static MyLog* log;

  public static getInstance() {
    if (log==NULL) {
      log=new MyLog();
      //inicializacia a otvorenie suboru
    }
    return log;
  }

  public int write(char * message) {
    ...
  }

  ...  //dalsie metody pre pracu s dennikom

  MyLog::log=NULL;
```

Príklad 1-1. Zdrojový kód vzoru Unikát.

Možným rozšírením založeným na vzore Unikát (pričom sa návrh do istej miery odkláňa od pôvodnej myšlienky vzoru) je vytvorenie triedy s maximálnym počtom inštancií vyšším ako 1. Takáto trieda predstavuje akýsi všeobecnejší typ. Unikát by predstavoval jeho špecializáciu pre maximálny počet inštancií = 1.

1.1.7 Príbuzné vzory

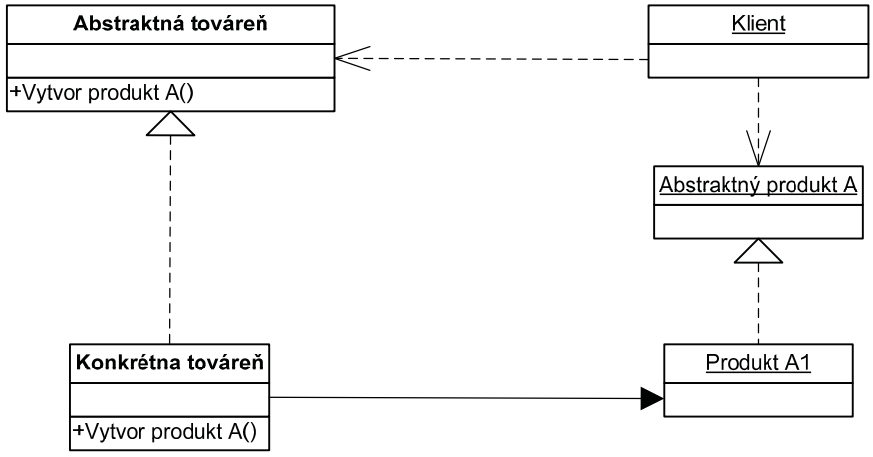
Vzor Unikát sa môže použiť aj pri implementácii iných vzorov, napr. abstraktná továreň (angl. *Abstract factory*), staviteľ (angl. *Builder*), prototyp (angl. *Prototype*), fasáda (angl. *Facade*), stav (angl. *State*).

1.2 Abstraktná továreň

V prehľadovej časti tejto kapitoly sme sa zaoberali existenciou a použitím návrhových vzorov. Môžeme povedať, že základom takéhoto vzoru je znovupoužiteľné riešenie pre opakujúci sa problém. Jedným z veľmi často používaných návrhových vzorov je Abstraktná továreň *(angl. Abstract Factory)*. Nasledujúca časť je venovaná použitiu a možným implementáciám Abstraktnej továrne.

Ako už sme spomenuli v úvodnej časti, návrhové vzory vytvárania predstavujú zovšeobecnenie tvorby objektov. Jedným zo vzorov vytvárania je tiež Abstraktná továreň. Vzory vytvárania zjednodušujú proces vytvárania objektov tým, že zakrývajú konkrétnu implementáciu vytváraných objektov a spôsob, akým sú tieto objekty navzájom pospájané.

Aplikácia má k dispozícii len rozhrania tried, ktoré môže používať, ako to je znázornené na obrázku 1-2. Konkrétna implementácia a samotný spôsob vytvárania objektov ostáva skrytý. Tento fakt dáva riešeniu problému vytvárania objektov veľkú flexibilitu a aplikácia sa nemusí starať o to, čo sa vytvorí, kedy sa to vytvorí a ani ako sa to vytvorí. O tieto aktivity sa postará abstraktná továreň. Aplikácia používa len definované rozhrania.



Obrázok 1-2. Abstraktná továreň.

1.2.1 Príklady použitia Abstraktnej továrne

Môžeme identifikovať viacero možností aplikovania vzoru Abstraktná továreň.

Príklad z reálneho sveta

Abstraktnú továreň možno pripodobniť stroju na výrobu cestovín. Použitím tohto stroja môžeme vytvoriť cestoviny rôznych tvarov v závislosti od nadstavca, ktorý je na konci stroja. Do stroja vstupuje vždy rovnaká hmota, ale zo stroja vychádza cestovina požadovaného tvaru. Stroj má teda úlohu abstraktnej továrne a nadstavec má úlohu konkrétnej továrne.

Grafické prvky pre viaceré operačné systémy

Ďalším príkladom Abstraktnej továrne je trieda `WindowFactory` s metódou napríklad `CreateButton`. WindowFactory je abstraktná továreň. Jednoduchým príkladom použitia vzoru Abstraktná továreň môže byť prenosná klientska aplikácia, ktorá chce používať viaceré grafické prvky operačného systému.

Ovládače pre rôzne databázové systémy

Ďalším príkladom použitia Abstraktnej továrne je pripojenie na rôzne databázové systémy. Môžeme si predstaviť situáciu, kedy používateľ používa bežné príkazy na komunikáciu s databázou, avšak pracuje naraz s viacerými druhmi databázových systémov.

Vzor Abstraktná továreň môže byť použitý aj v prípade, že chceme oddeliť aplikáciu od úložiska dát (napr. relačná databáza), ktoré táto aplikácia používa. Toto oddelenie aplikácie od databázy umožňuje zmeniť relačnú databázu bez potreby zmien v aplikácii. Môžeme si predstaviť, že aplikácia používa niekoľko základných operácií na prístup do databázy: DBTransaction, DBQuery a DBReport.

Implementácia týchto operácií je pre rôzne databázové systémy odlišná, pričom funkcionalita týchto operácií ostáva v princípe rovnaká. Takže potrebuje sady jednotlivých operácií pre rôzne databázové systémy. Pre konkrétnu operáciu určenú pre konkrétny systém potrebujeme vygenerovať samostatné SQL príkazy a tiež ďalšie parametre, ktoré možno použiť pri ladení výkonnosti jednotlivých operácií.

Existuje viacero možností riešenia. Môžeme každú operáciu implementovať zvlášť pre každý databázový systém. Príslušný kód, ktorý bude vykonaný sa vyberie v čase kompilácie pomocou konštrukcie *switch*. Je však potrebné pri každej zmene rekompilovať kód aplikácie. Ďalšou možnosťou by bolo dynamické vytváranie SQL príkazov, ale v tomto prípade by bolo potrebné udržiavať všetky verzie programu v čase jeho vykonávania.

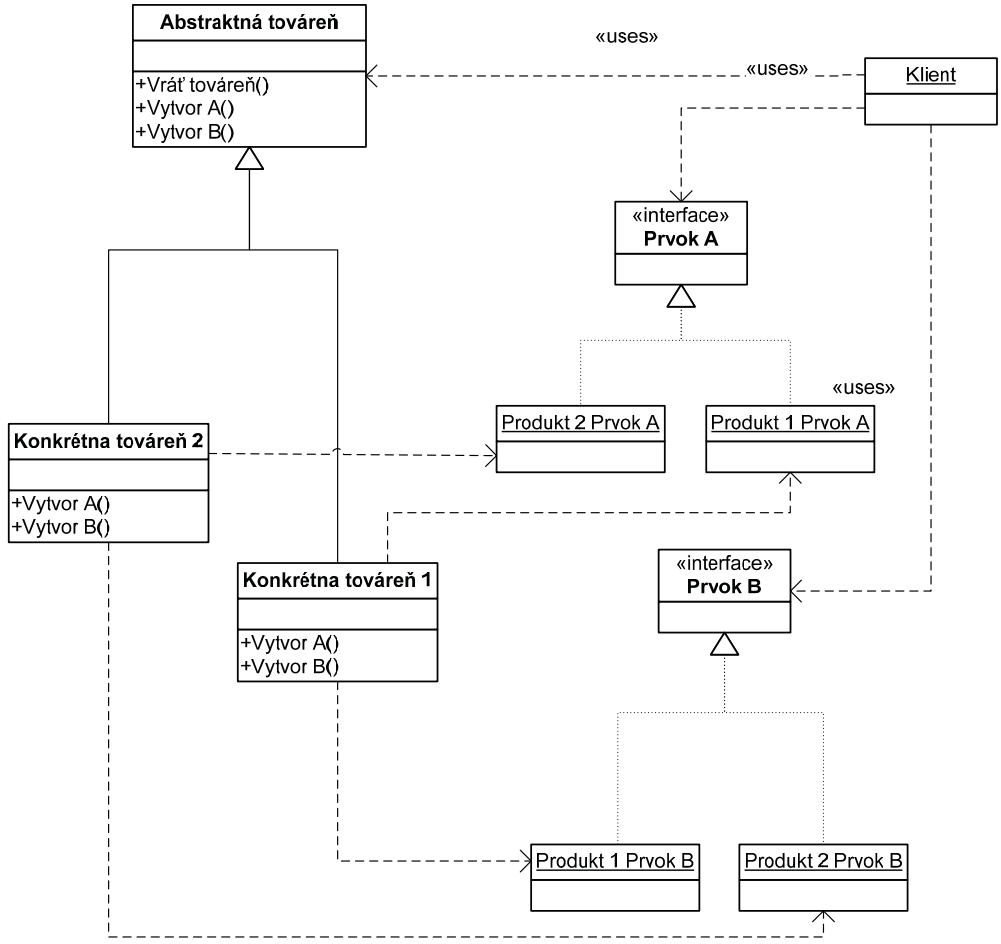
Za najvhodnejšie je však možné považovať riešenie, kde bude použitý vzor Abstraktná továreň. Aplikácia požiada abstraktnú továreň o sprístupnenie objektu, ktorý bude odpovedať zvolenej databázovej technológii. Aplikácia bude používať funkcionalitu databázy pomocou definovaného rozhrania.

1.2.2 Použitie vzoru

Z príkladov uvedených v predchádzajúcej časti možno vytušiť účel použitia vzoru Abstraktná továreň. Abstraktnú továreň použijeme v prípade, ak chceme vytvárať skupiny súvisiacich objektov a pritom nechceme špecifikovať konkrétne triedy vytváraných objektov. Taká situácia môže nastať vo viacerých prípadoch:

- Systém má mať možnosť používať jednu zo skupiny súvisiacich produktov na základe konfigurácie (grafické prvky pre určitý operačný systém).
- Súvisiace objekty sú navrhnuté tak, aby boli používané spolu (zväčša potrebujeme viacero grafických prvkov).
- Treba použiť knižnicu tried bez odhalenia konkrétnej implementácie (prístupné budú len rozhrania).

Abstraktná továreň sa použije na vytvorenie konkrétnych tovární. Všeobecný príklad použitia vzoru Abstraktná továreň opisujúci jeho podstatu je uvedený na obrázku 1-3.



Obrázok 1-3. Všeobecný príklad použitia vzoru Abstraktná továreň – poďla (Grand, 2008).

Podobný diagram uvádzajú viacerí autori. Vysvetlíme si podstatu jednotlivých prvkov:

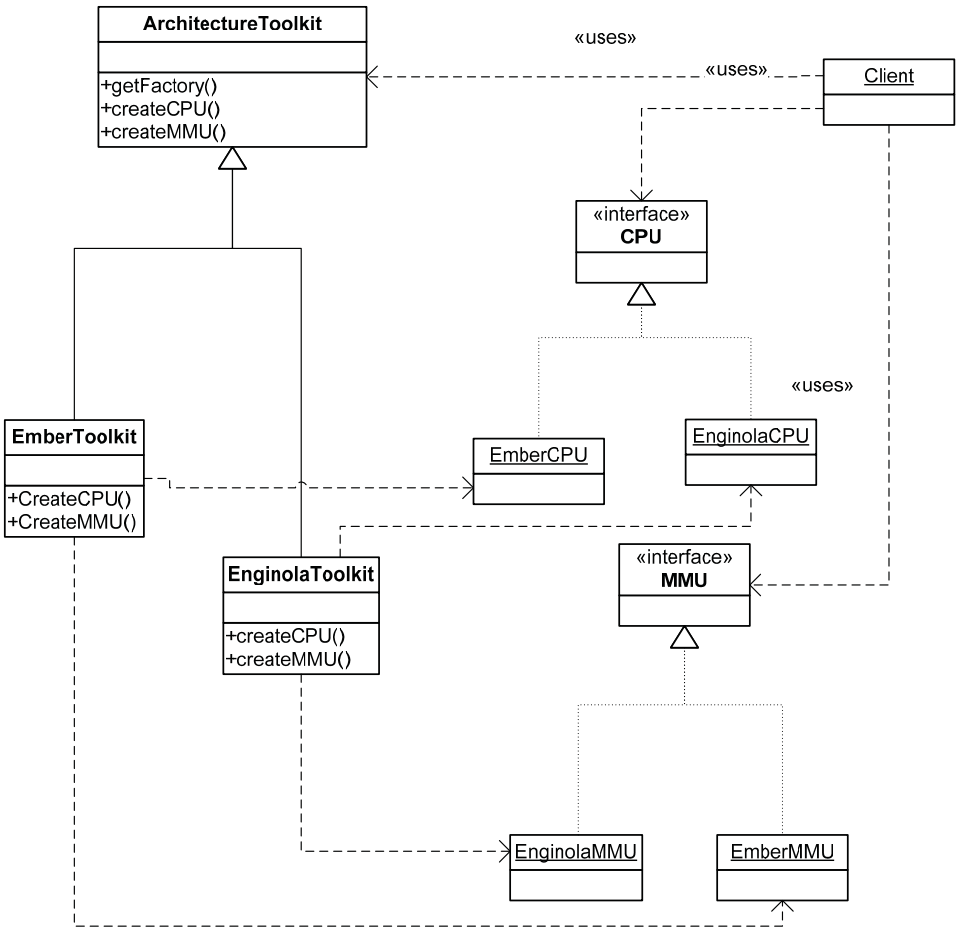
- Klient – má sprístupnené len abstraktné rozhrania. Triedy implementované na strane klienta používajú rôzne produkty, ale poznajú len ich abstraktné rozhrania a nemajú žiadnu informáciu o implementáciách týchto tried.
- Abstraktná továreň – je zodpovedná za vytváranie konkrétnych tovární a obsahuje abstraktné definície metód pre vytváranie súvisiacich produktov.

- Konkrétna továreň – zodpovedá za vytváranie konkrétnych objektov, reprezentuje konkrétne implementácie Abstraktnej továrne.
- Abstraktná trieda alebo tiež rozhranie (vo vyššieuvedenom diagrame Prvok A, Prvok B) definuje základné vlastnosti vytváraných produktov v podobe svojich metód.
- Konkrétna trieda (Produkt 1 Prvok A, Produkt 2 Prvok A, ...) – predstavuje implementáciu rozhrania. Klient k implementácii tejto triedy nemá prístup.

1.2.3 Implementácia

V časti implementácia sa budeme venovať prezentácii niekoľkých jednoduchých príkladov, ktoré predstavujú implementáciu jednotlivých častí podieľajúcich sa na vzore Abstraktná továreň. Jednotlivé príklady zdrojového kódu sa vzťahujú k obrázku 1-3.

Na obrázku 1-4 je zobrazený príklad použitia Abstraktnej továrne, ktorá je aplikovaná v aplikácii pre testovanie hardvérových súčiastok.



Obrázok 1-4. Konkrétny príklad Abstraktnej továrne.

Klient testuje `CPU` a `MMU` jednotky s dvoma rôznymi inštrukčnými sadami – *Ember* a *Enginola*. Nechceme, aby sa klient musel zaoberať detailami implementácie testovacích

tried pre jednotlivé inštrukčné sady. Preto má klient k depozícii len abstraktné rozhranie, ktoré mu umožňuje testovať `CPU` aj `MMU`. O výber konkrétnej sady požiada abstraktnú továreň (pozri príklad 1-2).

```
public class Client {
  public void doIt () {
    AbstractFactory af;
    af = AbstractFactory.getFactory(AbstractFactory.EMBER);
    CPU cpu = af.createCPU();
    ...
  } // doIt
} // class Client
```

Príklad 1-2. Zdrojový kód aplikácie na strane klienta.

Zdrojový kód Abstraktnej továrne opisuje funkcionalitu tejto súčiastky. Abstraktná továreň vytvorí inštancie konkrétnych tovární, v našom prípade ide o konkrétnu továreň pre inštrukčnú sadu počítača *Ember – EmberToolkit* a inštrukčnú sadu pre počítač *Enginola – EnginolaToolkit*. Abstraktná továreň obsahuje metódu `getFactory`. Táto metóda vracia referenciu na konkrétnu továreň. Z pohľadu klienta je `getFactory` metóda, ktorá mu umožňuje zvoliť si typ inštrukčnej sady, ktorú chce testovať. Okrem metódy `getFactory` abstraktná továreň obsahuje definície metód, ktoré budú implementované konkrétnymi továrňami.

```
public abstract class ArchitectureToolkit {
  private static final
   EmberToolkit emberToolkit = new EmberToolkit();
  private static final
   EnginolaToolkit enginolaToolkit = new EnginolaToolkit();
   ...

  static final ArchitectureToolkit getFactory(int architecture) {
    switch (architecture) {
      case ENGINOLA: return enginolaToolkit;
      case EMBER: return emberToolkit;
      ...
    }
  } // getFactory()

  public abstract CPU createCPU() ;
  public abstract MMU createMMU() ;
  ...

} // AbstractFactory
```

Príklad 1-3. Zdrojový kód Abstraktnej továrne. – podľa (Grand, 2008)

Konkrétna továreň predstavuje implementáciu rozhraní, ktoré boli definované v abstraktnej továrni. V tomto prípade ide o metódy `createCPU` a `createMMU`. Metódy sú určené na vytváranie testovacích tried pre jednotlivé inštrukčné sady. Klient nemá o spôsobe vytvárania týchto tried priamu vedomosť.

```
class EmberToolkit extends ArchitectureToolkit {
  public CPU createCPU() {
    return new EmberCPU();
  } // createCPU()
```

```
  public MMU createMMU() {
    return new EmberMMU();
  } // createMMU()
  ...

} // class EmberFactory
```

Príklad 1-4. Zdrojový kód konkrétnej továrne.

Posledná časť zdrojového kódu predstavuje implementáciu tried. Tieto triedy sú implementáciou rozhraní `CPU` a `MMU`, ktoré má klient k dispozícii. Klient nepozná implementáciu tried, ktoré implementujú rozhrania `CPU` a `MMU`. Klient nemôže priamo určiť, s ktorou triedou bude pracovať. Postará sa o to abstraktná továreň.

```
public abstract class CPU {
    ...
} // class CPU

class EmberCPU extends CPU {
    ...
} // class EmberCPU
```

Príklad 1-5. Zdrojový kód abstraktného rozhrania.

Uvedené ukážky zdrojového kódu sú len jednou z možností, ako možno implementovať vzor Abstraktná továreň.

1.2.4 Zhrnutie použitia Abstraktnej továrne

Abstraktná továreň je jeden z najviac používaných návrhových vzorov najmä pre jeho rozširovateľnosť a všeobecnosť.

Výhody

Abstraktná továreň umožňuje klientovi abstrahovať od implementácie jednotlivých tried. Na strane klienta nie je potrebné sa zaoberať tým, ako je daný produkt implementovaný, klient má k dispozícii len abstraktné rozhrania. Klient sa nemusí starať o to, ako sa objekty vytvárajú, a ani kedy sa vytvárajú.

Ďalšou výhodou je rozširovateľnosť, čo znamená, že keď klient potrebuje nový druh produktu, možno na strane abstraktnej továrne pridať ďalšiu konkrétnu továreň.

Nevýhody

Pri pridávaní ďalších produktov je nutné pridať metódy pre vytváranie pre všetky konkrétne továrne. V našom príklade mal klient k dispozícii dva produkty – testovanie `CPU` a testovanie `MMU`. V prípade, že by chcel pridať ďalší produkt na testovanie, napr. grafický koprocesor, bolo by potrebné implementovať tento produkt vo všetkých konkrétnych továrňach. A to i v prípade, že klient by nechcel používať všetky produkty v rámci každej sady.

Ďalšou nevýhodou je vytváranie konkrétnych tovární pre všetky sady v abstraktnej továrni. Klient testoval dve inštrukčné sady – *Ember a Enginola*. Konkrétne továrne pre obe sady boli vytvorené priamo pri vytvorení abstraktnej továrne. Táto skutočnosť má

za následok vyššie pamäťové nároky v prípade používania vzoru abstraktnej továrne. Riešenie tohto problému dáva návrhový vzor Prototyp.

Vzor Prototyp ako alternatíva k Abstraktnej továrni

Abstraktná továreň je veľmi podobná vzoru Prototyp. Prototyp je vzor, kde sa nové objekty vytvárajú kopírovaním prototypového objektu. Môžeme povedať, že prototyp sa dá použiť všade, kde možno použiť abstraktnú továreň. Vzor Prototyp poskytuje väčšiu flexibilitu za cenu nižšieho výkonu. Prototyp má však nižšie pamäťové nároky, vytvára totiž menej abstraktných tried ako abstraktná továreň.

1.3 Obaľovač

Pri vývoji softvérových systémov sa často stretávame s požiadavkou na vzájomnú spoluprácu súčiastok, ktoré boli vyvinuté nezávisle od seba. Napriek tomu, že z pohľadu požadovanej a poskytovanej funkčnosti by takéto súčiastky mohli spolupracovať, nekompatibilita rozhraní vo väčšine prípadov ich spolupráci zabraňuje.

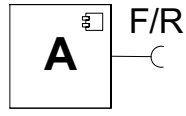
Vzhľadom na to, že zmena jednej alebo oboch súčiastok, ktorá by kompatibilitu rozhraní zaistila, nie je niekedy možná alebo vhodná, ako riešenie sa často používa softvérový vzor Obaľovač (angl. *Wrapper*).

1.3.1 Definícia

Gamma a kol. (Gamma, 1995) nazvali tento vzor adaptérom[1] (angl. *Adapter*), pričom ho definujú takto: „Adaptér konvertuje rozhranie triedy na iné, očakávané klientmi. Takto umožňuje spolupracovať aj triedam, ktoré by pre nekompatibilné rozhrania inak spolupracovať nemohli.“ (s. 139)

Pokúsme sa teraz o mierne zovšeobecnenie tejto definície, najmä s ohľadom na to, že spolupracujúce súčiastky nemusia byť nutne triedami. Majme teda softvérovú entitu `A`. Môže ísť o súčiastku v užšom zmysle slova, ako ju definuje napr. (Szyperski, 2002) – teda typicky o procedúru, triedu, balík (angl. *package*), knižnicu alebo aplikáciu – alebo o iný typ softvérovej entity, ako je napríklad objekt, informačný systém, (webová) služba a podobne. Pre jednoduchosť vyjadrovania pripusťme teda istú nepresnosť a hovorme zjednodušene o (softvérovej) súčiastke `A`.

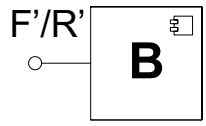
Súčiastka `A` potrebuje použiť funkčnosť `F`. Očakáva pritom, že ju má k dispozícii prostredníctvom rozhrania `R` (obrázok 1-5).



Obrázok 1-5. Súčiastka A očakávajúca funkčnosť F prostredníctvom rozhrania R.

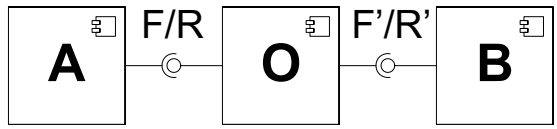
Častým javom je, že súčiastku poskytujúcu funkčnosť `F` prostredníctvom rozhrania `R` nemáme k dispozícii; máme však súčastku `B` poskytujúcu funkčnosť `F'`, totožnú s `F` alebo jej podobnú, prostredníctvom rozhrania `R'` (obrázok 1-6).

[1] Poznámky k tejto terminologickej odlišnosti (Obaľovač vs. Adaptér) sú uvedené v časti 1.3.3 nižšie.



Obrázok 1-6. Súčiastka B poskytujúca funkčnosť F' prostredníctvom rozhrania R'.

Prirodzeným riešením tohto problému je vytvorenie súčiastky `O` poskytujúcej funkčnosť `F` prostredníctvom rozhrania `R`, ktorá bude využívať funkčnosť `F'` poskytovanú súčiastkou `B` prostredníctvom rozhrania `R'`. `O` sa nazýva obaľovačom. Situácia je znázornená na obrázku 1-7.



Obrázok 1-7. Použitie obaľovača O na zaistenie spolupráce nekompatibilných súčiastok A, B.

Obaľovač sa používa v dvoch typických situáciách:

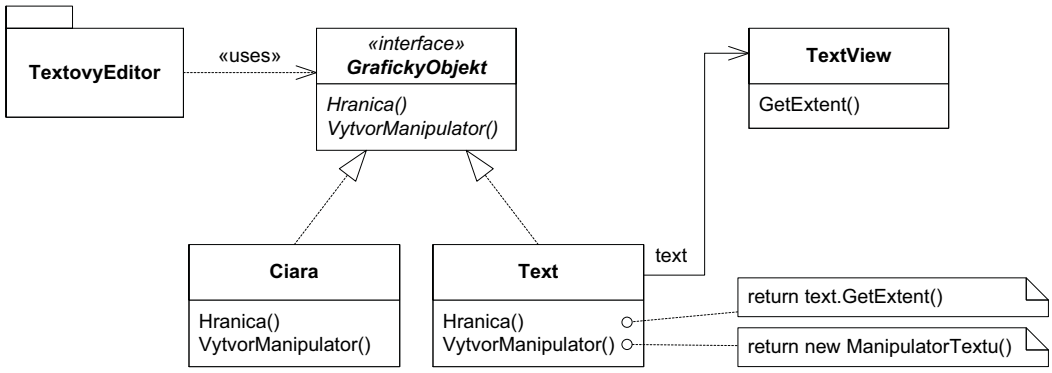
1. Súčiastky `A` a `B` už existujú a potrebujeme, aby spolupracovali, pričom ich nemôžem (resp. to nie je vhodné) meniť.
2. Súčiastku `A` vytvárame. Súčiastka `B` už existuje, avšak jej rozhranie `R'` nám nevyhovuje. Dôvodom môže byť napríklad nízka úroveň abstrakcie, na ktorej `R'` pracuje alebo technologická odlišnosť `R'` od `A`.

1.3.2 Vzor Obaľovač v rôznych kontextoch

Pozrime sa teraz na to, ako sa myšlienka vzoru Obaľovač vyskytuje vo vybraných publikáciách venovaných softvérovým vzorom.

V (Gamma, 1995) je opísané použitie tohto vzoru v kontexte objektovo orientovanej paradigme vývoja softvéru. Na ilustráciu uveďme príklad prevzatý z tejto publikácie, mierne upravený: Predstavme si, že vytvárame grafický editor. Jeho kľúčovou abstrakciou je grafický objekt, pre ktorý definujeme rozhranie `GrafickyObjekt`. Toto rozhranie je implementované rôznymi triedami: `Ciara`, `Polygon`, `Kruznica` a ďalšími, slúžiacimi na vykreslenie a editáciu čiar, mnohouholníkov, kružníc a iných objektov. Ďalším grafickým objektom, ktorý chceme mať možnosť editovať, je text.

Funkčnosť týkajúca sa editácie textu je typicky implementovaná priamo v používanej grafickej knižnici – v našom prípade nech na tento účel slúži trieda `TextView`. Ideálne by bolo, keby sme existujúcu triedu `TextView` mohli v grafickom editore priamo použiť, avšak z pochopiteľných dôvodov toto nie je možné. Totižto napriek tomu, že poskytuje potrebnú funkčnosť, má iné rozhranie – napríklad na zistenie hranice poskytuje metódu `GetExtent()`, kým `GrafickyObjekt` na tento účel predpisuje metódu `Hranica()` (obrázok 1-8). Aby sme `TextView` mohli použiť, vytvoríme adaptér: triedu `Text` poskytujúcu rozhranie `GrafickyObjekt`, využívajúcu pritom triedu `TextView`.



Obrázok 1-8. Príklad použitia vzoru Adaptér – adaptované z (Gamma, 1995).

Ak by sme mali túto situáciu opísať v symboloch zavedených vyššie, potom `A` je `TextovyEditor`, `R` je `GrafickyObjekt`, `F` je očakávaná funkčnosť týkajúca sa zobrazenia a editácie objektu. Na druhej strane `B` je existujúca trieda `TextView`, `R'` je jej rozhranie, `F'` je jej funkčnosť. Poznamenajme, že `F` a `F'` sa zhodujú len čiastočne: napr. `TextView` neposkytuje možnosti posúvania textu po obrazovke. Na tento účel musí byť preto použitá pomocná trieda `ManipulatorTextu` (na obrázku nie sú jej detaily zobrazené).

V ďalšej knihe, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects* (Schmidt, 2000), venovanej softvérovým vzorom v oblasti sieťových a paralelných aplikácií, je definovaný vzor Obaľujúca fasáda (angl. *Wrapper Facade*), ktorý slúži na „zabalenie funkcií a údajov poskytovaných existujúcimi nie-objektovo orientovanými aplikačnými programátorskými rozhraniami (API) do koncíznejších, robustnejších, portabilnejších, udržiavateľnejších a súdržnejších objektovo orientovaných rozhraní“ (s. 48). Tento vzor je v istom zmysle zúžením myšlienky Adaptéra podľa GoF, nakoľko sa špecializuje na sprístupnenie nie-objektovo orientovaných API vhodnou formou. Zodpovedá nami opísanej situácii č. 2, kedy rozhranie `R'` existujúcej súčiastky `B` nevyhovuje z dôvodu nízkej úrovne abstrakcie. Autormi uvádzané príklady použitia sa týkajú rozhraní na sieťovú komunikáciu, prácu s vláknami a s mechanizmami vzájomného vylúčenia v prostrediach so súbežným vykonávaním úloh.

V knihe *Patterns of Enterprise Application Architecture* (Fowler, 2002) venovanej vzorom používaným pri vývoji podnikových aplikácií sa opisuje vzor Brána (angl. *Gateway*) slúžiaci na sprístupnenie zdroja externého vzhľadom na vyvíjanú aplikáciu. Takýmto zdrojom môže byť relačná databáza, partnerská aplikácia, externé údaje (napr. v XML) a podobne. Externé zdroje sú typicky sprístupňované prostredníctvom špecializovaného aplikačného programátorského rozhrania, často pomerne komplikovaného. Z dôvodov jednoduchosti vyvíjanej aplikácie a jej flexibility (vzhľadom k možnej výmene externého zdroja, prípadne zmene jeho rozhrania) Fowler odporúča obaliť toto API triedou s jednoduchým rozhraním vhodným pre vyvíjanú aplikáciu. Takto chápaný vzor je zúžením vzoru Adaptér (GoF) v tom zmysle, že brána sprístupňuje len zdroje externé vzhľadom k aplikácii, na druhej strane je jeho rozšírením, nakoľko API externého zdroja môže (ale nemusí) byť objektovo orientované.

Zvláštny význam má obaľovač pri integrácii aplikácií, nakoľko v tejto oblasti je odstraňovanie nekompatibilít medzi jednotlivými aplikáciami prakticky hlavnou úlohou. Pozrime sa preto na výskyt tohto vzoru v niektorých publikáciách venovaných tejto problematike.

Kniha *Integration Patterns* (Trowbridge, 2004) sa zaoberá architektonickými vzormi používanými pri integrácii podnikových aplikácií. Opisuje vzor Brána (angl. *Gateway*), ktorý slúži na zjednodušenie prístupu k externému systému pre množinu aplikácií, zvýšenie flexibility a bezpečnosti. Myšlienka je analogická rovnomennému vzoru vo (Fowler, 2002), avšak v tomto prípade realizáciou vzoru nie je trieda v rámci vyvíjanej aplikácie, ale samostatne nasadená aplikácia, resp. služba sprostredkúvajúca komunikáciu s externým systémom. So vzorom Adaptér (GoF) má spoločnú teda len myšlienku, nie implementáciu. Ako príklad použitia tohto vzoru uvádzajú autori bránu zaisťujúcu prístup k špecializovaným tlačovým službám pre rôzne aplikácie v poisťovacej spoločnosti: evidenciu zmlúv, likvidáciu poistných udalostí a účtovníctvo.

Kniha *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (Hohpe, 2004) je tiež venovaná integrácii podnikových aplikácií, na rozdiel od predchádzajúcej však špeciálne integrácii prostredníctvom výmeny správ (angl. *messaging*). Úlohu sprostredkovateľa komunikácie medzi nekompatibilnými aplikáciami tu hrá vzor Prekladač správ (angl. *Message Translator*) a jeho špecializácie. Tento vzor slúži na konverziu správ z formátu vysielajúcej strany na formát, ktorý očakáva príjemca. Rozdiely, ktoré prekladač vyrovnáva, sú na viacerých úrovniach:

1. na úrovni prenosového protokolu: TCP, HTTP, SOAP, Java Message Service a pod.,
2. na úrovni spôsobu reprezentácie údajov: XML, hodnoty oddelené čiarkou, záznamy s pevnou dĺžkou polí a podobne, prípadne použitie rôznych znakových sád,
3. na úrovni dátových typov: názvy polí, obory hodnôt, spôsoby kódovania,
4. na úrovni dátových štruktúr: vymedzenie entít a asociácií (napr. ich kardinality).

Tento vzor má s pôvodným adaptérom (GoF) spoločnú myšlienku – umožňuje spoluprácu dvoch súčiastok, ktoré kvôli nekompatibilným rozhraniam priamo komunikovať nemôžu. Technologicky ide o úplne odlišné použitia – „správy" v objektovo orientovaných systémoch majú podobu volaní metód objektov, kým správy, ako ich chápu Hohpe a Woolf, sú skutočnými správami prenášanými špecializovanými systémami (tieto systémy sa v angličtine nazývajú *Message Oriented Middleware* resp. *MOM*).

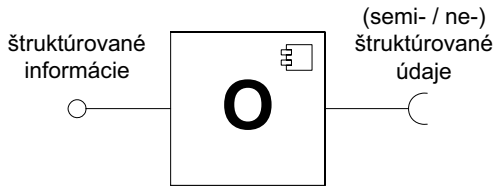
Ďalšie integračné vzory, ktoré majú myšlienku podobnú vzoru Obaľovač, sú Brána správ (angl. *Messaging Gateway*), ktorá slúži na oddelenie biznis logiky integrovaných aplikácií od kódu slúžiaceho na prácu s prostriedkami na prenos správ (MOM)[2], prípadne vzor Adaptér kanála (angl. *Channel Adapter*) poskytujúci to isté riešenie, s tým rozdielom, že kód na prístup k MOM je externý vzhľadom k integrovanej aplikácii.

Ako poznamenáva (Kaisler, 2005), myšlienka obaľovača sa používa tiež na rozšírenie funkčnosti existujúcich programov napísaných v tradičnom procedurálnom programovacom jazyku (typicky COBOL) kódom v niektorom z moderných objektovo orientovaných jazykov (napr. Java) – detaily opisuje napr. (Flint, 1997). Iným príkladom použitia sú rôzne

[2] T.j. ide o špeciálny prípad vzoru Brána z (Fowler, 2002), resp. v istom zmysle aj vzoru Obaľujúca fasáda z (Schmidt, 2000).

technologické brány. Napríklad v databázovom svete to môžu byť brány medzi databázami (napr. DB2 ↔ Oracle) alebo medzi prístupovými protokolmi (napr. ODBC ↔ JDBC).

Pojem Obaľovač sa používa ešte v jednom kontexte, a to v systémoch na extrakciu informácií z rôznych zdrojov. Konkrétne (Grlický, 2003) definuje obaľovač ako procedúru na extrahovanie informácií z určitého, napr. webového, zdroja, ktorá ako vstup berie odpoveď na dopyt, teda množinu neštruktúrovaných, resp. semištruktúrovaných informácií a na výstupe vracia množinu n-tíc opisujúcich informačný obsah tejto odpovede. Schematicky je takto definovaný obaľovač znázornený na obrázku 1-9.



Obrázok 1-9. Obaľovač v kontexte systémov na extrakciu informácií.

1.3.3 Abstraktnejší pohľad: vzor Rukoväť-Teleso

Vráťme sa teraz do prostredia objektovo orientovaných programov a pozrime sa na vzor Obaľovač (resp. Adaptér podľa GoF) abstraktnejším pohľadom. Tento vzor je súčasťou širšej množiny vzorov, označovanej niekedy ako vzor Rukoväť-Teleso (angl. *Handle-Body Pattern*)[3]. Charakteristickou črtou týchto vzorov je, že implementácia niečoho (teleso, angl. *body*) je oddelená od rozhrania, s ktorým pracujú klienti (rukoväť, angl. *handle*).

Volania, zachytené rukoväťou, sú štandardne poslané na spracovanie telesu, avšak môžu byť predtým upravené, prípadne aj úplne potlačené. Rukoväť teda istým spôsobom obaľuje teleso, preto sa vzoru Rukoväť-Teleso tiež niekedy hovorí Obaľovač.

Vzor Rukoväť-Teleso zahŕňa, okrem iných, nasledujúce vzory (názvy podľa GoF):

1. *Adaptér* (angl. *Adapter*), ktorý upravuje rozhranie triedy do požadovaného tvaru (povedané našou terminológiou, rieši situácie, kedy `R` ≠ `R'`, pričom `F` a `F'` môžu, avšak nemusia byť rovnaké).
2. *Dekoratér* (angl. *Decorator*), ktorý pridáva triede dodatočnú funkčnosť bez zmeny rozhrania (t.j. `F` ≠ `F'`, `R` = `R'`).
3. *Most* (angl. *Bridge*), ktorý umožňuje dynamicky meniť implementáciu realizujúcu dané rozhranie (t.j. `F` = `F'`, `R` = `R'`, avšak súčiastku `B` vieme počas vykonávania nahradiť súčiastkou `C` poskytujúcou funkčnosť `F` prostredníctvom rozhrania `R`).
4. *Zástupca* (angl. *Proxy*), ktorý modifikuje alebo dopĺňa niektoré vlastnosti triedy nie priamo súvisiace s jej funkčnosťou (ako sú vzdialený prístup, neskorá inicializácia, ochrana a iné) – t.j. mohli by sme povedať, že `F` = `F'`, `R` = `R'`, s tým, že menia sa nefunkčné aspekty poskytovanej služby.

---

[3] Portland Pattern Repository, `http://c2.com/cgi/wiki?HandleBodyPattern`
Z tejto stránky sú čerpané informácie uvedené v tejto časti.

So vzorom Rukoväť-Teleso sú tiež spojené isté problémy vo vzťahu k evolúcii obaľovaných súčiastok. Z priestorových dôvodov sa nimi na tomto mieste nebudeme zaoberať, viac informácií nájde čitateľ na stránkach Portland Pattern Repository[4].

Záverečná terminologická poznámka: Ako vidíme, pojem Obaľovač nie je celkom jasne vymedzený. V širšom zmysle ide o synonymum pre všeobecný vzor Rukoväť-Teleso. Podobne autori (Gamma, 1995) uvádzajú pojem Obaľovač ako synonymum pre dva konkrétne vzory: Adaptér a Dekoratér. V užšom zmysle ho možno použiť v súlade s našou definíciou v časti 1.3.1 na označenie súčiastky umožňujúcej spoluprácu iným dvom nekompatibilným súčiastkam – teda ako synonymum vzoru Adaptér podľa GoF. Ak neuvádzame inak, máme pod názvom Obaľovač na mysli práve tento posledný koncept.

1.3.4 Záver

Ako vidíme, Obaľovač sa vyskytuje v katalógoch vzorov, a teda aj v praktických systémoch, pomerne často. Podľa nášho názoru nejde o náhodný jav: dôvodom je potreba spolupráce nezávisle vyvinutých súčiastok, ktorá – ako očakávame – bude čím ďalej, tým výraznejšia.

1.4 Pozorovateľ

Návrhový vzor Pozorovateľ je v anglickom jazyku známy ako *Observer*.

1.4.1 Účel

Pozorovateľ umožňuje entitám sledovať zmeny iných entít. Definuje vzťah *1:m*, kde pri zmene jednej entity je ostatných *m* automaticky notifikovaných a aktualizovaných. V tomto kontexte rozlišujeme dva typy entít:

- predmet pozorovania alebo tiež subjekt (angl. *observable*, *subject*),
- pozorovateľ (angl. *observer*).

Predmet pozorovania je entita, zmeny ktorej pozorovateľ sleduje. Predmet pozorovania pozná všetkých svojich pozorovateľov. Poskytuje rozhranie umožňujúce notifikovať pozorovateľov v prípade výskytu udalosti. Notifikácia je realizovaná automaticky ako odpoveď na zmenu stavu entity. Úlohou pozorovateľa je realizácia logiky zmien súvisiacich so zmenou pozorovanej entity.

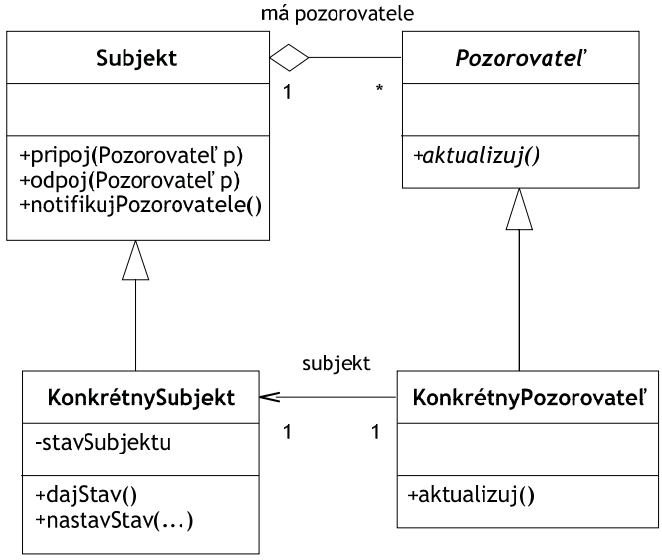
Typickým prípadom použitia vzoru sú udalostné systémy, napr. interakcia v grafickom používateľskom rozhraní (GUI). Uvažujme jednoduchú kalkulačku. Zadávanie údajov pomocou tlačidiel vedie k zmenám obsahu jej displeja. Na zmenu stavu entity `Tlačidlo` zareaguje pozorovateľ `Displej` aktualizáciou výstupu. Štruktúra vzoru je znázornená na obrázku 1-10.

1.4.2 Súčasti

Vzor Pozorovateľ má nasledujúce súčasti:

- `Pozorovateľ` – deklaruje rozhranie, pomocou ktorého bude implementovaná logika reakcie na zmenu stavu predmetu pozorovania,

[4] `http://www.c2.com/cgi/wiki?HandleBodyPatternProblem`



Obrázok 1-10. Štruktúra vzoru Pozorovateľ.

- `KonkrétnyPozorovateľ` – implementácia rozhrania `Pozorovateľ`. Metóda `aktualizuj()` implementuje vyžadovanú reakciu na zmenu stavu predmetu pozorovania,
- `Subjekt` – deklaruje rozhranie pre pripojenie a odpojenie entít `Pozorovateľ` a pre notifikáciu o zmene stavu predmetu pozorovania,
- `KonkrétnySubjekt` – rozširuje triedu `Subjekt`, čím dedí funkcionalitu pripájania a odpájania entít `Pozorovateľ`. Metódy, ktoré umožňujú zmenu vnútorného stavu objektu `KonkrétnySubjekt` (tzv. *mutator* alebo tiež „*setter*“ metódy), volajú metódu `notifikujPozorovatele()` pre notifikáciu pozorovateľov o zmene.

1.4.3 Dôsledky

Výsledkom použitia vzoru je nízka zviazanosť entít, pretože väzba medzi predmetom pozorovania a pozorovateľom je zredukovaná len na zoznam pozorovateľov, ktoré predmet pozorovania pozná. To vedie k znovupoužiteľnosti predmetov pozorovania bez nutnosti znovupoužitia pozorovateľov.

1.4.4 Implementácia

Implementácia vzoru spočíva vo využití princípu tzv. *callback* funkcií. *Callback* funkcia je časť kódu, ktorá je odovzdaná jednou entitou druhej, aby bola vykonaná neskôr v prípade potreby.

Pri implementácii môžeme naraziť na niekoľko problémov, resp. potenciálnych prekážok. Je potrebné dať pozor na „prehnané“ používanie vzoru, ktoré môže vyústiť do nekontrolovateľných cyklov. Každý pozorovateľ môže byť zároveň predmetom pozorovania, čo môže viesť k nežiaducej zložitosti, a to aj napriek jednoduchej logike, ktoré od entít požadujeme.

Problémy môže spôsobiť aj nekonzistencia stavu entity pri odoslaní notifikácie o jeho zmene. Môže sa stať, že notifikácia sa odošle v čase, keď ešte nedošlo k úplnej zmene stavu, a nastane konflikt. Práve z tohto dôvodu býva metóda `notifikujPozorovatele()` niekedy zapuzdrená ako privátna metóda.

Nástrahou použitia vzoru je aj výkonnosť metódy `aktualizuj()`, do ktorej je sústredená celá logika reakcie na zmenu stavu predmetu pozorovania.

1.4.5 Príklad

Použitie vzoru demonštrujeme na príklade načítavania znakov z klávesnice a následného výpisu na výstup. Ide o jeden z najjednoduchších príkladov použitia vzoru v prostredí jazyka Java, ktorý bol prevzatý z Wikipedie[5] a mierne upravený .

Trieda `EventSource` dedí od triedy `Observable`, ktorá zapuzdruje funkcionalitu predmetu pozorovania. Stav inštancie triedy `EventSource` sa zmení po zadaní znaku z klávesnice. Metódou `notifyObservers()` notifikujeme pozorovateľov o tejto zmene.

```
public class EventSource extends Observable implements Runnable {

  public void run() {
    try {
      final InputStreamReader isr =
          new InputStreamReader(System.in);
      final BufferedReader br = new BufferedReader(isr);

      while(true) {
        final String response = br.readLine();
        setChanged();
        notifyObservers(response);
      }
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

Príklad 1-6. Zdrojový kód triedy EventSource.

Metódou `update()` triedy `ResponseHandler` implementujeme už samotnú logiku zmeny súvisiacej so zmenou predmetu pozorovania. V našom príklade ide o výpis načítanej klávesy na výstup.

```
public class ResponseHandler implements Observer {
  private String resp;

  public void update (Observable obj, Object arg) {
    if (arg instanceof String) {
      resp = (String) arg;
      System.out.println("\nReceived: "+ resp );
    }
  }
}
```

Príklad 1-7. Zdrojový kód triedy ResponseHandler.

[5] http://en.wikipedia.org/wiki/Observer_pattern

Kľúčovým krokom je pripojenie pozorovateľa na subjekt pozorovania. Inštancii triedy `EventSource` ho pripojíme zavolaním zdedenej metódy `addObserver()`.

```
public class ObserverDemo {
  public static void main(String args[]) {
    System.out.println("Enter text >");

    final EventSource evSrc = new EventSource();
    final ResponseHandler respHandler = new ResponseHandler();

    evSrc.addObserver(respHandler);
    Thread thread = new Thread(evSrc);
    thread.start();
  }  }
```

Príklad 1-8. Zdrojový kód triedy ObserverDemo.

1.4.6 Príbuzné vzory

Návrhovému vzoru je sčasti príbuzný vzor Príkaz (angl. C*ommand*).

1.5 Rozhranie človek-počítač

Návrh používateľských rozhraní v rôznych počítačových systémoch v sebe zahŕňa riešenie podobných problémov, ako napr. registrácia nového zákazníka, výpis dlhého zoznamu položiek alebo komplikované ovládanie kvôli obmedzenej veľkosti displeja zariadenia. V roku 1988 Norman, ovplyvnený vývojom vzorov v architektúre, navrhuje použitie vzorov v procese návrhu používateľských rozhraní (Norman, 1988).

Vzory predstavujú spôsob ako zachytiť vhodné riešenia opakujúcich sa problémov. V súčasnosti sa vzory používajú pri návrhu rozhraní medzi človekom a počítačom (angl. *human-computer interface – HCI*) v desktopových aplikáciách, webových stránkach, mobilných telefónoch, ale aj v predtým netradičnejších kontextoch ako napr. palubné počítače v automobiloch, ovládanie chladničiek a pračiek.

1.5.1 Motivácia

Dobré prístupy pri návrhu rozhraní sú spomínané v mnohých publikáciách, avšak začínajúci návrhári majú ťažkosti si ich zapamätať a správne používať (Tidwell, 1999). Tidwell preto navrhuje izolovať kľúčovú funkcionalitu z používaných súčiastok a vytvoriť tzv. jazyk vzorov, teda kolekciu vzorov, ktorá by:

- umožňovala zachytiť kolektívnu múdrosť tak, aby ju mohli jednoducho použiť aj začínajúci návrhári,
- poskytovala jednotný jazyk pre komunikáciu s ďalšími návrhármi, vývojármi a zákazníkmi,
- upriamila pozornosť na kľúčové vlastnosti a funkcionalitu návrhu,
- zjednodušila prenosnosť vytvorených návrhov na ostatné platformy bez zmien alebo iných neželaných obmedzení funkčnosti.

Použitie vzorov pri návrhu HCI dáva vývojárom možnosť predpokladať vhodnosť použitého vzoru v danej situácii. Skúsení návrhári nepoužívajú stále rovnaké vzory, ale svojimi skúsenosťami vzory upravujú a kombinujú, avšak presný spôsob, ako bude rozhranie

používané, sa môže ukázať až pri testovaní použiteľnosti (angl. *usability testing*) s reálnymi ľuďmi. Testovanie použiteľnosti je náročné na čas, finančné prostriedky a ľudské zdroje (Nielsen, 1993), a preto je to relatívne nepraktická alternatívna pre vývojárov oproti použitiu overených vzorov, najmä počas vývoja softvéru.

1.5.2 Návrh používateľských rozhraní

Jeden spôsob ako zachytiť vzory pri návrhu HCI je použiť odporúčania výrobcu príslušného operačného systému. Príkladom sú *Apple Leopard User Experience Guidelines*[6] alebo *Windows User Experience Interaction Guidelines*[7]. Týmto sa zaručí jednotný vzhľad a funkcionalita na mikroúrovni (tvar tlačidiel, formuláre), avšak tieto odporúčania sa viažu na konkrétny systém a nie sú prenosné a rovnako použiteľné aj inde.

Pre lepšie abstrahovanie funkcionality v rozhraniach sa sústredíme na podstatu softvéru (Tidwell, 1999). Všetky efektívne softvérové nástroje zabezpečujú viac menej aspoň nasledujúce dve veci (Kaisler, 2005):

1. formujú používateľovo pochopenie obsahu pomocou štylizovanej prezentácie, vhodným spôsobom rozbaľujúc prezentovaný obsah,
2. umožňujú používateľovi vykonať úlohu postupným odhaľovaním možných akcií podľa aktuálneho stavu interakcie.

Tieto dimenzie sú skoro ortogonálne (Tidwell, 1999) a vzory HCI predstavujú rovnováhu medzi množstvom zobrazovaného obsahu a rozsahom ponúkaných akcií. Zvyšok tejto časti venujeme príkladom vzorov a ich rozdeleniu podľa (Tidwell, 2005) do kategórií, ktoré možno vnímať aj ako rozdelenie s dôrazom na obsah alebo akcie.

Organizácia obsahu

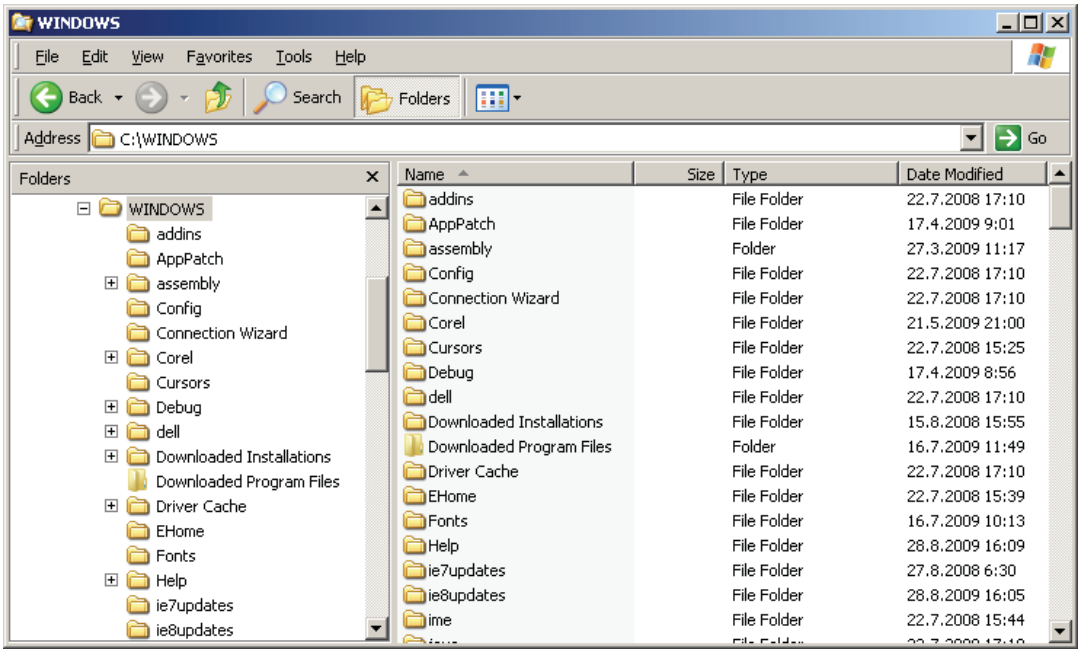
Existujú viaceré prístupy k zobrazeniu obsahu tak, aby bola jeho prezentácia pre používateľa zrozumiteľná. Vzor *Dvojitý panel* (na obrázku 1-11) je vhodný v prípade zoznamu objektov, kde každý prvok zoznamu obsahuje aj ďalší zaujímavý obsah, ktorý chceme zobraziť. Použijeme dva panely vedľa seba tak, že v jednom bude zoznam prvkov, v ktorom môže používateľ vybrať ľubovoľný prvok a v druhom sa potom pre prvok zvolený v prvom paneli zobrazí jeho obsah. Správy v schránke emailového klienta alebo fotografie v albume sú dobrí kandidáti pre tento vzor. Takéto použitie je zaužívané a používatelia rýchlo pochopia vzájomnú interakciu oboch panelov.

Vzor *Sprievodca* (angl. *Wizard*) je užitočný v prípade dlhej a komplikovanej úlohy. Vhodné je úlohu rozdeliť do menších ľahšie pochopiteľných častí a viesť používateľa v procese úlohy vo vopred určenom poradí. Očíslovanie jednotlivých menších úloh uľahčuje používateľovi orientáciu a znižuje riziko, že niektorú úlohu omylom preskočí.
Vzor *Viac na požiadanie* (obrázok 1-12) zjednodušuje prezentáciu väčšieho množstva obsahu, ktorý nie je všetok až tak dôležitý. Pri jeho použití sa najskôr zobrazí len základná funkcionalita a zvyšok funkcionality je dostupný jednoduchou akciou používateľa. Použitie tohto vzoru šetrí využiteľnú plochu v rozhraní a uľahčuje prácu väčšine používateľov, ktorí dodatočnú funkcionalitu zvyčajne nepotrebujú.

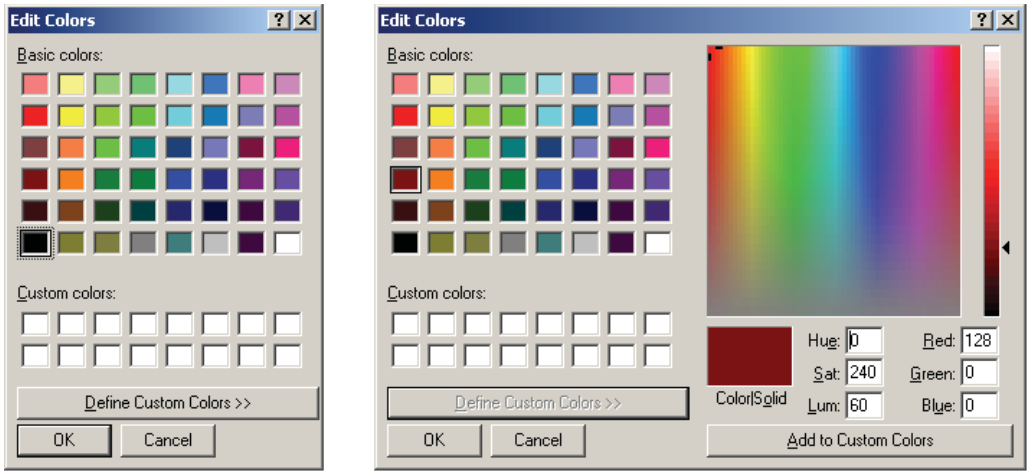
---

[6] `http://developer.apple.com/documentation/UserExperience/index.html`

[7] `http://msdn.microsoft.com/en-us/library/aa511258.aspx`



Obrázok 1-11. Dvojitý panel (angl. Two-Panel Selector).



Obrázok 1-12. Viac na požiadanie (angl. Extras On Demand).

Navigácia

V rozsiahlych aplikáciách je nutné zabezpečiť navigáciu medzi jednotlivými podčasťami alebo modulmi aplikácie. Vzor *Jasné vstupné body* (angl. *Clear Entry Points*) je vhodný pre aplikácie používané zvyčajne začínajúcimi používateľmi alebo používané len sporadicky, je vhodné na úvod zobraziť niekoľko jasných vstupných bodov do rozhrania. Vstupné body umožňujú používateľom rozhodnúť sa, čo budú s aplikáciou robiť, a mali by teda reprezentovať najčastejšie dôvody pre použitie aplikácie.

Vzor *Farebné sekcie* (obrázok 1-13) rôznymi farbami zvýrazňuje rozdielne podčasti rozsiahlejšej aplikácie alebo webovej stránky. Použitie rôznych farieb nielen skrášľuje navrhnuté rozhranie, ale aj uľahčuje používateľom orientáciu potom, ako si vžijú zvolené kombinácie farieb a podčasti aplikácie. Farboslepí používatelia však môžu mať problém orientovať sa výhradne podľa farieb a farby by preto nemali byť jediným rozlišovacím kritériom. Na zvýraznenie sa môže použiť aj tučné písmo.

Obrázok 1-13. Farebné sekcie (angl. Color-Coded Sections).

Organizácia stránky

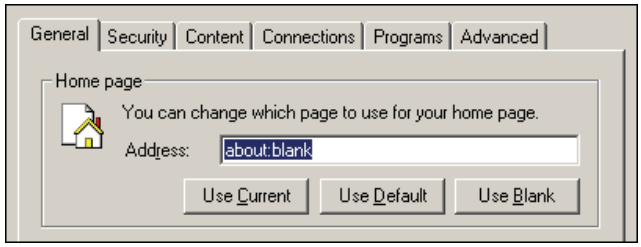
Celkový vzhľad okna aplikácie zvyčajne obsahuje viaceré funkčné prvky. Odporúča sa používať rovnaké rozmiestnenie prvkov rozhrania vo všetkých častiach aplikácie tak, aby pri vykonávaní rozličných typov úloh používateľ využil svoje znalosti z predchádzajúcich použití aplikácie.

Vzor *Hlavné javisko* (angl. *Center Stage*) umiestňuje najdôležitejšiu časť rozhrania do najväčšej časti okna alebo stránky a ostatné prvky niekde naokolo. Používa sa vtedy, keď hlavným účelom aplikácie je prezentácia koherentných informácií, editácia konkrétneho objetku (napr. textový dokument) alebo vykonanie nejakej dobre vymedzenej úlohy. Používateľova pozornosť sa upriami na najväčšiu (a teda najdôležitejšiu) časť aplikácie a následne používateľ vyhodnotí okolité prvky, ktoré sú určené pre prácu s hlavným obsahom. Hlavné javisko je zreteľne vymedzené svojou veľkosťou, farbou a nadpismi. Poloha umiestnenia nie je až taká dôležitá, a veľkosť hlavnej plochy by mala byť dostatočná nato, aby zaujala pozornosť používateľov.

Vzor *Kôpka kariet* (obrázok 1-14) je vhodný, keď máme na stránke veľké množstvo prvkov bez nejakej konkrétnej štruktúry, a nemôžeme ich preto umiestniť do stĺpcov alebo mriežky. Rozdelením prvkov do sekcií a umiestnením týchto sekcií do panelov (alebo kariet), ktoré naukladáme na seba tak, aby bol vždy viditeľný práve jeden panel nám umožňuje uľahčiť pochopenie prvkov a ušetriť miesto v rozhraní.

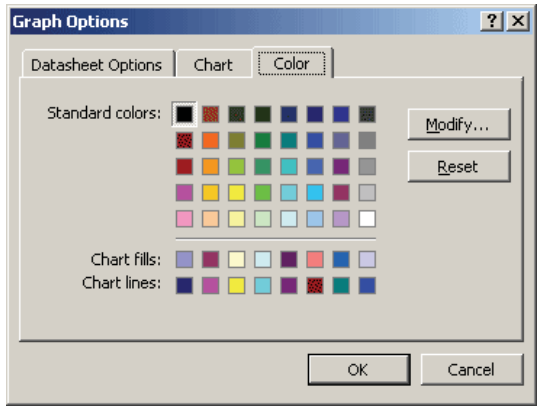
Rozloženie prvkov na stránke by malo spĺňať aj základné estetické kritériá. Najneskôr od čias renesancie sa zlatý rez[8] považuje za esteticky príjemný, a viaceré časti diel umenia a architektúry sú navrhnuté v pomere zlatého rezu (napríklad Akropolis v Aténach). Podobne, v profesionálnej fotografii sa hlavný motív často umiestňuje do tretiny výšky a šírky snímku.

[8] Hodnota podielu φ, keď $(a+b) / a = a / b = \varphi$



Obrázok 1-14. Kôpka kariet (angl. Card Stack).

Vzor *Uhlopriečková rovnováha* (obrázok 1-15) odporúča rozmiestniť prvky rozhrania do rovnováhy po hlavnej diagonále podľa prirodzeného smeru čítania. Tradičný smer v Európe a Amerike je zľava hore do pravej časti dole, existujú však používatelia z iných krajín, ktorý čítajú sprava doľava (napr. Japonsko).



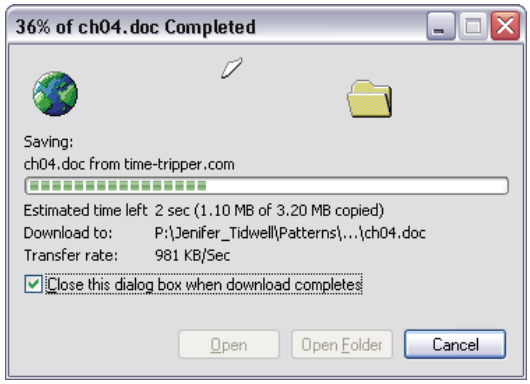
Obrázok 1-15. Uhlopriečková rovnováha (angl. Diagonal Balance).

Príkazy a akcie

Okrem prezentovania obsahu musia aplikácie vo svojom rozhraní vhodným spôsobom štruktúrovať dostupné príkazy a akcie, ktoré môžu používatelia s obsahom vykonávať.

Vzor *Panel akcií* (angl. *Action Panel*) namiesto použitia výsuvnej ponuky menu zobrazuje všetky akcie, ktoré možno použiť v zozname, ktorý je stále viditeľný. Výsuvné menu ukrýva jednotlivé akcie a používatelia si nemusia hneď uvedomiť, že vôbec existuje. Panely akcií sú funkčne totožné s menu, len nie sú výsuvné a používatelia nemusia vykonať žiadnu akciu, aby získali prehľad o všetkých dostupných akciách.

Vzor *Indikátor priebehu* (obrázok 1-16) zhromažďuje dôležité informácie o priebehu dlhotrvajúcej akcie iniciovanej používateľom ako napr. množstvo a detaily práve spracúvaných údajov, čas zostávajúci do skončenia a iné štatistiky. Používatelia sú pri používaní aplikácií zvyčajne netrpezliví a pri úlohách trvajúcich dlhšie ako cca 2 sekundy je vhodné umožniť používateľovi naplánovať si ďalšiu prácu odhadnutím zostávajúceho času spracovania.



Obrázok 1-16. Indikátor priebehu (angl. Progress Indicator).

Zobrazenie komplexných dát

Počítače dokážu vizualizovať obrázky, mapy, štatistiky a iné komplexné dáta. Spôsob prezentácie formuje používateľovo pochopenie zobrazovaných dát, a preto je dôležité podporiť používateľa pri práci a umožniť mu objaviť v dátach zaujímavé vlastnosti.

Vzor *Striedavé pásiky* (angl. *Row Striping*) používa rôzne pozadia riadkov v tabuľke striedavo pre párne a nepárne riadky. Tabuľky obsahujúce veľa stĺpcov nemusia byť ľahko čitateľné a striedavé pozadia riadkov vizuálne uľahčia vyhľadať údaje v konkrétnom riadku. Zvyčajne sa používajú farby s nízkou sýtosťou.

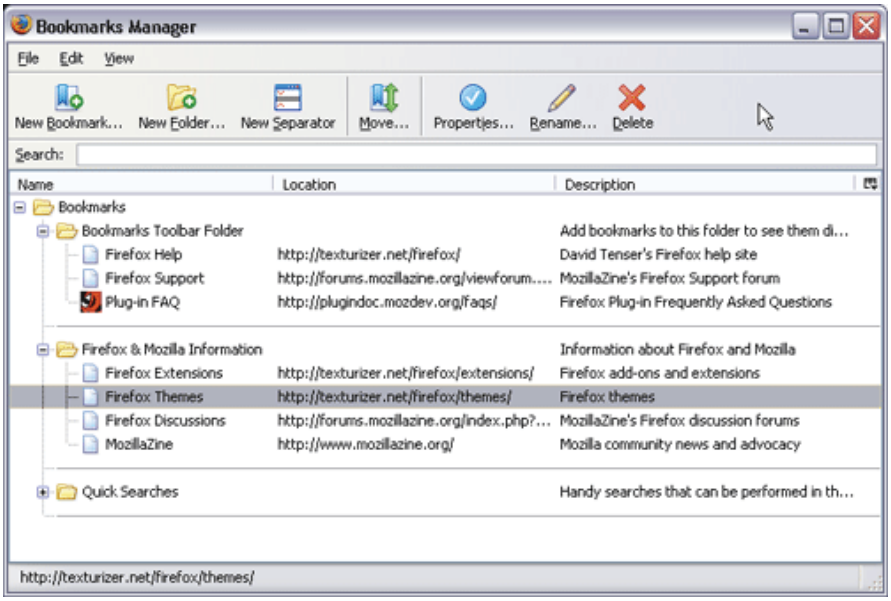
Vzor *Usporiadateľná tabuľka* (angl. *Sortable Table*) umožňuje používateľovi usporadúvať riadky tabuľky podľa hodnoty v konkrétnom stĺpci. Riadky sa usporiadajú po kliknutí na hlavičku stĺpca a vyobrazená šípka určuje smer usporadúvania; po opätovnom stlačení hlavičky stĺpca sa smer usporiadania otočí. Usporiadateľné tabuľky podporujú objavovanie nových zaujímavých vlastností v dátach.

Vzor *Stromová tabuľka* (obrázok 1-17) sa používa v prípade, keď môžeme dáta radiť hierarchicky. Teda, v jednej časti tabuľky používateľ prehliada hierarchickú (stromovú) štruktúru prvkov (riadkov), pričom v ostatných stĺpcoch sa nachádzajú zvyšné údaje pre príslušný prvok.

Získavanie vstupu od používateľov

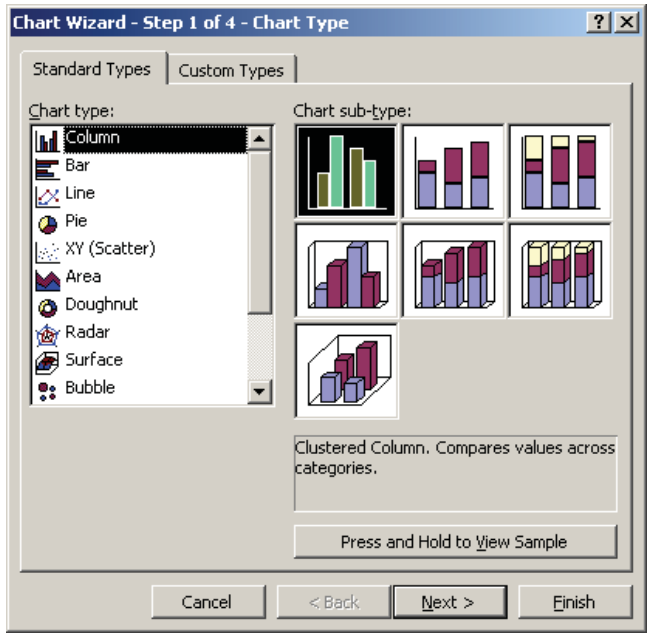
Používatelia nad obsahom vykonávajú akcie, ktoré zvyčajne vyžadujú zadanie nejakých údajov. Väčšinou sa vstupné dáta zbierajú použitím formulárov. Vhodný postup je navrhnúť políčka vo formulári tak, aby používateľ jednoznačne pochopil ich význam. Použitím vhodných popisiek, ukážok hodnôt a najčastejšie používaných predvyplnených hodnôt sa minimalizuje riziko vykonania chybnej operácie.

Vzor *Tolerantný formát* (angl. *Forgiving Format*) zjednodušuje pre používateľa vkladanie vstupnej hodnoty vo formáte preferovanom používateľom. Do políčka (napr. pre telefónne číslo) umožníme zadať aj neočakávané kombinácie znakov, ktoré sú potom analyzované na rôzne formáty a použije sa ten, ktorý vyhovuje. Použitie tohto vzoru vlastne prenáša problém z oblasti návrhu rozhraní do oblasti programovania.



Obrázok 1-17. Stromová tabuľka (angl. Tree Table).

Vzor *Ilustrované voľby* (obrázok 1-18) zjednodušuje výber hodnoty z ponuky pomocou obrázkov a dodatočných textov vysvetľujúcich jednotlivé hodnoty. Vhodným použitím obrázkov sa zníži kognitívna záťaž na používateľa, ľahšie sa rozhodne, znižuje sa množstvo spätných opráv a súčasne je tiež rozhranie atraktívnejšie. Obrázky by mali zachytávať hlavné rozdiely medzi ponúkanými možnosťami a nemusia zachytávať nepodstatné detaily nesúvisiace s výberom.

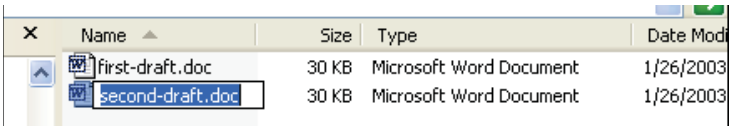


Obrázok 1-18. Ilustrované voľby (angl. Illustrated Choices).

Vytváranie a editovanie

Návrh nástrojov pre tvorbu a úpravu obsahu, editorov, je komplikovanejší ako návrh nástrojov určených len na prehliadanie obsahu. Komplikácie sú spôsobené pridaním dodatočnej funkcionality a nutnosťou zachovať rozhranie zrozumiteľné aj v režime úpravy obsahu.

Vzor *Editácia na mieste* (obrázok 1-19) používa pre úpravu obsahu malý editor, ktorý sa otvorí na mieste pôvodného obsahu. Umiestnenie editora presne na mieste obsahu, ktorý práve upravuje, zachováva používateľove sústredenie a nevyžaduje od používateľa hľadať prípadné nové okno editora na inom mieste v rozhraní. Editor sa zvyčajne otvorí dvojklikom na text, ktorý chceme upraviť.



Obrázok 1-19. Editácia na mieste (angl. Edit-in-Place).

Vzor *Inteligentné označenie* (angl. *Smart Selection*) zjednodušuje vyznačovanie koherentnej skupiny objektov ako napr. vyznačenie celej vety v textovom editore alebo súvislej oblasti pixelov rovnakej farby v grafickom editore. Nie všetci používatelia sú schopní vykonávať precízne koordinované pohyby myšou (alebo iným vstupným zariadením), a preto netriviálne vyznačenie obsahu im môže spôsobovať problémy. Používateľom sa takto zjednoduší práca tým, že proces vyznačovania je podporený inteligentným algoritmom.

1.5.3 Vzory pre web

Webové stránky sa líšia od desktopových aplikácií a pre návrh webových stránok sú preto potrebné odlišné prístupy ako sme doteraz spomínali. Hlavný rozdiel webových stránok (aplikácií) oproti klasickým desktopovým aplikáciám je ten, že časť aplikácie je umiestnená na serveri. Výhodou je, že všetci používatelia ľahko prejdú na novšiu verziu, pretože stačí verziu upraviť na serveri a pri najbližšej návšteve servera sa používateľom spustí už nová verzia. Nevýhodou je vyššia latencia pri práci, súvisiaca s časom potrebným na prenos požiadavky a odpovede po sieti, a skutočnosť, že rôznym používateľom sa môže stránka zobraziť na iných zariadeniach alebo prehliadačoch odlišne. Z tohto dôvodu sa pri návrhu webových stránok a aplikácií kladie hlavný dôraz na funkčnosť obsahu, a nie na integritu alebo identitu zobrazenia.

Pohyb medzi webovými stránkami zabezpečujú odkazy, a preto vzory pre navigáciu sú prirodzene aplikovateľné aj pri návrhu webových stránok a sídiel. Stále viac sú tu tiež užitočné aj ostatné typy vzorov, keďže najnovší trend vo vývoji webových stránok sa posúva od statických HTML stránok smerom k dynamickým webovým aplikáciám realizovanými technológiami *AJAX*, *Macromedia Flash* a *Microsoft Silverlight*. Knižnice vzorov ako napr. *Yahoo! Design Pattern Library*[9] a *Interaction Design Pattern Library*[10] (Welie, 2008) obsahujú často používané prvky na webových stránkach.

---

[9] Yahoo! Design Pattern Library, `http://developer.yahoo.com/ypatterns/`

[10] Interaction Design Pattern Library, `http://www.welie.com/patterns/`

Welie rozdeľuje vzory pre web do troch hlavných kategórii: *navigácia*, *vyhľadávanie* a *nakupovanie*. Predpokladá, že dobre navigovateľná stránka upúta pozornosť návštevníkov a umožní im efektívne vykonávať úlohy. Welie navrhuje viaceré typy navigácie na stránkach: horný panel, ľavý panel, omrvinky[11] (angl. *Breadcrumbs*) a rôzne typy menu. Vzory pre hľadanie sú inšpirované súčasnými prístupmi v internetových vyhľadávačoch (Google, Yahoo!) a obsahujú prvky pre jednoduché a pokročilé vyhľadávanie (angl. *Advanced Search*), zobrazovanie výsledkov vyhľadávania, často kladené otázky (angl. *Frequently Asked Questions – FAQ*), atď.

Vzory pre nakupovanie zahŕňajú celý proces nakupovania vo webovom sídle: rezervácia tovaru/služieb, porovnanie produktov, konfigurátor produktov, jednotlivé kroky nákupu, nákupný košík a geografický vyhľadávač obchodov. Vzor *Nákupný košík* (obrázok 1-20) združuje dôležité údaje o objednávke tovaru alebo služieb od predajcu. Obsahuje zoznam položiek, poskytuje prvky rozhrania pre pridanie a odobratie položiek z košíka, zobrazuje ceny, množstvá, celkovú sumu pred zdanením, daň, a celkovú cenu. Tiež umožňuje výpočet prepravných nákladov.

Zo všeobecnejších vzorov, vzor *Pošli odkaz priateľovi* (angl. *Send-a-Friend Link*) umožňuje poslať odkaz na aktuálnu stránku emailom spolu s krátkou správou pre prijímateľa.

Obrázok 1-20. Nákupný košík (angl. Shopping Cart).

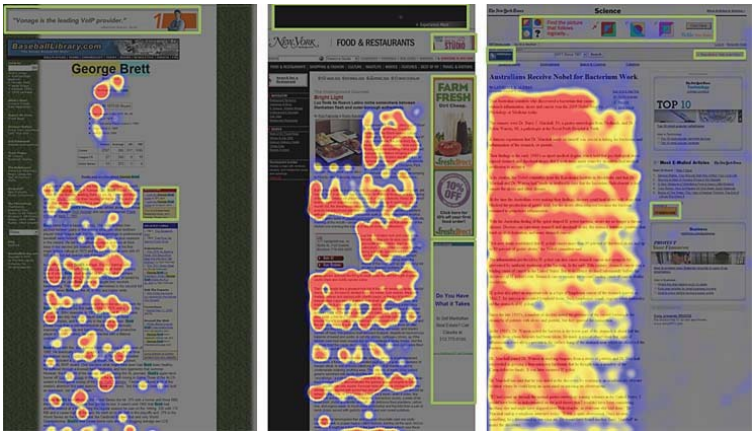
Rozdielnosť v zobrazovaní webových stránok na rôznych zariadeniach a prehliadačoch a v súčasnosti kľučová úloha webovej prezentácie pri upútaní zákazníkov motivuje rozsiahle používateľské štúdie prvkov webových stránok. Pozri napr. pravidelný stĺpček Jakoba Nielsena[12].

Používateľské štúdie odkrývaju často neočakávané súvislosti. Pre ilustráciu sa pozrime na použitie bannerov. Banner je prvok rozhrania obdĺžnikového formátu, väčšinou obrázok alebo krátky film, slúžiaci na štylizovanú prezentáciu nejakého obsahu. Jeho hlavnou úlohou je upútať návštevníka a používa sa preto väčšinou na reklamné účely. Návštevníci webových stránok si však bannery už prestali všímať, keďže väčšinou sprostredkúvajú len reklamný obsah, a tak prvky rozhrania, ktoré sa vizuálne ponášajú na bannery návštevníci ignorujú (Nielsen, 2007). Na obrázku 1-21 je znázornená teplotná mapa návštev troch webových stránok (svetlejšie miesta používatelia čítajú častejšie ako

[11] Jeden riadok textu zobrazujúci umiestnenie aktuálnej stránky v celkovej hierarchii.

[12] Jakob Nielsen's Alertbox, `http://www.useit.com/`

tmavšie miesta) a reklamné bannery sú zarámované. Zaujímavé je, že stránky obsahujú aj prvky nereklamného charakteru vizuálne sa podobajúce na typický banner a návštevníci ich tak pri čítaní automaticky ignorujú. Použitie banneru by sa teda dalo charakterizovať ako antivzor.



Obrázok 1-21. Ako používatelia čítajú webovú stránku (Nielsen, 2007).

1.5.4 Vzory pre sociálne aplikácie

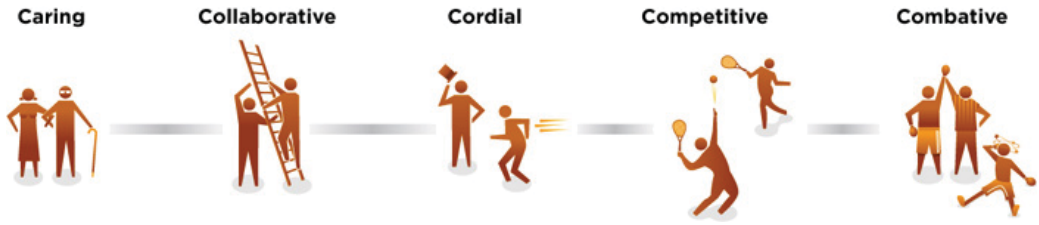
Vzory pre sociálne aplikácie zachytávajú odporúčané postupy pri návrhu rozhraní pre aplikácie tzv. sociálneho webu, v ktorom sa dôraz kladie na podporu interakcie medzi ľuďmi pomocou použitia počítača a internetu. V súčasnosti asi najpoužívanejšími sociálnymi aplikáciami sú sociálne portály (napríklad *Facebook*, *MySpace*), portály pre zdieľanie fotografií (napr. *Flickr*) a iné sociálne služby poskytované prostredníctvom Internetu, ako napríklad sociálne značkovanie *del.icio.us*.

Ľudia sa k počítačom správajú prirodzene sociálne (Reeves, 2003). Reeves a Nass uskutočnili sériu štúdií, ktoré demonštrujú zaujímavé správanie používateľov. Napríklad: podobne ako pri hodnotení druhých ľudí, používatelia sú zdvorilí pri hodnotení počítača, keď sa ich spýta na svoje správanie; pri skupinových úlohách, používatelia majú počítač radšej, keď je s nimi v rovnakom tíme.

Knižnica *Yahoo! Design Pattern Library* uvádza základné vzory pre sociálne aplikácie. Vo webových komunitách je veľký potenciál pre interakcie medzi používateľmi (členmi komunity). Základným prvkom každej komunity je preto *reputácia*, teda hodnotiaci údaj člena vytvorený agregáciou predchádzajúcich interakcií príslušného člena komunity. Reputačný systém komunity odmeňuje niektoré typy správania a iné penalizuje. V závislosti od stupňa súťaživosti v komunite sú vhodné iné reputačné mechanizmy, keď napr. zavedením odmeňovania súťaživého správania sa členov môžeme porušiť nesúťaživé prostredie niektorých komunít.

Spektrum súťaživosti (obrázok 1-22) zachytáva rôzne typy komunít od nesúťaživých až po priam bojové prostredia. V súcitných (angl. *caring*) komunitách sú členovia motivovaní pomáhať ostatným členom komunity. Vhodné je reputáciou identifikovať skúsených členov komunity, aby ich mohli druhí vyhľadať pre pomoc a radu. V kolaboratívnych (angl. *collaborative*) komunitách zdieľajú členovia rovnaké ciele a pracujú spolu pre ich

dosiahnutie. Reputáciou je vhodné odmeňovať dôveryhodné a spoľahlivé správanie. V srdečných (angl. *cordial*) komunitách majú členovia vzájomne nekonfliktné ciele, nie nutne rovnaké. Reputácia by mala zobrazovať históriu predchádzajúcich akcií člena, aby ostatní získali všeobecný prehľad o aktivitách a záujmoch príslušného člena. V súťaživých (angl. *competitive*) komunitách zdieľajú členovia rovnaké ciele, ale súťažia medzi sebou o ich dosiahnutie. Vhodné je, keď reputácia prezentuje dosiahnuté úspechy člena tak, aby ich mohli druhí členovia uznať, prípadne obdivovať. Nakoniec, v bojovných (angl. *combative*) komunitách majú členovia rozdielne vzájomne sa vylučujúce ciele. Reputácia sa používa na zobrazenie histórie úspechov, prehier, víťazstiev a slúži na vystatovanie sa členov komunity.



Obrázok 1-22. Spektrum súťaživosti v sociálnej komunite (Yahoo!, 2008).

Ďalšie princípy a vzory pre sociálne aplikácie navrhuje (Malone, 2009). Vo zvyšku tejto časti spomenieme aspoň niektoré. V sociálnych aplikáciach rozhranie obsahuje generické prvky spoločné pre všetkých používateľov a vlastné personalizované prvky príslušného používateľa. Pri návrhu je problém ako pomenovať tieto prvky. Použiť privlastňovacie zámeno *môj* (angl. *My*) alebo *tvoj* (angl. *Your*)? Teda, napr. Moji Priatelia (angl. *My Friends*) vs. Tvoji Priatelia (angl. *Your Friends*), resp. aj názvy známych sociálnych portálov obsahujú privlastňovacie zámeno (*MyYahoo*, *MySpace*, *YouTube*). Podľa (Malone, 2009) použitie zámena *môj* imituje situáciu z pohľadu používateľa, ako keby si používateľ označil svoje vlastné veci. Toto označenie preto dobre funguje pre súkromné individuálne prostredia. V sociálnych aplikáciach je introvertné správanie nevýhodné a návrhári sa pokúšajú podporiť používateľov, aby sa otvorili ostatným a komunikovali spolu. Použitie zámena *tvoj* podporuje konverzáciu, a používa sa na vtiahnutie mysle používateľov do dialógu s ostatnými.

Malone ďalej analyzuje vhodné prístupy pri registrácii používateľov do systému, prihlasovania sa do a odhlasovania sa zo systému, posielanie a príjmanie pozvánok od iných používateľov a potrebu zaujímavej a vyzývavej uvítacej stránky. Vzor *Uvítanie* (obrázok 1-23) je analógia vzoru *Jasné vstupné body* a predstavuje používateľovi základnú funkcionalitu aplikácie pri prvom prihlásení. Používateľ dostane od reprezentanta komunity srdečné uvítanie, po ktorom sa cíti ako vítaný hosť v príslušnej komunite. Zároveň môžeme odoslať uvítaciu emailovú správu.

Ďalšia séria vzorov slúži na reprezentovanie jednotlivca v komunite. Členovia majú v komunite jednoznačnú identitu, profil, s ktorou je väčšinou spojená aj webová stránka, ktorú druhí členovia navštevujú a reprezentuje príslušného člena voči ostatným. Vhodné je umožniť používateľom si túto stránku prispôsobiť a umožniť im tak lepšie vyjadriť svoju osobitosť. Zvyčajne je člen na ostatných stránkach reprezentovaný svojim menom a obráz-

kom (tzv. avatarom), prípadne môže uvádzať aj ďalšie údaje afektívneho charakteru, napr. aktuálnu náladu alebo aktuálny partnerský stav.

Obrázok 1-23. Uvítanie (angl. Welcome Area).

Asi najdôležitejšia funkcionalita sociálnych aplikácií je socializovanie sa s ostatnými členmi komunity. Vzory pre manažment priateľov zahŕňajú aktivity ako vyhľadávanie členov podľa ich charakteristík, prídávanie priateľov (obrázok 1-24) a odstraňovanie priateľov. Veci však nie sú také jednoduché a identifikovali sa už aj nevhodné riešenia. Napr. antivzor *Expriateľka* (angl. *Ex-girlfriend*) existuje ak sociálna aplikácia odporúča členom ďalších členov (možných nových priateľov) na základe systému priateľ-priateľa, a teda po rozviazaní krátkodobého vzťahu môže ďalej odporúčať neželaných priateľov z okolia predchádzajúceho partnera. Sociálna aplikácia preto musí udržiavať viaceré typy vzťahov medzi členmi a poskytovať podporu interakcí vzhľadom na tieto typy.

Obrázok 1-24. Pridaj priateľa (angl. Add Friend).

1.5.5 Zhodnotenie vzorov HCI

Vzory pri návrhu rozhraní človek-počítač zachytávajú overené prístupy k riešeniu opakujúcich sa problémov. Neuviedli sme všetky doteraz existujúce vzory, nakoľko to nie je

v takomto rozsahu technicky možné. Knižnice vzorov pre návrh desktopových aplikácií, webových stránok a najnovšie aj sociálnych aplikácií sú priebežne dopĺňané podľa meniacich sa trendov v správaní sa používateľov.

Návrh rozhrania väčšinou nie je čierno-biely a neexistujú len dobré a len zlé návrhy, ktoré by sme vedeli ľahko rozlíšiť a vybrať si ten správny. Rôzne návrhy umožňujú používateľom vykonávať požadované funkcie s rôznou efektívnosťou (použiteľnosťou), a na zlepšenie použiteľnosti rozhrania je potrebné realizovať testy použiteľnosti, ktoré sú náročné na čas, ľudské zdroje a finančné prostriedky. Použitie vzorov je preto výhodná alternatíva pri tvorbe efektívnych rozhraní človek-počítač.

Použitá literatúra

[1] Bieliková, M., Návrat, P. et al.: *Štúdie vybraných tém softvérového inžinierstva 1*. Vydavateľstvo STU, Bratislava, 2006.

[2] Bieliková, M., Návrat, P. et al.: *Štúdie vybraných tém softvérového inžinierstva 2*. Vydavateľstvo STU, Bratislava, 2006.

[3] Flint, E. S.: *The COBOL jigsaw puzzle: Fitting object-oriented and legacy applications together.* IBM Systems Journal, Vol. 36, No. 1, 1997.

[4] Fowler, M.: *Patterns of Enterprise Application Architecture.* Reading, MA: Addison-Wesley Professional, 2002.

[5] Gamma, E. et al.: *Design Patterns*. 1st edition. Massachusetts: Addison-Wesley, pp. 395, 1995.

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Professional, 1995.

[7] Grand M.: Pattern Summaries: Abstract Factory Pattern [online, február 2008]. Dostupné z: `http://www.developer.com/java/other/article.php/626001`

[8] Grlický, V.: *Information integration from disparate Web sources.* Diplomová práca. Bratislava: Fakulta informatiky a informačných technológií STU, 2003.

[9] Hohpe, G., Woolf, B: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Boston, MA: Pearson Education, Inc., 2004.

[10] Kaisler, S. H.: *Software Paradigms.* Hoboken, NJ: John Wiley & Sons, Inc., 2005

[11] Kraval, I.: Design Patterns v OOP, 2002. `http://www.objects.cz/`

[12] Malone, E., Crumlish, Ch.: *Designing Social Interfaces: Principles, Best Practices and Patterns for Designing the Social Web.* O'Reilly Media, 2009.

[13] Nielsen, J.: *Usability Engineering.* Morgan Kaufmann, San Francisco, 1993.

[14] Nielsen, J.: *Banner Blindness: Old and New Findings.* Jakob Nielsen's Alertbox, 2007. Dostupné z: `http://www.useit.com/ alertbox/banner-blindness.html`

[15] Norman, D.: *The Psychology of Everyday Things*. Basic Books, New York, 1988.

[16] Reeves, B., Nass, C.: *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places.* Center for the Study of Language and Information, 2003.

[17] Riehle, D., Züllighoven, H.. "Understanding and Using Patterns in Software Development." Theory and Practice of Object Systems 2(1):3–13. 1996.

[18] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2*. Chichester, England: John Wiley & Sons, Ltd., 2000.

[19] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. 2nd Edition. Reading, MA: Addison-Wesley Professional, 2002.

[20] Trowbridge, D., Roxburgh, U., Hohpe, G. et al.: Integration Patterns. Microsoft Corporation, 2004.

[21] Tidwell, J.: *Common Ground: A Pattern Language for Human-Computer Interface Design*, 1999. Dostupné z: `http://www.mit.edu/~jtidwell/interaction_patterns.html`

[22] Tidwell, J.: *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media, 2005. Dostupné z: `http://designinginterfaces.com/`

[23] van Welie, M.: *Interaction Design Pattern Library* [online, 2008]. Dostupné z: `http://www.welie.com/patterns/`

[24] Yahoo! Inc.: Yahoo! Design Pattern Library, 2008. Dostupné z: `http://developer.yahoo.com/ypatterns/`

2
SOFTVÉROVÉ SÚČIASTKY

Nikoleta Habudová, Tomáš Kuzár, Pavol Mederly,
Marián Šimko, Jozef Tvarožek, Ivan Kapustík

Súčiastky predstavujú stavebné bloky informačného systému. Ich najlepšou analógiou je stavebnica Lego. V nej má každý blok akéhokoľvek tvaru štandardné rozhranie, ktoré mu umožňuje jednoduché spojenie s ostatnými súčiastkami. Tieto malé bloky sú vďaka kolíkom na jednej strane a dutinám na strane druhej štandardizované, čím ich možno spájať do komplexných štruktúr.

Na rozdiel od Lega, softvérové súčiastky môžu mať rôzne funkcie ako aj rôzne rozhrania. Napriek tomu rozhrania súčiastok musia zodpovedať určitým štandardom, aby bolo možné zlučovať súčiastky do jednej systémovej štruktúry. Podobne ako pri Legu, škála systému môže byť variabilná, od malého systému po veľký. Na rozdiel od návrhových vzorov sú súčiastky čiastočne alebo úplne implementované (v zdrojovej alebo binárnej podobe).

Softvérová súčiastka je znovupoužiteľný softvérový stavebný blok: kus zapuzdreného aplikačného kódu, ktorý môže byť kombinovaný s inými súčiastkami a doplnený o prídavný kód, s cieľom vytvoriť požadovanú aplikáciu. Súčiastky môžu byť jednoduché alebo komplexné. Neexistuje ale všeobecná dohoda o tom, čo je a čo nie je softvérová súčiastka. Súčiastky majú rôzne tvary a veľkosti. Môžu byť veľmi malé (napr. tlačidlo v grafickom používateľskom rozhraní) alebo môžu implementovať komplexnú aplikačnú službu.

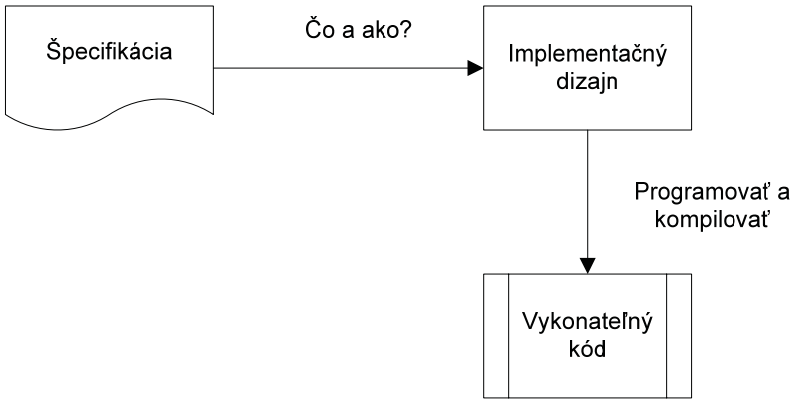
Pod pojmom súčiastka sa vo všeobecnosti myslí tzv. „binárna súčiastka“, ktorá predstavuje skompilovaný softvérový artefakt integrovateľný do aplikácie kedykoľvek počas vykonávania programu. Zdrojový kód obyčajne nie je k dispozícii, takže súčiastku nie je možné modifikovať. Príkladmi binárnych súčiastkových systémov sú OpenDoc v Apple, Dynamically Linked Library (DLL) v Microsoft Windows 9x/NT, Taligent CommonPoint system a Microsoft Foundation Classes (MFC).

Aplikácia pozostáva z množiny súčiastok a zároveň poskytuje prostredie, do ktorého sú súčiastky vložené. Súčiastky interagujú so svojim okolím a môžu interagovať s operačným systémom daného počítača, na ktorom sú spúšťané. Súčiastky zároveň interagujú navzájom medzi sebou pomocou rozhrania.

Aspekty súčiastok

Súčiastka má tri aspekty. Schematicky ich znázorňuje obrázok 2-1.

- *špecifikácia* – opisuje funkciu, ktorú súčiastka plní. Napríklad, čo súčiastka vykonáva a ako ju možno použiť,
- *implementačný návrh* – opisuje ako má implemetátor navrhnúť a skonštruovať softvér a dátové štruktúry, aby splnil danú špecifikáciu,
- *vykonateľný kód* – realizuje požadovanú funkcionalitu súčiastky v rámci konkrétnej platformy.



Obrázok 2-1. Aspekty softvérovej súčiastky.

Rozhranie (angl. *interface*) sumarizuje ako má klient interagovať so súčiastkou, ale skrýva implementačné detaily. Klient naopak môže byť závislý od tohto rozhrania. Špecifikácia modelu rozhrania nie je implementačným návrhom. Rozhranie neobsahuje informáciu o tom, ako sú dáta vnútri súčiastky uložené alebo organizované. Cez rozhranie nie je možné pristupovať k implementácii súčiastky.

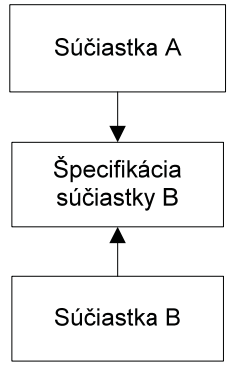
Vzťah medzi dvoma súčiastkami je znázornený na obrázku 2-2. Súčiastka B má špecifikáciu, ktorá je implementovaná v nejakom programovacom jazyku. Súčiastka A spĺňa požiadavky aplikačného rozhrania (API; angl. *Aplication Programming Interface*), ktoré je obsiahnuté v špecifikácii súčiastky B, práve vtedy, keď A zavolá funkcionalitu, ktorú poskytuje súčiastka B.

V ideálnom prípade by implementácia súčiastok mohla byť napísaná v rôznych programovacích jazykoch a mohla by sa vykonávať na rôznych softvérových platformách (odlišných od platformy klientskeho programu). Avšak v súčasnosti sa spolupracujúce súčiastky programujú väčšinou len v rovnakom programovacom jazyku.

Opis rozhrania

Rozhranie je sada sémanticky prepojených funkcií implementovaných na objekte (Brockschmidt, 1994). Rozhranie sumarizuje správanie a zodpovednosti, ktoré súčiastka bude musieť dodržať, keď bude súčasťou zostavy (angl. *assembly*) reprezentujúcej nejakú aplikáciu. Rozhranie špecifikuje API, ktoré používajú klienti súčiastok, aby zavolali služby súčiastok, ale nešpecifikuje implementáciu. Toto oddelenie umožňuje zameniť implementá-

ciu služby bez toho, aby bolo nutné upraviť rozhrania, a zároveň pridávať nové služby bez zmeny existujúcej implementácie.



Obrázok 2-2. Vzťah medzi dvoma softvérovými súčiastkami.

Sú súčiastky objektmi?

Vo všeobecnosti súčiastky nie sú objektmi. Ale objekty môžu byť jednoduchými súčiastkami. Veľa objektov môže obsahovať nejakú súčiastku. Dôležitý rozdiel je v poslaní. Objekt slúži na opis entity reálneho sveta, kým súčiastka je určená na opis služieb tejto entity. Inak povedané, objekty sú vhodné na opis doménovej štruktúry, zatiaľ čo súčiastky opisujú jej funkcionalitu. Keď objekt požiada o službu, nevie ktorý z ostatných objektov túto službu poskytuje. Keďže súčiastka musí poskytovať služby rôznym objektom – často anonymne – neuchováva si stavovú informáciu (samozrejme, len vtedy, ak to práve nie je vlastnosť poskytovanej služby).

Predstavme si, že máme hardvérový ovládač (angl. *driver*), ktorý zabezpečuje prístup k viacerým podobným zariadeniam, pričom každé zariadenie má svoje jedinečné črty. Ak programu nezáleží na výbere zariadenia, ktoré vykoná požadovanú službu, môže požiadať o službu od súčiastky a zároveň ju poveriť výberom zariadenia. Súčiastka následne zavolá vhodnú metódu na objekte príslušného zariadenia. Naopak, ak program presne vie, od ktorého z dostupných zariadení službu vyžaduje, môže zavolať alebo súčiastku s konkrétnou špecifikáciou zariadenia, alebo priamo objekt požadovaného zariadenia.

Súčiastky poskytujú služby. Súčiastky sú zložené z objektov, ktoré vzájomnou spoluprácou implementujú súčiastkami poskytovanú službu.

V súčasnosti softvér čím ďalej tým viac využíva súčiastky. Výhody ich použitia sú:

- zvýšená produktivita vďaka použitiu hotových súčiastok,
- zvýšená spoľahlivosť použitím dostatočne otestovaného zdrojového kódu,
- nízke náklady na údržbu, pretože rozsah kódu je malý,
- minimálne zmeny v kóde,
- zabezpečenie dostatočne uzatvoreného mechanizmu na zabalenie, distribúciu a znovupoužitie softvéru.

Nevýhodou môžu byť tieto riziká:

- nemožnosť opraviť alebo skontrolovať kód (angl. *bugs and backdoors*),

- vzájomná nekompatibilita súčiastok,
- požiadavky na rôzne verzie tej istej súčiastky od rôznych aplikácií,
- licencovanie.

Zloženie súčiastok

Kľúčovou výhodou používania súčiastok je možnosť kombinovať dve a viac súčiastok a vyrobiť tak novú súčiastku. Všeobecne predpokladáme, že nová súčiastka zapuzdruje použité súčiastky do čiernej skrinky (angl. *black box*). Každá množina súčiastok sa môže stať súčasťou väčšej množiny, a môže teda tvoriť potencionálny subsystém. Keďže súčiastky poskytujú služby, musíme dokázať odhadnúť a predvídať správanie sa spolupracujúcich skupín súčiastok. Často práve toto nie je možné predpovedať len z analýzy interakcie súčiastok.

Pokiaľ ide o hardvérový priemysel, spájanie súčiastok je tu skôr normou než len výnimkou. Ešte nedávno výrobcovia počítačov vyrábali takmer každú časť svojich počítačových systémov až po monitor a tlačiareň. Dnes sa dá počítač poskladať ako skladačka z Lega zo súčiastok pochádzajúcich od rôznych výrobcov.

V softvérovom priemysle to takto nefunguje, hoci existuje mnoho príkladov softvérových aplikácií, ktoré dovoľujú pripojenia (angl. *plug-ins*). Napríklad aplikácia *Netscape Browser*. Otázka teda znie: Prečo to v hardvérovom priemysle funguje, a v softvérovom nie? Odpoveďou je pravdepodobne to, že zatiaľ neexistujú definície štandardných súčiastok pre všeobecne poskytované služby. A ak aj niekde existuje štandard, predajcovia si zvykli ponúkať „zlepšenia“, ktoré vedú k nekompatibilite.

2.1 Distribuované súčiastky

Distribuovaný systém je množina samostatných geograficky oddelených výpočtových uzlov prepojených prostredníctvom určitého komunikačného systému – v rámci lokálnej počítačovej siete, metropolitnej siete, internetu, a pod. Distribuovaná aplikácia je aplikácia, ktorej časti sú vykonávané na uzloch distribuovaného systému, a distribuované súčiastky sú základné stavebné bloky takýchto distribuovaných aplikácií. Distribuované aplikácie tak, vďaka využitiu zdrojov v jednotlivých uzloch systému, môžu dosahovať lepšiu priepustnosť pri spracúvaní veľkého množstva dát, neporovnateľne vyššiu v porovnaní s jedným bežným počítačom.

V tejto časti opisujeme všeobecné charakteristiky distribuovaných súčiastok, konkrétne technológie, ako napr. súčiastky založené na udalostiach a technológia CORBA, sú potom opísané v nasledujúcich častiach.

Na problematiku distribuovaných súčiastok sa pozeráme z dvoch odlišných hľadísk – z pohľadu spájajúceho softvéru a z pohľadu vytvárania rozsiahleho systému založeného na nejakom spájajúcom softvéri.

2.1.1 Spájajúci softvér

Pri tvorbe distribuovaného systému očakávame, že budeme mať k dispozícii veľké množstvo samostatných súčiastok, ktoré budeme vedieť vzájomne spájať do väčších celkov s cieľom získať komplexnú aplikáciu podľa požiadaviek.

Na prepojenie distribuovaných súčiastok sa používa tzv. spájajúci softvér (angl. *middleware)*. V súčastnosti existuje niekoľko rozšírených technológii slúžiacich ako spájajúci softvér, napr.:

- súčiastkový objektový model alebo distribuovaný súčiastkový model od firmy Mircosoft (angl. *Microsoft COM/DCOM*),
- objektový sprostredkovateľ požiadaviek (angl. *CORBA*),
- podnikové bôby jazyka Java (angl. *Enterprise JavaBeans*),
- webové služby (angl. *Web Services*).

Problematika distribuovaných súčiastok je spojená s rozvojom viacerých súvisiacich technológií. Rozvoj distribuovaných systémov súvisel na jednej strane s rozvojom veľkých distribuovaných podnikových aplikácií a na druhej strane s väčšou ponukou hardvérovej základne. Zistilo sa napríklad, že je lacnejšie prepojiť niekoľko bežných počítačov ako použiť jeden supervýkonný stroj. S touto skutočnosťou súvisí napríklad rozvoj gridového počítania (angl. *Grid computing*), oblakového počítania (angl. *Cloud computing)*, či použitie softvéru ako služby (angl. *Software as a Service*). Ale samotnou podstatou spájajúceho softvéru je vytvorenie softvérovej infraštruktúry, ktorá by umožňovala jednoduché prepojenie distribuovaných súčiastok. Rozvoj technológií umožňuje spracovávať a uchovávať obrovské objemy dát.

2.1.2 Charakteristika distribuovaného systému

Podstatou distribuovaného systému je, že jednotlivé súčiastky dokážu spolupracovať i napriek tomu, že sú geograficky oddelené, i napriek tomu, že sa vykonávajú na rôznych operačných systémoch a niekedy aj na rozdielnych hardvérových platformách. Aby jednotlivé distribuované súčiastky dokázali spolupracovať, je potrebné zaoberať sa tromi základnými problémami: umiestnením súčiastok, vzájomným používaním súčiastok a vysporiadaním sa s chybami. Preto je pri navrhovaní distribuovaného systému nutné vziať do úvahy niekoľko základných princípov:

- umiestnenie – každá súčiastka v systéme musí mať možnosť nájsť a vidieť ostatné súčiastky v rámci systému,
- použitie – súčiastky musia byť schopné používať funkcionalitu iných súčiastok,
- vysporiadanie sa s chybami – súčiastka musí byť schopná vysporiadať sa so zlyhaním inej súčiastky, hardvérovou chybou, chybou v sieťovej komunikácii a podobne.

V začiatkoch distribuovaných systémov boli tieto problémy riešené na úrovni aplikácie – nejakým spôsobom bolo zabezpečené, aby sa jednotlivé súčiastky dokázali vidieť, aby sa vedeli navzájom používať a nejakým spôsobom bol zabezpečený mechanizmus vysporiadavania sa s chybami. Vývojár sa pri tvorbe nemohol naplno zaoberať doménovou logikou súčiastky, ale musel sa vysporiadavať s mnohými technickými detailami spomenutými vyššie. Tieto skutočnosti iniciovali vznik spájajúceho softvéru, ktorý by umožnil prepojiť softvérové súčiastky bez ohľadu na operačný systém či hardvérovú platformu, na ktorej sa vykonávajú. Každý typ spájajúceho softvéru rieši otázky umiestnenia, používania a vysporiadania sa s chybami svojim spôsobom. Vývojár distribuovaného systému sa tak

môže zamerať na doménovú logiku aplikácie a nemusí riešiť technické otázky ohľadom spolupráce distribuovaných súčiastok.

2.1.3 Spôsob fungovania spájajúceho softvéru

Aby mohol distribuovaný systém vôbec začať pracovať, je potrebné jednotlivé súčiastky distribuovať na jednotlivé výpočtové uzly. Rozlišujeme dva spôsoby distribúcie podľa toho kedy k nej v programe dochádza – statická distribúcia a dynamická distribúcia.

Statická distribúcia

Statická distribúcia prebieha počas kompilácie a spájania (angl. *linking)*. Určitá časť kódu je priradená na vykonanie konkrétnemu hardvéru.

Dynamická distribúcia

Na druhej strane pri dynamickej distribúcii dochádza k priradeniu softvérovej súčiastky konkrétnemu hardvéru počas vykonávania programu. Spájajúci softvér musí zabezpečiť, aby súčiastka alebo služba boli zaregistrované a tým použiteľné aj pre ďalšie súčiastky v systéme.

Ďalšou otázkou, ktorá je riešená na strane spájajúceho softvéru, je vzájomné použitie. V predchádzajúcom texte sme spomínali, že jednotlivé súčiastky sa musia navzájom vidieť a musia byť schopné navzájom zdieľať svoju funkcionalitu. Z pohľadu komunikácie, existujú dva základné spôsoby ako môžu súčiastky v rámci distribuovaného systému vzájomne komunikovať: volanie vzdialených procedúr (angl. *remote procedure call)* alebo posielanie správ (angl. message passing*)*. Ostatné metódy sú kombináciou týchto dvoch. Detaily týkajúce sa komunikácie, vysporadúvania sa s chybami či viditeľnosti vzdialených súčiastok sú výlučne v kompetencii spájajúceho softvéru, a pri návrhu a tvorbe systému sa vývojári týmito skutočnosťami zaoberať nemusia.

Volanie vzdialených procedúr

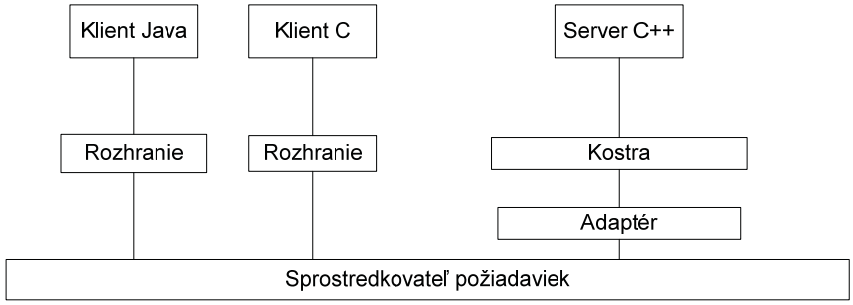
Volanie vzdialených procedúr je spôsob komunikácie umožňujúci programu transparentne volať kód (procedúru) v adresnom priestore iného programu. V prípade implementácie mechanizmu volania vzdialených procedúr na strane spájajúceho softvéru tak môže programátor využívať funkcionalitu distribuovaných súčiastok rovnako ako keby boli umiestnené na lokálnom stroji.

Posielanie správ

Komunikácia medzi súčiastkami pomocou posielania správ je založená na asynchrónnom odosielaní požiadaviek v balíčkoch (správach). Správy, na rozdiel od volaní vzdialených procedúr, nezanikajú v prípade ak niektorá z komunikujúcich súčiastok prestane fungovať, ale zvyčajne zostávajú uložené na niektorom z pomocných serverov, ktoré slúžia na preposielanie správ v takejto architektúre. Mechanizmus založený na posielaní správ je preto robustnejší pri neočakávaných komunikačných problémoch, vyššia réžia ale mierne znižuje rýchlosť a priepustnosť takejto komunikácie.

Obrázok 2-3 zachytáva príklad architektúry spájajúceho softvéru, ktorý umožňuje volanie vzdialených metód. Softvéry klienta a servera môžu byť napísané v ľubovoľných programovacích jazykoch, potrebné je ale mať k dispozícii vhodné súčiastky v týchto jazy-

koch, ktoré umožnia klientskej súčiastke a serverovej súčiastke komunikovať prostredníctvom spájajúceho softvéru. Implementácia spájajúceho softvéru je pre vývojára klientskej alebo softvérovej súčiastky ukrytá. Vývojár potrebuje len možnosť používať spájajúci softvér v príslušnom programovacom jazyku (napr. pomocou podpornej knižnice).



Obrázok 2-3. Príklad architektúry spájajúceho softvéru.

2.1.4 Návrh distribuovanej aplikácie

Návrh distribuovanej aplikácie je z viacerých dôvodov komplikovanejší ako návrh centralizovanej aplikácie vykonávanej v homogénnom prostredí jedného výpočtového uzla. Uvedieme niekoľko základných aspektov, ktoré distribuovanej aplikácii pridávajú na zložitosti: vývoj softvéru, plánovanie zdrojov, správa chýb, používanie nehomogénnych prostredí a problematickou sa javí aj otázka bezpečnosti distribuovaného systému.

Vývoj softvéru

Samotná zložitosť vývoja softvéru je často spojená so skutočnosťou, že neexistuje možnosť ladenia distribuovanej aplikácie, ako to poznáme pri centralizovaných systémoch. Vývoj aplikácie je spojený s problémami ako napr. vzdialená komunikácia, transakcie, udalosti, pomenovávanie (angl. *naming*, t.j. priraďovanie človeku zrozumiteľných mien entitám v distribuovanom systéme) či bezpečnosť.

Plánovanie zdrojov

Plánovanie zdrojov v distribuovanej aplikácii je náročné na plánovanie jednotlivých aktivít. Vo väčšine prípadov sa totiž softvérová súčiastka priraďuje na vykonanie konkrétnemu hardvéru až v čase vykonávania programu. Pri plánovaní zdrojov je preto potrebné brať ohľad na požiadavky služieb a súčiastok a tiež na možnosti jednotlivých hardvérových súčiastok.

Správa chýb

Správa chýb v distribuovanom systéme je odlišná od správy chýb v centralizovanom systéme najmä tým, že v prípade neošetrenej chyby v centralizovanom systéme dôjde k pádu celého systému, na druhej strane v prípade distribuovanej aplikácie pri zlyhaní jednej súčiastky alebo procesu v nejakom uzle nedôjde k pádu celej aplikácie. Dokonca napr. neukončenie niektorého z procesov môže spôsobiť zlyhanie inej, na prvý pohľad s daným procesom nesúvisiacej, súčiastky. Z tohto dôvodu nemožno v distribuovanom systéme zanedbať mechanizmus správy chýb.

Nehomogénne prostredie

Distribuované súčiastky sa nespájajú len s distribuovaným geografickým rozmiestnením výpočtových uzlov ale aj rôznorodosťou (heterogénnosťou) použitého softvéru a hardvéru. Spájajúci softvér sa dokáže vysporiadať s rôznorodosťou prostredia. Na zjednotenie hardvérových a softvérových platforiem je však možné tiež použiť koncept virtuálneho stroja, kde distribuovaná aplikácia sa bude reálne vykonávať na rozdielom hardvéri a rôznych operačných systémoch. Použitím virtualizácie teda zabezpečíme jednotnosť hardvéru a softvéru na všetkých uzloch v systéme.

Bezpečnosť

Obzvlášť dôležitou témou v prípade distribuovaných aplikácií je bezpečnosť. Mnoho distribuovaných systémov je založených na komunikácii prostredníctvom verejných kanálov, akým je napríklad internet. V prostredí internetu pomerne často dochádza k neželanému vniknutiu do systému alebo neautorizovanej zmene informácie. Otázka bezpečnosti v prostredí distribuovaných systémov je komplexná téma.

2.1.5 Spolupráca a súčinnosť súčiastok

Distribuovaný systém je postavený na možnosti používať viaceré súčiastky, ktoré sa vykonávajú na rôznych softvérových a hardvérových platformách. Nevyhnutnosťou pri komunikácii medzi jednotlivými súčiastkami je súčinnosť zúčastnených súčiastok. Súčinnosť je založená na výmene správ medzi súčiastkami na základe stanovených konvencií. Súčinnosť súčiastok je silne závislá na ich zložitosti. Jednoduchšie súčiastky sú menej náchylné na chyby v komunikácii.

V závislosti od použitia konkrétneho spájajúceho softvéru, môže byť súčinnosť realizovaná viacerými spôsobmi ako:

- výmena správ,
- zdieľanie informácií,
- synchronizácia operácií.

Súčinnosť medzi súčiastkami silne závisí od zložitosti súčiastok a zložitosti celého systému. Systém postavený na jednom type hardvéru a softvéru je jednoduchší ako systém, ktorý spája súčiastky fungujúce na rôznych hardvérových a softvérových platformách. Preto pri analýze spolupráce medzi súčiastkami v rámci veľkého distribuovaného systému je potrebné zvážiť viaceré skutočnosti:

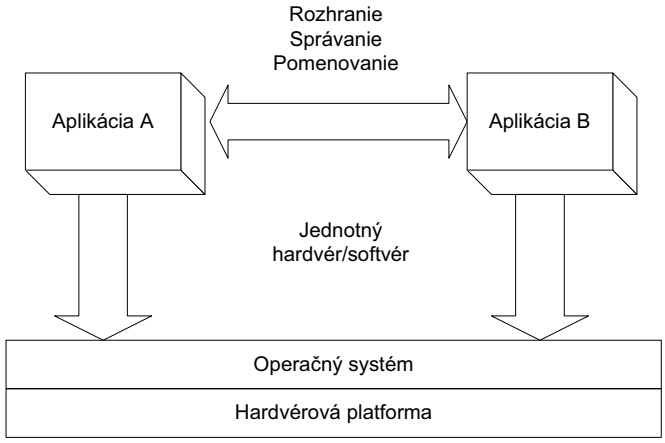
- spolupráca medzi aplikáciami,
- spolupráca medzi aplikáciou a operačným systémom,
- spolupráca medzi službou cieľového a zdrojového operačného systému,
- dohody ohľadom rozhraní, syntaxe a sémantiky,
- architektúra súčiastok a ich vzájomná spolupráca.

2.1.6 Hardvérové a softvérové prostredie

Distribuovanou aplikáciou môžeme nazvať aplikáciu, ktorá sa vykonáva na viacerých počítačoch a často na rozdielnych operačných systémoch. Nastáva viacero problémov,

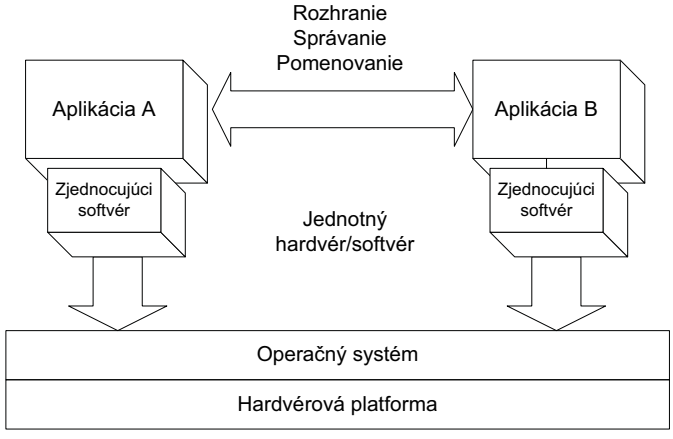
ktoré súvisia s používaním rozdielnych hardvérových a softvérových platforiem. Použitie jednotného hardvéru a softvéru je najjednoduchším spôsobom inštalácie distribuovaného systému.

Jednotlivé súčiastky sa volajú pomocou programovacieho rozhrania aplikácie. Avšak napriek použitiu jednotnej hardvérovej a softvérovej platformy môže dochádzať k nesúladu medzi súčiastkami na sémantickej úrovni – jedna súčiastka si napr. daný typ správy vysvetlí iným spôsobom ako druhá komunikujúca súčiastka. Obrázok 2-4 znázorňuje spoluprácu súčiastok na jednotnej softvérovej a hardvérovej platforme. V tomto prípade je potrebné sa zaoberať vhodným definovaním rozhraní, správania sa súčiastok a konvenciami pre pomenúvanie.



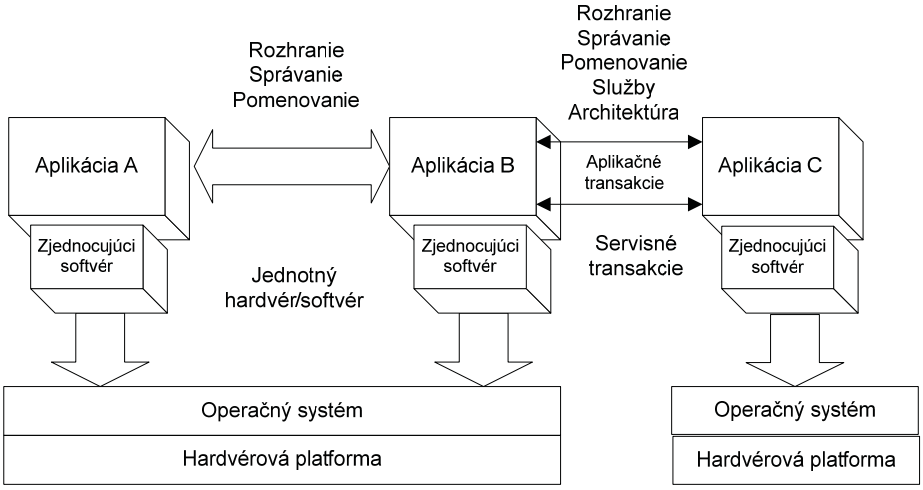
Obrázok 2-4. Spolupráca na rovnakom hardvéri a softvéri.

Zložitejším prípadom distribuovaného systému je, keď sa súčiastky vykonávajú na rozdielnych operačných systémoch. V tomto prípade je potrebné použiť zjednocujúci operačný systém (obrázok 2-5). Použitím zjednocujúceho operačného systému možno vytvárať prenosné aplikácie.



Obrázok 2-5. Spolupráca na rozdielnom softvéri.

Situácia sa ešte viac komplikuje v prípade, ak chceme distribuovanú aplikáciu nasadiť na rozdielnom softvéri aj hardvéri. V tomto prípade je preto potrebné nasadiť viacero druhov spomínaného zjednocujúceho operačného systému. V prípade použitia viacerých druhov zjednocujúcich operačných systémov je potrebné zaoberať sa komunikáciou na úrovni transakcií služieb a aplikácií (obrázok 2-6).



Obrázok 2-6. Spolupráca na rozdielnom hardvéri a softvéri.

2.1.7 Výhody a nevýhody distribuovaných systémov

Aj bez toho, aby sme vymenovali všetky výhody a nevýhody nasadenia distribuovaných systémov, je zrejmé, že práve snaha o distribuované riešenia dominuje svetu veľkých softvérových systémov. Ako dôvod možno uviesť potrebu distribuovaného spracovávania obrovského množstva dát a transakcií a tiež veľký pokrok v rámci dostupnosti softvérových a hardvérových riešení. Výhody:

- nárast výkonnosti,
- zlepšenie škálovateľnosti aplikácií/otvorenosť,
- zdieľanie zdrojov,
- odolnosť voči chybám,
- spoľahlivosť,
- nárast výpočtovej sily.

Nevýhody:

- oneskorenosť komunikácie,
- problémy synchronizácie,
- možnosť čiastkového zlyhania aplikácie,
- bezpečnosť,
- obmedzené výpočtové možnosti uzlov,
- distribúcia zdrojov.

2.2 Súčiastky založené na udalostiach

Súčiastky, ktoré pracujú na základe udalostí, patria k najvoľnejšie prepojeným prvkom softvéru. Nepotrebujú poznať s kým spolupracujú, stačí, keď sa dozvedia, že nastala udalosť, na ktorú dokážu zareagovať. Sú preto veľmi vhodné na prepájanie existujúcich aplikácií a to nie len na jednom počítači, ale aj v rámci počítačových sietí a to aj pri občasne pripojených počítačoch.

Svoje využitie však nachádzajú aj v kompaktných aplikáciách. Typicky je to podpora používateľského rozhrania, kde vzniká množstvo udalostí, na ktoré musí aplikácia reagovať. Iným príkladom sú riadiace systémy, ktoré reagujú na udalosti z výrobného alebo iného riadeného procesu.

Veľká voľnosť prepojenia umožňuje aj priamo počas vykonávania softvérového procesu meniť zdroje, ktoré udalosti vytvárajú, aj pozorovateľov, ktorí na udalosti reagujú. Ak pribudnú zdroje udalostí, jediné čo sa z pohľadu pozorovateľa zmení je počet udalostí, na ktoré má reagovať. Zdroju udalostí je zvyčajne tiež jedno, koľko pozorovateľov na jeho udalosť reaguje. Softvérové systémy, ktoré využívajú súčiastky založené na udalostiach, sú preto ľahko konfigurovateľné aj za chodu.

2.2.1 Udalosti

Udalosť je niečo, čo sa stane, o čom chceme vedieť a prípadne na to reagovať. V softvérových systémoch je udalosť definovaná ako špecifický vznik bodu interakcie dvoch výpočtových jednotiek. Tento bod interakcie sa chápe v čase, nie ako vytvorenie nejakého prepojenia.

Na udalosti je dôležité to, že vo všeobecnosti nevieme kedy nastane a či vôbec nastane. V konkrétnych prípadoch však zvyčajne vieme aspoň odhadnúť, ako často môže udalosť nastať. Udalosti, napríklad z časovača, môžu prichádzať v rozpätí mikrosekúnd, iné udalosti, napríklad vydanie opravenej verzie softvéru, sa vyskytujú raz za niekoľko mesiacov alebo aj rokov. Podľa toho sa potom vyberie vhodný model spracovania udalostí.

Z pohľadu softvérového systému možno udalosti členiť na vnútorné a vstupné. Vnútorné udalosti si vytvára systém sám a zvyčajne sa jedná o zmenu stavu niektorej jeho časti. Vstupné udalosti môžu byť vytvárané používateľmi prostredníctvom vstupných zariadení, napríklad stlačením znaku na klávesnici alebo pohybom myšou. Iné vstupné udalosti môžu byť generované externými alebo internými hardvérovými prostriedkami počítača a môžu ich tiež vytvoriť iné aktívne procesy na počítači. Toto rozlíšenie je dôležité preto, lebo na vstupné udalosti súčiastky zvyčajne nedokážu reagovať priamo a je potrebné udalosť prispôsobiť použitým súčiastkam.

Podľa (Faison, 2006) súčiastky, ktoré reagujú na udalosti, vyžadujú udalosti vo forme správy. Správa je vo všeobecnosti postupnosť znakov, ktoré určujú typ správy a jej parametre. Typ správy určuje typ udalosti, ktorá nastala a parametre správy bližšie určujú vlastnosti tejto udalosti. Súčiastka podľa typu správy zistí, či je to udalosť, na ktorú má reagovať, a ako má na ňu reagovať. Parametre správy môžu priamo niesť údaje, napríklad kód stlačeného znaku na klávesnici alebo referenciu na údaje, napríklad na súbor údajov.

Udalosti si vyžadujú asynchrónne spracovanie. Keď raz udalosť vznikla, tak jej tvorcu môže zaujímať, či bola nejako spracovaná, ale nemôže ovplyvniť kedy a či vôbec bude spracovaná. Preto by nemal čakať na potvrdenie ani o doručení, ani o spracovaní správy.

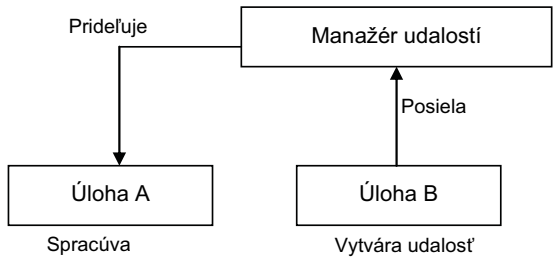
Ak potrebuje výsledky spracovania, je vhodné vytvorenie výsledku definovať ako novú udalosť. Synchrónne spracovanie udalostí možno využiť len ak sa jedná o vnútorné udalosti, kde je zaručená odozva. Vyvolanie udalosti je teda podobné volaniu procedúry s niekoľkými rozdielmi – nemusí byť známe kto obslúži udalosť, štandardne neexistuje návrat z udalosti a vyvolanie udalosti nie je určené na lokálne spracovanie.

2.2.2 *Reakcia na udalosti*

Keď je potrebné reagovať na vstupné udalosti, použije sa vzor Obaľovač alebo Zástupca. Dosiahne sa tak, že s vonkajším zdrojom udalostí môžeme pracovať ako so zdrojom vnútorným. Súčiastky, ktoré reagujú na vnútorné udalosti, sú potom vytvárané na základe návrhového vzoru Pozorovateľ (angl. *Observer*), ktorý je na obrázku 1-10 v kapitole 1. Zdroj udalostí je tu zobrazený ako pozorovateľný subjekt.

Každý pozorovateľ, ktorý chce prijímať udalosť, sa zaregistruje u zodpovedajúceho subjektu volaním jeho metódy `Pripoj`. Subjekt pri zmene svojho stavu metódou `Notifikuj` volá metódu `Aktualizuj` každého zaregistrovaného pozorovateľa. Pozorovateľ si následne vyžiada nový stav subjektu. Keď už nechce pozorovateľ prijímať ďalšie udalosti od tohto subjektu, odregistruje sa volaním metódy `Odpoj`.

Sprehľadnenie procesu posielania, rozdeľovania a spracovania udalostí umožňuje manažér udalostí, ktorého zapojenie do celého procesu je znázornené na obrázku 2-7.



Obrázok 2-7. Zapojenie manažéra udalostí do procesu šírenia udalostí.

Aj zdroj udalostí aj pozorovateľ sa registrujú v manažéri udalostí pre vybranú udalosť. Manažér potom zabezpečuje prevzatie každej udalosti a tiež, aby sa každá udalosť dostala ku každému zaregistrovanému pozorovateľovi. V niektorých systémoch manažér posiela aj spätnú správu od pozorovateľa ku zdroju, že pozorovateľ ukončil spracovanie udalosti. V takých systémoch je možné vytvoriť aj synchrónne spracovanie udalostí.

Synchrónne spracovanie udalostí sa používa len vtedy, keď je potrebné spracovať niektoré udalosti v presne určenom poradí. Na túto úlohu sa ale lepšie hodí vzor Postupnosť (angl. *Chain of Responsibility*), preto sa synchrónne spracovanie udalostí využíva zriedkavo.

2.2.3 *Model spracovania udalostí*

Každý programovací jazyk alebo vývojový systém, ktorý podporuje spracovanie udalostí, špecifikuje aj spôsob opisu a šírenia udalostí – model spracovania udalostí. Každý model

teda musí mať mechanizmus zabezpečujúci šírenie udalostí a musí jednoznačne špecifikovať, čo sa stane, keď vznikne udalosť.

Dva základné modely na spracovanie udalostí sa nazývajú *pull model* (model ťahania) a *push model* (model tlačenia).

Pull model

V pull modeli súčiastka, ktorá spracováva udalosti, požiada o novú udalosť hneď ako má predchádzajúcu udalosť spracovanú. Ak ďalšia udalosť nie je k dispozícii, súčiastka čaká určenú dobu a potom opakuje svoju žiadosť.

Hlavným problémom tohto modelu je, že ak vznikne viac udalostí počas spracovania predchádzajúcej, všetky okrem poslednej sa môžu stratiť. Tento problém čiastočne rieši vytvorenie radu udalostí. Ak sa rad preplní, systém musí vedieť akceptovať stratenie udalosti. Dĺžka radu je limitovaná z dvoch dôvodov. V systémoch sa často používa veľké množstvo typov správ a dlhý rad pre každý typ správy by spôsobil neúmernú spotrebu pamäti. Druhý dôvod je, že reagovať na udalosť má zvyčajne význam len do určitého času. Ak udalosť čaká príliš dlho, zastará a jej spracovanie už nemusí mať žiadny význam, len zbytočne spotrebuje výpočtové zdroje.

Manažér udalostí pre tento model je pomerne jednoduchý. V určených časoch prezerá všetky zaregistrované zdroje udalostí a keď niektorý vytvorí udalosť, uloží ju do príslušného radu. Keď si udalosť vyzdvihnú zaregistrované súčiastky, prípadne po stanovenom čase alebo po zaplnení radu, manažér túto udalosť z radu odstráni.

Udalosť pritom môže byť určená len jedinému z príjemcov alebo všetkým zaregistrovaným príjemcom. Ak je určená len pre jedného, vymaže sa z radu hneď ako sa odovzdá prvému, kto o ňu požiadal. Ak je určená pre všetkých, tak sa značí kto už udalosť prevzal a vymaže sa až s posledným odberom.

Udalosť pre jedného sa používa najmä vtedy, keď je viacero súčiastok na spracovanie rovnakej udalosti – záťaž sa rovnomerne rozdeľuje. Udalosť si prevezme tá súčiastka, ktorá práve skončila predchádzajúcu činnosť.

Dobrým príkladom na spracovanie udalosti pre každého je udalosť „vyšla upravená verzia softvérového systému“. Každá inštancia tohto softvérového systému si raz denne alebo raz za týždeň na zodpovedajúcom serveri skontroluje, či bola vytvorená novšia verzia. Ak bola vytvorená, môže sa nainštalovať, ak nie, softvér znovu pošle požiadavku po stanovenom čase. Ak bol nejaký počítač dlhší čas vypnutý, zvyčajne nevadí, že sa preskočili dve alebo tri verzie softvérového systému a nainštaluje sa najnovšia.

Tento príklad ukazuje aj výhody modelu. Drobnou výhodou je, že stačí implicitná registrácia pozorovateľa – vie kde sa má pýtať. Veľkou výhodou je jednoduchý manažér udalostí. Potrebuje len zabezpečiť, aby sa každá udalosť dostala do svojho radu a poskytuje rýchle odpovede na stav radu. Nemusí sa trápiť a opakovane vyzývať všetkých, ktorí zatiaľ nereagovali na udalosť.

Push model

V push modeli súčiastka, ktorá vytvára udalosť, pošle správu všetkým súčiastkam, ktoré sa prihlásili na jej spracovanie. Ak príjemca udalosti ešte nie je pripravený túto správu spracovať, správa sa stratí. Tento problém znovu môže čiastočne riešiť rad na správy, tentoraz na strane príjemcu.

Manažér udalostí je komplikovanejší, lebo, okrem prijatia udalosti, musí oznámiť každému zaregistrovanému prijímateľovi, že si má prevziať túto udalosť. Okrem toho samozrejme kontroluje, kto si už udalosť prevzal. Manažér má teda viac práce ako v prípade pull modelu. Zvlášť výrazné to je, keď potrebujeme prideliť udalosť práve jednému prijímateľovi. Manažér sa postupne a opakovane dopytuje všetkých prijímateľov, či sú pripravení prijať novú udalosť. Príliš veľa udalostí a dlhá doba odozvy na udalosť môžu zahltiť systém tak, že väčšina výpočtových zdrojov sa bude využívať len na rozdeľovanie správ.

Zahltenie manažéra udalostí sa dá riešiť vytvorením jeho distribuovanej verzie a najmä vhodným určením intervalu, ako často má opakovať dopyty na prijatie správy.

Výhodou tohto modelu je, že ho možno použiť aj keď nie sú k dispozícii súčiastky, ktoré si dokážu vyžiadať udalosť samé. Nevýhodou je už spomínaná vyššia spotreba výpočtových zdrojov a možnosť zahltenia.

Preto sa tento model používa najmä pri práci s riedkymi alebo vnútornými udalosťami. Súčiastky, ktoré súčasne vytvárajú aj prijímajú udalosti sú o niečo jednoduchšie ako pre predchádzajúci model, poznáme typickú dobu odozvy na udalosť a veľmi často je táto odozva kratšia než je čas príchodu ďalšej udalosti. Typický príklad je používateľské rozhranie, kde je odozva na udalosti o niekoľko rádov kratšia, než je hustota generovania udalostí používateľom.

2.2.4 Existujúce riešenia

Zaujímavé vlastnosti systémov vytvorených zo súčiastok založených na udalostiach, ako už spomínaná asynchrónnosť spracovania a veľká voľnosť prepojenia, spôsobili značné rozšírenie ich použitia v súčasných systémoch. Ani do budúcnosti nie je vidieť dôvod, prečo by mal záujem o ne ochabnúť.

Z veľkého množstva spôsobov použitia sú tu stručne opísané tri príklady. Jedná sa o programovací jazyk, webové služby a integračnú platformu.

Udalosti v jazyku Java

V programovacom jazyku Java sú udalosti reprezentované ako objekty. Odkazy na tieto objekty sú posielané ako správy, zodpovedajúce príslušným udalostiam.

Model udalostí je však v tomto jazyku prístupný len cez príslušné rozšírenie, nazvané *Advanced Windowing Toolkit – AWT*. Preto je potrebné vždy, keď chceme tento model využiť, importovať balíček `jawa.awt.event`. K dispozícii sú však aj novšie balíčky, s väčším počtom typov udalostí, napríklad `swing`. Jedná sa o podporu používateľského rozhrania, preto sa používa *push* model udalostí.

Vzor, podľa ktorého sú v jazyku Java vytvárané súčiastky sa nazýva Posluchač (angl. *Listener*). Je to v princípe ekvivalent už opísaného vzoru Pozorovateľ. Súčiastky pracujú v režime klient-server. Zdroj udalostí je serverom, u ktorého si klient registruje prijímanie zodpovedajúcej udalosti. Klient potom „počúva“, či mu server neposiela túto udalosť.

Z implementačného hľadiska je to navrhnuté tak, že server aj klient implementujú im zodpovedajúce rozhranie. V rámci programového kódu je potom špecifikované, na ktorú udalosť bude súčiastka reagovať.

Tento prístup poskytuje aj ďalšiu zaujímavú možnosť – delegovanie spracovania udalostí. Súčiastka môže odchytávať viacero typov udalostí, ale niektoré z nich nespracuje

a pošle ich nezmenené alebo ako nový typ udalosti ďalej. Udalosti je preto možné spracovávať podobne ako výnimky.

Webové služby

Pod službou si môžeme predstaviť nejakú činnosť, ktorá sa začne vykonávať keď sú splnené podmienky na jej aktivovanie. Splnenie zodpovedajúcich podmienok je teda udalosťou, ktorá aktivuje túto službu. Vytvorenie služby zapojením súčiastok, ktoré sú založené na udalostiach je preto úplne žiaduce.

Špecifikom webových služieb je, že webová komunikácia je štandardne bezstavová a jednosmerná. To znamená, že klient na jednej strane otvorí komunikáciu a pošle svoju požiadavku. Služba (server) na druhej strane odpovie a tým sa aj komunikácia ukončí. Neudržuje sa žiadna informácia či komunikácia prebieha, kto je na rade a podobne. Služba tiež nemá možnosť poslať neskôr nejakú dodatočnú informáciu, ani vyzvať klienta aby si novú informáciu vyžiadal.

Tento spôsob komunikácie je jednoduchý a postačuje, ak má služba všetky odpovede pripravené alebo ich vie v krátkom čase (jednotky sekúnd) spočítať. Ak ale služba potrebuje na získanie odpovede viac času, napríklad niekoľko minút alebo aj hodín, klient vyhodnotí situáciu ako poruchu spojenia a komunikáciu ukončí. Služba potom nemá ako informovať klienta, že riešenie je už k dispozícii.

Webová služba aj klient preto využívajú vzor Pozorovateľ „obojsmerným“ spôsobom prostredníctvom `callback` procedúry. Na začiatku sa služba zaregistruje v rámci systému, kde bude dostupná a nastaví sa do režimu pozorovateľa. Služba je štandardne opísaná prostredníctvom *Web Services Description Langague – WSDL*. Klient ju teda môže nájsť a poslať jej svoju požiadavku. Okrem toho pošle službe dodatočnú „*callback*“ informáciu – kde a ako ho možno aktivovať aby prijal odpoveď. Sám sa potom prepne do režimu pozorovateľa a zaregistruje si túto službu ako zdroj udalosti. Keď služba dokončí svoju prácu, na základe uvedenej dodatočnej informácie otvorí komunikáciu s klientom a odovzdá mu svoje riešenie. Potom sa znovu prepne do režimu pozorovateľa a je pripravená prijať ďalšiu úlohu.

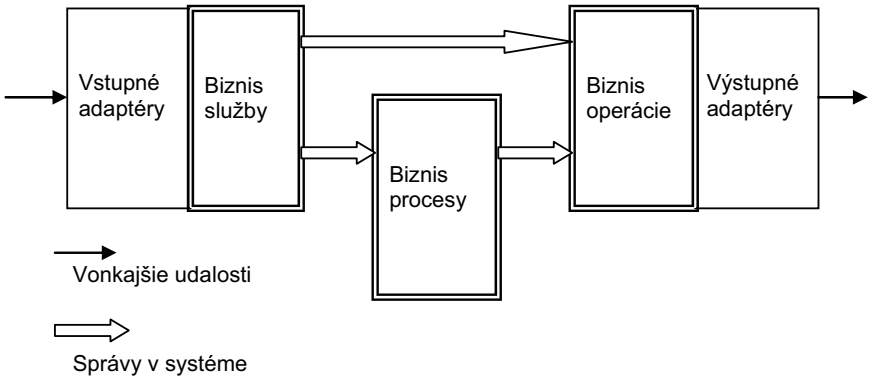
Integračná platforma Ensemble

Mnohé organizácie potrebujú pracovať s viacerými informačnými systémami, ktoré sú takmer vždy nejakým spôsobom nekompatibilné. Pritom priame prepojenie týchto informačných systémov by výrazne zvýšilo efektívnosť práce – množstvo údajov by nebolo potrebné kopírovať z jedného systému do druhého, ale stačilo by sa na ne odkazovať.

Rozličné štandardy, databázové systémy, programovacie jazyky, ba aj operačné systémy, nad ktorými tieto informačné systémy pracujú, však spôsobujú značné problémy s ich prepojením. Keď k tomu pridáme možnosť, že sa každý z pripojených systémov môže nezávisle vyvíjať, zdá sa byť vytvorenie dlhodobo funkčnej integrácie nemožné. A s týmito problémami sa snažia vyrovnať takzvané integračné platformy, ktoré majú za úlohu prepojiť rôzne systémy do jedného funkčného celku.

Na obrázku 2-8 vidíme zjednodušený pohľad na platformu Ensemble. Všetky entity, označené slovom biznis zodpovedajú súčiastkam založeným na udalostiach. Na začiatku sú vstupné adaptéry, ktoré transformujú rôzne štandardy a protokoly z pripojených informačných systémov do jednotného vnútorného systému. Na ne sú priamo napojené

biznis služby, ktoré už prijímajú udalosti ako správy a z vonku to naozaj funguje ako volanie služby.



Obrázok 2-8. Prepojenie súčiastok pomocou správ v systéme Ensemble.

Biznis procesy realizujú celú prepojovaciu logiku a biznis operácie zabezpečujú spracovanie jednotlivých úloh. Výstupné adaptéry potom výsledné riešenie prispôsobia jednotlivým pripojeným systémom.

Celé riešenie je založené na udalostiach, posielaných prostredníctvom správ. Všetky prepojenia je tak možné nezávisle meniť, pridávať a uberať. Tento prístup tak umožňuje nielen rýchlu integráciu ale aj efektívnu údržbu celého prostredia.

2.3 CORBA

Významným problémom, ktorý musí byť pri vývoji distribuovaných aplikácií vyriešený, je otázka vzájomnej komunikácie jednotlivých častí takýchto aplikácií. Predstavme si napríklad, že v module `A` vytváranej aplikácie potrebujeme zavolať funkciu modulu `B`, ktorý je vykonávaný na inom počítači než modul `A`, a získať späť výsledok tohto volania. Nech volaná funkcia je `add` s dvoma celočíselnými parametrami. Napriek výhodám vyššieuvedeného prístupu založeného na udalostiach sa rozhodneme použiť mechanizmus pripomínajúci tradičné synchrónne volanie procedúry. Otázkou však je – ako ho implementovať?

Jednou z možností je použitie štandardných programovacích rozhraní k službám sieťových komunikačných protokolov, ako je napríklad *Berkeley Sockets API* pre rodinu protokolov TCP/IP. Výhodou týchto rozhraní je ich univerzálna dostupnosť, nakoľko sú k dispozícii prakticky pre všetky operačné systémy; ich hlavnou nevýhodou je pomerne nízka úroveň abstrakcie, z ktorej vyplýva najmä nutnosť vytvárania väčšieho množstva kódu potrebného na ich použitie. V našom prípade by sme museli na strane modulu `A` naprogramovať nasledujúce kroky:

1. otvorenie spojenia – volaním funkcií `gethostbyname`, `socket`, `connect`,
2. príprava správy – vytvorenie balíka údajov obsahujúceho napríklad informácie „add“, 20, 30 (t.j. názov funkcie a jej parametre),
3. poslanie správy – volaním funkcie `write`,
4. prečítanie odpovede – volaním funkcie `read`,

5. interpretácia odpovede – extrakcia návratovej hodnoty, prípadne chybovej správy,
6. zatvorenie spojenia – volaním funkcie `close`.

Na strane modulu B by bola situácia analogická. Okrem zložitosti vytvorených programov má takéto riešenie aj ďalšie nevýhody, konkrétne:

1. V prípade, že volajúci a volaný modul sú spustené na počítačoch s rôznou architektúrou, nemusia byť vymieňané údaje správne interpretované. Ak by sa napríklad `A` vykonával na počítači s procesorom ukladajúcim najvýznamnejší bajt najprv („big endian“, napríklad PowerPC), volanie `add (20, 30)` by mohlo byť zakódované do správy ako sekvencia oktetov (‘a’, ‘d’, ‘d’, ‘\0’, 0, 0, 0, 20, 0, 0, 0, 30). Takáto správa prijatá na počítači s procesorom ukladajúcim najvýznamnejší bajt na konci („little endian“, napríklad Intel x86) by bola interpretovaná ako volanie `add (335544320, 503316480)`. Analogické problémy môžu vznikať tiež pri interpretácii čísel s pohyblivou rádovou čiarkou alebo textových reťazcov (z dôvodu rôznych spôsobov kódovania znakov: ASCII, EBCDIC, UNICODE, rôzne kódovania národných abecied a podobne).
2. V prípade, že potrebujeme zabezpečiť ďalšie služby, ako je napríklad bezpečnosť informácií pri prenose (typicky použitím protokolu SSL/TLS[1]), prípadne korektné správanie sa systému pri nedostupnosti počítača s volaným modulom, je objem „infraštruktúrneho kódu“ ešte omnoho vyšší.
3. Podobne v prípade, že volané rozhranie je zložitejšie, čo sa týka množstva funkcií, počtu ich parametrov a prenášaných dátových typov, objem práce programátora pri kódovaní a dekódovaní volaní funkcií do formy údajových balíkov rýchlo narastá.

Z týchto a podobných dôvodov sa pri volaní funkcií v distribuovanom prostredí používa elegantná myšlienka volania vzdialených procedúr (angl. *remote procedure call*), ktorú prvýkrát uviedli do praxe Birrell a Nelson (Birrell, 1984). Základným princípom je, že programátor so vzdialenými procedúrami resp. funkciami pracuje tým istým spôsobom ako s lokálnymi – samozrejme s výnimkou aspektov výkonu a spoľahlivosti.

2.3.1 *Volanie vzdialených procedúr*

Typické použitie vzdialeného volania procedúr je znázornené na obrázku 2-9. *Stub*[2] pre `B` je modul, ktorý umožňuje komunikáciu klientov s modulom `B`. Pre klienta – v našom prípade pre modul `A` – poskytuje funkciu `add (x,y)`, pričom ju neimplementuje priamo, ale pri každom jej volaní vytvára balík údajov, ktorý prostredníctvom komunikačného jadra posiela serveru. Na strane servera balík prevezme tzv. *skeleton* pre B, ktorý z prijatých údajov extrahuje názov funkcie a jej parametre a túto funkciu zavolá. Návratovú hodnotu pošle naspäť modulu *stub*, ktorý ju odovzdá klientovi.

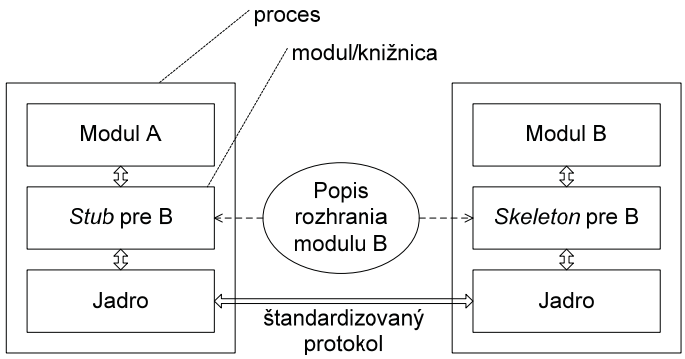
Pre programátora je dôležité, že jadro je spravidla k dispozícii vo forme knižnice a moduly *stub* a *skeleton* sú automaticky generované z opisu rozhrania volaného modulu

---

[1] Secure Sockets Layer resp. Transport Layer Security

[2] Slovo *stub* v angličtine označuje peň, pahýľ. Niekedy sa pre tento modul používa aj výraz *proxy* (zástupca). Na strane servera sa analogický modul nazýva *server-side stub* alebo aj *skeleton* (kostra).

vhodným predkompilátorom – znamená to, že programátor musí vytvoriť len opis rozhrania volaného modulu, volaný a volajúci modul. „Infraštruktúrnym kódom“ sa zaoberať nemusí.



Obrázok 2-9. Volanie vzdialených procedúr.

Myšlienka vzdialeného volania procedúr má veľké množstvo realizácií, spomeňme najvýznamnejšie z nich:

1. mechanizmy na tradičné vzdialené volanie procedúr, ako sú napríklad ONC RPC (Open Network Computing RPC; Srinivasan, 1995) alebo DCE RPC (Distributed Computing Environment RPC[3]),
2. architektúra CORBA,
3. technológia Java Remote Method Invocation,
4. webové služby, ktoré je tiež možné použiť na volanie vzdialených procedúr.

Väčšina z týchto technológií poskytuje vysokú mieru nezávislosti komunikujúcich súčiastok: moduly pre klient a server môžu byť vytvorené v rôznych programovacích jazykoch a môžu byť prevádzkované na počítačoch s rôznou hardvérovou architektúrou a operačným systémom.

2.3.2 *Architektúra CORBA*

CORBA (Common Object Request Broker Architecture) poskytuje otvorenú infraštruktúru, ktorá slúži na zaistenie spolupráce aplikácií, resp. ich častí v prostredí počítačovej siete (Object Management Group, 2009). Je implementáciou myšlienky volania vzdialených procedúr, a to v prostredí objektovo orientovaného prístupu k vývoju softvéru – konkrétne to znamená, že volané entity nie sú funkcie, ale operácie, resp. metódy vzdialených objektov.

CORBA je dielom organizácie Object Management Group (OMG), ktorá je významným neziskovým združením viac než 400 spoločností pôsobiacich v oblasti informačných technológií – či už ide o poskytovateľov riešení, ich zákazníkov alebo o akademické inštitúcie. Teraz si ukážeme použitie architektúry CORBA na jednoduchom príklade.

[3] `http://www.opengroup.org/dce/`

Predpokladajme, že potrebujeme sprístupniť objekty realizujúce počítadlo implementované triedou `Counter` pre klientov v distribuovanom prostredí.

```
class Counter
{
  private int value;
  public int getValue()
  {
    return value;
  }
  public void setValue(int v)
  {
    value = v;
  }
  public void increment()
  {
    value++;
  }
}
```

Príklad 2-1. Zdrojový kód triedy Counter.

Prvým krokom je vytvorenie opisu rozhrania pre počítadlo. Na opis rozhraní CORBA – podobne ako väčšina technológií na volanie vzdialených procedúr – používa špecializovaný jazyk s názvom CORBA IDL (Interface Definition Language). Ide o jazyk vychádzajúci zo syntaxe C++ s podporou preddefinovaných aj používateľských dátových typov, postupností, polí, dedenia, výnimiek a vstupných i výstupných (in/out) parametrov. Opis rozhrania pre počítadlo je veľmi jednoduchý:

```
interface Counter
{
  long getValue();
  void setValue(in long value);
  void increment();
};
```

Príklad 2-2. Opis rozhrania počítadla.

Tento opis rozhrania je vstupom pre predkompilátor, ktorý vygeneruje niekoľko súborov, realizujúcich *stub* a *skeleton* pre počítadlo.

Aby sme triedu `Counter` mohli použiť ako implementáciu objektu v distribuovanej aplikácii, musíme ju trochu upraviť (zmeny sú vyznačené tučným písmom):

```
class CounterImpl extends CounterPOA
{
  // ostatne casti triedy ostali bez zmeny
}
```

Príklad 2-3. Upravený zdrojový kód triedy Counter.

Okrem zmeny názvu triedy (nutnej pre jej odlíšenie od abstraktného rozhrania `Counter`) bola pridaná väzba implementačným dedením na vygenerovaný *skeleton* obsiahnutý v triede `CounterPOA`.

Nasledujúci kód na strane klienta ilustruje princíp transparentného prístupu k vzdialeným procedúram resp. metódam:

```
public class CounterClient
{
  public static void main(String argv []) throws Exception
  {
    // tu musime ziskat referenciu na pocitadlo
    String ior = ...;

    // inicializacia ORB na strane klienta
    ORB orb = ORB.init(argv, null);

    // prevedenie externej formy (IOR) na lokalnu referenciu
    org.omg.CORBA.Object obj = orb.string_to_object(ior);
    Counter c = CounterHelper.narrow(obj);

    // dalej pracujeme s lokalnou referenciou (c)
    c.setValue(0);
    for (int i = 0; i < 1000; i++)
      c.increment();
    System.out.println("Vysledok: " + c.getValue());
  }
}
```

Príklad 2-4. Zdrojový kód klienta objektu Counter.

Na začiatku je potrebné získať referenciu na počítadlo, s ktorým budeme pracovať. Pri bežnej „nedistribuovanej" aplikácii by sme takúto referenciu získali vytvorením novej inštancie počítadla[4], v tomto prípade musíme referenciu získať inak – napríklad pomocou služby Naming Service, ku ktorej sa vrátime neskôr. Ďalšie riadky kódu predstavujú inicializáciu implementácie CORBA a vytvorenie inštancie zástupcu (t.j. *stub*) pre počítadlo.

Kód vyznačený tučným písmom už pracuje priamo so zástupcom, takže programátor sa môže plne venovať aplikačnej logike bez nutnosti riešenia otázok komunikácie v distribuovanom prostredí.

Poznámka: Okrem tu uvedených modulov je potrebný ešte server, ktorý vytvorí a zaregistruje inštanciu počítadla. Ide opäť o niekoľko riadkov kódu, pre pochopenie myšlienky architektúry CORBA však nie sú podstatné.

2.3.3 Ďalšie možnosti a služby v rámci architektúry CORBA

Okrem už naznačených možností poskytuje architektúra CORBA mnohé ďalšie, najmä:

1. Napriek tomu, že najčastejším prípadom je volanie známych a vopred definovaných rozhraní (čo umožňuje použitie predkompilátora generujúceho kód pre zástupcov na strane klienta), CORBA poskytuje možnosť volať aj rozhrania definované *až v čase vykonávania klienta*. Prostredníctvom špecializovaného rozhrania (Dynamic Invocation Interface) môže klient postupne špecifikovať názov operácie, ktorá má byť zavolaná (napr. „setValue") a hodnoty jednotlivých parametrov (napr. 100) a následne operáciu zavolať. Princíp je podobný dynamickému volaniu metód objektov, napr. prostredníctvom Java Reflection API a je vhodný vtedy, keď konkrétne rozhranie, ktoré sa má zavolať, nie je v čase kompilácie klienta známe.

[4] Napríklad takto: Counter c = new CounterImpl()

2. Analogický mechanizmus (Dynamic Skeleton Interface) existuje aj na strane servera a umožňuje prijať ľubovoľné volanie a až v čase vykonávania zistiť, ktorá metóda a s akými parametrami bola volaná.
3. Dôležitou súčiastkou na strane servera je tzv. objektový adaptér (angl. *Object Adapter*), ktorý umožňuje zachovať voľnú väzbu medzi objektmi viditeľnými pre klientov a implementačným kódom, ktorý môže, ale nemusí byť objektovo orientovaný.

Napriek tomu, že volanie vzdialených funkcií, resp. metód, je dôležitou otázkou v distribuovaných aplikáciách, nie je jedinou: takéto aplikácie musia riešiť množstvo ďalších problémov týkajúcich sa napríklad vyhľadávania objektov alebo bezpečnosti.

S cieľom uľahčiť vývoj aplikácií sú súčasťou architektúry CORBA viaceré podporné služby. Ako príklad uveďme služby v oblasti vyhľadávania objektov podľa mena (Naming Service), resp. podľa vlastností (Trading Object Service), služby v oblasti zaistenia bezpečnosti (Security Service), správy transakcií (Transaction Service), asynchrónnej komunikácie (Event Service, Notification Service), perzistencie objektov (Persistent State Service) a ďalších.

Od verzie 3.0 uvoľnenej v roku 2002 je súčasťou architektúry CORBA aj model súčiastok (tzv. CORBA Component Model, skrátene CCM), ktorý je platformovo nezávislým rozšírením modelu súčiastok Enterprise JavaBeans. CCM umožňuje pre súčiastky definovať poskytované aj požadované rozhrania, a to synchrónne (volanie metód) aj asynchrónne (posielanie správ). Podobne ako súčiastky v iných modeloch, aj súčiastky CCM majú definované používateľom nastaviteľné parametre; a podobne ako súčiastky Enterprise JavaBeans, sú aj súčiastky CCM rôznych typov (service, session, entity, process) a sú prevádzkované v špecializovaných kontajneroch.

2.3.4 Zhodnotenie

CORBA predstavovala v 90-tych rokoch sľubnú technológiu, používanú v mnohých softvérových projektoch. V súčasnosti sa o nej hovorí pomerne málo a existujú protichodné názory na jej úspech resp. neúspech – napr. (Henning, 2006), (Schmidt, 2009). Vo všeobecnosti sa dá povedať, že napriek tomu, že v roli „univerzálnej komunikačnej zbernice“ ju – aspoň v niektorých doménach – vystriedali webové služby, ide o zrelú technológiu, ktorá má desiatky komerčných aj voľne dostupných implementácií (Puder, 2009) a veľké množstvo použití v rôznych aplikačných oblastiach.

2.4 JavaBeans

JavaBeans je názov platformovo-nezávislého súčiaskového modelu založeného na jazyku Java. Jeho pomocou môžeme vytvárať znovupoužiteľné prenosné softvérové súčiastky. Model bol vyvinutý s ohľadom na (vizuálne) autorské zostavovateľské nástroje, resp. pracovné rámce, ktoré umožňujú jednotlivé súčiastky prepájať s cieľom vytvorenia zložených súčiastok alebo samostatnej aplikácie.

Elementom súčiastkového modelu je tzv. JavaBean, v preklade *bôb* jazyka Java (v kontexte kávovej metafory týkajúcej sa celej platformy). Bôbmi sú vizuálne súčiastky – napr. tlačidlo, kalkulačka, databázový zobrazovač ale i nevizuálne súčiastky – napr. kontrola pravopisu. Koncepcia súčiastkového modelu stojí na týchto bodoch:

- *introspekcia* – možnosť odhalenia vlastností bôbu (t.j. jeho atribútov, metód a udalostí).
- *atribúty* – každý bôb má atribúty, ktoré ho charakterizujú a ktoré možno meniť v čase návrhu.
- *metódy* – každý bôb má metódy, pomocou ktorých možno bôbom manipulovať.
- *udalosti* – pomocou udalostí sa dá komunikovať – posielať a prijímať správy medzi bôbmi.
- *upravovanie* – vzhľad, resp. správanie bôbu môže byť v čase návrhu upravované.
- *perzistencia* – každý bôb môže byť perzistovaný, t.j. po upravovaní možno jeho stav zachovať pre použitie v iných zostavovateľských nástrojoch.

Je zrejmá podobnosť s objektom ako elementárnym prvkom objektovo-orientovanej paradigmy, nakoľko z nej vychádza aj samotný programovací jazyk Java. Model JavaBeans nestavia žiadne ďalšie reštrikcie na svoje súčiastky. Nie je potrebné, aby bôby podliehali definovaným rozhraniam, generalizovali iné súčiastky alebo obsahovali dodatočné metadáta (napr. pomocou tzv. anotácií). Aby bôby spĺňali uvedené požiadavky, pri ich tvorbe je potrebné riadiť sa niekoľkými konvenciami:

- trieda bôbu musí obsahovať verejný bezparametrový konštruktor, ktorý umožní jej jednoduchú inštanciáciu v autorských nástrojoch.
- atribúty triedy sú manipulovateľné pomocou prístupových, resp. zmenových metód (angl. *accessor, mutator methods*), ktoré zodpovedajú vzoru:

  ```
  getXXX()
  setXXX(...)
  ```

  čím je zaručená možnosť introspekcie a manipulovateľnosť bôbmi v rámci použitých autorských nástrojov,
- trieda bôbu by mala byť serializovateľná. To umožní autorským nástrojom spoľahlivo perzistovať a pristupovať k stavom bôbu nezávisle od platformy, kde je použitý.

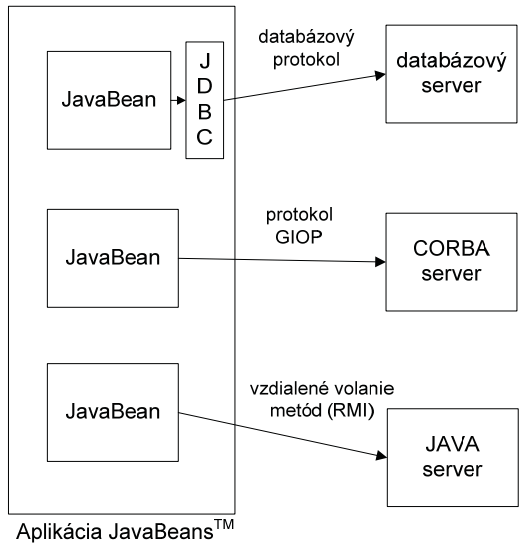
Základným rozdielom oproti platformovo-závislým prvkom, akými sú OLE (Object Linking and Embedding) alebo ActiveX, je fakt, že JavaBeans definuje rozhranie súčiastky už v čase návrhu, t.j. pri zostavovaní novej súčiastky alebo aplikácie autorským nástrojom.

Spolu s JavaBeans Activation Framework (JAF), aktivačným pracovným rámcom pre bôby jazyka Java, je bôb možné vytvoriť z ľubovoľného typu dát. JAF ponúka služby pre identifikáciu typu dát, zapuzdrenie prístupu, odhalenie metód a inštanciáciu, napr. obrázku vo formáte JPEG.

Na použitie bôbov jazyka Java v distribuovanom prostredí sú tri mechanizmy prístupu (Sun Microsystems, 1997; obrázok 2-10).

Použitím tzv. Java RMI (Remote Method Invocation), vzdialeného volania metód, je podporované distribuované spracovanie v prostredí jazyka Java. Je vytvorený priestor pre automatické a transparentné doručenie vzdialených volaní všade, kde je prítomný virtuálny stroj jazyka Java (Java Virtual Machine; JVM).

Pomocou Java IDL (Interface Description Language) ako implementácie špecifikácie CORBA a GIOP (General Inter-ORB Protocol), protokolu pre výmenu správ medzi zástupcami vzdialených objektov, možno realizovať interoperabilitu v heterogénnom distribuovanom prostredí.



Obrázok 2-10. Mechanizmy vzdialeného prístupu.

Pre prístup do databázového úložiska je implementovaná podpora pomocou rozhrania JDBC (Java DataBase Connectivity). Súčiastky modelu JavaBeans môžu poskytovať prispôsobený prístup ku konkrétnym tabuľkám.

2.4.1 Enterprise JavaBeans

Enterprise JavaBean (EJB) je súčiastkový model založený na jazyku Java zameraný na komplexné podnikové systémy. Bol vyvinutý s cieľom rýchleho a jednoduchého vývoja distribuovaných, transakčných, bezpečných a prenosných aplikácií založených na platforme Java.

Model EJB bol vytvorený pre tzv. „klient-server" prostredie. Špecifikuje architektúru na strane servera, ktorá zapuzdruje doménovú logiku aplikácie. Súčiastky EJB, označované rovnomenne podľa modelu, pôsobia v tzv. kontajneroch EJB, ktoré sú súčasťou aplikačných serverov modelu EJB (obrázok 2-11). Kontajnery EJB sa starajú o životný cyklus bôbov a sú prostredníkom medzi doménovou logikou aplikácie a prostredím aplikačného servera. Služby kontajnera EJB zahŕňajú:

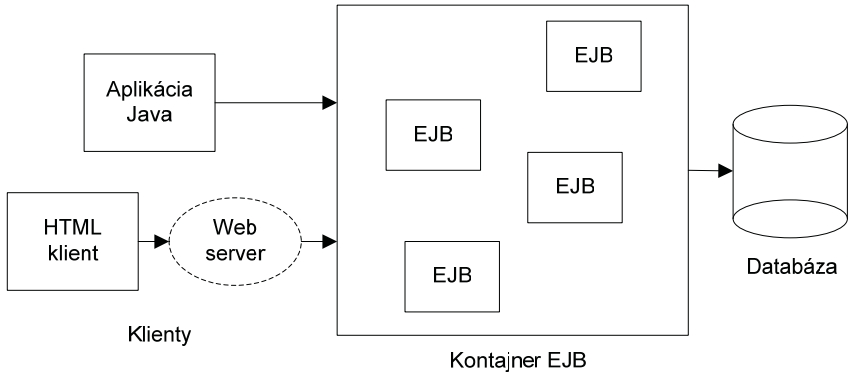
- správu transakcií pri prístupe do úložiska: začatie, odosielanie a späťvzatie,
- udržiavanie zdieľaných inštancií bôbov pre spracovanie prichádzajúcich požiadaviek a prepínanie medzi aktívnym a neaktívnym stavom bôbu,
- synchronizáciu atribútov bôbov s ich perzistovaným stavom.

V modeli EJB pôvodne existovali 3 typy súčiastok: entitné bôby (angl. *entity beans*), bôby sedenia (angl. *session beans*) a správami riadené bôby (angl. *message-driven beans*).

Entitný bôb/perzistentná entita

Tento typ EJB bol používaný vo verziách súčiastkového modelu nižších ako verzia 3.0. Reprezentoval distribuované objekty, ktoré mali perzistentný stav uložený v databáze. Rozlišovali sa dva podtypy, na základe toho, kto riadil tento stav. Buď išlo o kontajnerom

manažovanú perzistenciu (angl. *Container-Managed Persistence*; CMP) alebo o bôbom manažovanú perzistenicu (angl. *Bean-Managed Persistence*; BMP).



Obrázok 2-11. Schematický náčrt kontajnera EJB v architektúre modelu EJB.

Entitné bôby boli s príchodom EJB verzie 3.0 nahradené tzv. Java Persistence API (JPA). To maže komplikované a obmedzené ponímanie perzistencie a perzistentnou entitou umožňuje byť akejkoľvek odľahčenej triede jazyka Java, ktorá zvyčajne reprezentuje tabuľku v relačnej databáze. Inštanciami tejto triedy sú jednotlivé riadky tabuľky. Relačné vzťahy v databáze sú vyjadrené použitím metadát – napr. pomocou anotácii priamo v programovom kóde alebo explicitne pomocou XML opisu.

Bôb sedenia

Na rozdiel od entitného bôbu, ktorý reprezentuje perzistentné dáta, bôb sedenia reprezentuje určitú doménovú úlohu a je spravovaný EJB kontajnerom. Je vytvorený na požiadanie klienta a existuje len počas jedného sedenia. Okrem realizácie doménovej úlohy môže slúžiť na zriadenie prístupu do databázy. Existujú dva typy bôbov sedenia: stavové a bezstavové. Stavový bôb sedenia (angl. *Stateful Session Bean*) uchováva stav sedenia a v prípade jeho odstránenia z pamäti jeho životný cyklus riadi EJB kontajner. Príkladom takéhoto bôbu je napr. nákupný košík známy z webového prostredia. Bezstavový bôb sedenia (angl. *Stateless Session Bean*) naopak nemá asociovaný stav sedenia, preto prístup k nemu môže byť konkurentný.

Správami riadený bôb

Tento typ súčiastky reprezentuje integráciu JMS (Java Message Service) – služieb pre voľne zviazanú distribuovanú komunikáciu – do modelu EJB. Bol predstavený vo verzii EJB 2.0 a slúži na spracovanie asynchrónnych správ.

2.5 System Object Model

Súčiastkové (alebo komponentové) technológie sa snažia riešiť problémy spojené s používaním súčiastok ako sú definovanie a opis súčiastok, rozhraní a interakcií medzi nimi a tiež spôsob ich zabalenia a znovupoužitia.

Spoločnosť IBM vytvorila technológiu System Object Model (SOM) (Dansforth, 1994), ktorá adresuje veľkú časť nedostatkov, ktoré prišli s objektovo orientovanými jazykmi. SOM obsahuje flexibilný objektový model, ktorého dve hlavné výhody sú binárna kompatibilita novších verzií knižnice so staršími a možnosť znovupoužitia knižnice objektov v iných (aj procedurálnych) programovacích jazykoch bez ohľadu na jazyk, v ktorom bola knižnica vytvorená.

SOM umožňuje vývojárom implementovať objekty v ich preferovanom programovacom jazyku, pričom výstupy budú použiteľné aj v ostatných jazykoch. Objektový model SOM je oddelený od objektových modelov konkrétnych programovacích jazykov a umožňuje tak pridať objektové mechanizmy aj do procedurálnych jazykov; existujú implementácie pre C a Cobol.

2.5.1 Problémy a ciele

Objektovo orientované programovacie jazyky prinášajú výhody oproti procedurálnym jazykom, najmä vďaka zapuzdreniu dát a operácií nad nimi. Vznikajú ale aj nové problémy súvisiace s binárnou kompatibilitou predchádzajúcich verzií knižníc a zdieľaním objektov medzi rôznymi programovacími jazykmi (Hamilton, 1996).

V procedurálnom jazyku postačuje, aby nová verzia knižnice zachovala kompatibilné signatúry funkcií a aby pridané nové funkcie nekolidovali s menami v klientskej aplikácii. Udržiavať kompatibilitu definícií tried v objektovo orientovaných jazykov ale nie je také ľahké. V statických jazykoch (napr. C++, Java) je v klientskom kóde skompilované množstvo informácií o použitej triede (veľkosť, poradie a umiestnenie metód, offset[5] od rodičovskej triedy) a aj malá zmena vo vnútornej štruktúre triedy spôsobí nutnosť prekompilovať klientsky kód. V dynamických jazykoch (napr. Smalltalk), ktoré informácie o triedach spravujú počas vykonávania programu, tieto problémy nevznikajú.

Ďalší problém je spôsobený zdieľaním kódu medzi jazykmi, keď v prípade objektovo orientovaných programovacích jazykov neexistuje žiadna štandardná reprezentácia objektov, ktorá by umožnila používať objekty napr. zo Smalltalku v C++.

Problémy nastávajú už aj medzi rôznymi kompilátormi C++, kde každý môže používať rozličné zarovnanie objektov v pamäti a použitie objektovej knižnice vytvorenej v inom kompilátore sa tak stáva neschodné. Naopak, v procedurálnych jazykoch linkovací proces v operačnom systéme dodržuje určité konvencie, čo umožňuje volať knižnice medzi rôznymi procedurálnymi jazykmi (napr. C, Fortran) relatívne jednoducho.

Technológia SOM sprístupňujúca objekty bola navrhnutá ako riešenie obmedzení pre širšie použitie knižníc objektových tried (IBM, 1994) s nasledujúcimi cieľmi:

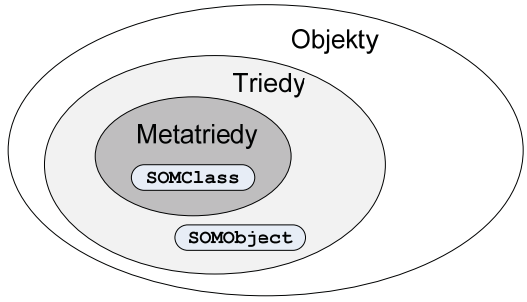
1. Možnosť odvodiť novú objektovú triedu z binárneho tvaru knižnice, bez nutnosti mať k dispozícii pôvodný zdrojový kód základnej triedy.
2. Možnosť používať objektové triedy a odvodzovať nové triedy bez ohľadu na programovací jazyk, v ktorom bola objektová knižnica alebo klientska aplikácia implementovaná.
3. Umožniť následné zmeny (vylepšenia, opravy) vo vytvorených súčiastkach bez nutnosti rekompilácie existujúcich klientskych aplikácií, ktoré ich používajú.

[5] Posun v skompilovanom (binárnom) kóde.

V nasledujúcich častiach opíšeme architektúru SOM, jej vlastnosti vzhľadom na vyššie uvedené ciele, spôsob tvorby objektov a ich zdieľanie medzi rôznymi programovacími jazykmi.

2.5.2 SOM objekty

SOM oddeľuje rozhranie objektov od ich implementácie pomocou jazykovo nezávislého objektového modelu. Knižnica a (klientska) aplikácia používajúca túto knižnicu tak môžu byť implementované v rôznych programovacích jazykoch a pokiaľ zmena nevyžaduje úpravu zdrojového kódu klientskej aplikácie, nová verzia triedy sa môže nasadiť bez nutnosti rekompilácie klientskej aplikácie.



Obrázok 2-12. Hierarchia objektov v SOM.

Existujú tri typy objektov v SOM: inštancie tried (objekty, ktoré nie sú triedami), triedy (inštancie metatried; objekty, ktoré nie sú metatriedami) a metatriedy (obrázok 2-12).

Každý SOM objekt je odvodený od základného objektu `SOMObject`, ktorý definuje správanie spoločné pre všetky SOM objekty; okrem iného definuje metódu `somDispatch`, ktorá poskytuje všeobecný spúšťací mechanizmus pre volanie metód na objektoch (angl. *method dispatch mechanism*).

Trieda je odlišná od obyčajného objektu (inštancie triedy) tým, že obsahuje tabuľku metód definujúcu metódy, ktoré môžeme na inštanciách tejto triedy volať. Počas inicializácie triedy SOM prostredie spustí inicializačnú metódu, ktorá informuje triedu o jej rodičovských triedach a umožní jej tým inicializovať si vlastnú tabuľku metód. Následne pokračuje inicializácia metódami, ktoré preťažia zdedené metódy alebo pridajú do triedy nové metódy.

V hierarchii je objekt `SOMObject` trieda, ktorá je inštanciou metatriedy `SOMClass` (tiež objektu). Všetky metatriedy v SOM sú eventuálne zdedené z metatriedy `SOMClass`, ktorá do zdedených objektov pridáva metódu `somNew`, ktorá slúži na vytváranie inštancií triedy a tiež pridáva metódy pre vytváranie a modifikáciu tabuľky metód.

Hierarchia objektov v SOM je flexibilná a umožňuje tzv. metaprogramovanie, teda programátorovi počas vykonávania zisťovať informácie o objektoch a triedach a meniť ich za chodu. Objekty sa počas vykonávania programu vytvárajú (vzorom Továreň) zavolaním továrenskej metódy na objekte triedy príslušného objektu, ktorý chceme vytvoriť. Po vytvorení objekt existuje dovtedy, kým nie je explicitne uvoľnený alebo pokým existuje proces, ktorý ho vytvoril. Ak chceme, aby SOM objekt žil dlhšie ako proces, ktorý ho vytvoril, je nutné realizovať vlastný perzistenčný mechanizmus.

Nové triedy sa vytvárajú odvodením od existujúcich tried. Podtrieda zdedí rozhrania a ich implementácie z rodičovskej triedy. Metódy môžeme preťažiť. Nové metódy môžu byť pridané, staré môžu byť preťažené. SOM podporuje viacnásobné dedenie.

Rozhrania k SOM objektom sa definujú pomocou SOM IDL (Interface Definition Language), ktoré je rozšírením CORBA IDL o možnosť špecifikovať dodatočné informácie o implementácii. Objekt špecifikujeme v IDL (príklad 2-5), pre ktoré SOM IDL kompilátor vytvorí jazykovo špecifické previazania pre cieľový kompilátor (podporované sú C a C++) korešpondujúce s definíciou triedy v jazyku IDL.

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void sayHello();
}
```

Príklad 2-5. Jednoduchý IDL súbor.

Previazania sú jazykovo špecifické makrá a procedúry obsahujúce volania do SOM spúšťacieho prostredia, ktoré zabezpečuje vytváranie a manipuláciu so SOM objektmi. Previazania umožňujú programátorovi jednoduchú interakciu so SOM pomocou syntaxe vhodnej pre príslušný programovací jazyk.

Napr. pre implementáciu objektu v jazyku C, IDL kompilátor vytvorí súbory .H, .IH a .C, ktoré obsahujú prázdne funkcie pripravené pre programátora na doplnenie špecifickej funkcionality (príklad 2-6).

```
#include <hello.ih>

SOM_SCOPE void SOMLINK sayHello(Hello somSelf, Environment *ev)
{
  /* HelloData *somThis = HelloGetData(somSelf); */
  HelloMethodDebug("Hello", "sayHello");

  printf("Hello!\n");
}
```

Príklad 2-6. Implementácia metódy objektu.

Výsledná implementácia je zlinkovaná do knižnice DLL alebo EXE súboru. Použitie DLL súboru umožňuje flexibilne použiť vytvorenú knižnicu SOM objektov v klientskej aplikácií, ktorá je schopná DLL súbor načítať (napr. aj v procedurálnom jazyku, príklad 2-7).

```
#include <hello.h>

int main(int argc, char *argv[])
{
  Hello obj;
  obj = HelloNew();
  _sayHello(obj, somGetGlobalEnvironment());
  _somFree(obj);
  return 0;
}
```

Príklad 2-7. Ukážka implementácie klienta v jazyku C.

Operácie a metódy

Každý SOM objekt má špecifikované rozhranie, ktoré opisuje signatúru operácií, ktoré možno nad objektom vykonať. Každá operácia sa skladá z názvu operácie, vstupných argumentov a výstupných hodnôt. Operácie sú vykonávané metódami, ktoré implementujú správanie objektu.

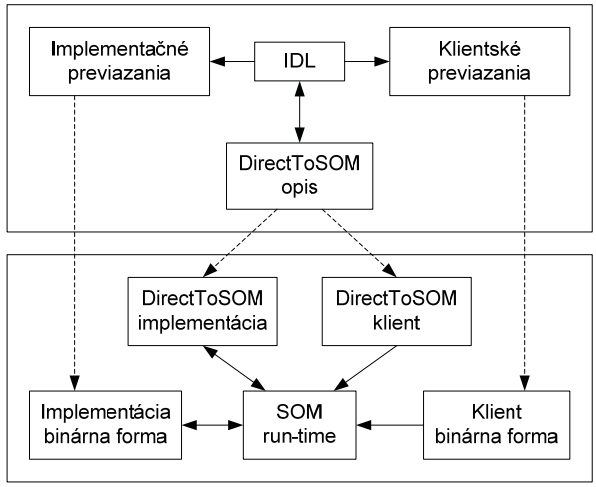
Klient požiada o službu špecifikovaním mena objektu a názvu operácie spolu s argumentmi. Objekt môže podporovať viaceré operácie. Niektoré programovacie jazyky podporujú polymorfizmus, teda operácie s rovnakým názvom ale odlišnými argumentmi. Metóda je procedúra spustená na SOM objekte, ktorej parametre zodpovedajú parametrom požadovanej operácie.

SOM podporuje tri možné mechanizmy vyhľadávania metód, ktoré sa majú vykonať na požiadavku pre vykonanie operácie: rezolvencia ofsetom (angl. *offset resolution*), rezolvencia menom (angl. *name resolution*) a rezolvencia spúšťacou funkciou (angl. *dispatch function resolution*).

Pri rezolvencii ofsetom klientsky kód spustí metódu cez smerník na špecifickom ofsete v tabuľke vypočítanej v čase kompilácie. Tento spôsob je do istej miery podobný virtuálnej tabuľke v C++. Rezolvencia menom vyhľadáva metódu dynamicky počas vykonávania programu podľa požadovaného mena a umožňuje tak prístup k objektom, ktorých trieda nie je známa v čase kompilácie. Rezolvencia spúšťacou funkciou umožňuje volanému objektu riadiť proces vyhľadávania metódy ľubovoľnými pravidlami.

Podpora v iných programovacích jazykoch

Okrem podpory SOM v jazykoch C a C++ pomocou SOM IDL kompilátora pre generovanie jazykovo špecifických previazaní, existoval ešte alternatívny spôsob použitia SOM pre C++, Smalltalk a OO-COBOL, tzv. DirectToSOM (Hamilton, 1996). V tomto prípade priamo kompilátor cieľového jazyka generoval a konzumoval IDL a generoval volania do SOM spúšťacieho prostredia, čím bola zabezpečená jednoduchšia (bez nutnosti vytvárať IDL programátorom) natívna podpora SOM priamo v programovacom jazyku (obrázok 2-13).



Obrázok 2-13. Schéma SOM (Hamilton, 1996).

Pre C++ a OO COBOL existuje plná DirectToSOM podpora, pre Smalltalk existuje len podpora DirectFromSOM, čo je len podmnožina umožňujúca SOM objekty používať len z klienta a nie je možné implementovať SOM rozhranie. SOM objekty sa v Smalltalku používajú cez obaľovacie triedy. Vytvorenie inštancie obaľovacej triedy vytvorí aj príslušnú inštanciu SOM triedy a volania metód na obaľovacom objekte spôsobia volania na SOM inštancii.

Mapovanie zo Smalltalku do SOM naráža ale aj na problém správy pamäte; keď sa už objekt v Smalltalku prestane používať, je automaticky uvoľnený správcom pamäti, ale keďže korešpondujúci SOM objekt existuje mimo pamäťového priestoru klienta, je nutné ho explicitne uvoľniť volaním metódy `somFree`. Navyše, keďže sa absolútne umiestnenie objektu v pamäti môže v Smalltalku zmeniť, nie je možné posielať do SOM jeho aktuálnu adresu a je potrebné pre každé volanie objekt nanovo skopírovať do pomocnej pamäte v SOM, v ktorej ale nezostáva po skončení volania metódy a pre prípadné odkazovanie v budúcnosti ho musí SOM celý skopírovať, čo spôsobuje ďalšiu neefektivitu spolupráce Smalltalku so SOM.

Okrem základného prostredia, boli vytvorené aj rozširujúce prostredia, ako napr. distribuovaná verzia SOM pre zdieľanie objektov medzi logicky nezávislými systémami. Distribuované SOM (DSOM) je sada SOM tried rozširujúca mechanizmus spúšťania metód v spúšťacom prostredí o možnosť transparentného volania metód na SOM objektoch, ktoré existujú v inom adresnom priestore ako volajúci proces, väčšinou na inom (virtuálnom) stroji. DSOM je založený na mechanizme CORBA.

Nezávislosť od programovacieho jazyka

SOM umožňuje interoperabilitu medzi jazykmi vďaka svojmu mechanizmu spúšťania metód, ktorý je založený na konvenciách linkovania procedúr v operačnom systéme (IBM, 1994). Teda SOM využíva okrem iného konvencie pre používanie zásobníka a spôsob posúvania návratových hodnôt a môže tak spúšťať metódy nezávisle od programovacieho jazyka, v ktorom je spustiteľný kód vytvorený. SOM teda vie spolupracovať s každým jazykom, ktorý podporuje zaužívané linkovacie konvencie.

Binárna kompatibilita nahor

Binárna kompatibilita nahor, teda správne fungovanie klientskej aplikácie aj po zmene knižnice, ktorú používa, je zabezpečená úplným zapuzdrením informácií o triedach (Forman, 1995). Klientske previazanie na SOM triedu neobsahuje informácie o jej veľkosti alebo vstupných bodoch a spúšťanie metód a prístup k dátam objektov sa realizuje pomocou dátových štruktúr, ktoré sú vypočítané počas inicializácie triedy v spúšťacom prostredí. SOM trieda preto môže byť počas vykonávania programu upravená (napr. upravenie hierarchie tried, pridávanie alebo odstránenie metód) bez nutnosti rekompilácie klientskej aplikácie.

2.5.3 Zhodnotenie

Objektovo orientované jazyky priniesli viacero výhod oproti procedurálnym jazykom, vznikli však aj fundamentálne problémy, ktoré bránili efektívnejšiemu použitiu týchto nových jazykov pre tvorbu zdieľaných knižníc objektov.

Na jednej strane boli technológie ako napr. Component Object Model (COM) spoločnosti Microsoft, ktoré sa vzdali objektovo orientovaných princípov v prospech jednoduchšej architektúry, na druhej strane, v spoločnosti IBM sa postavili k vzniknutým problémom zoči-voči a vznikla technológia SOM, ktorá umožňuje zdieľať knižnice objektov v rôznych (nielen objektových) programovacích jazykoch a zachovať binárnu kompatibilitu nahor, teda po uvoľnení novšej verzie knižnice nie je nutné rekompilovať existujúce aplikácie, ktoré ju používajú.

SOM bolo komerčne dostupné v produktoch IBM od roku 1991, kedy sa objavilo ako súčasť OS/2 verzie 2.0, neskôr bolo dostupné aj na platformách AIX, Windows a Mac System 7. Objektový model SOM bol na svoju dobu veľmi pokročilý a okrem možnosti distribuovaného vykonávania (pomocou technológie CORBA) obsahoval aj možnosti pre viacnásobné dedenie a metaprogramovanie. Pri tvorbe formátu pre zložené dokumenty OpenDoc v Apple bola technológia SOM vybraná ako najvhodnejšia spomedzi dostupných technológií v tej dobe (Alfke, 1995).

Po prvotnom nadšení sa ale operačný systém OS/2 nedokázal presadiť na spotrebiteľskom trhu a prestal tak byť pre IBM komerčne zaujímavý. Ukončením podpory pre OS/2 v roku 2006 potom de facto zaniklo aj používanie SOM, ktoré ako technológia zostalo už len ako inšpirácia pre ďalší výskum.

2.6 Softvérové inžinierstvo založené na súčiastkach

Tvorcovia programov sa od vzniku programovacích jazykov snažili svoju prácu čo najviac zjednodušiť a po príklade stavebných inžinierov budovať programy zo „stavebných blokov“. Prvými takýmito blokmi boli podprogramy a knižnice.

Postupne sa zvyšovala snaha tieto bloky štandardizovať, aby sa nemuseli opätovne vyvíjať. Tak boli vytvorené jazyky, ktoré podporovali moduly a balíky. Modulárne jazyky boli neskôr rozširované o možnosti zakomponovania súčiastok. Jazyk Ada 83 takto rozšíril Grady Brooch v roku 1987 a jazyk Eiffel upravil Bertrand Mayer v roku 1997.

V deväťdesiatych rokoch tak vzniklo softvérové inžinierstvo založené na súčiastkach (angl. *Component-based software engineering – CBSE*). V jeho rámci bol definovaný životný cyklus súčiastky a spôsob vytvárania aplikácií zo súčiastok. Na základe týchto znalostí začali vznikať špecifické prostredia a vývojové nástroje založené na súčiastkach – ActiveX, JavaBeans, DCOM, .NET, CORBA a iné.

2.6.1 Základné princípy CBSE

CBSE sa často uvádza ako nadstavba objektovo orientovaného programovania. Ale súčiastky možno vytvárať nezávisle od programovacej paradigmy a nezávisle je definovaná aj CBSE metodológia. Zahŕňa opis procesov – návrh systému, manažment projektu, životný cyklus systému; a tiež technológiu – spôsob vytvárania aplikácie.

Nezávislosť CBSE vyjadrujú aj jeho tri základné princípy:

- *Špecifikácia súčiastky je oddelená od jej návrhu a implementácie*. To znamená, že môžeme robiť nezávislé implementácie súčiastky, ktoré by mali byť medzi sebou zameniteľné.
- *Sústredenie sa na návrh rozhrania*. Definovanie dohody (kontraktu) medzi klientom a serverom, ktorá umožňuje zaobaliť správanie sa každej súčiastky.

– *Formálna špecifikácia významu súčiastky*. Spolu s definíciou rozhrania pomáha orientovať sa implementátorovi pri výbere a vývoji súčiastky.

Na uvedených princípoch sú definované všetky časti metodológie CBSE a každá konkrétna realizácia alebo ďalší vývoj metodológie na ne musí brať ohľad.

2.6.2 *Problémové oblasti v CBSE*

Hlavným vývojovým procesom v rámci CBSE je práve životný cyklus súčiastky. Rieši množstvo otázok, ktoré sa vynárajú od vzniku súčiastky až po jej nasadenie a údržbu. Životný cyklus súčiastky je rovnaký ako pri každom softvérovom systéme. Špecifické ale je, že musíme brať ohľad na to, že súčiastka je vytváraná typicky aj za účelom znovupoužitia a súčiastka nie je to isté, čo finálna aplikácia. Problémy, ktoré vznikajú v rámci životného cyklu súčiastky sa týkajú týchto oblastí:

– vytváranie súčiastok,
– hľadanie súčiastok,
– použitie súčiastok,
– testovanie súčiastok,
– zužitkovanie súčiastok na znovupoužitie.

Vytváranie súčiastok

Pretože jednou z vlastností súčiastky je jej znovupoužitie, vynára sa kľúčová otázka – môže byť softvér navrhnutý bez napojenia na jeho konkrétne použitie? Pre nízkoúrovňové súčiastky je odpoveď – áno. Dôkazom sú napríklad často používané knižnice implementujúce abstraktné údajové typy.

Pre ostatné súčiastky je však potrebné si uvedomiť základné princípy CBSE. A tie nám hovoria, že súčiastka by mala byť navrhnutá a vytvorená tak, aby podporovala všetky etapy práce s ňou. Teda aj etapu jej použitia. A keďže aj pre túto etapu by mala mať súčiastka nejaký formálny opis, je potrebné niečo vedieť aj o jej konkrétnom použití.

V praxi sa implementácia takmer vždy odlišuje od toho, čo predpisuje návrh. Je to z toho dôvodu, že v etape návrhu zvyčajne nie je možné myslieť na všetky detaily, ktoré sa pri konkrétnej realizácii vyskytnú. Súčiastky, ktoré boli vytvárané bez napojenia na nejaký konkrétny problém, tieto detaily neriešia. Preto je ich znovupoužitie problematické a dopracovanie týchto „drobností okolo“ môže byť náročnejšie, než vývoj bez použitia danej súčiastky.

Ďalší problém pri vytváraní súčiastky je rozmanitosť štandardov. Existujú rôzne programovacie paradigmy, ktoré poskytujú rôzne možnosti znovupoužitia a rôzne možnosti modifikovania súčiastky. Existujú rôzne štandardy rozhraní – CORBA, DCOM a iné.

Najlepšie je, keď je možnosť implementovať súčiastku pre viacero štandardov. Možnosť tu ale znamená čas, ľudí a peniaze. Preto sa zvyčajne vyberá najvhodnejší alebo najrozšírenejší štandard pre očakávané použitie súčiastky.

Hľadanie súčiastok

Pri vytváraní aplikácie je často výhodnejšie použiť existujúcu súčiastku, než vytvárať vlastnú. Je ale potrebné nájsť vhodnú súčiastku. Ak je hľadané riešenie nízkoúrovňové

alebo málo špecifické, je možné nájsť aj presnú zhodu s opisom súčiastky. Niekedy vôbec nenájdeme súčiastku s danými vlastnosťami. Vtedy môžeme skúsiť dva prístupy. Prvým je, že začneme hľadať kompozíciu súčiastok. Vtedy každá súčiastka spĺňa len časť požiadaviek a je ich potrebné poskladať do želaného celku. Druhým prístupom je faktorizácia problému. Určíme si podstatné vlastnosti súčiastky a hľadáme súčiastku len podľa týchto vlastností.

Vo väčšine prípadov však nájdeme viacero súčiastok, ktorých opis je ale len podobný našim požiadavkám. A vzniká podobný problém ako pri faktorizácii – musíme určiť, ktoré vlastnosti súčiastky sú pre nás dôležité a ktoré nie.

Na dosiahnutie čo najlepšieho kompromisu sa pri výbere snažíme zodpovedať na nasledujúce otázky:

- Aké štandardy budem pri vytváraní aplikácie preferovať?
- Ktorá súčiastka vyžaduje menší zásah do implementácie?
- Sú väčšie nároky na prispôsobenie rozhrania alebo prispôsobenie funkcionality?
- Je dôležitejšie to, čo súčiastka robí alebo to, čo nerobí?

Aby sme mohli na všetky otázky dobre odpovedať, je vhodné vytvoriť si istý predvýber súčiastok, ktoré sú potenciálne použiteľné pre finálnu aplikáciu. Ak sa napríklad zistí, že väčšina použiteľných súčiastok je vytvorená na základe iného komunikačného štandardu než bol pôvodne navrhnutý, možno zavčasu upraviť návrh a znížiť tak nároky na úpravy počas implementácie. Príklad procesu hľadania súčiastok je dobre opísaný v prípadovej štúdii (Ulkuniemi, 2004).

Použitie súčiastok

Pre túto etapu je dôležité vyriešiť nasledovné tri úlohy:

- porozumenie vlastnostiam súčiastky,
- modifikovanie súčiastky na dosiahnutie požadovanej funkcionality,
- spájanie súčiastok do finálnej aplikácie.

Dôkladné porozumenie vlastnostiam súčiastky je nevyhnutné pre správnu modifikáciu súčiastky, dosiahnutie zhody programu so špecifikáciou a v neposlednom rade pre správnu činnosť celého systému.

Modifikovanie súčiastky je kľúčový proces v rámci znovupoužitia. Pretože súčiastka musí spĺňať aj požadovanú funkcionalitu, aj komunikovať so zvyškom systému prostredníctvom definovaného rozhrania, je skôr výnimkou, ak súčiastka vyhovuje obom požiadavkám. Súčiastka má väčšiu šancu na znovupoužitie, ak je všeobecnejšia. Príliš konkrétne zameraná súčiastka je použiteľná len na úzky okruh problémov. Keď sa však súčiastka použije, bude riešiť konkrétnu úlohu a z tohto dôvodu sa musí väčšina súčiastok modifikovať.

Je dôležité rozlíšiť dva spôsoby modifikácie súčiastky. Ak sú k dispozícii zdrojové kódy súčiastky, možno modifikovať súčiastky priamo. Častým prípadom však je, že je k dispozícii len vykonateľný kód, najmä keď sa používajú komerčne dostupné súčiastky. V tom prípade modifikácia súčiastky znamená jej obalenie novým kódom, ktorý zabezpečí parametrizáciu všeobecných vlastností súčiastky a podľa potreby aj zmenu jej rozhrania.

Spájanie súčiastok do finálnej aplikácie znamená vytvorenie jednotnej a pritom efektívnej architektúry a je to z hľadiska kreativity najnáročnejší proces. Tento proces je nutné začať už keď sa začína s výberom súčiastok, inak vôbec nemusí mať použiteľné riešenie. Často vzniká potreba vytvorenia spojovacieho programu (angl. *glue code*). Na vývoj takéhoto programu zatiaľ existujú len všeobecné odporúčania, integračné architektúry sú v štádiu vývoja, takže väčšina práce ostáva na skúseného softvérového inžiniera.

Testovanie súčiastok

Testovanie je zásadnou aktivitou vo vývoji softvérových systémov a možnosť vhodného testovania môže priamo rozhodnúť, či sa pri tvorbe softvérového systému súčiastky použijú alebo nie.

Hotové súčiastky majú tú výhodu, že už boli nejako testované autorom. Problémom je ale dôveryhodnosť tohto testovania. Pretože úplné testy sú príliš náročné, autor súčiastky zvyčajne robí len tak rozsiahle testovanie, aké považuje za rozumné. Preto je jedným z kritérií výberu súčiastky aj dôveryhodnosť jej autora. Najlepšie je, ak sa jedná o často používanú súčiastku, ktorú bola použitá v mnohých systémoch a tam boli tiež overené jej vlastnosti.

Problém dôkladného otestovania súčiastky typicky zhoršuje neustále vytváranie nových verzií súčiastky. Ako každý softvér, aj súčiastka časom zastaráva a je potrebné vydávať jej nové verzie. Súčasný trend je navyše taký, že sa všade pridáva nová funkcionalita. S novou funkcionalitou zvyčajne prichádzajú aj nové chyby a často aj nekonzistentnosť a to nielen s predchádzajúcou verziou, ale aj v rámci samotnej súčiastky, najmä jej dokumentácie.

Preto býva potrebné súčiastku testovať aj pri jej použití. Vtedy sa snažíme vyriešiť dva základné problémy:

- Aké testovacie údaje použiť?
- Ako vieme, že testovanie je postačujúce?

Súčiastka bola vytvorená „bez napojenia na jej konkrétne použitie“. To znamená, že jej dokumentácia môže obsahovať len informácie o všeobecnom testovaní, ak vôbec takéto informácie obsahuje. Pri komerčných súčiastkach zvyčajne ani nie je k dispozícii zdrojový kód a preto je nutné vytvoriť testovacie údaje na základe metódy čierna skrinka. Ak však súčiastka môže mať veľké množstvo vnútorných stavov, metóda čiernej skrinky je príliš slabá na jej dôveryhodné otestovanie.

Znovupoužitie súčiastok

Problém znovupoužitia súčiastky vzniká, keď sa rozhodujeme, či súčiastku vytvoríme tak, aby bola znovupoužiteľná alebo bude vytvorená len pre tento špecifický prípad. Aj keď je myšlienka znovupoužiteľnosti lákavá, na vytvorenie súčiastky tak, aby sa dala opakovane použiť, sú potrebné ďalšie zdroje. Preto je potrebné riešiť otázku prínosu takéhoto riešenia.

V prvom rade môže byť prínosom opakované použitie vo vlastných budúcich aplikáciách. V budúcnosti tým odpadajú spomínané problémy s učením sa novej technológie, poznávaním vlastností súčiastky a jej testovaním. Podmienkou ale je, že organizácia musí už teraz vedieť, že bude riešiť podobný typ projektov a že súčasná technológia je dosť perspektívna.

Iným prínosom môže byť predaj technológie, teda vytvorených súčiastok. Podmienkou je, že súčiastky budú použiteľné v dostatočnom množstve aplikácií a budú tiež vytvorené dostatočne kvalitne, aby o ne bol vyšší záujem ako o prípadnú konkurenciu. To kladie vyššie nároky na tvorbu takýchto súčiastok. Najlepšie je, keď súčiastky dodržiavajú známe štandardy a sú uvádzané prostredníctvom katalógov, kde je ich možné efektívne vyhľadávať. Katalógy sú vytvárané extrakciou metadát z opisu jednotlivých súčiastok. Činnosti pri vytváraní znovupoužiteľných súčiastok môžeme zhrnúť do týchto krokov:

- analýza existujúceho softvéru na výskyt opakujúcich sa štruktúr a súčastí,
- určenie, čo z toho sa využije v budúcnosti a čo by mohlo mať prínos z predaja,
- reinžiniering vybraných súčiastok a vytvorenie adekvátneho opisu.

2.6.3 Problémy s prepojením súčiastok

Čím je vytváraný systém zložitejší, tým väčšie problémy s prepojením jednotlivých súčiastok môžu vznikať. Od súčiastok sa vyžaduje vysoký stupeň integrovateľnosti a zavádza sa preň pojem interoperabilita. Interoperabilita je definovaná ako schopnosť súčiastok komunikovať a spolupracovať bez ohľadu na implementačný jazyk, operačný systém alebo abstraktný model.

Vyžadujú sa dva typy interoperability – syntaktická (alebo tiež statická) a sémantická (alebo tiež dynamická) interoperabilita.

Syntaktická interoperabilita

Z pohľadu syntaxe potrebujeme mať opis rozhraní súčiastok jednotným spôsobom. Overíme si tak, či súčiastky majú jednotnú komunikáciu – súčiastky si dokážu vymieňať údaje a rozumieť aký typ údajov si vymieňajú. Tým je zabezpečená syntaktická interoperabilita.

Špecifickou súčasťou syntaktickej interoperability je protokolová interoperabilita. Táto je dodržaná, keď obe komunikujúce súčiastky akceptujú obmedzenia dané komunikačným protokolom.

V praxi treba sledovať aj verzie jednotlivých súčiastok, lebo pri rozširovaní funkcionality sa často rozširuje aj spôsob komunikácie a tým súčiastka prestane dodržiavať protokolovú interoperabilitu.

Sémantická interoperabilita

Sémantická interoperabilita znamená, že správanie sa súčiastky musí byť v zhode s nadväzujúcimi súčiastkami. To znamená, že ak súčiastka očakáva od nejakej inej súčiastky, že jej nejakým spôsobom spracuje údaje, nemala by ich v novšej verzii spracovať inak.

Ako príklad možno uviesť súčiastku, ktorá po spracovaní údajov dáva na výstup abecedne utriedené údaje. V novej verzii ale poskytuje viac druhov triedenia a bez dodatočnej informácie poskytuje údaje neutriedené. Sú to stále tie isté údaje ako predtým, nasledujúcej súčiastke to však môže ale aj nemusí vadiť. V každom prípade je sémantická interoperabilita porušená, aj keď sa to v lepšom prípade nemusí prejaviť.

Aby sa predišlo podobným problémom, boli definované dve úrovne spolupráce:

- úroveň definujúca zhodný mechanizmus komunikácie údajov,
- úroveň definujúca zhodný spôsob interpretácie metaúdajov.

Obe úrovne by mali byť dodržané. Zatiaľ sú však problémy so štandardizáciou týchto odporúčaní. Zatiaľ sa nenašlo všeobecné riešenie pre nasledovné otázky:

- Ako vyjadriť informácie o interoperabilite?
- Ako zverejniť tieto informácie, aby boli dostupné komukoľvek, kto ich potrebuje?

Skryté predpoklady

Poslednou skupinou problémov, ktoré sa objavujú pri práci so súčiastkami sú skryté predpoklady vývojárov.

Každý vývojár má vlastné predstavy o modeli správy objektov, perzistencii, paralelizme, distribuovanosti, spôsobe aktivácie súčiastok, ich bezpečnosti a spoľahlivosti. Mnohé z týchto informácií nie sú dostupné pre toho, kto bude tieto súčiastky využívať. Musí teda urobiť svoj predpoklad o tom, ako to v danej súčiastke je.

Tento problém sa vyskytuje najmä pri komerčných súčiastkach (angl. *Commercial off-the-shelf – COTS*) a je pôvodcom ťažko odhaliteľných chýb v správaní sa systému. Zdrojom týchto problémov je neúplná funkcionalita, neočakávané obmedzenia, chyby a závislosti medzi verziami súčiastok. Odhaleniu týchto nedostatkov pritom bránia legálne, etické a technické problémy pri použití týchto súčiastok.

Možné riešenia týchto problémov sú opísané v časti o testovaní súčiastok.

2.6.4 Zhodnotenie

Softvérové inžinierstvo založené na súčiastkach je spôsob vývoja softvérových aplikácií, v ktorom sa využívajú vopred vytvorené otestované znovupoužiteľné kusy softvéru (súčiastky), z ktorých sa pružne vyskladá výsledná aplikácia.

Okrem samotnej tvorby požadovanej aplikácie je teda potrebné predtým navrhnúť a vytvoriť znovupoužiteľné súčiastky. Myšlienka modulárnych aplikácií je tu už dlho, ale efektívne spôsoby tvorby znovupoužiteľných súčiastok sú stále predmetom výskumu.

Použitá literatúra

[1] Alfke, J. P.: Learning to Love SOM. In: *MacTech Magazine,* Vol.11, No.1, Apple Computer, Inc., 1995.

[2] Birrell, A. D., Nelson, B. J.: Implementing remote procedure calls. In: *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, 1984.

[3] Dansforth, S., Koennen, P., Tate, B.: *Objects for OS/2*, Van Nostrand Reinhold, 1994.

[4] Faison, E. W., Faison, T.: *Event-based Programming: Taking Events to the Limit*. Apress, ISBN 1590596439, 2006.

[5] Forman, I., Conner, M., Dansforth, S., Raper, L.: Release-to-Release Binary Compatibility in SOM, *ACM SIGPLAN Notices*, Vol. 30, No. 10, 1995.

[6] Hamilton, J.: Interlanguage Object Sharing with SOM. In: *Proc. of the 1996 USENIX Conference on Object-Oriented Technologies*, 1996.

[7] Henning, M.: The rise and fall of CORBA. In: *ACM Queue*. Vol. 4, No. 5, pp. 28-34, 2006.

[8] InterSystems Corporation: *Ensemble Best Practices*. Dostupné z: `http://docs.intersystems.com/documentation/ensemble/20091/pdfs/EGBP.pdf`, 2009.

[9] IBM: *SOMobjects Developer Toolkit Users Guide*, IBM Corporation, 1993.

[10] IBM: IBM's System Object Model (SOM): Making reuse a reality. *First Class. Object Management Group*, 1994.

[11] Kaisler, S. H.: *Software Paradigms.* Hoboken, NJ: John Wiley & Sons, Inc., 2005.

[12] Lowe, W.: *Software from Components*. Dostupné z: `http://w3.msi.vxu.se/~wlo/files/SoftwareFromComponentsWT08/Slides1.pdf`

[13] Object Management Group: CORBA Basics. Dostupné z: `http://www.omg.org/gettingstarted/corbafaq.htm`, `2009.`

[14] Puder, A.: CORBA Product Profiles. Dostupné z: `http://www.puder.org/corba/matrix/`, 2009.

[15] Schmidt, D. C.: Response to 'The Rise and Fall of CORBA' by Michi Henning. Dostupné z: `http://www.dre.vanderbilt.edu/~schmidt/corba-response.html`, 2009.

[16] Srinivasan, R.: *Request for Comments 1831: RPC: Remote Procedure Call Protocol Specification Version 2* [online]. 1995 [cit. 02-05-2009]. Dostupné z: `http://www.ietf.org/rfc/rfc1831.txt.`

[17] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. 2nd Edition. Reading, MA: Addison-Wesley Professional, 2002.

[18] Sun Microsystems: JavaBeans™ Specification v1.0.1., 1997.

[19] Tanenbaum, A. S., van Steen, M.: *Distributed Systems: Principles and Paradigms*. 2nd Edition. Prentice Hall, 2006.

[20] Ulkuniemi, P., Seppänen, V.: *Definition of a COTS Software Component Acquisition Process – The Case of a Telecommunication Company*. Kung-Kiu L. (Ed.) Component-based software development: Case Studies. World Scientific Publishing Company, ISBN: 9812388281, pp. 57-79, 2004.

[21] Ulkuniemi, P., Seppänen, V.: *Definition of a COTS Software Component Acquisition Process – The Case of a Telecommunication Company*. Kung-Kiu L. (Ed.) Component-based software development: Case Studies. World Scientific Publishing Company, 2004, pp. 57-79. ISBN: 9812388281

3
ARCHITEKTÚRY SOFTVÉRU

Ivan Kišac, Tomáš Kuzár, Pavol Mederly,
Jozef Tvarožek, Ivan Kapustík, Nikoleta Habudová

S rozvojom informačných systémov narástlo aj ich využitie pri riešení problémov v rôznych oblastiach každodenného života. Čím zložitejšie problémy sa začali pomocou nich riešiť, tým zložitejšie systémy bolo treba vyvíjať. Navyše v súčasnosti sa už len zriedkavo vyvíjajú systémy, ktoré by riešili len jeden problém. Obyčajne systém rieši určitú množinu problémov z problémovej domény. Problémy z jednej domény vytvárajú *problémový priestor*.

Pri riešení problémov z problémového priestoru sa identifikujú tie, ktoré sú si navzájom podobné. Na základe opisu riešenia jedného problému vytvárame predlohu spôsobu riešenia problémov z danej oblasti. Získavame *architektúru softvéru*. Táto špecifikuje zloženie systému zo súčiastok, ich charakteristiky, interakcie a komunikáciu. Premosťuje požiadavky na systém a jeho implementáciu. Ako uvádza (Eoin Woods): „Softvérová architektúra je množina návrhových rozhodnutí...“. Správna architektúra je veľmi dôležitá. Aj optimálne algoritmy a dátové štruktúry použité v systéme môžu stratiť svoju efektivitu a výkon, ak sú zakomponované do systému s nevhodnou štruktúrou.

3.1 Prehľad architektúr softvéru

Na rôzne triedy problémov boli vytvorené rôzne zodpovedajúce architektúry ako výsledky vývoja snáh o navrhnutie riešení na špecifikované problémy. Nasledujúce podkapitoly budú pojednávať o opise architektúr a architektonických štýloch.

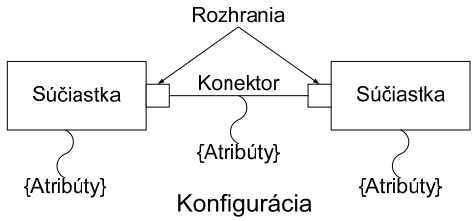
3.1.1 Elementy a opis softvérových architektúr

Existuje viacero definícií, ktoré sa pokúšajú správne a čo najvýstižnejšie zadefinovať pojem architektúry softvéru, napríklad:

- zbierka súčiastok a konektorov spolu s opisom interakcií medzi týmito súčiastkami a konektormi (Garlan, 1994),
- štruktúra alebo štruktúry systému, ktoré pokrývajú softvérové súčiastky systému, externe viditeľné vlastnosti týchto súčiastok a vzťahy medzi nimi (Bass, 1998).

Pomocou týchto definícií môžeme na systém a jeho architektúru nahliadať z pohľadu jednotlivých zložiek: súčiastok, konektorov, ich vlastností a celkovej konfigurácie. Takýto

náčrt jednoduchého systému znázorňuje obrázok 3-1. Jednotlivé prvky zohrávajú v systéme rôzne roly.



Obrázok 3-1. Koncept architektúry softvéru.

Súčiastky

Základnými stavebnými prvkami systémov sú súčiastky. Sú to výpočtové entity, ktoré riešia čiastkové úlohy pri procese riešenia úlohy, ktorou sa systém zaoberá. Proces čiastkového príspevku k riešeniu problému softvérovou súčiastkou má 2 fázy:

- vnútorný výpočet – predstavuje samotnú funkcionalitu súčiastky z hľadiska úlohy v systéme,
- vonkajšia komunikácia – je potrebná na prijímanie úloh a odovzdávanie výsledkov výpočtu. Je realizovaná prostredníctvom portov, rozhraní (angl. *interfaces*).

Atribúty predstavujú informáciu pre analýzu a vývoj softvéru.

Konektory

Vzájomnú komunikáciu súčiastok systému zabezpečujú konektory. Definujú interakciu medzi súčiastkami a jej pravidlá. Konektor môže spájať 2 a viac súčiastok. V rámci pravidiel definovaných konektorom sú aj roly v komunikácii, napr. definované roly zdroj – príjemca.

Konektor predstavuje aplikačne nezávislý mechanizmus spojenia. Má zadefinované svoje vlastnosti a rozhranie pre použitie a zabezpečuje komunikáciu bez ohľadu na aplikačné určenie systému. Toto umožňuje lepšiu modularitu systému a ďalšiu nahraditeľnosť samotného konektora (napr. lepším, efektívnejším a pod). Pre konektory existujú rôzne implementácie, ktoré majú aj rôzne vlastnosti, napr. oneskorenie pri komunikácii a pod.

Konektor ako sprostredkovateľ komunikácie svojimi vlastnosťami umožňuje rôzne typy interakcií:

- binárne, n-árne – definujú, koľko súčiastok sa môže na komunikácii podieľať,
- symetrické, asymetrické – definujú princípy synchronizácie v rámci komunikácie,
- protokoly výmeny informácií – definujú, ako sa budú informácie prenášať.

Rozhrania

Súčiastky si definujú svoje vstupno-výstupné rozhrania. Tieto rozhrania takto vytvárajú kritérium pre spájanie súčiastok. Konektorom môžu byť prepojené len kompatibilné rozhrania. Každá súčiastka môže mať viacero rozhraní. No konkrétne rozhranie patrí vždy práve jednej súčiastke. Vlastnosti rozhraní sú opísané ich atribútmi, ako napr. smer komunikácie, vstupné a výstupné protokoly, kapacita zásobníka (ak je), atď.

Konfigurácia

Dôležitou časťou architektúry systému je jeho konfigurácia (topológia). Tento súvislý graf súčiastok a konektorov opisuje architektonickú štruktúru systému. Konfigurácia musí spĺňať požadované vlastnosti:

- dodržať pravidlá správneho spojenia, najmä z hľadiska kompatibility spájaných súčiastok,
- uspokojiť požiadavky na zabezpečenie rôznych typov spracovania (sekvenčné, paralelné) a distribúcie informácií,
- dodržať heuristiky návrhu a pravidlá štýlu.

Na základe analýzy topológie systému spolu s informáciami o súčiastkach a konektoroch možno odvodiť vlastnosti celého systému ako aj identifikovať mnohé problémy. Medzi takéto vlastnosti a problémy patria napr. výkonnosť, problém úzkeho hrdla, otázky paralelizmu, bezpečnosti a pod.

3.1.2 Architektonické štýly

Architektonickým štýlom (resp. vzorom) nazývame opakujúci sa vzor alebo idióm na vyjadrenie softvérovej architektúry na vyriešenie problému. Pri riešení problémov boli identifikované napríklad nasledujúce výrazné štýly:

Dátovody a filtre

Ide o architektonický štýl, ktorý je najvhodnejší pre systémy založené na prúdovom spracovaní informácií. Súčiastkami v tomto štýle sú filtre slúžiace na spracovanie toku údajov. Tok dát je charakterizovaný pomocou dátovodov, ktoré prepájajú výstupy jedných filtrov so vstupmi nasledujúcich filtrov v reťazi spracovania toku údajov. Modulárnosť systému vychádza z možnosti zamieňania a znovupoužitia filtrov, pričom treba dodržať len špecifikácie ich rozhraní, a z možnosti zmien v topológii zmenou prepojení – dátovodov.

Tabuľové systémy

Tento architektonický štýl je vhodný najmä pre triedu problémov, kde existujú agenty schopné vykonávať operácie nad údajmi, pričom nie je jasné, v akom poradí ich treba na dosiahnutie riešenia uplatniť. Agenty operujú nad spoločným úložiskom údajov – tabuľou, ktorá predstavuje aktuálny stav riešeného problému. Ak niektorý z agentov identifikuje stav, v ktorom by svojimi schopnosťami (svojou funkcionalitou) mohol prispieť k riešeniu problému, vykoná príslušný výpočet a výsledkami aktualizuje stav tabule.

Klient-server systémy

Populárnym štýlom pre riešenie architektúry softvéru sú klient-server systémy. Uplatňujú sa najmä v distribuovanom prostredí. Založené sú na poskytovaní funkcionality a uchovávaní údajov na strane servera a na prístupe k nim pomocou klienta. Jednotlivé architektonické štýly sú bližšie opísané v rámci tejto kapitoly.

Vlastnosti architektúr a ich opis

Na samotný opis architektúry bolo vyvinutých viacero formálnych resp. menej formálnych prostriedkov, napríklad:

- PSL/PSA (jazyk na vyjadrenie problémov/analyzátor vyjadrenia problémov, angl. *Problem Statement Language*/*Problem Statement Analyser*) – vysoko úrovňové opisy správania, CASE nástroj na podporu analýzy požiadaviek a štruktúry počas fázy návrhu,
- IDEF0, IDEF1X – štrukturálna dekompozícia,
- ADLs (jazyky na opis architektúry, angl. *architecture description languages*) – pre strojové spracovanie, vyvinuté agentúrou pre pokročilé výskumné projekty obrany,
- upravený formát opisu návrhových vzorov – neopisuje však ďalšie adaptácie architektúry.

Doménovo-špecifickú architektúru treba opisovať tak, aby nám tento opis následne pomohol získať prehľad a uľahčil používanie rámca (angl. *framework*), ktorý ju implementuje.

Architektúra systému môže byť pokrytá viacerými vzormi, ktoré sa prejavujú v rôznej intenzite. Niektoré vzory sú viac architektonické a prejavujú sa pri opise infraštruktúry, napr. vzor MVC (Model-Pohľad-Ovládač, angl. *Model-View-Controller*), ktorý sa prejavuje najmä pri tvorbe architektúry a stránke grafického používateľského rozhrania. Iné vzory majú pre danú úroveň pohľadu príliš jemnú granularitu, sú vo vnútri súčiastok a preto ich na architektonickej úrovni nie je vidieť.

Dobrá architektúra je pre systém veľmi potrebná, nakoľko umožní naplno využiť efektívnosť algoritmov, identifikovať problémy v tokoch údajov, vizualizovať systém ako taký, čo umožňuje lepšiu modularitu, zapracovanie zmien, odhaľovanie a odstraňovanie chýb. Ďalej poskytuje z organizačného hľadiska podklady pre komunikáciu a manažment projektu (identifikovanie, odhad a plánovanie potrebných zdrojov, optimalizácia a pod.).

Úlohy architektúry možno zaradiť do viacerých oblastí:

- *porozumenie* – dosahuje sa vďaka vysokoúrovňovému návrhu, pri tvorbe a analýze ktorého možno odvodiť obmedzenia systému,
- *znovupoužitie* – navrhnutá architektúra má v rámci svojej domény možnosť znovupoužitia a podobne aj systémy a ich časti podľa nej vytvorené,
- *konštrukcia* – v oblasti konštrukcie poskytuje plány pre vývoj a integráciu modulov,
- ďalší vývoj – identifikuje miesta vhodné na ďalšie rozširovanie,
- *analýza* – z architektúry systému možno urobiť kontrolu konzistencie, závislostí a obmedzení systému,
- *manažment* – slúži ako podklad pre komunikáciu a identifikáciu problémov.

Samotná architektúra sa od návrhu odlišuje najmä svojou úrovňou abstrakcie. Kým ona sa zaoberá štruktúrou, súčiastkami a ich interakciami, návrh sa zaoberá detailmi, procedúrami a rozhraniami.

V 80. rokoch minulého storočia prebiehal výskum v oblasti vývoja doménovo-špecifických architektúr. Hlavnými cieľmi bolo zvýšiť efektivitu vývoja a znížiť náklady pomocou znovupoužitia (návrhov, postupov, metód a objektov) a riešenia problémov v danej oblasti raz a správne v danej doméne namiesto toho, aby sa v tejto doméne vhodné riešenie opakovane znova hľadalo.. Výsledkami výskumu boli okrem referenčných architektúr pre dané problémy aj závery o samotných doménach:

- doména nie je svojím rozsahom totožná s trhom alebo priemyselným odvetvím,

- architektúra pre doménu nie je trvalo konzistentná – doména sa neustále vyvíja a mení,
- je vysoká potreba dobrých podporných nástrojov.

Pri výbere architektúry sa treba zamerať najmä na tieto vlastnosti:

- *interakcia* – spôsoby komunikácie medzi súčiastkami a použité protokoly,
- *interoperabilita* – miera vzájomného dopĺňania funkcionality súčiastok,
- *odlišnosť* – podpora rôznych funkcií a služieb,
- *výber* – na riešenie určitého problému existuje obyčajne viacero architektúr,
- *agilita* – univerzálnejšie sú architektúry, ktoré vychádzajú z čím menej predpokladov a umožňujú aj podporu zastaraných systémov.

3.2 Systémy riadené tokom údajov

V systémoch, v ktorých výpočtový proces je riadený tokom údajov, je základným predpokladom uskutočnenia výpočtu dostupnosť príslušných údajov. Základné elementy systému riadeného tokom údajov sú: proces, súbor a tok údajov. Údaje tečú od dátového zdroja k dátovému spotrebiču. Dátový zdroj a dátový spotrebič predstavujú špeciálne procesy, ktoré sa nachádzajú na okrajoch systému.

Každý proces vykonáva nad dátami operácie. Tieto operácie môžeme zatriediť do troch kategórií:

- pridávanie informácie,
- agregácia alebo extrakcia informácie,
- transformácia údajov.

Rovnako možno kategorizovať spôsob, akým dáta tečú v rámci systému:

- lineárne,
- cyklicky,
- ľubovoľným iným spôsobom.

3.2.1 Model toku údajov

Tradičná von Neumannova koncepcia počítača je založená na vykonávaní sledu príkazov, kedy počítač v každom kroku vyberie príslušné inštrukcie a dáta z pamäti a vykoná operáciu.

Na druhej strane model toku údajov možno reprezentovať ako graf, kde uzly predstavujú výpočty – výpočtové uzly a hrany predstavujú dátové cesty. Údaj možno považovať za dynamickú entitu, ktorá je konzumovaná procesmi – inštrukciou alebo programom – v závislosti od granularity opisu systému. Tento spôsob výpočtu je vo veľkej miere deklaratívny, čo znamená, že dôraz sa kladie na to, čo sa má vypočítať a nie na to, kedy a ako sa má vykonať výpočet. Definovanie pravidiel výpočtu na jednotlivých uzloch umožňuje vniesť do procesu výpočtu viac abstrakcie a menej sa zaoberať podrobnosťami týkajúcimi sa samotného toku dát.

3.2.2 *Charakteristika toku údajov*

Tok údajov medzi uzlami môžeme charakterizovať z viacerých hľadísk, pričom systémy riadené tokom údajov môžeme zaradiť do dvoch základných skupín: systémy založené na transformačnom toku a systémy založené na transakčnom toku.

Pri transformačnom toku sa v systéme používajú dva formáty pre reprezentáciu údajov: interný a externý. Prichádzajúce údaje sú pri vstupe do systému najskôr transformované do interného formátu, v ktorom sa ďalej vykonávajú všetky výpočty vnútri systému. Po ukončení výpočtu sa dáta vychádzajúce zo systému transformujú do externého formátu.

Pri transakčnom toku údaje prúdia v rámci systému priebežne. Jeden vstupný údaj môže spustiť viacero tokov údajov. Túto logiku výpočtu možno nájsť napr. pri dopytovacích a interaktívnych systémoch. Jeden dopyt môže mať aj viacero výstupov a výstupy jednotlivých výpočtov sa môžu podľa potreby ľubovoľne kombinovať.

V oboch prípadoch však tok údajov môžeme chápať ako matematickú kompozíciu. Máme teda dva moduly, jeden môžeme opísať funkciou $f(x)$ a druhý $g(y)$. Výstup modulu opísaného funkciou $f(x)$ bude vstupom pre modul s funkciou $g(y)$. Takto získame výsledok $h(x) = g(f(x))$.

Systémy riadené tokom údajov sú deterministické. Základná jednotka spracovania, ktorú nazývame *token*, načítaná v rámci jedného modulu je vždy výsledkom výpočtu na nejakom predchádzajúcom module. Tokeny sú čítané v poradí v akom boli zapísané. Nie je možné povedať, či token prišiel načas, či prišiel príliš skoro alebo či prišiel neskoro. Výpočet sa uskutoční vždy, keď sú dáta k dispozícii.

Aplikácie riadenia tokom údajov

Systém riadený tokom údajov je obzvlášť vhodný v prípade, že sa vyžadujú dynamické zmeny v aplikácii v závislosti na vstupných dátach, v prípade meniacich sa externých podmienok alebo pri používaní rôznorodých metód spracovania.

Systém riadený tokom údajov má svoje špecifické vlastnosti, ktoré ho predurčujú na nasadenie vo viacerých konkrétnych situáciách. Spomenúť možno:

- zariadenia s vysokými požiadavkami na výkon viazané na špecifickú doménu,
- aplikácie s potrebou častých zmien v čase vykonávania,
- systémy s požiadavkou na adaptivitu správania sa aplikácie vzhľadom na vstupy,
- zložité aplikácie, kde nie je možné vymenovať všetky možné nastavenia a stavy,
- časti systémov fungujúcich na báze čiernej skrinky s cieľom zvýšenia výkonu,
- aplikácie využívajúce obojsmerný dátový tok a cykly.

Architektúru založenú na riadení tokom údajov nie je vhodné použiť všeobecne, napr. nie je vhodná pre textové aplikácie. Na druhej strane môže byť vhodne použitá v tabuľkovom nástroji typu Excel, kde bunka v tabuľke predstavuje uzol, na ktorom prebehne výpočet v okamihu keď sú k dispozícii príslušné vstupné dáta. Ďalším vhodným príkladom použitia architektúry riadenej tokom dát je spracovanie signálov a obrazu, kde sa uskutočňujú opakujúce sa výpočty nad zložitými dátovými elementmi.

3.2.3 Implementácia

Systém riadený tokom údajov je často implementovaný ako množina súčiastok, ktoré si medzi sebou vymieňajú správy prostredníctvom jednosmerných portov. Čiže každý port danej súčiastky dokáže správu buď odoslať alebo prijať. A každá súčiastka má jeden alebo niekoľko prijímajúcich portov a tiež jeden alebo niekoľko odosielajúcich portov. V prípade dvojíc komunikujúcich súčiastok môžeme vždy jednu z dvojice považovať za prijímajúcu (angl. *downstream*) a druhú za odosielajúcu (angl. *upstream*) súčiastku. V niektorých implementáciách systému riadenom tokom údajov sa medzi odosielajúcou a prijímajúcou súčiastkou môže nachádzať zásobník slúžiaci na priebežné uskladnenie nespracovaných dát, ktoré vznikajú, keď odosielajúca súčiastka spracúva údaje rýchlejšie ako jeho prijímajúca nasledovníčka.

Ako už bolo spomenuté v predchádzajúcom texte, výpočet prebiehajúci na konkrétnom uzle možno reprezentovať pomocou transformačnej funkcie. V rámci správneho návrhu transformačných funkcií by mali byť splnené základné požiadavky na tieto funkcie.

Funkcia nemá byť doménovo špecifická. Súčiastka by mala byť schopná spracovať ľubovoľný problém. Jednotlivé súčiastky by mali fungovať nezávisle od používania. A ďalšou dôležitou požiadavkou je, že spracovanie prebiehajúce v rámci súčiastok by nemalo mať žiadne vedľajšie efekty, čo znamená, že spracovanie závisí od aktuálnych vstupných dát. Požiadavky kladené na súčiastky:

- bezpamäťové súčiastky,
- vymeniteľné v čase vykonávania programu,
- adaptácia meniacim sa požiadavkám, napr. výmena $h(x) = g(f_1(x))$ za $h(x) = g(f_2(x))$.

Základné entity, ktoré charakterizujú systém riadený tokom údajov sú, ako sme spomínali vyššie, proces a tok dát medzi procesmi. V závislosti od úrovne abstrakcie procesov a dát rozoznávame jemný tok údajov a hrubý tok údajov.

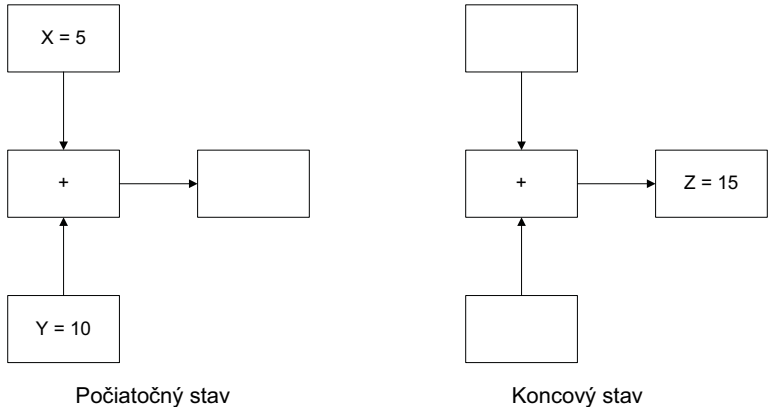
3.2.4 Jemný tok údajov

Základnou jednotkou vykonávanou v rámci procesu pri jemnom toku údajov je strojová inštrukcia. Jemný tok údajov vyžaduje podporu programovacieho jazyka, vhodnú architektúru stroja pre reprezentáciu problému, efektívne mapovanie programu na výpočtové uzly, implementáciu, vykonanie.

Základné časti jemného modelu toku údajov:

- graf toku údajov,
- množina uzlov – operátory,
- množina hrán – tok medzi operátormi,
- počiatočný a koncový uzol,
- množina predchodcov a množina nasledovníkov.

Na obrázku 3-2 je zobrazený jednoduchý príklad výpočtu. Premenným X a Y sú priradené hodnoty 5 a 10. Operácia sčítania môže byť vykonaná až vtedy, keď sú k dispozícii príslušné operandy. Následne sa operácia uskutoční a v ďalšej súčiastke je premennej Z priradená hodnota 15.



Obrázok 3-2. Jemný výpočet.

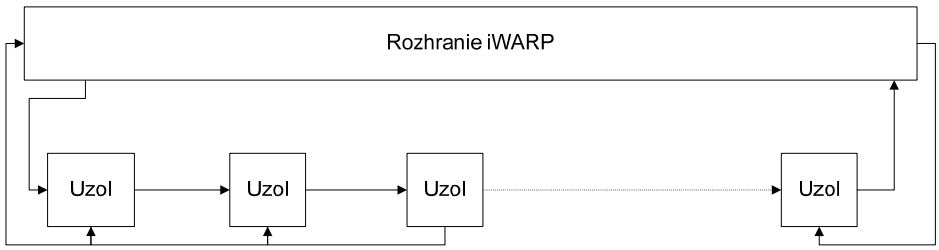
3.2.5 *Hrubý tok údajov*

Hrubý model toku údajov predstavuje mierne odlišný pohľad na túto architektúru. Základnou jednotkou, ktorá je vykonávaná na uzle, je programová súčiastka. Programová súčiastka môže byť časť programu, aplikácia ale aj celý zložitý systém. Tok dát predstavuje výmenu správ medzi súčiastkami. Správy môžu mať tiež ľubovoľne zložitú formu. Samotný princíp systému riadenom tokom údajov však ostáva nezmenený: vykonanie výpočtu závisí od dostupnosti operandov.

3.2.6 *Systolické polia*

Ďalším príkladom použitia architektúry pre systémy riadené tokom údajov sú systolické polia. Systolické pole môže byť lineárne, kde sa jednotlivé výpočtové uzly nachádzajú lineárne za sebou alebo sieťové, kde jednotlivé uzly vytvárajú zložitejšiu štruktúru (napr. 2D mriežku).

V systolickom poli tečú údaje medzi susedmi. Dáta tečú z jednej strany, na uzle sa uskutoční výpočet a výsledok prejde na druhú stranu. Vhodným príkladom použitia výpočtu realizovateľného týmto spôsobom môže byť násobenie matíc. Známy príklad implementácie systolického poľa je procesor iWarp schematický zobrazený na obrázku 3-3.



Obrázok 3-3. Schéma procesora iWarp.

Systolické polia sú obzvlášť použiteľné pri výpočtoch, ktoré majú lineárnu alebo pravidelnú štruktúru: rýchla Fourierova transformácia, spracovanie obrazu, radar, násobenie matíc, vyhľadávanie a ďalšie.

3.2.7 *Výmena správ*

Komunikácia v rámci systému riadeného tokom údajov je založená na posielaní správ. Tento spôsob komunikácie umožňuje efektívnu komunikáciu medzi programami, synchronizáciu, plánovanie úloh, či reprezentáciu dátových závislostí. Účelom posielania správ je predovšetkým teda:

- komunikácia,
- synchronizácia,
- plánovanie,
- reprezentácia závislosti medzi údajmi.

3.2.8 *Zhrnutie*

Hlavnou výhodou architektúry založenej na riadení tokom údajov je podpora paralelizmu. Vlastnosti tohto paralelizmu možno zhrnúť do niekoľkých bodov:

- Výpočet a komunikácia predstavujú dve akoby samostatné časti systému. Súčiastky medzi sebou komunikujú zasielaním správ a samotný výpočet v rámci súčiastky je vykonávaný v podstate sekvenčne.
- Výpočet je aktivovaný dostupnosťou dát, čo znamená, že výpočet je v samej podstate paralelný a asynchrónny.
- Možnosť abstrakcie od detailov stroja je v réžii systémového softvéru v čase vykonávania programu. Programovanie sa zameriava na reprezentáciu problému a možno abstrahovať od detailov stroja.
- Znovupoužiteľnosť súčiastok je dosiahnutá tým, že medzi jednotlivými súčiastkami nie sú závislosti (okrem používania dátových typov), je zaručená vysoká modularita a znovupoužiteľnosť súčiastok.

Architektúra založená na riadení tokom údajov nie je nový fenomén. Táto architektúra nedokázala nahradiť tradičnú koncepciu počítača založenú na toku riadenia avšak bola v mnohých špeciálnych prípadoch vhodne použitá.

3.3 Dátovody a filtre

3.3.1 *Základná charakteristika vzoru*

Architektonický vzor Dátovody a filtre (angl. *Pipes and Filters*) definuje štruktúru pre systém spracúvajúci prichádzajúci prúd údajov formou postupnosti krokov spracovania (angl. *processing steps*). Tento architektonický vzor je vhodný na realizáciu systémov riadených tokom údajov opísaných v predchádzajúcej časti.

Architektonický vzor Dátovody a filtre je vhodné použiť vtedy, ak pri návrhu architektúry potrebujeme dosiahnuť jednu alebo viacero z nasledujúcich vlastností:

1. ľahkú modifikovateľnosť systému, a to konkrétne v zmysle ľahkej modifikovateľnosti jednotlivých krokov spracovania ako aj ich postupnosti,
2. zvýšenie výkonu prípadne dostupnosti riešenia prostredníctvom paralelného resp. distribuovaného spracovania,

3. možnosť využiť pri tvorbe systému už existujúcu implementáciu jednotlivých krokov spracovania, prípadne naopak, opakovane použiť implementáciu jednotlivých krokov spracovania v nových kontextoch.

Predpokladom pre použitie tohto architektonického vzoru je, že požadované spracovanie údajov môžeme rozdeliť na postupnosť krokov spracovania, a to tak, že nesusedné kroky spracovania nezdieľajú informácie – t.j. komunikácia v systéme je obmedzená na odovzdávanie informácií smerom od daného kroku spracovania k jeho nasledovníkovi (v niektorých prípadoch nasledovníkom) v postupnosti krokov spracovania.

Systém založený na architektúre Dátovody a filtre pozostáva z nasledujúcich druhov súčiastok (Buschmann, 1996). Prvé dva predstavujú vstup a výstup vytváraného systému, posledné dva tvoria jeho vnútornú štruktúru.

1. Zdroj údajov (angl. *data source*) – predstavuje zdroj, ktorý obsahuje alebo generuje údaje určené na spracovanie systémom, napr. súbor, klávesnica, senzor, sieťové spojenie.
2. Dátový spotrebič (angl. *data sink*) – predstavuje cieľ, kam sú údaje po spracovaní systémom odoslané, napr. súbor, obrazovka, databázová tabuľka, sieťové spojenie.
3. Filter (angl. *filter*) – predstavuje jeden krok spracovania. Filter má zvyčajne jeden vstup, z ktorého prijíma prichádzajúce údaje. Tieto údaje spracuje a pošle zvyčajne na jeden výstup. Existujú varianty tohto vzoru, kedy filter môže mať viac vstupov, resp. viac výstupov. Filter zvyčajne nemá vnútorný stav, teda spracovanie prichádzajúcich údajov je typicky závislé len od týchto údajov.

   Príkladmi úloh vykonávaných filtrami sú transformácia údajov, doplnenie údajov informáciami odvodenými z týchto údajov prípadne z externého zdroja, odstránenie časti údajov, usporiadanie údajov (táto operácia vyžaduje vnútorný stav filtra) a podobne.

   Množina filtrov môže byť realizovaná programovým kódom v rámci jedného procesu (kedy jednotlivé filtre sú realizované procedúrami, funkciami alebo metódami tried v programovacom jazyku), ale aj programovým kódom distribuovaným vo viacerých procesoch, či už na jednom alebo rôznych počítačoch v počítačovej sieti. Jednotlivé filtre sú vtedy implementované v samostatných procesoch.
4. Dátovod (angl. *pipe*) – spája jednotlivé filtre, resp. zdroj údajov s prvým filtrom v postupnosti spracovania a posledný filter v postupnosti spracovania s dátovým spotrebičom. Dátovody môžu byť realizované prostriedkami v rámci jedného procesu (napríklad volaním procedúry, funkcie, resp. metódy), prostriedkami medziprocesovej komunikácie v rámci jedného počítača (napríklad dátovody na úrovni operačného systému) alebo prostriedkami medziprocesovej komunikácie v rámci počítačovej siete (napríklad TCP sokety).

   Dátovody v prípade potreby zabezpečujú synchronizáciu medzi účastníkmi komunikácie, t.j. ukladajú správy v rade, kým ich príslušná komunikujúca strana neprevezme. Vzor predpokladá neobmedzenú kapacitu vyrovnávacej pamäte (angl. *buffer*) dátovodu.

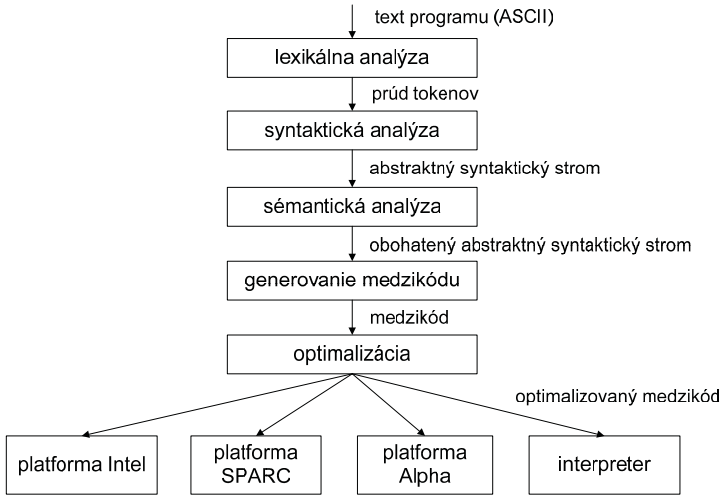
3.3.2 *Príklady*

Typickým príkladom použitia architektonického vzoru Dátovody a filtre je prostredie príkazového interpretera (angl. *shell*) v operačnom systéme typu Unix. Príkazy v tomto prostredí môžu byť zložené z viacerých elementárnych príkazov spojených pomocou dátovodu (anonymného alebo pomenovaného), a tak môžu poskytovať bohatú funkčnosť vytváranú z pomerne jednoduchých súčiastok. Príklady:

```
# vypise prvych 20 podadresarov s najvacsim objemom udajov

du | sort -nr | head -n 20

# jednoducha kontrola pravopisu textu ziskaneho z urcenej
# webovej stranky
# (zdroj: Wikipedia, http://en.wikipedia.org/wiki/Unix_pipe)

curl "http://en.wikipedia.org/wiki/Pipeline_(Unix)" | \
sed 's/[^a-zA-Z ]/ /g' | \
tr 'A-Z ' 'a-z\n' | \
grep '[a-z]' | \
sort -u | \
comm -23 - /usr/share/dict/words
```

Príklad 3-1. Zložené príkazy v prostredí príkazového interpretera v operačnom systéme typu Unix.

Iným príkladom je typická architektúra kompilátora pozostávajúca zo sekvencie krokov spracovania (obrázok 3-4).



Obrázok 3-4. Typická architektúra kompilátora ako príklad použitia architektonického vzoru Dátovody a filtre – prevzaté z (Buschmann, 1996).

Na obrázku sú zobrazené filtre (obdĺžniky) a dátovody realizujúce spojenia medzi nimi. Opis jednotlivých dátovodov zodpovedá typu údajov, ktoré sú daným dátovodom prenášané. Zdroj údajov a dátové spotrebiče nie sú znázornené. Všimnime si, že táto abstraktná schéma architektúry nehovorí nič o spôsobe realizácie filtrov a dátovodov. Tieto môžu byť implementované napríklad ako procedúry v rámci programu, spojené tradičnými vola-

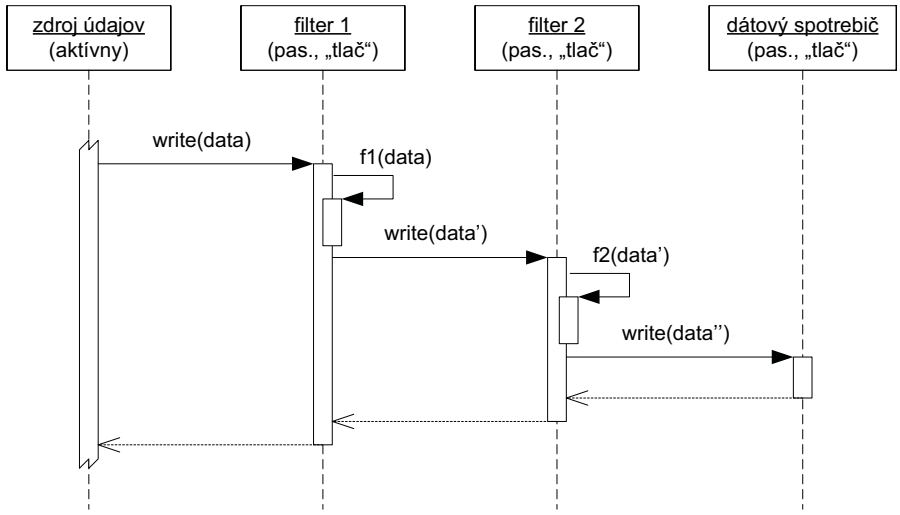
niami (architektonický vzor Volanie a návrat) alebo ako nezávislé procesy spojené dátovodmi na úrovni operačného systému.

3.3.3 Tok riadenia

Z hľadiska toku riadenia rozlišujeme dva druhy filtrov: aktívne a pasívne filtre (Buschmann, 1996). Aktívne filtre implementujú vlastný tok riadenia: preberajú údaje zo vstupu, spracúvajú ich a posielajú na výstup. Naproti tomu pasívne filtre nemajú vlastný tok riadenia, t.j. čakajú, kým od nich nebude žiadať údaje výstupný dátovod resp. dátový spotrebič alebo naopak, kým im údaje na spracovanie nepošle vstupný dátovod resp. zdroj údajov. V prvom prípade hovoríme o pasívnom filtri typu „ťahaj“ (angl. *pull*), v druhom o pasívnom filtri typu „tlač“ (angl. *push*). Analogicky môžeme uvažovať o aktívnom a pasívnom zdroji údajov a dátovom spotrebiči.

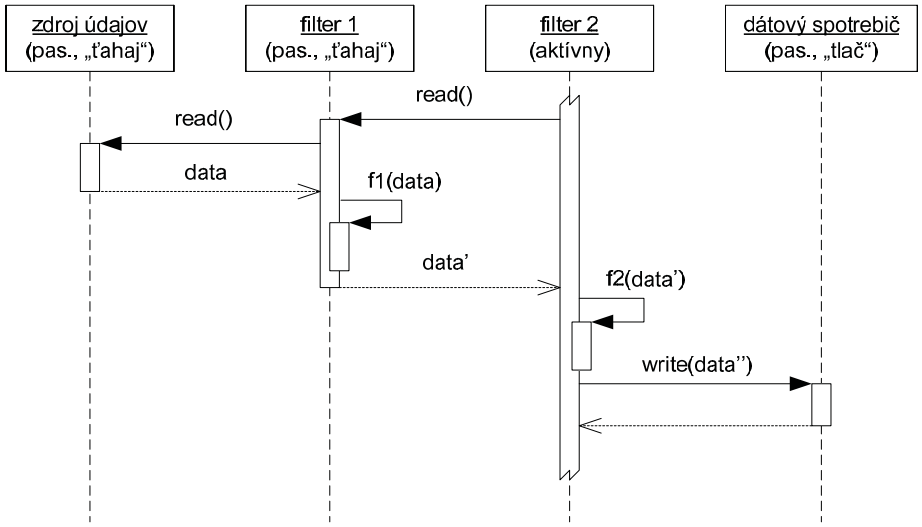
Dátovod spájajúci dva aktívne prvky musí realizovať synchronizáciu medzi nimi, teda musí riešiť napr. prípady, kedy prvý prvok pošle údaje, ale druhý si ich ešte nemá záujem prevziať. Dátovod spájajúci aktívny prvok s pasívnym, prípadne dva kompatibilné pasívne prvky môže byť realizovaný jednoduchým volaním procedúry alebo metódy, typicky s názvom `read`, `write`, resp. analogickým.

Nasleduje niekoľko sekvenčných diagramov znázorňujúcich jednotlivé situácie, prevzatých z (Buschmann, 1996). V prvom diagrame (obrázok 3-5) sú zobrazené dva pasívne filtre a pasívny dátový spotrebič, všetky sú typu „tlač“. Aktívny je zdroj údajov, ktorý posiela údaje prvému z filtrov volaním procedúry `write`. Prvý filter údaje spracuje (naznačené volaním procedúry `f1`) a pošle druhému filtru volaním jeho procedúry `write`. Analogicky i tento filter údaje spracuje (procedúra `f2`) a pošle dátovému spotrebiču (procedúra `write`). Toto všetko prebieha v rámci vlákna riadenia umiestneného primárne v zdroji údajov.



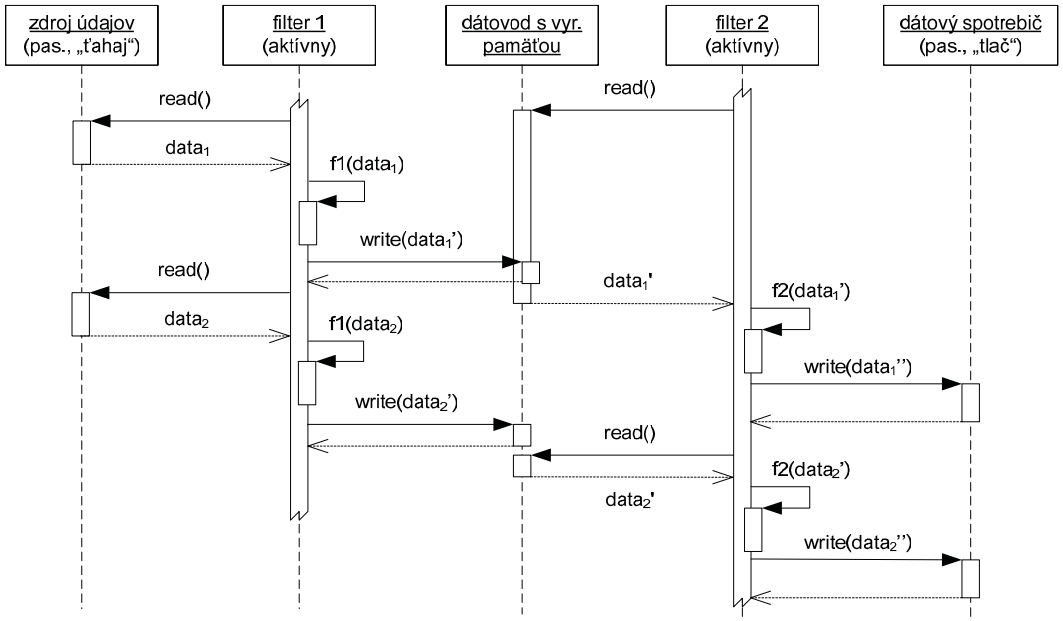
Obrázok 3-5. Aktívny zdroj údajov, pasívne filtre a dátový spotrebič, prevzaté z (Buschmann, 1996).

V nasledujúcom diagrame (obrázok 3-6) je aktívny filter 2, pričom zdroj údajov a filter 1 sú pasívne, typu „ťahaj“ a dátový spotrebič je pasívny, typu „tlač“.



Obrázok 3-6. Aktívny filter 2, prevzaté z (Buschmann, 1996).

V sekvenčnom diagrame na obrázku 3-7 sú oba filtre aktívne, pričom zdroj údajov a dátový spotrebič sú pasívne. Medzi dvoma aktívnymi filtrami je umiestnený dátovod s vyrovnávacou pamäťou. V diagrame je znázornená situácia, kedy oba filtre začínajú svoju prácu volaním procedúry `read` na vstupe, ktorým je pre filter 1 zdroj údajov a pre filter 2 dátovod.



Obrázok 3-7. Aktívne filtre 1 a 2, prevzaté z (Buschmann, 1996).

V prípade filtra 1 procedúra `read` vracia hneď údaje, kým filter 2 musí počkať, kým sa do dátovodu údaje z filtra 1 dostanú. Filter 1 po spracovaní údajov (procedúra `f1`) ich posiela

do dátovodu (procedúra `write`). Vtedy dátovod pošle údaje filtru 2 formou návratovej hodnoty volania procedúry `read` (údaje sú znázornené symbolom $data_1'$). Analogicky je spracovaný i ďalší balík údajov ($data_2$), pričom v tomto prípade sú údaje poslané filtrom 1 uložené v dátovode, kým si ich filter 2 prostredníctvom volania procedúry `read` neprevezme.

3.3.4 Viac vstupov a výstupov filtra

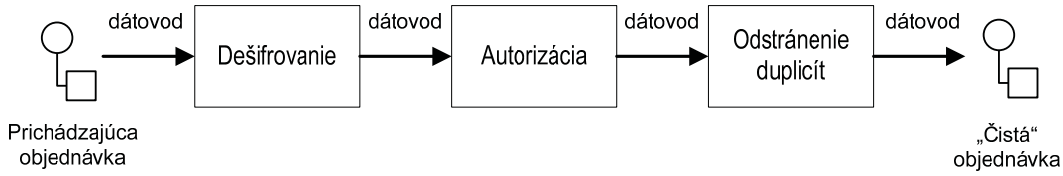
Doteraz prezentované príklady predpokladali najčastejšiu situáciu, ktorou je použitie filtrov s práve jedným vstupom a jedným výstupom. Vzor Dátovody a filtre existuje aj vo variantoch umožňujúcich použiť filtre s viacerými vstupnými prípadne výstupnými bodmi. V takýchto prípadoch sa výstupné údaje z filtra posielajú buď na všetky jeho výstupné body alebo len na jeden (resp. niekoľko) z nich a analogicky sa spustí spracovanie v rámci filtra, keď sú k dispozícii údaje na všetkých vstupných bodoch, alebo postačuje príchod údajov na jeden (resp. niektoré) vstupné body filtra.

3.3.5 Ďalšie príklady použitia

Okrem doteraz spomenutých príkladov, ďalšími príkladmi použitia architektonického vzoru Dátovody a filtre je vykresľovanie grafických scén (Kaisler, 2005), prípadne knižnica LASSPTools pre oblasť numerickej analýzy a grafiky (Buschmann, 1996).

Významnou oblasťou použitia tohto vzoru je integrácia podnikových aplikácií, kde sú integračné riešenia často založené na tomto architektonickom vzore (Hohpe, 2004).

Príkladom takéhoto integračného riešenia môže byť spracovanie prichádzajúcich objednávok, ktoré majú byť najskôr dešifrované (predpokladá sa, že prichádzajú v šifrovanej podobe), potom má byť vykonaná ich autorizácia (predpokladá sa nutnosť overenia, či je odosielateľ oprávnený poslať objednávku s danými charakteristikami, ako je napríklad finančná hodnota) a napokon odstránenie duplicít (predpokladá sa, že z technických dôvodov sa môže stať, že jedna objednávka je poslaná viackrát). Architektúra tohto integračného riešenia je zobrazená na obrázku 3-8.



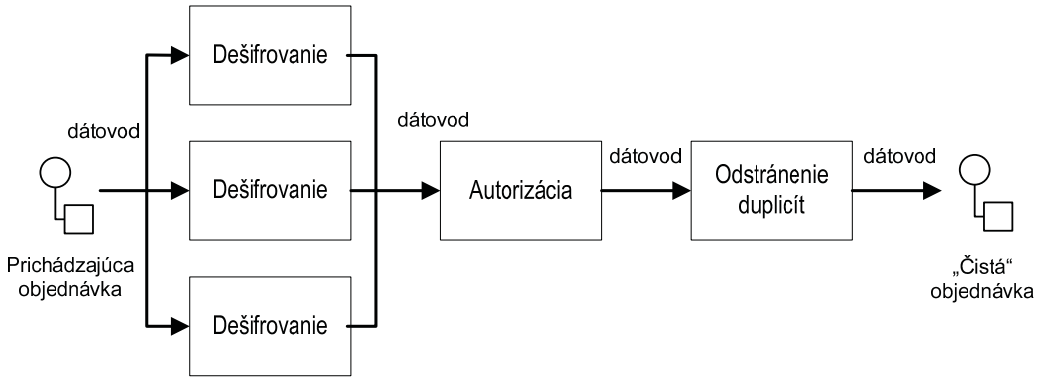
Obrázok 3-8. Integračné riešenie s architektúrou Dátovody a filtre – časť procesu spracovania objednávky, prevzaté z (Hohpe, 2004).

Charakteristickými vlastnosťami týchto integračných riešení, ktoré ich zároveň čiastočne odlišujú od bežného použitia vzoru Dátovody a filtre, sú:

1. Údaje prechádzajú systémom v diskrétnych „balíkoch“ (správach), kde správa zodpovedá typicky dokumentu, udalosti alebo príkazu – na rozdiel napríklad od kompilátora alebo sústavy dátovodov a filtrov v OS typu Unix, kde je medzi filtrami zvyčajne prenášaný homogénny prúd údajov.
2. Častým prvkom je vetvenie, spravidla formou smerovania správy na základe obsahu.

3. Na úrovni implementácie sa vlastnosti jednotlivých dátovodov a filtrov často dajú nastavovať administrátorom; ide o vlastnosti ako:
   a. perzistentnosť prenášaných správ (pre dátovody),
   b. transakčné spracovanie, umiestnenie na konkrétny server, počet inštancií, vlákien, procesov (pre filtre).

Posledný uvedený bod predstavuje dôležitú vlastnosť týchto riešení, ktorá poskytuje vysokú flexibilitu v určitých smeroch. Predpokladajme napríklad, že dešifrovanie prichádzajúcich objednávok je výpočtovo náročný proces a teda na dosiahnutie požadovanej priepustnosti celého procesu je potrebné ho vykonávať paralelne na viacerých počítačoch naraz. Vďaka flexibilnosti architektúry Dátovody a filtre je takáto zmena ľahko vykonateľná, mnohokrát (ako je uvedené vyššie) stačí dokonca len jednoduchá konfiguračná zmena realizovaná administrátorom systému. Výsledné riešenie je znázornené na obrázku 3-9.



Obrázok 3-9. Integračné riešenie s architektúrou Dátovody a filtre – upravené pre potreby dosiahnutia požadovanej priepustnosti – prevzaté z (Hohpe, 2004).

3.3.6 Zhrnutie vlastností vzoru Dátovody a filtre

Na záver zhrňme najdôležitejšie vlastnosti vzoru Dátovody a filtre, vychádzajúc prevažne z (Buschmann, 1996). Prínosy použitia tohto vzoru sú:

1. vysoká flexibilita riešenia v zmysle ľahkej výmeny, resp. preskupenia filtrov,
2. opakovaná použiteľnosť filtrov, najmä ak sú tieto filtre parametrizovateľné (ako je to napríklad pri príkazoch operačného systému typu Unix); táto opakovaná použiteľnosť je dosahovaná najmä minimalizáciou väzieb medzi filtrami v systéme,
3. ľahká paralelizovateľnosť a/alebo distribuovateľnosť jednotlivých filtrov (za predpokladu použitia vhodných dátovodov),
4. rýchla tvorba riešení, najmä prototypov.

Naopak, nedostatkami, resp. rizikami pri použití tohto vzoru sú:

1. ťažkopádne zdieľanie globálneho stavu,
2. vysoká réžia daná viacerými aspektmi, napríklad:
   a. nutnosťou prenosu údajov medzi filtrami, najmä ak prenos prebieha medzi procesmi resp. dokonca medzi počítačmi v sieti,

b. nutnosťou transformácie údajov, ak sa používa všeobecný formát na prenos údajov medzi filtrami,

3. problematická správa chýb,
4. fakt, že táto architektúra je použiteľná len pre isté aplikačné oblasti; príkladom oblasti, kde je použiteľná len ťažko, sú systémy vyžadujúce intenzívnu interakciu s používateľom.

3.4 Volanie a návrat

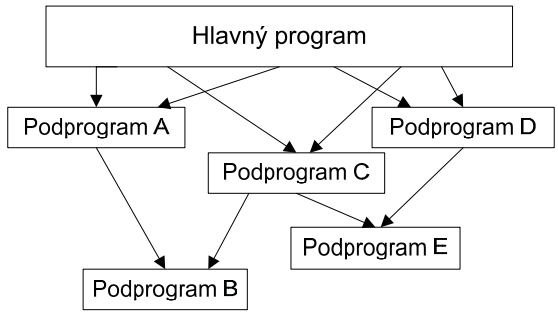
Volanie a návrat (angl. *call-and-return*) je synchrónna softvérová architektúra (Kaisler, 2005), v ktorej súčiastky komunikujú tak, že klient (volajúca súčiastka) pozastaví svoje vykonávanie počas spracovania požiadavky u poskytovateľa služby (volanej súčiastky). Teda riadenie (angl. *control flow*) sa z klienta presunie na poskytovateľa a po ukončení sa opäť vráti na klienta. V tejto architektúre tak súčiastky nevykonávajú zmysluplný výpočet počas čakania na odpoveď od iných súčiastok.

V nasledujúcich častiach opíšeme štyri hlavné architektúry typu volanie a návrat, a zhrnieme výhody a nevýhody spojené s ich použitím.

3.4.1 Hlavný program a podprogramy

V architektúre Hlavný program a podprogramy (angl. *main program and subroutines*) rozdeľujeme systém na hlavný program, ktorý slúži ako riadiaca jednotka výpočtu a podprogramy volané hlavným programom, ktoré vykonávajú špecifické funkcie podľa požiadaviek systému. Hlavný program môže volať podprogramy viackrát a z viacerých miest kódu. Podobne podprogramy programu môžu volať iné podprogramy, pozri obrázok 3-10.

Výpočet v tejto architektúre prebieha v jednom vlákne, čo zjednodušuje vizualizáciu a pochopenie samotného výpočtu. Správnosť výstupu hlavného programu je preto priamo závislá len od správnosti výstupov volaných podprogramov. Nevýhodou použitia je slabá škálovateľnosť. Pri rozsiahlejších projektoch sa ľahko stane, že je potrebné vytvoriť až stovky alebo tisíce podprogramov a stráca sa tak prehľad o toku riadenia vo výslednom programe. Podobne pri nových požiadavkách sa zvyčajne len doprogramuje potrebný podprogram bez toho, aby sa využili už existujúce, čím postupne vzniká neprehľadný kód, v ktorom viaceré podprogramy vykonávajú rovnaké alebo podobné operácie bez toho, aby o sebe vedeli.



Obrázok 3-10. Príklad architektúry Hlavný program a podprogramy.

V prípade súbežného vykonávania hovoríme o tzv. *Master-Slave* architektúre, v ktorej jeden hlavný proces koordinuje prácu ostatných podriadených procesov, ktoré pracujú nezávisle od seba, pričom na vykonanie úloh môžu potrebovať rozličné množstvo času. Po ukončení výpočtu podriadený proces odošle hlavnému procesu výsledky a čaká na ďalšiu úlohu. Tento prístup je vhodný pri výpočtovo náročných úlohách, ktoré sa dajú rozdeliť na viacero úloh tak, aby nemuseli podriadené procesy medzi sebou komunikovať. Používa sa napr. pri replikácii databáz a rozsiahlych distribuovaných výpočtoch[1]. Pri použití tejto architektúry je nutné realizovať nielen rozdelenie úlohy na menšie a spojenie výsledkov, ale aj mechanizmy na vyhľadávanie a komunikáciu podriadených procesov s hlavným koordinujúcim procesom.

3.4.2 Systémy Klient-server

V architektúre klient-server sú dva typy entít: *klient* – entita, ktorá požaduje vykonanie služby, *server* – entita, ktorá vykoná službu na požiadanie. Výsledok vykonania služby sa potom odošle klientovi. Vo svojej podstate je architektúra Klient-server podobná predchádzajúcemu typu Hlavný program a podprogramy. Rozdiel spočíva v tom, že v prípade Klient-server sa predpokladá, že klient a server pracujú na oddelených počítačoch. Oproti distribuovanej architektúre Master-Slave, v ktorej hlavný proces požaduje vykonanie úloh od podriadených, je rozdiel v tom, že v prípade systému klient-server sú to naopak „podriadené procesy“ (klienti), ktoré požadujú vykonanie úloh od hlavného procesu (servera). Väčšina internetových služieb (elektronická pošta, prehliadanie webových stránok) sú realizované ako systémy klient-server.

Podľa rozdelenia kompetencií a funkcionality medzi klientom a serverom, rozlišujeme dva typy klienta: *tučný* a *tenký*. Tenkí klienti obsahujú len prezentačnú logiku, v prípade webových systémov je to zvyčajne len štandardný webový prehliadač. Tenký klient posiela požiadavky na server a ďalej len vizualizuje jeho odpovede (v prípade webového systému klient-server sú to zvyčajne priamo webové stránky).

Naopak, tuční klienti obsahujú celú alebo aspoň nejakú časť aplikačnej logiky. Presunutie časti aplikačnej logiky na klienta môže byť výhodné na zníženie množstva sieťovej komunikácie a zlepšenie interaktivity klientskej aplikácie. Nevýhoda vzniká, ak je tučný klient implementovaný ako nezávislá používateľská aplikácia, potom v prípade zmeny požiadaviek a úpravy funkcionality tučného klienta je nutné novú verziu distribuovať zákazníkom. Pri použití tenkých klientov stačí zmeny vykonať na strane softvéru servera. Softvér servera spracúva požiadavky klientov súbežne nezávisle od seba (príklad 3-2).

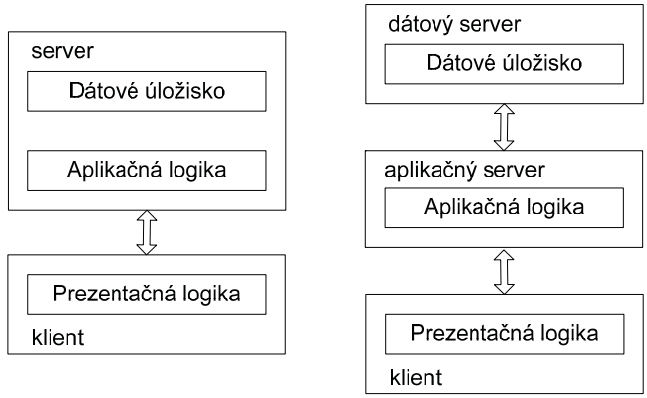
```
Bind(port);       // Rezervovanie portu v OS
Listen(port);     // Začni počúvanie na prichádzajúce požiadavky

while(spojenie = Accept()) {          // Nová požiadavka
  stop = AsyncProcess(spojenie);      // Asynchrónne spracovanie
  if (stop == true)                   // Ukončiť?
    break;
}
Close(port);      // Ukonči príjmanie požiadaviek
```

Príklad 3-2. Pseudokód spracovania požiadaviek na serveri.

[1] SETI@home, http://setiathome.berkeley.edu/

Najskôr si v operačnom systéme vyhradí port, na ktorom chce počúvať a následne spustí počúvanie na spojenia od klientov. Keď príde nové spojenie, asynchrónne ho spracuje v inom vlákne, pričom hlavný program pokračuje v čakaní na ďalšie spojenie. Softvér servera je prirodzene viacvláknový a teda z hardvérového hľadiska je výhodné použiť počítačový systém s čo možno najviac nezávislými výpočtovými jadrami.

Základná architektúra klient-server je realizovaná v dvoch vrstvách. Pre lepšiu škálovateľnosť, dostupnosť, atď. môže byť výhodné použiť ďalšie vrstvy (obrázok 3-11). Pri použití trojvrstvovej architektúry je prepojenie častí voľnejšie a môžeme ľubovoľnú z vrstiev zameniť alebo škálovať podľa vznikajúcich požiadaviek.



Obrázok 3-11. 2-vrstvová vs. 3-vrstvová klient-server architektúra.

V závislosti od požiadavky na spoľahlivosť komunikácie rozlišujeme dva typy komunikácie medzi klientom a serverom:

- *Spojenie neudržujúca* (angl. *connectionless*) komunikácia, je typicky realizovaná protokolom UDP (User Datagram Protocol), v ktorom doručovanie správ nie je spoľahlivé v zmysle, že odoslané správy nemusia byť nutne aj prijaté. Je potrebné realizovať vlastný mechanizmus spracovania chybových stavov v prípade nedoručenia paketu. Používa sa napr. pri audiovizuálnych prenosoch, ktoré nemusia byť nutne dokonalé a v ktorých sporadicky stratené pakety len mierne znižujú kvalitu prenosu.
- *Spojenie udržujúca* (angl. *connection-oriented*) komunikácia, typicky realizovaná protokolom TCP (Transmission Control Protocol), zaručuje doručenie odoslaných paketov. Netreba ošetrovať spracovanie chýb a implementácia je preto jednoduchšia. Naopak, zaručené doručenie zvyšuje komunikačnú réžiu.

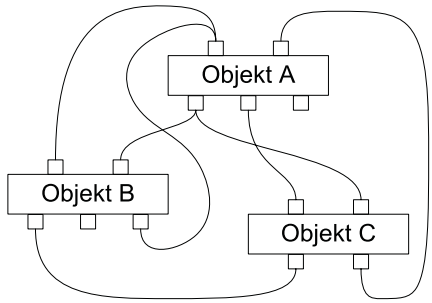
V prípade, že komunikácia medzi serverom a klientom prebieha vo viacerých správach, je niekedy výhodné, aby si server udržiaval stavovú informáciu o obsluhovaných klientoch. V takom prípade nemusí klient opakovane posielať informácie, ktoré si server už pamätá v stave pre daného klienta, čo znižuje veľkosť správ potrebných na realizovanie požiadaviek. Naopak, zhoršuje sa škálovateľnosť spracovania, keďže sa spracovanie na serveri musí koordinovať so stavovou informáciou uloženou v zdieľanej pamäti. Pri bezstavovom spracovaní sa môžu požiadavky spracovať nezávisle od ostatných a na ľubovoľnom výpočtovom uzle.

Výhody aj nevýhody architektúry klient-server vyplývajú najmä z jej centralizovaného charakteru. Všetky dáta sú uchovávané na serveroch, čo umožňuje vykonávať striktné dodržiavanie pravidiel pre pristupovanie k nim. Naopak, v prípade zlyhania kritickej časti servera nie je možné požiadavky klientov spracúvať. Tiež, keďže sú všetky požiadavky spracúvané centralizovane, vzniká riziko preťaženia centrálneho uzla a tak znefunkčnenia služby pre všetkých klientov.

3.4.3 Objektovo orientované systémy

V objektovo orientovaných systémoch sú komunikujúce entity (objekty) rovnocenné. Každý objekt môže principiálne volať metódy iného objektu a vystupovať tak v úlohe klienta (žiadateľa o službu) alebo servera (vykonávateľa požiadavky), pozri obrázok 3-12.

Tento model výpočtu sa odlišuje od ostatných tým, že objekty sú úplne zapuzdrené, čo umožňuje transparentne meniť implementáciu objektu pokiaľ sa dodrží špecifikované správanie. Každý objekt predstavuje abstrakciu zapuzdrených dát a metódami poskytuje služby nad týmito dátami, ktoré zachovávajú integritné ohraničenia. Takýto systém je rozšíriteľný a objekty môžu byť vytvárané počas vykonávania.



Obrázok 3-12. Príklad objektovo orientovaného systému.

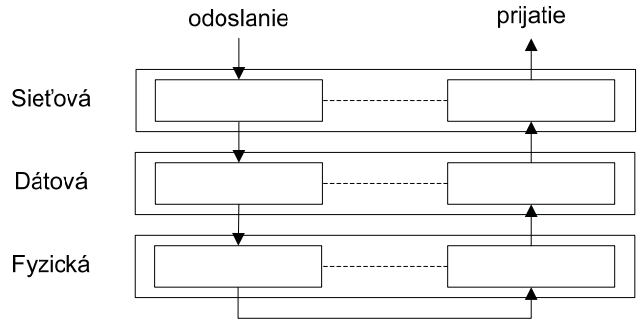
Vďaka úplnému zapuzdreniu objektov objektovo orientované systémy podporujú súbežný výpočet úloh v objektoch, čo je vhodné napr. aj pri realizácii multiagentových systémov. Tiež objekty sa nemusia vykonávať len na jednom počítači a použitím spojovacej vrstvy (angl. *middleware*) môžeme realizovať distribuovaný výpočet. Spojovacia vrstva udržiava identity komunikujúcich objektov a umožňuje lokalizačné služby, čo ale celkovo pridáva na réžii pri výpočte s distribuovanými objektmi.

3.4.4 Hierarchicky rozvrstvené systémy

Hierarchicky rozvrstvený (angl. *hierarchically layered*) systém je softvérový systém, ktorý je rozdelený do vrstiev, z ktorých každá predstavuje určitú abstrakciu funkcionality. Vrstva poskytuje služby vyšším vrstvám (umiestnením nad ňou) a používa služby nižších vrstiev, zvyčajne len služby vrstvy bezprostredne pod ňou. Hierarchicky rozvrstvená architektúra je výhodná, keď sú požiadavky jasne špecifikované. Používa sa napr. v jadrách operačných systémov alebo v referenčnom modeli komunikácie OSI (Open System Interconnection) (obrázok 3-13).

Funkcionalita každej vrstvy je jasne špecifikovaná rozhraním, čo umožuje v tejto architektúre niektorú vrstvu zameniť (úplne vymeniť) pokiaľ sa zachová jej verejné rozhra-

nie. Detaily implementácie vrstvy sú teda skryté a vrstvy možno navrhovať oddelene. Platformové závislosti môžu byť izolované v nižších vrstvách, čo umožňuje vytvoriť portabilné riešenie.



Obrázok 3-13. Hierarchické vrstvy časti referenčného modelu OSI.

Nevýhodou je, že nie vždy sa dá funkcionalita takto abstrahovať do vrstiev, príp. určiť správnu granularitu funkcionality vo vrstvách. Pridanie ďalšej funkcionality do vrstvy je relatívne jednoduché, naopak odobratie funkcionality môže spôsobiť problém vyšším vrstvám, ktoré túto funkcionalitu používajú.

3.4.5 Zhodnotenie

Architektúra volanie a návrat predstavuje základ ďalších odvodených architektúr, pričom každá zavádza nejaké dodatočné podmienky na obsah, smer a spôsob komunikácie medzi časťami systému.

Jednoduchá architektúra Hlavný program a podprogramy je vhodná pre malé projekty, pri väčších projektoch začína byť neprehľadná. Pre efektívne použite sofistikovanejších architektúr je nutné pochopiť najmä spôsob komunikácie medzi časťami systému, kde napr. medziprocesová komunikácia je relatívne vysoko výpočtovo náročná – vyžaduje okrem iného zabalenie parametrov a návratových hodnôt funkcií do formátu v komunikačnom protokole.

3.5 Systémy nezávislých súčiastok

Nezávislé súčiastky poznáme podľa toho, že si svoju prácu vykonávajú nezávisle od toho, s kým spolupracujú. Zatiaľ čo ostatné typy súčiastok potrebujú presne vedieť ktorú metódu alebo procedúru pri komunikácii volajú, pre nezávislé súčiastky je dôležité akurát rozpoznať s akými údajmi pracujú a spracujú ich bez ohľadu na to, kto ich poskytol alebo ku komu sa ďalej dostanú. Informáciu o konkrétnom odosielateľovi alebo prijímateľovi pritom môžu ale nemusia využiť.

3.5.1 Komunikujúce sekvenčné procesy

Najjednoduchší systém nezávislých súčiastok predstavuje množina nezávislých procesov, ktoré môžu medzi sebou komunikovať alebo sa navzájom aktivovať. Už v roku 1987 navrhol C. A. R. Hoare model paralelných procesov, kde nebolo presne určené kedy a s kým bude každý proces komunikovať.

Základnou Hoarovou myšlienkou bolo, že dva procesy sa najjednoduchšie zosynchronizujú pri svojich vstupno/výstupných operáciách. Na to je potrebné splniť práve dve podmienky:

1. proces A musí oznámiť, že je pripravený odovzdať údaje procesu B,
2. proces B musí oznámiť, že je pripravený prijať údaje od procesu A.

Ak niektorá z podmienok nie je splnená, pripravený proces prejde do stavu čakania až kým nie je pripravený aj druhý proces.

V roku 1985 rozšíril tento systém tak, že každý proces mohol mať viacero vstupov, pričom každý vstup mal určené podmienky na aktivovanie. Vždy mohol byť aktívny len jeden vstup, ale pokiaľ mali viaceré vstupy splnenú podmienku, vybral sa náhodne jeden z nich. Toto umožnilo modelovať nedeterministické vykonávanie procesov.

Hoare navrhol svoj systém len ako modelovací nástroj, neočakával, že sa využije v komerčných aplikáciách. To sa aj potvrdilo, lebo sa zistilo, že tento systém má viacero vážnych nedostatkov.

Prvým problémom bolo, že vyžadoval explicitné pomenovania. Každý proces musel poznať názvy všetkých procesov, s ktorými chcel komunikovať. Nebolo tak možné vybrať alternatívny proces na komunikáciu.

Druhým problémom bolo, že systém nemal definované žiadne fronty (angl. *queues*) na ukladanie správ. Preto bola možná len synchrónna komunikácia. Na modelovanie nezávislého, asynchrónneho procesu spracovania údajov sa vkladal dodatočný proces, ktorého jedinou úlohou bolo podržať údaje, kým nie je pripravený aj prijímací proces.

Posledným problémom bolo, že nebolo zabezpečené detekovanie ani ochrana pred uviaznutím.

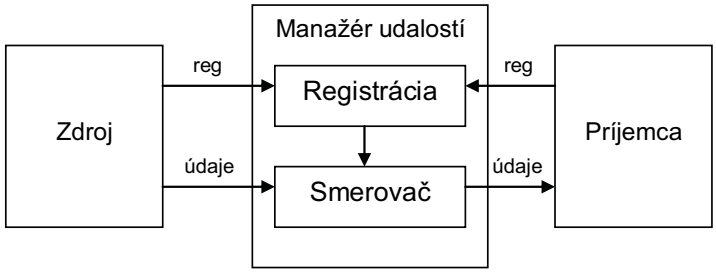
Hoarov systém však má zásluhu aj na tom, že sa tieto problémy objavili a vyriešili, takže v novších systémoch sa už zvyčajne nevyskytujú.

3.5.2 Systémy založené na udalostiach

V systémoch založených na udalostiach sa niečo vykonáva len vtedy, keď nastane nejaká udalosť. Zdroj udalosti pritom vôbec nemusí vedieť kto jeho udalosť spracuje a či ju vôbec niekto spracuje. Podobne príjemca udalosti sa tiež nemusí zaujímať o toho, kto tú udalosť vytvoril. Dôležité je len aký je to typ udalosti a aké údaje s touto udalosťou súvisia. Udalosť sa v týchto systémoch prenáša ako správa, kde typ správy určuje typ udalosti a obsah správy reprezentuje údaje alebo odkaz na údaje, ktoré sa majú spracovať.

Na obrázku 3-14 je základná štruktúra systému založeného na udalostiach. Jeho podstatnou časťou je manažér udalostí. V manažéri sa zaregistruje zdroj udalostí a oznámi aký typ udalosti bude posielať. Podobne sa tam zaregistruje aj príjemca udalostí a uvedie aký typ udalostí chce prijímať. Na základe registrácie sa nastaví smerovač, ktorý prepája informácie od zdrojov k príjemcom.

Od momentu, keď je zaregistrovaný aspoň jeden zdroj a aspoň jeden príjemca rovnakého typu udalostí, udalosti môžu byť spracovávané. Zdroj aj príjemca sa tiež môžu kedykoľvek odregistrovať. To umožňuje úplnú nezávislosť činnosti jednotlivých súčiastok. Každá súčiastka potrebuje poznať výlučne manažér udalostí. Manažér sa oboznámi s každou zaregistrovanou súčiastkou, ale vždy pozná len tie aktuálne zaregistrované.



Obrázok 3-14. Systém založený na udalostiach.

Smerovač

Smerovač aktívne mení smerovanie udalostí podľa zaregistrovaných súčiastok. Sám je zvyčajne tvorený jedinou súčiastkou, ale môže byť vytvorený aj ako distribuovaný. Pre každý typ udalosti môže byť vytvorený vlastný smerovač. Efektívnosť smerovača je podrobnejšie opísaná v kapitole Súčiastky založené na udalostiach.

Niektoré systémy, napríklad CORBA, realizujú smerovač tak, že vytvoria priamy komunikačný kanál medzi zdrojom a príjemcom udalosti. Tento prístup môže byť značne neefektívny pri veľkom množstve kanálov. A zaťažuje zdroj s veľkým množstvom prijímateľov, lebo musí poslať informáciu do každého kanálu zvlášť.

Smerovač môže vykonávať okrem základnej funkcie distribuovania udalostí aj ďalšie funkcie. Najčastejšie je to zabezpečenie vyrovnávacej pamäti pre udalosti. Odľahčuje tým prijímateľov od tejto starosti, ale zvyšujú sa jeho vlastné pamäťové aj výpočtové nároky.

Zaujímavou schopnosťou smerovača býva možnosť definovať si podmienky doručenia udalostí. Udalosti sú zvyčajne doručované v poradí, v akom boli vytvorené. Prijímateľ si však môže vyžiadať prioritne udalosti z určeného zdroja. Doručenie udalosti môže byť tiež viazané na vznik inej udalosti alebo parametrov udalosti.

Poslednou používanou schopnosťou smerovača býva transformácia správy. Častejšie sa správa upravuje priamo pri zdroji, ale ak je zdrojov s rovnakým tvarom správy viacero, jednoduchšie je použiť jednu súčiastku priamo pri smerovači. A podobne možno upravovať správy aj smerom k prijímateľom.

Cambridge Event Architecture

Cambridge Event Architecture (Bacon, 2000) je príkladom dobre prepracovaného systému založeného na udalostiach. Registrácia prebieha cez synchrónne rozhranie a zaregistrovať sa je možné nielen na typ udalosti, ale aj na podmienky jej poslania. Posielanie udalosti je vždy asynchrónne.

Smerovač používa filtrovanie na strane zdroja. Je to pomalšie, ale bezpečné a lepšie škálovateľné. Je možné aj filtrovanie na strane klienta. To je rýchlejšie, ale zaťažuje klienta a komunikáciu a klienti môžu dostať udalosť, na ktorú nemajú oprávnenie.

Zaujímavou vlastnosťou je, že môžeme pracovať so zloženými udalosťami. Klient je informovaný len keď vznikne istá skupina alebo postupnosť udalostí. Smerovač dokáže posielať a spracovávať aj informácie o vlastnej aktivite. Tým možno v komunikačnej sieti obchádzať výpočtové uzly, kde vznikla chyba. Tiež je možné presunúť časť smerovania z preťaženého uzla na menej zaťažený.

Posudzovanie systémov s udalosťami

Posudzovať ľubovoľný zložitý systém nie je nikdy jednoduché, lebo v každom špecifickom prípade je potrebné uvážiť, ktoré vlastnosti sú práve dôležitejšie a ktorým nie je potrebné priradiť až takú váhu. Preto je vhodné mať prehľad, aké vlastnosti by nás mali zaujímať a prečo.

Synchrónnosť riadiaceho toku. Tu sa myslí nezávislosť práce smerovača a ostatných súčiastok. Väčšina súčasných systémov je asynchrónna – oddeľuje prácu manažéra a príjemcu udalosti. Napriek tomu stále môžu existovať jednoduché synchrónne systémy kde stačí, keď sa spracuje len jedna udalosť v danom čase. Takýto prístup je vhodný, ak sa udalosti spracovávajú v rámci jedného procesu.

Statická alebo dynamická registrácia a možnosť odregistrovania sa. Jednoduchšie systémy môžu mať zadefinované pripojenia už v čase implementácie a nie je ich potrebné rekonfigurovať počas vykonávania aplikácie.

Časy rozdeľovania a spracovania udalostí. Čas rozdeľovania je potrebný na rozpoznanie vzniku udalosti, zistenie akého je typu a odovzdanie udalosti na spracovanie. Čas spracovania udalosti už závisí od príjemcu udalosti.

Spracovanie výnimiek. Či smerovač rozpozná neznámu alebo chybnú udalosť a čo s ňou vie urobiť. Chybné udalosti vznikajú zvyčajne poruchou hardvéru. Softvérová súčiastka môže tiež generovať udalosť, ktorú nemá zaregistrovanú. Udalosť je však jej hlavný výstup a preto sa tento problém zvyčajne vyrieši už počas vývoja súčiastky.

Škálovateľnosť. Závisí od modelu distribuovania udalostí a konštrukcie smerovača. Pri horizontálne distribuovaných smerovačoch sa záťaž zníži tým, že každá jednotka smerovača sa venuje len menšiemu počtu typov udalostí. Spracovanie je veľmi rýchle, ale je tu limit na maximálny počet jednotiek – toľko, koľko je typov udalostí. Vertikálne distribuované smerovače sú vlastne rozdelené hierarchicky. Znižuje sa tu počet príjemcov správy na jednotku smerovača. Nevýhodou je, že udalosť prechádza cez viacero smerovačov a narastá čas rozdeľovania udalostí.

Bezpečnosť. Jednotlivé typy udalostí a udalosti z rôznych zdrojov môžu mať rôzne úrovne oprávnenia. Príjemcovia udalostí môžu získať len tie správy, na ktoré majú oprávnenia.

Použitie vyrovnávacej pamäte. Zvyšuje šancu spracovania všetkých udalostí ale spomaľuje prácu manažéra. Je potrebné definovať čas „prežitia" udalosti, to znamená, po akom čase už nemá význam na udalosť reagovať.

Kvalita služieb. Pod týmto názvom sú zahrnuté ostatné vlastnosti systému, ako je spoľahlivosť doručenia udalostí, rekonfigurovateľnosť, automatické vyváženie záťaže smerovačov, možnosti úpravy správ a ďalšie.

Výhody a nevýhody systémov založených na udalostiach

Medzi výhody patrí vysoká rekonfigurovateľnosť týchto systémov. Súčiastky sa môžu pripájať a odpájať bez toho, aby to funkčne ovplyvnilo zvyšok systému. Tieto systémy výrazne podporujú znovupoužitie. Od súčiastky sa vyžaduje len správne rozhranie na príjem alebo posielanie udalostí a nie je dôležité ako je vytvorená ani v akom prostredí sa vykonáva.

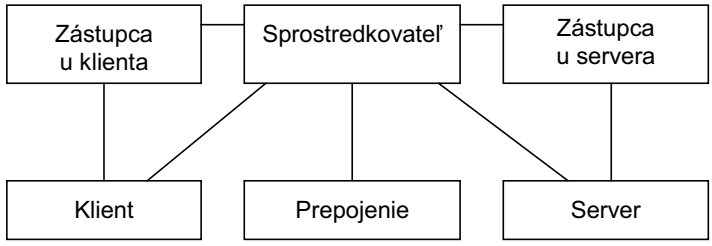
Výhodou je tiež, že súčiastka pozná len manažér udalostí a nemusí poznať nič iné. Údaje prichádzajú vo forme akú pozná a odosiela ich rovnako. Pri vytváraní súčiastky nie

je nutné vedieť, odkiaľ údaje prichádzajú, ani kam odchádzajú. Hlavnou nevýhodou týchto systémov je, že súčiastka nevie, či niekto na jej udalosť vôbec odpovie. Ak je potrebné zareagovať na každú udalosť, musí sa to realizovať iným typom systému. Súčiastka nevie, ako dlho budú trvať jednotlivé odpovede, preto je to nevhodné pri použití napríklad zdieľaných dátových štruktúr, kde sa výhody potlačia nútenou synchronizáciou.

Nevýhodou je, že komunikácia prebieha cez jedno hrdlo. Problémy nastávajú pri presunoch väčšieho množstva informácií vzťahujúcich sa k jednej udalosti. Toto nevie vyriešiť ani distribuovaný smerovač. Na prenos veľkého množstva údajov sa potom používa iná technológia, ale to už vytvára zložitý systém s viacerými technológiami.

3.5.3 Sprostredkovateľské (angl. broker) systémy

Sú určené hlavne na prepojenie vzdialených systémov. Jednotlivé objekty sú umiestnené v rôznych adresných priestoroch a navzájom komunikujú pomocou vzdialeného volania procedúr (angl. *remote method calls*). K najznámejším prostriedkom, nad ktorými sa vytvárajú takéto systémy, patria CORBA, OLE, DCOM, *Java Remote method invocation*.



Obrázok 3-15. Zjednodušený vzor Sprostredkovateľ.

Sprostredkovateľské systémy sú založené na návrhovom vzore Sprostredkovateľ, ktorého zjednodušená schéma je na obrázku 3-15.

Klient aj server sa musia najskôr zaregistrovať u sprostredkovateľa, potom už komunikujú prostredníctvom svojho zástupcu (angl. *proxy*), ktorý zabezpečuje komunikačné služby. Pre klienta sa jeho zástupca správa ako server a pre servera predstavuje zástupca klienta. Zástupcovia medzi sebou komunikujú cez prepojenie, ktoré bolo vytvorené pri zaregistrovaní oboch účastníkov komunikácie. Klient teraz môže vyslať požiadavku zavolaním obslužnej procedúry servera. Požiadavka sa dostane cez celý komunikačný kanál až k serveru, ktorý začne pracovať na odpovedi. Klient sa medzitým môže venovať inej činnosti. Keď server vytvorí odpoveď, pošle ju naspäť klientovi. Sprostredkovateľ zabezpečí, aby si klient odpoveď prevzal.

Výhody a nevýhody sprostredkovateľských systémov

Použitie sprostredkovateľských systémov prináša tieto výhody:

- systém je rozdelený na oddelené procesy; každý proces možno vyvíjať a spravovať nezávisle, čo znižuje zložitosť týchto úloh,
- server má vyššiu súdržnosť s klientmi,
- redukuje počet prepojení; medzi dvoma procesmi sa vytvára len jeden komunikačný kanál,

- zvyšuje abstrakciu návrhu; možno mať nezávislý pohľad na procesy, čím ďalej zlepšuje možnosti znovupoužitia,
- zvyšuje flexibilitu riešenia; do existujúceho systému je možné pridávať tak klientov ako aj servery a zvyšovať tak možnosti celého systému,
- jednoduchšia prenosnosť; keď zmeníme klienta, napríklad ho prenesieme na inú platformu, stačí len zmeniť *proxy* a zvyšok sa nemení.

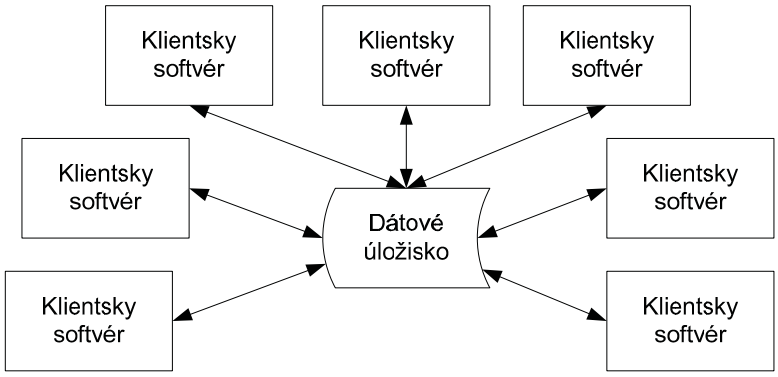
Nevýhody sprostredkovateľských systémov:

- *Limitovaná efektívnosť*. Rozhrania musia byť vytvárané tak, aby korektne zvládli chybné volania služby zo strany klienta. Istá kontrola chýb sa preto musí robiť pri každom volaní služby.
- *Nízky výkon*. Transparentnosť umiestnenia klientov aj serverov neumožňuje vybrať najkratšie alebo najrýchlejšie prepojenie medzi vybraným klientom a serverom.
- *Nízka odolnosť voči chybám*. Systém obsahuje veľa súčastí a každá je dôležitá. Chyba v ktorejkoľvek časti spôsobí nefunkčnosť celého systému.

Sprostredkovateľské systémy sa využívajú všade tam, kde je potrebné zabezpečiť odozvu na každú požiadavku. Dnes sú ale postupne nahrádzané webovými službami, ktoré majú väčšinu výhod uvádzaných pre sprostredkovateľské systémy a nemusia mať žiadnu z uvedených nevýhod.

3.6 Úložisko

Databázové systémy alebo nazývané aj systémy s úložiskom dát používajú centrálnu databázu na uloženie informácií súvisiacich s problémom. Dátové úložisko poskytuje vhodný prístup k informáciám, pričom zjednodušuje proces výberu dát v rôznych formátoch. Napríklad dáta je možné zobraziť formou zoznamu alebo v grafickej forme a pod. Obrázok 3-16. znázorňuje všeobecnú štruktúru takejto architektúry.



Obrázok 3-16. Štruktúra systému s centrálnym úložiskom dát.

Najdôležitejšími výhodami architektúry s centrálnym úložiskom dát je:

- *Dátová integrita* (angl. *Data Integrity*) – údaj sa vloží raz, kedykoľvek; je odstránená chybovosť a duplicita údajov.

- *Znovupoužitie návrhu* (angl. *Design Reuse*) – presné a spoľahlivé údaje sú v prípade potreby k dispozícii.
- *Generovanie pohľadov* (angl. *View Generation*) – jediný zdroj údajov umožňuje generovať rôzne pohľady.
- *Flexibilita procesu* (angl. *Process Flexibility*) – proces riadenia údajov nie je viazaný na aplikáciu.
- *Nezávislosť interakcie dát od aplikácie* (angl. *Data Interaction Independent of Application*) – dáta sú používateľovi prístupné z viacerých aplikácií.
- *Škálovateľnosť* (angl. *Scalability*) – databáza môže narastať v súlade s potrebami aplikácie a domény.

3.6.1 Databázové systémy

Databáza predstavuje množinu dát, ktoré sú navzájom v určitom vzťahu. Databáza má nasledujúce vlastnosti (Elmastri, 2003):

- je to logicky súdržná množina dát s istým vnútorným významom,
- je navrhnutá, vytvorená a naplnená dátami s určitým cieľom, pre cieľovú skupinu používateľov a aplikáciu,
- reprezentuje isté aspekty reálneho sveta tzv. minisveta (angl. *Universe af Discourse*),
- zmeny v „minisvete“ sa odrážajú v databáze.

Systémy riadenia databázy

Systémy riadenia databázy (angl. *Database Management System*; DBMS) poskytujú úložiská (angl. *repository*) pre skladovanie a riadenie dát. DBMS poskytuje operácie pre prácu s uloženými dátami a zvyčajne realizuje paradigmu abstraktnej dátovej štruktúry, ktorej základným predpokladom je oddelenie logickej definície dát od samotnej fyzickej implementácie. Samotná logická štruktúra databázy predstavuje koncept logického dátového modelu (angl. *logical data model*).

Tieto systémy predstavujú účelové softvérové systémy navrhnuté s cieľom spravovať veľké množstvo údajov. Poskytujú:

- *perzistenciu* – dáta spravované v DBMS majú byť perzistentné,
- *ochranu dát pred neautorizovaným a chybným prístupom* – DBMS by mali podporovať mechanizmy, ktoré zabránia narušeniu konzistencie dát, mechanizmy na znovunadobudnutie a zálohovanie pred prípadným zlyhaním hardvéru a bezpečnostné mechanizmy, ktoré zabránia neautorizovaným prístupom,
- *sekundárny manažment dát* – DBMS spravuje veľké objemy údajov. Preto používa rôzne techniky na ich organizáciu ako indexáciu (angl. *indexing*), klasterizáciu (angl. *clustering*) a alokáciu zdrojov (angl. *resource allocation*),
- *kompiláciu a optimalizáciu* – DBMS musia poskytovať mechanizmy na prenos medzi aplikáciami a externými a logickými úrovňami,
- *rozhrania*, ktorými definujú štruktúru dát (DDL - Data Definition Language) a manipuláciu s nimi (DML - Data Manipulation Language),

– *distribúciu* – transparentný prístup k lokálnym zdrojom.

Prístup k dátam a ich získavanie sa uskutočňuje prostredníctvom transakcií. V jednoduchom prípade poradie aktualizácie databázy udáva poradie transakcií. Moderné databázové systémy ako DB2, Oracle, Sybase a MS SQL Server podporujú súbežný transakčný proces.

Objektovo orientované databázové systémy

Objektovo orientované databázové systémy (angl. *Object-Oriented Database System*; OODBS) boli vyvinuté s cieľom podporiť funkcionalitu niektorých tried aplikácií, ktorá si vyžaduje pokročilejší spôsob štruktúry dát ako pri relačných databázových systémoch. Výskum OODBS sa začal v 80. rokoch a viedol k vývoju takého systému ako je O2. Podporuje dátový model založený na množine objektov (inštancie s vlastnou identitou a stavom). Naviac (voči funkcionalite tradičného DB systému) musí OODBS obsahovať nasledovné objektovo-orientované črty (Atkinson, 1992):

– *Zložené objekty* (angl. C*omplex Objects*): OODBS musia podporovať tvorbu zložených objektov z jednoduchých aplikovaním vhodných konštruktorov. Zároveň jednoduché operácie ako kopírovanie alebo mazanie musí byť možné aplikovať na tieto zložené objekty.

– *Identita objektu* (angl. *Object Identity*): Dátový model musí podporovať identitu objektu tak, aby objekt existoval nezávisle od hodnoty jeho stavu. Dva objekty môžu byť identické (sú jedným a tým istým objektom) alebo rovnaké (hodnota ich stavu je rovnaká).

– *Zapuzdrenie objektu* (angl. *Object Encapsulation*): Objekt zapuzdruje oboje program aj dáta. Objekt v OODBS má teda aj dátovú aj operačnú časť.

– *Typy a triedy* (angl. *Types and Classes*): V závislosti od zvoleného priblíženia by OODBS mali podporovať buď pojem typu alebo triedy.

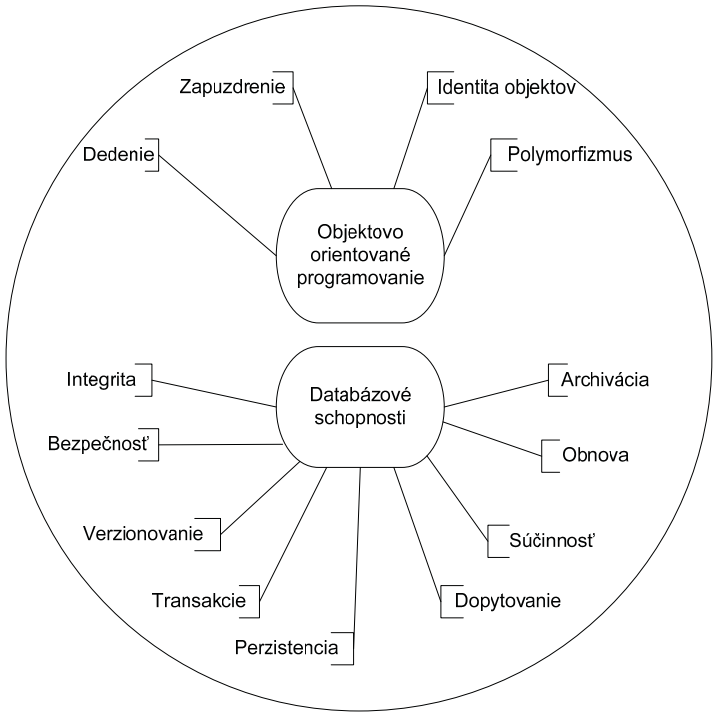
– *Hierarchie typov alebo tried* (angl. *Type or Class Hierarchies*): OODBS musia podporovať dedenie.

– *Prekonanie, preťaženie a neskoré viazanie* (angl. *Overriding, Overloading and Late Binding*): OODBS musia umožňovať predefinovať operácie určitých typov (prekonanie), použiť rovnaké mená pre rôzne implementácie (preťaženie) a vybrať vhodnú implementáciu počas vykonávania programu (neskoré viazanie).

– *Úplnosť výpočtu* (angl. *Computational Completeness*): Jazyk pre manipuláciu s dátami, ktorý sa používa v rámci OODBS, musí umožňovať vyjadriť akúkoľvek výpočtovú funkciu.

– *Rozpínavosť* (angl. *Extensibility*): OODBS musia umožňovať definovať nové typy, ktoré pochádzajú z preddefinovaných typov a nesmú rozlišovať typy definované systémom a typy definované používateľom.

Objektovo orientované databázové systémy môžu obsahovať ďalšie rozšírenia ako napríklad kontrola typov, zložené transakcie, mechanizmy pre podporu verziovania.



Obrázok 3-17. Charakteristické črty objektovo-orientovaného programovania a databázových technológií.

Aktívne databázové systémy

Tradičné relačné a objektovo-orientované databázové systémy sú pasívne: vykonávajú akcie prostredníctvom explicitných operácií dopytu a aktualizácie (angl. *query and update*). Podľa (Dittrich, 1995, Widom, 1996) databázový systém nazývame aktívnym, ak okrem „normálnej" databázovej funkcionality je schopný samostatne reagovať na situácie definované používateľom a následne vykonávať používateľom definované činnosti. Základ aktívnych databázových systémov (angl. *Active Database System;* ADBS) tvoria ECA-pravidlá (angl. *event-condition-action rules*), tiež nazývané spúšťače (angl. *triggers*), ktorými sa špecifikuje správanie systému. Vo všeobecnosti ECA-pravidlo znamená:

Ak sa definovaná udalosť vyskytne a definovaná podmienka je platná, potom vykonaj definovanú činnosť.

Najväčším rozšírením voči funkcionalite tradičných databázových systémov je vloženie definície udalosti a pravidla ako aj vykonanie tohto pravidla. Pravidlá ADBS pozostávajú z troch častí:

- udalosť (angl. *event*) – zapríčiňuje spustenie pravidla,
- podmienka (angl. *condition*) – je overená pri spúšťaní pravidla,
- činnosť (angl. *action*) – vykoná sa pri spustení pravidla a zároveň kladnom vyhodnotení podmienky.

ADBS obsahuje bázu pravidiel. Ak sú pravidlá definované, potom ADBS monitoruje relevantné udalosti. Pre každé pravidlo, ktorého udalosť sa vyskytne, ADBS vyhodnotí jeho

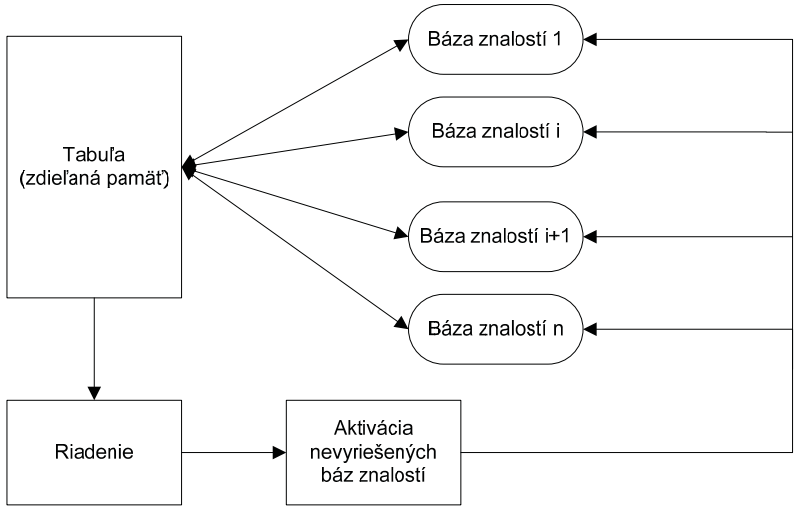
podmienku; ak je podmienka splnená, vykoná sa činnosť. Pre jednu udalosť môže byť vykonaných viac pravidiel.

3.6.2 *Tabuľové systémy*

Tabuľová architektúra napodobňuje odborníkov okolo tabule s kriedou. Odborníci riešia problém napísaný na tabuli pripísaním vlastnej informácie. Niektorí prispievajú hneď, iní musia čakať, aby nadviazali na iného. Každý príspevok môže a nemusí umožniť ostatným, aby sa vyjadrili a tak posunúť problém k vyriešeniu. Proces pokračuje pokým všetci neodsúhlasia, že sa podarilo dosiahnuť riešenie.

Tabuľa predstavuje databázu a odborníci softvérové procesy. Každý proces dokáže čítať tabuľu a informácie na nej. Ak má proces dostatok informácií na vykonanie výpočtu, vykoná ho a výsledok zobrazí na tabuľu. Jeden alebo viac procesov priebežne testuje obsah tabule, či už bolo dosiahnuté riešenie.

Technológia tabuľových systémov (angl. *Blackboard System*) bola vyvinutá začiatkom 70-tych rokov. Štruktúra tabuľového systému je znázornená na obrázku 3-18.



Obrázok 3-18. Schéma tabuľového systému.

Bázy znalostí (angl. *knowledge source*) predstavujú nezávislé procesy, obsahujúce počítačový kód a špecifickú znalosť. Báza znalostí reaguje na zmeny obsahu na tabuli. Tabuľa (angl. *blackboard*) je zdieľanou pamäťovou štruktúrou, ktorá obsahuje vnútornú reprezentáciu problému alebo stav aplikácie. Báza znalostí komunikuje s tabuľou posielaním a získavaním informácií. Zmenu dát na tabuli zachytáva Riadenie (angl. *control*), rozlišuje danú bázu znalostí a vytvára z nich rad, v akom budú aktivované na súčinnosť s tabuľou.

3.7 Architektúry súbežného softvéru

Rôzne architektúry sú aplikovateľné na rozličné triedy problémov. Niektoré problémy, či už z hľadiska svojej povahy alebo z dôvodov vyplývajúcich z prostredia, do ktorého budú nasadené, sú vhodnými kandidátmi na súbežné spracovanie vo viacerých vláknach či procesoch.

Súbežný systém je z pohľadu jedného z (jeho) procesov viditeľný ako ďalšie procesy, s ktorými proces komunikuje, ale nevie o nich nič – ako rýchlo a kedy sa spúšťajú. Takýto súbežný systém má teda viacero častí (procesov) vykonávajúcich sa súčasne.

Stupeň súbežnosti je počet paralelných operácií vykonateľných naraz. Je založený na jednotkách granularity. Pri skutočnom paralelizme je limitovaný počtom fyzických procesorov. Činnosti, ktoré sa môžu v čase prekrývať vo svojom vykonávaní, predstavujú možný paralelizmus, tzv. *súbežné činnosti*. Súbežné činnosti sa môžu, ale *nemusia* vykonávať súčasne. Z hľadiska granularity pritom rozoznávame:

- granularitu jemnozrnnú – napr. na úrovni príkazov,
- granularitu hrubozrnnú – napr. na úrovni procesov.

3.7.1 Paralelizmus na úrovni príkazov

Pri jemnej úrovni granularity môžeme hľadať optimalizáciu spracovania údajov paralelizovaním vykonávania programu, čím sa dosiahne spracovanie viacerých inštrukcií programu za kratší čas. Príkladom môžu byť nasledovné riadky kódu (pozri príklad 3-3).

```
X := 2;              :P
Y := A = B;          :Q
Z := sin(30.0);      :R
```

Príklad 3-3. Séria inštrukcií vhodná na paralelné spracovanie.

Jednotlivé riadky programu – P,Q,R od seba navzájom nezávisia a môžu byť spracované paralelne. Ak by sa každý vykonával za rovnakú jednotku času, pri ich súbežnom spracovaní by klesla časová náročnosť tohoto programu na tretinu.

3.7.2 Paralelizmus na úrovni procesov

Pri hrubej granularite sa zameriavame na paralelizmus vo vykonávaní programu na úrovni samostatne vykonávaných postupností inštrukcií, o čom pojednáva táto podkapitola.

Proces, vlákno

Proces je artefaktom operačného systému. Jeden proces však môže prebiehať vo viacerých vláknach (angl. *threads*). Výhodou týchto vlákien je nižšia réžia pri ich prepínaní, pretože sa spúšťajú v spoločnom adresnom priestore. Na základe týchto vlastností sa vlákna zvyknú nazývať aj odľahčené procesy. Na druhú stranu vyplýva z použitia spoločného pamäťového priestoru menšie vzájomné zabezpečenie vlákien.

Na základe paralelizácie vykonávania poznáme:

- *multiprogramové systémy* – systémy, na ktorých je spustených viacero programov naraz, napr. Unix – programy sa spúšťajú ako samostatné procesy.
- *multivláknové systémy* – samotný program sa vykonáva pomocou viacerých vlákien naraz.

Pri multiprogramových systémoch sa najčastejšie vykonávanie viacerých programov (procesov) naraz realizuje prerušovaním a prepínaním vykonávania týchto procesov na procesore. Cieľom je dosiahnuť, aby sa procesu javilo, akoby sa vykonával na procesore jediný a zároveň, akoby sa vykonávali viaceré vlákna naraz, pričom vďaka prepínaniu sa vyko-

náva v jednom okamihu v skutočnosti len jedno – *pseudoparalelizmus*. V súvislosti s prepínaním vlákien (resp. procesov) je potrebné riešiť aj prístup k zdrojom systému vzájomným vylúčením a synchronizačnými mechanizmami, ktoré majú rôzne jazyky rozpracované na rôznej úrovni.

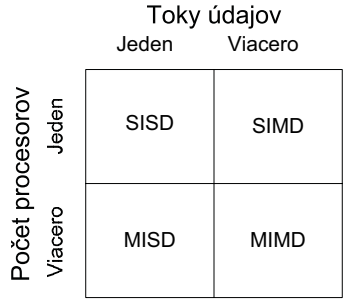
Vlákna sa používajú v rámci vykonávania procesu napr. pri čakaní na vstup, zápis na disk a pod. Svoje využitie majú napr. na serveroch, kde sa môže pre každého používateľa vytvoriť samostatné vlákno. Poznáme nasledujúce typy vlákien (Robbins, 1996):

- *používateľské vlákna* – jadro operačného systému (OS) o nich nevie; manažment vlákien je vykonávaný pomocou knižníc; majú vlastné plánovanie a na prepínanie nie sú potrebné privilégiá jadra,
- *vlákna jadra* – komunikuje sa s nimi cez API (angl. *Application Programming Interface*), udržiavajú informácie o kontextoch; majú na starosti aj plánovanie, prepínanie a blokovanie vykonávajúcich sa vlákien,
- *hybridné vlákna* – vytvárajú sa v používateľskom kontexte, ale spracúva ich jadro.

3.7.3 Paralelné programovanie

Pri paralelnom programovaní sa autori programov pokúšajú využiť paralelizmus (súbežnosť) na zefektívnenie činnosti programov, pričom sa využívajú napr. zmeny štruktúry riešenia, zisk z mapovania softvérovej architektúry na hardvér, či paralelizácia vzhľadom na povahu vykonávaných činností – čakanie na vstup a pod.

Z hľadiska paralelizmu spracovania a údajov rozlišujeme nasledujúce tradičné architektúry (obrázok 3-19).



Obrázok 3-19. Tradičná klasifikácia architektúr.

Jednotlivé architektúry majú nasledujúcich typických predstaviteľov, resp. typické riešené úlohy:

- SISD (angl. *Single Instruction Single Data*) – tradičné PC,
- SIMD (angl. *Single Instruction Multiple Data*) – vektorové PC,
- MISD (angl. *Multiple Instruction Single Data*) – viaceré činnosti nad jednými údajmi,
- MIMD (angl. *Multiple Instruction Multiple Data*) – riešenie nezávislých úloh, resp. úloh rozdeliteľných do samostatných procesov.

Jednoduché paralelné problémy, nazývané až triviálne, či „trápne“, sa veľmi dobre nasadzujú na paralelné spracovanie. Majú totiž súbežné riešenie pomocou sady nezávislých

úloh. Jedinou úlohou návrhu je rozdeliť úlohy, aby skončili približne naraz a rovnomerne pokryť zaťaženie procesorov.

Takmer „trápne“ problémy majú o 2 kroky navyše – distribúcia inicializačných údajov na procesory a zozbieranie výsledkov do konečného výsledku.

V súčasnosti je značným trendom pre náročné či rozsiahle paralelizovateľné úlohy zavedenie *výpočtov v sieti* (angl. *grid computing*). Keďže sa využíva aj na výpočty na používateľských staniciach, na ktorých ich majitelia dobrovoľne poskytnú výpočtový čas, kladie sa dôraz na bezpečnosť. Nezanedbateľnými sú aj kritériá rýchlosti a zopakovateľnosti výsledkov. Okrem uvedených spôsobov existujú aj rôzne variácie, napríklad:

- akumulácia pomocou zdieľanej údajovej štruktúry, keď procesy potrebujú údaje z čiastočných riešení, napríklad spracovanie bankomantových transakcií nad účtom klienta,
- iné ako úplné riešenie – riešenie bez splnenia všetkých úloh; používa sa v prípade, ak nie je možné určiť, že existuje úplné riešenie a treba určiť, kedy riešenie postačuje,
- čiastočná definícia problému – nie je možné určiť všetky úlohy na začiatku, tvoria sa dynamicky. Ťažko je určiť podmienku ukončenia.

3.7.4 Údajovo paralelné systémy

Pri údajovo-paralelných systémoch ide o systémy, v ktorých sa vykonávajú rovnaké operácie nad všetkými elementmi údajovej štruktúry. Komunikácia a synchronizácia pritom zostávajú implicitné (neviditeľné). Napr. pri násobení matíc: $C = A \times B$ sa zdrojové matice nemenia a vykonáva sa stále ten istý program pre rôzne prvky matice.

Pri údajovo-paralelnom modeli všetky procesory vykonávajú ten istý program. Všetky údajové štruktúry sú priradené do obdĺžnikovej mriežky virtuálnych procesorov. Virtuálne procesory sú namapované na fyzickú štruktúru, na jeden fyzický procesor môže byť namapovaných viacero virtuálnych procesorov.

Údajovo-paralelné jazyky

Na paralelné programovanie slúžia programovacie jazyky, ktoré obsahujú dodatočné rozšírenia pre prácu s paralelizmom a paralelnými výpočtami. Zástupcami sú napr.:

- High Performance Fortran – sada rozšírení do Fortran-u 90. Špecifikuje, ako sú údaje rozdelené medzi procesory. Nešpecifikuje sa nízkoúrovňová komunikácia.
- Data Parallel C Extensions (DPC) – používa šablóny (tvar, angl. *shape*). Každá štandardná premenná v jazyku C sa dá špecifikovať cez DPC. Šablóna musí mať stupeň (počet rozmerov), rozmery a rozloženie (ako je paralelný objekt distribuovaný), pozri nasledujúcu ukážku programu pre paralelné spracovanie v DPC.

```
shape [20][20] S;
main()
{
  int sum;
  int:S a;
  a = 1;
  sum +=a ;
}
```

– Parallel Extensions to .NET – paralelné rozšírenia pre .NET (verzia 4.0) o.i. pridávajú do jazyku C# koncept malých blokov nezávislého kódu tzv. úloh (angl. *task*), ktoré je možné vykonať súbežne pomocou príkazu paralelného cyklu (angl. *parallel for*). Tiež, rozšírenia obsahujú sadu dátových štruktúr pre súbežné spracovanie.

3.7.5 Systémy zasielajúce správy

Do tejto kategórie patria sekvenčné, pokiaľ možno nezávislé programy, ktoré komunikujú pomocou volaní *send* a *receive*, ktoré bývajú synchrónne alebo asynchrónne. Obyčajne sa realizuje jeden program n-krát, t.j. ide o SPMD (jeden program, viacero údajov, angl. *single program, multiple data*) programovanie. Komunikácia prebieha podľa modelu zasielania správ:

– množina procesov používa lokálnu pamäť,
– procesy komunikujú posielaním a prijímaním správ,
– prenos údajov vyžaduje spoluprácu (poslanie a prijatie).

Najpopulárnejšími verziami systémov zasielajúcich správy sa stali rozhranie zasielania správ a paralelný virtuálny stroj, lebo nevyžadujú modifikáciu hostiteľského programovacieho jazyka.

Rozhranie zasielania správ (angl. *Message Passing Interface* – MPI)

Je to definícia portovateľnej knižnice na zasielanie správ. Táto špecifikácia má aj svoju implementáciu v programovacom jazyku C. Aplikáciu napísanú pomocou tejto knižnice treba vidieť ako zbierku súbežných komunikujúcich programov, ktoré sa vykonávajú na jednom alebo viacerých procesoroch. Každý program má stupeň a ID úlohy.

Programy založené na MPI sú programy napísané v tradičných jazykoch (C, ...) rozšírené o príkazy definície prostredia a komunikácie. Komunikácia v rámci aplikácie je riadená pomocou konceptov komunikátorov. Komunikátor (angl. *communicator*) vytvára komunikačný kanál medzi skupinami úloh:

– *intrakomunikátor* (angl. *intracommunicator*) – prepája úlohy jednej skupiny. Predvoleným typom je *MPI_COMM_WORLD* (MPI komunikátor pre svet) na komunikáciu so všetkými dostupnými úlohami,
– *interkomunikátor* (angl. *intercommunicator*) – komunikácia medzi úlohami dvoch alebo viacerých skupín.

V rámci komunikácie sa používajú typované správy zasielané synchrónnymi aj asynchrónnymi spôsobmi. Používajú sa virtuálne topológie – mapovanie úloh na problém a algoritmy.

Paralelný virtuálny stroj (angl. *Parallel Virtual Machine* – PVM)

Je to programový systém zasielajúci správy. Je určený na vývoj a prevádzku veľkých súbežných aplikácií, ktoré pozostávajú z interagujúcich nezávislých súčiastok.

Pozostáva z knižnice funkcií a démona (pvmd – *PVM daemon*). Jeden program sa spustí ručne a volá ostatné (majú ID). Komunikácia potom prebieha asynchrónnym zasielaním, pričom príjem môže byť blokujúci. PVM potrebuje explicitnú distribučnú a komunikačnú schému.

Programovacím jazykom tohto stroja je Fortran D s podporou viacerých nástrojov na vývoj a ladenie paralelných programov, napr. D Editor poskytuje spätnú väzbu o paralelizme a komunikácii; dPablo Performance Browser zobrazuje a spracováva nazbierané informácie o výkone.

3.7.6 Metodológia paralelného programovania

Paralelné programovanie je činnosť veľmi náročná na autorovu tvorivosť a predstavivosť. Postupom času bola vytvorená metodológia, ktorá uľahčuje vykonanie niektorých činností. Ideálnym zavedením paralelizmu by bolo jeho použitie bez ohľadu na hardvérovú architektúru a programovací jazyk. V praxi však treba brať všetky tieto faktory do úvahy a preto je dôležitý aj výber programovacieho jazyka a k nemu najvhodnejšej architektúry.

- Prvou etapou návrhu je výber modelu paralelného programovania. Opisuje mapovanie jednotiek súbehu (paralelne vykonateľných sekvencií operácií) na výkonné jednotky. Treba pritom brať ohľad aj na obmedzenia cieľovej platformy. V návrhu treba použiť dobrý paralelný algoritmus – abstraktný, efektívny, adaptovateľný.
- Dekompozícia problému a implementácia algoritmu – pri dekompozícii sa snažíme o vytvorenie takých prúdov inštrukcií, ktoré budú mať medzi sebou čo najmenšiu interakciu. Pre vzájomný prístup k informáciám stanoviť pravidlá a zabezpečovacie mechanizmy. Dekompozícia pozostáva z dvoch hlavných častí:
  - dekompozícia úloh – úlohou je zoskupiť operácie do výkonných entít – procesov, úloh a distribuovať prácu po úlohách za účelom dosiahnutia najvyššej vyťaženosti,
  - dekompozícia údajov – preskúmanie štruktúr a vzorov použitia údajov v programe. Určenie logických hraníc medzi podmnožinami.
- Výber štruktúry algoritmu – vykoná sa na základe počtu úloh, ich poradia a spôsobu zdieľania údajov. Hlavné štruktúry algoritmov:
  - organizácia podľa poradia – dobre definovaná interakcia skupín úloh, rozhodujúce je ich vzájomné poradie, napr. MVC,
  - organizácia podľa úloh – aktívna je naraz iba jedna skupina a interakcie sú v nej.

3.8 Výzvy softvérovej architektúry

Vedecká komunita a komerčné firmy sa problematike architektúry softvéru venujú už niekoľko rokov. Táto téma je teda značne rozsiahla a dobre zdokumentovaná. Avšak veľká časť výskumu bola zameraná predovšetkým na modelovanie a opis architektúry softvéru. Problematika vyhodnocovania a porovnávania jednotlivých architektúr bola zatiaľ pokrytá len v obmedzenej miere. To znamená, že sme schopní navrhnúť a namodelovať zložitú architektúru, ale nie sme schopní exaktne vyhodnotiť vhodnosť použitia navrhnutej architektúry.

Rozvoj v rámci návrhu a modelovania architektúry softvéru viedol k vývoju softvéru vedenému architektúrou (angl. *architecture-driven software development*). Vývoj softvéru vedený architektúrou mal však za následok, že softvér sa stal veľkým a zložitým. Máme teda veľký počet implementácií vychádzajúcich z rôznych architektúr avšak bez možnosti

vyhodnotiť úspešnosť a vhodnosť jednotlivých implementácií vzhľadom na použitú softvérovú architektúru. V ďalšom texte sa teda budeme venovať problematike vyhodnocovania softvérovej architektúry.

3.8.1 Opis softvérových architektúr

Pôvodne sa na kreslenie architektúry systému používali jednoduché kresliace nástroje. Softvérový architekt zakreslil architektúru systému pomocou niekoľkých škatuliek a čiar, ktoré reprezentovali jednotlivé súčiastky a prepojenia. Opísanie architektúry týmto spôsobom sa obmedzuje len na syntaktické vyjadrenie architektúry a neobsahuje sémantiku jednotlivých súčiastok a prepojení. Tieto neformálne diagramy však nemohli byť formálne analyzované s cieľom posúdiť konzistentnosť, úplnosť a správnosť architektúry.

Tento stav bol východiskom pre hľadanie nových možností, ktoré boli zamerané buď na definovanie opisných jazykov pre architektúru alebo na rozšírenia existujúceho jazyka UML (*Unified Modelling Language*). Opisné jazyky pre architektúru (angl. *Architecture description languages – ADLs*) sa však v komerčnom sektore zatiaľ nerozšírili a v rámci rozšírení UML jazyka nie je zatiaľ k dispozícii modul, ktorým by bolo možné plnohodnotne architektúru softvéru opísať.

3.8.2 Jazyky na opis architektúr

Existuje viacero snáh o vytvorenie všeobecne akceptovateľného jazyka na opis architektúry. Aj organizácia DARPA financovala snahy o vytvorenie ADL podporujúceho syntax aj sémantiku v rámci návrhu architektúry. Existuje niekoľko základných požiadaviek na ADL. A to, že by mal opisovať vysokoúrovňovú štruktúru systému, mal by umožniť opis statických vlastností a tiež podporovať formálnu analýzu. Ako príklady možno spomenúť opisné jazyky ako Acme, C2, Wright, ArTek a mnohé ďalšie ako to je možné vidieť v tabuľke 3-1.

Tabuľka 3-1. Príklady opisných jazykov pre architektúry.

ADL	Developer	Použitie
Acme	Carnegie-Mellon University	Podporuje integráciu viacerých modelov napísaných v rozdielnych ADL jazykoch.
C2	USC ISI	Vyvinutý špeciálne pre príkazové a riadiace architektúry.
MetaH	Honeywell	Podporuje opis funkcionálných systémov.
ArTek	Teknowledge Corp.	Umožňuje opis štruktúry veľkých systémov, ktoré podporujú koordináciu a komunikáciu v rámci veľkých projektov.
Darwin	Magee	Zameriava sa na opis dynamických, konkurentných a distribuovaných systémov.
Rapide	Stanford University	Ide o vykonateľný ADL jazyk pre prototypovanie, simulovanie a analýzu softvérových systémov.
SADL	SRI	Podporuje opis štruktúry a sémantiky prostrednítvom explicitného mapovania.
UniCon	Carnegie-Mellon	Podporuje vytváranie architektúry prostredníctvom prepájania používateľom definovaných súčiastok.
Wright	Carnegie-Mellon University	Zameriava sa na formálne modelovanie typov konektorov a ich prepojenia v rámci architektúry.

Veľký počet rôznych opisných jazykov je spôsobený tým, že jednotlivé tímy vytvárali jazyk pre určitú sadu architektúr avšak nikdy sa nepodarilo dostatočne pokryť všetky architektúry. Preto bolo potrebné vytvoriť ďalší opisný jazyk, ktorý by bol vhodný pre konkrétnu architektúru.

Ako to bude vidieť na konkrétnom príklade opisného jazyka, v niektorých prípadoch sa opisný jazyk v značnej miere podobá na programovací jazyk. S tým súvisí problémovosť štandardizácie a kombinovania výsledkov. Bolo teda vytvorené veľké množstvo opisných jazykov, ktoré nie je možné vzájomne prepojiť.

3.8.3 Príklad opisného jazyka architektúry

V tejto kapitole uvedieme jednoduchý príklad jazyka pre opis architektúry klient-server. Najprv opíšeme rozhrania servera, potom rozhrania klienta a nakoniec prepojenia medzi nimi.

Server

V opisnom jazyku je najprv uvedený opis rozhraní, potom opis obmedzení a nakoniec opis správania. Opis servera obsahuje žiadosti o inicializáciu, ktorú je možné prijať prostredníctvom operácie `Initialize()`. Prijatie žiadosti o výpočet je zrealizovateľné prostredníctvom operácie `Compute()`. Súčiastka server vráti výsledok prostredníctvom operácie `Result()`. Server má definované jedno obmedzenie a na konci v rámci príkladu 3-4 je opis správania. Pri inicializácii sa vytvorí príslušný objekt. Pri žiadosti o výpočet server vráti výsledok.

```
type Server is interface
action     in Initialize();
           in Compute(Value: Float);
     out Result(Value: Float);

constraint
     match Start -> Initialize'Call -> (Compute'Call *);

behavior
     NewValue : var Float;
begin
     (?x in Float) Compute(?x) => Result($NewValue);;
end Server;
```

Príklad 3-4. Príklad ADL kódu pre server.

Klient

Klient má rozhranie na prijatie výsledku `Result()`. Klient dokáže inicializovať pripojenú súčiastku a zároveň môže požiadať o vykonanie výpočtu `Calculate()`. Správanie klienta je definované ako žiadosť o výpočet.

```
type Client is interface
action     in Result(Value: Float);

     out Initialize();

     out Calculate(Value: Float);
```

```
behavior
     InitialValue : var Float := 0.0;
Begin

     Start => Initialize;
     Calculate($InitialValue);;

end Client;
```

Príklad 3-5. Príklad ADL kódu pre klienta.

Prepojenie

Vyššie sme opísali súčiastky Server a Klient. Nakoniec je potrebné oba súčiastky prepojiť. Prepojenie definuje túto postupnosť udalostí:

1. klient pošle požiadavku,
2. server vykoná operáciu,
3. server vráti výsledok,
4. klient prijme výsledok.

Jednotlivé kroky sú formálne zapísané v opisnom jazyku architektúry v rámci príkladu 3-6, ktorý je uvedený nižšie.

```
architecture ClientServer() return root is
C : Client;
S : Server;
Connect

     (?x in Float) C.Calculate(?x) => S.Compute(?x);
     (?y in Float) S.Result(?y) => C.Result(?y);

end ClientServer;
```

Príklad 3-6. Príklad ADL kódu pre prepojenie.

3.8.4 Dokumentácia

Veľké množstvo softvérových projektov býva nedostatočne zdokumentovaných. Nedostatky je možné nájsť v používateľskej dokumentácii ako aj v technickej dokumentácii pre vývojárov. Štandardizácia dokumentácie na úrovni architektúry by mohla prispieť v možnosti porovnať jednotlivé architektúry.

3.8.5 Návrh

Pri návrhu architektúry softvéru existuje výskumná snaha zamerať sa na známe atribúty kvality softvéru, akými sú napríklad:

- škálovateľnosť,
- transparentnosť,
- integrovateľnosť súčiastok,
- rozširovateľnosť,
- flexibilita.

Je zrejmé, že nie je možné dosiahnuť všetky atribúty kvality naraz. Atribúty kvality, na ktoré sa kladie pri návrhu dôraz, vychádzajú z požiadaviek, ktoré sú kladené na vyvíjanú architektúru.

V posledných rokoch je možné pozorovať odklon od vývoja monolitického softvéru smerom k softvéru založenom na súčiastkach. Prepojenie súčiastok v rámci zložitého systému vychádza z navrhnutej architektúry. Aby však jednotlivé súčiastky mohli spolu komunikovať je potrebné mať k dispozícii monolitický spájajúci softvér. Problematike spájajúceho softvéru sa venuje kapitola 2.2.

Ako bolo spomínané v úvode, jednou zo základných výskumných výziev je vytvorenie postupov pre porovnávanie architektúr. Čiže existuje snaha o vytvorenie reflektívnej architektúry.

3.8.6 Analýza softvérových architektúr

Ďalším krokom pri vytváraní architektúry softvérového systému je analýza navrhnutej architektúry. Pri návrhu boli stanovené základné atribúty kvality. Nie všetky sú však pre konkrétny systém rovnako dôležité a niektoré si vzájomne odporujú a preto každý návrh vyžaduje kompromisy.

Je možné teda navrhnúť niekoľko alternatívnych architektúr, ktoré by boli vhodné pre vyvíjaný systém. Každá z navrhnutých architektúr je však zameraná na iné atribúty kvality. Pre výber finálnej architektúry je potrebné rozhodovať sa medzi navrhnutými alternatívami. Na podporu voľby medzi navrhnutými architektúrami je možné použiť niekoľko druhov analýz:

- štrukturálna analýza,
- analýza systémových atribútov,
- analýza výkonnosti.

Štrukturálna analýza

Štrukturálna analýza architektúry sa zameriava na analýzu funkcionálnych častí vyvíjaného systému. V prvom kroku sa celková funkcionalita systému rozloží na množiny jednotlivých funkcií. Štruktúru systému môžeme chápať ako súbor častí a prepojení medzi nimi. Keďže funkcionalita systému má byť realizovateľná na navrhovanej architektúre, musí existovať prepojenie medzi štrukturálnymi časťami architektúry a funkcionalitou, ktorú má systém poskytovať. Je potrebné namapovať funkcionalitu systému na štrukturálne časti systému.

Štrukturálna analýza predstavuje statický pohľad na architektúru, kde nás zaujíma topológia architektúry a nie správanie systému. Touto statickou analýzou je možné skontrolovať konzistenciu medzi súčiastkami a ich konektormi. Štrukturálnou analýzou je možné skontrolovať úplnosť návrhu architektúry a to na základe analýzy mapovania štrukturálnych častí architektúry a funkcionality systému. Očakáva sa, že každá súčiastka implementuje aspoň jednu funkciu a tiež, že každá súčiastka má aspoň jeden konektor na inú súčiastku. Týmto spôsobom je možné nielen odhaliť nepresnosti a chyby v rámci štrukturálnych častí architektúry, ale je tiež možné odhaliť chýbajúcu funkcionalitu systému.

Analýza systémových atribútov

Ďalším typom analýzy architektúry je analýza systémových atribútov. Systémové atribúty môžeme rozdeliť do troch základných skupín: topologické atribúty kvality, atribúty kvality týkajúce sa správania sa systému a atribúty vzťahujúce sa na systém ako celok verzus jednotlivé časti systému.

Medzi topologické atribúty kvality môžeme zaradiť:

- modifikovateľnosť,
- prenosnosť,
- integrovateľnosť,
- testovateľnosť,
- znovupoužiteľnosť.

Medzi atribúty kvality správania sa systému môžeme zaradiť:

- výkonnosť,
- bezpečnosť,
- dostupnosť,
- použiteľnosť,
- funkčnosť.

Tretím typom analýzy je analýza systémových atribútov častí systému vzhľadom na systém ako celok. Ako sme spomínali vyššie, dosiahnutie niektorých atribútov kvality sa vzájomne vylučuje. Napr. v prípade znižovania pamäťovej zložitosti typicky narastajú časové nároky výpočtu a naopak. Môže však nastať situácia, že len niektoré moduly zložitého systému majú špeciálne nároky na čas výpočtu. Pritom celý systém by bol koncipovaný pre minimálne pamäťové nároky.

Analýza výkonnosti

Ďalšia možnosť analýzy architektúry je analýza výkonnosti navrhovaného systému. Výskumná výzva sa zameriava na overenie správnosti voľby dizajnu vzhľadom na požiadavky výkonnosti. Cieľom je porovnať rozdielne architektonické dizajny vzhľadom na vhodnosť ich použitia v konkrétnych implementáciách. Aj v tomto prípade narážame na problém porovnávania architektúr a absenciu vhodnej referenčnej architektúry.

3.8.7 Analytické techniky

Existuje viacero techník, ktoré umožňujú analyzovať navrhnutú architektúru:

- „na servítke“,
- UML nástroje,
- ADL.

Prvý spôsob je možné nazvať analýza „na servítke“. Tento spôsob analýzy vychádza zo skúseností softvérového architekta, ktorý je expertom v danej doméne. Architekt nakreslí architektúru pomocou obrázkov a diagramov. Problém je, že interpretovať tieto diagramy vie len daný softvérový architekt, ktorý vychádza so svojich skúseností.

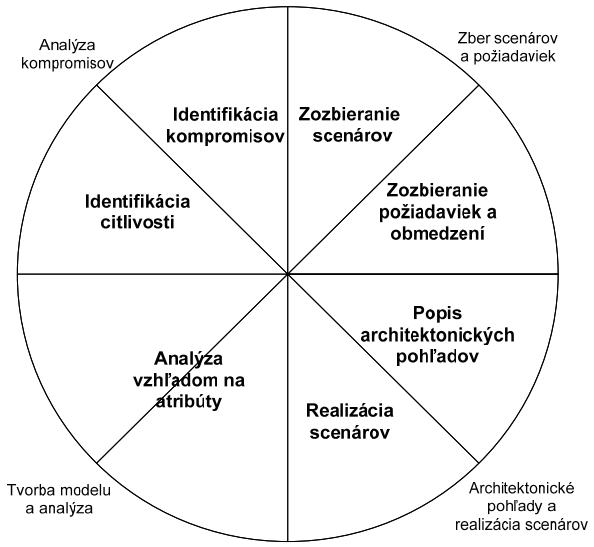
UML nástroje predstavujú ďalšiu možnosť analýzy architektúry. Použitie UML nástrojov je v tomto prípade pomerne jednoduché a rýchle. Nedostatkom je však, že pomocou UML nástroja nie je možné pokryť všetky možnosti interakcie medzi súčiastkami. Ďalším nedostatkom použitia UML nástrojov pri analýze architektúry je obmedzená možnosť vyjadrenia sémantiky jednotlivých častí architektúry.

Ďalšou možnosťou pri analýze architektúry je použitie jazyka určeného na opis architektúry (ADL). Tento spôsob opisu zatiaľ nie je celkom rozšírený v komerčnej sfére. Opisný jazyk umožňuje presne opísať jednotlivé časti architektúry tiež prepojenia medzi nimi. Nad navrhnutou štruktúrou je následne možné uskutočniť rôzne simulácie. Hlavným nedostatkom tejto metódy je jej náročnosť a zložitosť.

Metóda analýzy kompromisov

Architektúra je, zdá sa, dôležitejšia ako samotné algoritmy a dátové štruktúry, lebo presahuje vykonávanie na úrovni danej súčiastky a závisia od nej atribúty ako výkonnosť, dostupnosť či modifikovateľnosť. A ako už bolo spomínané niekoľkokrát v tejto kapitole, návrh vhodnej architektúry silne závisí od požiadaviek na konkrétny systém a nie je možné dosiahnuť súčasne všetky atribúty kvality a je nevyhnutné nájsť vhodný kompromis. Metóde hľadania kompromisov pri návrhu architektúry sa venuje (Kazman, 1998). Na identifikáciu kompromisov autori navrhujú použiť 6-krokový špirálový model, ktorý zahŕňa nasledovné kroky:

- zozbieranie scenárov (angl. *use cases*),
- zozbieranie požiadaviek a obmedzení,
- výber možných architektúr a ich atribútov,
- porovnanie atribútov architektúr,
- identifikovanie citlivosti na zmeny atribútov,
- identifikovanie kompromisov.



Obrázok 3-20. Analýza kompromisov.

3.8.8 Zhrnutie

Architektúra softvéru je pomerne nový pojem, ktorý so sebou nesie viacero výskumných výziev. Do popredia v rámci vedeckej komunity sa dostávajú snahy o návrh jazyka pre opis architektúr, ktorý by umožnil jednotlivé architektúry softvéru vyhodnotiť a vzájomne porovnať.

Použitá literatúra

[1] Albin, S. T.: *The Art of Software Architecting*. John Wiley & Sons, Hoboken, NJ., 2003.

[2] Atkinson, M., Bancilhon, F., DeWitt, D., a kol.: *The Object-Oriented Database System Manifesto,* In F. Bancilhon, C. Delobel, and P. Kanellakis (eds.): *Building an Object-Oriented Database System: The Story of O2*. Morgan Kaufmann, Los Altos, CA, 1992.

[3] Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., . Seidel, O. and Spiteri. M.: *Generic support for distributed applications*. IEEE Computer, pages 68-76, March 2000.

[4] Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice.* Addison Wesley, Reading, Massachusetts, 1998.

[5] Bosch, J.: *Design and Use of Software Architectures: Adopting and Evolving a Product- Line Approach*. Addison Wesley, Reading, MA., 2000.

[6] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-Oriented Software Architecture. A System of Patterns.* Chichester, England: John Wiley & Sons, Ltd., 1996.

[7] Dittrich, K. R., Gatziu, S., Geppert, A., "The Active Database Management System Manifesto: A Rulebase of ADBMS Features." In T. Sellis (ed.), *Proceedings 2nd International Workshop on Rules in Databases* (Athens, Greece, September 1995). Springer Verlag, Berlin, 1995.

[8] Elmasri, R. a Navathe, S. B.: *Fundamentals of Database System*, 4th Ed. Benjamin Cummings, San Francisco, CA., 2003.

[9] Garlan, D., Shaw, M.: *An Introduction to Software Architecture.* School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1994.

[10] Haus, J.: *Počítače s jedným a viacerými prúdmi inštrukcií a viacerými prúdmi údajov.* `http://spseke.sk/web/haus/eps/58.doc`, 25.3.2009.

[11] Hohpe, G., Woolf, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Boston, MA: Pearson Education, Inc., 2004.

[12] Kaisler, S. H.: Software Paradigms. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

[13] Kazman, A., et al.: The Architecture Tradeoff Analysis Method, *ICECCS98*, 1998.

[14] Robbins, K., Robbins, S.: *Practical UNIX Programming*. New Jersey, Prentice-Hall, 1996

[15] Widom, J., Ceri, S.: *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, Los Altos, CA, 1996.

[16] Woods, E.: How Do You Define Software Architecture? Dostupné z: `http://www.sei.cmu.edu/architecture/definitions.html` (June, 2009).

4
RÁMCE

Ivan Kišac, Marián Šimko

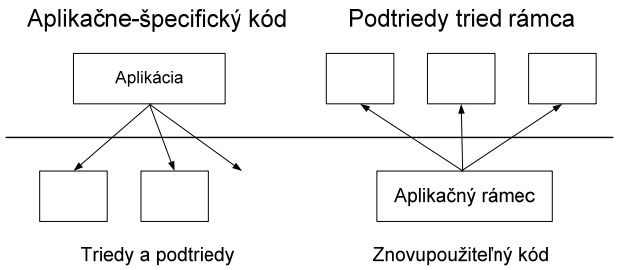
Znovupoužitie softvéru sa stalo stále používanejším spôsobom rýchleho vývoja kvalitných aplikácií. Presunulo sa zo znovupoužitia jednotlivých súčiastok na opätovné využitie celých návrhov systémov alebo štruktúry aplikácií. Softvérový systém, ktorý sa dá znovu použiť na vytvorenie úplných aplikácií, sa nazýva *rámec* (angl. *framework*).

Princíp použitia rámcov je založený na myšlienke, že bude jednoduchšie vytvoriť množinu špecifických ale podobných systémov v rámci určitej domény tak, že sa začne vývojom z určitej generickej štruktúry aplikácie. V tom sa rámce líšia od rôznych balíkov nástrojov – zdôrazňujú znovupoužitie návrhu oproti znovupoužitiu kódu (pričom aj kód môže byť znovupoužitý). Ide tu o znovupoužitie na omnoho väčšej úrovni zrnitosti (granularity).

4.1 Základné koncepty rámcov

Rámec opisuje súpravu objektových tried a ich interakcií medzi sebou, t.j. špecifikuje protokol pre výmenu informácií medzi súpravou tried. Samotný rámec môže obsahovať podrámce, ktoré reprezentujú podsystémy, moduly systému. Pri vývoji výslednej aplikácie potom treba rozvinúť túto základnú kostru, napr.: rozšíriť abstraktné triedy, doplniť vlastné triedy a podobne.

Riadenie medzi aplikáciou a infraštruktúrou sa pri použití rámcov obráti. Infraštruktúra volá časti aplikácie, čiže pri vývoji sa implementujú len konkrétne moduly, ktoré reagujú v prípade, keď ich infraštruktúra aktivuje, na rozdiel od štýlu, kedy používateľská aplikácia volá podľa potreby infraštruktúru. Uplatňuje sa tzv. hollywoodsky princíp – „Nevolajte nám, my sa Vám ozveme“, pozri obrázok 4-1.



Obrázok 4-1. Knižnica v porovnaní s rámcom.

Rozdiely medzi klasickým použitím knižníc a vývojom aplikácie pomocou rámcov možno vidieť v nasledujúcom porovnaní:
Knižnica tried:

- inštancie tried vytvára klient,
- klient volá funkcie v triedach,
- tok riadenia nie je preddefinovaný, stanovuje ho klient,
- nie je preddefinovaná interakcia medzi klientom a knižnicou tried, rozhodnutie je ponechané na klient,
- nie je stanovené preddefinované správanie.

Rámec:

- prispôsobenie tried sa uskutočňuje vytváraním podtried,
- rámec volá funkcie v kliente,
- rámec určuje tok riadenia vzhľadom na problém,
- vzor interakcie je založený na riešení problému,
- poskytuje preddefinované správanie alebo riešenie.

4.1.1 Klasifikácia rámcov

Existujú viaceré klasifikácie rámcov podľa rôznych kritérií. Uvádzame tri spôsoby klasifikácie. Dva vytvorili Fayad a Schmidt (1997), ďalej FS, a tretiu spoločnosť Taligent (1994).

Prvou klasifikáciou podľa FS je klasifikácia pre rámce orientované na jednotlivé štrukturálne súčiastky systémovej architektúry. Používajú sa na výstavbu podnikovej systémovej architektúry.

- *rámce systémovej infraštruktúry* – zjednodušenia vývoja prenosných a efektívnych systémov infraštruktúry (OS, komunikačné rámce a pod.),
- *rámce spájajúceho softvéru* – slúžia na integráciu distribuovaných aplikácií a súčiastok, rozširujú schopnosti softvéru na modularitu, ďalšie rozširovanie a pod.,
- *rámce podnikových aplikácií* – slúžia na vývoj aplikácií a produktov pre koncového používateľa. Používajú sa aj pri vývoji doménovo-špecifických aplikácií, napr. pre vedu, letectvo a pod.

Druhým spôsobom klasifikácie podľa FS je klasifikácia, kde je ako kritérium stanovený spôsob, akým sa rámec môže adaptovať alebo upravovať, aby spĺňal požiadavky na zmeny v doméne:

- *Rámec biela skrinka* (angl. *White-Box Framework*) – ide o rámec riadený architektúrou. Zakladá sa na dedení a dynamickom naviazaní, z čoho vyplýva jeho rozširovanie pomocou dedenia a preťažovania metód. Na jeho používanie treba poznať, ako triedy spolupracujú, ako sprístupniť verejný a súkromný kód, ako preťažovať existujúce triedy a sprístupňovať rodičovské metódy.

  Využíva sa dvoma spôsobmi: nájdením najvhodnejšej triedy a jej úpravou alebo úpravou vhodného príkladu.

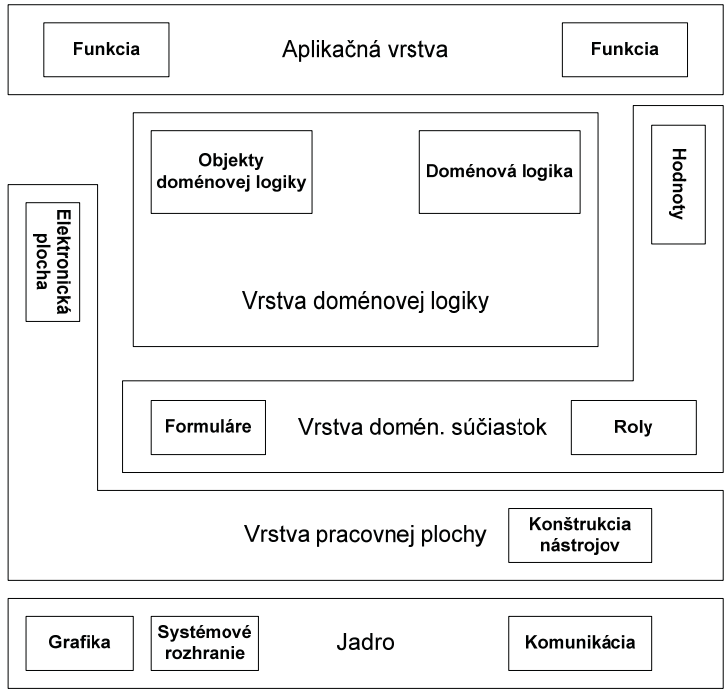
- *Rámec čierna skrinka* (angl. *Black-Box Framework*) – ide o rámec riadený údajmi. Konštruuje sa kompozíciou a delegáciou súčiastok a použitím parametrizácie. Rozširuje sa novou kompozíciou súčiastok a zavedením novej funkcionality. Ľahšie sa používa, ale je náročnejší na vývoj.
- *Rámec šedá skrinka* (angl. *Gray-Box Framework*) – je hybridom predošlých prístupov. Používa sa napríklad pre komplexnejšie aplikácie.

Druhú klasifikáciu vytvorila firma Taligent Corp., ktorá rozdeľuje rámce do 3 kategórií:

- *aplikačné rámce* – poskytujú plný rozsah funkcionality, napr.: MFC (Microsoft Foundation Classes, JFC (Java Foundation Classes),
- *doménové rámce* – slúžia na implementáciu pre špecifickú doménu (bankovníctvo, letectvo a pod.). Často sú šité na mieru pre konkrétnu organizáciu a vytvárané „z ničoho“,
- *podporné rámce* – sú zamerané veľmi špecificky, v doménach súvisiacich s počítačmi, napr. správa pamäti pre systémy súborov.

4.1.2 *Prvky rámcov*

Aj samotné rámce majú svoju architektúru. Existuje veľa rôznych architektúr na tvorbu rámcov. Príklad jednej z možných architektúr zobrazuje obrázok 4-2 a bude sa ním zaoberať aj táto podkapitola:



Obrázok 4-2. Vrstvová architektúra rámca.

- *jadro* – poskytuje základnú funkcionalitu, všeobecné služby; architektúra biela/čierna skrinka,

- *pracovná plocha* – zabezpečuje spoločné správanie sa aplikácií – základnú architektúru, vzhľad, konzistentnosť; biela skrinka,
- *doménové súčiastky* – základné koncepty domény, základ pre aplikácie domény, doménovo-špecifické triedy a typy; biela skrinka,
- *doménová logika* – vzniká dedením tried z vrstiev Doménové súčiastky a Pracovná plocha,
- *aplikácia* – samotná aplikácia poskytujúca výslednú funkcionalitu.

Každý rámec poskytuje priestor na implementáciu služieb, ktoré sú však už aplikačne špecifické a rôznia sa v závislosti od vyvíjanej aplikácie.

Existujú viaceré návrhy prvkov, ktoré by mali tvoriť vývojové rámce. Medzi základné sa radia napr. tieto (Mattson, 1996):

- *abstraktné triedy* – netvoria inštancie a podľa pravidla ASR (pravidlo abstraktnej nadtriedy, angl. *abstract superclass rule*) by mali byť abstraktné všetky nadtriedy. Tvoria tak všeobecný koncept:
  - trieda na tvorenie inštancií – konkrétna trieda,
  - trieda na vytvorenie podtriedy – abstraktná trieda,
- *návrhové vzory* – špecifikujú súčiastky rámca; načrtajú riešenie,
- *dynamické nadväzovanie* – výber najvhodnejšej súčiastky počas vykonávania,
- *kontrakty* – každá súčiastka má špecifikáciu (konkrétne aj abstraktné triedy).

4.1.3 Práca s rámcami

Pri používaní rámcov sa vyskytujú viaceré roly na oboch stranách – na strane vývoja rámca aj na strane jeho používania. Pre použitie rámca na vývoj aplikácie treba poznať ciele vyvíjanej aplikácie a mať prehľad v dostupných rámcoch a ich vlastnostiach. Výber rámca pre projekt realizuje osoba s potrebnými znalosťami, ktorá sa potom môže stať aj *používateľom rámca*. Používateľ rámca sa priamo podieľa na produkcii aplikácie. Musí vedieť ako (nemusí vedieť, prečo práve tak) používať rámec. Vykonáva úpravy rámca na horúcich bodoch (angl. *hot spots*) – dedenie, preťažovanie funkcií, pridávanie kódu.

Ďalšou osobou podieľajúcou sa na práci s rámcom je *údržbár rámca*, človek, ktorý sa stará o úpravy rámca na pokrytie požiadaviek domény. Musí disponovať požadovanými znalosťami samotného rámca aj domény. O vývoj nových alebo alternatívnych rámcov sa stará *vývojár rámca*.

Implementácia rámcov

Rámce sú implementované na základe dvoch prístupov – kontraktov a horúcich bodov:

- *kontrakty* – funkcionalita je zahrnutá v kontraktoch a určuje, ktoré časti sa majú znovu použiť. Takto určuje štruktúru. Implementácia kontraktu býva pred používateľom ukrytá.
- *horúce body* – umožňujú používateľovi zapojiť aplikačne špecifickú triedu alebo podsystém výberom z dodaných rámcov typu čierna skrinka alebo naprogramovaním v rámcoch typu biela skrinka.

Výhody a nevýhody rámcov

Používanie rámcov v mnohom uľahčuje tvorbu softvéru. Ide však o dosť rozsiahle a zložité nástroje, ktoré sa dajú efektívne využívať až potom, čo sa s nimi používateľ oboznámi. Potreba rozsiahlejšieho učenia sa práci s rámcami je hlavnou nevýhodou pri ich používaní. Naproti tomu, hlavnými výhodami sú:

- *znovupoužiteľnosť* – návrh kostry systému opakovateľne použiteľný pre viaceré problémy z tej istej oblasti,
- *spoločné vlastnosti* – rámce zachytávajú spoločné vlastnosti softvéru, ktoré je dobré spoznať a využiť pri vývoji,
- *štandardizácia* – s jej využitím vyvíjané aplikácie môžu používať existujúce overené súčiastky a tým sa znižujú náklady a zväčšuje sa robustnosť aplikácií,
- *inžinierska základňa* – vďaka štandardizácii, otvorenej architektúre a rozšírenému používaniu rámca sa vytvára dobrá základňa trénovaných používateľov,
- *tréning* – zvýšenie výkonnosti používaním ustáleného rozhrania človek-počítač a dokumentácie,
- *interoperabilita* – aplikácie, ktoré používajú ten istý rámec sú skôr kompatibilné,
- *škálovateľnosť* – použitím architektonickej infraštruktúry sa zlepšuje škálovateľnosť systému,
- *prenosnosť* – použitím otvorených konceptov a štandardov,
- *bezpečnosť* – je vstavaná a nie len pridaná,
- *čas vývoja* – je po naučení sa používania rámca podstatne kratší,
- *vyspelosť* – rámce sa zlepšujú používaním a ďalšími vylepšeniami.

Rámce však prinášajú aj svoje nevýhody. Nevýhody začínajú už pri *úsilí na vývoj znovupoužiteľného rámca*. Potom pred začatím používania je potrebné sa *naučiť* vlastnosti a možnosti rámca. Rámce *ukrývajú* svoju *architektúru* pred používateľom, čo sťažuje porozumenie a neštandardné použitie rámca. *Prechod* medzi typmi rámcov je náročný (čierna skrinka na bielu a naopak). V rámci *vnútornej architektúry* rámce môžu rozlične spracovávať udalosti, čo sťažuje vzájomné použitie rámcov.

Návrh rámcov

Ide o iteratívny proces na základe doménových expertíz. Problémom je, koľko funkcionality „nadrôtovať" napevno a koľko riešiť cez horúce body; aký bohatý má byť rámec, aby nebol príliš zložitý na učenie, a pod. Trendom je spájanie menších rámcov, pričom vývoj rámcov ide zdola nahor. Návrh rámca by mal byť pružný, úplný, rozšíriteľný a zrozumiteľný. Pri identifikovaní entít nepostačujú len metódy objektovo-orientovanej analýzy. Sústrediť sa treba najmä na služby a nie na objekty.

Ďalej treba stanoviť správanie sa rámca – ako sú entity aktivované a ako prispievajú ku správaniu systému. Správanie súčiastok má byť pre každý klient rovnaké. Ak je aplikácia komplexná, bude vyžadovať aj použitie komplexných entít, ktoré sú realizované použitím zložených objektov. Pri týchto typoch objektov ako aj pri samotnom paralelizme v aplikácii je potrebné zabezpečiť riadny chod aplikácie z hľadiska súbežnosti.

Pri návrhu rámca treba myslieť na to, že je často jednoduchšie vytvárať aplikáciu inštanciovaním existujúcich štandardných objektov, čím sa dosiahne, že časti aplikácie je možné vytvoriť bez programovania, napríklad cez grafické rozhranie rámca.

Keďže v mnohých organizáciách sa stále používa kvalitný hoci zastaraný softvér, treba pri návrhu myslieť aj na možnosti jeho integrácie ako aj na integráciu aplikácií vytvorených v rozdielnych rámcoch.

Návrh rámcov by mal byť vykonávaný až po riadnom pochopení doménovej oblasti, pričom by mal byť čo najľahšie pochopiteľný. Z hľadiska štruktúry sa odporúča navrhovať rámce tak, aby podporovali najmä zameniteľnosť, dokonca ešte pred znovupoužiteľnosťou. Každý dobrý návrh by mal obsahovať aj možnosti testovania.

Problémy s rámcami

Rámce sú založené na určitých základných štruktúrach riešení problémov a vychádzajú z mnohých predpokladov, na ktoré reagujú vlastnými spôsobmi. Tieto vlastnosti však môžu spôsobovať viaceré problémy, či už ide o vývoj rámcov so zapracovávaním zmien v doménovej oblasti alebo rozsahom pokrytia domény alebo kompozíciou, kedy môžu vznikať napríklad problémy s riadením, ak spojíme dva rámce, ktoré aktívne riadia tok aplikácie. Ďalšími problémami bývajú napríklad kompozícia so zastaranými komponentmi, ktorá sa zvykne riešiť pomocou adaptéra, či pokazenie rámca, kedy v rámci vývoja rámca prestanú byť funkčné staršie aplikácie.

Rámce predstavujú značnú pomoc pri vývoji softvéru. Predstavujú posun v znovupoužití a zefektívnení práce. Majú vyššie vstupné náklady (najmä z hľadiska naučenia) a svoje nevýhody, ale ponúkajú aj podstatné výhody, ktoré umožňujú vyvíjať kvalitné aplikácie.

4.2 Rámce GUI

Rámce sa krátko po svojom vzniku začali spájať aj s tvorbou grafického používateľského rozhrania (angl. *graphical user interface*, GUI). S postupným rozvojom aplikácií disponujúcimi iným ako konzolovým používateľským rozhraním sa začali vyvíjať aj rámce pokročilejšieho GUI. Rámec GUI môžeme definovať ako softvérový systém znovupoužiteľný na úrovni návrhu systému a jeho štruktúry pre tvorbu aplikácií založených na GUI .

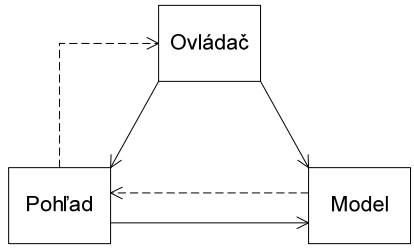
4.2.1 História

Začiatkom 70-tych rokov sa začali objavovať prvé súčiastky znovupoužiteľné pri tvorbe GUI. Nešlo zatiaľ o plnohodnotné rámce, ale o tzv. súpravy nástrojov (angl. *toolbox*). Vo svojej podstate to boli len kolekcie knižničných funkcií, ktoré implementovali nízkoúrovňovú funkcionalitu (napr. Motif Toolbox alebo Macintosh Toolbox). Rozšíriteľnosť týchto nástrojov bola obmedzená na úpravu zdrojových kódov. Neposkytovali žiadnu architektonickú podporu pre zostavenie aplikácií používajúcich zložitejšie GUI.

V roku 1980 vzniká programovacie prostredie Smalltalk-80, v ktorom sa prvýkrát objavuje koncepcia vzoru Model-Pohľad-Ovládač (angl. *Model-View-Controller*; MVC; obrázok 4-3).

Použitím tohto vzoru sú na úrovni návrhu oddelené tri aspekty vytváranej aplikácie. *Pohľad* reprezentuje prezentáciu aplikácie, jej vizuálnu podobu. *Ovládač* interpretuje pou-

žívateľské vstupy z klávesnice alebo myši a mení pohľad a model. *Model* reprezentuje doménovo-špecifické dáta, s ktorými pracuje aplikácia. Jednou z najväčších výhod vzoru MVC je možnosť pridania alebo výmeny pohľadov pri zachovaní doménového modelu. Rámcovosť prostredia Smalltalk-80 je tak chápaná aj v kontexte možnosti jednoduchej implementácie nových typov grafických klientov.



Obrázok 4-3. Vzor Model-Pohľad-Ovládač.

V druhej polovici 80-tych rokov prichádza na scénu programovacie prostredie MacApp slúžiace na vývoj aplikácií pre operačný systém Apple Macintosh. Stavia na existujúcej súprave nástrojov Macintosh Toolbox a predstavuje akúsi pracovnú kostru pre tvorbu aplikácií, ktorá môže byť jednoducho upravovaná. Aplikácie vytvorené pomocou rámca využívajú vzor Rozkaz spolu s udalostno-slučkovým modelom (angl. *event-loop model*). Vývoj rámca, ktorý sa stal základom ďalších riešení (Microsoft MFC, Borland OWL), sa zastavil v roku 2001.

S nastupujúcou dominanciou platformy Microsoft Windows bol spojený rámec Microsoft Foundation Classes (MFC), ktorý sa používa až dodnes. Jemu najväčším konkurentom sú dnes rámce rodiny Java.

4.2.2 Microsoft Foundation Classes

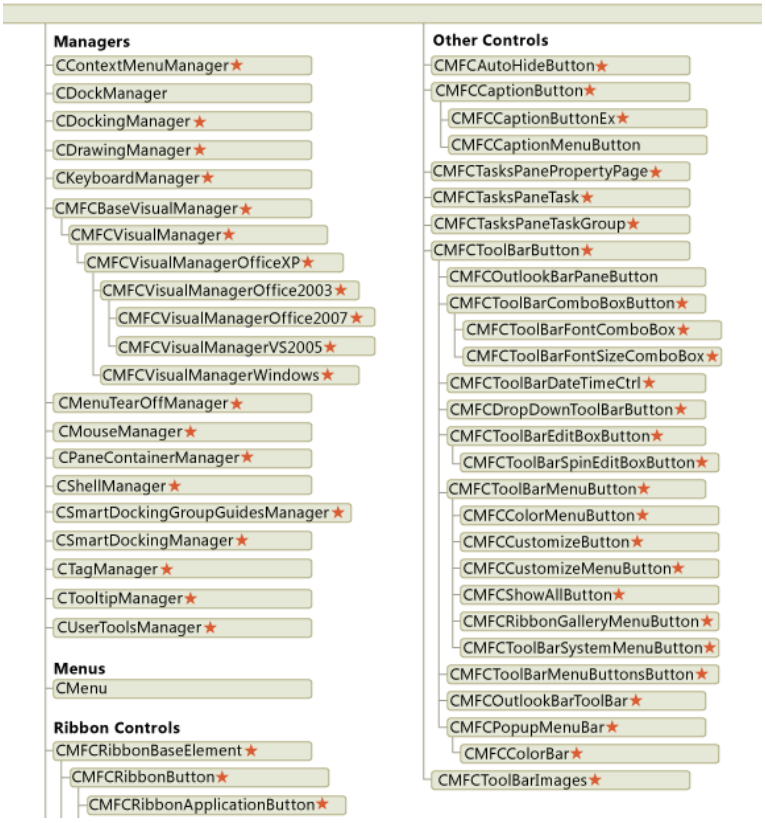
Microsoft Foundation Classes (MFC) je knižnica, ktorá obaľuje časti Windows API do tried programovacieho jazyka C++. MFC predstavuje podporu pre správu správ, spracovanie výnimiek, typovú identifikáciu počas vykonávania, serializáciu a dynamickú inštanciáciu tried. Knižnica sa o.i. dá chápať aj ako pracovný rámec pre tvorbu GUI, nakoľko obsahuje rozsiahlu podporu nástrojov pre tvorbu používateľských rozhraní (obrázok 4-4) a má podporu priamo v integrovanom vývojovom prostredí Visual Studio. V súčasnosti MFC tvorí integrálnu súčasť platformy .NET.

4.2.3 Java Foundation Classes

Java Foundation Classes (JFC) je pracovný rámec pre tvorbu prenosných GUI založených na jazyku Java. JFC tvoria tri komponenty: Abstract Window Toolkit (AWT), Swing a Java2D.

AWT je platformovo-nezávislá grafická, na oknách založená súprava nástrojov pre tvorbu používateľských rozhraní. Súprava AWT je chápaná v dvoch rovinách:

1. ako základné rozhranie medzi jazykom Java a pôvodným systémom,
2. ako súprava grafických vizuálnych prvkov (súčiastok).

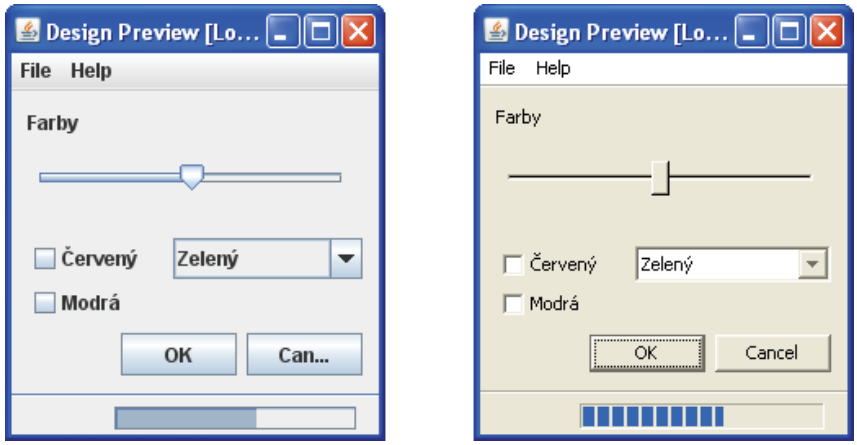


Obrázok 4-4. Výsek hierarchie tried pre podporu používateľského rozhrania v MFC demonštrujúci rozsiahlu podporu GUI nástrojov (Microsoft, 2009).

AWT ako základné rozhranie medzi jazykom Java a pôvodným systémom obsahuje podporu pre tvorbu okien, základný udalostný subsystém, manažment rozloženia prvkov a rozhrania pre vstupné zariadenia. Na druhej strane obsahuje množinu rozšíriteľných základných grafických vizuálnych prvkov, akými sú tlačidlá, textové polia, menu. Disponuje tiež rozhraním, ktoré umožňuje pôvodnému kódu vykresľovať na povrch AWT súčiastok.

Swing ja ďalšia súprava grafických vizuálnych prvkov pre jazyk Java a nástupca súpravy AWT. Obsahuje ďalšie GUI súčiastky a pridáva podporu platformovo-nezávislého vzhľadu (angl. *look and feel*). Vytvorené aplikácie tak môžu mať vzhľad zodpovedajúci pôvodnej platforme, v ktorej sú spúšťané (obrázok 4-5). Na rozdiel od svojho predchodcu, súčiastky rámca Swing sú označované ako tzv. odľahčené (angl. *lightweight*), nakoľko ich vykresľovanie nie je závislé od pôvodných systémových funkcií. Swing je komponentovo-orientovaný rámec založený na použití návrhového vzoru MVC. Súčiastky majú definované vlastné modely a pohľady. Rámec Swing je rozšíriteľný, časti súčiastok (napr. model) sú do veľkej miery definované rozhraniami, pre ktoré je zároveň vytvorená aj referenčná implementácia. Súčiastky podliehajú špecifikácii modelu JavaBeans, čím je garantovaná podpora pre vizuálne autorské zostavovacie nástroje.

Trojicu komponentov rámca JFC uzatvára knižnica *Java 2D*. Ide o aplikačné programovacie rozhranie pre vykresľovanie dvojrozmernej grafiky v jazyku Java.



Obrázok 4-5. Ukážka rozdielneho vykreslenia tej istej aplikácie. Vzhľad Motif (vľavo) a vzhľad Windows Classic (vpravo).

4.2.4 *Ďalšie rámce GUI*

Medzi ďalšie významné rámce GUI patrí Object Windows Library (OWL) fy. Borland. Vznikol začiatkom 90-tych rokov a bol konkurentom MFC. Rámec OWL umožňuje používať jedno-/viacdokumentové rozhrania, podporuje internacionalizáciu, koncept tzv. Potiahni&Polož (angl. *Drag&Drop*). Poskytuje nástroje pre tlač, ako aj jej nadhľady. Výhodou rámca je rýchlejší a menší spúštač aplikácie oproti tradičným riešeniam od firmy Microsoft. Rámec bol neskôr postupne nahradený rámcom VCL (Visual Component Library), ktorý sa dodnes používa na tvorbu aplikácií pre Microsoft Windows vo vývojových prostrediach založených na jazykoch C++ a Delphi.

Alternatívou k JFC pre platformu Java je Java SWT (Standard Widget Toolkit). Rámec je úzko spojený s vývojovou platformou Eclipse. Pre vykresľovanie používa priamy prístup k pôvodným knižniciam operačného systému prostredníctvom tzv. Java Native Interface (JNI). Aplikácie obsahujúce súčiastky rámca SWT sú prenosné, ale implementácia samotných súčiastok je platformovo-závislá. Spôsob realizácie grafickej podoby súčiastok obalením pôvodných objektov v konečnom dôsledku ovplyvňuje rýchlosť aplikácií. Aplikácie využívajúce SWT v porovnaní s JFC vykresľujú grafické súčiastky rýchlejšie, ale sú pomalšie, keď medzi nimi a inými objektmi jazyka Java dochádza k prenosu dát.

4.2.5 *Webové rámce GUI*

V súčasnosti majú nepopierateľne dôležité postavenie webové aplikácie. S ich rozvojom sa rozšírili aj tzv. webové rámce GUI. Aj keď čas života webu je len zlomkom zo života aplikácií samých osebe, aj v oblasti webového inžinierstva vznikli viaceré rámce GUI, označované tiež ako prezentačné rámce. Tieto rámce adresujú požiadavky na vytváranie aplikácií v špecifickom prostredí webu. Riešia podporu bezstavovosti aplikácie, sú previazané s webovým prehliadačom ako vykresľovačom webového obsahu, sú prispôsobené hypertextovému charakteru webu. Medzi najpoužívanejšie takéto rámce patria Google Web Toolkit (GWT), Sun Java Server Faces (JSF), Apache Wicket alebo Adobe Flex.

4.3 Vývojové rámce

Vývojové rámce sú softvérové systémy znovupoužiteľné v procese *vývoja* celých aplikácií. Sú to v podstate nasledovníky GUI rámcov zameraných „len“ na vývoj vizuálnej časti aplikácie.

V súčasnosti medzi najznámejšie vývojové rámce patrí Java Enterprise Edition a .NET. S rozvojom webových aplikácií vznikajú aj tzv. webové vývojové rámce, akým je napríklad rámec Spring.

4.3.1 Java Enterprise Edition

Java Enterprise Edition (JEE) je súprava špecifikácií, z ktorej každá opisuje podporu určitej technológie v jazyku Java. JEE je kompletný rámec pre návrh, vývoj, zostavenie a nasadenie aplikácií jazyka Java postavených na viacvrstvovom distribuovanom aplikačnom modeli. Primárne je orientovaný na vývoj a nasadzovanie podnikových, webovo-orientovaných aplikácií.

Základné architektonické prvky rámca sú:

- *Enterprise JavaBeans* – komponentový model jazyka Java, ktorý definuje rozhranie pre perzistenciu, vzdialené volania metód, kontrolu súbežnosti a kontrolu prístupu distribuovaných súčiastok nazývaných ako tzv. podnikové bôby jazyka Java (angl. *Enterprise JavaBean*; EJB). Pre viac informácií pozri kapitolu 2-4.
- *Java Transaction API (JTA)* – rozhranie pre realizáciu transakcií pre zdroje špecifikovaných štandardom X/Open XA (ďalej XA). Cieľom XA je umožniť v rámci jednej transakcie pristupovať k rôznym zdrojom, akým je napr. databáza alebo aplikačný server, a zaručiť pritom tzv. ACID vlastnosti transakcií (atomicita, konzistencia, izolácia a odolnosť; z angl. *Atomicity, Consistency, Isolation, Durability*). JTA reprezentuje rozhranie pre využitie dvojfázového odoslania transakcie, ktorou je zabezpečené, že transakcia je buď odoslaná alebo navrátená. JTA predstavuje rámcovú podporu tak pre aplikačný server, ako aj samotné súčiastky.
- *Java Messaging Service (JMS)* – rozhranie pre komunikáciu medzi dvoma a viacerými klientmi, resp. ich softvérovými súčiastkami. JMS definuje základné koncepty tzv. voľne zviazanej formy komunikácie. Elementmi sú poskytovatele správ, klienty, producenty, konzumenty, správy, rady a témy.
- *JavaServer Faces (JSF)* – webový prezentačný rámec (rámec GUI) pre vývoj a tvorbu grafických používateľských rozhraní v podnikových aplikáciách Popri rámcoch vychádzajúcich zo vzoru MVC, JSF používa prístup založený na vlastných vizuálnych súčiastkach. Stav vizuálnej súčiastky je uložený, keď si klient vyžiada novú stránku, a je obnovený, keď je požiadavka vrátená. JSF umožňuje použitie viacerých zobrazovacích technológií, napr. JSP (JavaServer Pages), XUL (XML User interface Language) i ďalších. Okrem rozhrania pre opis súčiastok, manažmentu ich stavu, správ, kontroly vstupu a pod., JSF poskytuje tiež súpravu základných súčiastok a tzv. knižnice štítkov (angl. *tag libraries*), ktoré reprezentujú deklaratívny spôsob programovania vizuálnych prvkov, resp. ich logiky.
- *Java Persistence API (JPA)* – podpora pre správu perzistentných dát v aplikáciách JEE. JPA predstavuje nasledovníka tzv. entitných bôbov jazyka Java, ktoré v skorších ver-

ziách tohto vývojového rámca reprezentovali distribuované objekty, ktoré mali perzistentný stav uložený v dátovom úložisku. Okrem rozhrania definuje JPA tiež vlastný dopytovací jazyk (Java Persistence Query Language; JPQL) a špecifikuje podobu objektovo-relačných metadát. Kľúčovým architektonickým prvkom rámca JEE je tzv. EJB kontajner, ktorý sa stará o životný cyklus bôbov (pozri obr. "Schematický náčrt kontajnera EJB v architektúre modelu EJB" v kapitole 2-4).

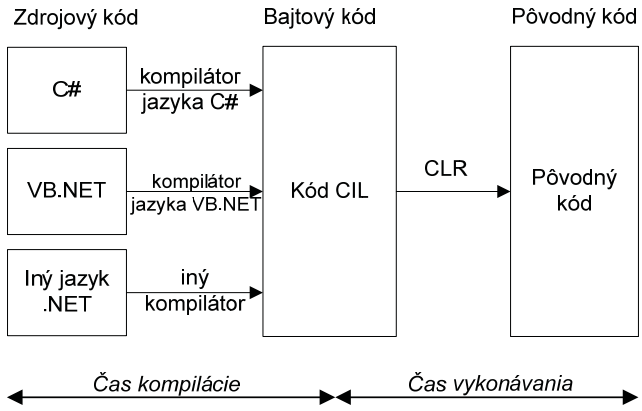
Vývojový rámec JEE je robustný systém pre podporu vývoja komplexných aplikácií pomocou jazyka Java. Prostredníctvom definovanej architektúry umožňuje oddelenie prezentačnej a doménovej logiky. Rámec podporuje distribuované nasadenie a integráciu s inými platformami nezávislými od jazyka Java. Koncept súčiastok prevádzkovaných v kontajneri, ktorý rieši ďalšie aspekty vývoja aplikácie, akými sú napr. bezpečnosť, spracovanie transakcií, spracovanie viacerých vláken, alokácia vláken, výrazne uľahčuje prácu programátora, vývojára aplikácie.

4.3.2 .NET

Druhým významným vývojovým rámcom je platforma .NET z dielne firmy Microsoft. Ide o distribuovaný architektonický koncept hostenia aplikácií v sieťovom (internetovom) prostredí. Platforma .NET predstavuje vývojové a podporné softvérové prostredie založené na operačnom systéme Windows a programovacích jazykoch fy. Microsoft.

Fundamentmi rámca sú knižnica BCL (Base Class Library) a spoločné jazykové prostredie pre vykonávanie programu (angl. *Common Language Runtime*; CLR). BCL je základná knižnica rámca a poskytuje rozsiahle množstvo funkcií zahŕňajúc používateľské rozhranie, jednoduchú grafiku, prístup k dátam, databázové spojenie, kryptografiu, numerické algoritmy, sieťovú komunikáciu a pod.

CLR definuje prostredie pre vykonávanie programového kódu. Ide o implementáciu štandardu CLI (Common Language Infrastructure), na ktorej sa vykonáva forma bajtového kódu CIL (Common Intermediate Language). Vývojári tak majú možnosť napísať programový kód v ľubovoľnom z podporovaných jazykov. Počas kompilácie kompilátor skompiluje takýto kód do bajtového kódu CIL. Počas vykonávania programu, tzv. „práve-načas-kompilátor" (angl. *just-in-time compiler*) CLR premení kód CIL do pôvodného kódu operačného systému (pozri obrázok 4-6).



Obrázok 4-6. Proces prekladu programového kódu.

Uvedená koncepcia dovoľuje abstrahovať od konkrétneho prostredia, kde bude aplikácia nasadená. CLR poskytuje služby týkajúce sa manažmentu pamäti, manažmentu vláken, spracovania výnimiek, tzv. čistenia pamäti (manažment odstraňovania nepotrebných entít z pamäti) a bezpečnosti.

Okrem BCL a CLR medzi ďalšie kľúčové vlastnosti rámca patrí interoperabilita. Rámec .NET obsahuje podporu pre realizáciu funkcionality súčiastok, ktorá je vykonávaná mimo rámec (poskytuje súpravu služieb pre prístup k súčiastkam modelu COM a obsahuje podporu pre iné modely prostredníctvom metódy `P/Invoke`). Rámec ďalej obsahuje podporu pre nasadzovanie softvéru, ktorá zahŕňa kontrolu bezpečnostných požiadaviek a integrity voči predchádzajúcim verziám. Rámec poskytuje spoločný bezpečnostný model pre aplikácie a vlastný manažment pamäti. Ten zabezpečuje zberač odpadu, ktorý je spúšťaný periodicky v samostatnom vlákne, aby identifikoval nepoužité objekty a získal späť nimi alokovanú pamäť.

Autori rámca deklarujú aj platformovú nezávislosť. Tá je však limitovaná na komerčné implementácie rámca, ktoré pokrývajú platformy Windows, Windows CE a Xbox 360.

4.3.3 Rámec Spring

Alternatívou pre vývoj webových aplikácií v jazyku Java je rámec Spring, ktorý vznikol v roku 2002 ako odpoveď na zložitosť rámca JEE (v tom čase vo verzii 1.2, označovaný tiež ako J2EE). Služby, ktoré rámec ponúka vývojárom, sú nasledovné (Johnson, 2005):

Princíp tzv. obrátenia kontroly (angl. *Inversion of Control*; IoC). Základom je tzv. kontajner IoC, ktorý reprezentuje konceptuálny prostriedok pre konfigurovanie a manažment životného cyklu objektov jazyka Java. Objekty sú vytvárané na základe konfigurácie prostredníctvom XML súborov alebo anotácií. Zostavovanie objektov prebieha prostredníctvom tzv. vstreknutia, resp. vyhľadania závislosti (angl. *dependency injection*, resp. *dependency lookup*). Ide o vzory, pri ktorých samotný kontajner posiela (vstrekuje) objekty iným objektom na základe ich požiadavky alebo konfigurácie.

Aspektovo-orientované programovanie (angl. *Aspect-Oriented Programing*; AOP). Rámec Spring obsahuje podporu pre aspektovo-orientované programovanie prostredníctvom vlastného rámca SpAOP, ktorý modularizuje pretínajúce záležitosti v aspektoch. Rámec SpAOP bol vytvorený so snahou priniesť do vývoja webových aplikácií základné vlastnosti AOP bez zbytočnej zložitosti v návrhu, implementácii alebo konfigurácii. Rámec SpAOP je založený na tzv. zachycovaniach (angl. *interceptions*), čo redukuje potrebu kompilácie aspektov alebo ich vtkania (angl. *weaving*) v čase naťahovania. V porovnaní s rámcom AspectJ, de facto štandardom AOP pre platformu Java, má rámec SpAOP menšiu výrazovú silu, ale zároveň je menej komplikovaným. Rámec Spring má priamu podporu pre konfiguráciu aspektov priamo v kontajneri.

Prístup k dátovej vrstve. Rámec Spring obsahuje rozsiahlu podporu pre existujúce rámce prístupu k dátovej vrstve pre jazyk Java. Pokrýva manažment zdrojov (automatické získavanie a uvoľňovanie databázových zdrojov), riešenie výnimiek (preklad nízkoúrovňových výnimiek do vlastnej hierarchie), sledovanie transakcií, rozbaľovanie databázových zdrojov (z vyrovnávacích pamätí databáz) a podporu pre spracovávanie veľkých binárnych alebo znakových objektov. Samostatnou kapitolou je manažment spracovania transakcií.

Manažment transakcií. Okrem práce s lokálnymi, globálnymi a vnorenými transakciami obsahuje aj podporu pre tzv. bezpečné body transakcií (angl. *transaction safepoints*). Spolu s rámcom pre prístup k dátovej vrstve je možné transakčný systém výslednej aplikácie definovať bez previazania na JTA a EJB.

Vlastný rámec MVC. Centrálnym konceptom je frontálny kontrolér `DispatcherServlet`, ktorý požiadavky posiela na konkrétne kontroléry, ktoré sú obyčajné objekty[1] implementujúce jednoduché rozhranie `Controller`. Medzi výhody patrí explicitné a jasné definovanie rolí jednotlivých súčiastok pre validáciu, spracovanie formulárov, mapovanie kontrolérov, voľby pohľadu, atď. Rámec MVC poskytuje možnosť konfigurovateľnosti pomocou rozhraní založených na návrhovom vzore Stratégia. Voľné zviazanie návrhu celého rámca umožňuje použitie viacerých prezentačných technológií.

Rámec Spring obsahuje aj podporu ďalších aspektov vývoja softvéru (vzdialený prístup, dávkové spracovávanie, autentizácia a autorizácia, testy, atď.), čo len dokazuje jeho rámcovosť ako znovupoužiteľného systému pre tvorbu aplikácií.

Použitá literatúra

[1] Hürsch, W. L.: *Should Superclasses be Abstract?* In: Proceedings of the 8th European Conference on Object-Oriented Programming (July 04-08, 1994). M. Tokoro and R. Pareschi, Eds. Lecture Notes In Computer Science, vol. 821. Springer-Verlag, London, pp. 12-31, 1994.

[2] Jendrock, E., Ball, J., Carson, D., et al.: *The Java EE 5 Tutorial.* `http://java.sun.com/javaee/5/docs/tutorial/doc/` [14.6.2009], 2008.

[3] Johnson, R.: Introduction to the Spring Framework. Dostupné z: `http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework` [14.6.2009], 2005.

[4] Kaisler, S. H.: Software Paradigms. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

[5] Microsoft: MFC Hierarchy Chart – .NET Framework v 4.0. Dostupné z: `http://msdn.microsoft.com/en-us/library/ws8s10w4(VS.100).aspx` [14.6.2009], 2009.

[1] V tomto kontexte sa používa pojem starý dobrý Java objekt (angl. Plain Old Java Object; POJO), ktorý bol zavedený na zdôraznenie jednoduchosti v kontraste s súčiastkovým modelom EJB, ktorý pre svoje súčiastky definuje príliš zložité pravidlá.

VYBRANÉ TÉMY PROGRAMOVÝCH A INFORMAČNÝCH SYSTÉMOV

5

ZNOVUPOUŽITIE NÁVRHOVÝCH VZOROV NA ÚROVNI MODELU

Ľubomír Majtás

Zavedenie vzorov znamenalo prínos pre viaceré disciplíny. Myšlienky publikované v práci Christophera Alexandra [1] o urbanistických riešeniach našli rýchlo uplatnenie aj v rámci informatických vied. Keďže úspech procesu tvorby softvéru je často závislý od správnych rozhodnutí vývojárov, idea možnosti aplikovania overených vzorových riešení na opakujúce sa problémy našla rýchlo svoje uplatnenie. Po zavedení pojmu vzor v odbore softvérového inžinierstva začala jeho popularita prudko stúpať. Vzory sa uplatnili v rôznych fázach životného cyklu vývoja softvéru: boli definované analytické vzory [9], návrhové vzory [12], vzory pre integráciu [13], testovanie [2] a mnoho iných (nasadenie vzorov v rôznych odvetviach informatiky sa analyzuje napr. v práci [16]). S ich použitím došlo k skvalitneniu procesu vývoja v rôznych ohľadoch: zjednodušila a zefektívnila sa komunikácia medzi vývojármi, vzrástla informovanosť nielen o rôznych riešeniach, ale aj nástrahách, ktorým je vhodné sa vyhnúť.

Asi najznámejšie uplatnenie pojmu vzor v softvérovom inžinierstve priniesla práca GoF [12], v rámci ktorej autori identifikovali a podľa definovanej šablóny podrobne opísali 23 návrhových vzorov. Súčasťou opisu každého vzoru je slovný opis jeho hlavnej myšlienky, príklad jeho vhodného použitia (vrátane ukážky zdrojového kódu), opis riešenia, ktoré vzor poskytuje a diskusia o jeho dôsledkoch či alternatívach. Hlavnou časťou opisu riešenia je zobrazenie jeho modelu pomocou štandardných OMT/UML diagramov. Tie vhodne slúžia na zachytenie príkladu použitia vzoru, no nedokážu plne zachytiť jeho celú štruktúru ani myšlienku.

Práca GoF vytvorila veľký ohlas medzi odbornou verejnosťou, myšlienky v nej prezentované si osvojili mnohí autori, ktorí na nich ďalej stavali a postupne ich obohacovali. Nasledujúca kapitola sa taktiež zaoberá touto problematikou, pričom sa z veľkej časti sústreďuje na možnosti znovupoužitia návrhových vzorov na úrovni modelov. Opiera sa pri tom o poznatky opísané vo viacerých publikáciách, v rámci ktorých sa ich autori okrem iného snažili zodpovedať na mnohé otázky, ktoré zostali v práci GoF otvorené. Zaoberá sa problematikou životného cyklu inštancií návrhových vzorov, formami opisu vnútornej štruktúry návrhových vzorov a ich kompozícii. V závere prináša vlastný pohľad na možnosti modelovania s využitím návrhových vzorov.

5.1 Životný cyklus inštancií návrhových vzorov

Podobne ako môžeme v procese vývoja softvéru identifikovať fázy jeho životného cyklu, môžeme tak urobiť aj v prípade návrhových vzorov a ich inštancií. V rámci kapitoly budeme rozlišovať medzi pojmami návrhový vzor a inštancia návrhového vzoru.

- Za návrhový vzor považujeme celkovú všeobecnú myšlienku riešenia definovanú príslušným katalógom. Definícia návrhového vzoru v sebe spája opis problému, jeho odporúčaného riešenia a konsekvencie, ktoré toto riešenie so sebou prináša. Súčasťou opisu bývajú aj príklady jeho konkrétnej inštancie na úrovni návrhu a zdrojového kódu.
- Za inštanciu návrhového vzoru považujeme konkrétne riešenie vychádzajúce z možnosti znovupoužitia všeobecného riešenia definovaného v rámci opisu daného vzoru. Inštancia návrhového vzoru je priamou súčasťou návrhu, resp. implementácie vytváraného konkrétneho softvérového systému. Je vytvorená zo štandardných stavebných prvkov objektovo-orientovaného vývoja softvéru akými sú triedy, objekty, atribúty či metódy. Inštancia vzoru nebýva fyzicky spätá so vzorom, podľa ktorého bola vytvorená. Identifikácia príslušnosti inštancie k samotnému vzoru býva často ponechaná na schopnosti vývojárov, ktorí s ňou prídu do kontaktu.

Životný cyklus inštancií návrhových vzorov pozostáva z dvoch etáp. Prvá z nich zachytáva proces vytvorenia korektnej inštancie návrhového vzoru, nazýva sa tiež inštanciácia vzoru. Druhá etapa, nazývaná tiež evolúcia inštancie, opisuje proces zmien po vytvorení inštancie vzoru.

5.1.1 Vytvorenie inštancie návrhového vzoru

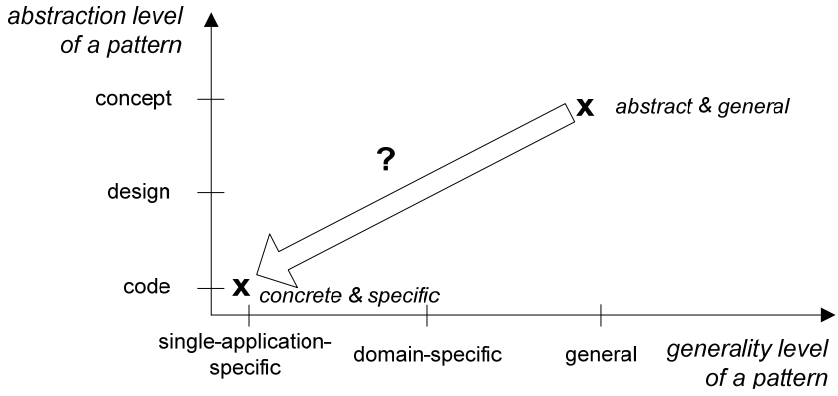
Procesom vytvárania inštancie návrhového vzoru prechádzame pri aplikácii riešenia ponúkaného vzorom do vytváraného softvérového systému. Medzi vstupy vytvárania inštancie patrí aktuálny stav vytváraného systému a všeobecný opis návrhového vzoru. Za výstup považujeme pozmenený stav vytváraného systému, ktorý je rozšírený o korektne vytvorenú inštanciu návrhového vzoru.

Rozlišujeme dve činnosti [31], ktoré je potrebné vykonať pri tvorbe inštancie vzoru: abstraktnú a všeobecnú inštanciu vzoru je potrebné konkretizovať a vyšpecifikovať. Tieto činnosti sa môžu na prvý pohľad javiť ako totožné, no nie je tomu tak. Každá približuje prvotnú myšlienku vzoru k finálnej inštancii, pričom je potrebné vykonať obe, aby bolo možné inštanciu prehlásiť za korektne vytvorenú. Odlišnosti medzi týmito činnosťami sú prezentované na obrázku 5-1, v rámci ktorého sú miery všeobecnosti a abstrakcie prezentované v dvojrozmernom priestore (miera všeobecnosti horizontálne, miera abstrakcie vertikálne.).

Vytváraná inštancia sa stáva konkrétnejšou, keď obsahuje viac stavebných prvkov tvoriacich korektnú inštanciu. Z počiatočnej abstraktnej myšlienky použitia vzoru sa pridávaním tried, ich atribútov a metód postupne stáva konkrétna inštancia.

Špecifikovanie inštancie znamená priblíženie všeobecného opisu vzoru do kontextu vyvíjaného systému. Za všeobecné možno považovať príklady inštancií vzorov prezentované v katalógu GoF [12]. Tie obsahujú hráčov všetkých rolí (takže ich možno považovať za konkrétne), ale títo hráči nezodpovedajú žiadnej doméne. Špecifikovanie predstavuje

úpravy inštancie vzoru, na základe ktorých sa stáva doménovo špecifická, až špecifická pre potreby konkrétnej aplikácie. Medzi príklady krokov procesu špecifikácie možno považovať definovanie voliteľných počtov hráčov jednotlivých rolí, či adekvátne pomenovanie hráčov rolí v kontexte vyvíjaného systému.



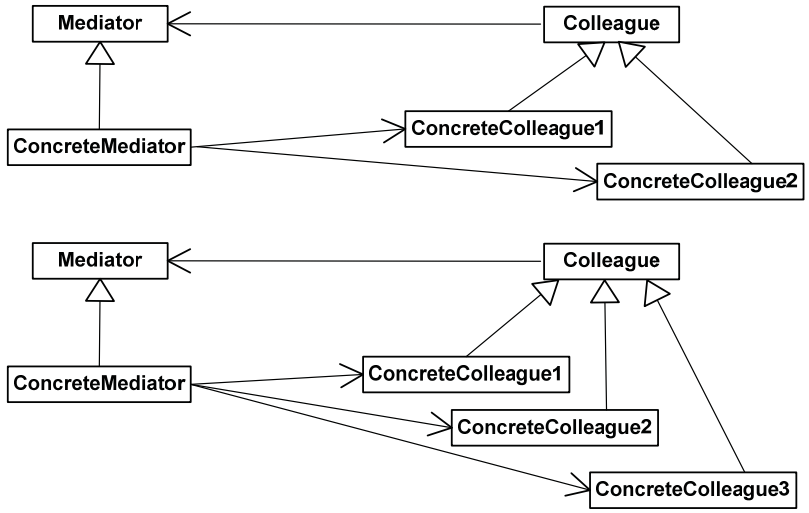
Obrázok 5-1. Dvojrozmerný priestor zachytávajúci mieru všeobecnosti a abstrakcie [32].

5.1.2 *Evolúcie inštancií vzorov*

Druhou etapou životného cyklu inštancií vzorov je ich evolúcia, ktorá znamená zmeny v rámci už raz vytvorených inštancií. Zmeny sú neoddeliteľnou súčasťou vývoja softvéru a je problémom, keď si jedna malá zmena vyžiada sériu zmien v celom systéme. Preto je potrebné navrhovať a tvoriť systémy tak, aby ich zmeny v budúcnosti vyžadovali čo možno najmenej ľudského úsilia. Jedným z prínosov návrhových vzorov je možnosť elegantnejších zmien v systéme. V rámci štruktúry inštancií návrhových vzorov môžeme rozlíšiť fixné časti a meniteľné časti, ktoré sú okamžite pripravené na zmeny. Isté zásahy však môžu viesť k poškodeniu štruktúry inštancií, čo môže mať negatívny vplyv na ďalší rozvoj systému. V prípade potreby zmien je preto nutná dobrá znalosť problematiky návrhových vzorov: je potrebné rozpoznať, ktoré časti sú jednoducho modifikovateľné a s ktorými sa naopak manipulovať neodporúča.

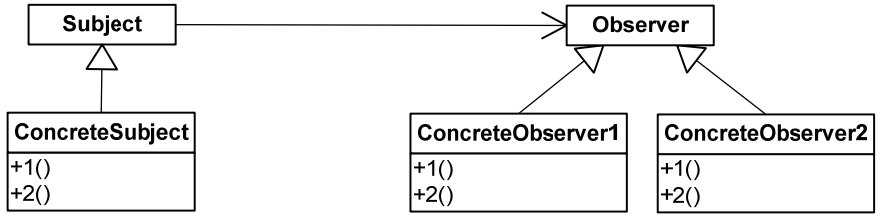
Práca [5] sa zaoberá problematikou evolučných zmien inštancií vzorov. Definuje dve hlavné úrovne zmien implementácii vzorov:

1. Primitívne (*Primitive level*) – základné operácie nad triedami. Množina primitívnych operácií obsahuje: pridanie / odobratie triedy, atribútu, operácie, asociácie, generalizácie, agregácie, kompozície, závislosti.
2. Vzorové (*Pattern level*) – charakterizujú zmeny, ktoré je možné uskutočniť nad vzormi, pozostávajúce zo sekvencie primitívnych operácií. Autori definovali 5 rôznych vzorových operácií (ich počet sa môže v budúcnosti rozšíriť). Nad jednotlivými vzormi sa dajú vykonať len niektoré operácie (nie všetky).
   a. Nezávislá (*Independent*) zmena – jednoduché pridanie alebo odstránenie triedy a k nej prislúchajúcich väzieb. Napríklad rozšírenie vzoru Mediator o nový ConcreteCollegue (obrázok 5-2).

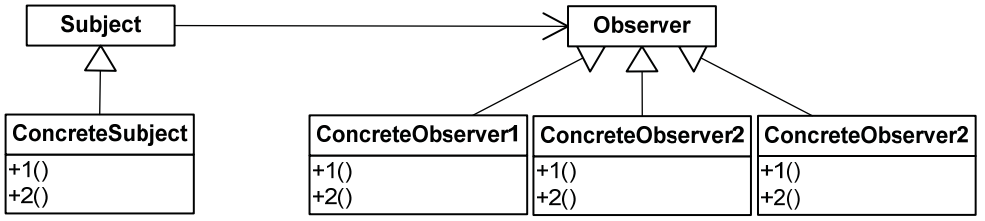


Obrázok 5-2. Nezávislá zmena na vzore Mediator [5].

b. Balíčková (*Packaged*) zmena – dochádza k pridaniu alebo odobratiu nezávislej triedy a k nej prislúchajúcich väzieb a súčasne k pridaniu (odobratiu) atribútov alebo operácii danej triedy. Napríklad pridanie ConcreteObserver do vzoru Observer (obrázky č. 5-3 a 5-4).

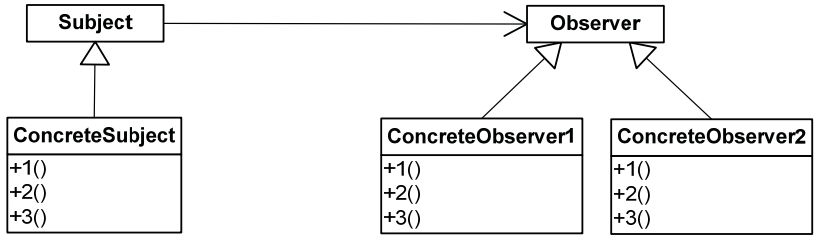


Obrázok 5-3. Pôvodná štruktúra vzoru [5].



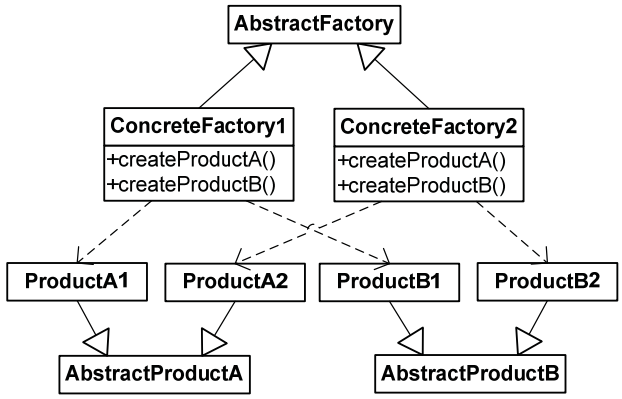
Obrázok 5-4. Balíčková zmena na vzore Observer [5].

c. Zmena skupiny tried (*Class Group*) – dochádza k pridaniu (odobratiu) atribútov/operácií vo viacerých triedach súčasne. Napr. pridanie atribútu k Observeru (obrázky č. 5-3 a 5-5):

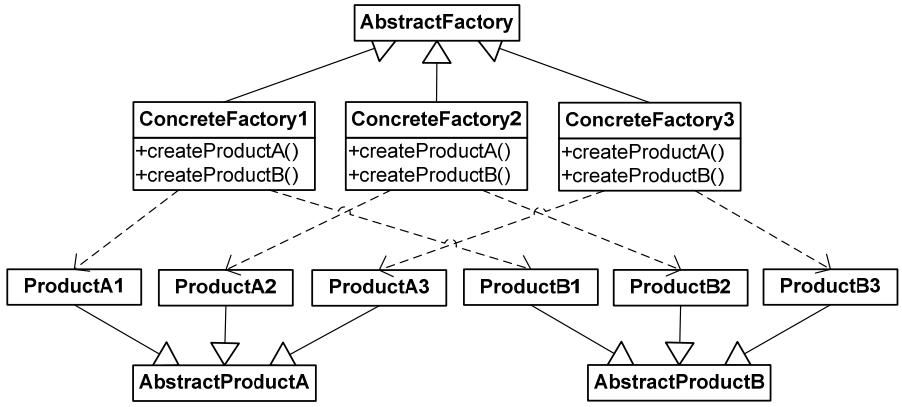


Obrázok 5-5. Zmena skupiny tried na vzore Observer [5].

d. Zmena korelovaných tried (*Correlated Classes*) – pridanie (odobratie) jednej triedy, ktoré vedie k pridaniu (odobratiu) ďalších tried. Napríklad pridanie ConcreteFactory do Abstract factory znamená pridanie ďalších produktov (tých, ktoré generuje nová fabrika; pozri obrázky 5-6 a 5-7).



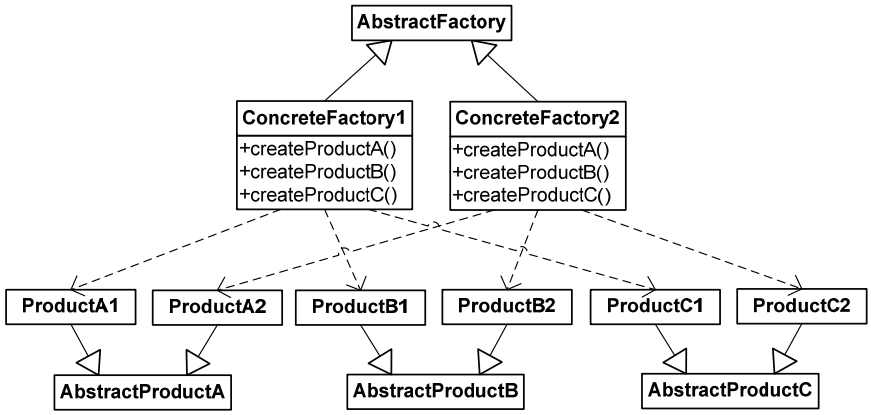
Obrázok 5-6. Pôvodná štruktúra vzoru Abstract factory [5].



Obrázok 5-7. Zmena korelovaných tried na vzore Abstract factory [5].

e. Zmena korelovaných atribútov/operácií (*Correlated Attributes/Operations*) – pridanie (odobratie) triedy, ktoré si vyžiada pridanie (odobratie) atribútov/operácií aj v pôvodných triedach vzoru. Napríklad pridanie nového typu produktov v Ab-

stract factory vedie k pridaniu metód vytvárajúcich produkty daného typu (obrázky 5-6 a 5-8).



Obrázok 5-8. Zmena korelovaných operácií na vzore Abstract factory [5].

Nie každý typ zmeny je vykonateľný na každom vzore. Tabuľka 5-1 obsahuje zoznam vzorov a k nim prislúchajúce zmeny, ktoré na nich možno vykonať. Len vykonávaním korektných zmien možno zabezpečiť dlhotrvajúcu prítomnosť korektnej inštancie návrhového vzoru v systéme aj po jeho značných modifikáciách. Preto je potrebné poznať, ako sú jednotlivé vzory modifikovateľné a tomu prispôsobiť aj zásahy vykonávané v rámci zmien softvéru.

Tabuľka 5-1. Zmeny prislúchajúce k jednotlivým vzorom [5].

Návrhový vzor	Typ operácie	Návrhový vzor	Typ operácie
Abstract Factory	4,5	Chain of Responsibility	2
Builder	4,5	Command	4
Factory Method	4	Interpreter	2
Prototype	2	Iterator	4
Singleton	-	Mediator	1
Adapter	4,5	Memento	3
Bridge	2	Observer	2,3
Composite	2	State	2
Decorator	2,3	Strategy	2
Façade	1	Template Method	2,3
Flyweight	2	Visitor	2,5
Proxy	4		

5.2 Opisy návrhových vzorov

Súčasný výskum sa snaží nájsť prostriedky, ktoré by umožnili automaticky podporovať rôzne aktivity vzorov v troch hlavných aspektoch: aplikovaní inštancií návrhových vzo-

rov, verifikácii ich implementácii a ich vyhľadávaní v existujúcich systémov za účelom ich porozumenia a zdokumentovania. Aby bolo možné zautomatizovať tieto aktivity, je potrebné podrobne zachytiť opakujúcu sa štruktúru a správanie vzorov, ktoré sú často označované ako leitmotif vzoru [8]. Leitmotify vzoru možno chápať ako abstraktné návrhové modely v návrhárovej mysli. Zachytávajú najdôležitejšie invarianty, ktoré generujú konkrétne riešenia špecifického návrhového problému. Štruktúra vzoru nepredstavuje riešenie, ale generuje riešenia [3]. Práve táto flexibilita oddeľuje vzory od návrhových modelov, šablón tried či OO frameworkov. Nanešťastie modelovací jazyk, ktorý by dokázal presne špecifikovať nemenné časti leitmotifov vzorov, zatiaľ neexistuje.

V nasledujúcich kapitolách budú podrobnejšie opísané vybrané prístupy opisu leitmotifov návrhových vzorov.

5.2.1 Metamodelovací jazyk založený na roliach

Práca [10] predstavuje jazyk určený na špecifikáciu vzorov: Metamodelovací jazyk založený na roliach (*Role Based Metamodeling Language* – RBML). RBML umožňuje autorom špecifikovať návrhové vzory z viacerých perspektív: statickej štruktúry, interakcie, stavového správania sa. RBML používa na špecifikáciu vlastností vzorov vizuálnu notáciu založenú na UML [33] a textové obmedzenia zaznamenané v jazyku OCL [26]. Ide o špecifikačný jazyk opisujúci skupiny UML modelov. Samotná špecifikácia vzorov je založená na báze rolí, ktoré sú asociované s UML metatriedami. Role špecifikujú vlastnosti, ktoré musia spĺňať elementy modelu hrajúce danú rolu. To znamená, že model, ktorý zodpovedá špecifikácií vzoru, pozostáva z elementov hrajúcich role tejto špecifikácie. Môže obsahovať aj iné aplikačne špecifické elementy, ak sa tým nedostáva do konfliktu s metamodelovou špecifikáciou vzoru.

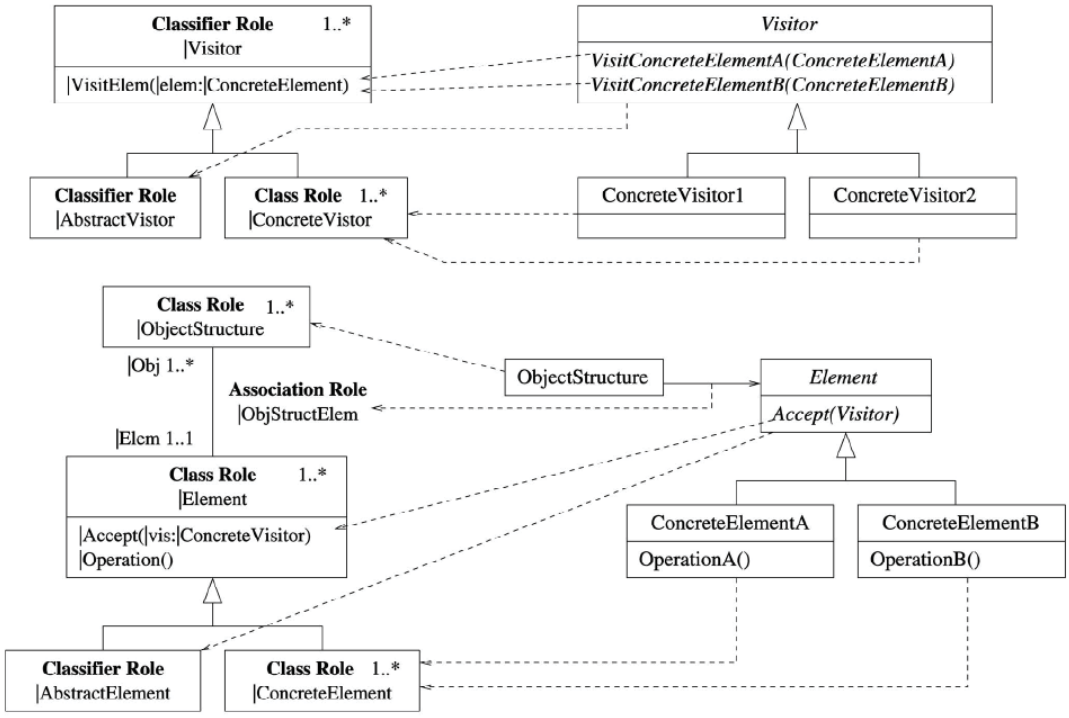
Základným modelom RBML je Statická špecifikácia vzoru (*Static Pattern Specification* – SPS) predstavujúca obmedzenia UML metamodelu, ktoré spresňujú priestor riešenia vzoru zo štrukturálneho aspektu. SPS pozostáva z klasifikátorov (*classifier*) a vzťahov (*relationship*), ktoré vychádzajú z metatried Classifier a Relationship UML metamodelu. Rola definovaná v rámci SPS môže byť asociovaná s obmedzeniami definovanými pomocou OCL. Príklad SPS špecifikácie vzoru Visitor sa nachádza na obrázku 5-9.

Na zachytenie dynamického správania sa účastníkov vzoru slúži Interakčná špecifikácia vzoru (*Interaction Pattern Specifications* – IPS). IPS pozostáva z interakčnej role, ktorá je špecializáciou triedy Interaction UML metamodelu. Interakčná rola je štruktúrou lifeline a správy (*message*), ktoré sú založené na metatriedach Lifeline a Message UML metamodelu. Z vizuálneho hľadiska sa IPS je obdobou sekvenčného diagramu UML.

Na opis stavového správania sa účastníkov vzoru RBML definuje Stavovú špecifikáciu vzoru (*StateMachine Pattern Specifications* – SMPS), ktorá vychádza zo stavového diagramu UML.

5.2.2 Precízne modelovanie vzorov na základe UML metamodelu

Alternatívu k predchádzajúcemu prístupu predstavuje práca [21]. Jej hlavným cieľom je opäť podrobne zachytiť leitmotif vzoru do modelu pozostávajúceho z navzájom spolupracujúcich rolí pomocou UML metamodelu.



Obrázok 5-9. SPS model vzoru Visitor (vľavo) a k nemu prislúchajúca inštancia (vpravo) [11].

Tento prístup však kladie väčší dôraz na prácu s rolami, pričom podrobnejšie definuje vlastnosti ich použitia a súčasne využíva mieru abstrakcie, ktorú tento pojem umožňuje. Pri práci s rolami uplatňuje nasledovné pravidlá:

1. Rola predstavuje buď štrukturálnu entitu (zodpovedá jej napr. trieda, objekt) alebo entitu správania (zodpovedá jej napr. metóda). Leitmotif vzoru je opísaný ako množina spolupracujúcich rolí oboch typov.
2. Každej roli môže zodpovedať niekoľko konkrétnych hráčov reprezentujúcich inštanciu role. Niektoré sú definované tak, aby im zodpovedal iba jeden hráč, iné sú určené tak, aby im zodpovedalo viac (0 - N) hráčov. Hovoríme o početnosti role.
3. Role môžu mať viac dimenzií: napríklad vo vzore Abstarct factory každý hráč role ConcreteProduct zodpovedá určitému typu produktu (AbstractProduct) a súčasne rodine produktu (= produkty vytvárané rovnakým hráčom ConcreteFactory). V tomto prípade je rola ConcreteProduct definovaná ako dvojrozmerná, pretože počet hráčov role je viazaný na počty hráčov dvoch nezávislých rolí.
4. Vzťahy medzi rolami sa nie sú mapované jedna k jednej k výsledným hráčom:
   a. Jedna relácia na úrovni modelu rolí môže byť mapovaná do viacerých vzťahov na úrovni návrhu. Napríklad vo vzore Abstract factory je každý hráč role ConcreteFactory zodpovedný za inštanciáciu hráčov role ConcreteProduct. To, akým spôsobom sa to deje, nie je definované: môže to byť lokálne na báze Factory method (klasicky) alebo delegovaním pomocou vzoru Prototype (vzniká vzor Plugable Factory [35]).

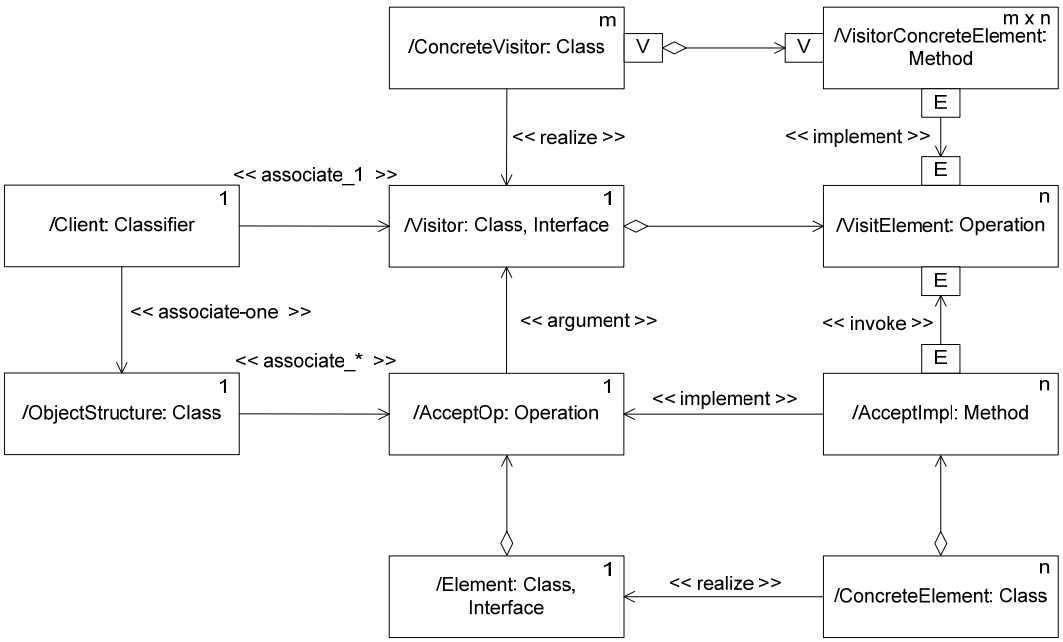
b. Vzťahy medzi vzormi môžu byť mapované na iné typy vzťahov na úrovni návrhu. Napríklad vo vzore Observer je vzťah medzi rolami Observer a ConcreteObserver definovaný ako realizácia rozhrania. Ak však spojíme role Observer a ConcreteObserver do jedného hráča, vzťah medzi rolami nebude zodpovedať žiadnemu vzťahu na úrovni návrhu.

Modelovanie štruktúry leitmotifu je realizované na báze meta-úrovňovej spolupráce, presnejšie ako spolupracujúce elementy UML metamodelu. Mapovanie entít z domény návrhových vzorov do UML metamodelu sa nachádza v tabuľke 5-2.

Tabuľka 5-2. Mapovanie entít z domény návrhových vzorov do UML metamodelu [21].

Doména vzorov	UML doména
Špecifikácia vzoru	Collaboration
Inštancia vzoru	CollaborationInstanceSet
Rola	ClassifierRole
Hráč role	Instance
Vzťah medzi rolami	AssociationRole

Na obrázku 5-10 je zachytený leitmotif vzoru Visitor na báze UML metamodelu. Pre každú rolu je v pravom hornom rohu zachytený počet hráčov, ktorí k danej roli prislúchajú. V prípade role VisitConcreteElement je počet definovaný ako m*n, čiže počet hráčov role ConcreteVisitor krát počet hráčov role VisitElement. Ide o príklad dvojrozmernej role, keďže počet hráčov zodpovedajúcich tejto roli je závislý od počtu hráčov dvoch nezávislých rolí.

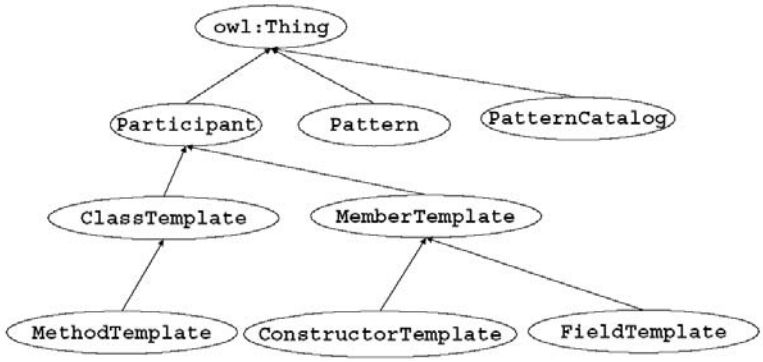


Obrázok 5-10. Leitmotif vzoru Visitor na báze UML metamodelu [21].

5.2.3 Modelovanie založené na OWL

Iný spôsob modelovania štruktúry vzorov je predstavený v práci [4]. Jej prístup sa na rozdiel od predchádzajúcich nezakladá na modelovaní pomocou UML, ale používa nástroje vyvinuté v rámci iniciatívy Webu so sémantikou (*Semantic Web*): RDF (*Resource Description Framework*) [29] a z neho vychádzajúci jazyk OWL (*Web Ontology Language*) [27]. Dôvodom použitia týchto riešení je voľnosť, ktorú poskytujú. Neboli vyvinuté na ukladanie znalostí o vzoroch či softvérových riešeniach, ale na zachytávanie informácií o akýchkoľvek zdrojoch, ktoré sú jasne identifikovateľné pomocou svojich URI (*Uniform Resource Identifier*). Z toho vyplývajú široké možnosti, ktoré jazyky RDF a OWL poskytujú. Napríklad triedam možno definovať vlastnosti presne v takej forme, ako je potrebné (napr. isAbstract, isStatic, isSubclass-Of).

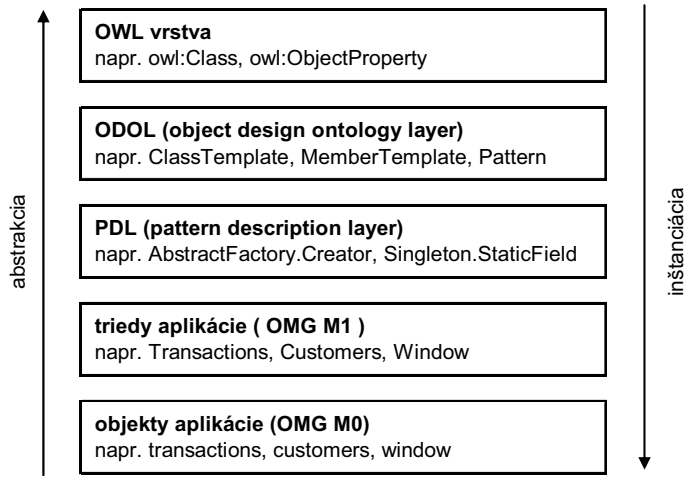
Na modelovanie vzorov a ich štruktúry bola použitá hierarchia OWL tried nachádzajúca sa na obrázku 5-11. V hierarchii sa nenachádzajú konkrétne prvky modelov ako triedy či metódy, ale len ich šablóny (ClassTemplate, MemberTemplate). Dôvodom toho je fakt, že táto hierarchia neobsahuje konkrétne stavebné jednotky návrhových vzorov, ale role, ktoré bývajú najčastejšie realizované práve týmito stavebnými prvkami. Takže napríklad ak pre vzor Abstract factory je ConcreteFactory inštanciou triedy ClassTemplate, znamená to, že ide o rolu, ktorá je štandardne realizovaná triedou.



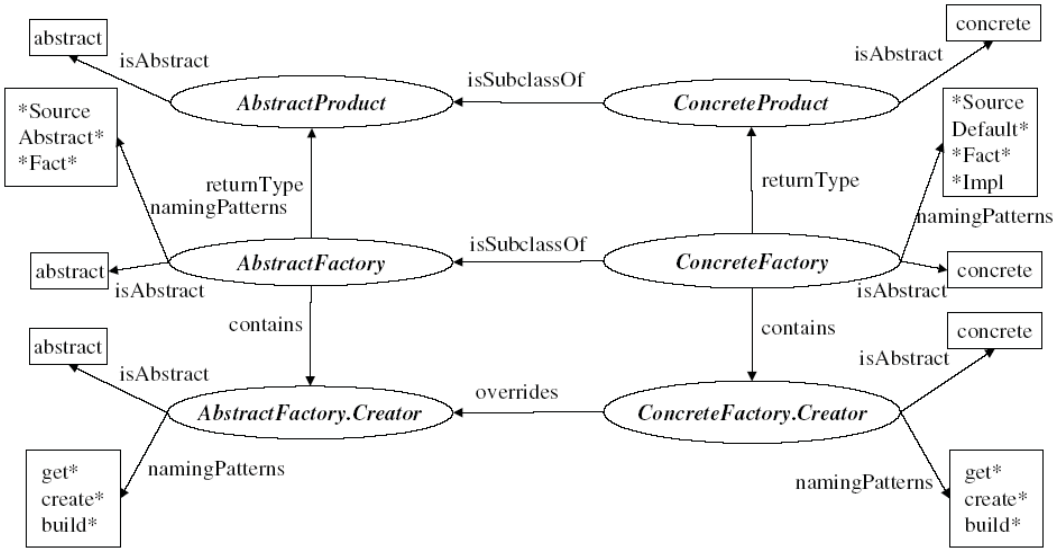
Obrázok 5-11. Základná hierarchia OWL tried ontológie (vrstva ODOL) [4].

Autori definovali vlastnú metamodelovú architektúru, ktorá sa odlišuje od štandardnej štvorvrstvovej OMG architektúry (M0 – inštancie, M1 – model tried, M2 – metamodel, M3 – metametamodel MOF). Schéma tejto architektúry sa nachádza na obrázku 5-12. Dolné dve vrstvy zodpovedajú štandardným OMG modelom M0 a M1. Najvyššia vrstva predstavujúca jazyk OWL zodpovedá modelu M3. Vrstva ODOL (*Object Design Ontology Layer*) je metamodelová vrstva zodpovedajúca vrstve M2 (obrázok 5-11). Obsahuje definície základných pojmov potrebných na opis vzorov, napr. triedy *Template, Participant, Pattern a iné. Vrstva PDL (*Pattern Description Layer*) obsahuje tvrdenia o vzoroch vytvorené pomocou konceptov z vrstvy ODOL, napríklad definície rolí a vzťahov medzi nimi.

Príklad opisu návrhového vzoru Abstract factory sa nachádza na obrázku 5-13. Zdroje modelu (zobrazené ako elipsy) predstavujú role vzoru, ktoré sú v rámci OWL opisu definované ako inštancie tried definovaných vo vrstve ODOL.



Obrázok 5-12. Použitá metamodelová architektúra [4].



Obrázok 5-13. OWL opis návrhového vzoru Abstract factory [4].

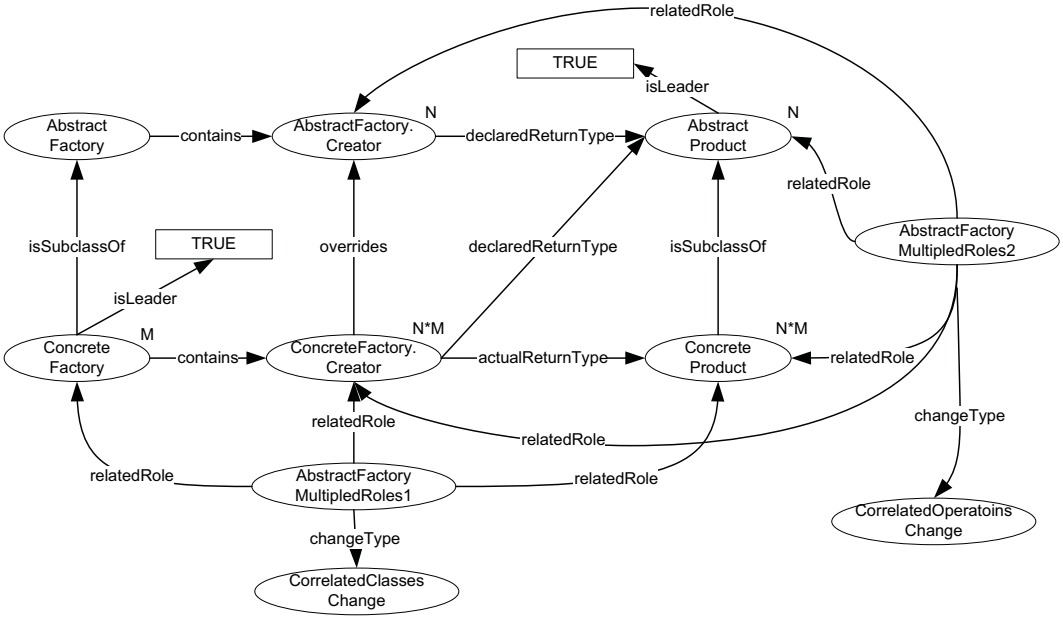
Rozšírenie o znalosti týkajúce sa rolí

Predchádzajúci prístup predstavuje zaujímavé riešenie z hľadiska použitých technológií. Uloženie znalostí o vzoroch pomocou jazyka definovaného v rámci iniciatívy Webu so sémantikou dáva predzvesť ich ďalšieho možného využitia. V súvislosti s Webom so sémantikou dochádza k rozvoju odvodzovacích nástrojov, ktorých úlohou je umožniť spracovanie uložených znalostí. Tým sa otvára možnosť využitia znalostného prístupu pri základných operáciách so vzormi ako sú vytváranie, modifikácia alebo verifikácia inštancií.

Nedostatkom opísaného prístupu založenom na OWL v porovnaní s predošlými opisovanými prístupmi je menšie množstvo informácií, ktoré o vzoroch poskytuje. Tie sa týkajú najmä informácií o roliach ako sú napríklad ich početnosť či dimenzia. Pre spomínaný

potenciál, ktorý so sebou prináša použitie technológií Webu so sémantikou, bolo vytvorené rozšírenie ontologického modelu o tieto chýbajúce znalosti, ktoré je podrobne opísané v práci [20]. Okrem doplnenia chýbajúcich informácií o roliach boli pridané aj informácie o možných evolúciách vzorov podrobnejšie opísaných v kapitole 5.1.2 Evolúcie inštancií vzorov. Príklad rozšíreného opisu návrhového vzoru Abstarct factory sa nachádza na obrázku 5-14. V prípade Abstract factory možno identifikovať dve hlavné role, od ktorých početnosti závisí početnosť ostatných: AbstractProduct a ConcreteFactory. Počet hráčov rolí ConcreteProduct a ConcreteFactory.Creator je od nich priamo závislý.

Rozšírením pôvodného opisu vzorov došlo k odstráneniu jeho hlavného nedostatku spočívajúceho v nemožnosti uchovávania dostatočného množstva informácií. Po odstránení tohto nedostatku možno považovať obsahovú hodnotu tohto prístupu za porovnateľnú s prístupmi založenými na UML.



Obrázok 5-14. Diagram rozšíreného OWL modelu vzoru Abstract factory.

5.2.4 Grafická notácia

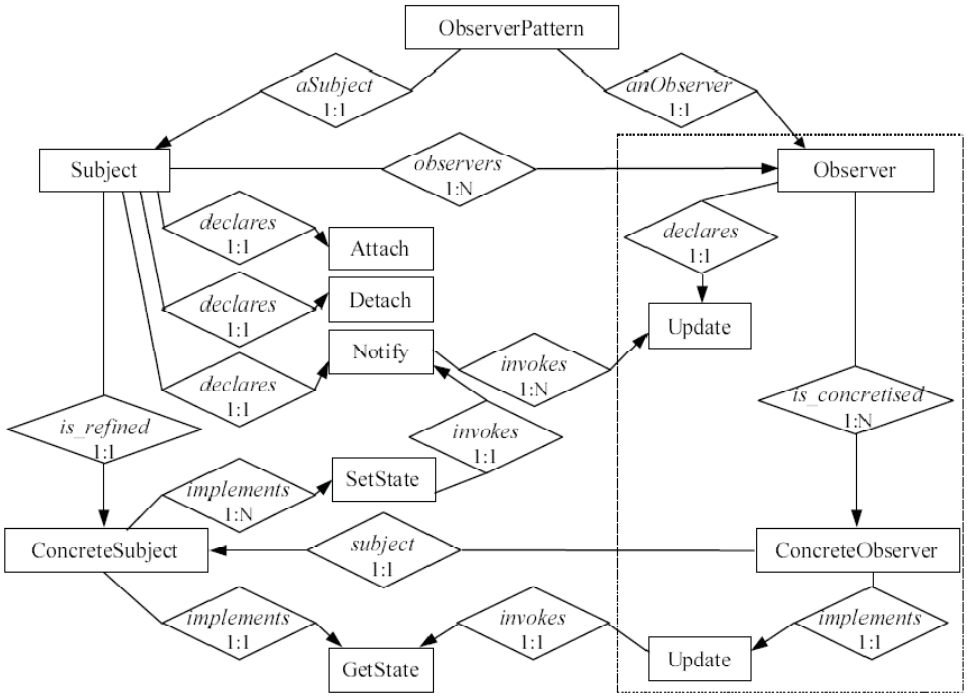
Prístup [31] bol vytvorený na pôde Katedry informatiky a výpočtovej techniky FEI STU, ktorá bola predchodcom dnešnej FIIT STU. Predstavuje grafickú notáciu zachytenia leitmotifu vzorov, ktorá je doplnená o pravidlá zaznamenané v textovej forme. Neopiera o existujúcu technológiu ako tomu bolo predchádzajúcich prípadoch (UML, OWL), čo dalo autorom viac voľnosti a priestoru, keďže nemuseli brať v úvahu obmedzenia, ktoré by plynuli z jej použitia.

Leitmotif vzoru je zachytený pomocou tzv. metaschémy. Tá pozostáva z tzv. fragmentov a vzťahov medzi nimi. Fragmenty zodpovedajú jednotlivým roliam vzoru (reprezentujúcim štrukturálne entity aj entity správania), v rámci diagramu sú zobrazené v obdĺžnikoch. Okrem fragmentov zodpovedajúcich roliam obsahuje diagram aj jeden špeciálny fragment vzoru, reprezentujúci vzor ako celok. Vzťahy medzi vzormi sú zobra-

zené v kosoštvorcoch. Okrem pomenovania vzťahu obsahuje opis vzťahu aj definíciu minimálnej a maximálnej kardinality vzťahu, čím dochádza k definícií početnosti zúčastnených rolí. Príklad metaschémy zachytávajúcej štruktúru vzoru Observer sa nachádza na obrázku 5-15.

Súčasťou opisu vzoru sú aj textovo definované pravidlá. Príklad takého pravidla pre vzor Observer je:

```
for each ConcreteObserver
cardinality_of(Subject.observer) =
cardinality_of(ConcreteObserver.implements)
```



Obrázok 5-15. Metaschéma vzoru Observer [31].

5.2.5 *Prístup založený na formálnych metódach*

Práca [6] predstavuje prístup zahŕňajúci formálnu špecifikáciu a verifikáciu návrhových komponentov a vzorov (prístup pracuje so vzormi ako s komponentmi na úrovni návrhu). Poskytuje systematický prostriedok konštrukcie formálnych modelov návrhových vzorov, ktoré sú generické (nezávislé od aplikačnej domény) a vo vhodnej forme pre strojové odvodzovanie. Okrem toho definuje vlastnosti týkajúce sa ako jednotlivých vzorov tak aj celých systémov strojovo spracovateľným spôsobom, čím otvára možnosti formálnej verifikácii.

V rámci prístupu sa pre návrhové vzory definujú Štrukturálne kontrakty zachytávajúce štruktúru vzoru, Kontrakty správania zachytávajúce správanie vzoru a vlastnosti zachytávajúce pravidlá, ktoré identifikujú problémy v rámci celkového návrhu. V nasledujúcich podkapitolách budú tieto časti opisu návrhových komponentov podrobnejšie opísané.

Štrukturálny kontrakt

V rámci Štrukturálneho kontraktu vzoru je definovaná štruktúra vzoru a operácie umožňujúce vykonávať inštanciáciu a evolúciu inštancie vzoru a integráciu viacerých inštancií vzorov. Opis štruktúry komponentu pozostáva z týchto častí :

Konštanty – predstavujú množinu tried *C,* množinu atribútov tried *AV,* množinu metód *M,* množinu typov *T* a množinu prístupových práv *AR*.

Predikáty

- Predikáty rolí *RP* (*Role predicates*) tvoria množinu informácií o roliach.
- Predikáty vzťahov *CP* (*Connection predicates*) tvoria množinu definícií vzťahov medzi jednotlivými rolami a spôsob ich prepojenia.
- Predikáty akcií *AP* (*Action predicates*) tvoria množinu obsahujúcu opisy akcií, ktoré môže vykonávať rola návrhového vzoru.
- Množinové predikáty *SP* (*Set predicates*) tvoria množinu prezentujúcu informácie o ľubovoľnom počte inštancií role.
- Kvantifikačné predikáty *QP* (*Quantification predicates*) tvoria množinu definujúcu spôsob kvantifikácie množiny elementov.

Príklad Štrukturálneho kontraktu vzoru Observer:

```
C = {Subject, Observer, ConcreteObserver, ConcreteSubject}
AV = {subject, observers, subjectState, observerState}
M = {attach, detach, getState, update, notify, append, remove}
T = {void, DataType}
AR = {public, protected, private}

RP = {
  abstractclass(Subject),
  abstractclass(Observer),
  class(ConcreteObserver),
  class(ConcreteSubject),
  variable(Subject, private, observers, Observer),
  variable( ConcreteObserver, private, subject, Subject),
  variable(ConcreteSubject, private, subjectState, DataType),
  variable(ConcreteObserver, private, observerState, DataType),
  method(Subject, public, attach, void),
  method(Subject, public, detach, void),
  method(Subject, public, notify, void),
  method(ConcreteSubject, public, GetState, DataType),
  method(Observer, public, update, void),
  method(ConcreteObserver, public, update, void)
}
CP = {
  inherit(Observer, ConcreteObserver),
  inherit(Subject, ConcreteSubject)
}
AP = {
  invoke(Subject, attach, observers, append),
  invoke(Subject, detach, observers, remove),
  invoke(Subject, notify, Observer, update),
  invoke(ConcreteObserver, update, subject, GetState),
  return(ConcreteSubject, GetState, subjectState)
}
```

```
SP = {
  element(ConcreteSubject, ConcreteSubjectSet),
  element(ConcreteObserver, ConcreteObserverSet)
}
QP = {
  forall(element(ConcreteSubject, ConcreteSubjectSet), CS),
  forall(element(ConcreteObserver, ConcreteObserverSet), CO)
}
CS = {
  class(ConcreteSubject),
  inherit(Subject, ConcreteSubject),
  variable(ConcreteSubject, private, subjectState),
  method(ConcreteSubject, public, GetState),
  return(ConcreteSubject, GetState, subjectState)
}
CO = {
  class(ConcreteObserver),
  inherit(Observer, ConcreteObserver),
  variable(ConcreteObserver, private, subject, Subject),
  variable(ConcreteObserver, private, observerState, DataType),
  method(ConcreteObserver, public, update, void),
  invoke(ConcreteObserver, update, subject, GetState),
  invoke(ConcreteObserver, ConcreteObserver, subject, attach),
  invoke(ConcreteObserver, destructor, subject, detach)
}
```

Tabuľka 5-3 obsahuje opisuje spôsob realizácie operácií, ktoré sú definované nad opísaným modelom Štrukturálneho kontraktu.

Tabuľka 5-3. Operácie vykonávané nad Štrukturálnym kontraktom.

Operácia	Spôsob realizácie operácie
Inštanciácia	Inštancie definovaných návrhových vzorov sa vytvárajú nahradením všeobecných konštánt reprezentujúcich role vzoru názvami konkrétnych inštancií.
Evolúcia	Evolúcie definovaných inštancií návrhových vzorov sa realizujú doplnením alebo odstránením konštánt a predikátov.
Integrácia	Integrácia inštancií je realizovaná zjednotením množín všetkých konštánt a predikátov integrovaných inštancií. Na to, aby mala integrácia zmysel je potrebné, aby prienik týchto množín nebol prázdny.

Kontrakt správania

Kontrakt správania opisuje dynamické informácie ako kooperácia medzi objektmi zúčastnenými v návrhovom vzore či vytváranie nových objektov. Modeluje sa pomocou kolaborácií skupín objektov hrajúcich rôzne role a pracujúcich na vykonaní spoločnej činnosti.

Na vytváranie formálneho sémantického modelu Kontraktu správania autori zvolili procesný kalkulus pre jeho možnosti modelovať správanie a súbežné spracovanie. Procesný kalkulus umožňuje hierarchický opis procesov, ktorý je možné verifikovať, analyzovať a odvodzovať nad ním. Ako konkrétny procesný kalkulus bol zvolený CCS [24], pre silu jeho modelovacieho jazyka a dobrú podporu nástrojov určených na kontrolu modelov. Pre potreby prístupu je CCS syntax nasledovná:

```
P ::= 0 | a.P | P + P | P|P |P \ A | P[f] | x
```

kde *P* predstavuje proces, *a.P* sekvenčnú kompozíciu, *P* + *P* nedeterministickú voľbu, *P*|*P* paralelné spracovanie, *P* \ *A* obmedzenie, *P[f]* preznačenie, 0 prázdny proces a *x* premennú. Pri definovaní kontraktu správania je potrebné definovať:

- *P* – konečnú množinu procesov,
- *IP* – konečnú množinu vstupných portov pripojených k procesom. Proces prijíma vstupné správy cez svoje vstupné porty,
- *OP* – konečnú množinu výstupných portov pripojených k procesom. Proces vysiela výstupné správy cez svoje výstupné porty,
- *IM* – konečnú množinu vstupných správ procesov,
- *OM* – konečnú množinu výstupných správ procesov,
- IM_I – konečnú množinu vstupných správ procesov pochádzajúcich mimo komponentu.
- OM_I – konečnú množinu výstupných správ procesov smerovaných mimo komponentu.
- *A* – konečnú množinu akcií, ktoré môžu byť vykonané procesom.

Príklad Kontraktu správania návrhového vzoru Observer:

```
P = {aConcreteSubject, ConcreteObserverA, ConcreteObserverB}
IP = {O2S, S2O, Ntfy, Input}
OP = {O2S, S2O, Ntfy}
IM = {Attach, Detach, SetState, GetState, Update, Notify, Change}
OM = {Attach, Detach, SetState, GetState, Update, Notify}
IM_I = {Change}
OM_I = { }

CCS opis vzoru:

OberverBehavior ::= Subject(aConcreteSubject) |
                    Observer(ConcreteObserverA) |
                    Observer(ConcreteObserverB)
Observer(Name) ::= out(O2S, Attach).Observer
                   out(O2S, Detach).Observer +
                   in(Input, Change).action(Change).
                     out(O2S, SetState).Observer +
                   in(S2O, Update).action(Update).
                     out(O2S, GetState).Observer
Subject(Name) ::= in(O2S, Attach).action(Attach).Subject +
                  in(O2S, Detach).action(Detach).Subject +
                  in(O2S, SetState).action(SetState).
                    out(Ntfy, Notify).Notifying
Notifying ::= in(Ntfy, Notify).action(Notify).
                out(S2O, Update).Updating
Updating ::= in(O2S, GetState).action(GetState).Subject
```

Podobne ako pri Štrukturálnom kontrakte, aj pri Kontrakte správania sú definované operácie inštanciácie, evolúcie a integrácie. V rámci kontextu tejto kapitoly sa nimi nebudeme podrobnejšie zaoberať.

Verifikácia kontraktov

Výsledkom operácií inštanciácie, evolúcie či integrácie môžu byť pri nesprávnom vykonaní nesprávne inštancie návrhových komponentov. Aby sa zabránilo takýmto dôsledkom, boli definované vlastnosti, ktoré identifikujú chybné miesta vo vytvorených inštanciách. Vlastnosti sú koncipované tak, že sú pravdivé v prípade prítomnosti problému. Ku každej vlastnosti sú definované sady pravidiel, ktoré identifikujú jednotlivé konkrétne problémy.

Tabuľka č. 5-4 opisuje vlastnosti identifikujúce chyby v Štrukturálnych kontraktoch. Obdobným spôsobom sú definované aj vlastnosti týkajúce sa Kontraktov správania.

Súčasťou práce [6] je aj ukážka realizácie verifikačného nástroja, v rámci ktorej bol ako implementačný jazyk pre modely a verifikačné vlastnosti zvolený jazyk XSB Prolog [37].

Tabuľka 5-4. Vlastnosti identifikujúce chyby v modeloch štrukturálnych kontraktov.

Vlastnosť	Opis vlastnosti + príklad
Konzistencia	Vlastnosť je pravdivá, ak sa v modeli štrukturálneho kontraktu nachádza nekonzistencia. Príklad problému: Trieda dedí sama zo seba.
Integrita rolí	Vlastnosť zabezpečuje, že atribút alebo metóda nemôžu byť definované bez definície triedy, do ktorej patria. Príklad problému: Atribút nie je priradený ani abstraktnej ani obyčajnej triede.
Integrita akcií	Vlastnosť zabezpečuje, že predikáty akcií sú vykonávané na dobre definovanom základe Príklad problému: Ak metóda B definovaná v triede A volá metódu D v triede C, niektorá z metód B,D alebo tried A,C nie je definovaná
Vzťahy	Vlastnosť zabezpečuje, že v rámci definície štrukturálneho kontraktu nie sú zle definované vzťahy medzi elementmi. Príklad problému: Vo viacnásobnom dedení obsahujú obaja predkovia metódu s rovnakým názvom.

5.2.6 Porovnanie prístupov

V predchádzajúcich kapitolách bolo prezentovaných 5 rôznych prístupov určených na zachytenie leitmotifov návrhových vzorov. Dva z nich sú založené na UML metamodeli, jeden na báze Webu so sémantikou, jeden predstavuje čisto grafické riešenie a jeden sa opiera o metódy formálnej logiky implementovateľné pomocou Prologu. V nasledujúcich kapitolách porovnáme tieto prístupy medzi sebou.

Porovnanie z hľadiska použitých technológií

Technológie, o ktoré sa tieto prístupy opierajú, majú vplyv na možnosti, ktoré so sebou prinášajú. V nasledujúcej časti sú porovnané možnosti plynúce z využitia jednotlivých technológií.

UML metamodel umožňuje veľmi dobre definovať opis pozostávajúci z pravidiel pre možnú kompozíciu objektovo orientovaných stavebných elementov, pomocou ktorých sa dajú opísať vzory. Okrem toho môžeme rozšíriť tento opis o obmedzenia vo forme OCL, ktorými sa dajú presnejšie vyšpecifikovať pravidlá spolupráce OO elementov. Spolu tvoria tieto technológie komplexné riešenie schopné zachytiť značnú časť leitmotifu vzorov.

Výhodou tohto prístupu je použitie štandardných riešení pre opis OO softvéru, čoho dôsledkom by mohla byť dostupnosť nástrojov umožňujúca pracovať s takto uchovanými informáciami, podobne ako existencia pridružených štandardov, napr. štandard XMI [36] určený na výmenu a zdieľanie takto uložených znalostí.

Nevýhoda prístupu rovnako tkvie v použití UML. UML je totiž veľmi komplikovaný a elementy jeho metamodelu komplexne prepojené, čo výrazne znižuje jeho pochopiteľnosť a rozširovateľnosť. Problém nastáva, ak by sme potrebovali uložiť do leitmotifu znalosti, ktoré sa pomocou štandardnej kombinácie UML+OCL zachytiť nedajú. V tomto prípade sa javia dve možnosti: použiť vstavané rozšírenie UML, čím dôjde k vytvoreniu vlastného UML Profilu (také riešenie využíva jeden z prístupov) alebo rozšíriť samotný UML metamodel v prípade, ak by sme ani UML Profilom nedokázali zachytiť naše požiadavky. Vzhľadom na komplexnosť UML metamodelu je však jeho rozšírenie veľmi problematické, takže takéto riešenie možno považovať za jedno z krajných. Za mierny problém možno považovať aj výmenu takto uložených informácií. Síce existuje štandard XMI definovaný skupinou OMG, ktorý je založený na štvorvrstvovej MOF [23] architektúre, existuje a používa sa však už aj jej klon v podobe Ecore od skupiny Eclipse. To by mohlo byť varovaním pred možným hroziacim efektom Babylonskej veže, keď by napriek jednému štandardu existovalo niekoľko paralelných, čo by výrazne zredukovalo možnosti znovupoužitia takto uložených znalostí.

Ako alternatívu k UML prístupu možno uvažovať prístupy založené na iniciatíve Webu so sémantikou a formálnych logických metódach. Tie poskytujú väčšiu voľnosť pri definovaní spôsobu uchovávanie znalostí, ktorá môže byť užitočná pri pokusoch o zachytenie atypických znalostí. Obe techniky so sebou prinášajú aj nástroje schopné odvodzovať nad definovanou bázou znalostí, ktoré možno využiť pri operáciách s danými znalosťami. V prípade formálnych logických metód je vhodným nástrojom jazyk Prolog, v prípade Webu so sémantikou možno použiť odvodzovacie nástroje vytvorené v rámci tejto iniciatívy, akými sú napr. Racer alebo FaCT++. Jedným z primárnych cieľov iniciatívy Webu so sémantikou je umožniť zdieľanie znalostí v strojovo spracovateľnej forme, takže túto schopnosť je možné plne využiť aj v tomto prípade.

Poslednú skupinu tvorí čisto grafický prístup. Tým, že sa neoperia o žiadny štandard alebo technológiu, sa síce otvárajú možnosti zachytenia veľkého množstva znalostí, na druhej strane je však použiteľnosť takto uložených znalostí z pohľadu možností strojového spracovania veľmi nízka.

Porovnanie z hľadiska obsahu

Z hľadiska množstva znalostí, ktoré je množné pomocou opísaných prístupov zaznamenať, možno medzi nimi identifikovať rozdiely. Veľkým prínosom prístupu Precízne modelovanie vzorov na základe UML metamodelu je podrobné rozpracovanie problematiky rolí, v rámci ktorej došlo okrem iného k zavedeniu pojmu dimenzia role. Kladom iných prístupov je zase opis nielen z hľadiska statickej štruktúry, ale aj dynamického správania sa vzoru z pohľadu interakcie jeho účastníkov a stavového opisu. Prístup založený na formálnych metódach so sebou prináša nielen uchovanie informácií o vzore ale aj možnosti zachytenia konkrétnych inštancií, čo mu následne umožňuje vykonávať zmeny podľa evolučných pravidiel nad týmito inštanciami a verifikovať ich štruktúru.

Podrobnejšie informácie o možnostiach, ktoré jednotlivé prístupy umožňujú sa nachádzajú v tabuľke 5-5. Z nej vyplýva, že najviac informácií zachytávajú prístupy Precízne modelovanie vzorov na základe UML metamodelu, Rozšírený OWL prístup a Prístup založený na formálnych metódach. Rovnako však z tabuľky vyplýva, že ani jeden z prístupov nemá výraznejšie navrch oproti ostatným, každý má svoje síce svoje prednosti, ale v inom naopak zaostáva. Preto sa zrejme v budúcnosti nevyhneme modifikáciám niektorého z prístupov, ak by sme chceli, aby poskytoval všetky schopnosti, ktoré poskytujú aj ostatné prístupy.

Tabuľka 5-5. Porovnanie prístupov z hľadiska obsahu.

	UML 1	UML 2	Pôvodné OWL	OWL po rozšírení	Grafická notácia	Formálne metódy
Prístup založený na roliach	áno	áno	áno	áno	áno	áno
Zachytenie dynamického správania sa	áno (pomocou IPS a SMPS)	nie	nie	nie	nie	áno (pomocou procesného kalkulu)
Podpora pre evolúciu inštancií vzorov	nie	nie	nie	áno	nie	áno
Existujúca priama podpora pre verifikáciu inštancií	nie	nie	nie	nie	nie	áno
Zachytenie informácie o kardinalite rolí	áno	áno	nie	áno	áno	nie
Zachytenie informácie o kardinalite rolí v závislosti od kardinality iných rolí	nie	áno	nie	áno	áno	nie

Pozn. UML 1 – Metamodelovací jazyk založený na roliach (Role Based Metamodeling Language), opísaný v kapitole 5.2.1; UML 2 – Precízne modelovanie vzorov na základe UML metamodelu, opísané v kapitole 5.2.2.

5.3 Kompozícia vzorov

Myšlienka kompozície viacerých vzorov za účelom dosiahnutia požadovaného cieľa bola prezentovaná už autorom celej iniciatívy nasadenia vzorov Christopherom Alexandrom. Vo svojej práci A Pattern Language [1] diskutuje o spôsoboch kompozície vzorov. "*It is possible to make buildings by stringing together patterns, in a rather loose way. A building made like this, is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such a way that many patterns overlap in the same physical space: the building*

is very dense; it has many meanings captured in small space; and through this density, it becomes profound." Autor identifikuje dva spôsoby kompozície vzorov: zreťazenie, v rámci ktorého sú vzory umiestnené voľne vedľa seba a pevné spojenie, v rámci ktorého sa jednotlivé časti spolupracujúcich vzorov prekrývajú.

Práca GoF [12] sa tiež zaoberá problematikou spolupráce návrhových vzorov. Katalogizovaný opis vzorov obsahuje sekciu „Príbuzné vzory“, ktorá informuje o možnosti použitia daného vzoru v spolupráci s iným. Môžeme to považovať za základy interakcie medzi návrhovými vzormi, aj keď práca GoF neodpovedá na otázku, ako presne by jednotlivé vzory mali byť spolu prepojené a ako by mali spolupracovať. Pokusy o zodpovedanie podobných otázok môžeme nájsť v prácach viacerých autorov.

Všeobecne existujú dva hlavné prístupy ku kompozícii vzorov:

- Kompozícia správania je založená na elementoch hrajúcich určité role v rôznych vzoroch. Prístup používa modely založené na roliach, ktoré zachytávajú statické a z časti aj dynamické aspekty vzorov v jednom diagrame. UML nepodporuje taký spôsob integrácie, čo znamená menšiu podporu pre tento prístup. Tento spôsob kompozície vzorov bol použitý napr. v [17].
- Štrukturálna kompozícia je založená na spojení vzorov podľa ich diagramov tried. Tento prístup využíva UML diagramy, takže sa vyhýba problémom s podporou vyskytujúcou sa v predošlom prístupe. Na druhej strane majú diagramy tried menšiu expresivitu v porovnaní s diagramami založenými na roliach. Príkladom tohto spôsobu kompozície je práca [30].

Nasledujúce kapitoly obsahujú podrobnejší opis vybraných prístupov ku kompozícii návrhových vzorov.

5.3.1 Vzorovo orientovaná analýza a návrh

Práca Vzorovo orientovaná analýza a návrh (*Pattern Oriented Analysis and Design - POAD*) [38] [39] predstavuje systematický prístup ku kompozícií návrhových vzorov. Vychádza pritom z myšlienky, že základnými stavebnými blokmi softvéru by mali byť spolupracujúce inštancie návrhových vzory, ktoré dokážu poskytnúť funkčne overené riešenia. Aby sa zjednodušil proces vývoja obdobným spôsobom, práca POAD predstavuje metodológiu, ktorá vedie vývojárov od počiatočných analýz k optimalizovanému OO návrhu, ktorý je založený na použití kooperujúcich inštanciách návrhových vzorov.

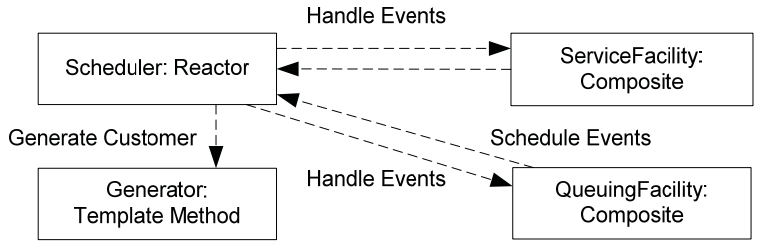
V rámci POAD sa pozeráme na návrhový vzor ako na komponent návrhu, ktorý je s ostatnými komponentmi či inými elementmi návrhu spojený pomocou rozhraní. Vzor má teda definované rozhrania, pomocou ktorých môže spolupracovať s inými vzormi. Taký prístup so sebou prináša nasledujúce výhody:

- skrytie detailov návrhových vzorov, ktoré nie sú relevantné vo vyššej úrovni abstrakcie,
- odlíšenie tých častí návrhových vzorov, ktoré sú kľúčové pre kompozíciu,
- flexibilita umožňujúca aplikovať rôzne implementácie vnútorných častí vzorov.

Pre vizuálnu podporu metodológie, POAD definuje tri nové modely, ktoré sú používané pri kompozícii inštancií vzorov. Úlohou týchto modelov je sprehľadniť a uľahčiť proces návrhu softvéru pozostávajúceho z komponentov. Problémom bežných UML modelov je

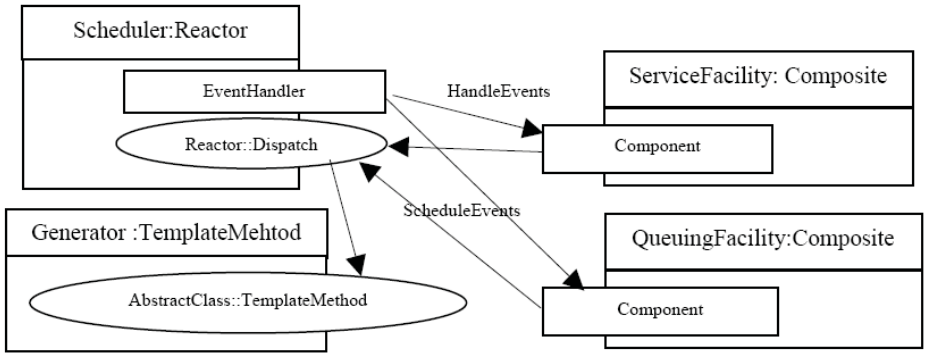
nedostatočné vyjadrenie prítomnosti inštancie vzoru v návrhu, čo je v prípade, keď je potrebné pracovať s viacerými inštanciami vzorov, značný nedostatok. Modely definované v rámci POAD tento problém odstraňujú, čím sa stávajú vhodnejšie na vytváranie kompozícií inštancií návrhových vzorov.

Prvým modelom je Model na úrovni vzorov (*Pattern Level Model*), ktorý zachytáva spoluprácu vzorov na vysokej úrovni abstrakcie. Obsahuje inštancie návrhových vzorov a vzťahy opisujúce spôsob, akým budú navzájom inštancie spolupracovať. Príklad Modelu na úrovni vzorov sa nachádza na obrázku 5-16.



Obrázok 5-16. Príklad Modelu na úrovni vzorov [39].

Rozšírením Modelu na úrovni vzorov je Model na úrovni vzorov s rozhraniami (*Pattern-Level with Interfaces Model*). Oproti prvému modelu obsahuje rozšírenie inštancií vzorov o ich rozhrania. Vzťahy medzi inštanciami sú v tomto modeli realizovane výlučne cez rozhrania. Príklad Modelu na úrovni vzorov s rozhraniami sa nachádza na obrázku 5-17.

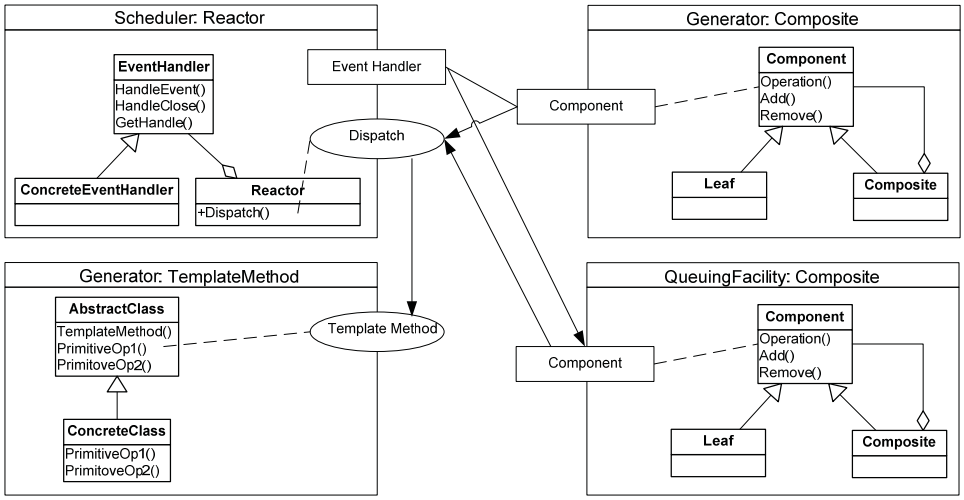


Obrázok 5-17. Príklad Modelu na úrovni vzorov s rozhraniami [39].

Posledným modelom je Podrobný model na úrovni vzorov (*Detailed Pattern-Level Model*). Predchádzajúci model rozširuje o interné časti vzorov, ktoré zostávali doposiaľ skryté. Okrem toho pridáva prepojenia medzi internými časťami vzorov a rozhraniami komunikujúcimi s okolitými komponentmi. Príklad Podrobného modelu na úrovni vzorov sa nachádza na obrázku 5-18. Samotná metodológia pozostáva z troch hlavných krokov:

1. Analýza – Počas analýzy sa získavajú požiadavky kladené na vyvíjaný systém. Z pohľadu POAD je hlavným výsledkom zoznam vzorov, ktoré budú navzájom spolupracovať.

2. Návrh na vyššej úrovni – V tejto fáze sa definuje spôsob spolupráce medzi vzormi. Postupne sa vytvorí Model na úrovni vzorov, Model na úrovni vzorov s rozhraniami a Podrobný model na úrovni vzorov. Výsledkom kroku je návrh pozostávajúci z inštancií vzorov, ktorý nie je optimalizovaný.
3. Zdokonalenie návrhu – V poslednom kroku dochádza k optimalizácií návrhu. Pri prepájaní inštancií mohlo dôjsť napr. k vzniku duplicitných rozhraní, prípadne tried, ktorých význam je minimálny. V tomto kroku sú také nedostatky odstránené, čoho výsledkom je výsledný optimalizovaný návrh systému pozostávajúci zo spolupracujúcich inštancií návrhových vzorov.



Obrázok 5-18. Príklad Podrobného modelu na úrovni vzorov [39].

Práca POAD definuje metodológiu tvorby softvéru na základe vzorov použitých ako komponenty návrhu, nedefinuje však konkrétne spôsoby prepojenia vzorov. Spolieha sa na vedomosti a skúsenosti vývojárov, ktorí majú sami identifikovať vhodné spojenia medzi vzormi. Jedinou konkrétnou pomôckou pre vývojárov je mini katalóg 10 návrhových vzorov (Strategy, Observer, Composite, Reactor, Template method, Proxy, Abstract factory, Builder, Mediator, Command) obsahujúci špecifikácie rozhraní týchto vzorov pre interakciu s inými vzormi.

Na to, aby sa prístup presadil v praxi, je potrebná jeho podpora CASE nástrojmi, vďaka ktorej by bolo možné efektívne vytvárať všetky súvisiace modely, čím by sa zjednodušil proces návrhu od definície komponentov až po optimalizovaný návrh. CASE nástroj, ktorý by to umožňoval, doposiaľ neexistuje, čo výrazne redukuje potenciál aplikovania POAD prístupu v reálnom prostredí.

5.3.2 Hybridizácia vzorov

Hybridizácia vzorov (*Pattern hybridization*) [28] predstavuje iný prístup, ktorý prináša viac formalizmu do procesu kompozície vzorov. Jeho cieľom je na základe formálnych metód určiť, či sú dva vzory vhodné na kompozíciu. V terminológii tohto prístupu sa kompozícia

dvoch vzorov nazýva hybridný vzor, z čoho vyplýva aj pomenovanie prístupu. Vzory sú v tomto kontexte reprezentované pomocou trojice vlastností klasifikačných kritérií:

```
Vzor [Aplikované na, Účel, Jedinečný význam],
```

kde jednotlivé kritéria majú nasledovný význam:

- *Aplikované na* – reprezentuje entitu, na ktorú sa vzor aplikuje. Môže mať nasledovné kategórie: objekty (*objects*), rodiny objektov (*object-families*), prepojené rodiny objektov (*related-object families*).
- *Účel* – reprezentuje účel aplikácie vzoru. Kategórie kritéria sú definované v [15]: funkcionalita (*functionality*), rozhranie (*interface*), stav (*state*), prístup (*access*), komunikácia (*communication*), uloženie (*physicality*), inštanciácia (*instantiation*).
- *Jedinečný význam* – poskytuje krátky verbálny opis jednoznačného správania, ktoré vzor poskytuje.

Napríklad vzor Singleton je pomocou trojíc opísaný nasledovne:

```
Singleton[objekty, inštanciácia, trieda potrebuje byť inštanciovaná
max. jednou inštanciou].
```

Prístup definuje dva odlišné spôsoby interakcie, podľa ktorých môžu byť vzory hybridizované: Používa (*Uses*) a Kombinuje (*Combines*)

Používa

Vzor používa iný vzor na riešenie niektorého zo svojich podproblémov. Vzory môžu byť hybridizované týmto spôsobom, ak je hodnota ich kritéria Účel rovnaká. Sú definované exaktné pravidlá určujúce za akých podmienok a ako môžu byť vzory podľa tohto kritéria hybridizované:

- *Vzor1[spojené rodiny objektov, ÚČEL, A]* používa *Vzor2[objekt, ÚČEL, B]*, čím vzniká nový vzor *HybridnýVzor[spojené rodiny objektov, ÚČEL, A + ΔB]*, kde ÚČEL predstavuje spoločný účel pre oba vzory a A + ΔB predstavuje význam prvého vzoru doplnený o časť významu druhého vzoru.
- *Vzor1[spojené rodiny objektov, ÚČEL, A]* používa *Vzor2[rodiny objektov, ÚČEL, B]*, čím vzniká nový vzor *HybridnýVzor[spojené rodiny objektov, ÚČEL, A + ΔB].*
- *Vzor1[rodiny objektov, ÚČEL, A]* používa *Vzor2[spojené rodiny objektov, ÚČEL, B]*, čím vzniká nový vzor *HybridnýVzor[rodiny objektov, ÚČEL, A + ΔB].*
- *Vzor1[rodiny objektov, ÚČEL, A]* používa *Vzor2[rodiny objektov, ÚČEL, B]*, čím vzniká nový vzor *HybridnýVzor[rodiny objektov, ÚČEL, A + ΔB].*
- *Vzor1[objekty, ÚČEL, A]* používa *Vzor2[objekty, ÚČEL, B]*, čím vzniká nový vzor *HybridnýVzor[objekt, ÚČEL, A + ΔB].*

Príklad hybridného vzoru podľa interakcie Používa je Prototype používajúci Singleton:

```
Prototype [objekty, inštanciácia, klonovanie]
```

používa

```
Singleton[objekty, inštanciácia, trieda potrebuje byť
          inštanciovaná max. jednou inštanciou]
```

čím vzniká

```
Hybridný vzor[objekty, inštanciácia, klonovanie s využitím
              jednej inštancie]
```

Kombinuje

Vzor sa kombinuje s iným vzorom, aby sa dosiahlo zmiešané správanie. Na rozdiel interakcie Používa, interakcia Kombinuje spája vzory s rozličnou hodnotu kritéria Účel. Podobne ako v predošlom prípade, aj pre túto interakciu sú definované formálne pravidlá určujúce, ktoré vzory a ako môžu spolu tvoriť hybridný vzor.

Na úrovni návrhu sú definované tri spôsoby kompozície: zdieľanie hierarchie, väzba medzi vzormi a interakcie podtried. Výberom spôsobu kompozície sa prístup nezaoberá, opäť je ponechaný na schopnosti a skúsenosti vývojára. Prístup bol úspešne implementovaný v rámci práce [18].

5.3.3 Zhodnotenie prístupov ku kompozícii vzorov

Oba predchádzajúce prístupy sa snažia riešiť problém kompozície vzorov univerzálnym spôsobom. Prvý z nich predpokladá komponentový spôsob vývoja, pričom prepojenie komponentov necháva plne v zodpovednosti vývojára bez toho, aby riešil, či je dané spojenie vhodné, alebo nie. Druhý prístup síce definuje pravidlá, ktoré naznačia, či sú dva vzory vhodné na vzájomnú kompozíciu, no opäť nedefinuje, akým spôsobom by sa kompozícia mala realizovať. Je zrejmé, že oba prístupy sa spoliehajú na znalosti vývojárov, ktorí ich majú používať, pričom im pomáhajú len čiastočne. Keďže správna kompozícia vzorov je zložitý problém, nie je možné očakávať, že by sa dal hneď plne automatizovať.

Prístupy sú tvorené tak, aby viedli k čo možno najčastejšiemu použitiu vzorov v riešeniach. Najmä prvý ich predstavuje ako overené riešenia, ktoré je vhodné použiť ihneď pri prvej príležitosti. Taký spôsob sa môže javiť na prvý pohľad vhodný, no neskôr sa môžu prejaviť jeho nedostatky. Tým, že sa vývojár snaží umiestniť do softvéru čo možno najviac vzorov, softvér sa stáva zbytočne univerzálnym, komplikovaným a ťažko pochopiteľným. Jedným z dôvodov aplikácie návrhových vzorov je príprava softvéru na možné zmeny v budúcnosti.

Napríklad, ak sú takéto zmeny málo pravdepodobné a dlhodobo nenastávajú, použitie návrhového vzoru v aplikácii môže byť zbytočné. Použitie zbytočných návrhových vzorov so sebou prináša problémy. Prvým je potreba uchovania informácií o použití vzorov, aby v budúcnosti pri zmenách nedošlo k poškodeniu ich vnútornej štruktúry. Druhým je znížená efektivita zdrojového kódu, ktorá je častým dôsledkom nasadenia vzorov. Preto by sa malo pristupovať k aplikácii vzorov obozretne. Aj podľa názorov E. Gammu [34] by sa mali používať len v prípadoch, keď to vyplýva zo situácie.

Keďže opísané univerzálne prístupy nehovoria nič o tom, ako by mali byť vzory kombinované na úrovni návrhu, riešením by mohlo byť vytvorenie katalógu korektných kompozícii návrhových vzorov. Taký katalóg by mal vysvetliť význam spojenia vzorov a definovať spôsob, akým dané vzory korektne prepojiť na úrovni návrhu. Myšlienka takého katalógu bola podrobnejšie opísaná v práci [19].

5.4 Modelovanie s návrhovými vzormi na vyššej úrovni abstrakcie

Pri používaní návrhových vzorov v praxi sa spravidla stretávame iba so štandardnými UML modelmi. Obsahujú konkrétne inštancie vzorov vo forme spolupracujúcich OO prvkov, pričom informáciu, že ide o použitie nejakého vzoru, naznačujú len pomenovania týchto prvkov. Presná informácia o tom, akú rolu hrá daný prvok diagramu, chýba. Potom sa môže ľahko stať, že vývojár spätne nerozpozná, že ide o použitie určitého vzoru, čo v konečnom dôsledku môže znamenať stratu významu jeho nasadenia, či iné komplikácie. Tým, že v diagramoch štandardne neuchovávame explicitné informácie o vzoroch, často sa v návrhu strácajú, čím sa sami pripravujeme o mnohé výhody, ktoré plynú z ich použitia. Následne vývojári pracujú opäť na OO úrovni návrhu, zatiaľ čo by mohli pracovať s vyššou úrovňou abstrakcie, ktorú vzory prinášajú.

5.4.1 Možnosti práce na vyššej úrovni abstrakcie

Existuje niekoľko riešení, ktoré sa snažia pridať informácie o aplikovaných inštanciách vzorov do modelov zachytávajúcich návrh systému. V rámci práce POAD podrobnejšie opisovanej v kapitole 5.3.1 sú prezentované tri nové modely, ktoré slúžia tomuto účelu. Prístup počíta sa postupným vytváraním modelov od najvyššej úrovne abstrakcie na úrovni komponentov až po optimalizovaný OO návrh. Ten je však zachytený vo forme bežného UML diagramu, takže informácie o aplikovaných návrhových vzoroch vo finálnom modeli opäť chýbajú.

V práci [7] sú opísané rôzne rozšírenia UML diagramov, ktoré dopĺňajú informácie o prítomnosti inštancií vzorov. Tie síce predstavujú prínos pri modelovaní softvérového návrhu s použitím vzorov, stále však kladú dôraz na rozmiestnenie a spoluprácu tried, pričom informácie týkajúce sa vzorov sú až druhoradé. Znamená to, že návrhár pracujúci s takýmito modelmi robí stále na nižšej úrovni abstrakcie, než umožňujú vzory. Samotné značky týkajúce sa vzorov sú do diagramov pridané až po vytvorení modelu tried. To znamená, že návrhár musí najskôr sám pochopiť vzor, manuálne vytvoriť jeho inštanciu v návrhu a tú označkovať tak, aby bolo možné vzor spätne rozpoznať.

Oveľa zaujímavejší prístup by bol, keby bolo možné priamo pracovať na vyššej úrovni abstrakcie - na úrovni vzorov. Návrhár by definoval, aký vzor chce použiť a ako ho chce pripojiť do kontextu riešeného systému. Nástroj by následne zabezpečil vytvorenie inštancie vzoru, ktorú by korektne napojil k ostatným častiam návrhu podľa špecifikácie používateľa. Na to, aby bolo možné nad niečím podobným uvažovať, je potrebné zadefinovať dve základné techniky:

- formu modelovania, pomocou ktorej by bolo možné modelovať na vyššej úrovni abstrakcie vzorov,
- metódy, ktoré by transformovali modely na vyššej úrovni abstrakcie do modelov na klasickej OO úrovni.

Pri uvažovaní o takomto spôsobe riešenia je možné využiť výsledky súvisiace s iniciatívou Modelom riadená architektúra (*Model Driven Architecture* – MDA) [25]. MDA prístup umožňuje modelovať na vyššej úrovni abstrakcie nezávisle od platformy, na ktorej by mal byť výsledný systém nasadený. Následne by malo byť možné po dodaní špecifikácií jednotlivých platforiem transformovať takto vytvorené modely do modelov, ktoré sú posta-

vené pre konkrétne platformy. Z nich by sa následne malo dať vytvoriť konkrétne, v rámci platformy použiteľné riešenie (napr. zdrojové kódy, konfiguračné súbory a pod.).

Pre naše potreby definovania návrhu na vyššej úrovni abstrakcie vzorov sa javí MDA prístup ako vhodné riešenie. Za platformovo nezávislé modely (PIM) považujeme modely na úrovni návrhových vzorov. Za platformovo špecifické modely (PSM) môžeme považovať klasické UML modely obsahujúce konkrétne aplikované inštancie vzorov, z ktorých je neskôr možné vygenerovať zdrojové kódy. Za platformu možno považovať definície štruktúr návrhových vzorov z pohľadu OO návrhu. Zostáva došpecifikovať formu zápisov platformovo nezávislých modelov (PIM) a nástroje, ktoré budú použité na transformáciu z PIM do PSM.

Pred výberom jazyka PIM je nutné uviesť požiadavky, ktoré budú od neho vyžadované. Najdôležitejšou je možnosť modelovania na úrovni vzorov, pričom je nutné mať možnosť pripojiť prvky modelu na úrovni vzorov do kontextu ostatného OO návrhu. Tu sa javí ako najvhodnejšia možnosť používať rozšírené UML modely, pričom samotné prvky na úrovni vzorov budú odlíšené od ostatných pomocou vlastného UML Profilu.

5.4.2 Modelovanie vzorov

V rámci štruktúry návrhových vzorov možno identifikovať role, ktoré vykonávajú určitú doménovo špecifickú funkcionalitu a role, ktorých činnosť je z väčšej časti nezávislá od doménovej oblasti. Takéto role sú viac-menej v rámci vzoru implicitné, vyplývajú z použitia vzoru, no pri približovaní vzoru doménovej oblasti sa nimi nemusíme zaoberať. Touto myšlienkou je motivovaný aj predstavovaný spôsob modelovania vzorov. Jeho princíp tkvie v definovaní použitia samotného vzoru a rolí, ktoré sú doménovo závislé. Doménovo nezávislými rolami sa nemusíme pri modelovaní zaoberať, nakoľko ich prítomnosť je definovaná už prítomnosťou samotnej inštancie vzoru, a teda môžu byť automaticky doplnené v rámci transformácie vzoru do finálneho OO návrhu.

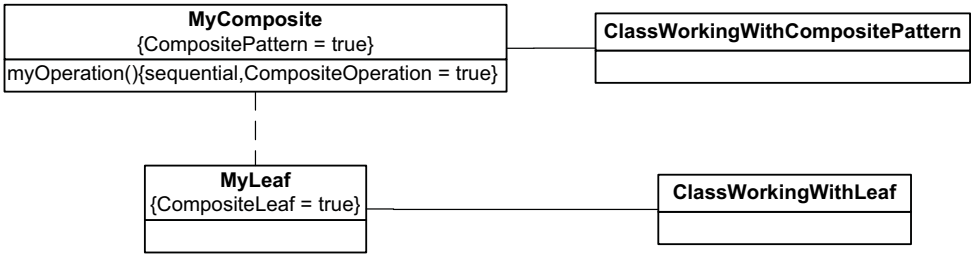
V nasledujúcej časti sú prezentované ukážky modelovania na vyššej úrovni abstrakcie pre vzory Composite a Decorator vrátane ukážky ich možnej kompozície.

Composite

Cieľom vzoru Composite je umožniť zhromažďovanie objektov do stromových štruktúr, pričom by malo byť možné pracovať s jedným objektom rovnako ako so skupinou.

Keď sa pozrieme na vzor z pohľadu, ako by sa mal modelovať, môžeme si všimnúť, že jediné, čo potrebujeme definovať, je trieda (prípadne skupina tried), ktorá má byť uložiteľná v stromovej hierarchii. Ostatné informácie, ako napríklad konkrétna forma vytvárania stromovej štruktúry, nie je pre nás pri modelovaní na vyššej úrovni podstatná. Takéto informácie by mali byť obsiahnuté v konfigurácii vytvárania inštancie vzoru a nie neoddeliteľnou súčasťou modelu.

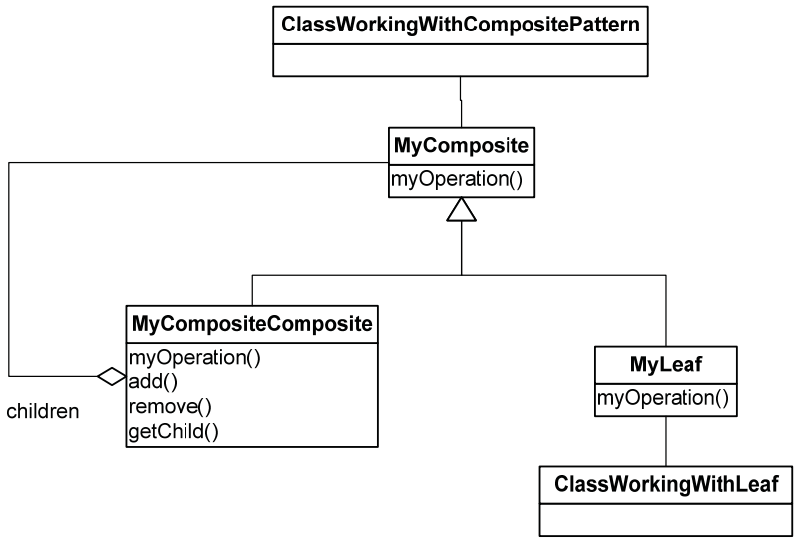
Z tohto dôvodu sa javí vhodné modelovať vzor Composite pomocou dvoch prvkov modelu: prvku reprezentujúceho samotný vzor a triedy, ktorá má byť pomocou vzoru ukladaná. Na obrázku 5-19 sa nachádza ukážka modelovania vzoru na vyššej úrovni.



Obrázok 5-19. PIM so vzorom Composite.

Pomocou UML Profilu (a pre prehľadnosť taktiež šedým podfarbením) sú odlíšené prvky vzorov od ostaných OO prvkov. Prvok `MyComposite` predstavuje inštanciu vzoru ako celok. Jeho súčasťou je aj metóda `myOperation` predstavujúca metódu, ktorú dokáže vykonávať celá hierarchia a rovnako každý jej člen. Trieda `MyLeaf` predstavuje triedu, ktorá môže byť pomocou vzoru uložená. Model zachytáva okrem prvkov na úrovni vzoru aj triedy na úrovni bežného OO návrhu, pričom tie dokážu navzájom spolupracovať: model obsahuje jednu triedu pracujúcu s celou hierarchiou poskytovanou vzorom a jednu triedu, ktorá dokáže pracovať len s triedou predstavujúcou list v hierarchii vzoru.

Opísaný model môže byť transformovaný do modelu klasického OO návrhu, výsledok transformácie pre predchádzajúci model sa nachádza na obrázku 5-20. Ten obsahuje korektne vytvorenú inštanciu vzoru Composite spolu s triedami, ktoré so vzorom spolupracujú tak, ako to definuje model na obrázku 5-19.



Obrázok 5-20. PSM so vzorom Composite.

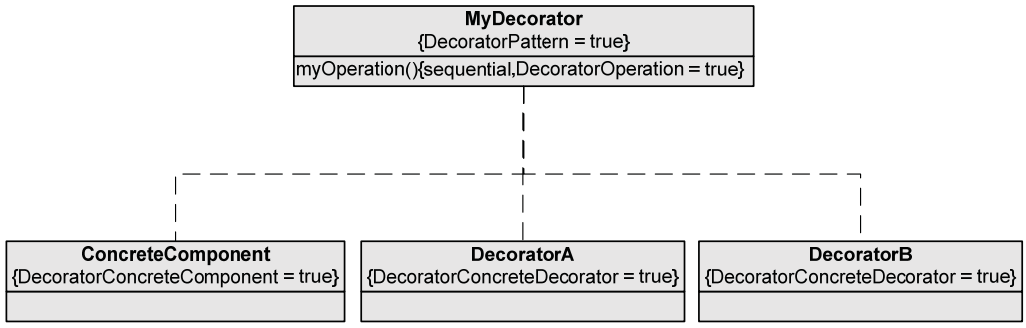
Pri praktickej realizácii transformácie je vhodné označiť elementy výsledného návrhu hrajúce role vo vzoroch vybraným mechanizmom z [7], čím by sa zachovali informácie o aplikácii vzoru aj v návrhu na nižšej úrovni, uľahčila by sa možnosť generovania zdrojového kódu z modelu a súčasne sa by sa zjednodušila možnosť spätného vytvorenia modelu PIM z PSM.

Decorator

Vzor Decorator slúži na dynamické pridávanie zodpovednosti triedam. Predstavuje flexibilnú alternatívu ku klasickému rozširovaniu systému pomocou pridávania potomkov do hierarchií dedenia. Jeho idea spočíva v definovaní tried, ktorých objekty majú byť dekorovateľné a dekorátorov, ktoré majú rozširovať funkcionalitu tried (dekorovať triedy). Princíp dekorovateľnosti spočíva v postupnom volaní metód rozširujúcich dekorátorov v rámci volania pôvodnej metódy.

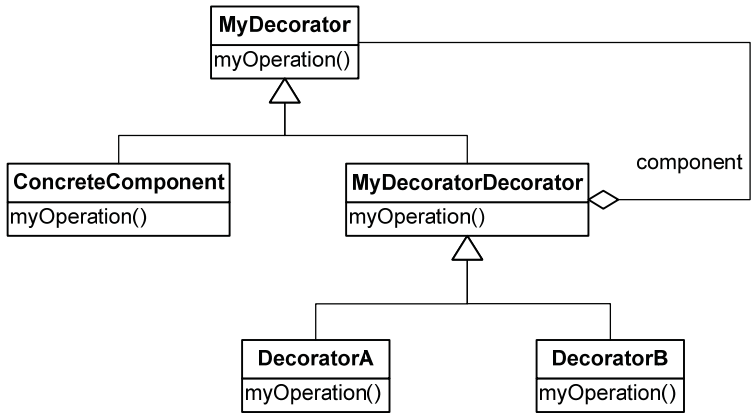
Keď sa opätovne pozrieme na vzor Decorator z pohľadu modelovateľnosti, môžeme si všimnúť, že prioritou je v rámci modelu definovať dekorovateľné triedy a ich dekorátory. Forma, ako tieto triedy konkrétne spolupracujú za účelom dosiahnutia svojho cieľa, pre nás nie je v danom momente podstatná.

Modelovať vzor sme sa rozhodli podobným spôsobom ako v prípade vzoru Composite: definovaním prvku predstavujúcim inštanciu vzoru a triedami predstavujúcimi dekorátory a dekorovateľné triedy. Prvok predstavujúci samotný vzor navyše obsahuje definíciu metódy, ktorej funkcionalita môže byť v rámci inštancie vzoru dekorovateľná. Obrázok 5-21 zachytáva ukážku vyššieho modelu vzoru Decorator pracujúceho s jednou dekorovateľnou triedou `ConcreteComponent` a dvomi dekorátormi `DecoratorA` a `DecoratorB`.



Obrázok 5-21. PIM so vzorom Decorator.

Obrázok 5-22 zachytáva výsledok transformácie modelu z obrázku 5-21 do bežného modelu na úrovni OO návrhu.

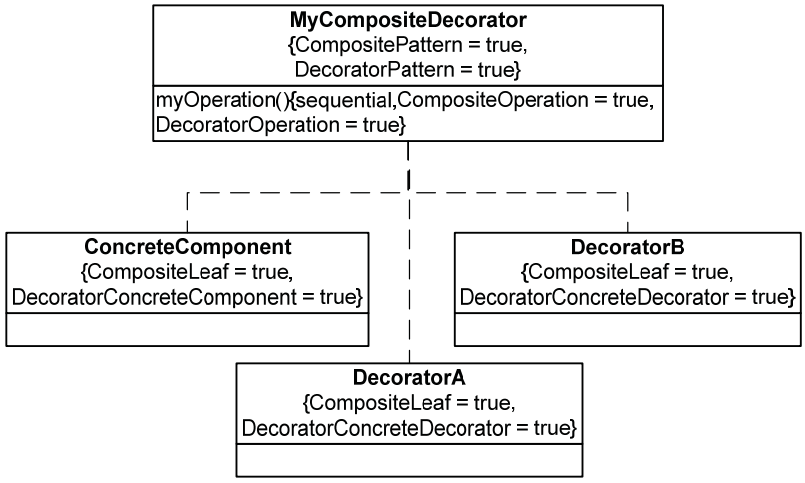


Obrázok 5-22. PSM so vzorom Decorator.

Kompozícia viacerých vzorov

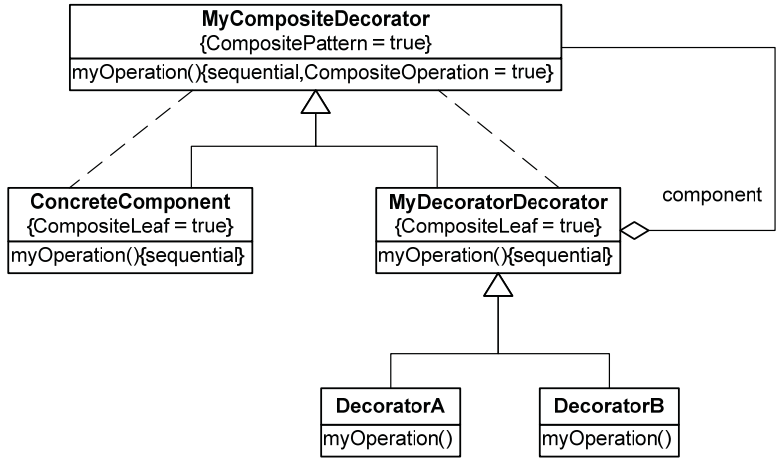
Podobne ako sme pristupovali k modelovaniu jednoduchých inštancií vzorov, môžeme pristúpiť aj k ich kompozícii. Predviesť možnosti modelovania na vyššej úrovni môžeme na spolupráci už opísaných vzorov Composite a Decorator, ktorých spojenie umožňuje vytvárať hierarchické štruktúry dekorovateľných objektov.

Kompozíciu vzorov modelujeme podobne, ako sme v predchádzajúcich príkladoch modelovali samostatné inštancie vzorov. Opäť definujeme jeden prvok modelu, v tomto prípade predstavujúci kompozíciu viacerých vzorov. K tomuto prvku pripojíme triedy hrajúce role jednotlivých vzorov. Príklad takto definovanej kompozície je na obrázku 5-23.



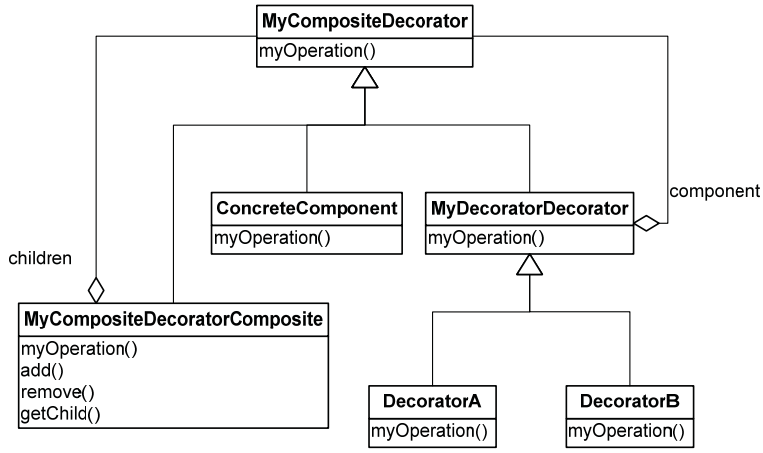
Obrázok 5-23. PIM s kompozíciou vzorov Decorator a Composite.

Model môžeme transformovať podobne ako v predošlých prípadoch do klasického OO návrhu. Jedným zo spôsobov, ako to môžeme urobiť, je transformovať ho po častiach – najskôr vytvoriť inštanciu jedného vzoru a následne druhého. Príklad priebehu takejto postupnej transformácie sa nachádza na obrázku 5-24.



Obrázok 5-24. Model s kompozíciou vzorov Decorator a Composite po tranformácii vzoru Decorator.

Ten predstavuje krok po vytvorení inštancie vzoru Decorator, ale ešte bez inštancie vzoru Composite. Obrázok 5-25 obsahuje výslednú kompozíciu inštancií vzorov.



Obrázok 5-25. PSM s kompozíciou vzorov Decorator a Composite.

5.4.3 Transformácie medzi modelmi

Dôležitou časťou, ktorú je potrebné riešiť, sú transformácie medzi modelmi. Ich úlohou je na základe špecifikácie definovanej v rámci vyššieho modelu vytvoriť korektné inštancie vzorov a prepojiť ich časti s ostatnými prvkami modelu, čím vznikne návrh na OO úrovni.

Úlohou transformácie nemá byť len vytvoriť "nejakú" inštanciu vzoru. Každý vzor je možné navrhnúť a implementovať viacerými korektnými spôsobmi, úlohou transformácie je aplikovať v danej situácii najvhodnejší variant vzoru a v takej podobe pripraviť jeho inštanciu. Ako príklad môžeme definovať rôzne varianty inštancií opísaných vzorov Composite a Decorator.

Composite

Najčastejšie opisované varianty vzoru na úrovni návrhu sa líšia v umiestnení metód pre pridanie alebo odstránenie prvkov hierarchie. Tieto metódy môžu byť umiestnené buď v rámci najvyššej triedy celej hierarchie (hráč roly Component) alebo len v rámci triedy slúžiacej na zoskupenie ostatných tried (hráč roly Composite) [14]. Ak sa metódy umiestnia do najvyššej triedy hierarchie, zjednotí sa síce celé jej rozhranie, ale na druhú stranu sa tieto metódy stanú bezvýznamné pre triedy, ktoré neobsahujú ďalšie podtriedy (listy). Určiť, ktorý zo spôsobov je vhodnejší, nie je jednoduché, pretože závisí od toho, akým spôsobom si vývojár želá pracovať so vzorom. Preto rozhodnutie o variante v prípade samostatnej inštancie vzoru Composite by malo byť definované používateľom, napríklad v konfigurácii transformácie.

Decorator

V prípade vzoru Decorator možno identifikovať alternatívy vo forme použitia či nepoužitia rozhrania zastrešujúceho konkrétne dekorátory. V tomto prípade sa môže javiť voľba alternatívy jednoduchšie: ak je definovaných viac dekorátorov, použiť toto rozhranie, v prípade jedného dekorátora rozhranie stráca význam, a preto ho môžeme vynechať.

Kompozícia vzorov Composite a Decorator

Pri vytváraní kompozície viacerých vzorov je nutné vybrať také alternatívy jednotlivých spájaných vzorov, aby bolo možné zabezpečiť ich vzájomnú spoluprácu čo možno najjednoduchšie. V rámci príkladu došlo k použitiu vzoru Decorator so samostatným rozhraním pre dekorátory, pretože inštancia obsahuje viac dekorátorov. Súčasne bol použitý variant vzoru Composite s metódami na pridávanie a odoberanie objektov len v rámci kompozitnej triedy, nakoľko pridanie metód do celej hierarchie by znamenalo pridanie týchto metód aj do tried definovaných vzorom Decorator, čo by zbytočne komplikovalo návrh.

Realizácia transformácií

Existuje viacero jazykov určených na transformáciu medzi modelmi. Jednou z alternatív je jazyk QVT (Query View Transformation), ktorý bol definovaný pre potreby MDA skupinou OMG. Jeho špecifikácia je však natoľko zložitá, že momentálne neexistuje implementácia, ktorá by úplne vyhovovala tomuto jazyku. Preto je vhodné prikloniť sa k niektorému z jazykov, ktoré sa svojou funkčnosťou blížia ku QVT, ale na rozdiel od neho už majú funkčnú implementáciu. Príkladom takého jazyka je ATL, ktorý bol vytvorený v rámci Eclipse Modeling Framework. V úvahu pripadajú aj jazyky definované v rámci existujúcich vývojových prostredí, akými sú napr. Borland Together alebo Rational Software Architect.

Nasledujúci príklad je ukážkou transformácie vzoru Singleton, ktorá pridáva triede statickú metódu určenú na vytvorenie inštancie.

```
rule SingletonClass {
     from s : UML2!uml::Class (
              if thisModule.inElements->includes(s) then
                 s->oclIsTypeOf(UML2!uml::Class) and
                 s->getAppliedStereotypes()->
                    exists(m|m.qualifiedName='MyProfile::Singleton')
              else false endif)

     to   t :  UML2!uml::Class (name <- s.name + 'Singleton'),
          op : UML2!uml::Operation (
                name <- 'getInstance',
                class <- t,
                visibility <- #public,
                isStatic <- true),
          p :  UML2!uml::Property (
                name <- 'instance',
                type <- t,
                visibility <- #private,
                isStatic <- true)
}
```

V príklade bol prezentovaný pevný spôsob transformácie, keď transformačné pravidlá priamo definujú, ako sa majú jednotlivé elementy modelu transformovať. Proces transformácie by však mohol byť riadený vybraným opisom štruktúry vzorov (napr. niektorým z opísaných v rámci kapitoly 5.2 Opisy návrhových vzorov), pomocou ktorého by sa vytvárali inštancie vzorov. V takom prípade by bolo možné zapojiť do procesu transformácie možnosť výberu vhodného variantu vzoru či vykonania verifikácie vytváraných inštancií, ktorej výsledkom by mohlo byť upozornenie vývojára na nekonzistencie návrhu.

5.4.4 *Prístup k tvorbe inštancií návrhových vzorov*

Podkapitola 5.1.1 opisuje dva odlišné procesy, ktoré je potrebné vykonať pri vytváraní korektnej inštancie vzoru: konkretizáciu a špecializáciu. Konkretizáciou sa do abstraktnej inštancie postupne pridávajú hráči všetkých rolí, ktoré je potrebné v rámci inštancie realizovať. Špecializácia znamená priblíženie vzoru zo všeobecného opisu do konkrétnej doménovej oblasti vyvíjaného softvéru (napr. korektné pomenovanie rolí). V rámci nášho prístupu sa snažíme zautomatizovať oba procesy v čo možno najväčšej miere. Viac priestoru sa objavuje v procese konkretizácie, nakoľko od používateľa dostaneme iba opis niektorých hráčov rolí, zvyšných hráčov doplní proces automatickej transformácie. V prípade špecifikácie inštancie vzoru sa veľká časť ponecháva na používateľa, ktorý definuje doménovo závislých hráčov rolí a súčasne špecifikuje napojenie vzoru do kontextu zvyšného softvéru. V tomto smere sú možnosti automatickej transformácie značne limitované, nakoľko je práve úlohou používateľa špecifikovať ako priblížiť inštanciu vzoru do doménovej oblasti.

5.4.5 *Zhodnotenie prístupu*

Cieľom prístupu je poukázať na možnosti modelovania návrhu softvéru na rôznych úrovniach abstrakcie s použitím návrhových vzorov. Vychádza pritom z iniciatívy MDA, ktorá je postavená na myšlienke používania modelov na rôznom stupni nezávislosti od použitej platformy. Prínosom takého spôsobu modelovania je možnosť pracovať, uvažovať a komunikovať na úrovni vzorov podľa ideí autorov, ktorí vzory definovali. S využitím MDA prístupu môžeme nielen modelovať na úrovní vzorov, ale naše modely transformovať do bežných OO modelov a následne do zdrojových kódov.

Podkapitola poukazuje na možnosti modelovania na vyšších úrovniach abstrakcie, no nepokúša sa riešiť túto komplexnú problematiku ako celok. Za jej primárny cieľ možno považovať identifikáciu problému a prezentácie príkladov formy riešenia, pomocou ktorej by mal byť podobný problém eliminovaný. Pri vytváraní príkladov sa vynárali ďalšie otázky, na ktoré bude potrebné v budúcnosti zodpovedať. Možno ich zhromaždiť do už spomínaných dvoch skupín: otázky ohľadom špecifikácie spôsobu modelovania práce so vzormi a otázky ohľadom transformácií medzi modelmi.

5.5 Zhodnotenie

Kapitola sa zaoberá problematikou modelovania návrhových vzorov a možnosťami práce s návrhovými vzormi na úrovni modelov. Prezentuje opisy problematiky z viacerých pohľadov, čím poskytuje komplexnejší nadhľad na aktuálny stav poznania.

Prvý z opisovaných pohľadov je opis životného cyklu inštancie návrhového vzoru pozostávajúci z vytvorenia a evolúcie inštancie. Vytváranie inštancie tvoria dve navzájom nezávislé činnosti. Často si to neuvedomuje, nakoľko ich obe vykonávame intuitívne, bez toho aby sme sa nad tým dôslednejšie zamýšľali. Prvou je postupná dekompozícia vzoru, keď od prvotnej myšlienky použitia vzoru postupne pripájame do vznikajúceho riešenia hráčov jednotlivých rolí, až pokiaľ nevytvoríme kompletnú OO štruktúru inštancie. Druhou činnosťou, ktorú musíme vykonať, je priblíženie všeobecného opisu vzoru do doménovej oblasti. Keď sa na daný stav pozrieme z pohľadu možností automatizácie procesov, možno vidieť potenciál v podpore dekompozície vzoru, než doménového priblíženia.

Druhým nemenej dôležitým opisovaným pohľadom je problematika modelovania vnútornej štruktúry vzorov nazývanej tiež leitmotif. Dôkladný formálny opis návrhových vzorov je nutnou podmienkou, ak sa chceme zaoberať automatizáciu procesov, akými sú napr. vytváranie inštancií vzorov, ich verifikovanie či identifikovanie v už exitujúcich riešeniach. Problém sa snaží riešiť viacero prístupov, no žiaden z nich nezískal status všeobecne uznávaného štandardu. Z tohto dôvodu predkladaná práca obsahuje opisy rôznych prístupov, ktoré medzi sebou porovnáva a zachytáva ich potenciál v súvislosti s ich ďalším využitím.

Možnosti spolupráce návrhových vzorov predstavujú ďalší opisovaný pohľad. Ku kompozícii vzorov možno pristupovať intuitívne alebo identifikovať pravidlá, podľa ktorých sa dá vykonávať systematicky. Ak chceme uvažovať o automatizácii procesov, je potrebné sa zamerať na systematický prístup. Opísané prístupy POAD a Hybridizácie vzorov predstavujú dva odlišné pohľady na danú problematiku.

Posledný pohľad, ktorý kapitola zachytáva, predstavuje vízia možností modelovania OO návrhu na úrovni návrhových vzorov. Ide o prístup, ktorým by sa zjednodušil návrh softvéru s využitím vzorov. Medzi jeho hlavné prínosy patrí:

- možnosť vytvárať návrh na vyššej úrovni abstrakcie, ktorú vzory poskytujú,
- zachovanie informácií o použitých vzoroch v návrhu. Tým sa okrem iného zjednodušuje pochopiteľnosť návrhu ako celku a zvyšuje sa kvalita dokumentácie riešenia. Vďaka tomu klesá riziko hrozby poškodenia návrhu chybnými zásahmi vykonávanými počas údržby a navyše možno uvažovať o automatizácii realizovania evolučných zmien v inštanciách vzorov.

Ako vyplynulo z mnohých prác, stále zostáva veľa nevyjasnených otázok, ktorých zodpovedaním by sa zvýšil potenciál využitia návrhových vzorov ako vhodných praxou overených riešení. V podkapitole 5.4 Modelovanie s návrhovými vzormi na vyššej úrovni abstrakcie bola prezentovaná vízia, ktorá by zefektívnila proces modelovania návrhu s použitím vzorov. Na to, aby bolo možné presunúť túto víziu bližšie k realite, je potrebné nájsť riešenia týchto problémov:

- Určiť vhodný spôsob ako modelovať softvér tak, aby sa v jednom modeli zlúčili informácie o nasadených inštanciách vzorov (v podobe hráčov rolí danej inštancie spolu) s ostatnými elementmi návrhu, ktoré nie sú súčasťou žiadnej inštancie. To znamená presne zadefinovať metamodel, ktorý bude opisovať sémantické možnosti vytváraných modelov. Okrem toho je vhodné definovať vhodný spôsob grafického zachytenia modelu, ktorý by v konečnom dôsledku umožnil prehľadne zmiešať dve rôzne úrovne abstrakcie obsiahnuté v modeli.
- Vytvoriť vhodné transformácie, ktoré by presunuli elementy modelu z úrovne abstrakcie vzorov na vhodné elementy modelu OO úrovne. Vzhľadom na možnosti variability vzorov a ich inštancií možno predpokladať, že nebude možné vytvoriť jednoduché transformačné pravidlá z jedného modelu do druhého. Riešením by mohol byť znalostný spôsob realizácie transformácií, ktorý by sa opieral o bázu znalostí o vzoroch, ich variabilite a možnostiach vzájomných kompozícií. Tu sa javí vhodné použiť a vhodne rozšíriť niektorý zo zápisov vnútornej štruktúry vzorov prezentovaných v kapitole 5.2 Opisy návrhových vzorov.

Použitá literatúra

[1] Alexander, C., et al.: *A pattern language. Towns, buildings, construction.* Oxford University Press, New York, USA, ISBN 0-19-501919-9, 1977.

[2] Clifton, M.: Advanced Unit Test, Part V – Unit Test Patterns. Code Project 2004. `http://www.codeproject.com/gen/design/autp5.asp`, 16.5.2007.

[3] Coplien, J. O.: Software Patterns, SIGS Management Briefings. *SIGS Books*, New York, 1996.

[4] Dietrich, J., Elgar, C.: A formal description of design patterns using OWL. In *Proceedings of Australian Software Engineering Conference 2005*, pp. 243-250, 2005.

[5] Dong, J., Yang, S., Zhang, K.: A Model Transformation Approach for Design Pattern Evolutions. In *Proceedings of 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*. ISBN 0-7695-2546-6, 2006.

[6] Dong, J.: Design Component Contracts: Modeling and Analysis of Pattern-Based Composition. *PhD Thesis*, 2002, University of Waterloo, Canada, 2002.

[7] Dong, J.: UML Extensions for Design Pattern Compositions. In *Journal of Object Technology*, Vol. 1, No. 5, 2002, pp. 151-163, 2002.

[8] Eden, A. H.: Precise Specification of Design Patterns and Tool Support in Their Application. *PhD Thesis*, 1999, University of Tel Aviv, Izrael, 1999.

[9] Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[10] France, R. B., Kim, D. K., Ghosh, S., Song, E.: A UML-Based Metamodeling Language to Specify Design Patterns. In *Proceedings of Workshop on Software Model Engineering (WiSME) with UML 2003*, San Francisco, 2003.

[11] France, R. B., Kim, D. K., Ghosh, S., Song, E.: A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, Vol. 30, No. 3, pp. 193-206, 2004.

[12] Gamma, E., Helm, R., Johnson, R., Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley professional computing series, 1995.

[13] Hohpe, G., Woolf, B.: Enterprise Integration Patterns: *Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, ISBN 0321200683, 2003.

[14] Jakubík, J.: Izolácia všeobecných častí vzoru Composite. In *Objekty 2005, Sborník príspevku desátého ročníku konference*, Václav Snášel (ed.), Vydala Fakulta elektrotechniky a informatiky, VŠB – Technická unverzita Ostrava, 2005, pp. 74 – 84, 2005.

[15] Kardell, M.: A Classification of Object-Oriented Design Patterns. *Master Thesis*, Department of Computing Science, Umeå University, Sweden.

[16] Kudělka, M., Lehečka, O., Snášel, V.: Klasifikace vzorů v širších souvislostech, *Zborník konferencie ITAT 2006*, Košice, 2006.

[17] Larsen, G.: Designing Component-Based Frameworks using Patterns in the UML. In *Communications of the ACM*, Vol. 42, No. 10., pp. 38-45, 1999.

[18] Majtás, Ľ.: Catalogue of Design Patterns. In *Proceedings of the Seventh Joint Conference on Knowledge-Based Software Engineering*, Tallin, Estonia, 2006, pp. 139-142, IOS Press, 2006.

[19] Majtás, Ľ.: Design Pattern Composition from the Perspective of Roles. In *IIT Student Research Conference 2007*, Mária Bieliková (Ed.), Slovak University of Technology, Bratislava, 2007, pp. 59-66, ISBN 978-80-227-2631-3, 2007.

[20] Majtás, Ľ.: Určenie vhodného spôsobu ukladania modelov návrhových vzorov. In *Objekty 2006, Sborník příspěvků jedenáctého ročníka konference*, Česká zemědělská univerzita, Praha, 2006, ISBN 80-213-1568-7, 2006.

[21] Mak, J. K. H., Choy, C. S. T., Lun, D. P. K.: Precise Modeling of Design Patterns in UML. In *Proceedings of 26th International Conference on Software Engineering*, 2004, IEEE Computer Society, Washington, USA , pp. 252-261, ISSN 0270-5257, 2004.

[22] Mapelsden, D., Hosking, J. and Grundy, J.: Design Pattern Modelling and Instantiation using DPML. In *Proceeding of TOOLS Pacific 2002*, Sydney, Australia. Conferences in Research and Practice in Information Technology, ACS, pp. 3-11, ISBN 0-909925-88-7, 2002.

[23] Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group, 2003, `http://www.omg.org/docs/ptc/04-10-15.pdf` (8.8.2007).

[24] Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, 1999.

[25] Model Driven Architecture White Paper. Object Management Group, 2000, `http://www.omg.org/docs/omg/00-11-05.pdf` (8.8.2007).

[26] Object Constraint Language. Object Management Group, 2006, `http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf` (8.8.2007).

[27] OWL Web Ontology Language Overview, W3C, 2004, `http://www.w3.org/TR/owl-features` (8.8.2007).

[28] Ram, J., Reddy, J. K., Rajasree M. S.: Pattern Hybridization: Breeding New Designs Out of Pattern Interactions. In *ACM SIGSOFT Soft. Eng. Notes*, Vol. 29, No. 3, pp. 1-10, 2004.

[29] RDF/XML Syntax Specification. W3C, 2004, `http://www.w3.org/TR/rdf-syntax-grammar` (8.8.2007).

[30] Riehle, D.: Composite Design Patterns. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'97*, ACM Press, Atlanta, pp. 218-228, 1997.

[31] Smolárová, M., Návrat, P., Bieliková, M.: A Technique for Modelling Design Patterns. In *Knowledge-Based Software Engineering - JCKBSE'98*, Smolenice, pp. 89-97, IOS Press, 1998.

[32] Smolárová, M., Návrat, P.: Pattern-Supported Software Development: Rôle of Abstraction and Generality. *Technical Report*, Slovak University of technology in Bratislava, 2000.

[33] Unified Modeling Language. Object Management Group, 2003, `http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf` (8.8.2007).

[34] Venners, B.: How to Use Design Patterns, A Conversation with Erich Gamma, `http://www.artima.com/lejava/articles/gammadp.html` (27.2.2007).

[35] Vlissides, J.: Pluggable Factory Part II, *C++ Report*, Feb., 1999, pp. 51-57, 1999.

[36] XML Metadata Interchange. Object Management Group, 2005, `http://www.omg.org/cgi-bin/doc?formal/2005-09-01` (8.8.2007).

[37] XSB. The XSB Logic Programming System, Version 2.1. 1999, `http://xsb.sourceforge.net` (8.8.2007).

[38] Yacoub, S. M., Ammar, H. H.: Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems. Addison-Wesley, ISBN 0-201-77640-5, 2003.

[39] Yacoub, S. M., Ammar, H. H.: Pattern-Oriented Analysis and Design (POAD): A Structural Composition Approach to Glue Design Patterns. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00), IEEE Computer Society*, 2000.

6
SEMANTIC WEB SERVICES

Peter Bartalos

From the beginning of Web existence a huge evolution of it is observed. We see this from different points of view. There is a progress of the Internet technologies which are the bases for making the Web working. The research in communication and data store brought new approaches to transport and collecting huge amount of data. This caused that the Web is enlarging in volume of resources available and also in number of users. The Internet and thus also Web is now available for a big number of people from all over the world. It has extended into households and is increasingly used not only in business-to-business interaction. There are also new types of resources available on the Web. It has moved forward from a network of documents. Now it offers also functionality and operation performance. Altogether are called services. Here belong for example online e-shops, various reservation systems, e-banking applications and others.

Web possibilities allow developing applications where the basic building components are services, i.e. service oriented architecture (SOA) based applications. SOA allows creating applications from loosely-coupled, distributable, platform independent services. The prominent technology to implement these services is web services. Web services are software systems we can interact with through the Web (more detailed definition is presented later).

Web services are mainly used today in practice to integrate applications. Different systems are wrapped into a web service and this way other systems can benefit from their functionality not caring about the platform it is working on. However, this kind of utilization of web services is effective, they are intended to be used in more sophisticated way too.

There is a vision to use services for almost any kind of operation, not only for those which can be performed using applications preprogrammed for certain area and thus limited. The aim is to recognize the users' goal, solve the problem by using available services and to bring a solution. The intention is to exploit wide opportunities of the Web to automatically deal with various user's objectives. It is not clear which of the approaches are matured enough to do this.

One initiative which wants to bring a solution is the Semantic web services approach. The basic idea is to add additional meta-data to web services' description. This should allow working with them in automatic manner. This includes the discovery of web services to find the usable ones. Based on the available meta-data we are able to find more relevant

services for our goal. Although, a lot of services realizing different functions exist, it is not guaranteed that we find such one for solving our problem. This can have several reasons. One is that there are no available services for the problem domain. Another reason is that the given user's goal is very complex and no individual service can solve it. Here, Semantic web service composition tries to bring resolution. The aim is to combine several services and arrange them in such a way, that these services deal with certain parts of the problem, to create intermediate data and to bring together a solution.

The idea of Semantic web service composition is nice and there already exist methods and implemented tools to realize it. Despite of this, current solutions for automatic composition are unmature to be used in practice for variety of problems in different domains. These problems are also discussed in this document.

The aim of our work is to deal with Semantic web service composition problem. We want to examine how a user can help when solving them. There already exist approaches including the user in this process. Our focus is to extend these approaches and investigate the benefit of user collaboration to define its goal, compose and execute the service. This document presents an overview of the interested area. This includes primary the Semantic web service composition and related topics.

6.1 Towards Semantic Web Services

6.1.1 Service Oriented Architecture

Service oriented architecture (SOA) is an architectural design pattern relying on service-orientation as its fundamental design principle [31, 32, 33]. There exist numbers of domains where systems benefit from the advantages of SOA. In [27] the authors describe an application and explains how SOA helped to meet the business needs of a credit card issuer with particular challenges. In [25] an framework for developing a component-based distributed simulation and executing the simulation in service-oriented architecture on the GRID is proposed.

One of the most characteristic advantages of SOA is the principle of loosely coupling between the service provider and a consumer. SOA allows building systems from components providing their encapsulated functionality in services to each other and have minimal dependencies between themselves.

Services can be divided into two categories: stateless and stateful. Stateless services do not store state information between requests. This makes the service more reliable because the recovery from partial failure is relatively easy. Stateful services require both the customer and the provider to share the consumer-specific context. This reduces the overall scalability of the services and increases the coupling between the consumer and provider.

Figure 6-1 presents the stack and the elements of SOA. The stack is divided into two parts. First, it contains the functional aspect and second, the quality of services aspect. The functional aspect includes the *transport* component responsible for transferring the messages between the service consumer and the requester. The *service communication protocol* is an agreed mechanism for interaction. *Service description* is used to define what the service is and how it should be used. The *service* describes an actual service which is made available for use. *Business process* is a collection of services invoked in a particular sequence to meet the business requirement. The *service registry* is a place where the providers can publish

their services and consumers can find the available ones. The quality of service part of the SOA stack includes the following elements. Policy is a set of rules under which the service provider makes the service available to consumers. *Security* is the set of rules that might be applied to the identification, authorization, and access control of service consumers invoking services. *Transaction* is the set of attributes that might be applied to a group of services to deliver a consistent result. *Management* is the set of attributes that might be applied to managing the services provided or consumed.

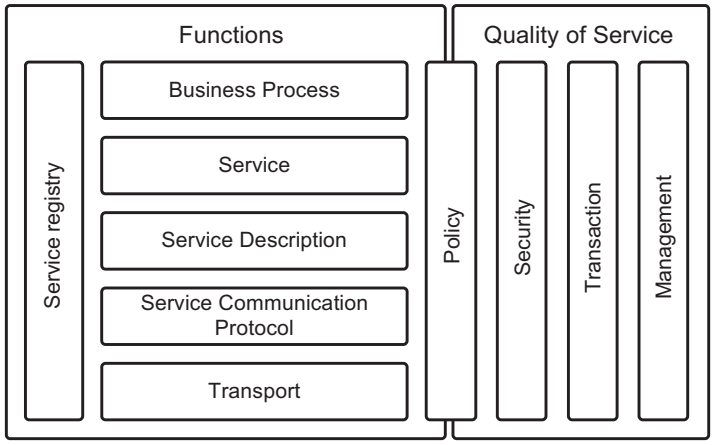


Figure 6-1. SOA stack (retrieved from [31]).

The main advantages of SOA can be summarized as follows:

- SOA provides loosely coupled services which can be thus very easily distributed. This contributes to its effective usage in distributed, grid computing [18].
- SOA provides an effective way for software application integration.
- SOA uses platform independent building elements contributing to higher flexibility of the software application.

Despite of that SOA offers several advantages there are cases when it is not suitable. First, if the application is already built, its modification to SOA requires redesigning and reimplementation. This means that there will be additional costs. The access to certain functionalities has also disadvantages. If some modules are closely related and they communicate extensively, the use of web services may be an unreasonable trade off. Web services are not suitable also when the component is highly bound to a user interface. In each situation concrete circumstances and requirements need to be analyzed.

6.1.2 Web Services

The term web service [3, 24] is nowadays a very often used word, although not always in the same meaning. Even though there exist a lot of definitions of web services, there exist discussions which attributes must the service have [41]. For us, the most appropriate definition is the one, given by W3C:

> "A Web service is a software system identified by a *URI*, whose public interfaces and bindings are *defined* and *described* using XML. Its definition can be *discovered* by other software systems. These systems may then *interact* with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols."

Web services are one of the implementations of SOA. They play an important role in B2B (business-to-business) integration. It is assumed that the companies will expose the functionality they offer as a service. It means that they will publish an interface via which this service can be invoked. The invocation is done by a program. Thus, requesting and executing a service involves a program calling another program. There occur confusions because there are a lot of services available through the Web which are not web services. An example is a fly ticket booking system. A web service is a software application with a published a stable programming interface, not a set of Web pages.

Web services are used to make some utility available through the Web. They wrap a functionality of an internal system and make it available for other applications, see Figure 6-2. Web services are developed as loosely coupled application components using any programming language, any protocol, and any platform.

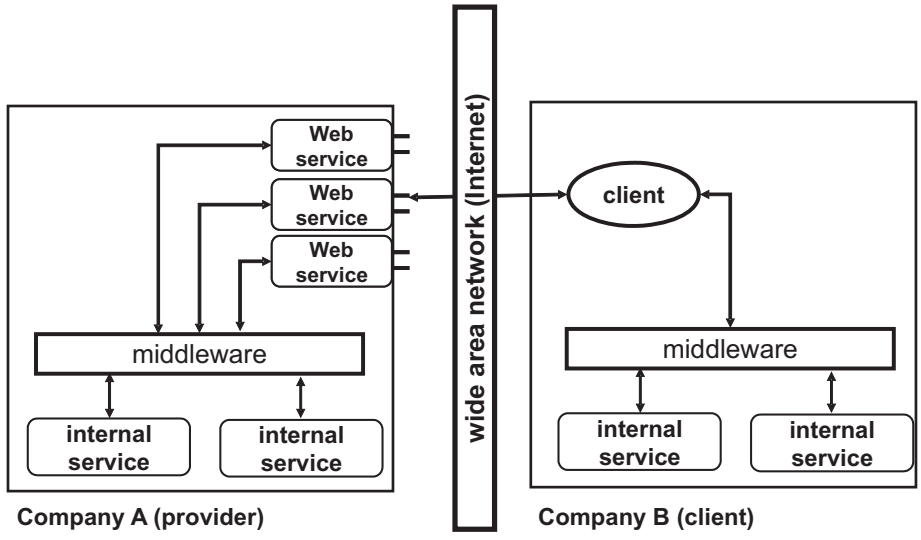


Figure 6-2. Web services provide an entry point for accessing local services (retrieved from [3]).

The web service framework is divided into three areas [28]:

- the simple object access protocol (SOAP, `www.w3.org/TR/soap/`) which enables communication among Web services,
- Web Services Description Language (WSDL, `www.w3.org/TR/wsdl.html`), which provides a formal, computer-readable description of Web services,
- Universal Description, Discovery, and Integration (UDDI, `www.uddi.org`) directory, which is a registry of Web services descriptions.

SOAP

SOAP [52] is a protocol for exchanging XML-based messages over computer networks. It normally uses existing transport protocols HTTP/HTTPS. SOAP is fundamentally stateless, one-way messaging, but by combining the one-way exchanges, more complex communication patterns can be created. A core SOAP has a very simple structure. It is an XML element with two child elements for the header and the body. One of the most common messaging patterns in SOAP is the RPC (Remote Procedure Call). In this pattern the client sends a request message to the server which sends immediately the response.

WSDL

SOAP offers a basic communication pattern for web services. To be able to successfully interact with a service, we need to know which messages must be exchanged. For this purpose we use WSDL, an XML based language for description of the interfaces of Web services.

UDDI

UDDI is a way how to find a web service. It is a centralized registry of web services. It has three components:

- *White pages*: name and contact details.
- *Yellow pages*: categorization based on standard taxonomies.
- *Green pages*: technical data about services.

UDDI can be accessed using SOAP API. Using it, one can perform querying and updating of the registry. In Figure 6-3, you see how the SOAP, WSDL and UDDI work together to provide means for using web services.

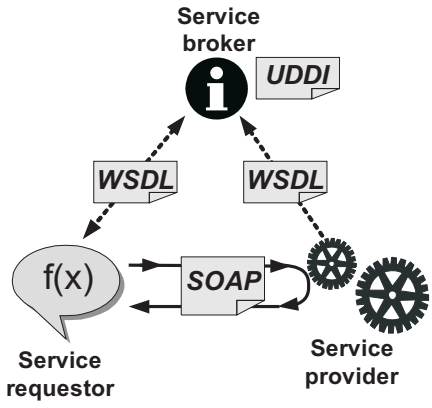


Figure 6-3. Web service standards' collaboration.

6.1.3 Semantic Web Services

The Web is primarily designed for use by humans. Nevertheless, there is an effort to automate its use and bring the Web more accessible for machines. This has brought forward the need for machine processable representations of semantically rich information: a vision at the heart of the Semantic web [17]. Nowadays, approaches based on giving additional information – semantics, to the content of the Web are researched. The semantic description focuses both on the content information and services available on the Web. In the case of Web services having meta-data about their semantic, we talk about Semantic web services [22, 23, 29, 55, 66, 74]. Semantic web services are the result of the evolution of the syntactic definition of web services and the Semantic web [23]. There exist approaches for the design and development of applications employing Semantic web services [20]. In [76], a MusicBrainz project is described, which aims to be the first using semantic web services. It's a large database of music meta-data which is accessible via a web service. Based on these meta-data it is easier to search the music repository and find a relevant song. The meta-data are described using RDF (Resource Description Framework).

Semantic web service can be considered as a result of the Web evolution (see Figure 6-4). At the beginning, the Web was only a collection of documents, i.e. it offered static content. This has evolved to an environment offering also different functionalities through services, i.e. dynamic elements were appended. The popularity of the Web led to its enormous expansion of documents and services available. This has been causing problems when searching for a relevant information or service.

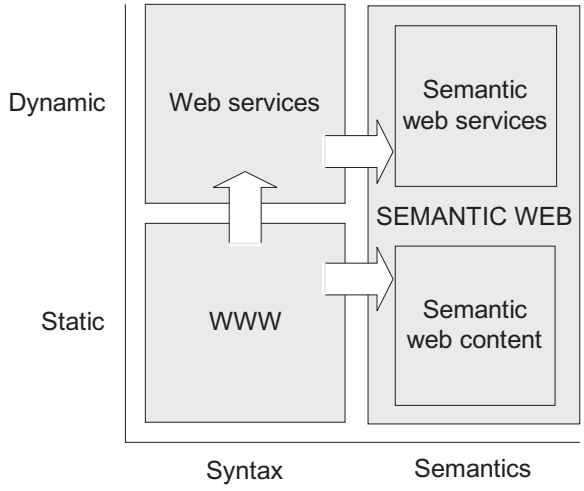


Figure 6-4. Web evolution to Semantic web services.

There is also change in the aspect of the Web user. Not only humans are considered as the users of the Web. There are initiatives which aim is to bring the Web accessible for machines – software agents. These can use methods which are more resistant to the problem of Web expansion than humans are. The performance of the software agents is limited due that the Web was initially intended for human interaction only. The main problems are weak structuring and hard machine interpretation of the Web resources. The initiative which tries

to bring solution to this problem is a Semantic web. Its aim is to ad additional semantic meta-data to the resources available on the Web, i.e. it moves from pure syntactic level to richer semantics level. This means that the Web resources are bound to meta-data describing the meaning of the data. Hence, we are moving to the Semantic web including Semantic web content and Semantic web services.

6.1.4 *Developing Semantic Web Portals: without SOA versus SOA Based*

The Web is a place where different information and services are available. There are several cases when these are focused on a certain community of people with close interests. For these groups different Web portals are suitable providing an entry point for accessing information and services.

The Semantic web technologies offer means to support searching, organizing, processing, and presenting information. These can be applied in the context of Web portals resulting in Semantic web portals [48]. Semantic web portals are web portals collecting information and functionalities for a community and are based on Semantic web technologies.

Semantic web portals can be built using different software development approaches. In the next, we discuss the differences between building Semantic web portals without SOA and based on SOA. We use a job offer portal (JOP) developed in the context of research project NAZOU[1] [61] as an example to show the differences. First we describe the way this portal was developed. Then we concentrate on how SOA can change this.

JOP offers its users several ways of navigation through the information space of job offers using different presentation tools working with the ontological database produced by a chain of data harvesting tools acquiring and processing data from the Internet. The main requirements to JOP were:

- Adaptivity and adaptability of the system's presentation and functionality.
- Built-in automatic user modeling based on user action logging with semantics and automatic user characteristic estimation.
- Reusability and generic design suitable for multiple application domains.
- Extensibility with additional tools for specific tasks and overall flexibility with respect to tool orchestration.
- Tolerance towards changes in the domain and user ontologies.

JOP [7, 8] was developed using an MVC based framework built on the pipes and filters architectural pattern – Cocoon[2]. We have used the available portal block provided by Cocoon to build JOP. It offered us a basic infrastructure which was configured and into which our modules were put into. Figure 6-5 depicts an overview of the portal architecture that extends the basic functionality of Cocoon with additional software components in order to fulfill the aforementioned requirements.

[1] Project NAZOU, `http://nazou.fiit.stuba.sk/`

[2] `http://cocoon.apache.org/`

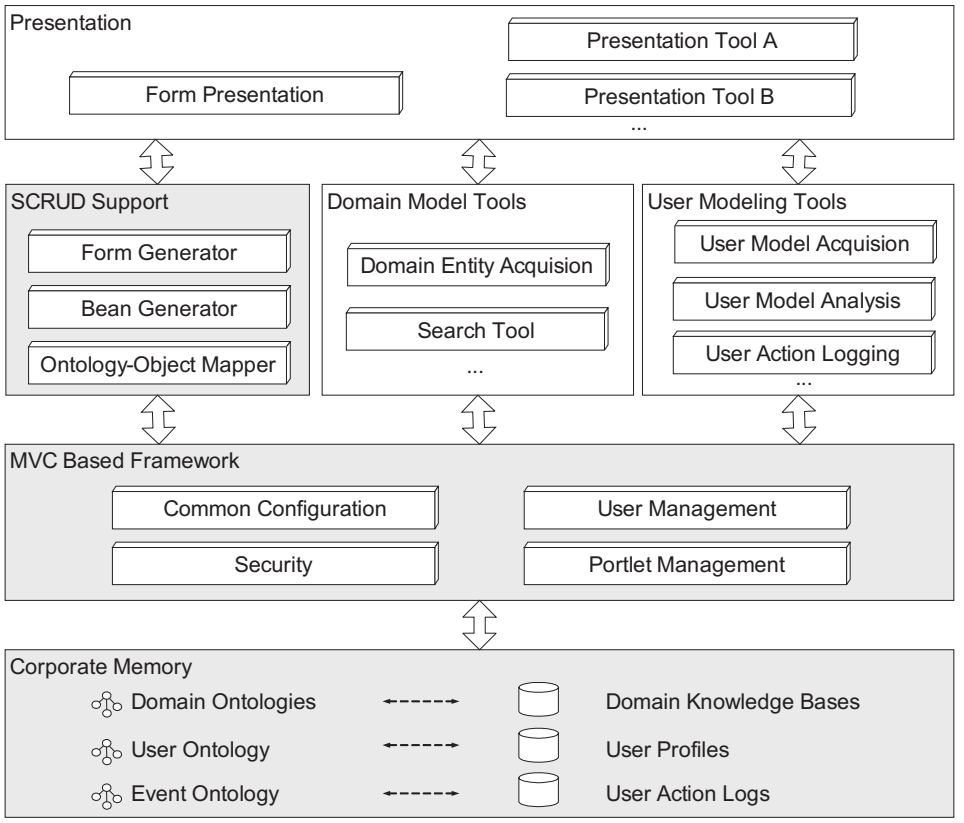


Figure 6-5. JOP architecture overview.

At the bottom of the architecture the corporate memory is placed which stores the domain, user and event ontologies. As storage, the Sesame[3] ontological repository is used. To access it, the offered sesame API is used by modules which need to work with it. The second layer includes the common modules for configuration, security, user management and portlet management. These modules are provided by the portal block of the Cocoon framework. The next layer contains different functional modules. First, the SCRUD support component, which performs the persistent operations over the domain data entities [11, 13]. Second, the domain model tools with various functionality required in the particular domain. Third, the user modeling tools for the acquisition, analysis of the user model. The top layer contains the presentation tools.

The first two layers of the architecture together with the SCRUD support are domain independent and can be used in different Semantic web portals (highlighted parts in the figure). The rest components include tools tailored or configured to a given job offer domain. This includes for example tool *Factic*, *SemanticLog*, and *JDBSearch*. Factic is a personalized semantic faceted browser. Its aim is to navigate the user in large information space to find relevant information. The user can restrict the information space using facets to search

[3] `http://www.openrdf.org/`

for such information he is interested in. SemanticLog is used for logging different events occurring during user interaction with JOP. JDBSearch is a full text indexer and allows querying to search in the information space. Each of these tools is integrated in the Cocoon and thus deployed on the server where Cocoon is running (see Figure 6-6). This means that they are centralized into one execution point.

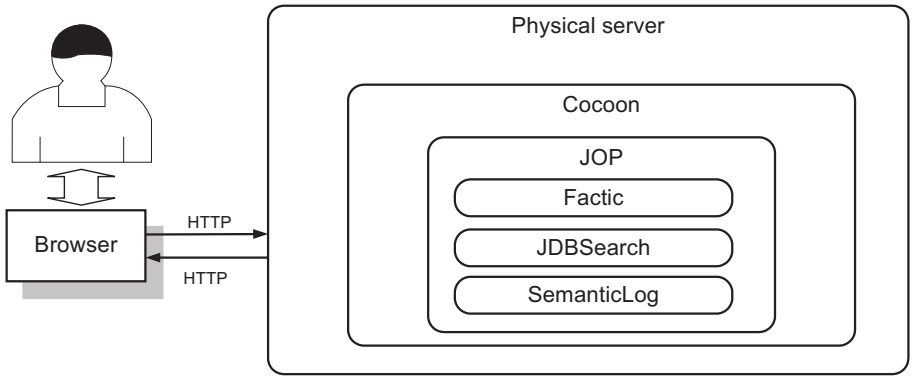


Figure 6-6. Classic job offer portal.

If JOP is build using SOA concepts, web services wrap application parts of various granularity. It is not enough to just transform some of them to web services. If an application uses web services it still does not mean that it is SOA based. Also other concepts must be applied. This includes for example the use of enterprise service bus (provides a messaging bus and content-based routing between service providers and consumers) and distribution of the services. Those parts of JOP which can be built as web services can be whole modules, tools or only some methods offered by tools. For example each module in the boxes from Figure 6-5 can be transformed into a web service. We have mentioned that JOP is built using the pipes and filters architectural patterns. Here, we have a chain of processing elements where the output of an element is the input of the next. Each element performs some processing of the input data resulting in output data. When moving to SOA, each of these elements should be realized as web service.

The benefits of web services include the possibility of use the wrapped elements not only from the user interface of JOP but also programmatically by other applications. For example in the case of Factic, we can wrap the method which cut of the information space based on given facets. Similarly, JDBSearch searching method can be used via web service call. Transforming the logging method of SemanticLog into web service brings the possibility of calling it also from other systems. This way, user activities not only from JOP can be gathered. Also other systems the user is working with can effectively log the events. After this, by analyzing the logs, we can obtain much more information about the user than if only those activities realized in JOP are captured. This results in richer user model and better personalization. If the mentioned methods are wrapped as web services, their execution does not need to be performed on the same physical server the JOP is running on (see Figure 6-7). Hence, distribution of the execution may take place. This is required when higher computation power is in demand.

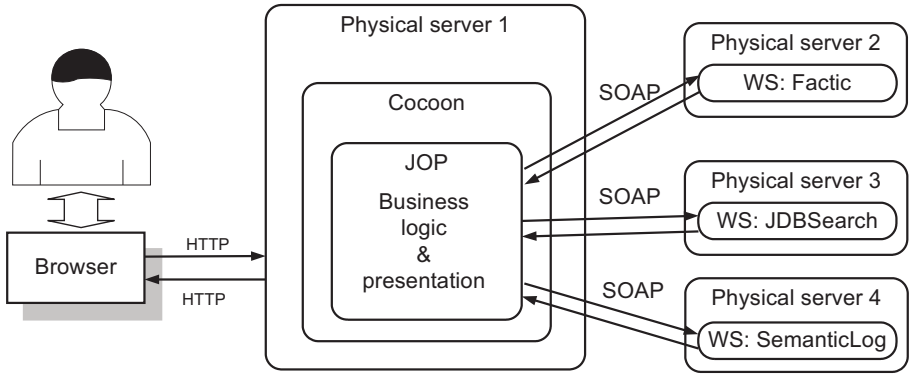


Figure 6-7. SOA based job offer portal.

SOA based JOP has also weaknesses. One of them may be the transformation of the Factic tool into web service. This causes problems from the point of view that in Factic the definition of the facets is realized by a user in the graphical user interface and there is intensive interaction with him. Other cases are components which are intensively called, for example SemanticLog. It is overextended if several thousands of users work with the system. The realization of the logging as web service is convenient but it may slow down the application. However, this depends on the concrete implementation and service distribution over physical servers.

SOA is not universally the best way how any system can be built. We presented JOP, a Semantic web portal for job offer domain to discuss different aspects of SOA. From this it arises that SOA brings benefits but its usage has also disadvantages in some situation. These positive and negative aspects must be taken into account before we decide to develop application based on SOA.

6.2 Semantic Web Services Description

The vision of Semantic web services brought a need of additional meta-data to the description of WSDL based services. This meta-data must contain enough information for understanding what the service does and how it should be used. All of this must be described in such a way that machines can process it.

There exist several formalisms for Semantic web service description [77]. They annotate elements of services with terms from domain models, including industry standards, vocabularies, taxonomies, and ontologies. The most know approaches are WSDL-S [1], OWL-S [63] and WSMO [70]. Each of these allows adding semantic information to services to be able to perform effective discovery, composition, execution, and monitoring. They differ in the complexity and expressivity of their construction elements. Nevertheless, there are some similarities [65]. Even though existing approaches are a step forward in semantic annotation of web services, [47] argues that they are insufficient for the efficient usage in SOA. The problems occur mainly when composition of web service is required.

Web service description is performed manually or using (semi)automatic approaches. In [37] a tool for annotating Semantic web service is presented. This tool provides a user interface for manual annotation, as well as machine-learning assistance for semi-automatic

annotation. The tool uses WSDL for description but offers also the possibility to export them to OWL-S.

6.2.1 WSDL-S

WSDL-S [1] is one of the possible formal descriptions of services. It defines the meaning of the inputs, outputs, preconditions and effects of the operations described in a service interface.

WSDL-S does not define a language that spans across the different levels of the web service stack, rather it is limited to WSDL simplifying the task of providing semantic representation of services. WSDL-S makes a low commitment to OWL and it provides a general annotation mechanism which could be used in conjunction with any meta language including UML. This allows to annotate web services with concepts from multiple ontologies from different sources.

6.2.2 OWL-S

OWL-S (also known as DAML-S) [63] presents an OWL [54] ontology of services motivated by the need of providing knowledge about a service. Figure 6-8 shows the core of the OWL-S. The base is the Service class which has three main properties:

- *presents*, which range is a *ServiceProfile*: class defining what does the service require from the consumers and what it offers them.
- *describedBy*, which range is a *ServiceModel*: class defining how the service works.
- *supports*, which range is a *ServiceGrounding*: class defining how is the service used.

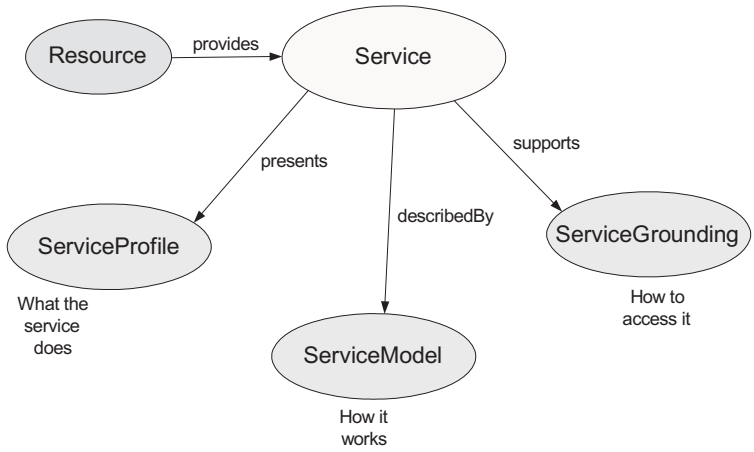


Figure 6-8. OWL-S core (retrieved from [63]).

Each instance of a Service class present a descendant class of a *ServiceProfile*, be *describedBy* a descendant class of *ServiceModel*, and support a descendant class of *ServiceGrounding*.

The service profile contains information usable by a seeking agent to search and find a service which meets its needs. The profile can be used also by a service seeking agent to express its needs, so the match-maker has a convenient dual-purpose representation.

The service model describes what happens when the service is carried out. This can be used for deeper analysis of whether the service meets the requirements. Other usage is in the situation when the service is a part of a composed service. Model is also used to monitor the execution of the service.

The service grounding specifies the details how an agent can access the service. It specifies the communication protocol, message formats, serialization techniques, and other service specific details.

In [53] other details about OWL-S are presented. The paper discusses the benefits of the richer service description supported by OWL-S. It describes how OWL-S is used in context of other standards, such as WSDL, UDDI and BPEL (Business Process Execution Language).

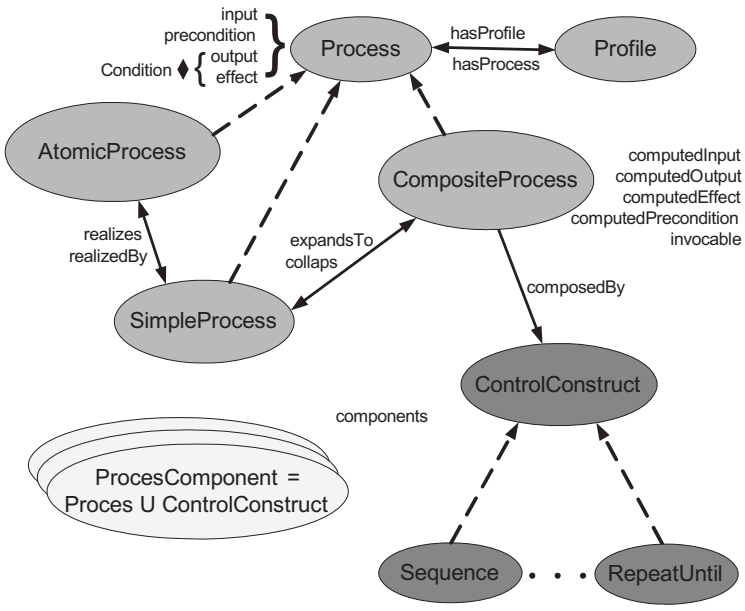


Figure 6-9. Top level of the OWL-S process ontology (retrieved from [63]).

OWL-S defines a subclass of the ServiceModel called ProcessModel. It is used to describe the service as a process. The process ontology adopts two views of processes. First, a process transforms input data into output data. Inputs specify the information required for the execution of the process. The outputs specify the information which is produced by the execution of the process. Second, the process transforms a world's state from one into another, what is described by preconditions and effects. The process model identifies three types of processes, see Figure 6-9:

– *Atomic process*: a directly executable process. It has no sub-processes and executes in one step. The requester has no visibility into the service's execution. For each

atomic process, a grounding that enables the requester to construct input and output messages must be provided.

- *Simple process*: is not executable and is not associated with grounding, but like atomic process, it is executed in one step. It can be used as an element of abstraction of an atomic process using the realizedBy property or as a simplified representation of a composite process using the expandsTo property.

- *Composite process*: is decomposable into other non-composite or composite processes. The decomposition is specified using the control constructs by a composedBy property: SEQUENCE, IF-THEN-ELSE, REPEAT-UNTIL and others. Each control construct is associated with a property called components to indicate the ordering and the conditional execution of the sub-processes.

An XML fragment of OWL-S ontology describing composite process of airline ticked booking is presented below[4].

```
<process:CompositeProcess rdf:ID="BravoAir_Process">
  <rdfs:label> This is the top level process for BravoAir</rdfs:label>
  <process:composedOf>
     <process:Sequence>
        <process:components rdf:parseType="Collection">
          <process:AtomicProcess rdf:about="#GetDesiredFlightDetails"/>
          <process:AtomicProcess rdf:about="#SelectAvailableFlight"/>
          <process:CompositeProcess rdf:about="#BookFlight"/>
        </process:components>
     </process:Sequence>
  </process:composedOf>
</process:CompositeProcess>
```

6.2.3 WSMO

WSMO [70] provides ontological specifications for the core elements of Semantic Web Services. It has its conceptual basis in the Web Service Modeling Framework (WSMF). WSMO is meant to be a meta-model for Semantic web service related aspects. It uses the MOF (Meta Object Facility) [62] to define such a model. The MOF metadata architecture defines four layers for models, see Figure 6-10:

- information layer is comprised of data we want to describe.

- model layer is comprised of the metadata that describes data in the information layer.

- the metamodel layer is comprised of the descriptions (i.e., meta-metadata) that define the structure and semantics of metadata.

- The meta-metamodel layer is comprised of the description of the structure and semantics of meta-metadata.

[4] Retrieved from `http://www.daml.org/services/owl-s/1.0/examples.html`

WSMO corresponds to the meta-model layer of the architecture. A WSMO description of a concrete web service is related to the model layer. The concrete web service is placed in the information layer.

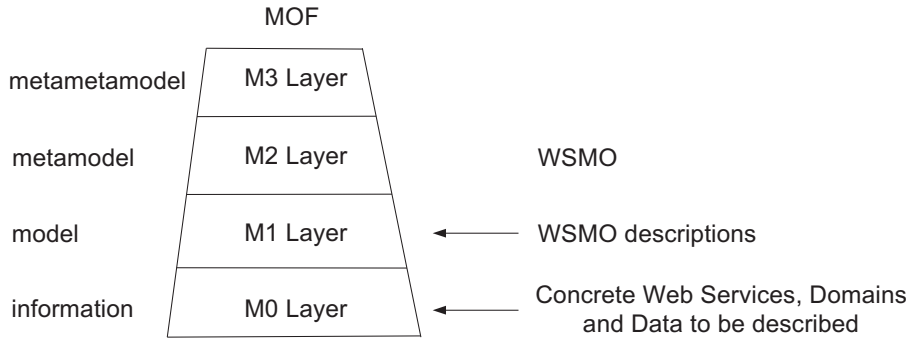


Figure 6-10. The relation between WSMO and MOF (retrieved from [70]).

WSMO identifies four top-level elements which define the semantics of the web service:

- *Ontologies*: provide a vocabulary to describe different aspects of web services.
- *Web services*: represent computational entities able to provide access to services. Web service is in WSMO described from three points of view: non-functional, functional and behavioral. The functionality is described by one and only one capability. The behavior may be described by multiple interfaces.
- *Goals*: describe aspects related to user desires with respect to the requested functionality. It is characterized in dual way. The goal's description includes the requested capability and requested interface.
- *Mediators*: describe elements that handle interoperability problems between different WSMO elements. They describe elements aiming at overcoming structural, semantic and conceptual mismatches between them.

To express the WSMO model, WSMO uses the WSML language [30].

6.3 Semantic Web Service Coordination

Semantic web service coordination aims at satisfying the users' goals using Semantic web service discovery, composition and execution [45]. Figure 6-11 presents an overview of the utilization of web service composition by a user. The user can be a human, an agent or an application invoking the service. The first step of this process is gathering the information about the goal. For this goal, taking the input data and considering preconditions, the composition process starts including web services discovery. The discovery is responsible for searching for the web services in the available registries. During composition the domain knowledge is exploited to enhance the used methods. The domain knowledge is stored in ontologies linked to the Semantic web services' descriptions. When the composition is constructed, it is executed. This results in output data and effects in which the user is interested in.

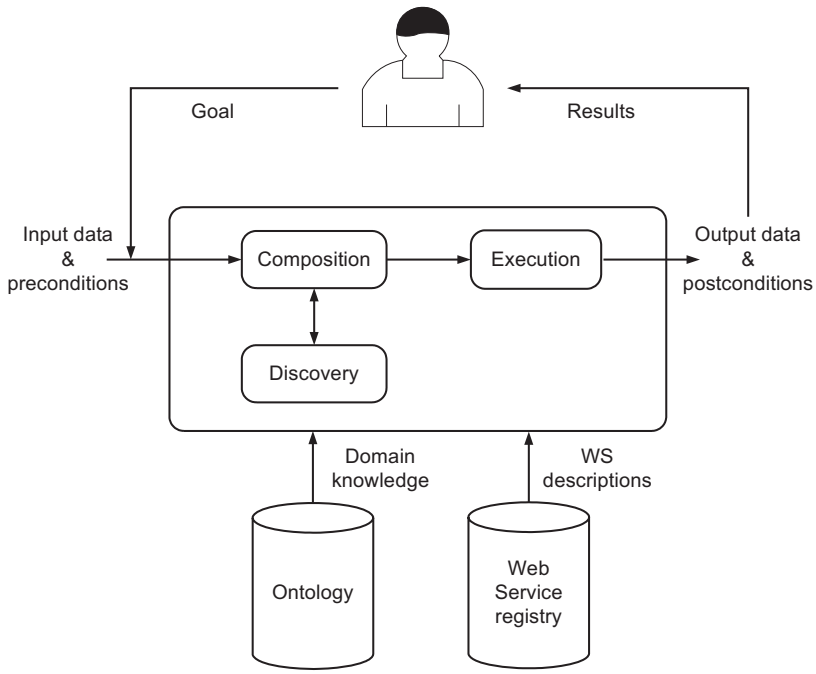


Figure 6-11. Web service composition utilization.

6.3.1 Semantic Web Service Discovery

To make the web service accessible for the users, service providers create web service registries. In these registries the available web service can be found. The initial intend of the UDDI specification was to work with a UBR (Universal Business Registry), a directory for all public web services. Later, the UDDI specification recognized the need of multiple registries and interactions among them. The existence of multiple registries brings new challenges in web service discovery. The problem is now first found the appropriate registry and then the suitable web service.

To discover the web service which matches the requirements, semantic based methods are used. In [78] a scalable, high performance environment for federated web service publication and discovery among multiple registries method is presented. It is called MWSDI (METEOR-S Web Services Discovery Infrastructure). It uses an ontology-based approach to organize registries, enabling semantic classification of all Web services based on domains. MWSDI makes accessible multiple registries provided by different registry operators.

Next approach based on semantic is presented in [2]. First, it describes an extension to the UDDI inquiry API specification to enable requesters to specify the required capabilities of a service. Second, it enhances the service discovery of UDDI by performing semantic matching and automatic service composition using planning algorithms. In [72] an algorithm for search in UDDI based on OWL-S is presented. OWL-S is used to semantically describe web service in terms of capabilities. Using logic inference offered and requested capabilities are matched to find the relevant web services. The paper includes also a review of other approaches to web service discovery.

Another approach to web service discovery is presented in [81]. The authors state that the existing web service definition languages have lack of the particular specification to the function of web services. From this reasons they define a *functional semantic* of web services. The functional description is expressed in a *functional ontology*. It helps the user to understand the function of the web service unambiguously. For semantic annotation of web service a mechanism presented by authors is used.

The results of the web service discovery is a set of services, which are assumed to be helpful. From this set we need to select the most suitable one. Most approaches for web service discovery offer as a result an ordered list of services based on their relevance. To select the most relevant web service we exploit any additional information describing also non-functional properties of web services. In [64] a method for evaluation of the semantic composability of Semantic web services is presented. This approach tries to completely utilize the semantic aspect and use ontological description of the services and their properties to measure the relevance.

The effective and successful web service discovery is one of the basic aspects of the Semantic web services vision. If we do not have methods to efficiently find required web services, the whole vision falls down. However, there are approaches to find relevant web services, the discovery problem is still not solved. The main problems are:

- *Insufficient description of services*: The web service descriptions are often not precise and complete. Hence, it is hard to determine the usefulness of the service.

- *Credibility of descriptions*: The web service descriptions offers information about the functionality, required input data, pre and post conditions, output data. Despite that these information may be relevant, there is no guarantee that the description absolutely corresponds with the reality. The descriptions may contain incorrect information. This causes problems when the execution of the service does not achieve the expected results. The detection of this presents a serious problem if no checking is available.

- *Computational strenuosity*: The sophisticated web service discovery approaches may require a lot of computation to find a set of candidate services and determine their relevance. The required resources do not have to be available in each situation.

In [5] the problems of web service discovery are discussed. The paper deals with the reasons why the brokerage aspect of the web service vision has proven so difficult to realize in practice. It also describes a pragmatic approach to web service brokerage based on automated indexing and discuss the required technological foundations. Ideas for improving the existing standards are presented too.

6.3.2 Semantic Web Service Composition

Although, a variety of different web services are available, there often occur situations when no service can satisfy our needs. This can be because there is no service which can fully achieve our goal, but only its part. Between the available services may exist such one, which is able to solve the rest of the problem and bring a solution.

One of the most appealing attribute of the web service is the ability to aggregate with other services and thus create a composite one, called also Web process. Taking this advantage, we

can organize more services together into one composite. This way we can offer more complex services realizing more complex workflows. These composite services are also called web processes. Our opinion is that this term may have much wider meaning, thus we do not use it. Each step of the workflow is realized by a service, which can by atomic either composite. Atomic services are those which can not be decomposed and are immediately executable. We denote the services from which the composite is built, i.e. the union of atomic and composite services as *sub-services*.

The process of arranging the structure of the complex service from other services is called composition of web services. Its aim is to create a scheme which describes how the sub-services are organized. More precisely, it unambiguously describes:

- when the sub-service will be called – what is the previous called sub-service (if it is not one of the first), i.e. control definition,
- which previously called sub-service will give the input data – to which next called sub-service will the output data be provided, i.e. data flow definition.

Each sub-service must be executable. One sub-service can be called more times during execution of the composite web service (the input data may be different), if required. The web service composition can be described using the same languages (mentioned in section 6.2), which contain constructs to do this. The composition scheme may contain controls like *sequence*, *repeat-until*, *decisions*. The individual step depicted in the composition presents the sub-services which will be called at the corresponding execution point. Figure 6-12 presents an example of a web service composition which goal is to create meteorological forecast maps. The problem is that we do not have a web service which does this. However, we have other services which can be combined to achieve the goal, because we have services for partial tasks. The whole composed process has three steps. First, the meteorological model from the gathered data is created. Second, the forecast is computed. Third, based on the forecast from the previous step, the meteorological maps are created.

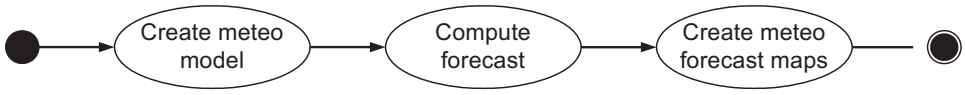


Figure 6-12. Web service composition example.

Web services are specified by preconditions, effects, input and output parameters. The precondition defines the propositions for the state of the world which has to be satisfied. Effects are the postconditions which will be true after successful execution of the web service. Input and output parameters define the input data for the web service and the data produced by the service. The goal we want to reach can be related to the output parameters, effects or both. In the example above, the goal is the data of meteorological forecast maps and we do not care about the effects. There are situations when our aim is not to get data, but to affect the world. These two cases divide the services to data-centric and state-centric. Although, there are also cases, when we are interested in resulting data and also state.

The aim of web service composition is to solve problems like the creation of the meteorological forecast maps described above. We have no service to achieve our goal, but we can compose it from sub-services. There are two ways in web service composition. In first, the composition is created manually, i.e. somebody creates it. The second way is the automatic composition of services. Here, the composition method automatically creates a composed service based on a defined goal. These approaches use AI (Artificial Intelligence) planning techniques [68].

The approaches based on AI planning convert the web service composition problem to AI planning problem [67]. Then several existing approaches can be used to solve it. Using these approaches we can create a plan which determines the action sequence leading from the initial state to the goal state. The creation of the plan does not solve the whole problem. The found sequence does not need to bring the desired results. We are successful only if we get the required output data by execution of this sequence. This means that also the data flow between the actions must be correct, i.e. the successive services are compatible on I/O data level and input data for each service has defined source. To summarize, we are looking for a composite service which execution in the initial state leads to the goal state and desired data.

The composers work with different formalisms to encode and solve the planning problem [58]. Here belong the pure backward chaining, any of the search space algorithms or the most trivial shortest path search. Real semantic web service composers usually combine and enhance these algorithms. Their aim is to reduce the search space to find a plan in acceptable time. In the following we take a brief overview of the approaches formalizing the web service composition problem or are used to find its solution.

State-Space Based Planning

The web service composition problem based on state-space planning is described by a five-tuple $\langle S, s_0, G, A, \Gamma \rangle$, where S is a set of the possible states of the world, s_0 is the initial state, G denotes the goal state which we want to reach, A is the set of actions the planner can use to change one state to other, and Γ is the set of translation relations $\Gamma \subseteq S \times A \times S$ defining the preconditions and effects for the execution of each action, it maps one state into another.

The goal of the planning is to find such a sequence of actions which when are applied to the initial state, will lead to the goal state. Formally, we look for a sequence $a_0, \ldots, a_n$, where $a_i \in A, \forall i = 0, \ldots, n$ which generates a state trajectory $s_0, \ldots, s_{n+1}$, where $s_i \in S, \forall i = 0, \ldots, n+1$ and $s_{n+1} = G$. In each step, there must exist $\gamma_i \in \Gamma, \gamma_i = (s_i, a_i, s_{i+1}), \forall i = 0, \ldots, n$.

The transition relations can by related to a cost function $c(a, s) > 0$. This function measures the cost of execution the action a in state s. The aim is to find the plan with the lowest cumulative cost computed from the costs of each state transition of the plan.

State-space based search can use any of the space search algorithms. The aim of these algorithms is to reduce the search space which has 2^p states, where p is the number of propositions. To find a state transition path a simple shortest path algorithm can by used solving the problem in $O(m \log m)$ time, where $m = 2^p$ [21]. Considering a problem where we have 20 propositions, the search space consists from $2^{20} = 1\,048\,576$ states. In reality, we often have more hundreds of propositions. This seriously increases the number of states. Hence, the simple algorithms are not usable. A lot of known algorithms try to reduce the search

space. These use different heuristics or approaches utilizing evolution algorithms such as *hill climbing*. Different approaches tailored to the web service composition problem dealing with the space searching are discussed later. The correspondence between AI planning and web service elements is depicted in Table 6-1.

Table 6-1. Correspondence between web services and space–base planning elements.

AI planning element	Web service element
s_0	initial state
G	desired goal state
A	available web services
Γ	web service state change function

OWL-S is the language containing elements directly corresponding with state-space based AI planning. The state change produced by the execution of the web service is described by the preconditions and effects of the OWL-S *ServiceProfile*.

Graph Based Planning

In this approach graph structures are used when solving the planning problem. This structure is called planning graph [21]. It is a directed leveled graph consisting of two types of nodes: action nodes and proposition nodes. These nodes are organized into altering levels (one level includes nodes only of one type). The first level of the planning graph contains proposition nodes describing the initial situation. It is then followed by a level of action nodes. This level contains nodes for each action whose preconditions are satisfied by the proposition nodes of the first level. Next level is again proposition level. It contains each node from the precedent proposition level and nodes representing the effects of the actions from the precedent action level. The construction of the planning graph stops when the successive proposition levels are identical. If the goal is not reached either in the last proposition layer or in previous layers, the plan can not be constructed and the given problem is unsolvable.

In Figure 6-13 a planning graph example is depicted. It presents a situation, when Eva and Adam are in room number one (R1) and they want to move into the room number two (R2). To do this, the room R1 must be opened and only a man can open a room. We have the following predicates:

```
(at ?who ?r) - true if who is in a room r (opened ?r) - true if r
is opened (man ?m) - true if m is a man
```

At the beginning, the following propositions hold:

- Eva is in the room R1
- Adam is in the room R1
- Adam is a man
- The room R1 is closed

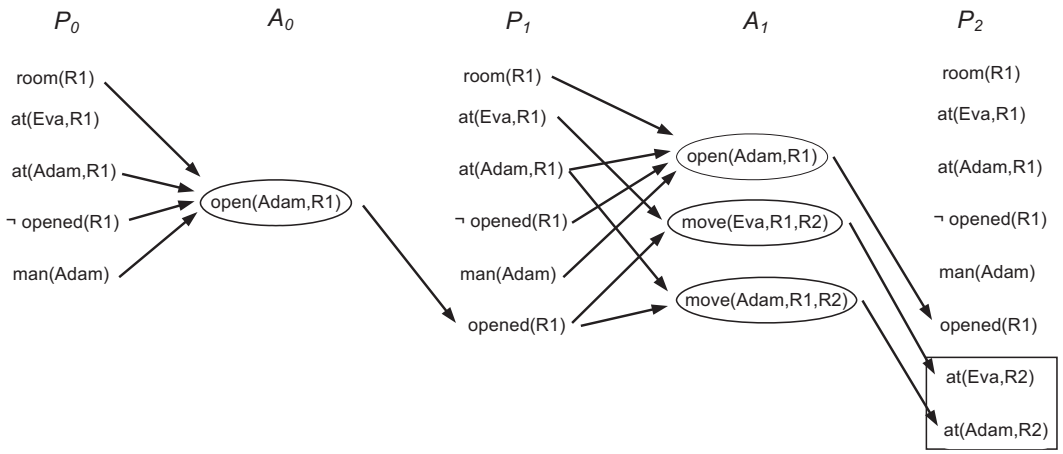


Figure 6-13. Planning graph example.

The following actions are available:

```
open() Parameters: ?r, ?who Preconditions: (room ?r), not(opened
?r), (man ?who), (at ?who ?r) Effects: (opened ?r)

move() Parameters: ?who, ?from ?to Preconditions: (at ?who ?from),
(room ?from), (room ?to), (opened ?from) Effects: (at ?who ?to),
not (at ?who ?from)
```

In this example, the goal (Eva and Adam are in room R2) is achieved in the last, second layer.

In [19] a *mutex* relation (Mutual Exclusions) between planning graph nodes is defined. The possible mutex relations between two action nodes are: *inconsistent effects* – one action negates the effect of the other, *inference* – one of the effect of one action negotiates a precondition of another, *competing needs* – one of the precondition of one action is mutually exclusive with the precondition of the other. In [19] also GRAPHPLAN is presented. It is an algorithm consisting of two stages altering in a loop: *graph expansion* and *solution extraction*. Graph expansion grows the graph until the reached proposition level includes goal propositions without mutex links. In solution extraction stage, backward-chaining strategy is used to look forward a potential plan.

Backward Chaining

One of the simplest methods used in web service composition is *backward chaining*. Its basic principle is to begin with the goal and search for services which produce its parts. This is used to reach the goal propositions and also the output. We take the set of goal propositions and look for services which produce them as effects. These services may have preconditions which are not satisfied. For this set of unsatisfied propositions we again look for services which produce them. This backward search is performed until we have service with unsatisfied preconditions. Similarly, this method is applied to get the required

output. Here we look for services producing the required output data. The goal is to achieve a situation that each service has defined a source for its input data. If it is possible to compose a service from the available ones, backward chaining is a method which always finds it. Despite this, pure backward chaining is unusable effectively because of the computational complexity of this approach.

Planning Based on Logical Programming

There exist approaches which uses logical programming to solve the planning problem. In these, the planning problem is encoded as logic program composed by a set of Horn clauses. The plan is created using reasoning. In [75] it is shown that the planning problem can be converted to a logic program in linear time.

Hierarchical Task Network Planning

HTN (Hierarchical Task Network) provides a hierarchical abstraction to deal with the complexity of the planning problem. HTN like other planning techniques assumes a set of operators applicable when preconditions are satisfied to achieve defined effects. In addition it assumes also methods prescribing how tasks can be decomposed into subtasks. These prescriptions are a part of the domain knowledge. The decomposition is applied recursively until the resulting set of tasks consists only from executable ones. After each step of decomposition it is tested whether given conditions are violated. The goal is to achieve decomposition consisting of executable tasks without violating any given condition. A variant of HTN planning is an ordered task decomposition planning. Here the tasks are planned in the same order that they will be executed. This reduces the complexity of the planning problem.

Temporal Planning

Temporal planning is not a planning technique, but it rather refers to the ability of planners to deal with temporal aspects of the planning problem. It takes into the consideration the real duration of the actions and allows their parallel execution. Concerning also these aspects makes the problem more difficult but better reflects the real conditions of the problem that the planner solves.

6.3.3 Web Services Execution

The result of the web service compositors is a plan describing a sequence of actions which need to be performed to achieve a goal. Each sequence is bound with a corresponding executable web service. To execute a composed web service an engine interpreting the composition is required [43, 44]. The execution can be performed in distributed manner [59]. In the case of services with high computational requirements, the execution is distributed in Grid environment. There exist approaches for description, construction and execution of Semantic web services in Grid [6, 36, 38]. Project K-Wf Grid[5] has brought results in this area.

[5] `http://www.gridworkflow.org/snips/gridworkflow/space/K-Wf+Grid/`

A language *GWorkflowDL* for Grid workflow description and execution environment *GWES* were developed.

Before this, the composition must be transformed into a form the interpreter works with. The web service execution process can interleave with the planning. Based on this, we divide the approaches to static and dynamic. Static means that during the planning no service is executed. Dynamic approaches execute some services to examine the state of the world or the availability of data.

One solution for web service execution is offered by WSMX[6] [26]. It provides an environment for discovery, selection, mediation, invocation, and interoperation of web services. WSMX is based on the conceptual model provided by WSMO. The main components of WSMX are:

- *Core component*: central part of WSMX which coordinates all other components.
- *Resource manager*: manages the repositories of WSMO entities (web services, goals, ontologies, and mediators) and other data.
- *Service discovery*: locates the web services that fulfill the user's requirements.
- *Service selection*: selection of the best or optimal Semantic web service from the list of those satisfying matching service.
- *Data and process mediator*: takes care about mediation when two entities are not able to communicate or interact.
- *Communication manager*: supports communication between service requestor and provider.
- *Choreography engine*: makes sure that the requestor's and provider's communication patterns match.
- *Web service modeling toolkit*: supports the creation and deployment of tools for Semantic web services for the developers.
- *Reasoner*: provides reasoning for the validation of composite services and checking if it is executable in a given context.

WSMX presents a complex solution for all web service execution related concerns, however not fully implemented at this time. Another instrument for web service execution is offered by BPEL4WS[7] [4]. It is of the leading notation describing executable web service compositions. The processes – workflows described in BPEL4WS can be executed with engines working with this language. One of the existing, available implementations are *BPWS4J*[8]

[6] Web Service Execution Environment, `http://www.w3.org/Submission/WSMX/`

[7] Business Process Execution Language for Web Services, `http://www.ibm.com/developerworks/library/specification/ws-bpel/`

[8] `http://www.alphaworks.ibm.com/tech/bpws4j/`

and *ActiveBPEL*[9]. These two provide means to execute composite web services. BPEL4WS is not so complex solution as WSMX, but it is already implemented and also used in practice.

Business processes can be described by BPEL4WS in two ways:

- *Executable business processes*: model actual behavior of a participant in a business interaction.
- *Abstract business processes*: describes the mutually visible message exchange behavior of each of the parties involved, without revealing their interior behavior.

BPEL4WS is meant to describe both kinds of these processes [80]. It provides an XML language for formal specification and extends the web service interaction model and enables it to support business transactions. It is itself a flow-chart where each element is a primitive or structured activity. The primitive activities include the following:

- *invoke*: invocation of web service operations,
- *receive*: waiting for an message from external source,
- *reply*: replying to an external source,
- *waiting*: waiting for a certain time,
- *assign*: copying data,
- *throws*: indication of errors in execution,
- *terminate*: terminate the process,
- *empty*: does nothing.

The structured activities contain the control constructs like *sequence*, *switch*, *while*. BPEL4WS contain also elements for fault and exception handling. Using the primitive and structured activities complex, executable workflows can be described.

Before execution of composite web services it is suitable to analyze it. The aim is to examine the correctness of web services. This include checking the pre and postconditions, input and output data compatibility between successive services, infinite loops or deadlocks detection. The compatibility of successive services is checked by a simple algorithm. It verifies if for each service the preconditions are satisfied and input data are available as a result of the previous services or as initial data. More complicated is the infinite loop of deadlock detection in the created composition. OWL-S and BPEL4WS do not offer ways to verify this. We need other means to do this [58]. Approaches based on Petri nets, II-calculus exist to perform this.

Petri nets are used for controlling the execution of the workflow [79]. Petri net is a mathematical representation of discrete systems. It can be seen as a graphical notation

[9] `http://www.active-endpoints.com/active-bpel-engine-overview.htm`

based on directed bipartite graphs. There are two types of nodes: places and transitions. Directed arcs exist only between different types of nodes (it is a bipartite graph). The places from (to) which an arc runs to (from) a transition are called input (output) places. Places contain tokens which are distributed over them. This distribution is called marking. Transitions act in such a way that they take input tokens, process them and put the output tokens into output places (the input tokens are consumed, i.e. will not be found far in the input places). This is called firing. A transition is enabled if all input tokens are available. In this situation it automatically fires.

Using the elements of Petri nets we describe compositions of web services [39, 60]. In Table 6-2 the correspondence between Petri nets and web services' elements is shown.

Table 6-2. Correspondance between Petri nets and web service elements.

Petri net	**Web service**
Tokens in input places	Input data
Tokens in output places	Output data
Transitions	Web services
Firing	Web service method execution
Marking	State of the execution

More detailed overview of how Petri nets formalize the elements of web services is available in [60]. It also discusses how Petri nets are used for simulation, validation, verification, composition, and performance analysis of web service compositions. From the web service execution point of view the most important is the verification. It deals with the question if it is possible to achieve a goal with available services. If a Petri net depicts the behavior of available services, this problem is transformed to a reachability question. It is that if it is possible to reach a marking which depicts the user's goal. This question can be answered for example by searching the state-space with breath-first search, however this approach is in reality unpractical from computation reasons. Rather linear temporal logic with tableau method is used to decide if the goal state can be reached. Petri nets are also used to check the occurrence of deadlocks. A marking is a deadlock if it enables no transitions. We want to avoid this situation. There exist algorithms which solve the reachability and deadlock problem [60].

One of the most important topics within web services is the security [69]. In [35] a process for web service security – PWSec is presented. This process specifies how to define security requirements for web services-based systems describes logical security services-based reference security architecture and explains how to instantiate it to obtain a concrete security architecture based on the current web service security standards. In [50], they propose a uniform format to describe communications and computer security through security objects and coherent integrations of this format into WSDL as a proper extension and also into an ontology framework for security knowledge representation and reasoning.

6.3.4 Semantic Web Service Composition Problems

The automatic composition of web services has shown to be a challenging task and it is not clear which techniques serve the problem best [73]. There are several problems in different

phases of web service composition. It is important to know that not each real problem can be described and solved using planning. Examples are environments where the future state can not be determined before it is reached. For instance, the n-body problem belongs into this category. It deals with the finding of motions of n bodies determined by Newton's laws of motion and Newton's law of gravity. The first contribution to the solution of n-body problem has been done by Poincaré in 1892 [40]. The first version of his contribution contained an error, described in [34]. The problem was finally solved by Karl Fritiof Sundman (for $n = 3$) [42]. The result is that it is not possible to solve the n-body problem analytically. Only numerical methods can be used to find a solution with arbitrary precision (greater than zero). Consider a planning problem where we want to the fly to Mars and land on an exactly specified point. The existence of the n-body problem proves that there can not be found a plan which solves this problem.

The subset of problems solvable by planning techniques is restricted. Consider all domains solvable using a common computer, more precisely by a Turing machine with final tape. These domains are also solvable using a STRIPS-like or HTN-like planners, whose expressiveness is equivalent [49]. Hence, the set of problems which can be solved by planning is restricted at least to this domain. Based on the concrete technique used, other restrictions can occur.

Now we focus on more concrete problems we must deal with in Semantic web service composition. The first problem of the utilization of web service composition appears already at the first step. This is how to gather information about the initial situation and the goal state from the user. The web service composition methods already work with specific formalisms like OWL-S. Descriptions in these languages can not be directly obtained from the user. Different user-computer interactions can take place here.

One example of a user interaction tool is described in [56, 57]. The gathering of the information is based on context detection. The context was detected based on user's textual input of a note. A tool EMBET analyzes this input and finds relation between parts of this note and ontological concepts. After this a list of context suggestions are presented to a user who can select the relevant ones. Based on the context, those services which are closely related to it are suggested to be those producing the desired output for the user.

Another approach to gathering information about the user's goal is presented in [71]. The described system is a smartphone interface to the Semantic web – SmartWeb. It constitutes from a PDA client, dialog server, and the Semantic web access system. In the PDA client a Java-based control unite takes care about the inputs and outputs. The dialog server includes modules for speech interpretation, natural language generation, and user context detection. The Semantic web access system contains modules for question-answering sub-system, semantic web service composer, semantic web pages, and knowledge server. The aim of the system is to make the Semantic web accessible for the user in convenient way. It assists the user during finding information taking into account its context and exploiting also Semantic web services. When web service composition is required, the system generates questions to acquire the necessary information for construction and execution of the composite service.

It is not clear if the approach used in SmartWeb is sufficient to be used universally and how dependent it is on the domain. Very important role in this system plays the ontological knowledge base. Building a sufficiently large, consistent ontology which can be effectively processed by machines still presents a problem. Automatic approaches using for example

wrappers are often insufficient to perform this task. The problem of having high quality ontologies influences the Semantic web generally. Hence, this is not only the problem of using SmartWeb in different domains and situations. Some problems concerning this topic are discussed in [9] which deals with creation of ontological test bases for experimental evaluation of Semantic web applications.

Even if we have already described the goal in an acceptable form, there are still other problems. Despite of existing web service composition tools, the construction of an optimal composite web service can not be always achieved in sufficiently short time even if the necessary sub-services are available. This is due the complexity of this task (discussed in section 6.3.2).

When composite web service is constructed, other services are required to be organized into one. Only available web services which are able to be executed can be taken into account. Here the problems of discovering suitable services mentioned in section 6.3.1 must be considered. The composition method must also deal with situation when it is not possible to build a composite service, which fully satisfies the user goal.

The web services can be considered as transformations, which taking a set of input parameters, produce a set of output parameters. These can be described as $ws : (i_1, \ldots, i_n) \longrightarrow (o_1, \ldots, o_m)$. Before the constructed composite web service can be executed, the input parameters must be provided. The problem is that the user only defines the output parameters. It is not clear how to define the input parameters if we have described only the outputs. In example from Figure 6-12 the goal is to create forecast maps for a defined area and term. Based on this the presented composite service is constructed. This service composition is only the scheme which says in which order the services will be called and how will they shift the data between themselves.

It is clear that the input for the *Compute forecast* service is the output of the *Create meteo model* service. The input for the *Create meteo forecast maps* service is the output of the *Compute forecast* service. The problem is that the inputs of the first, *Create meteo model* service are not provided by other service(s). From the description of the service only the type of the input parameters are known. The concrete input data for this service must be defined and provided other way. Automatic definition of these data presents a hard problem. There is no universal method how to do this. It is a domain dependent problem and often domain expert knowledge is required for this task. In the meteorological forecast map example, a meteorologist must define different parameters required to create a meteorological model from which the forecast for the specified time and area can be generated. Hence, the definition of the input data is hard to realize in automatic manner or even impossible.

One possible solution may be the description of the data transformation the service performs. This means that we know what output parameters will be produced by a service if we know the input. For example a description of the web service which multiplies the input with two must explain that this service doubles the input. This still not solve the problem. The problem is solved only if we know also the inverse transformation ws^{-1}: $(o_1, \ldots, o_m) \longrightarrow (i_1, \ldots, i_n)$. In our example it means that the service description stands that the inverse transformation is the division by two. If this information is available, we are able to determine the required input parameters based on the output parameters. Problem is that it is not always possible to define the inverse function. An example is the service realizing addition of two numbers. The addition has no inverse function. Solutions for this case are not known.

We have discussed several approaches used to compose web services. Now, we present a new category of problems which can not be solved using these approaches. We extend the example from section 6.3.2 to describe the problem. In this example the goal is to move Eva and Adam from one room the other. This situation can be described by simple propositions, i.e. the goal is to reach a state in which certain propositions are true. Described AI planning techniques are able to solve this problem and determine the sequence of actions which need to be performed to achieve the goal state. We change now a little bit the problem. Consider that in room one is a bottle of wine (bottle one). In room two, there is an empty bottle (bottle two). The goal is to carry the wine from the bottle one into the empty bottle two. To do this a goblet is available. We can fill the goblet with wine from bottle one, carry it to room two and pour it into bottle two. This is being done until the whole bottle of wine is carried.

This extension of the example causes that we are not able to describe and solve the problem using the discussed approaches. The problem is that we introduce operations with effects that can not be described by simple propositions with true and false values. The fill and pour operations have effects that the bottle is fillet at certain level. This can not be described using true and false values. We need to capture the amount of wine in the bottle. To describe the effects of the fill and pour operation arithmetic needs to be employed.

The solution of the wine carrying problem is simple. It is depicted in Figure 6-14 where simplified process solving the problem is shown. In a loop we are carrying the wine in goblets until the bottle one is empty. Discussed approaches to problem description and solving are unmatured to handle this problem. First, there are no constructs to use arithmetic. Second, the methods to find the plan – sequence of actions which lead from the initial to goal state, do not work with constructs like conditional loops. Discussed planning methods create a plan from which it is clear what the sequence of actions to achieve the goal is. In the wine carrying problem, this can not be known without the information about the amount of wine in bottle one. Concretely, the number of loops depicted in Figure 14 is dependent on the amount of wine in bottle one, i.e. parameters decides what the sequence of actions required is. To deal with this category of problems, new methods working with arithmetic means and complex constructs have to be developed.

We have presented several problems related to Semantic web service composition. Discussed problems present issues we have to deal with. Until these problems will not be solved, the utilization of Semantic web services in practice will be limited.

6.4 Conclusions

Adding semantics to the web services description has shown to be useful in different contexts. The additional metadata describing the meaning of the different elements of web services makes more effective several processes. The most important are web service discovery and composition. The exploitation of the available semantics makes these processes more effective and the results are more relevant.

The research in the context of web service discovery is nowadays focused to solve the problem of service matchmaking. The aim of the discovery is to find services producing semantically suiting outputs. The suiting outputs are those which are semantically the same as the required data or are sufficiently close to it.

The automatic semantic web service composition has shown to be a challenging problem. A lot of research focuses on web service composition to bring solutions moving this

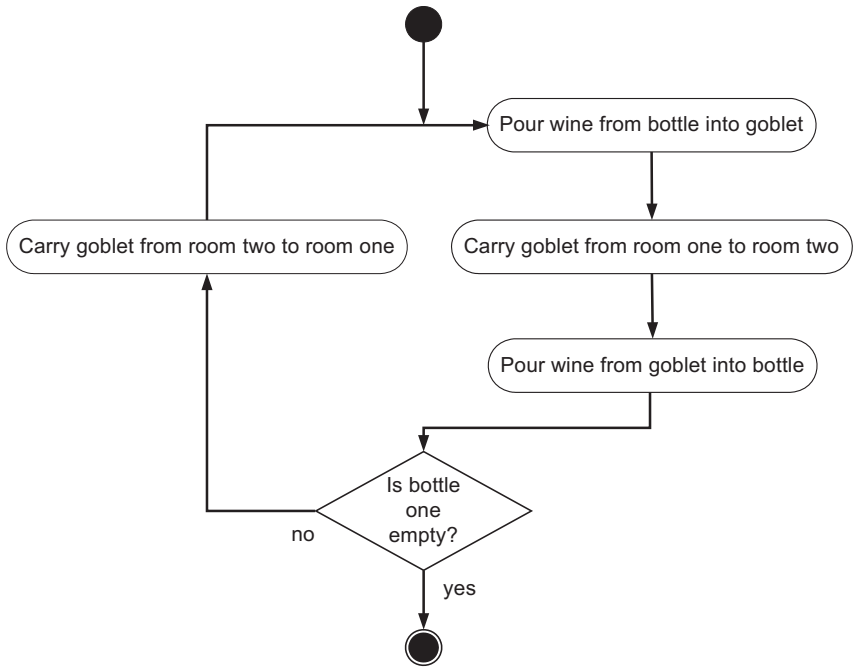


Figure 6-14. Wine carrying problem.

technology closer to the usage in practice. One of the most discussed topics are scalability of composition approaches from performance point of view and the evaluation of the aggregated QoS characteristic of the resulting solution. These aspect of the problem are necessary to deal with. There already are presented promising results regarding these topics [10, 12, 14, 15, 16, 46, 51].

References

[1] Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.T., Sheth, A., Verma, K.: *Web Service Semantics - WSDL-S*. W3C, retrieved from `http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.pdf`, 2007.

[2] Akkiraju, R., Goodwin, R., Doshi, P., Roeder, S.: A Method for Semantically Enhancing the Service Discovery Capabilities of UDDI. In: *IJCAI-03 Workshop on Information Integration on the Web, IIWeb-03*, 2003.

[3] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[4] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: *Business Process Execution Language for Web Services*, 2003, retrieved from `http://www.ibm.com/developerworks/library/specification/ws-bpel/`, 2007.

[5] Atkinson, C., Bostan, P., Hummel, O., Stoll, D.: A Practical Approach to Web Service Discovery and Retrieval. *icws*, 2007, pp. 241–248.

[6] Babik, M., Habala, O., Hluchy, L., Laclavik, M., Maliska, M.: Semantic Grid Services in K-Wf Grid. In: *Proc. of the 2nd International Conference on Semantics, Knowledge and Grid*. Lecture Notes in Computer Science, IEEE Computer Society, 2006, pp. 149–155.

[7] Barla, M., Bartalos, P., Bieliková, M., Filkorn, R., Tvarožek, M.: Adaptive portal Framework for Semantic Web applications. In: *Proc. of the 2nd Int. Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'07), Como, Italy, July 2007*, 2007.

[8] Barla, M., Bartalos, P., Sivák, P., Szobi, K., Tvarožek, M., Filkorn, R.: Ontology as an Information Base for Domain Oriented Portal Solutions. In: *15th Int. Conf. on Information Systems Development, ISD'06, Budapest, Hungary*, 2006.

[9] Bartalos, P., Barla, M., Frivolt, G., Tvarožek, M., Andrejko, A., Bieliková, M., Návrat, P.: Building an Ontological Base for Experimental Evaluation of Semantic Web Applications. In et al., J.v., ed.: *SOFSEM '07*, Springer, LNCS, 2007.

[10] Bartalos, P., Bielikova, M.: Adapting I/O Parameters of Web Services to Enhance Composition. In: *NWeSP '09: Proc. of the 2009 IEEE Conference on Next Generation Web Services Practices, year = 2009, publisher = IEEE CS*.

[11] Bartalos, P., Bieliková, M.: An approach to object-ontology mapping. In: *2nd IFIP Central and East European Conference on Software Engineering Techniques CEE-SET 2007*, 2007.

[12] Bartalos, P., Bielikova, M.: Enhancing Semantic Web Services Composition with User Interaction. In: *SCC '08: Proc. of the 2008 IEEE Int. Conf. on Services Computing*, IEEE CS, 2008, pp. 503–506.

[13] Bartalos, P., Bieliková, M.: (S)CRUD Pattern Support for Semantic Web Applications. In: *SOFSEM 2008*, 2008, Accepted in student research forum.

[14] Bartalos, P., Bielikova, M.: Fast and Scalable Semantic Web Service Composition Approach Considering Complex Pre/Postconditions. In: *WSCA '09: Proc. of the 2009 IEEE Congress on Services, Int. Workshop on Web Service Composition and Adaptation*, IEEE CS, 2009, pp. 414–421.

[15] Bartalos, P., Bielikova, M.: Semantic Web Service Composition Framework Based on Parallel Processing. In: *CEC '09: Proc. of the 2009 IEEE Conference on Electronic Commerce*, IEEE CS, 2009, pp. 495–498.

[16] Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for QoS-aware Web Service Composition. In: *ICWS '06: Proceedings of the IEEE Int. Conf. on Web Services*, IEEE CS, 2006, pp. 72–82.

[17] Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. *Scientific American*, 2001, pp. 34–43.

[18] Berstis, V.: *Fundamentals of Grid Computing*. IBM Redbooks, 2002.

[19] Blum, A., Furst, M.: Fast Planning Through Planning Graph Analysis. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, 1995, pp. 1636–1642.

[20] Brambilla, M., Ceri, S., Celino, I., Cerizza, D., Valle, E.D., Facca, F.M., Tziviskou, C.: Flexible Specification of Semantic Services using Web Engineering Methods and Tools. In: *Proceedings of SWESE 2006, ISWC 2006, Athens, GA, USA*, 2006, pp. 1–14.

[21] Bryce, D., Kambhampati, S.: A tutorial on planning graph–based reachability heuristics. *AI Magazine*, 2007, vol. 28, no. 1, p. 47.

[22] Bussler, C., Fensel, D., Maedche, A.: A conceptual architecture for semantic web enabled web services. *SIGMOD Rec.*, 2002, vol. 31, no. 4, pp. 24–29.

[23] Cardoso, J., Sheth, A.P.: *Semantic Web Services, Processes and Applications*. Springer, 2006.

[24] Cerami, E.: *Web Services Essentials*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[25] Chen, X., Cai, W., Turner, S.J., Wang, Y.: SOAr-DSGrid: Service-Oriented Architecture for Distributed Simulation on the Grid. In: *PADS '06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, IEEE CS, 2006, pp. 65–73.

[26] Cimpian, E., Moran, M., Oren, E., Vitvar, T., Zaremba, M.: *Overview and Scope of WSMX*, 2005, retrieved from `http://www.wsmo.org/TR/d13/d13.0/v0.2/index.pdf,2007`.

[27] Coenen, A.: An SOA Case Study: Agility in Practice. *The SOA Magazine*, 2006.

[28] Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S.: Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 2002, vol. 6, no. 2, pp. 86–93.

[29] Curbera, F., Khalaf, R., Mukhi, N., Tai, S., Weerawarana, S.: The next step in Web services. *Commun. ACM*, 2003, vol. 46, no. 10, pp. 29–34.

[30] de Bruijn, J., Lausen, H., Krummenache, R., Polleres, A., Predoiu, L., Kifer, M., Fensel, D.: *The Web Service Modeling Language WSML*, 2005.

[31] Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P., Krogdahl, P., Luo, M., Newling, T.: *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, 2004.

[32] Erl, T.: *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Upper Saddle River: Prentice Hall PTR, 2005.

[33] Erl, T.: *SOA Principles of Service Design*. Upper Saddle River: Prentice Hall PTR, 2007.

[34] F. Diacu: The solution of the n-body Problem. *The solution of the n-body Problem*, 1996, pp. 66–70.

[35] Gutierrez, C., Fernandez-Medina, E., Piattini, M.: PWSSec: Process for Web Services Security. *icws*, 2006, vol. 0, pp. 213–222.

[36] Habala, O., Babik, M., Hluchy, L., Laclavik, M., Balogh, Z.: Semantic Tools for Workflow Construction. In: *Proc.of International Conference on Computational Science*. Volume 3993 of *Lecture Notes in Computer Science.*, Springer Verlag, 2006, pp. 980–987.

[37] Heß, A., Johnston, E., Kushmerick, N.: Machine Learning Techniques for Annotating Semantic Web Services. In: *Semantic Web Services - 2004 AAAI Spring Symposium*.

[38] Hluchy, L., Habala, O., Babik, M.: K-WF Grid: Grid workflows with knowledge. In: *Proceedings of . 1st Austrian Grid Symposium*, Oesterreichische Computer Gesellschaft, 2006, pp. 296–307.

[39] Hoheisel, A., Der, U.: An XML-based framework for loosely coupled applications on Grid environments. In: *P.M.A. Sloot et al., eds., International Conference on Computational Science 2003, LNCS 2657*.

[40] J. H. Poincaré: Les méthods nouvelles de la mécanique céleste. *Gauthier-Villars, Paris*, 3 vols., 1892-99; Eng. trans., 1967.

[41] Jones, S.: Toward an Acceptable Definition of Service. *IEEE Software*, 2005, vol. 22, no. 3, pp. 87–93.

[42] K. E. Sundman: Memoire sur le probleme de trois corps. *Acta Matematica*, 1912, pp. 105–179.

[43] Keidl, M., Seltzsam, S., Kemper, A.: Flexible and Reliable Web Service Execution. In: *Proc. of the 1st Workshop on Entwicklung von Anwendungen auf der Basis der XML Web-Service Technologie*, 2002, pp. 17–30.

[44] Keidl, M., Seltzsam, S., Kemper, A.: Reliable Web Service Execution and Deployment in Dynamic Environments. In Benatallah, B., Shan, M.C., eds.: *TES*. Volume 2819 of *Lecture Notes in Computer Science*., Springer, 2003, pp. 104–118.

[45] Klusch, M.: Semantic Web Service Coordination. In Schumacher, M., Helin, H., Schuldt, H., eds.: *CASCOM - Intelligent Service Coordination in the Semantic Web*. Birkhaeuser Verlag, Springer, 2008.

[46] Kona, S., Bansal, A., Blake, M.B., Gupta, G.: Generalized Semantics-Based Service Composition. In: *ICWS '08: Proc. of the 2008 IEEE Int. Conf. on Web Services*, IEEE CS, 2008, pp. 219–227.

[47] Korotkiy, M., Top, J.: Onto-SOA: From Ontology-enabled SOA to Service-enabled Ontologies. In: *Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, Los Alamitos, CA, USA, IEEE Computer Society, 2006, p. 124.

[48] Lausen, H., Stollberg, M., Hernandez, R.L., Ding, Y., Han, S.K., Fensel, D.: Semantic web portals: state-of-the-art survey. *Journal of Knowledge Management*, 2005, vol. 9, no. 5, pp. 40–49.

[49] Lekavý, M., Návrat, P.: Expressivity of STRIPS-Like and HTN-Like Planning. In: *1st KES International Symposium, KES-AMSTA 2007, Wroclaw, Poland, Lecture Notes in Artificial Intelligence, Agent and multi-agent Systems, Technologies and applications*. Volume 4496., Springer-Verlag Berlin Heidelberg, 2007, pp. 121–130.

[50] Li, C., Pahl, C.: Security in the Web Services Framework. In: *ISICT '03: Proceedings of the 1st international symposium on Information and communication technologies*, Trinity College Dublin, 2003, pp. 481–486.

[51] Liu, A., Li, Q., Huang, L., Xiao, M., Liu, H.: QoS-Aware Scheduling of Web Services. In: *WAIM '08: Proceedings of the 2008 The Ninth Int. Conf. on Web-Age Information Management*, IEEE CS, 2008, pp. 171–178.

[52] Louridas, P.: SOAP and Web Services. *IEEE Software*, 2006, vol. 23, no. 6, pp. 62–67.

[53] Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing Semantics to Web Services: The OWL-S Approach. In: *1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, 2004.

[54] McGuinness, D.L., van Harmelen., F.: *OWL Web Ontology Language Overview*, 2004, retrieved from `http://www.w3.org/TR/owl-features/`,2007.

[55] McIlraith, S.A., Son, T.C., Zeng, H.: Semantic Web Services. *IEEE Intelligent Systems*, 2001, vol. 16, no. 2, pp. 46–53.

[56] Michal Laclavík, Martin Šeleng, L.H.: Towards Pro-active Knowledge Sharing in Grid Environment. In: *GCCP 2007 – 3rd International Workshop on Grid Computing for Complex Problems*, 2007.

[57] Michal Laclavík, Martin Šeleng, L.H.: User Assistant Agent: Towards Collaboration and Knowledge Sharing in Grid Workflow Applications. In: *Cracow ´06 Grid Workshop: K-Wf Grid. - Cracow, Poland: Academic Computer Centre CYFRONET AGH, 2007*, 2007, pp. 122–130.

[58] Milanovic, N., Malek, M.: Current Solutions for Web Service Composition. *IEEE Internet Computing*, 2004, vol. 8, no. 6, pp. 51–59.

[59] Nanda, M.G., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. In: *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, ACM, 2004, pp. 170–187.

[60] Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of web services. In: *WWW '02: Proceedings of the 11th international conference on World Wide Web*, New York, NY, USA, ACM, 2002, pp. 77–88.

[61] Návrat, P., Bieliková, M., Rozinajová, V.: Methods and Tools for Acquiring and Presenting Information and Knowledge in the Web. In: *Int. Conf. on Computer Systems and Technologies, CompSysTech'05*, Varna, Bulgaria, 2005.

[62] Object Management Group Inc. (OMG).: *Meta object facility (MOF) specification v1.4*. OMG, 2002.

[63] OWL services coalition: *OWL-S: semantic markup for web services*. OWL-S coalition, 2004, retrieved from `http://www.daml.org/services/owl-s/1.0/owl-s.html`, 2007.

[64] Paikari, E., Habibi, J., Yeganeh, S.H.: Semantic Composability Measure for Semantic Web Services. In: *AMS '07: Proceedings of the First Asia International Conference on Modelling & Simulation*, Washington, DC, USA, IEEE Computer Society, 2007, pp. 88–93.

[65] Paolucci, M., Wagner, M.: Grounding OWL-S in WSDL-S. In: *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, Washington, DC, USA, IEEE Computer Society, 2006, pp. 913–914.

[66] Payne, T., Lassila, O.: Guest Editors' Introduction: Semantic Web Services. *IEEE Intelligent Systems*, 2004, vol. 19, no. 4, pp. 14–15.

[67] Peer, J.: *Web Service Composition as AI Planning – a Survey*. University of St.Gallen, 2005.

[68] Pistore, M., Ambite, J.L., Blythe, J., Koehler, J., McIlraith, S., Srivastava, B.: AI for Service Composition. In: *AISC '06: Proceedings of the 4th International Workshop on AI for Service Composition*, 2006.

[69] Rahaman, M.A., Schaad, A., Rits, M.: Towards secure SOAP message exchange in a SOA. In: *SWS '06: Proceedings of the 3rd ACM workshop on Secure web services*, New York, NY, USA, ACM Press, 2006, pp. 77–84.

[70] Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web Service Modeling Ontology. *Applied Ontology*, 2005, vol. 1, no. 1, pp. 77–106.

[71] Sonntag, D., Engel, R., Herzog, G., Pfalzgraf, A., Pfleger, N., Romanelli, M., Reithinger, N.: SmartWeb Handheld - Multimodal Interaction with Ontological Knowledge Bases and Semantic Web Services. In Huang, T.S., Nijholt, A., Pantic, M., Pentland, A., eds.: *Artifical Intelligence for Human Computing*. Volume 4451 of *Lecture Notes in Computer Science*., Springer, 2007, pp. 272–295.

[72] Srinivasan, N., Paolucci, M., Sycara, K.: An Efficient Algorithm for OWL-S Based Semantic Search in UDDI. *Semantic Web Services and Web Process Composition*, 2005, vol. 3387/2005, no. 1, pp. 96–110.

[73] Srivastava, B., Koehler, J.: Web service composition – current solutions and open problems. In: *International Conference on Automated Planning and Scheduling 2003*, 2003.

[74] Studer, R., Grimm, S., Abecker, A.: *Semantic Web Services: Concepts, Technologies, and Applications*. Springer, 2007.

[75] Subrahmanian, V.S., Zaniolo, C.: Relating Stable Models and AI Planning Domains. In: *International Conference on Logic Programming*, 1995, pp. 233–247.

[76] Swartz, A.: MusicBrainz: A Semantic Web Service. *IEEE Intelligent Systems*, 2002, vol. 17, no. 1, pp. 76–77.

[77] Verma, K., Sheth, A.: Semantically Annotating a Web Service. *IEEE Internet Computing*, 2007, vol. 11, no. 2, pp. 83–85.

[78] Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S., Miller, J.: METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Inf. Tech. and Management*, 2005, vol. 6, no. 1, pp. 17–39.

[79] W.M.P. van der Aalst: The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 1998, vol. 8, no. 1, pp. 21–66.

[80] Wohed, P., van der Aalst, W.M., Dumas, M., ter Hofstede, A.H.: Pattern Based Analysis of BPEL4WS. QUT Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002.

[81] Ye, L., Zhang, B.: Web Service Discovery Based on Functional Semantics. *2nd International Conference on Semantics, Knowledge, and Grid (SKG'06)*, 2006, p. 57.

7

SPRÍSTUPŇOVANIE INFORMÁCIÍ POMOCOU GRAFOV

Ján Suchal

Sprístupňovanie informácií (*angl. information retrieval*) zahŕňa algoritmy, ktoré umožňujú efektívnejšie a kvalitnejšie vyhľadávanie a odporúčanie informácií v rozsiahlych databázach. Sprístupňovanie informácií spája široké spektrum problémov, od strojového určovania významu dopytov používateľa a sémantickú analýzu dát, cez efektívne reprezentácie informácií a zložitosť algoritmov až po distribuované systémy a paralelné algoritmy.

V súčasnosti sa nemalý dôraz kladie na možnosti personalizácie ako aj kolaborácie v odporúčacích a vyhľadávacích systémoch, ktoré umožňujú cielenejšie a tým aj kvalitnejšie výsledky. Pri momentálnom masovom komerčnom využití systémov využívajúcich takéto algoritmy sú však nezanedbateľné aj otázky, venujúce sa možnostiam a zamedzeniu ich úmyselného zneužitia či neúmyselného poškodenia.

Uvažujúc stránky na webe sú poprepájané odkazmi, vedecké publikácie pomocou citácii či ľudia pomocou priateľstiev. Informácie a objekty vo všeobecnosti málokedy existujú samostatne, ale majú prepojenia na iné objekty. Takto prepojené objekty je možné použitím rôznych transformácií modelovať grafmi.

Pomocou grafov je tak napríklad možné modelovať vzťahy medzi ľudmi (sociálne siete) na základe emailovej komunikácie, či ich spoločných záujmov. Stránky na webe sú medzi sebou poprepájané odkazmi čím predstavujú neustále sa meniaci graf obrovských rozmerov. Takéto modelovanie vzťahov objektov grafmi otvára nové možnosti využitia algoritmov a znalostí z teórie grafov pri odhaľovaní skrytých závislostí z veľkého množstva dát.

Táto práca sa zameriava na algoritmy, ktoré slúžia na ohodnocovanie takýchto grafov a ich následné využitie pri zlepšení sprístupňovania informácií.

Pre aplikáciou ohodnocovacích algoritmov je najskôr potrebné problémovú doménu vhodne transformovať na graf. Kapitola 7.1 predstavuje rôzne typy grafov a definuje ich vlastnosti.

Prehľadu vybraných ohodnocovacích algoritmov s ich vlastnosťami a špecifikami ako aj ich vzájomnému porovnaniu sa venuje kapitola 7.2. Spôsoby zneužitia jednotlivých ohodnocovacích algoritmov ako aj možnosti obrany voči takýmto zneužitiam sa nachádzajú v kapitole 7.3.

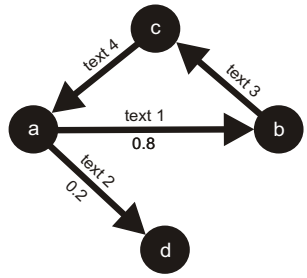
7.1 Modelovanie údajov grafmi

Údaje je možné modelovanie rôznymi spôsobmi, pričom jedným z nich sú aj grafy sústredʼujúce sa na vzťahy medzi logickými jednotkami. Táto kapitola sa venuje niektorým typom grafov, ktoré je možné využiť pri vyhľadávaní a odporúčaní informácií.

7.1.1 Graf ako vyjadrenie vzťahu objektov

Údaje modelované grafom sú z pravidla konštruované tak, že vrcholy grafu predstavujú objekty a hrany vzťahy medzi nimi. Typickým príkladom je graf prepojenia stránok na webe, kde vrcholy predstavujú stránky a orientované hrany predstavujú odkazy medzi nimi. Ak napríklad stránka a odkazuje na stránku b, potom v grafe musí existovať hrana z vrcholu a do vrcholu b.

Vo všeobecnosti môžu byť hrany ako aj vrcholy obohatené typom, ktoré reprezentujú dodatočný význam hrany. V súvislosti s odkazmi medzi stránkami je možné hrany grafu označiť textom, ktorým sa na danú stránku odkazuje.[1] Hrany v grafoch môžu byť ováhované, čím sa otvárajú nové možnosti rozšírenia. V kontexte odkazov medzi stránkami je tak možné napríklad priradiť vyššiu váhu odkazom, ktoré sú v nadpisoch a majú vyššiu váhu v rámci stránky.



Obr. 7-1. Ukážka všeobecného grafu s typovanými a ohodnotenými hranami.

7.1.2 k*-partitné grafy*

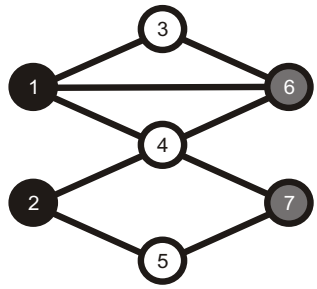
k-partitné grafy sú špeciálnou podmnožinou grafov, pre ktoré platí, že ich vrcholy je možné rozdeliť do k disjunktných množín, pričom existujú len hrany medzi jednotlivými množinami. Ukážka k-partitného grafu, kde $k = 3$ je na obrázku 7-2, pričom vrcholy rovnakej farby patria do rovnakej množiny.

7.1.3 Vrstvené grafy

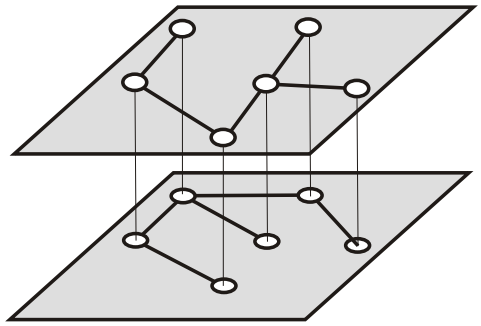
Vrstvené grafy sú špeciálnym typom k-partitných grafov, pre ktoré je možné vytvoriť viac vrstiev, pričom existujú iba hrany medzi vrcholmi v jednej vrstve, hrany medzi dvoma vrstvami a zároveň ak existuje hrana z vrstvy a do vrstvy b a z vrstvy b do vrstvy c, tak neexistuje žiadna hrana z vrstvy a do c.

Takéto grafy sú väčšinou konštruované tak, že každá vrstva väčšinou reprezentuje ten istý graf, kde v každej vrstve sú hrany len vybraného typu.

[1] V HTML sa odkaz v minimálnej forme zapisuje ako `<a href=“url”>text odkazu</a>`.



Obr. 7-2. Ukážka *k*-partitného grafu, kde $k = 3$ a farba vrcholu označuje príslušnosť do množiny.



Obr. 7-3. Ukážka vrstveného grafu s dvomi vrstvami.

7.1.4 Kontextové siete

Kontextová sieť je jednoduchá reprezentácia dát (*k*-partitným) grafom. Typicky sa používajú bipartitné grafy, v ktorých sú dva typy vrcholov. Napríklad pre doménu vyhľadávania v dokumentoch môžu byť vrcholy grafu dvoch typov: *dokument* a *slovný výraz*.

Tabuľka 7-1 obsahuje ukážkovú kolekciu dokumentov s označením vrcholov. Tabuľka 7-2 obsahuje slovné výrazy, ktoré sa v kolekcii nachádzajú vo viac ako dvoch dokumentoch.

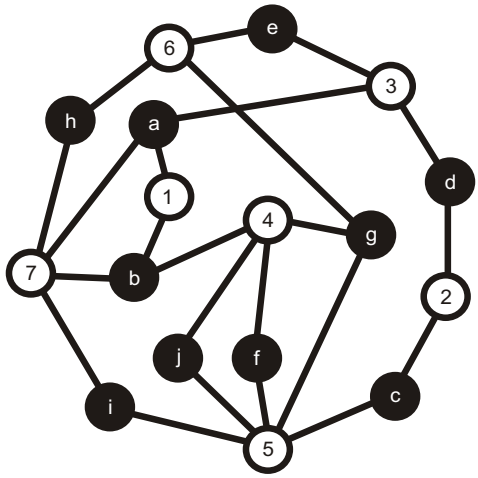
Tabuľka 7-1. Ukážka kolekcie dokumentov [14].

Vrchol	Obsah dokumentu
1	Glacial ice often appears blue.
2	Glaciers are made up of fallen snow.
3	Firn is an intermediate state between snow and glacial ice.
4	Ice shelves occur when ice sheets extend over the sea.
5	Glaciers and ice sheets calve icebergs into the sea.
6	Firn is half as dense as sea water.
7	Icebergs are chunks of glacial ice under water.

Tabuľka 7-2. Výrazy nachádzajúce sa aspoň v dvoch dokumentoch z tab. 7-1.

Vrchol	Slovný výraz
a	glacial ice
b	ice
c	glacier
d	snow
e	firn
f	ice sheet
g	sea
h	water
i	iceberg
j	sheet

Ak sa potom daný slovný výraz nachádza v danom dokumente, zodpovedajúce vrcholy sú v grafe spojené hranou. Graf (kontextová sieť) ukážkovej kolekcie dokumentov je na obrázku 7-4.



Obr. 7-4. Kontextová sieť ukážkovej kolekcie dokumentov.

Sémantika vzťahov v grafe je implicitne daná typom vrcholov, ktoré hrany spájajú a ich smerom. Pre uvedenú doménu fulltextového vyhľadávania má hrana smerujúca z vrcholu typu *dokument* do vrcholu typu *slovný výraz* význam <*dokument*> *obsahuje* <*slovný výraz*>. Opačný smer hrany má inverzný význam <*slovný výraz*> *sa nachádza v* <*dokumente*>.

7.1.5 Grafy v maticovom počte

V ďalšom texte budú využívané a vyžadované niektoré špeciálne vlastnosti grafov, ktoré sú charakterizované využitím maticového počtu. Táto kapitola v krátkosti definuje nevyhnutné pojmy maticového počtu z hľadiska teórie grafov.

7.1.6 Incidenčná matica grafu a jej vlastnosti

Incidenčná matica $\mathbf{M}$ grafu G je definovaná ako

$$\mathbf{M}_{ij} = \begin{cases} w_{ij} & \text{ak existuje hrana z vrcholu } i \text{ do vrcholu } j \\ 0 & \text{inak} \end{cases} \tag{7.1}$$

kde w_{ij} je váha hrany z vrcholu i do vrcholu j.

Matica $\mathbf{M}$ grafu G je *stochastická*, ak pre každý vrchol platí, že súčet váh hrán vystupujúcich z vrcholu je 1.

$$\sum_j \mathbf{M}_{ij} = 1; \forall i \tag{7.2}$$

Matica $\mathbf{M}$ grafu G je *ireducibilná*, ak existuje cesta z vrcholu i do vrchola j, pre každú dvojicu vrcholov i, j. Takýto graf sa nazýva *silne prepojený*.

Matica $\mathbf{M}$ grafu G je *aperiodická*, ak všetky vrcholy grafu G sú aperiodické. Vrchol i je aperiodický, ak najväčší spoločný deliteľ všetkých ciest vedúcich z vrcholu i naspäť do vrcholu i je 1.

7.2 Prehľad algoritmov ohodnocujúcich grafy

Táto kapitola sa zameriava na algoritmy ohodnocujúce grafy založené na iteračnom násobení priamo incidenčných matíc alebo ich transformácií. Iteračný algoritmus násobenie matíc využívaný vo väčšine popisovaných algoritmov je veľmi podobný mocninovej metóde hľadania vlastného vektora matice známej z lineárnej algebry.

7.2.1 Mocninová metóda hľadania vlastného vektora

Nech incidenčná matica $\mathbf{A}$ grafu G je ireducibilná, aperiodická a stochastická, potom vlastný (stacionárny) vektor $\mathbf{r}_\infty$ matice $\mathbf{A}$ je možné vypočítať iteračnou mocninovou metódou definovanou ako

$$\mathbf{r}_k = \mathbf{r}_{k-1}\mathbf{A} \tag{7.3}$$

a platí, že

$$\lim_{k\to\infty} \mathbf{r}_k = \mathbf{r}_\infty = \mathbf{r}_\infty\mathbf{A} \tag{7.4}$$

7.2.2 PageRank

Asi najpopulárnejším algoritmom na ohodnocovanie grafu je práve PageRank. Algoritmus vychádza z rekurzívne definovaného predpokladu, že stránka je tým lepšia, čím lepšie stránky na ňu odkazujú.

Výpočet PageRank ohodnotenia si je možné predstaviť pomocou modelu *náhodného surfera*, ktorý prehliada stránky a náhodne nasleduje odkazy na týchto stránkach. Ohodnotenie jednotlivých stránok je potom ustálené rozdelenie počtu návštev na jednotlivých stránkach. Problémovým miestom sú stránky, ktoré neobsahujú odkazy a cykly v odkazoch. Náhodný surfer by v prvom prípade uviazol na jednom mieste a v druhom prípade dokonca mohol naakumulovať nekonečne veľké ohodnotenie na stránkach v cykle. Tento problém je možné odstrániť zavedením malej pravdepodobnosti opustenia aktuálnej stránky a pokračovaniu na úplne náhodne vybranej stránke [15].

Formálne je možné tento proces zapísať ako

$$\mathbf{M} = (1-d)\frac{\mathbf{E}}{n} + d\mathbf{A} \tag{7.5}$$

kde $d \in \langle 0, 1\rangle$ je tlmiaci faktor, $\frac{\mathbf{E}}{n}$ je matica pravdepodobností skoku na úplne náhodnú stránku a $\mathbf{A}$ je matica pravdepodobností skoku na susedné stránky. Takáto úprava matice $\mathbf{A}$ zaručuje, že výsledná matica $\mathbf{M}$ je ireducibilná a aperiodická.

Mocninovou metódou je pre takto upravenú maticu možné nájsť stacionárne rozdelenie $\mathbf{r}_k$, ktoré predstavuje ohodnotenie stránok (vrcholov) PageRank algoritmom.

$$\mathbf{r}_k = \mathbf{r}_{k-1}\mathbf{M} \tag{7.6}$$

Zaujímavým rozšírením tohto algoritmu je výpočet tematického ohodnotenia stránky, pričom jedna stránka môže obsahovať viac tém [9, 22]. Ukazuje sa, že takéto tematické rozdelenie ohodnotenia stránky je možné využiť na boj proti spamu a dosahuje lepšie výsledky pri vyhľadávaní. Iným prístupom je rozdelenie ohodnotenia stránky na viaceré logické bloky, pre ktoré sa vypočíta čiastkové ohodnotenie [10].

7.2.3 HITS

HITS (*angl. Hypertext Induced Topic Selection*) taktiež patrí medzi najznámejšie algoritmy a je pôvodne určený na zistovanie autority stránok na webe [11, 12]. Vychádza z predpokladu, že stránky je možné ohodnotiť dvoma hodnotami. Prvá z nich označuje mieru autority (*angl. authority*) a druhá mieru rozcestnosti (*angl. hubness*). Tieto dve hodnoty sú zadefinované rekurzívnym vzťahom vychádzajúc z predpokladu, že na kvalitné autority odkazujú kvalitné rozcestníky a kvalitné rozcestníky odkazujú na kvalitné autority.

Nech $\mathbf{A}$ je incidenčná matica grafu odkazov medzi stránkami, potom pre vektor ohodnotenia autorít $\mathbf{a}$ a vektor ohodnotenia rozcestnosti $\mathbf{h}$ musí platiť

$$\mathbf{a} = \mathbf{A}^T\mathbf{h} \tag{7.7}$$

$$\mathbf{h} = \mathbf{A}\mathbf{a} \tag{7.8}$$

Výpočet týchto hodnôt je možný pomocou mocninovej metódy hľadania vlastného vektora matice po úprave vzorcov 7.7 a 7.8 na

$$\mathbf{a}_k = \mathbf{a}_{k-1}\mathbf{A}^T\mathbf{A} \tag{7.9}$$

$$\mathbf{h}_k = \mathbf{h}_{k-1}\mathbf{A}\mathbf{A}^T \tag{7.10}$$

pričom $\mathbf{a}_0 = \mathbf{h}_0 = (1, 1, \ldots, 1)$.

7.2.4 Šírenie aktivácie

Šírenie aktivácie (*angl. spreading activation*) je jednoduchý rekurzívny algoritmus inšpirovaný teoretickým modelom správania sa sémantickej pamäte človeka [18].

Na začiatku algoritmu šírenia aktivácie je aktivovaný jeden vrchol grafu energiou e. Táto energia sa šíri cez hrany vystupujúce z tohto vrchola do susedných vrcholov, pričom

Algorithm 1 ŠÍRENIE-AKTIVÁCIE$(v, e, \mathbf{c} \Leftarrow \mathbf{0})$

Require: Začiatočný v
Require: Aktivačnú energiu $e > 0$.
Require: Doteraz naakumulovanú energiu **c** na vrcholoch grafu.

1: $\mathbf{c}_v \Leftarrow \mathbf{c}_v + e$
2: $e' \Leftarrow e/$STUPEŇ-VRCHOLU(v)
3: **if** $e' > \theta$ **then**
4: **for all** vrcholy t také, že z v do t existuje hrana **do**
5: $\mathbf{c} \Leftarrow$ ŠÍRENIE-AKTIVÁCIE$(t, e', \mathbf{c})$
6: **end for**
7: **end if**
8: **return c**

sa rovnomerne rozdelí medzi nich. Energia, ktorá sa prešíri cez hrany, tak aktivuje susedov, čím sa celý proces šírenia energie rekurzívne opakuje. Zastavovacou podmienkou je prahová hodnota minimálnej energie θ, ktorá sa ešte bude prenášať na susedov. Mierou podobnosti vrcholov grafu je súčet energií, ktorými boli v tomto procese postupne aktivované. V zápise algoritmu 1 je súčet výsledných energií vektor **c**.

Prepísaním algoritmu šírenia aktivácie do maticového počtu dostávame vzťahy

$$\mathbf{r}_k = \tau_\theta \left(\mathbf{r}_{k-1} \mathbf{A} \right) \tag{7.11}$$

$$\mathbf{c}_k = \sum_{i=0}^{k} \mathbf{r}_i \tag{7.12}$$

kde $\mathbf{c}_k$ je výsledný vektor ohodnotenia vrcholov grafu a τ_θ je prahová funkcia s prahom θ definovaná ako

$$\left(\tau_\theta(\mathbf{r}) \right)_i = \begin{cases} \mathbf{r}_i & \text{ak } \mathbf{r}_i > \theta \\ 0 & \text{inak} \end{cases} \tag{7.13}$$

Šírenie aktivácie má veľmi pestré spektrum možností aplikácií. Bolo využité pri poloautomatickom rozširovaní vzťahov v ontológiach [13], obohacovaní kľúčových slov užívateľských dopytov [1], zisťovaní významu slov [19], odporúčaní [3], vyhľadávaní [4, 5, 6, 23] a šírení dôvery [24].

7.2.5 NodeRanking

Ohodnocovanie grafu *NodeRanking* [17] je vo svojej podstate veľmi podobné šíreniu aktivácie a PageRank algoritmu.

Pravdepodobnosť toho, že z vrcholu i sa preskočí na úplne náhodnú stránku je však závislá od počtu $\sigma(i)$ vystupujúcich hrán vrcholu i.

$$\mathbf{J}_{ii} = \frac{1}{\sigma(i) + 1} \tag{7.14}$$

Takáto úprava implicitne zabezpečuje to, že z vrcholu, ktorý nemá žiadne výstupné hrany, sa preskočí na úplne náhodnú stránku s pravdepodobnosťou 1.

Úpravou incidenčnej matice $\mathbf{M}$ grafu, ktorá spĺňa prechádzajúci predpoklad dostávame

$$\mathbf{M} = \mathbf{J}\frac{\mathbf{E}}{n} + (\mathbf{E} - \mathbf{J})\mathbf{A}. \tag{7.15}$$

Výpočet výsledného ohodnotenia grafu je opäť možné realizovať mocninovou metódou.

7.2.6 Model deravého kondenzátora

Model deravého kondenzátora (*angl. leaky capacitor model*) je ďalšou variáciou šírenia aktivácie.

Výsledná energia akumulovaná na vrcholoch sa v tomto modeli neukladá bez strát, ale v každej iterácii sa jej časť definovaná konštantou $\gamma < 1$ stratí [16]. Je zrejmé, že ak $\gamma > 0$, tak model nutne konverguje k stavu, kde sa výsledná aktivácia stratí úplne. Do modelu sa kvôli zamedzeniu tohto nežiadúceho správania v každej iterácii "pumpuje"ďalšia energia $\mathbf{r}_0$. Model deravého kondenzátora je možné zapísať pomocou vzťahov

$$\mathbf{r}_k = \mathbf{r}_0 + \mathbf{r}_{k-1}\mathbf{M} \tag{7.16}$$

$$\mathbf{M} = (1 - \gamma)\mathbf{I} + \alpha\mathbf{A} \tag{7.17}$$

kde α je pomer množstva energie, ktoré sa prešíri na vrchol od susedov. Základný algoritmus šírenia aktivácie (kapitola 7.2.4) je špeciálnym prípadom modelu deravého kondenzátora, kde $\alpha = 1, \gamma = 0$ a do grafu sa energia "napumpuje"len v prvej iterácii.

7.2.7 Porovnanie spôsobov výpočtu ohodnotenia

V tabuľke 7-3 je uvedený prehľad postupov výpočtu ohodnotenia grafu jednotlivými algoritmami. Pre uvedené algoritmy (s výnimkou modelu deravého kondenzátora) je zrejmé využitie výpočtu hľadania vlastného vektora matice iteračnou mocninovou metódou.[2]

V uvedených algoritmoch je z hľadiska výpočtu ešte jeden zásadný rozdiel. Zatiaľ čo algoritmy PageRank, HITS potrebujú na výpočet poznať celú incidenčnú maticu grafu, tak pri algoritmoch šírenie aktivácie a NodeRanking je pre výpočet potrebné iba okolie jednotlivých vrcholov. Druhá skupina algoritmov sa ukazuje byť vhodnejšia na distribuovaný výpočet, pretože nemusí dochádzať k synchronizáciam po každej iterácii.

7.3 Zneužiteľnosť algoritmov ohodnocujúcich grafy

Ohodnocovacie algoritmy sa vo vyhľadávaní a odporúčaní využívajú väčšinou ako dodatočná miera, ktorá v konečnom dôsledku určuje, ako silne je objekt vzhľadom k dopytu používateľa relevantný. V kontexte internetového vyhľadávania stránok je logické, že napríklad stránky firiem sa budú snažiť dosiahnuť vyššiu mieru relevancie ako ich konkurencia. Získajú vyššiu pozíciu vo vyhľadávači a tým pravdepodobne aj viac návštevníkov a potenciálnych zákazníkov.

Dôležitou otázkou v algoritmoch ohodnocujúcich grafy (stránky) sa stáva ich robustnosť voči úmyselnému zneužitiu, ale aj neúmyselnému poškodeniu.

[2] Pre algoritmus HITS sa hľadajú dva vlastné vektory pre matice $\mathbf{A}^T\mathbf{A}$, $\mathbf{A}\mathbf{A}^T$.

Tabuľka 7-3. Porovnanie algoritmov na výpočet ohodnotenia grafu.

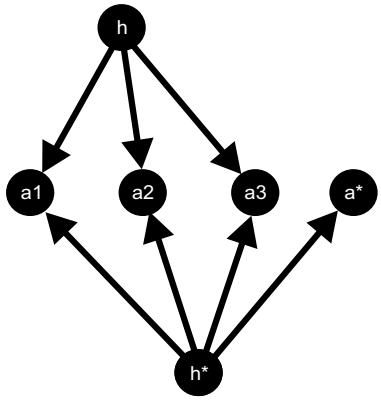
Algoritmus	*Inicializácia*	*Iteračný krok*	*Ohodnotenie*
Mocninová metóda	$\mathbf{r}_0$	$\mathbf{r}_k = \mathbf{r}_{k-1}\mathbf{A}$	$\mathbf{r}_k$
PageRank	$\overbrace{(1-d)\frac{\mathbf{E}}{n} + d\mathbf{A}}^{\mathbf{M}}$ $\mathbf{r}_0$	$\mathbf{r}_k = \mathbf{r}_{k-1}\mathbf{M}$	$\mathbf{r}_k$
HITS	$\mathbf{a}_0, \mathbf{h}_0$	$\mathbf{a}_k = \mathbf{a}_{k-1}\mathbf{A}^T\mathbf{A}$ $\mathbf{h}_k = \mathbf{h}_{k-1}\mathbf{A}\mathbf{A}^T$	$\mathbf{a}_k, \mathbf{h}_k$
Šírenie aktivácie bez prahu	$\mathbf{r}_0$	$\mathbf{r}_k = \mathbf{r}_{k-1}\mathbf{A}$	$\sum_{i=0}^{k} \mathbf{r}_i$
Šírenie aktivácie s prahom θ	$\mathbf{r}_0$	$\mathbf{r}_k = \tau_\theta\left(\mathbf{r}_{k-1}\mathbf{A}\right)$	$\sum_{i=0}^{k} \mathbf{r}_i$
NodeRanking	$\overbrace{\mathbf{J}\frac{\mathbf{E}}{n} + (\mathbf{E}-\mathbf{J})\mathbf{A}}^{\mathbf{M}}$ $\mathbf{J}_{ii} = \frac{1}{\sigma(i)+1}$ $\mathbf{r}_0$	$\mathbf{r}_k = \mathbf{r}_{k-1}\mathbf{M}$	$\mathbf{r}_k$
Model deravého kondenzátora	$\overbrace{(1-\gamma)\mathbf{E} + \alpha\mathbf{A}}^{\mathbf{M}}$ $\mathbf{r}_0$	$\mathbf{r}_k = \mathbf{r}_0 + \mathbf{r}_{k-1}\mathbf{M}$	$\mathbf{r}_k$

7.3.1 Umelé rozcestníky

Zneužitie HITS algoritmu na umelé zvýšenie autority ľubovoľného vrcholu je pomerne jednoduché. Z definície HITS algoritmu je zrejmé, že kvalitný rozceststník odkazuje na veľa kvalitných autorít. Takýto rozcestník je však do istej miery možné urobiť jednoduchou kópiou existujúceho rozcestníka. Do tohto umelého rozcestníka je následne možné vložiť odkaz aj na akýkoľvek iný (aj irelevantný) vrchol, pričom jeho autorita bude vďaka vysokému ohodnoteniu rozcestníka umelo zvýšená [12]. Na obrázku 7-5 je zobrazený umelý rozcestník $h*$ a stránka $a*$, ktorej je umelo zvyšovaná autorita, kde h predstavuje relevatný rozcestník a $a1$, $a2$, $a3$ relevantné autority autority.

7.3.2 Odkazové farmy

PageRank nie je náchylný na takúto formu útoku, pretože nie je možné vytvoriť umelý rozcestník a tak ani umelo zvyšovať autoritu vrcholu, na ktorý takýto rozcestník odkazuje. Ohodnotenie algoritmom PageRanke je však tiež možné umelo zvýšiť vytváraním odkazo-

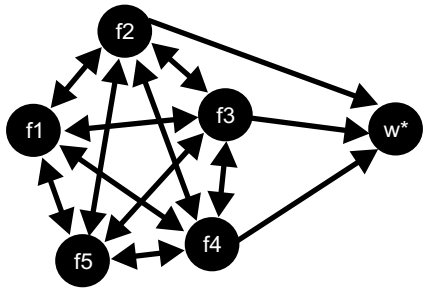


Obr. 7-5. Vytvorenie umelého rozcestníka $h*$ použitím relevantného rozcestníka h smerujúceho na relevantné autority $a1$, $a2$, $a3$ za účelom posilnenia autority vrcholu $a*$.

vých fariem (*angl. link farm*), ktoré automaticky generujú veľké siete prepojení na irelevantné vrcholy a vytvárajú tak ilúziu toho, že je populárny.

Zneužíva sa vlastnosť PageRank algoritmu, kde aj vrcholy, ktoré by nespĺňali podmienku ireducibilnosti, majú určité malé ohodnotenie. Táto vlastnosť je dôsledkom zavedenia náhodného skoku na ľubovoľný vrchol, ktorý zabezpečuje malú váhu každému vrcholu grafu.

Na obrázku 7-6 je zobrazený príklad odkazovej farmy, ktorá umelo zvyšuje ohodnotenie vrcholu $w*$.



Obr. 7-6. Odkazová farma pozostávajúca z piatich vrcholov $f1, f2, ..., f5$ umelo zvyšujúcich PageRank vrcholu $w*$.

Takéto odkazové farmy v kontexte vyhľadávačov sú však pomerne jednoducho odhaliteľné, pretože sú väčšinou až príliš pravidelných tvarov a na takto generované stránky väčšinou neodkazujú iné stránky [2, 7, 8, 20, 21].

7.3.3 Odkazový spam

V kontexte vyhľadávačov vznikla ďalšia forma odkazovej farmy, ktorú nie je možné analýzou štruktúry grafu jednoducho odhaliť. Takýto útok je založený na vkladaní odkazov do už existujúcich stránok (napríklad formou komentárov), pričom odkazovaná irelevantná stránka parazituje na ohodnotení už existujúcich reálnych stránok.

Odhaliť, ktorý odkaz je v kontexte danej stránky relevantný, je z algoritmického hľadiska netriviálny problém. Prvým priblížením je zavedenie tematického ohodnotenia stránok [22], ktoré rozdeľuje výsledné ohodnotenie stránky na viaceré zložky – témy. Zdá sa, že takéto rozdelenie má navyše úplne racionálny základ, pretože autorita v jednej téme by nemala mať rovnako veľký vplyv v téme úplne inej. Pridanou hodnotou tohto tematického rozdelenia ohodnotenia je zvýšenie obrany voči odkazovému spamu. Umelo vkladané odkazy väčšinou nesúvisia priamo s témou stránky, na ktorej parazitujú, čím je váha odkazu znižovaná a následne aj tematické ohodnotenie irelevantnej stránky.

7.3.4 Zneužiteľnosť ako oslabenie predpokladu

Vo všeobecnosti je možné útok na ohodnocovacie algoritmy chápať ako snahu o zneužitie predpokladu, z ktorého algoritmus vychádza.

Napríklad HITS predpokladá, že na kvalitné autority odkazujú kvalitné rozcestníky a kvalitné rozcestníky odkazujú na kvalitné autority. Vytvorením umelého rozcestníka, ktorý odkazuje na nekvalitnú (irelevantnú) stránku, je umelo oslabovaný predpoklad algoritmu. Oslabenie predpokladu má za následok oslabenie dôsledku. V tomto prípade kvalitu ohodnotenia autoritatívnosti a rozcestnosti stránok.

Podobne aj PageRank predpokladá, že čím viac stránok odkazuje na jednu stránku, tým je táto stránka populárnejšia. Odkazová farma umelo zvyšuje počet odkazov na stránky, čím oslabuje predpoklad, že viac odkazov znamená väčšiu popularitu. Dôsledkom je oslabenie kvality ohodnotenia popularity.

Tematický PageRank obsahuje predpoklad, že tematická popularita stránky sa zvyšuje s počtom odkazov rovnako tematicky orientovaných stránok. Novou formou útoku by mohla byť snaha o oslabenie tohto predpokladu, napríklad vytvorením tematicky rovnakých stránok a odkazovaním na tematicky irelevantnú stránku.

Použitá literatúra

[1] Aswath, D., Ahmed, S.T., D'cunha, J., Davulcu, H.: Boosting Item Keyword Search with Spreading Activation. In: *WI '05: Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*, Washington, DC, USA, IEEE Computer Society, 2005, pp. 704–707.

[2] Becchetti, L., Castillo, C., Donato, D., Leonardi, S., Baeza-Yates, R.: Link-based characterization and detection of web spam, 2006.

[3] Berger, H., Dittenbach, M., Merkl, D.: Activation on the Move: Querying Tourism Information via Spreading Activation. In: *DEXA*, 2003, pp. 474–483.

[4] Crestani, F.: Application of Spreading Activation Techniques in Information Retrieval. *Artif. Intell. Rev.*, 1997, vol. 11, no. 6, pp. 453–482.

[5] Crestani, F., Lee, P.L.: WebSCSA: Web Search by Constrained Spreading Activation. In: *ADL '99: Proceedings of the IEEE Forum on Research and Technology Advances in Digital Libraries*, IEEE Computer Society, 1999, p. 163.

[6] Crestani, F., Lee, P.L.: Searching the Web by constrained spreading activation. *Inf. Process. Manage.*, 2000, vol. 36, no. 4, pp. 585–605.

[7] Guha, R., Kumar, R., Raghavan, P., Tomkins, A.: Propagation of trust and distrust. In: *WWW '04: Proceedings of the 13th international conference on World Wide Web*, New York, NY, USA, ACM Press, 2004, pp. 403–412.

[8] Gyöngyi, Z., Garcia-Molina, H., Pedersen, J.: Combating Web Spam with TrustRank. In: *VLDB*, 2004, pp. 576–587.

[9] Haveliwala, T.H.: Topic-sensitive PageRank. In: *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, 2002.

[10] Kamvar, S., Haveliwala, T., Manning, C., Golub, G.: Exploiting the block structure of the web for computing PageRank, 2003.

[11] Kleinberg, J.M.: Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 1999, vol. 46, no. 5, pp. 604–632.

[12] Liu, B.: *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (Data-Centric Systems and Applications)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[13] Liu, W., Weichselbraun, A., Scharl, A., Chang, E.: Semi-Automatic Ontology Extension Using Spreading Activation. *Journal of Universal Knowledge Management*, 2005, vol. 0, no. 1, pp. 50–58.

[14] Maciej Ceglowski, A.C., Cuadrado, J.: Semantic Search of Unstructured Data using Contextual Network Graphs, 2003.

[15] Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the Web. In: *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, 1998, pp. 161–172.

[16] Pirolli, P., Pitkow, J., Rao, R.: Silk from a sow's ear: extracting usable structures from the Web. In: *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, ACM Press, 1996, pp. 118–125.

[17] Pujol, J.M., S.R., Delgado, J.: Chap. A Ranking Algorithm Based on Graph Topology to Generate Reputation or Relevance. In: *Web Intelligence*. Springer Verlag, 2003.

[18] Salton, G., Buckley, C.: On the use of spreading activation methods in automatic information. In: *SIGIR '88: Proceedings of the 11th annual international ACM SIGIR conference on Research and development in information retrieval*, New York, NY, USA, ACM Press, 1988, pp. 147–160.

[19] Tsatsaronis, G., Vazirgiannis, M., Androutsopoulos, I.: Word Sense Disambiguation with Spreading Activation Networks Generated from Thesauri. In: *IJCAI*, 2007, pp. 1725–1730.

[20] Vinay, B.W.: Propagating Trust and Distrust to Demote Web Spam.

[21] Wu, B., Davison, B.: Identifying link farm spam pages. In: *Proceedings of the 14th International World Wide Web Conference, Industrial Track*, 2005.

[22] Wu, B., Goel, V., Davison, B.D.: Topical TrustRank: using topicality to combat web spam. In: *WWW '06: Proceedings of the 15th international conference on World Wide Web*, New York, NY, USA, ACM Press, 2006, pp. 63–72.

[23] Xue, G.R., Zeng, H.J., Chen, Z., Yu, Y., Ma, W.Y., Xi, W., Fan, W.: Optimizing web search using web click-through data. In: *CIKM '04: Proceedings of the thirteenth ACM*

international conference on Information and knowledge management, New York, NY, USA, ACM Press, 2004, pp. 118–126.

[24] Ziegler, C.N., Lausen, G.: Spreading Activation Models for Trust Propagation. In: *EEE '04: Proceedings of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'04)*, Washington, DC, USA, IEEE Computer Society, 2004, pp. 83–97.

8
User Modeling for Personalized Web-Based Systems

Michal Barla

It seems that the fact that the Web should be able to reflect and adapt to user needs is commonly accepted nowadays. In order to do so, adaptive web-based systems need to have some prior knowledge about its users. Acquisition of such knowledge is a task of user modeling. User model can hold user characteristics related to user's interests, knowledge, background etc., according to actual used domain. However, it may take some time to populate correctly the user model of a new user, so the system could provide meaningful adaptation. This pose a risk, that user, unsatisfied by the system, leaves it and goes elsewhere before it can prove its utility. Lack of information about a new user in adaptive systems is called *cold-start* problem.

This chapter presents an overview of a domain of user modeling for web-based system, with focus on sources for user modeling useful for *cold-start* problem solving. The first section describes the basics of user modeling, introduces the modeling process and user model representations, while second section describes various sources for user modeling and discuss them from the *cold-start* problem point of view.

8.1 User Model Representations

Current adaptive web-based systems use mostly the following two types of user models [12]:

- *stereotype model* – is the simplest form of user model, which maps the individual user onto one of (usually) predefined groups. All members of the same group would experience the same "personalized" system. Even simpler form of stereotype model is the *scalar model* [15]. It is usually used to capture the level of user domain knowledge by a single value on a particular scale (e.g., good, average, none), usually set by the user itself prior to using the system. Adaptation is then tailored to individual values of the scale.

- *overlay model* – is currently the most popular user model representation, where the model reflects user characteristics related to individual concepts of a domain model.

> The system based on overlay model thus provides "real" personalization based on characteristics of each individual user.

The advantage of *stereotype model* is its ease of use and relatively simple initialization. User can be assigned a stereotype according to answers of few questions or eventually according to the initial behavior in the system (e.g., based on a rule: "user with a stereotype X will perform Y as a first action"). Stereotypes can be organized hierarchically and initially assigned stereotype can be thus further refined as new information about user become available.

Disadvantage of a *stereotype model* lies in limited personalization possibilities if the system does not concern the user as an individual and stereotypes are rather coarse grained (which is often the case if we define stereotypes manually). This results in "one-size-fits-all" problem of personalization for all users within one stereotype.

An *overlay model* deals with this kind of a problem. It creates a separate user model for each user by creating additional layer with user characteristics above the domain model. A problem of *overlay model* lies in increased demand for system resources as the number of user increase. It is necessary to use a specialized approach in case of a large or even open information spaces. The solution could be to create an overlay only in such parts of information space, which are related to the particular user (which were already visited by the user). Another solution is to create a more loose coupling of user and domain model, where characteristics relate to types and/or attributes of concepts, not the concepts themselves. For example, in the domain of job offers, user model does not hold characteristics related to each and every job offer, but rather characteristics related to attributes of a job offer, which are certainly shared among several job offers.

User model used in adaptive web-based systems can be represented in multiple ways [2]. Representations differ in level of expressivity and flexibility they provide as well as in shareability between several adaptive applications or possibilities of further work with characteristics.

8.1.1 Vector

The simplest way to represent user's characteristic is to represents user's attitudes to all concepts of used domain in multiple vectors, each representing particular information about a user (whether the user visited a concept, understood the concept, liked the concept etc.). However, this solution is applicable only on closed and non-changing information spaces. Such user model representation is easy to implement and work with, but lacks direct shareability feature.

8.1.2 Relational database

Many systems use relation databases [12] to represent user models. Since most of the domain-related information of web-based systems is already stored in a relational database, it is straightforward and easy to add user-related information as well. Compared to the vector representation, it allows us to store attribute-value pairs which are not directly connected to used domain.

Even if use of relational databases usually does not require additional resources and brings several advantages (performance, security, overall mature of used technologies), it

is not necessarily the best option for representing user model in a web-based system. User model must be able to store semi-structured data representing various domain-related user characteristics. Moreover, relational databases currently lack the direct support for reasoning mechanisms.

8.1.3 XML

Another approach to user model representation is to employ XML technologies (e.g., in AHA! system [22]). XML provides good enough expressivity allowing, similarly to relational databases, to store attribute-value pairs as values and attributes of XML tags.

Because building and maintaining user model is a non-trivial task, there is an effort to alleviate the design of adaptive web-based systems and pull the user modeling part out of the system into the form which would allow for its sharing across multiple adaptive applications. Both relational databases and XML-based approaches fail to provide a good-enough support for shareability. Relational databases are platform-dependent and an application needs to know the exact used relational model (database schema) to be able to work with it. While XML is a platform-independent and "web-ready" technology, it requires the definition of a common vocabulary and agreement on syntax to serve for sharing purposes among different adaptive applications.

8.1.4 Ontology

Ontology (in computer science) is defined as an "explicit formal specification of shared conceptualization" [58]. By a term *conceptualization* we mean a formally represented, abstract and simplified view of the world. Because term *ontology* encompasses a whole set of different models with different degrees of semantic richness and complexity [52], we specify that by using a term ontology, we mean a model expressed in OWL language[1], based on RDF[2]. Ontologies expressed in OWL language form ontological layer of the Semantic Web [54] and OWL has a key role in realizing the Semantic Web vision [11].

OWL language can be expressed using XML and thus fulfills the basic preconditions for shareability of models expressed with it. The language defines, along with RDF (more precisely RDF Schema), a way how to describe resources using common dictionaries. The basic concept of the language is a class, which has properties, instances and prescribes relations to another classes (and their instances).

As adaptive web-based systems start to use ontology-based domain and user models, we look for such model expression which would allow for the greatest possible reusability and shareability. Directly shared models facilitates reuse of knowledge about domain and user by multiple applications, but we can not assume that, in such a dynamic environment as the Web is, one common universal model acceptable by all adaptive web-based systems would emerge. That is the reason why OWL language defines means for ontology mapping (`owl:equivalentClass`, `owl:equivalentProperty`).

[1] OWL, Web Ontology Language, `http://www.w3.org/2004/OWL/`

[2] RDF, Resource Description Framework, `http://www.w3.org/RDF/`

Examples of user models represented by ontologies

GUMO. General User Model Ontology (GUMO) [29] is an effort to build commonly accepted user model ontology. GUMO ontology is developed within *Ubiquitous User Modeling*[3] project, which had a significant impact on ontology design. Authors were focused on building relatively broad taxonomy of classes representing concepts of actual user's context (e.g., actual movement, emotional state). Model provides classes for capturing static user characteristics (name etc.) and characteristics such as interests or goals.

However, ontology does not prescribe properties of individual classes nor relations between them. Authors did not employ any conditions and restrictions provided by OWL language. GUMO has a potential to become a *top-level* (*upper*) ontology in the field of user modeling, serving as a point of term unification. Individual specialized ontologies could define mapping of their own classes to classes of GUMO ontology.

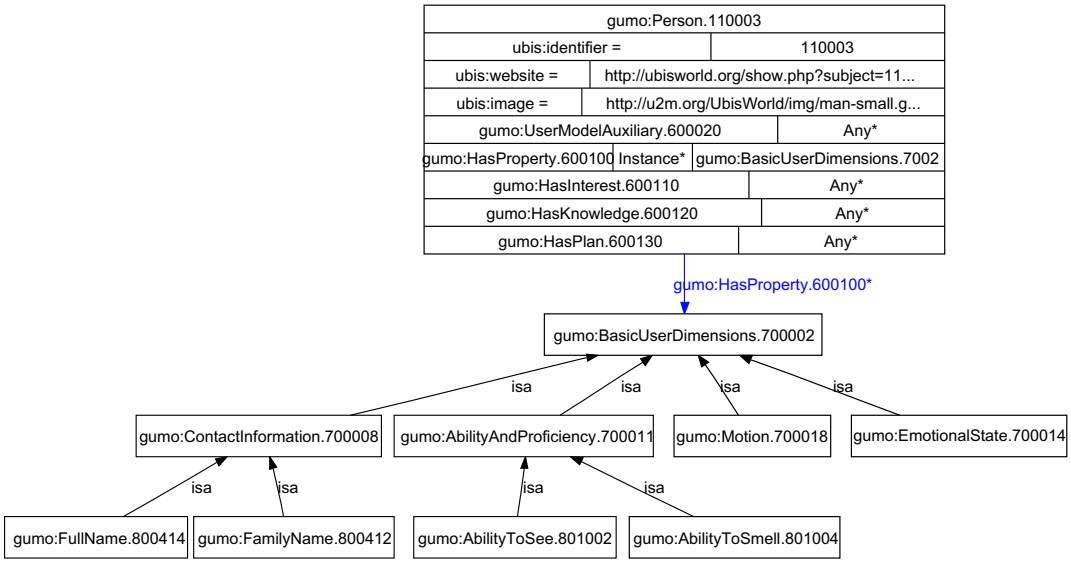


Figure 8-1. Visualization of selected parts of GUMO ontology. Basic concept Person is connected to a broad classification of basic user dimensions.

User model in NAZOU project. Several tools responsible for acquisition, organisation and presentation of knowledge from heterogeneous sources (e.g., the Web) are developed within NAZOU project[4]. Its presentation layer is based on ontological representation of domain data (job offers) as well as user model.

The model is divided into two parts: domain independent and domain specific. Each part defines various types of user characteristics, which contain respectively either generic information about a user or information related to domain content (interests on properties, values of various attributes of job offer). Each characteristic has a *timestamp*, holds a *number of*

[3] Ubiquitous User Modeling, `http://www.u2m.org/`

[4] NAZOU project, `http://nazou.fiit.stuba.sk/`

its updates and a *source* which contributed the characteristic to the model. Each characteristic contributes to certain user's *goal* with certain *relevance*.

Because characteristics stored in a model are only estimations of real user features based on analysis of user's behavior [6], each characteristic has a *confidence*, informing about quality of the estimation.

Complete description of the model can be found at `http://nazou.fiit.stuba.sk`.

User model in OntoAIMS system. Ontology-based Adaptive Information Management System (OntoAIMS) provides environment for navigation in and searching for information. This environment recommends its users (students) the most suitable tasks and helps them in exploration of the domain [23]. System represents the typical example of employment of ontologies representing a domain as well as user model. User model, created and maintained by a specialized component *OWL-OLM*, is forming a layer on the top of a domain model and adds user's perspective to it. This approach is suitable in this case, as the information space defined in domain model (OS Linux course) is closed and static enough. User model is built using an interactive dialogue with a user and contains information about each individual concept like number of concept's usage, number of correct usages, number of times user declared concept as understood etc. The model is consequently used to recommend content.

Conclusions on User Model Representation

Ontology is clearly the most promising alternative to represent user model. The basic ide of ontological modeling is very similar to the way how human think about things in the world. This allows for inclusion of domain experts into to the process of domain and user model creation. OWL language is expressive enough to capture even complex relationships present in reality. Ontologies are the basis for inference mechanisms, which could not only verify model consistency, but are also able to use conditions, restrictions and rules to infer new relationships and knowledge. Advantage of ontologies is its shareability, supported by mapping means provided by OWL.

Disadvantage of ontology based representation is currently the lack of tools and technologies, (editors, mappers, repositories etc.) in the production-grade quality, which would provide the same efficiency of work as it is with relational databases and XML-based technologies.

8.2 User Characteristics

When designing adaptive web-based system (and its user modeling part) we need to consider which features of users will be used for adaptation and thus should be modeled. Five most popular and useful features are [15]: *knowledge*, *interests*, *goals and tasks*, *background* and *individual traits*.

Knowledge is used mainly in adaptive educational systems, which keep track of user's knowledge of the domain as a whole as well as of smaller fragments. Interests (keyword or concept-based) are becoming important as volume of information space increases and new information-oriented systems (encyclopedias etc.) are being developed.

Goals and tasks represents the immediate purpose for a user's work within an adaptive system [15]. It can be a learning goal or immediate information need.

8.2.1 Categorization of User Characteristics

User characteristics being modeled in user model can be categorized using several different points of view.

The way of their acquisition divides user characteristics into those which can be *acquired automatically* and those which can be *acquired only in cooperation with the user*. An example of such information, which can not be gathered automatically in the domain of job offers is the list of all previous user's employments.

Characteristics can be further divided according to their *crispness*. Some characteristics are naturally *fuzzy*, e.g. preferred salary. Whether a salary is acceptable or good depends also on other attributes of a job offer. Sometimes even a user can not easily express the desired salary. However, the user is capable to express a value which is certainly unacceptable (such a low salary can not be compensated by other attributes of a job offer) and a value of total acceptance (this and any higher salary is certainly good). Two mentioned values present borders of a fuzzy set. Some characteristics are on the other hand strictly *crisp*, e.g. a gender.

An important point of view is *domain dependency* of user characteristics. *Domain independent* characteristics describes a user generically by attributes such as age, gender, education etc. These characteristic could be used by multiple applications from different domains. Second group of *domain dependent* user characteristics express user's relations to the concepts of a particular domain (research area in publication domain, preferred profession in job offers domain etc.). Domain dependent characteristics are of little use in other domains, their usage is however not restricted.

Finally, user characteristics can be divided into *long-term* and *short-term*. Characteristics from the former group are more stable, i.e., their change frequency is low enough and are used to represent user's core features and interests. Short-term characteristics are valid only for few sessions, sometimes they can be changed several times during one session.

8.2.2 Representation of User Characteristics

When considering user characteristics, it is important to think about their representation in a user model. Some models are designed to store only the value of a characteristic [21] (boolean value, string etc.), other systems are storing also a reference to a source, which provided a characteristic [36].

Generally, it is suitable to be able to store,apart from its value, some additional information about a characteristic (metadata). Such information could be relevance or confidence of a characteristic [6].

Relevance of a characteristic allows to capture an importance of a characteristic to the user. We can take preferred duty location in a domain of job offers as an example. If this characteristic would have a high relevance, adaptive system should present them in the first result set presented to the user. On the other hand, if a duty location has low relevance, system could omit this attribute when searching for suitable offers.

Confidence of a characteristic is related to the way how the characteristic was acquired. In [33] authors assign the highest confidence to characteristics provided by the user, lower confidence is assigned to inferred characteristics. Even lower confidence can be assigned to such a characteristic, which originates in another adaptive system and was stored in a shared user model.

8.3 User Model Life Cycle – User Modeling Process

User Model life cycle is defined by a sequence of actions performed by an adaptive web-based system during user's interaction with a system. The process is depicted in the Figure 8-2, adopted from [13]. It separates the process into three distinct stages:

1. collection of data about user,
2. user modeling,
3. adaptation (use of user model).

In the first stage, system collects data about user and use them to create and update user model in the second stage. Created user model is successively used in the third stage for the adaptation itself, so the desired effect is achieved.

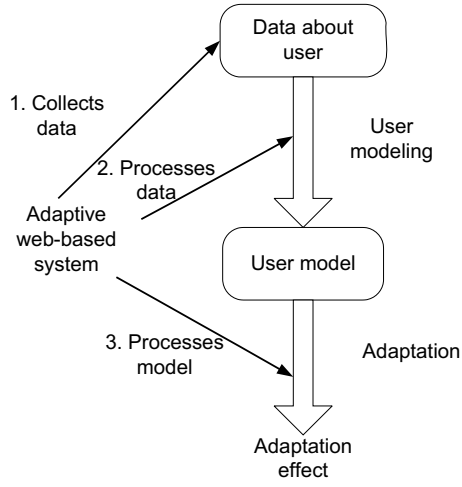


Figure 8-2. User model life cycle according to [13].

The process has a cyclic character, as an adaptive system continuously acquires new data about the user (influenced by already performed adaptation and thus already filled user model) and continuously refines the user model to better reflect reality and thus to serve as a better base for personalization.

8.4 User Modeling Servers

In the beginning of user modeling, it was performed by the application systems, without a clear distinction of user modeling components and components performing other tasks [39]. However, as user model construction and maintenance is a complex process which ends up in a hard-to-develop, complex applications, research was oriented on the creation of *generic user modeling systems*, called also *user modeling shell systems* or more recently *user modeling servers*.

Apart from the primary effect of alleviating application developers from "re-inventing the wheel", designing their own representations of user model, characteristics and modeling

processes, there is also a secondary, but probably even more important effect: the centralized user model gives an opportunity to share the same user model between several adaptive applications, where each one could take advantage of knowledge acquired by others.

8.4.1 CUMULATE

CUMULATE [16, 65] is an example of a generic user modeling server in the domain of e-Learning (so we call it also student-modeling server). It is based on the *KnowledgeTree* architecture [14]. This architecture assumes that server receives information about each important student action from the external adaptive applications. The server processes this information into a student model and provides student information by request to any adaptive applications that wants to adapt to the student.

Information about student is represented in two levels:

- The Event Storage – stores events reported by external adaptive applications. An application can store not only events, but also the student's progress as well as any additional, application-specific information. CUMULATE adds a timestamp to each received event;

- Inferred User Model – the event storage is open to a variety of inference agents that process this data and convert it into a form of attribute-value pairs that form inferred user model.

Authors of CUMULATE created a topic based inference agent which is responsible for topic-based knowledge modeling. CUMULATE provides a form-based authoring interface allowing for definition of topics and their relationships to activities. Implementation of the inference agent that exploits these relationships is then straightforward. However, it is not clear why these relationships belong to CUMULATE and not to the agent itself, i.e., the fact that CUMULATE, being a user modeling server, has to deal with form-based interfaces seems like a flaw in the design.

Another point is that authors do not mention the flexibility of created user model. Does CUMULATE allows for use of an ontology based user model?

8.4.2 MEDEA

Methodology and Tools for the Development of Intelligent Teaching and Learning Environments (MEDEA [60]) is a complex platform in the e-Learning domain allowing for the creation of intelligent systems offering domain-independence, extensibility and resource reusability and interoperability. MEDEA could enhance existing tutoring systems by user modeling and adaptivity feature, under condition that the tutoring system can be encapsulated as a web service that accomplishes a communication protocol. MEDEA is composed of a domain-independent kernel and set of instructional resources (see Figure 8-3).

Instructional resources are external educational systems which perform an individual, well defined pedagogical task (e-books, simulation systems, assessment tools, etc.). Each instructional resource has its own domain model, content authoring interface, student interface and may have its own student model (which contain information to be passed to MEDEA). MEDEA defines three types of instructional resources:

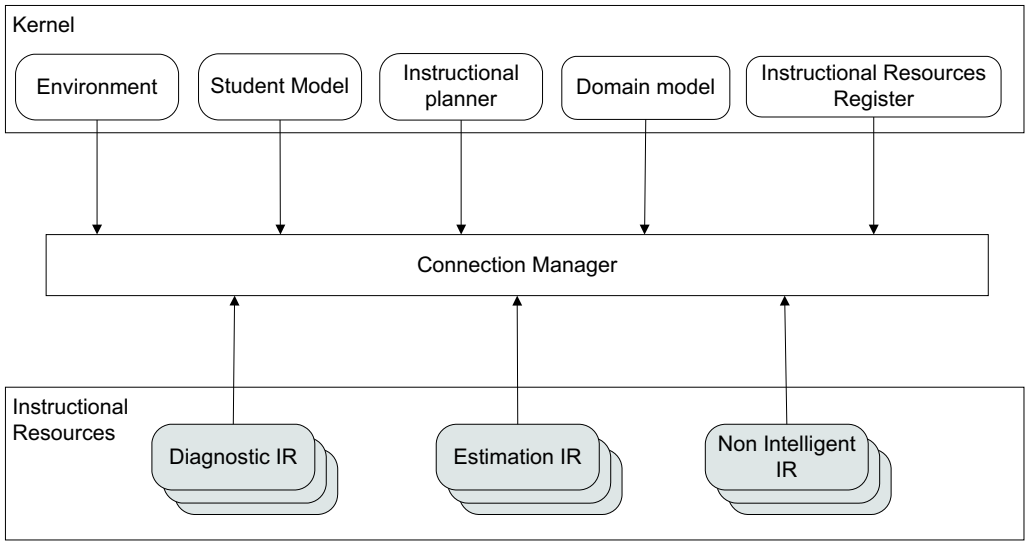


Figure 8-3. The architecture of Medea (according to [60]).

– Diagnostic Intelligent – use a particular diagnostic process to establish the student knowledge level (i.e. a test systems);

– Estimation Intelligent – use heuristics to estimate student knowledge level;

– Non Intelligent – do not have a student model and have no need of inferring it.

Kernel services

The kernel is responsible for guiding a student through the curriculum, and selecting additional educational tasks for domain learning. This functionality is provided by these modules:

– Environment – represents user interface and allows the student to execute pedagogical task, request instructional plan and scrutinize the student model;

– Domain Model – stores knowledge about the subject in semantic network model representation. Domain is modeled for pedagogical purposes and therefore contains only four relations: prerequisite_of and subtopic_of (pedagogical relations) and subconcept_of and part_of (classical ontological relations).

– Student Model – is composed of a Student Knowledge Model and Student Attitude Model. Student Knowledge Model is an overlay model consisting of four layers:

  ◦ Estimated mode – represents system's guesses about student's knowledge based on his or her interaction;

  ◦ Verified model – contains data obtained from evaluative components

    - Inferred mode from the prerequisite relationships and
    - Inferred model from part-of relationships, both based on independent Bayesian networks.

- Instructional Planner – responsible for guidance during learning process (selecting concept to be studied and the most adequate instructional resource).

- Connection Manager – manages the requests and responses between MEDEA's services (actually implemented as Web Services). The integration needs to deal with semantic perspective of the communication problem. Relationships between MEDEA's domain models and its components must be established. Therefore, MEDEA implements methods for automatic type conversion of different representations of student knowledge (binary, discrete, real, etc.).

The student model includes the *student model updating service*, that updates the student model whenever the student performs a pedagogical task (i.e., every time that an instructional resource is executed). When *Diagnostic Instructional Resource* evaluates (accurately) student's knowledge about some domain topics, MEDEA adds this information to the *verified* knowledge model. As *Estimate Instructional Resources* provide evaluation based on student observation, the available information is used to update the *estimated* knowledge model. *Non intelligent Instructional Resources* do not evaluate students in any way, so the *student model updating service* just informs that the student has executed that task.

It is unclear why authors of MEDEA decided to use Ontology eXtensible Markup Language (OXML) to represent domain model instead of widely used RDF/OWL approach. Moreover, authors do not describe this language, so we have no idea how the domain model is expressed. Current version of MEDEA forces authors to manually define mappings between MEDEA domain model and domain model of a particular Instructional Resource. As ontologies are used to represent the domain models, it is possible to apply semi-automated ontology mapping approaches to support authors in the mapping task.

Personis

Personis [37] is another generic user modeling server. It is distinct from others by emphasizing the scrutability of the user model [35].

Figure 8-4 depicts integration of Personis server with individual adaptive systems. Interesting feature is a specific scrutability interface of each system as an addition to the common one. This allows users to explore their models within a context of the particular application. This could simplify user's control of her model, which is used for personalization and thus indirectly significantly improve user's experience with the system.

Each application has its own *view* of the central user model, which decides which data should be accessible to the application. The view is set by the user, allowing her to decide upon availability of her (often) personal data to individual applications, thus preserving her privacy.

User model is represented in triples *component – evidence – source*, where component represents user's attribute, evidence its value and source identifies an application, which contributed this attribute to the user model.

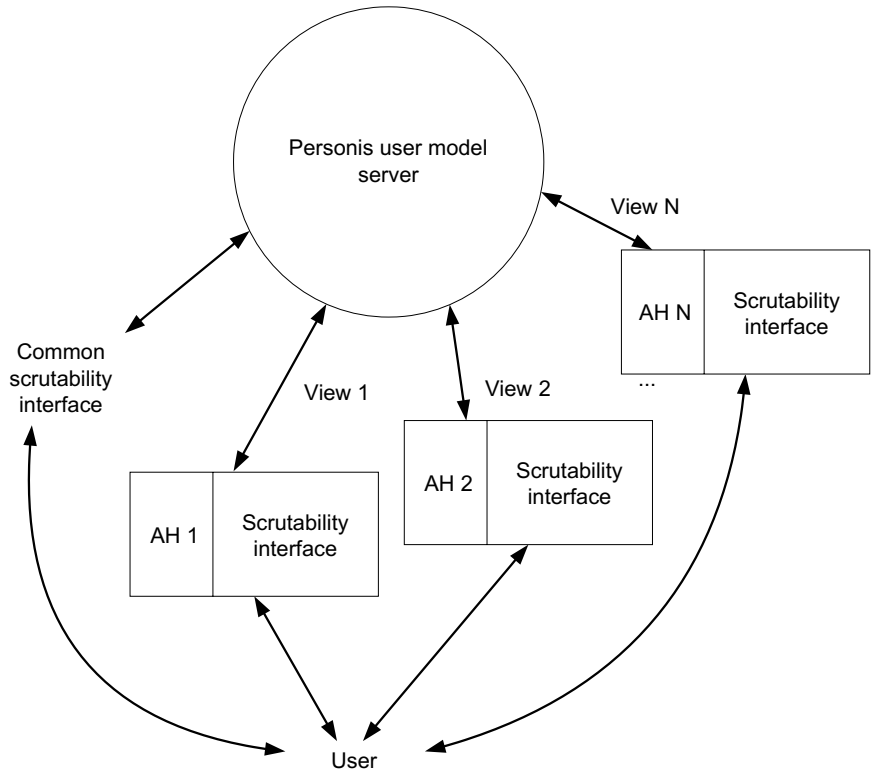


Figure 8-4. Integration of Personis server with adaptive systems, according to [37].

SemanticLog

SemanticLog [5] is a user modeling server based on OWL ontologies. It accepts notifications about occurrence of semantically defined events from multiple adaptive applications (informing about an action performed by an individual user) and stores these events as an evidence. Subsequently, it notifies registered user modeling inference agents about new event to process. After its processing, inference agents may update the user model in a central repository, where it is available to the adaptive applications.

The idea of SemanticLog is similar to CUMULATE. However, the representation of events in the log using semantics based on ontologies allows for better separation (and thus better reusability) of the log processing from the presentation tools and the server itself (responsible mainly for log acquisition and storage). Inference agents work with semantics of the events defined in the shared event ontology and do not need to know about the applications which supplied the events [6].

8.5 Sources for User Modeling

Disadvantage of an overlay user model representation is its complicated initialization. When user logs into the system for the first time, the system does not have enough information to provide efficient personalization (if it does not have any access to user modeling server holding information about the new user). This lack of new user characteristics is called

cold-start problem. However, personalization could be crucial if we want to attract the user and motivate him or her to use the system [61].

In fact, the adaptation is probably of highest benefit exactly to a new user of the system, which is not at all familiar with the domain nor the graphical user interface of the system and can not express clearly her requirements. System should pay a special attention to such user and should try to facilitate her first experience or it could easily happen that the user would get lost, confused and never returns again. Cold-start problem is generally being solved by:

- initializing the model by pre-defined values;
- acquiring needed information directly from the user or from user's behavior;
- using the knowledge from user models of users already present in the system;
- bootstrapping the user model from external sources.

8.5.1 Initializing the Model by pre-Defined Values

This is probably the most simple form of user modeling. For example, a great number of educational systems assume that a new student knows nothing (or that all users have some standard prior knowledge) about the domain and thus initialize the knowledge on all domain concepts to one pre-defined value [61]).

This category encompasses also the stereotype-based approach. According to [34] (as cited in [61]) a stereotype consists of three main components:

1. a set of trigger conditions;
2. a set of retraction conditions;
3. a set of stereotype inferences.

Trigger and retraction conditions are boolean expressions that activate or deactivate an active stereotype. Stereotype inferences of the particular stereotype serve as default assumption for the user, used for adaptation. However, stereotype-based approach is often considered static, meaning that the stereotype is assigned at the beginning of the user's work and is not changing ever after (in fact, the stereotype can be "forgotten" when it is used only for initialization of an overlay user model).

8.5.2 Acquiring Information from the User and from User's Behavior

Approaches to acquisition of information from the user can be discriminated into two categories:

- *explicit*, where user is providing information *explicitly*, by some special type of interaction (usually by filling-in a form);
- *implicit*, where system tracks user activity within the system and use this as a base for estimating and/or inferring the desired information.

Explicit User Modeling

Explicit user modeling consists of letting a user to fill-in a prepared set of forms and is the most basic approach used by the majority of adaptive systems. The system asks the user for all information needed for successful adaptation (or at least information to initialize the user model). This approach is used on almost every system to acquire at least some basic information about a user (see Figure 8-5).

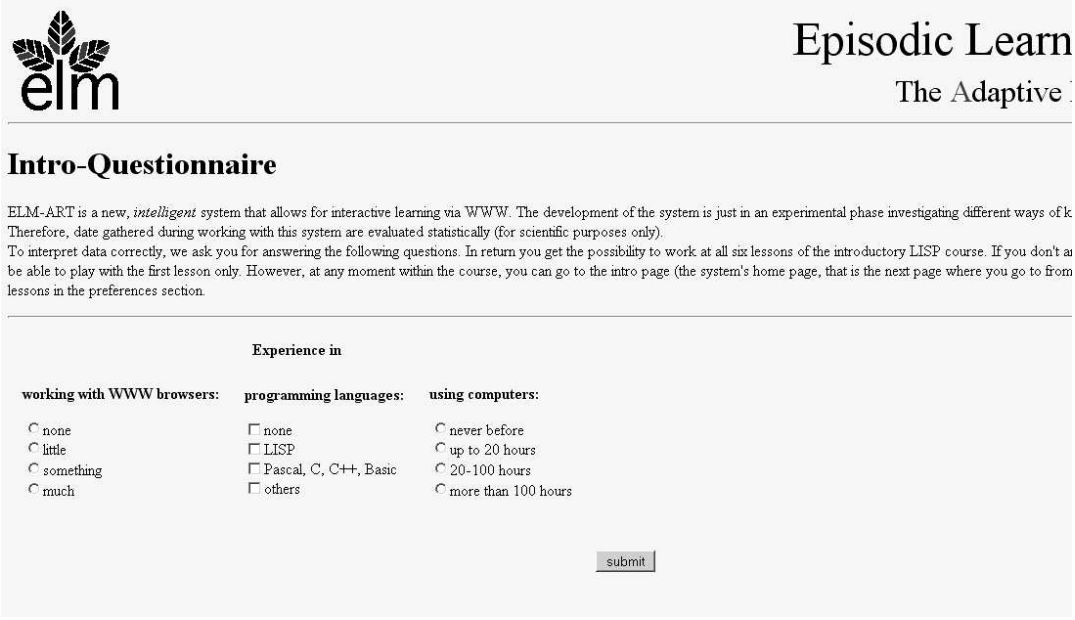


Figure 8-5. Part of a registration form of an adaptive educational system ELM-ART.

The advantage of this approach is the possibility to acquire practically any kind of information about user – even such information which can not be acquired automatically.

It is easy to believe that a substantial advantage is a high credibility of acquired information. It is natural, as the source of information (the user) is, in the same time, the subject the information is related to. However, it can easily happen that a user does not know the exact answer to given questions, she does not know a value of some required characteristic.

Another complication of credibility of information supplied by users comes from subjective point of view when they should evaluate their skills. User can easily overestimate or underestimate the real value of a characteristic (e.g., user might think she has a good knowledge on particular programming language even if it is not true).

Yet another severe drawback of classical form approach is the complexity of graphical user interface. For example to provide preferences on movies, the Internet Movie Database provides a Power Search mode[5] which allows users to combine several criterion to get more precise movie retrievals. For example, users can require certain cast or keywords in the plot summary and filter by genre, year, and other options. The search user interface

[5] IMDB Power search, `http://us.imdb.com/list/`

consists of 17 text fields, 20 choice boxes, 6 check boxes, and requires vertical scrolling before a query can be submitted [31]. Despite the interface complexity, users search preferences are limited in several ways (e.g., only two genres can be combined together). Moreover, form-based graphical interface have difficulties capturing all possibilities including negation, quantification etc. Some believe [31] that a solution could be a natural language processing engine which is communicating with the user in a dialogue form similar to human-to-human communication allowing the user to express any kind of preferences.

Last but not least, it is important to mention that the form-based methods force the users to spend too much time with activities which are not part of the user's goals and are not the reason why the user came to use the system (e.g., filling the forms instead of learning). User might easily get discouraged to use the system, or might lose the concentration etc.

In [23] authors use the explicit user modeling approach in an interesting way. The user model is built with the active user's participation ("interactive user modeling" also called "user model elicitation") in the educational domain. They create a system called OntoAIMS which uses OWL-OLM (OWL based framework for open user modeling) to elicit user (learner) model. It uses a dialogue-based approach where user composes composes statements by constructing diagrams using basic graphical operations such as "create", "delete" or "edit" a concept or a relation between concepts and defining his intention, e.g. to "answer", "question", "agree", "disagree", "suggest topic". OWL-OLM processes these answers to build user's conceptual model based on the domain concepts. When the required aspects of the user's conceptualization have been covered, the dialog can be terminated.

Implicit User Modeling

While explicit acquisition of information can be with no doubt used to alleviate cold-start problem (as user fills-in the form before the actual usage of the system), this is not necessarily the case in implicit acquisition information and thus implicit user modeling. It depends on the speed, how fast the user model converges to something useful, reflecting reality (i.e., how many actions (clicks) and/or sessions are necessary for the system to estimate a correct user model).

Implicit user modeling can be divided into three categories, depending on the complexity of the process:

- Monitoring Access to Resources;
- Monitoring User Feedback;
- Monitoring User Behavior.

Monitoring Access to Resources Users visit the web-based system in order to achieve a goal and what they actually do to achieve it is the formulation (visual or textual) of queries for system's resources and subsequent processing (i.e., reading) of those resources. Web-based system can hold evidence of individual accesses to the resources and provide adaptation upon this evidence.

This approach is used successfully in educational adaptive systems. For example, AHA! system [21] creates two records for each user's access to a page (resource): one record for the

beginning and one for the end of an interval, during which the system presumes that a page was displayed on user's screen. The system uses this information to infer user's knowledge on a related concept(s). The approach presumes that if the page was displayed, the user read it and hopefully understood it. This presumption can not be guaranteed to be true all the time, but the approach is achieving good results anyway, when system is used intensively enough.

Monitoring access to resources can be used also in other domains, e.g., in scientific publication domain we would acquire records about displayed publications. However, the interpretation of these records needs to be more complex than in educational domain. We can presume, that while searching for suitable publication user will find such papers, which are not of her interest. In this case, the simple list of displayed publications does not give a complete information, which could be used in the characteristics discovery process. We need to know the way how the user navigated to the results and/or user's feedback to individual items.

Monitoring User Feedback This approach is an extension of the above mentioned one, which processes user's feedback on the proposed content. We can divide user's feedback according to the nature of its acquisition, in the same way as the general acquisition of information from the user, into explicit and implicit feedback.

Explicit Feedback. This type of feedback is acquired when the user *explicitly* utter her attitude to the displayed content. This is usually done by rating it on a given (often Likert) scale [63].

Explicit feedback is considered to be a good source of user preferences, but one need to design it carefully from the HCI perspective, otherwise the results could be biased. We can apply the HCI key constructs to questionnaires such as *Perceived ease of use*, *Perceived usefulness*, *Disorientation* or *Flow*. These constructs may depend on the presentation (*response format* and *questionnaire layout*) as well as interaction mechanisms that are employed in the administration of scale items [63].

Especially important are *Perceived usefulness* which is defined as the degree to which a user believes that making a rating would give her some benefits and *Flow* which is a psychological state in which a person feels cognitively efficient, motivated and happy. When people are in the state of flow, they become absorbed in their activities and irrelevant thoughts and perceptions are screened out (as cited in [63]). It is important that questionnaires do not break this flow but rather become a part of it.

Implicit Feedback. Because explicit feedback acquisition is not reliable (user oversees the rating widget or is not motivated to rate items) research is oriented also on implicit feedback which is derived from the usage of the system. It is an estimation how would user explicitly rate the particular item.

Observable behavior for implicit feedback is stated in [51]. Authors define three main categories of behavior: *examination*, *retention* and *reference*. The most interesting category, from the perspective of web-based systems, is *examination* which encompasses following behavior:

- Selection – web-based information systems often display a list of results, each with only a brief description. Selection of one particular result for further examination provides information about user's interest.

- Duration – as positive correlation was found between reading time and and explicit rating in USENET applications (as cited in [51]), we can consider duration as another aspect determining user's interests.

- Read wear – extends the duration aspect by including the idea of "computational wear", which informs us about reading history of the document (e.g., which parts were read and which were just scrolled) [30].

- Repetition – repetitive behavior in exploring information space.

- Purchase – represents user's decision to perform some additional action with the item such as purchase, print, add to bookmarks etc. It is a strong indication of a positive feedback.

An example of work with implicit feedback is [17] in which authors were creating a model of a museum visitor to provide personalized reports about a particular museum visit. Each visitor possesses an electronic guide during the visit. Implicit feedback is represented by pressing keys "More" (positive feedback) or "Enough" (negative feedback) during the commentary. By pressing "More", user gets further, more detailed information about particular exhibit. As a side effect, system is informed about user interests. By pressing "Enough", user can interrupt the commentary on particular exhibit (which is interpreted as a negative feedback). Also the fact, that the user did not interrupt the commentary can be considered as rather positive feedback. The "More" button represents *Selection* from the above mentioned classification. Usage of "Enough" button is related to *Duration*.

Implicit feedback was used also in project Casper [56] in the domain of job offers. Authors use three metrics to estimate implicit feedback: re-visiting an offer, time spent reading an offer and further activities related to an offer (applying for a job, sending an offer via e-mail). Focus is given on log preprocessing, to leave out unwanted events such as multiple clicks on one offer, which should not be considered as a re-visit if it was caused by user's impatience or correction of an offer reading time according to mean reading time.

The analysis of user behavior is discussed also in [38]. Authors examine time related information (implicit feedback *Duration*) in an e-Learning course, where students have possibility to listen to the lectures and watch handouts.

Monitoring User Behavior This approach covers the implicit feedback acquisition approach, but is not restricted to feedback related behavior and tracks a complete behavior of the user within a system. The main difference is that the system records as many actions as possible such as the time the user spent viewing a particular page, the way the user navigates on a web site, clickstreams, *list – detail* transactions, page reading (i.e., scrolling down the page), eventually usage of active page element (e.g., *hover*) or additional system's functionality (e.g., adding a publication to Favorites).

The acquired data are not necessarily used to compute a feedback value as it is in the implicit feedback approach. The system could use the data to estimate other types of user characteristics.

The nature of acquired data implies that this approach is especially useful for open and ever-changing information spaces and non-linear navigational model. Many e-Learning courses are "forcing" the user to follow a predefined "optimal" route through the content (usually a linear sequence of pages without any branching). This could lead to uninteresting data of no use for further analysis as user's characteristics and background could not influence user's browsing behavior (as the user follows the route predefined by a course author) [40].

The approach is appropriate considering user involvement into the user modeling process. User is actively using the system (e.g., is taking an electronic course), which is exactly the reason why the user visited the system. Whole data collection and processing tasks are performed in the background, without user's intervention.

Majority of systems collect user behavior data on a server side in a form of log file. This results in lack of records for actions which do not reach the server. A typical example is usage of *back* button in the web browser which does not re-request the page from the server but use the cached copy instead. Server is thus not aware of the exact time user spent by viewing the page. Another example is user interaction with active page elements (e.g., using javascript) which are not communicating with a server.

The data collection part of user modeling (user action monitoring) is independent of the actual modeling process. In fact, it is identical to e.g., data collection for purpose of web sites usability evaluation. In [53] authors note that usability evaluation performed only on the server side is ineffective as it is too coarse grained, not providing enough details. They note that to acquire comprehensive data about web application usage, one need to employ client side monitoring. This was recognized also in the community of adaptive web-based systems [42, 18, 28, 38].

However, client side data availability can not be guaranteed because we have no control on logger execution, as it is on a server side. Therefore, the most suitable approach is to use the combination of the two logging approaches: use a server side logging and complete it with data from client side logging [4].

Software Solutions for Data Collection. Several software tools were developed for the purpose of web sites usability evaluation. Some representatives of such tools are WebVIP[6], WET [26], UAR [59] or Click [5]. WebVip (Web Variable Instrumenter Program) is a tool to augment traditional user testing on a given set of tasks. It allows for setting of actions to be logged, generates Javascript scripts and embeds them into the target pages. The scripts (executed on a client side within user's browser) creates a FLUD (Framework for Logging Usability Data) records. WebVip needs to thoroughly modify every HTML page of a web site and thus needs a static snapshot of a website apriori to its execution, which is obviously a severe disadvantage.

WET (Web Event-logging Tool) is also based on JavaScript technology. When using it, is is sufficient to add a reference to a script file into a header part of every HTML page, so we do not

[6] WebVip, `http://zing.ncsl.nist.gov/WebTools/WebVIP/overview.html`

need to have a static snapshot available as it is with WebVip. WET is based on principle of processing of events generated by web browser. The disadvantage of such an approach could be an overhead of records because what we would define as one user interaction could fire a vast amount of events. E.g., a simple click on a hyperlink could result in a sequence of *mouseover*, *mousedown*, *mouseup* a *click*. To address this issue, WET allows for setting explicitly which types of actions are to be logged. That means that we can trace only clicks, page loads etc.

The problem of WET is its tight connection to the domain of web sites usability evaluation. In fact, it creates an additional layer above the displayed page, providing control buttons to manage the monitoring process. The Click tool (Client side action recorder) is inspired by the logging part of WET (simple reference in a header part, optional types of events etc.) and sends acquired data asynchronously using SOAP messages to the server side, to a specialized web service.

UAR tool (User Action Recorder) is a standalone desktop application for Microsoft Windows environment. It allows for monitoring not only user's work within a web application but whole user – computer interaction by tracking keyboard and mouse usage in individual windows. These practically unlimited monitoring possibilities represents a significant disadvantage of this standalone desktop application approach as many users would feel this as a privacy threat.

8.5.3 Bootstrapping the User Model by Exploting Communities

We, people, are naturally integrated into several social communities (our relatives, friends, colleagues etc.) and are strongly influenced by these communities. If one needs to solve a "never-seen before" kind of a problem, she will very likely ask her friends for some hints and help. We rely on other people if we are in some new and unknown situation, where we are not sure about the most suitable action (e.g., "I will take the same food as others on the conference dinner"). We are behaving socially.

The idea and advantages of social aspects in the real life is being transformed to the web and its applications, namely to the Web 2.0. In fact, the models of social networks are suppressing and replacing the traditional role of user model in web-based systems [3]. It is becoming the community which has ever evolving characteristics and a helpful adaptation can be provided on their basis (the system is performing community-based personalization instead of user-model-based one).

The idea is taking advantage of the fact, that users are humans and humans tend to have a lot in common. Use of social relationships in computer systems is not new. We know collaborative filtering and history-enriched environments [30] as the results of pioneering work from the early 90's [27].

In fact, the *cold-start* problem got its name in the field of recommender systems based on collaborative (social) filtering. User is rating the content and system is using these ratings to compute user's similarity to other users of the system. Then the system is able to recommend new content which got high ratings from similar users. A recommender system can produce good recommendations only after it has accumulated a large set of ratings, which is obviously not the case for the new user, hence the *cold-start* problem arises.

The solution is to combine pure collaborative filtering with a content-based filtering techniques, which selects items based on the correlation between the content of the items and the user's preferences [62] and to bootstrap the user model by other means.

An example of a user model bootstrap based on social relationships can be found in [45]. Authors consider trust relationships between users and its propagation. A trust statement made by user A towards user B means that user A consistently finds the reviews and ratings of user B valuable. Trust statements from all users form a web of trust, or trust (social) network (this trust network is always personalized, from the point of view of an individual node (subject)).

The whole idea is to replace similarity between users (computed according to their ratings of the same item) in the recommendation algorithm by the trust between users (which is propagated as soon as new user defines her trust to one already existing user and is further refined as the user trust or distrust others).

We believe that the approach could benefit from the exploit of classical studies of human social interactions – social network analysis. Its goal is determining the functional roles of individuals in a social network and diagnosing network-wide conditions or states. Different individuals in a social network often fulfill different roles. Examples of intuitive roles include leaders, followers, regulators, "popular people" and early adopters [44]. These roles can influence the computation of the trust in the community and its spreading (e.g., trust of a leader towards a particular user is of greater value than a trust between ordinary users).

As we mentioned, models of social networks could overtake the role of the user model, which shrink to one information expressing membership in a network. We can find similarity of an approach built upon social networks with stereotype-based approach. However, where stereotypes are usually statically defined and users are assigned a stereotype only once, at the very beginning of their work with the system, the communities are usually discovered "on-the-fly", have their proper characteristics and memberships of the user to the community is decided automatically.

An example of a system which employes community-based personalization is *Community-based Conference Navigator* [27]. It uses social navigation support to help conference attendees to plan their attendance at the most appropriate sessions and make sure that the most important papers are not overlooked.

The system provides additional service as an adaptive layer over AACEE conference planning system, which allows the conference attendees to browse the conference program and plan their schedule. *Community-based conference navigator* tracks different activities of the community (such as paper scheduling) and allows users to add comments to papers. All activities result in updates of the community profile, which accumulates over time the "wisdom" of the community. The community profile and "wisdom" is used in the adaptation of the original system by adding adaptive icon annotations.

The selection of the community is done manually by each user. If the user does not find the suitable community, she is allowed to create a new one. Moreover, user can switch between communities anytime during the usage of the system, which gives instantly the annotations for a different community. However, it seems that user can act only as a member of one community at a time, so all actions contribute only to one community profile. However, many people belong to several communities and act as "bridges" between different communities, so it would not be easy for them to choose strictly one. It would be interesting to combine the community-based adaptation with the traditional personalization based on user model (which can nevertheless provide detailed characteristics).

The approach could also benefit from the "classical" social network analysis. Papers scheduled by somebody who is considered as an authority in the network could be considered as more interesting to see (so they could have yet distinct annotation so the user does not overlook them).

The ISM framework [61] is a mix of several approaches to cold-start problem solving: it employs explicit user modeling by letting the user to pass an interview as well as a pre-test. The knowledge acquired from the user is used to assign her a stereotype. However, this stereotype is not used directly to initialize the student model. The default assumptions of each stereotype are refined by taking into account the actual behavior of the other students belonging to the stereotype. Moreover, the contribution of these students to the initialization of the new student model is weighted based on their similarity with the new student [61].

The information acquired from both the interview and the preliminary test (see Figure 8-6) is represented as a feature vector, which contains student code, name, stereotype and a set of N characteristics (student's attributes). The initial information that has been acquired directly from the student, as well as information from existing students is then used in order to produce a second vector that represents the system's estimations of certain domain dependent attributes of the new student. The second student model vector contains student code and a set of M domain-related characteristics. While the first vector contains information such as mother tongue or level of carefulness, the second vector contains information referring to domain concepts: the degree of knowledge and error proneness for each concept.

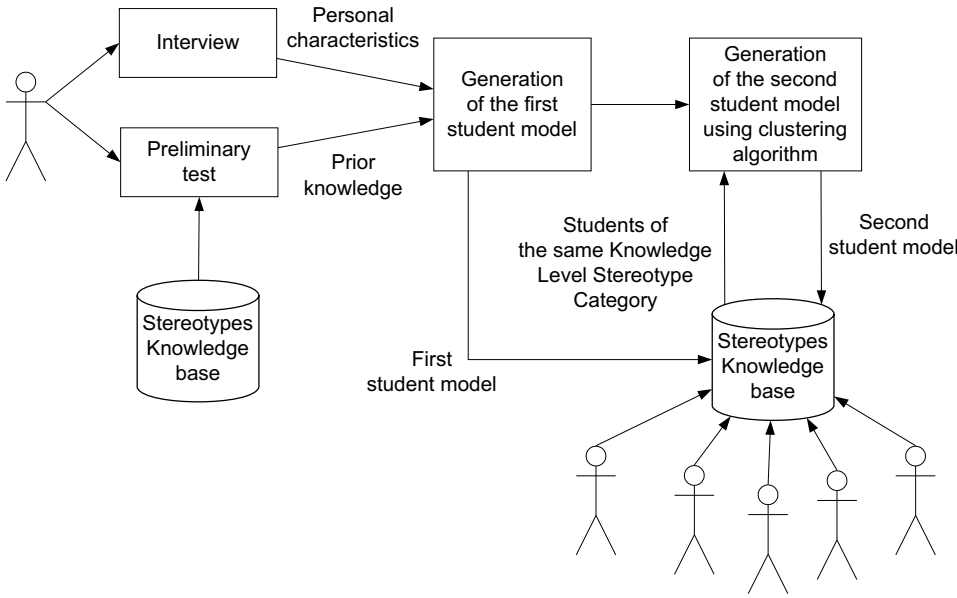


Figure 8-6. ISM Architecture, according to [61].

The second vector is computed using the distance weighted k-Nearest Neighbors (k-NN) algorithm. The actual number of neighbors is the number of students within a stereotype. The system presumes, that users belonging to different stereotypes do not have similar knowledge of the domain.

The actual classification function takes the new user instance (which should be classified) and the k nearest neighbors. The value of the degree of knowledge and error proneness of the new student (s_q) is computed as a distance weighted mean value of the degree of knowledge and error proneness of the k students that belong to the same stereotype with the new student $(s_1, s_2, \ldots, s_k)$. So the knowledge level for each concept is computed as:

$$KnowledgeLevel(Concept_x, s_q) = \frac{\sum_{i=1}^{k} w_i KnowledgeLevel(Concept_x, s_i)}{\sum_{i=1}^{k} w_i} \tag{8.1}$$

where w_i is the weight of the contribution of each student and is calculated as an inverse square of its distance from s_q:

$$w_i = \frac{1}{\Delta(s_q, s_i)^2} \tag{8.2}$$

8.5.4 Acquiring Information from External Sources

In [47] authors propose the idea of a cold-start problem solving by providing some initial knowledge about users and their domains of interest from shared ontologies. It should thus be possible to bootstrap the initial learning phase of a recommender system with such knowledge. They created the Quickstep recommender (Figure 8-7) which implements the idea in the domain of scientific publications.

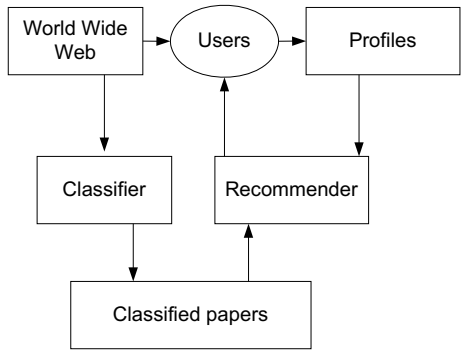


Figure 8-7. The Quickstep recommender system according to [47].

The ontology used to bootstrap the recommender was developed by Southampton's AKT team (Advanced Knowledge Technologies). It models people, projects, papers, events and research interests, was populated with information extracted automatically from a departmental personnel database and publication database. The ontology consists of around 80 classes, 40 slots, over 13000 instances [46].

When new user arrives, the recommender system retrieves an initial set of her publications from the ontology. These publications are then correlated with the classified paper database and a set of historical interests are compiled for that user. These historical interests form the basis of an initial profile, overcoming the new system cold-start problem. Moreover, system is trying to find out communities of practice for the current user (i.e., group of people who share some common interest in a particular practice – a social network) in the ontology

and provides a ranked list of similar users from the very beginning of user's work with the system (so the system is able to provide also collaborative filtering based on users similarity). During the usage of the system, user browsing behavior is monitored via a proxy server, where each URL browsed during normal work activity is logged. A classifying algorithm is used to classify browsed URL's based on a training set of labeled example papers, storing each new paper in a central database. The profiling algorithm performs correlation between paper topic classifications and user browsing logs. Whenever a research paper is browsed that has been classified as belonging to a topic or an explicit feedback on recommendation is provided, it accumulates an interest score for that topic. This is the way Quickstep contributes back to the bootstrap ontology.

The weak point of the approach is the assumption that ontology used for bootstrap contains knowledge about the new user and that this user has already written some publications which could be compared with publications in the information space.

User Model Mediation

The concept of *mediation* of partial user models between several adaptive systems was proposed in [8]. The basic idea is that different applications would benefit from enriching their UMs through importing and aggregating user models which were built by other, possibly related applications. It is different from the centralized user model approach using user modeling server, where each application extracts the required data from the central UM and updates it later. The described approach is designed to be decentralized and providing ad-hoc (for a specific purpose) generation of user model for the target application through translation and aggregation of partial UMs built by other applications. It is therefore useful for solving cold-start problem.

The mediation to be successful needs to solve several issues among which is user's privacy, the structural heterogeneity and incompleteness of the user models content, since every application refers to a specific application domain only.

The mediation process is partitioned to the following stages (see Figure 8-8):

1. A target application, required to provide personalization to a user, queries the mediator for the user model related to the application domain.

2. The mediator identifies the required personalization domain and the user model representation in the target application.

3. The mediator determines a set of other applications that can potentially provide partial domain-related user models of the given user and queries them.

4. Applications, actually storing the needed data, answer the query, and send to the mediator their partial user models of the given user.

5. The mediator translates and aggregates the acquired partial user models (using the knowledge base (KB)) into a single domain-related user model, represented according to the target application.

6. The generated domain-related user model is sent to the target application, which is capable of providing more accurate personalized service.

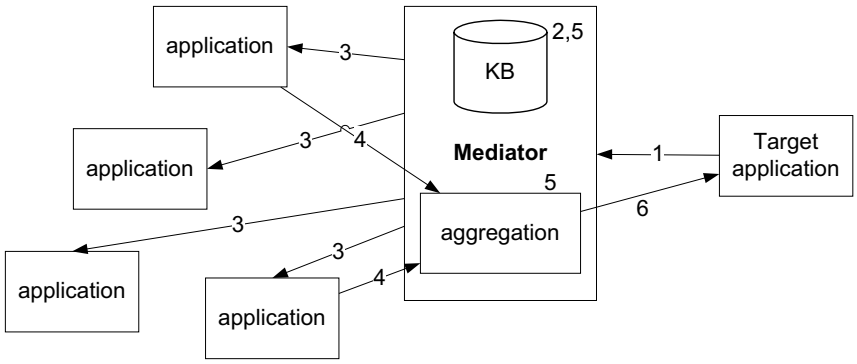


Figure 8-8. Architecture and stages of the user model mediation, according to [8].

For the purpose of mediator, the user model is considered as an aggregation of partial domain-related user models: $UM = aggr(UM_1, UM_2, \ldots, UM_k)$. The domain-related user model is then defined as an aggregation of domain-related user models built by applications exploiting different personalization techniques: $UM_d = aggr(UM_d^1, UM_d^2, \ldots UM_d^n)$, where UM_d^t denotes the partial UM referred to application domain d, built by an application exploiting personalization technique t.

Authorse can therefore divide all applications, which could potentially provide helpful partial user models into three distinct groups:

- applications from d that also exploit t,
- applications from d that exploit another technique t',
- applications from another, relatively similar, domain d' that also exploit t.

Applications are apriori organized in a hierarchical semantically demarcated structure, where upper level of the hierarchy represents different application domains. The domains are represented by the nodes of an undirected graph, where the weights of the edges reflect the similarity between the respective domains. The bottom layer represents specific applications within the domains, grouped according to the personalization techniques they exploit.

The actual translation and aggregation of the acquired partial user models is driven by a rich inter- and intra- domain knowledge base that allows for identification of commonalities between partial user models.

Three distinct types of translations are defined:

- simple concatenation of partial user models.
- cross-technique translation [9] – for example from collaborative to content-based movie recommender. The translation exploits a KB of movies data (e.g., genres, directors and actors), which allows the mediator to generalize a set of collaborative ratings into the content-based user model, containing a list of genres, directors and actors liked/disliked by the user.

– cross-domain translation [10] – for example from book recommender to movie recommender. The translation exploits a KB of books and movies genres that facilitates the translation through identifying the correlations between the contents of the user models (e.g., liked/disliked genres, common to movies and books).

The authors did not address completely the issue of actual aggregation of partial user models. We believe, that this step could be facilitated by exploiting OWL ontologies and its inherent mapping features, which allows for definition of equivalency between classes and properties. In [57] we can find a preliminary work on using ontological cultural user modeling to overcome the cold-start problem, however authors are more focused on the ontology itself rather than on the mapping problem.

Ontology Mapping Probably the most natural way of solving cold-start problem for ontology-based systems is to exploit user model ontologies published on the web. Ontologies were conceived to do so and sharing is one of their main benefits. However, in order to use data from other ontology, the system needs to "understand" them, i.e. it has to put it in relation with its own ontology – ontology mapping needs to be done manually or (preferably) automatically.

It is a process whereby two ontologies are semantically related at conceptual level, and the source ontology instances are transformed into the target ontology entities according to those semantic relations. This results in three dimensions of ontology alignment:

– Discovery – manually, automatically or semi-automatically defining the relations between ontologies

– Representation – a language to represent the relations between the ontologies

– Execution – changing instance of a source ontology to an instance of target ontology

Another point of view takes into consideration the types of mapping process:

– Mapping between an integrated ontology and local ontologies – ontology mapping is used to map a concept found in one ontology into a view, or a query over other ontologies (e.g. over the global ontology in the local-centric approach, or over the local ontologies in the global-centric approach) [19].

– Mapping between local ontologies – the process transforms the source ontology entities into the target ontology entities based on semantic relation. The source and target are semantically related at a conceptual level. This is the mapping which is (or is to be) used on the Semantic Web, because of its de-centralized nature.

– Ontology mapping in ontology merge and alignment – the process establishes correspondence among source (local) ontologies to be merged or aligned, and determines the set of overlapping concepts, synonyms, or unique concepts to those sources. This mapping identifies similarities and conflicts between the various source (local) ontologies to be merged or aligned [19].

Glue GLUE is a system that employs machine learning techniques to find ontology mappings on the Semantic Web [24]. Given two ontologies, for each concept in one ontology, GLUE finds the most similar concept in the other ontology. It calculates joint probability distribution of the concept using multiple learning strategies, each of which exploits different type of information present in data instances and ontology schema.

GLUE consists of *Distribution Estimator*, *Similarity Estimator*, and *Relaxation Labeler*. The *Distribution Estimator* takes as input two taxonomies O_1 and O_2 together with their data instances and compute the joint probability distribution for every pair $\langle A \in O_1, B \in O_2 \rangle$ of concepts, i.e. it computes $P(A, B)$, $P(A, \neg B)$, $P(\neg A, B)$, $P(\neg A, \neg B)$.

To achieve it, it uses three distinct learners:

- *The Content Learner* – takes into account the frequencies of words in the *textual content* of an instance.
- *The Name Learner* – makes predictions using the *full name* of the input instance, instead of its *content*.
- *The Meta Learner* – combines predictions of individual based learner via a weighted sum.

Results of *Distribution Estimator* are passed to *Similarity Estimator*, which applies a user-supplied similarity function to compute a similarity for each pair of concepts

$$\langle A \in O_1, B \in O_2 \rangle .$$

The *Relaxation Labeler* takes the similarity matrix and domain-specific constraints and heuristics to find the best mapping configuration.

GLUE was evaluated on several real-world domains and proved to accurately match 66-97 % of the nodes. The disadvantage of the approach is that it operates upon taxonomies, which are only a subset of ontologies.

Mafra MAFRA [43] stands for Ontology MAapping FRAmework for distributed ontologies in the Semantic Web. The framework consists of five horizontal modules describing the fundamental phases of a mapping process. Four vertical components run along the entire mapping process, interacting with horizontal modules.

The horizontal dimension consists of following modules:

- Lift & Normalization – assures that all data to be mapped are at the same representation level. It copes with with syntactical, structural and language heterogeneity. Elimination of syntax differences makes semantics differences between the source and the target ontology more apparent.
- Similarity – establishes similarities between entities from the source and target ontology, thus, it supports mapping discovery. MAFRA adopted a multi-strategy process that calculates similarities between ontology entities using different algorithms (*lexical similarity*, *property similarity*, *bottom-up similarity* and *top-down similarity* [43]).

- Semantic Bridging – similarities computed in the previously described phase are used in the semantic bridging phase to establish correspondence between entities from the source and target ontology, so that each instance represented according to the source ontology is translated into the most similar instance described according to the target ontology. It is performed in five steps:

    1. *Concept bridging step* – selection of pairs of entities to be bridged, according to the similarities found in previous phase. The same source entity may be part of different bridges.
    2. *Property bridging step* – specification of the matching properties for each concept bridge.
    3. *Inferencing step* – specification of bridges for concepts that do not have a specific counterpart target concept.
    4. *Refinement step* – improve quality of bridges between a source concept and sub concepts of target concepts.
    5. *Transformation specification step* – intends to associate a transformation procedure to the translation, in a way that source instance may be translated into target instances.

- Execution – actually transforms instances from the source ontology into target ontology by evaluating the semantic bridges defined earlier. It could operate in two distinct modes:

    - *offline* static one time transformation) and
    - *online* dynamic, continuous mapping between source and the target.

- Post-processing – takes the results of the execution module to check and improve the quality of the transformation results.

MAFRA is not a fully automated solution and requires a domain expert to drive the creation of semantic bridges. This requires an extensive graphical support, as deep understanding of conceptualizations on both sides (source and target ontology) is required on human side. Moreover, it is not clear whether a generic semantic bridges could lead to fully automatized solution or it would be useful to define domain-dependent bridges, which contain domain specific knowledge.

Information Extraction from Available Documents

Many approaches are employing user models to provide personalized information extraction (IE) which would significantly impact the web search experience [25, 1]. There are also efforts to use IE techniques to retrieve information about users, but usually for other purposes than to populate user model with data suitable for personalization. We can find approaches that are trying to automatically identify social networks around a particular user. We discussed in the section 8.5.3 that resulting social network can be used to bootstrap user model.

The IE techniques could be applied to user-supplied documents (such as CV, from which a system could deduce user's age, education or previous employments [41] or scientific paper the user submitted to a particular conference, from which a system could deduce user's domain of interests and further focus within the domain) or documents found on the web, where it can be inspired by classical IE techniques.

User Information Extraction

Web Appearance Disambiguation. Paper [7] proposes the Web appearance disambiguation methods (*Link Structure Model*, *Agglomerative/Conglomerative Double Clustering* and their combination) and uses a social networks of users as a background knowledge. The Web appearance disambiguation in general is inferring a model that ultimately provides a function f answering whether or not a Web page d refers to a particular person h, given a model M and background knowledge K.

Authors attempt to use as little background knowledge as possible and decided user's social network to be such knowledge. Therefore, instead of solving one problem, they solve N interrelated problems: for each person h_i in the group H (a group of people $H = \{h_1, \ldots, h_N\}$ who are related to each other), they find Web pages that refer to h_i. The group of people was defined manually based on e-mail correspondence.

The basic idea of *Link Structure Model* is that Web pages of a group of acquaintances are likely to be interconnected, while pages of their namesakes would not. However, the existence of a direct hyperlink from one relevant page to another may be rare. Two pages can be considered as linked if both contain a hyperlink to the same page, or both are hyperlinked from one page, or one page can be reached within three hyperlink hops from the other. Yet another approach can also be considered, for example, two pages are linked if both mention the same organization. Authors decided that for their purposes, two Web pages are *linked to* each other if their hyperlinks share something in common.

Their set of Web pages D is constructed by providing a search engine with queries $t_{h_1}, \ldots t_{H_N}$ (where t_{h_i} is a name of person h_i in user's h social network) and retrieving top K hits for each one of the query, so that $N \times K$ Web pages are retrieved overall. Every page d is already associated with a personal name t_{h_i}, however, it is yet unknown whether the page d refers to the actual person h or to his/her namesake (or to neither).

Based on a set D, the model M is constructed. Authors defined a *Link Structure Graph* over a set of Web pages D as $G_{LS} = (V, E)$ if nodes of the graph are the Web pages ($V \equiv D$) and there exists an edge between any pair of nodes d_i and d_j iff d_i and d_j are *linked to each other*. Than the the *Link Structure Model* M_{LS} is defined as a pair (C, δ), where C is the set of all connected components of the graph G_{LS} (note that $C_0 \in C$, where C_0 is the *central cluster*, the largest connected component in G_{LS} that consists of pages retrieved by more than one query) and δ is a distance threshold.

Finally, the discrimination function is defined:

$$f(d, h|M, K) = \begin{cases} 1 & \text{if } d \in C : \ \parallel C_i - C_0 \parallel \ < \ \delta, i = 0..M \\ 0 & \text{otherwise} \end{cases} \tag{8.3}$$

The intuition behind this definition is that the pages of the central cluster and of a few clusters that are close to the central cluster are considered to be relevant, while others are irrelevant.

Paper [64] similarly deals with ambiguity problems in person information mining on the Web. Authors propose five distinct features and a cascaded multiple-clusterer approach for name disambiguation using personal titles, community chains, contextual terms, temporal expressions and hostnames.

Web Object Extraction. Web object extraction/retrieval is a new approach to information retrieval on the Web being invented at Microsoft Research. Current search engines are working at a document-level, ranking documents by their relevance to a set of keywords (query). However, these documents embed various kinds of objects along with their attributes such as people, products, papers[7], organizations, etc. Web object extraction is aiming at extracting such objects to create an object-level vertical search engines (specialized on a particular domain). Such a search engine gives a list of object with explicit properties instead of list of URLS, which costs user's significant efforts to decipher for needed information [49]. Moreover, authors of web object extraction method deal also with integration of the same object retrieved from multiple sources into one "real-world" object.

Figure 8-9 depicts architecture of scientific papers extraction system. It is able to extract four types of objects (papers, authors, conferences and locations) and relationships between them. The architecture follows the method, where web crawler and classifier automatically collect all relevant webpages/documents that contain object information for a specific vertical domain. The crawled webpages/documents will be passed to the corresponding object extractor for extracting the structured object information and building the object warehouse. The task of aggregators is obvious: they aggregate information about the same object from multiple different data sources.

The key point is object extraction itself. The problem is that webpages are generated by tens of thousands of different templates. One possible solution is to distinguish webpages generated by different templates, and then build an extractor for each template (called template-dependent solution). This solution is of no use in real world applications: firstly, it is practically impossible to collect all possible templates (even webpages from the same website may be generated by several different templates0. Secondly, it would be impossible to train and maintain of all required extractors for each template.

Authors in [50] conducted an analysis of webpages across web sites and extracted some template-independent features from it:

- Information about an object in a web page is generally grouped together as an object block, which can be further segmented into object elements, providing information about individual attributes.

- Strong sequence characteristics exist for web objects of the same type across different web sites. For example, a person's name is always ahead of contact information (telephone, postal address) in all the pages.

Based on template-independent features authors propose template-independent method of Web Object Extraction based on linear-chain Conditional Random Fields (CRFs) and achieved

[7] A working object-level search engine for scientific papers can be found at `http://libra.msra.cn/`.

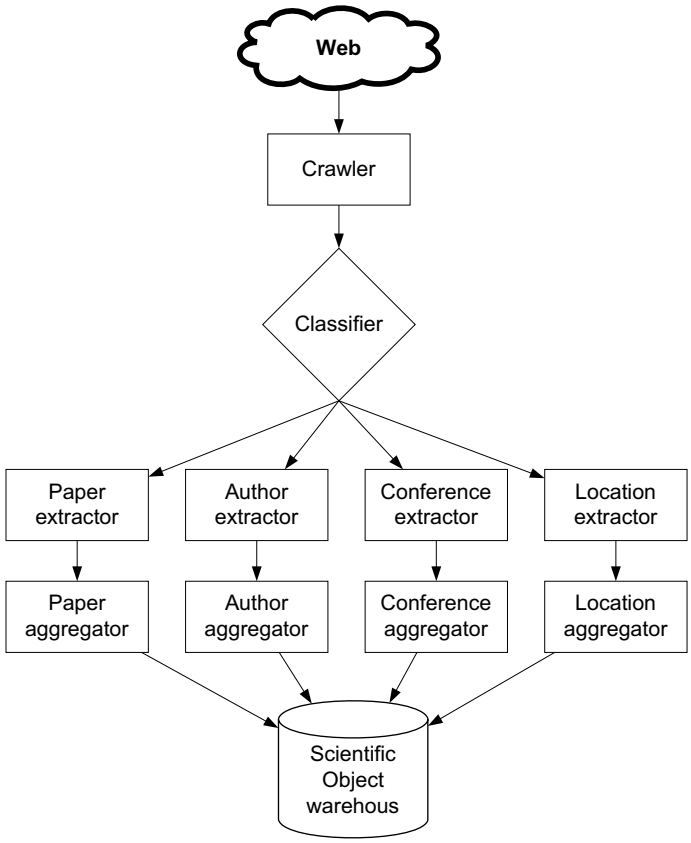


Figure 8-9. Architecture of Scientific Papers Extraction System (according to [50]).

good results in scientific papers domain. If applied to multiple domain, the system could possibly provide enough information about user to overcome the *cold-start* problem.

Social Networks Discovering The paper [55] proposes the idea, that each recommender system naturally fosters communities of users. It is exactly the community which drives the recommendation once enough users are engaged and modeled (see Figure 8-10).

However, as an implicit user modeling does not make the social network identification salient, authors propose to employ other techniques to discover social networks:

– *Link Analysis and Cyber-Communities* – evidence of community existence is often implicit in data, such as communications logs and webpages. These are fertile reflections of natural connectivity among people. Some recommender systems require users to create and maintain profiles. On the other hand, approaches which model people connections or social organization result in representations which are likely to be more accurate reflections than a user's perception of his own connections [55].

– *Mining and Exploiting Structure* – social networks also can be formed by applying transformations on other, typically bipartite, graph representations identified in datasets.

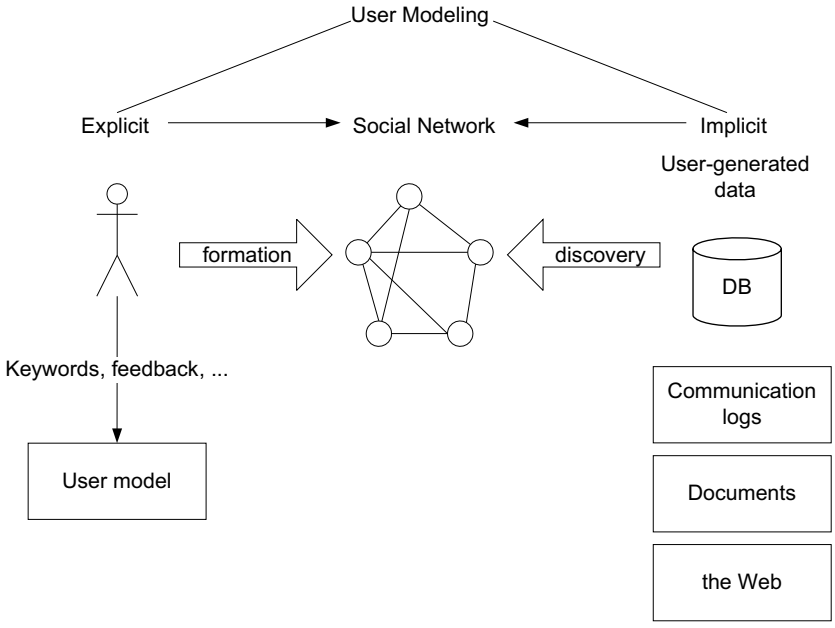


Figure 8-10. Formation of a social network by explicitly collecting ratings or profiles (left). Identification and discovery of a network by exposing self-organizing communities implicit in user-generated data such as communication or web logs (right), according to [55].

Typically, a ratings dataset can be modeled as a bipartite graph rather than a matrix (bipartite graph is often called *affiliation network* in social network theory). For example, if we consider an affiliation network formed by people and publications they (co)authored, we can bring people together via their relationships with publications.

– *Small-World Networks* – are networks naturally modeling the small-world phenomenon (a Harvard social psychologist Stanley Milgram conceived an experiment in the late 1960's which revealed that any two randomly picked individuals residing in the US were connected by no more than six intermediate acquaintances and thus that a human society is organized as a small world type network). According to [55], small-worlds present opportunities for recommender systems. If identified, not only do they help model users and communities implicitly by revealing social structure (via the structure of the connections between documents which members of those communities manually created), but also help connect people via short chains. For example, if search engines could take advantage of the web's small-world property, then users with only local knowledge of the web may actually be able to find and construct short paths between pairs of web pages.

ReferralWeb. An example of link analysis approach can be found in [32]. Authors propose a system which supports users in searching for a piece of information by searching the social network for an expert on the topic together with a chain of personal referrals from the searcher to the expert. The social network is constructed automatically, querying the Web.

The approach uses the co-occurrence of names in close proximity in any documents publicly available on the Web as an evidence of a direct relationship. Such sources include:

- Links found on home pages
- Lists of co-authors in technical papers and citations of papers
- Exchanges between individuals recorded in netnews archives
- Organization charts (such as for university departments)

The network model is constructed incrementally. When a user first logs into the system, it uses a general search engine to retrieve Web documents that mention him or her. The names of other individuals are extracted from the documents. Authors claim that they achieve a high degree of accuracy (better than 90%), using information extraction techniques, however, they do not provide additional details. The process is applied recursively for one or two levels, and the result merged into the global network model.

Jumping Connections. Paper [48] presents an example of *Mining and Exploiting Structure* approach. It describes a study algorithms for recommender systems from the perspective of the combinations of people and artifacts that they bring together. They named the approach *jumping connections*.

A recommender dataset R consists of the ratings (e.g., of movies) by a group of people. It can be represented as a bipartite graph $G = (P \bigcup M, E)$, where P is the set of people, M is the set of items (movies) and the edges in E represent the ratings. Let N_P and N_M be a number of people/items respectively.

A jump is a function $J : R \mapsto S; S \subseteq P \times P$ that takes as input a recommender dataset R and returns a set of (unordered) pairs of elements of P. This means that the two nodes described in a given pair can be reached from one another by a single jump. Obviously, jumps are made using the items in the set M. Authors defined the *skip* jump, which connects two members in P if they have at least one movie in common.

A jump induces a graph called a social network graph of a recommender dataset R. It is a unipartite undirected graph $G_S = (P, E_S)$, where the edges are given by $E_S = J(R)$. The graph could be disconnected based on the strictness of the jump function.

Extracting Social Networks from Communication Evidence Paper [20] proposes an end-to-end system that extracts a user's social network and its members' contact information given the user's email inbox. Social links are created by extracting mentions of people from Web pages and creating a link between the owner of the page and the extracted person. The system is called recursively on each newly extracted people, which result in a large "friends of friends of friends" network.

The process of social network extraction is depicted in the Figure 8-11. The bootstrapping set of names is extracted from email headers in user's inbox. Name coreference resolves multiple mentions of the same person in different format. Subsequently, the system attempts to find person's homepage by submitting queries based on the person's name and likely domain to Google search engine. The results are filtered according to URL features and

word distribution metrics. Then a probabilistic information extraction model is employed to find contact information and person names in the homepages. Newly extracted people who are coreferent to already discovered people are determined. Links are placed in the social network between a discovered person and the owner of the web page on which the person was discovered. The extraction was done using conditional random fields.

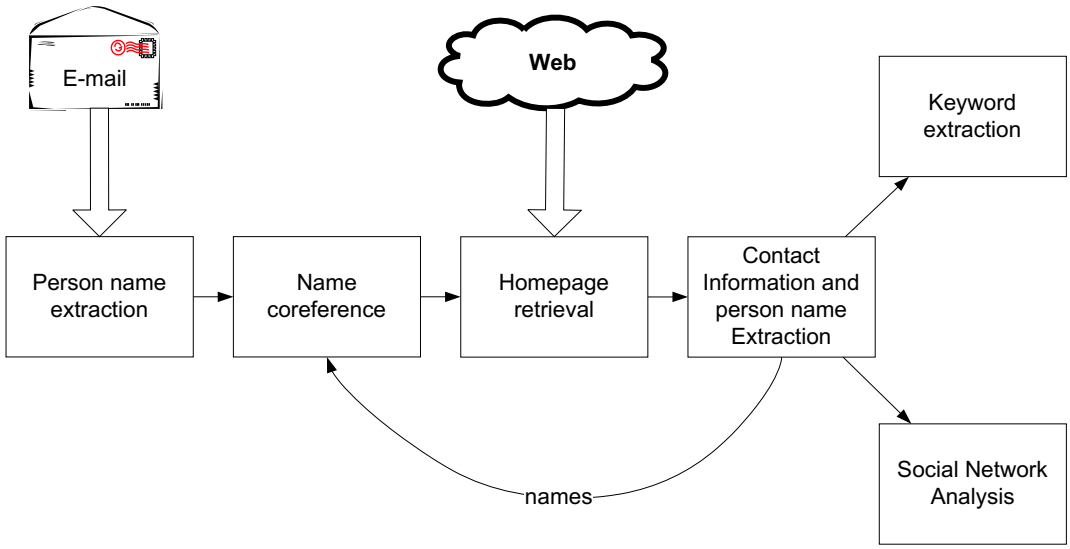


Figure 8-11. Overview of a system performing social network extraction, according to [20].

The approach suffers from names and web appearance ambiguity problems. In some cases, it can recursively extract social network of a namesake of people from original social network. This issue can be fixed as shown in [7].

References

[1] Agichtein, E.: Web Information Extraction and User Modeling: Towards Closing the Gap. *IEEE Data Eng. Bull.*, 2006, vol. 29, no. 4, pp. 37–44.

[2] Andrejko, A., Barla, M., Bieliková, M.: Chap. Ontology-based User Modeling for Web-based Information Systems. In: *Advances in Information Systems Development*. Springer, 2007, pp. 457–468.

[3] Atzenbeck, C., Tzagarakis, M.: Criteria for Social Applications. In Vassileva, J., Tzagarakis, M., Dimitrova, V., eds.: *Socium: Adaptation and Personalisation in Social Systems: Groups, Teams, Communities. Workshop held at UM 2007*, 2007, pp. 45–49.

[4] Barla, M.: Interception of User's Interests on the Web. In Wade, V., Ashman, H., Smyth, B., eds.: *Adaptive Hypermedia and Adaptive Web-Based Systems, AH'06*. LNCS 4018, Dublin, Ireland, Springer, 2006, pp. 435–439.

[5] Barla, M., Andrejko, A., Bieliková, M., Tvarožek, M.: User Characteristics Acquisition from Logs with Semantics. In Kelemenová, A., Kolář, D., Meduna, A., Zendulka, J., eds.: *ISIM ´07: Information Systems and Formal Models*, Slezská universita v Opavě, 2007, pp. 103–110.

[6] Barla, M., Bieliková, M.: Estimation of User Characteristics using Rule-based Analysis of User Logs. In: *Data Mining for User Modeling, Proc. of Workshop held at UM2007*, 2007, pp. 5–14.

[7] Bekkerman, R., McCallum, A.: Disambiguating Web appearances of people in a social network. In Ellis, A., Hagino, T., eds.: *WWW 2005*, ACM, 2005, pp. 463–470.

[8] Berkovsky, S.: Decentralized Mediation of User Models for a Better Personalization. In: *Adaptive Hypermedia and Adaptive Web-Based Systems, AH 2006*. LNCS 4018, Springer, 2006, pp. 404–408.

[9] Berkovsky, S., Kuflik, T., Ricci, F.: Cross-Technique Mediation of User Models. In Wade, V.P., Ashman, H., Smyth, B., eds.: *Adaptive Hypermedia and Adaptive Web-Based Systems, AH 2006*. LNCS 4018, Springer, 2006, pp. 21–30.

[10] Berkovsky, S., Kuflik, T., Ricci, F.: Cross-Domain Mediation in Collaborative Filtering. In Conati, C., McCoy, K., Paliouras, G., eds.: *UM 2007*. LNAI 4511, Corfu, Greece, 2007, pp. 355–359.

[11] Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. *Scientific American*, 2001, vol. 284, no. 5, pp. 34–43.

[12] Bieliková, M.: Presentation of Adaptive Hypermedia on the Web. In Popelínský, L., ed.: *DATAKON 2003*, Brno, ČR, 2003, pp. 72–91.

[13] Brusilovsky, P.: Methods and Techniques of Adaptive Hypermedia. *User Model. User-Adapt. Interact.*, 1996, vol. 6, no. 2-3, pp. 87–129.

[14] Brusilovsky, P.: KnowledgeTree: a Distributed Architecture for Adaptive e-Learning. In Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E., eds.: *WWW (Alternate Track Papers & Posters)*, New York, NY, USA, ACM, 2004, pp. 104–113.

[15] Brusilovsky, P., Millán, E.: Chap. User Models for Adaptive Hypermedia andÈAdaptiveÈEducational Systems. LNCS 4321. In: *The Adaptive Web*. Springer Berlin/Heidelberg, 2007, pp. 3–53.

[16] Brusilovsky, P., Sosnovsky, S., Shcherbinina, O.: User Modeling in a Distributed E-Learning Architecture. In Ardissono, L., Brna, P., Mitrovic, A., eds.: *User Modeling 2005*. Lecture Notes in Computer Science 3538, Edinburgh, Scotland, UK, 2005, pp. 387–391.

[17] Callaway, C., Kufflik, T.: Using a Domain Ontology to Mediate between a User Model and Domain Applications. In Brusilovsky, P., Callaway, C., Nürnberger, A., eds.: *Workshop on New Technologies for Personalized Information Access (PIA 2005)*, Edinburgh, Scotland, UK, 2005.

[18] Cassel, L., Wolz, U.: Client Side Personalization. In: *DELOS Workshop: Personalisation and Recommender Systems in Digital Libraries*, Dublin, Ireland, 2001.

[19] Choi, N., Song, I.Y., Han, H.: A survey on ontology mapping. *SIGMOD Rec.*, 2006, vol. 35, no. 3, pp. 34–41.

[20] Culotta, A., Bekkerman, R., McCallum, A.: Extracting Social Networks and Contact Information from Email and the Web. In: *Conference on Email and Anti-Spam, CEAS 2004*, 2004.

[21] De Bra, P., Calvi, L.: AHA: a Generic Adaptive Hypermedia System. In: *2nd Workshop on Adaptive Hypertext and Hypermedia*, Pittsburgh, USA, 1998, pp. 5–12.

[22] De Bra, P.e.a.: AHA! The adaptive hypermedia architecture. In: *Hypertext 2003*, Nottingham, UK, ACM, 2003, pp. 81–84.

[23] Denaux, R., Dimitrova, V., Aroyo, L.: Integrating Open User Modeling and Learning Content Management for the Semantic Web. In Ardissono, L., Brna, P., Mitrovic, A., eds.: *User Modeling 2005*. LNCS 3538, Edinburgh, Scotland, UK, Springer, 2005, pp. 9–18.

[24] Doan, A., et al.: Learning to Map Between Ontologies on the Semantic Web. In: *WWW 2002*, New York, NY, USA, ACM, 2002, pp. 662–673.

[25] Esteban, A.D., et al.: Using Linear Classifiers in the Integration of User Modeling and Text Content Analysis in the Personalization of a Web-based Spanish News Service. In: *Workshop on Machine Learning, Information Retrieval and User Modeling held at UM 2001*, 2001.

[26] Etgen, M., Cantor, J.: What does getting WET (Web Event-logging Tool) Mean for Web Usability? In: *Human Factors & The Web: The Future of Web Applications*, NIST, Gaithersburg, Maryland, USA, 1999.

[27] Farzan, R., Brusilovsky, P.: Community-based Conference Navigator. In Vassileva, J., Tzagarakis, M., Dimitrova, V., eds.: *Socium: Adaptation and Personalisation in Social Systems: Groups, Teams, Communities. Workshop held at UM 2007*, 2007, pp. 30–39.

[28] Fenstermacher, K., Ginsburg, M.: Mining Client-Side Activity for Personalization. In: *Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, WECWIS '02*, Newport Beach, California, USA, 2002, pp. 205–212.

[29] Heckmann, D., Schwartz, T., Brandherm, B., Schmitz, M., von Wilamowitz-Moellendorff, M.: GUMO – The General User Model Ontology. In Ardissono, L., Brna, P., Mitrovic, A., eds.: *User Modeling 2005*. LNCS 3538, Edinburgh, Scotland, UK, Springer, 2005, pp. 428–432.

[30] Hill, W.C., D., H.J., Wroblewski, D., McCandless, T.: Edit Wear and Read Wear. In: *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, ACM, 1992, pp. 3–9.

[31] Johansson, P.: Natural Language Interaction in Personalized EPGs. In: *3rd Int. Workshop on Personalization of Future TV*, 2003, pp. 27–31.

[32] Kautz, H., Selman, B., Shah, M.: Referral Web: Combining Social Networks and Collaborative Filtering. *Commun. ACM*, 1997, vol. 40, no. 3, pp. 63–65.

[33] Kay, J.: The um Toolkit for Cooperative User Modeling. *User Model. User-Adapt. Interact.*, 1995, vol. 4, no. 3, pp. 49–196.

[34] Kay, J.: Stereotypes, Student Models and Scrutability. In: *ITS '00: 5th Int. Conf. on Intelligent Tutoring Systems*, London, UK, Springer-Verlag, 2000, pp. 19–30.

[35] Kay, J.: Scrutable Adaptation: Because We Can and Must. In Wade, V.P., Ashman, H., Smyth, B., eds.: *Adaptive Hypermedia and Adaptive Web-Based Systems, AH 2006*. LNCS 4018, Springer, 2006, pp. 11–19.

[36] Kay, J., Kummerfeld, B., Lauder, P.: Personis: A Server for User Models. In De Bra, P., Brusilovsky, P., Conejo, R., eds.: *Adaptive Hypermedia and Adaptive Web-Based Systems, AH 2002*. Lecture Notes in Computer Science 2347, Malaga, Spain, Springer, 2002, pp. 203–212.

[37] Kay, J., Kummerfeld, B., Lauder, P.: Personis: A Server for User Models. In De Bra, P., Brusilovsky, P., Conejo, R., eds.: *Adaptive Hypermedia and Adaptive Web-Based Systems, AH 2002*. Lecture Notes in Computer Science 2347, Springer, 2002, pp. 203–212.

[38] Kay, J., Lum, A.: Creating User Models from Web Logs. In: *Intelligent User Interfaces Workshop: Behavior-Based User Interface Customization*, Funchal, Madeira, Portugal, 2004, pp. 17–20.

[39] Kobsa, A.: Chap. Generic User Modeling Systems. LNCS 4321. In: *The Adaptive Web*. Springer Berlin/Heidelberg, 2007, pp. 136–154.

[40] Krištofič, A., Bieliková, M.: Improving adaptation in web-based educational hypermedia by means of knowledge discovery. In Reich, S., Tzagarakis, M., eds.: *Hypertext 2005*, Salzburg, Austria, 2005, pp. 184–192.

[41] Lenčucha, L.: Mining User's Characteristics in the Text. Bachelor thesis, FIIT STU, Bratislava, 2006.

[42] Lu, H., Luo, Q., Shun, Y.: Extending a Web Browser with Client-Side Mining. In Zhou, X., Zhang, Y., Orlowska, M., eds.: *Web Technologies and Applications, 5th Asian-Pacific Web Conference, APWeb 2003*. LNCS 2642, Xian, China, Springer, 2003, pp. 166–177.

[43] Maedche, A., Motik, B., Silva, N., Volz, R.: MAFRA – A MApping FRAmework for Distributed Ontologies. In Gómez-Pérez, A., Benjamins, V.R., eds.: *Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, EKAW '02*, London, UK, Springer-Verlag, 2002, pp. 235–250.

[44] Marcus, S.E., Moy, M., Coffman, T.: Chap. Social Network Analysis. In: *Mining Graph Data*. John Wiley & Sons, Inc., 2007, pp. 443–468.

[45] Massa, P., Bhattacharjee, B.: Using Trust in Recommender Systems: an Experimental Analysis. In: *2nd Int. Conference on Trust Management*, 2004.

[46] Middleton, S., Shadbolt, N., De Roure, D.: Ontological User Profiling in Recommender Systems. *ACM Trans. Inf. Syst.*, 2004, vol. 22, no. 1, pp. 54–88.

[47] Middleton, S., et al.: Exploiting Synergy between Ontologies and Recommender Systems. In: *Semantic Web Workshop held at WWW2002*, ACM, 2002.

[48] Mirza, B.J., Keller, B.J., Ramakrishnan, N.: Studying Recommendation Algorithms by Graph Analysis. *J. Intell. Inf. Syst.*, 2003, vol. 20, no. 2, pp. 131–160.

[49] Nie, Z., et al.: Web Object Retrieval. In Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J., eds.: *WWW'07*, ACM, 2007, pp. 81–90.

[50] Nie, Z., Wen, J., Ma, W.: Object-level Vertical Search. In: *CIDR'07*, 2007, pp. 235–246.

[51] Oard, D., Kim, J.: Implicit Feedback for Recommender Systems. In: *AAAI Workshop on Recommender Systems, July 1998.*, Madison, Wisconsin, USA, 1998.

[52] Obrst, L.: Ontologies for semantically interoperable systems. In: *Information and Knowledge Management*, New Orleans, Louisiana, USA, ACM Press, 2003, pp. 366–369.

[53] Paganelli, L., Paternò, F.: Intelligent Analysis of User Interactions with Web Applications. In: *Intelligent User Interfaces, IUI '02*, New York, NY, USA, ACM Press, 2002, pp. 111–118.

[54] Passin, T.: *Explorer's Guide to the Semantic Web*. Manning Publications, 2004.

[55] Perugini, S., Gonçalves, M.A., Fox, E.A.: Recommender Systems Research: A Connection-Centric Survey. *Journal of Intelligent Information Systems*, 2004, vol. 23, no. 2, pp. 107–143.

[56] Rafter, R., Smyth, B.: Passive Profiling from Server Logs in an Online Recruitment Environment. In: *IJCAI Workshop on Intelligent Techniques for Web Personalisation (ITWP 2001)*, Seattle, Washington, USA, 2001, pp. 35–41.

[57] Reinecke, K., Reif, G., Abraham Bernstein, A.: Cultural User Modeling With CUMO: An Approach to Overcome the Personalization Bootstrapping Problem. In Aroyo, L., Hyvönen, E., van Ossenbruggen, J., eds.: *Proceedings of Workshop on Cultural Heritage on the Semantic Web held at ISWC 2007*, 2007, pp. 83–89.

[58] Studer, R., Benjamins, R., Fensel, D.: Knowledge Engineering: Principles and Methods. *Data Knowledge Engineering*, 1998, vol. 25, no. 1-2, pp. 61–197.

[59] Thomas, R.e.a.: Generic Usage Monitoring of Programming Students. In Crisp, G., Thiele, D., Scholten, I., Barker, S., Baron, J., eds.: *20th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education, ASCILITE'03*, Adelaide, Australia, 2003, pp. 715–719.

[60] Trella, M., Carmona, C., Conejo, R.: MEDEA: an Open Service-Based Learning Platform for Developing Intelligent Educational Systems for the Web. In Brusilovsky, P., Conejo, R., Millan, E., eds.: *Workshop on Adaptive Systems for Web-Based Education: Tools and Reusability*, Amsterdam, The Netherlands, 2005, pp. 27–34.

[61] Tsiriga, V., Virvou, M.: A Framework for the Initialization of Student Models in Web-based Intelligent Tutoring Systems. *User Model. User-Adapt. Interact.*, 2004, vol. 14, no. 4, pp. 289–316.

[62] van Meeteren, R., van Someren, M.: Using Content-based Filtering for Recommendation. In Potamias, G., Moustakis, V., van Someren, M., eds.: *ECML/MLNET Workshop on Machine Learning and the New Information Age*, 2000, pp. 47–56.

[63] Van Schaik, P., Ling, J.: Design Parameters of Rating Scales for Web Sites. *ACM Trans. Comput.-Hum. Interact.*, 2007, vol. 14, no. 1, p. 4.

[64] Wei, Y.C., Lin, M.S., Chen, H.H.: Name Disambiguation in Person Information Mining. In: *Web Intelligence, WI 2006*, Washington, DC, USA, IEEE CS, 2006, pp. 378–381.

[65] Yudelson, M., Brusilovsky, P., Zadorozhny, V.: A User Modeling Server for Contemporary Adaptive Hypermedia: An Evaluation of the Push Approach to Evidence Propagatation. In Conati, C., McCoy, K., Paliouras, G., eds.: *UM 2007*. LNAI 4511, Corfu, Greece, 2007, pp. 27–36.

9
PERSONALIZED COLLABORATION

Jozef Tvarožek

Our highly networked and computerized society facilitates effortless contact among different people, people that did not meet before and perhaps never ever meet in person, and they communicate with one another simply because it is easy and mutually beneficial. Research studies indicate that numerous tasks are achieved more effectively by collaboration of a group of people toward a common goal.

In this chapter, we explore individual components of computer-supported collaboration systems, namely we first examine typical types of collaboration and the system's architecture. Then we look at the group formation process from assembly of groups to evolution into effective virtual teams, and aspects of incentives and reputation mechanisms. Finally, we review algorithmic approaches to modeling human-computer relationships and describe an activation network-based mechanism that can be used to perform effective human-computer social dialogues.

We are interested in methods and processes that make the collaborative experience user-centered and beneficial for the individual user i.e. *personalized*. The principles are demonstrated here mainly on computer-supported collaborative learning systems (educational domain), however the same principles of collaboration can be successfully used in other contexts such as in workplace setting, and we try to mention similarities and differences to these systems through the text.

9.1 Collaborative Systems

We define the concept of *collaboration* very broadly as a joint activity of a group of entities toward a shared common goal. Next, we analyze collaboration on different dimensions and present exemplary instances of each type.

9.1.1 Types of Collaboration

Face-to-face vs. Computer-mediated

Approaches to face-to-face collaboration have been used since a long time with activities such as *role playing*, *simulations*, and *small-group projects*. For example, in the Jigsaw activity, a group of students receives several topics for research. Next, the group is divided into

smaller "expert" groups that explore a single topic. Once the work in expert groups is completed, they get back together to share the new expertise.

In *computer-mediated collaboration*, the communication is altered compared to the face-to-face setting, and by using technology social cues in communication are reduced. The level of anonymity, the way status in the group is perceived, the level of miscommunication, and the conformity pressure from others changes.

Synchronous vs. Asynchronous

Communication in network environment is facilitated through the use of tools; both synchronous and asynchronous tools can be used, each having its advantages and disadvantages. *Asynchronous collaboration* includes *blogs, document sharing*, *electronic mail (e-mail)*, *social bookmarking*, *threaded discussions,* and *wikis*. Asynchronous mode enables everyone to participate at their own will, not requiring rushed actions; it fosters reflective responses.

On the other hand, in *synchronous collaboration* such as *A/V conferencing*, *instant messaging,* and working in *shared workspaces*, the pressure to answer quickly increases. Therefore, the size of a collaborative group which is appropriate for synchronous environment decreases as following the thread of conversation of a sizable group of people is difficult.

Human vs. Artificial

Usually only human entities collaborate, both in face-to-face and computer-mediated environments. Today however, artificial agents can be easily put behind constrained channels such as text messaging, or controlling game characters in role-playing games. By implementing sophisticated relational strategies, artificial group members are (to the extent of the interface) almost human-like and are capable of a very caring and supportive behavior which is ironically not very human-like. Previously unthought-of questions arise as to whether exercising behavior, usually deemed inappropriate, such as egoism, flaming, or jealousy might be in any way beneficial for the human in consideration.

Under the notion of *personalized collaboration* we see a collaborative experience which is tailored to an individual *not necessarily to her liking*, but to achieve the best overall effect, by:

- manipulating the collaborative mode, group's size and structure (role-playing),
- picking appropriate collaborative peers,
- selecting appropriate tasks to work on, and
- supporting interactions by cognitively and socially skilled artificial agent.

Explicit vs. Implicit

We distinguish *explicit collaboration* when participants are well aware that they are in fact collaborating, and receive direct benefits out of it, examples include *direct communication* (face-to-face activities, A/V conferencing, instant messaging, etc.), and *wikis*. In *implicit collaboration* the benefits are indirect and not at all straightforward. For example, in Google's Translate (used for automated web-page translation), users can suggest a better translation than the automatically generated one, effectively producing a better training dataset for the underlying machine learning algorithms, which in turn also benefits others.

Not everything is so clear cut and *social bookmarking* and *social tagging systems* presumably are somewhere in between.

Structured vs. Unstructured

Collaborative activities generally begin as an *unstructured collaboration* which is a free flowing interaction, in computer-mediated setting usually in the form of free textual conversation. To understand and reason about such an unstructured flow of data, it needs to be analyzed into higher level indicators such as topics discussed, depth of knowledge shared, and relationships among the participants.

In *structured collaboration* a specific structure is imposed on the way participants are allowed to interact, such as in *brainstorming* which should ideally proceed in stages that do not get in the way of idea generation early on, thus allowing to build on one another's ideas later, and finally producing possibly novel solutions to the problem at hand. In computer-mediated environment the interaction can be structured by the use of *shared workspaces* (Figure 9-1), which provide a problem-related learning material for synchronous interaction. In shared workspaces the interactions of individual users are analyzed on the level of actions upon the objects in the workspace. Even inherently unstructured interaction types such as textual conversation can be unobtrusively structured, in this case for example by the use of *sentence openers* (Figure 9-4).

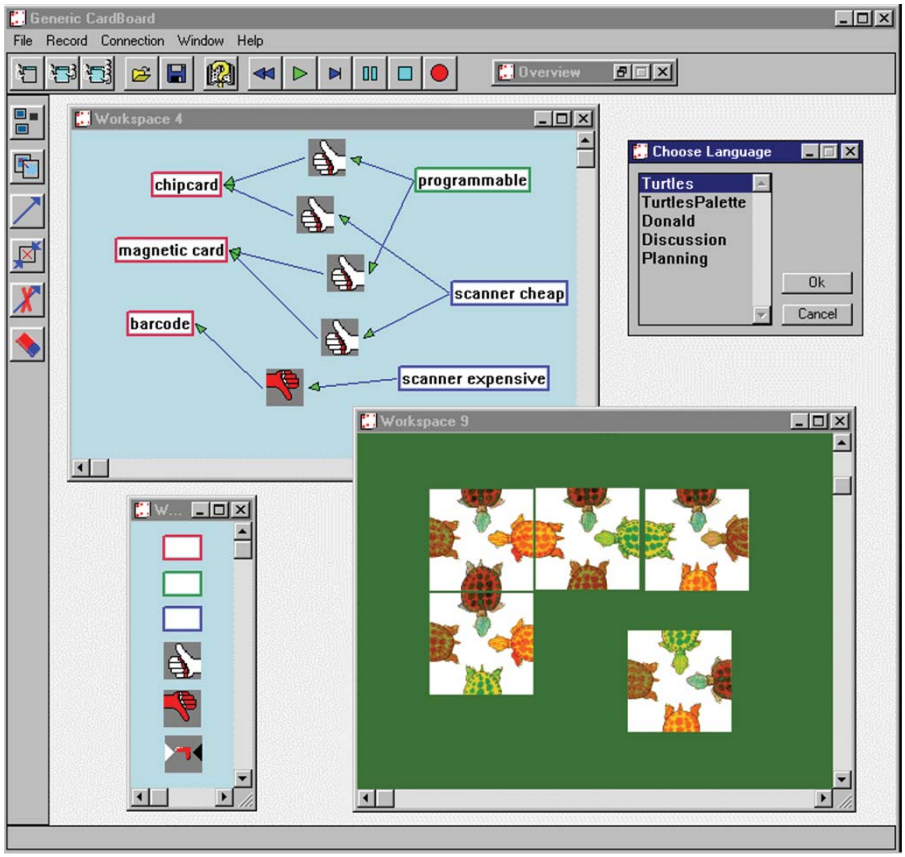


Figure 9-1. Shared workspace application CardBox (Mühlenbrock, 2001).

Size – Dyads, Small Groups, Communities, Masses

Different scales of collaborative interaction are appropriate in different situations. The smallest collaborative unit is a *dyad* in which two peers work on a common task. In a learning context, dyads enable learners to develop and refine their ideas together with a single peer, a process normally requiring a whole-class discussion. Growing in size, *small groups* of 3 to 7 members have more resources available to attempt dealing with more difficult problems that a simple dyad cannot solve. In contrast, when the group grows in size fewer members are given the opportunity to express themselves hindering group's overall creativity.

As people spend more time online they form *virtual communities*, groups of people organized around a specific interest. Within virtual communities people interact usually by the use of computer technology rather than face-to-face. Typically arranged around a restricted area of subjects, virtual communities facilitate exchange of experiences within the field of interest be it fishing or linear algebra; general-purpose communities are not seldom too, e.g. *Flickr*, *Facebook*.

At the largest scale, *mass collaboration* is a joint effort of a very large number of people toward a common goal usually coordinated by internet tools such as *wikis*, *blogs*, or custom-made ones, with examples such as *Wikipedia*, *Open Source Initiative* in software development, citizen science projects e.g. *Clickworkers*, *Startdust@home*, and distributed computing projects e.g. *SETI@home* in search for extra-terrestrial intelligence.

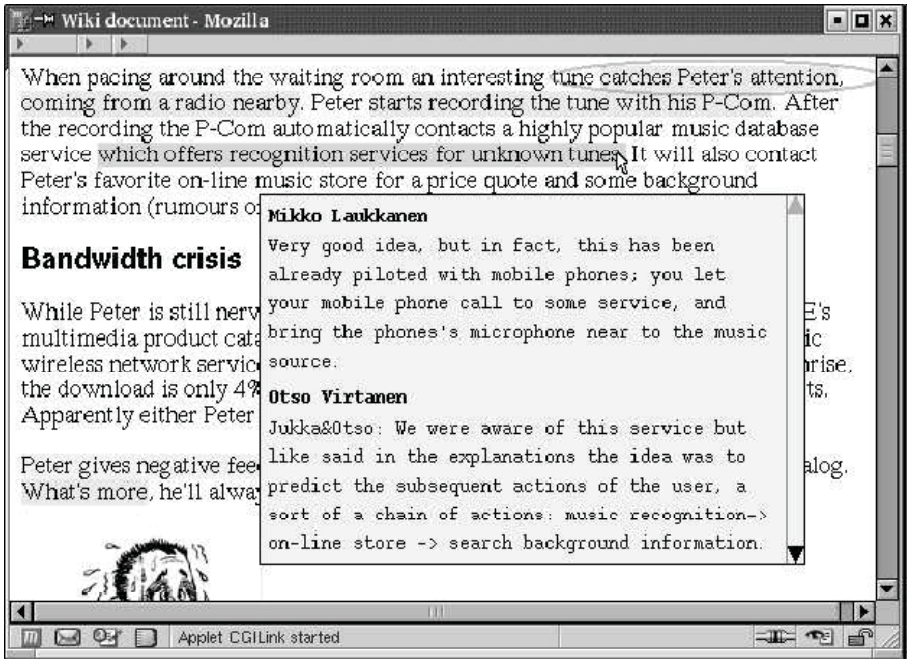


Figure 9-2. Comments in the OurWeb system (Miettinen, 2005).

As an example of a collaborative learning system, in *OurWeb* system (Miettinen, 2005) collaboration opportunities are provided asynchronously using annotations (highlights, comments) and threaded discussions (Figure 9-2). By the use of annotations, users engage

in artifact-centered discourses which may be so small that they would have never happened in a detached thread discussion forum. The system follows the openness and transparency as design principles, openness in using all available material on the Internet, and transparency in that all students should see, benefit, and participate in activities happening in the system. OurWeb works as a proxy to the Internet. Every visited page can be added to the shared document pool for others to search. Also, students prepare projects for collaborative document writing using an integrated Wiki in which others can participate.

Following our dimensions of collaboration processes, computer-supported collaborative learning systems typically use computer-mediated synchronous explicit collaboration of dyads and small groups that can be both structured (e.g. by the use of symbolic actions in workspace work) and unstructured (e.g. free flowing dialogue). Additionally, some degree of asynchronicity is involved as many synchronous tools are also often "abused" for asynchronous work, e.g. instant messaging.

9.1.2 System Architecture

Collaborative-learning systems support and manage learners in various ways; they provide performance indicators, appropriate feedback, etc. In *collaboration management cycle* framework (Jermann, 2001), the process of managing collaboration compares the current state of interaction to the desired state at every point during the collaboration process (Figure 9-3).

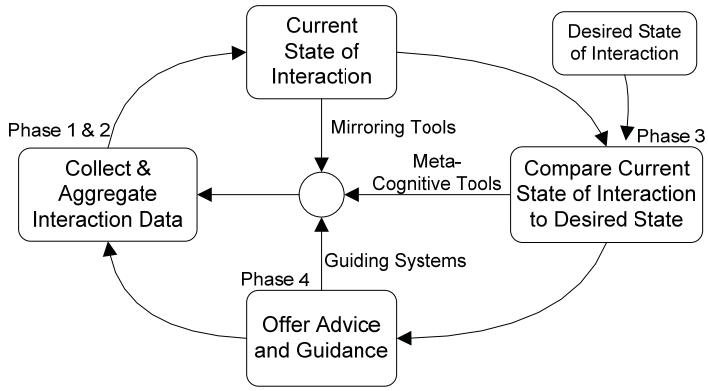


Figure 9-3. The Collaboration Management Cycle (Soller, 2005).

In phase 1, users' actions are observed and recorded for later processing in phase 2 where the set of one or more high-level indicators representing the current state is computed. Indicators represent a quantitative analysis of the observed interactions such as level of participation, social presence, and agreement between peers.

Next, in phase 3, the current state is compared to the desired state of interaction described as ideal values of different indicators. For example, we might want students to maintain high level of participation with enough symmetry between learners. Finally, in phase 4, remedial actions are proposed to minimize differences between the current and desired state of interaction.

Following this cycle, we distinguish three types of systems that differ in the locus of processing i.e. location where decisions about quality of interaction are made (Figure 9-3):

1. *Mirroring systems* aggregate data about the collaborative interactions and reflect this information back to the user for analysis (phases 1 and 2). These systems are designed to raise students' awareness about the actions and behaviors happening in the system. They place the locus of processing to the students or teachers, who must compare the reflected information with their own mental models of desired interaction and devise their own remedial actions as needed.
2. *Metacognitive tools* in addition to displaying the current state of indicators provide referential values of indicators in the desired state of interaction (phase 3). It is left up to the user to diagnose this information and decide about what interventions to undertake.
3. *Guiding systems*, furthering previous types of systems, propose remedial actions to aid learners in collaboration (phase 4) and thus bring the locus of processing to the system itself. The model of interaction used, and system's assessment of the current state are used by the system to moderate the interaction, and are usually left hidden from students.

Although these types of systems provide gradually more functionality they are in certain cases not that different. For example, in mirroring systems by displaying participation indicators students easily identify cases where more participation would be appropriate, having the effect comparable to an advice from a guiding system. Coaching advices become important only when non-trivial inference is required to devise the remedial action needed.

Computer-supported collaborative learning (CSCL) systems typically employ low-bandwidth communication facilities such as text-based discussions because the interactions need to be tracked and analyzed, and methods for analyzing text are currently in a relatively mature state. Video-conferencing and other high-bandwidth streams are, on the other hand, typically used in workplace collaboration – *Computer-Supported Cooperative Work* (CSCW) systems.

In addition to text-based interfaces, structured approaches for collaboration are also used. Shared workspaces, as an electronic version of paper that is jointly used by a group of people for writing and drawing, can provide graph-oriented visual representations for synchronous interaction (Mühlenbrock, 2003). In this context, building and maintaining shared workspace representations is regarded as helpful for the externalization of possibly conflicting problem-related conceptions (Roschelle, 1995).

Discussions can be structured by the use of *sentence openers* (Baker, 1996) that automate the analysis of peer interaction. (Soller, 2000) propose a communication tool based on their *Collaborative Learning Model* that contains groups of sentence openers organized in categories of the model (Figure 9-4). To contribute to the group's conversation, the student selects a sentence opener from one of the subskill categories displayed around the chatting box.

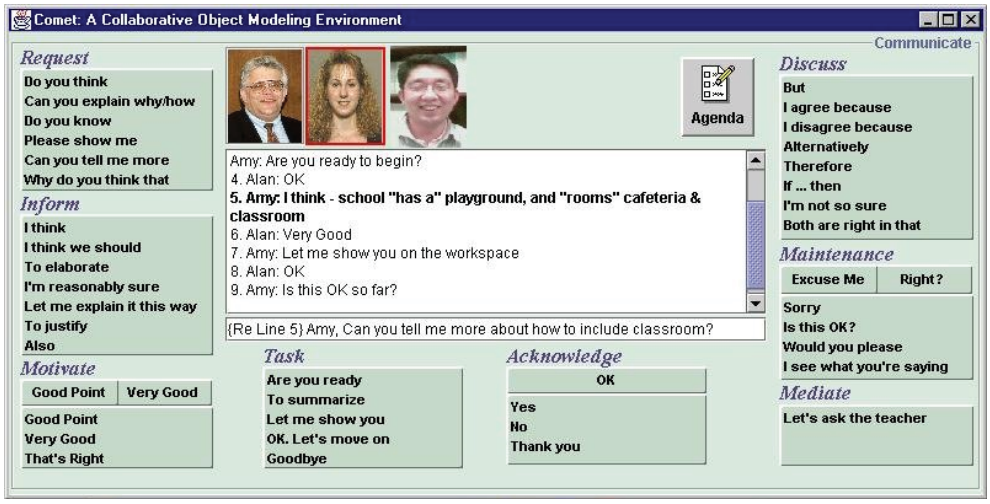


Figure 9-4. Sentence openers in text-based communication (Soller, 2000).

After selecting the opener, the student fills in the rest of the sentence. Students observe the group conversation as it progresses in the large window above the text box displaying the students' names and utterances. The sentence opener interface structures the group's conversation, making the students actively aware of the dialog focus and discourse intent. Sentence openers provide a natural way for users to identify the intention of their contributions without fully understanding the significance of the underlying communicative acts. Peer interaction is evaluated into performance indicators on the basis of the types of conversation acts used to communicate. As a kind of learning, a student who uses a structured sentence opener interface for a period of time internalizes the structure of the activity, and will continue to use the phrases from the interface even after it is removed.

9.2 Group Formation

Group formation is a broad concept involving allocating participants into groups, evolution of the group throughout the course of collaborative activities, and a possible dissolution after the group project ends. It is believed that to form an effective team time is needed, with the group of people pursuing several developmental stages of team formation (Tuckman, 1965).

Nevertheless, researchers believe that an opportunistic model in which groups form, break and recombine during the process as needs and goals of the participants change may uncover opportunities for collaborations that might otherwise go unnoticed (Moreno, 2003) and thus may lead to a higher level of collective responsibility and flexible collaboration.

9.2.1 Group Creation

Several approaches to allocating participants into groups currently exist depending on the amount of work that needs to be done. The easiest job in creating groups is when they are already in place and thus no further work needs to be done. This is the case of web communities which are in a sense self-emerging.

(Backstrom, 2006) analyzed the evolution and growth of communities on huge *LiveJournal* community of more than 10 million users with a significant fraction of the users being very active with about 300,000 profile updates in 24-hour period, and arrived at an interesting observation that not only the number of existing ties – friends that are already members – determines the probability of joining the community but also by how these friends are connected to one another with strong ties between these friends to be more favorable.

If the groups are not yet established they can be created trivially either randomly or self-selected by students according to learner profiles. Another way is to use computer-supported approaches. The research on algorithms for creating groups seems to divide into two principally different strands:

1. *One-time formation* methods produce a single assignment of participants into groups according to pre-defined criteria. Groups are created in this process only once and thus it cannot possibly take future collaboration outputs into account during the group creation process. Different methods of this type are used:
   a. *constraint-based*, where the allocation problem is expressed as a *Constraint Satisfaction Problem* (Kumar, 1992) and the negotiation problem can then be handled by a constrain satisfaction solver,
   b. *statistical* methods which represent students' features using a vector-based approach and generate assignments into groups employing various statistical or optimization algorithms.
2. *Repeated formation* methods assume that multiple rounds of collaboration will happen and take feedback on previous assignments into account during the creation of next assignment of students into groups. In effect, these methods are *self-optimizing* as the creation process is set up in such a way that the benefits of the current allocation are evaluated after the collaboration completes and further reused, thus rewarding a better allocation next time.

The actual process of setting up an individual group is usually a three stage process (Wessner, 2001): (1) *initiating* the formation manually by the learner or triggered automatically by the system, (2) *identifying* peer learners that meet certain requirements, and (3) *negotiating* with potential participants.

Collaborative tasks are often executed in a structured way and involve a certain amount of role-playing which further imposes skill and ability demands on possible actors. Demands for different roles are usually different so that even actors with sharply contrasting skill sets may find the collaborative experience of working together rewarding.

For example, in an early collaborative effort, the COSOFT project (Hoppe, 1995) when the learner encounters a problem he (phase 1) initiates the formation of a so-called learner-helper group by pressing the "Ask" button. System displays a list of potential peers (phase 2) based on the learner model. The learner can then select the helper who is subsequently asked if she wishes to help the learner with the current topic (phase 3). After this learner-helper group is successfully negotiated a shared communication channel between them is established.

In OurWeb system (Miettinen, 2005), group formation is supported by identifying students with shared interests; authors suggest that a suitable way of supporting group

formation might be to augment documents in the system's pool with information about people who have been actively utilizing them. Interest profiles would then emerge from the activity patterns of the students and the overlap in the navigation of students.

Constraint-based Group Creation

Suppose that for students in the class, instructor wants to partition them into groups for a collaborative course activity. Not only do we need to maximize the students' individual benefits from participating but we are also concerned with balancing the capabilities and resources each group has available, so that everyone has roughly the same chances. These additional constraints may be arbitrary provided they admit an efficient computational procedure; take balancing groups on average members' grades as an example.

Constraint-based methods have been researched in conjunction with the Semantic Web approaches, since students' characteristics and types of group formation constraints can be meaningfully described by ontologies.

Students' features are usually modeled by extending a standard ontology used to hold additional information that is required. In (Ounnas, 2008) the FOAF (friend-of-a-friend) ontology for social relationships is enhanced by additional student's personal, social, and academic data (i.e. preferred learning styles) into a so-called *Semantic Learner Profile* (SLP) holding a large range of information used for group formation.

Next, semantic information is put in from both sides (Figure 9-5): (1) by students submitting their FOAF + SLP profiles, and (2) by the instructor selecting appropriate constraints to be imposed. The framework enables instructor to specify two types of constraints: *strong* that have to be met in all resulting group assignments, and *weak* that need not be met necessarily at all times but the more weak constraints are met the better the resulting assignment. In addition, priorities can be assigned to weak constraints to facilitate generating more appropriate group formations when a perfect formation is not possible.

The group generation process itself is done by a DLV solver, an implementation of disjunctive logic programming, used for knowledge representation and reasoning. Instructor specifies the constraints in DLV's native language – Disjunctive Datalog extended with constraints, queries and true negation (Leone, 2006). Depending on the students' data and instructor constraints, DLV outputs more than one grouping of the students, and the best one considering the number of violated constraints and their priorities is selected. Authors claim that this formation process does not leave any students unselected – so-called *orphan problem*. This is achieved by the virtue of weak constraints, in such a way that students are assigned to groups in all cases; at worst some constraints are violated producing an imperfect solution.

With groups selected, we are interested in evaluating the quality of the selection. Evaluations are usually done subjectively by students and the use of questionnaires on team efficacy, peer rating, and individual satisfaction, and objectively by the demonstrated performance. Specific methods vary. In the case of semantic group formation framework outlined previously, authors propose various numerical metrics for evaluation (Ounnas, 2007) such as how well the constraints are satisfied, how is the group satisfied (depending on individual satisfaction), how well the group is formed (depending on all goals set by the instructor), etc. More or less all proposed metrics boil down to aggregating

previous simpler metrics by the use of mean or standard deviation, and without providing any relevant empirical results on these measures the utility of computing these numerical measures remains largely unanswered.

In addition, constraint-based group formation assumes that instructor knows exactly what constraints are good for making the groups collaborate effectively. However, knowing what exactly makes collaboration effective is still unclear and currently is the focus of intensive research. Therefore, methods that attempt to optimize further decisions based on previous possibly imperfect choices seem to be a more suitable alternative.

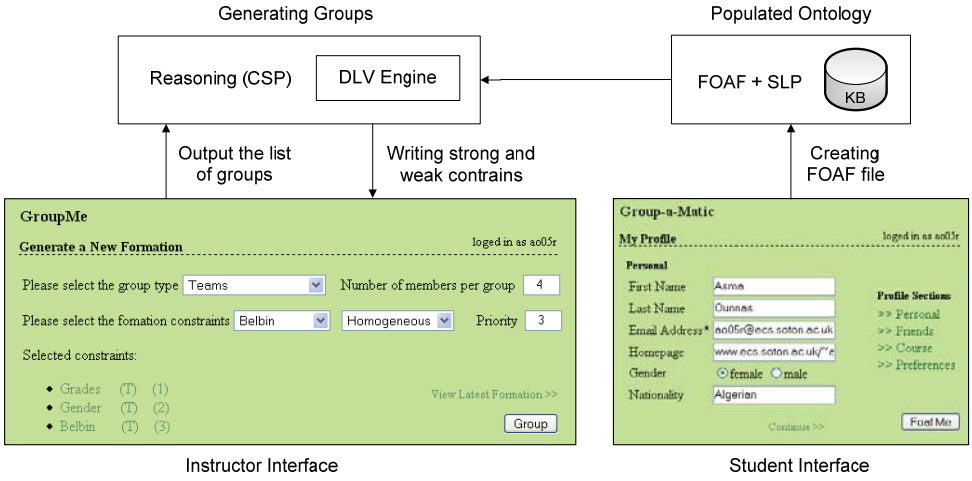


Figure 9-5. Semantic group formation framework (Ounnas, 2008).

Statistical Group Creation

Statistical methods compared to the constraint-based ones do not dwell into detailed constraint or rule selections but rather process students in a more data-driven way. Students' features are usually represented by a multidimensional vector-space model of attributes; i.e. a student is represented by an n-dimensional vector where the value in i-th dimension corresponds to the student's value for i-th attribute.

Having a vector-based representation in place, we can easily compare two different students on the basis of differences in respective dimensions of their vector representations. (Gogoulou, 2007) propose a group creation method that groups students with similar personality features. Learner's personality and performance attributes are represented by a n-dimensional vector whose values are from a five-level Likert scale, the difference between two students is the sum of differences on respective dimensions of their vectors, and group quality is evaluated with respect to attributes as the difference of the lowest and highest value for members of the group. Total group quality is the sum of group quality across all attributes, and the groups' assignment quality is the sum of group qualities across all the groups. Next, the proposed group formation method is analogous to the *k-means clustering* and assigns students into M groups of K students as follows: As a first step, M centers (points in the n-dimensional space) of clusters are chosen at random. Then, for each center, K closest (by Euclidean distance) students are assigned to the appropriate

cluster. Afterwards, for each cluster new center is calculated as the average of values of its member students. Given the new centers, students are re-assigned to the now closest clusters accordingly, etc. This process is repeated until the assignment into groups converges and no more students change their group during the final iteration. The method was evaluated on a set of 52 learners, and for the presented experimental data it outputs a rather good assignment with quality of 94 out of 101 possible.

Gogoulou further improves the group formation by using a genetic algorithm that for the given dataset produces an assignment having the quality of 96. These methods seem nearly perfect in respect to the proposed quality measures, and thus naturally posit a question of utility and appropriateness of their measures given that no breakthrough outputs in collaboration were achieved. A *genetic algorithm* is an optimization method that starts with a set (population) of randomly selected solutions and tries to modify (mutate) individual solutions and/or combine (crossover) different solutions to produce another possibly better (according to a pre-specified fitness function) set of solutions. Iteratively producing better sets of solutions it converges to the optimal solution. By having multiple solutions simultaneously the method is robust against falling into local optima. Besides devising the algorithm itself, which is rather straightforward, the input parameters such as mutation and crossover probabilities, number of generations, and population size with which the algorithms perform effectively are important. In this case though, authors do not mention the values of these important parameters that produced their results, and thus it is not clear if they can be recreated. Either way, authors demonstrated this to be a viable method for group formation, slightly outperforming the basic clustering method.

The use of statistical and/or optimization methods is not seldom, and other methods have been proposed such as repeated hill-climbing optimization with weighed constraints (Cavanaugh, 2004), *Fuzzy C-Means clustering* (Christodoulopoulos, 2007) and an interesting *Ant Colony Optimization* method (Graf, 2006).

Statistical methods represent students' characteristics in a coarse-grained way. Therefore statistical group formation methods that produce one-time assignments suffer from similar deficiencies as constraint-based methods, namely they simply generate an assignment of students into groups and evaluate its quality by how many rules are satisfied (or broken), all under the assumption that the proposed method might somehow improve the probability of successful collaboration; never receiving any feedback whether it really did. In the next section, we examine methods that generate group assignments repeatedly, each time hopefully better than the previous one driven by the feedback on the previous assignment.

Repeated Group Creation

Repeated group formation assumes that groups are created dynamically or need to be created for several successive occasions so that payoffs for individual students can end up more balanced compared to a single allocation methods described previously. In other words, possibly disadvantaged students in one round of collaboration might get a more favorable assignment in following runs.

In *Opportunistic Group Formation* (OGF) framework groups are formed dynamically at appropriate situations with the help of personal agents that negotiate and manage collaborative learning activities the students can engage in (Inaba, 2000). Agents in OGF sup-

port individual learning, propose shifting to collaborative learning, and negotiate to form a group of learners with suitable role assignment, based on the learners' information from individual learning. Authors devise the *Learning Goal Ontology* (LGO) that accommodates learning goals of several learning theories such as *learning by observing others*, *learning by self-expression*, *learning by teaching*, and *learning by diagnosing*. Furthermore, LGO describes dependencies between individual a whole-group goals, and thus is used as a rationale for forming appropriate groups.

Another interesting method of repeated group formation is described in (Supnithi, 1999) where students represented by agents bid in an iterative Vickrey auction to join their favorite group. Agents utilize virtual currency earned from previous collaborative activities. After each completed activity, agents are rewarded with virtual currency depending on their individual and group performance based on peer evaluations. Better achieving students receiving more are thereby capable to bid for the future membership in their favorite groups more successfully. This auctioning group-formation mechanism is also utilized in the *I-MINDS system* (Soh, 2006).

9.2.2 Group Evolution

Once the group participants are selected they can engage in a collaborative activity. In this section we describe how the interactions within the group affect the participating users and the group itself.

It is widely recognized that during their lifetime groups undergo an evolutionary process in which they become efficient teams. More than 100 theories of group development exist to date, the most prominently cited one being the Tuckman and Jensen's stages of group development (Tuckman, 1977) consisting of five stages:

1. *Forming* – group members get to know each other and the task to be solved. They can feel uneasy not knowing what is required and how the group will function.
2. *Storming* – when things get "stormy", individual differences surface and conflicts emerge, and roles of the participants and the group structure is put to question.
3. *Norming* – after resolving these problems the group starts to function harmoniously, establishes rules, and group members start to support each other.
4. *Performing* – at this stage rules are well in place, group is doing the work on the common task the way it is supposed to do.
5. *Adjourning* – group retires as the project ends, feedback on the group performance facilitates learning.

Different groups may spend different amounts of time in each of the stages but the theory suggests that all groups pass sequentially through all these stages.

Face-by-face vs. Computer-mediated Groups

Group evolution is the subject of extensive study in psychology and social sciences; called *group dynamics* that is examining how people work in small groups. For an up-to-date treatment of group dynamics and teamwork see (Forsyth, 2006; Levi, 2007). We briefly mention main differences between face-to-face groups and computer-mediated groups.

In computer-mediated setting, people communicate via e-mail, videoconferencing, discussion boards, social networking software, etc. This has led to the notion of *virtual team*, denoting any team whose members' interactions are mediated by time, distance, and technology (Driskell, 2003), i.e. that the team works together on a common task while physically separated. The effects of computer-mediated communication technologies are twofold. Positive effects include better access to and diffusion of information, and easier connection with others, and negative effects as information overload, less face-to-face communication, and increased isolation of individual members.

Earlier research showed that in a student group the differences in personal status are reduced in virtual setting (Parks, 1999). In face-to-face discussions few dominant members with higher status talk most of the time, and many group members limit their contributions to supporting the main positions that emerge. In computer-mediated setting on the other hand, social cues are reduced, and people communicate on the basis of their opinions and knowledge rather than their social status. In some tasks this proves counterproductive, and virtual groups perform poorly on decision-making and negotiation tasks where a consensus on issues is required. Then again, due to more effective and focused communication tasks such as brainstorming and problem-solving are better suited to virtual groups (Hertel, 2005). In organizational setting however, members of a virtual team remain aware of the status of others regardless of the technology used (Driskell, 2003).

Virtual groups also provide effective social support to people in need. Members of self-help groups instead of leaving their homes and travelling to the meeting are brought together by computer technology (Tate, 2004). When anonymous, members are not identifiable and reveal more intimate details about their experiences and respond more emotionally to others than in a face-to-face meeting. Members praise the quality and quantity of information they receive; more factual and practical advice is exchanged while instances of inappropriate behavior and hostile postings are rare (Houston, 2002).

Incentives & Reputation

Success of an interaction largely depends on the amount of effort invested and the reliability of the parties involved. By participating in large communities the potential for possible interactions is enormous, and before entering into relations with some unknown others we need to verify their credibility.

For this purpose, *reputation* as an aggregated record of previous interactions is collected, maintained and disseminated to others by the use of *reputation systems*. The history of entity's past interactions accounts for its ability and reliability, allowing others to make informed choices about how and whether to work together with that entity. Furthermore, the expectation that current performance will be made visible in the future discourages cheating and exerting poor effort, and thus creates *incentives* to perform well and reliably.

Reputation systems process (1) *objective feedback* such as time spent solving a particular task, agreed sale price, and (2) *subjective feedback* such as satisfaction ratings of working with others. Reputation is computed by aggregating previous reports across users and time, and often relies on *transitivity of trust* in that reports originating from more reputable users are weighted more that those from users having a lower reputation.

The presence of subjective feedback inherently generates problems. In the remainder of this section we examine the vulnerabilities of reputation systems and methods that were proposed to tackle them.

First problem, so-called *whitewashing* (Lai, 2003), arises when an entity can start over with a new pseudonym that is not associated with the interaction history of the previous pseudonym, effectively disposing of the evidence of past (possibly malicious) activities. Unbounded whitewashing can effectively disable any reputation system, while having a sufficiently high starting fee does indeed prevent this behavior, collecting fees is not always viable.

Therefore, indirect payments in the form of degraded service for newcomers were proposed (Resnik, 2001). In their model, the pay-your-dues (PYD) strategy distinguishes between newcomers and veterans, veterans being the users that have interacted positively at least once. Analogous to mistrust to newcomers in common social situations, in PYD veterans do not collaborate with newcomers until they have proved themselves enough to allow for a mutually beneficial interaction. Authors further proved that an extended stochastic version of this algorithm executes the highest fraction of cooperative outcomes and therefore is the socially most efficient strategy (in game-theoretical terms) in the presence of whitewashing.

Second major problem of reputation systems is dealing with the lack of objective feedback or so-called *phantom feedback* generated using false pseudonyms (sybils) created for the sole purpose of providing this phantom feedback. This problem can be modeled in systems based on transitive trust, i.e. the input is represented as a *trust graph* whose vertices are the entities, and directed (one-way) edges have associated trust value – nonnegative real value summarizing the feedback that one edge's vertex (entity) reports on the other one. Aggregation mechanism computes the reputations of the vertices based on the trust values. Entities are not directly affected by the feedback they provide, only from the ratings they receive from others, and therefore an entity has no incentive to provide relevant feedback.

On the contrary, a less credible entity has every reason to provide *dishonest feedback* so as to undermine the credibility of (possibly negative) incoming feedback. Thus, a robust reputation mechanism must ensure that an entity cannot increase its reputation by manipulating its own feedbacks. In the second major class of attacks studied, the *Sybil attacks* (Douceur, 2002), malicious entity creates fake pseudonyms to boost the reputation of its primary pseudonym. In the trust graph model, the attacker can specify arbitrary trust values originating from sybil nodes, and can divide incoming trust edges among the sybils provided that the total sum of trusts is preserved.

Several methods were proposed. The simple version of *PageRank* algorithm (Brin, 1998) as described below can be applied:

$$R(u) = \varepsilon + (1 - \varepsilon) \sum_{v|(v,u)\in E} [R(v) \cdot t(v,u)]$$

where $R(u)$ is the reputation of web page u, directed edge (v,u) corresponds to the hyperlink from page v to page u, and trust values are $t(v,u) = 1/OutDegree(v)$. Analogously, $u \in V$ can be an entity, directed edge $(v,u) \in E$ shows that entity v has interacted with u, and $t(v,u)$ is the degree of trust that v has in u. This simple version of PageRank is symmetric and therefore is prone to dishonest feedback; it is also prone to Sybil attacks (Cheng, 2006).

Trust aggregation method PathRank (Friedman, 2007) computes the reputation of entity v as the length of the shortest path (edges have length equal to inverse trust values) from some start node $v_0 \in V$. This allows for personalized reputation functions where each node uses itself as the start node. PathRank algorithm is robust against both dishonest feedback and Sybil attacks since it is asymmetric and sybils cannot manipulate the length of a shortest path between legitimate entities.

The list of above mentioned types of attack is by no means complete and reputation systems are under constant attack by previously unknown types of attack, requiring them to continuously tweak their ranking mechanisms.

9.3 Human-Computer Relationships

Interpersonal relationships were found to be useful in many contexts. Even in education, relationships between students are important in peer learning situations, and collaboration between friends was found to be more effective than collaboration between acquaintances (Hartup, 1998), as friends engage in more extensive discourses and are more supportive and critical at the same time.

People react to computer agents in fundamentally social ways (Reeves, 2003). The provisions of human relationships such as emotional support, group belonging, and social network support, can be made available by the use of intelligent computer technology. Embodied in various physical forms (e.g. toys, jewelry), or even purely software agents, *relational agents* are computational artifacts designed to build long-term social-emotional relationships with human users (Bickmore, 2003). Language is the primary modality in constructing human relationships, and even though many relational strategies are nonverbal, relational agents need to implement at least simple text interfaces. Animated humanoid agents employing speech, gestures, intonation and other nonverbal modalities that emulate face-to-face interactions are currently being studied in human-computer interaction community.

Typically, relationships span from the micro level of face-to-face relational conversation to the macro level of long-term maintenance, and thus relational agents must employ strategies for maintaining and developing relationships, just as people do.

Several interesting relational effects in non-embodied text-only human-computer interfaces were demonstrated (Reeves, 2003), such as computers which praise rather than criticize their users are liked more, users prefer the computer to match them in personality, and users prefer computers that become more like them over time over those which maintain a consistent level of similarity.

9.3.1 Relational Strategies

In social psychology, the concept of *relationship* is referring to the interaction between two people whose behavior is mutually dependent in that a change in the state of one produces a change in the other (Kelley, 2002). Furthermore, relationship is not defined by generic patterns of stereotypical interactions (e.g. buyer–seller) but rather by the unique patterns for a particular pair (Berschied, 1998).

In relationship, friends are expected to provide various provisions for each other (Duck, 2007) such as the sense of belonging, anchor points for opinions and beliefs, oppor-

tunities for self-expression and self-disclosure, physical support, and reassurance of worth and value. Different models of relationship were proposed.

In *social exchange models,* costs, benefits and investments in "relationship business" are modeled (Burke, 2006), i.e. the perceived costs and benefits of providing and receiving provisions respectively, and whether suitable alternatives are available affect the relationship's duration. In *dimensional models,* features that characterize different stereotypical interactions are used. (Svennevig, 1999) proposes an extension of earlier models into four dimensions of relationship as follows:

1. *Power.* The person's ability to control the behavior of the other. Typically, occurs when the distribution of rights and obligations resulting from the relationship roles is asymmetric, e.g. institutional roles and positions.
2. *Solidarity.* The degree of like-mindedness, e.g. political membership, religion, gender, birthplace. Rights and obligations are usually symmetrically distributed.
3. *Familiarity.* Mutual knowledge of personal information, i.e. breadth (number of topics) and depth (public or private) of information disclosed.
4. *Affect.* Mutual attraction, i.e. the degree of liking for each other.

These dimensions are usually interrelated, though it needs not to be the case. For example, high levels of solidarity based on group membership need not to elicit high levels of either familiarity or affect, and pure affect arises in some situations (e.g. love at first sight).

People engage in different relational strategies for maintaining relationships. Five *strategic maintenance behaviors* that are used intentionally to keep up the relationship were identified (Haas, 2005): *positivity* (e.g. cheerfulness, positive comments), *openness* (e.g. self-disclosure, meta-relational communication), *assurances* (e.g. verbal or nonverbal expressions of love and comfort), *shared tasks* (e.g. household duties), and *social networks* (e.g. communicating with mutual friendships). In addition to these, different *routine maintenance behaviors* that are regular or habitual behaviors that serve to maintain the relationship were identified, such as joint activities, affection, avoidance of conflict, and focus on self (e.g. watching weight, furthering career). Finally, people expect relationships to change over time (Duck, 2007) i.e. the content, quality and diversity of interactions and activities change, reciprocal behavior decreases and complementary behavior increases, among others.

With language being the principal means for developing relationships, relational agents need to implement the relational strategies outlined above in dialogues with their users. Pioneering work in this area is the computational model of mixed social-and-task dialogue addressing some of these strategies (Bickmore, 2003) evaluated within the *REA system* in which an *embodied conversational agent* (ECA) performs the role of a real-estate salesperson. Within this domain, the real-estate agent pursues several goals in parallel: determining clients' housing preferences (e.g. size, location), establishing trust and reducing clients' fear about such a big purchase, establishing agent's expertise and credibility. Although REA has a fully articulated graphical body, senses the user through cameras and microphones in real-time, and is capable of speech with intonation, facial display and gestural output, it is her conversational properties that are interesting to us. Indeed, later in evaluation it was found out that REA's nonverbal behavior was utterly insufficient (even inappropriate), and specifically her conversational properties exhibited in a second phone-

only evaluation were deemed more successful. Except task-oriented conversational moves, REA can engage in social dialog moves analogous to forms of small talk – questions ("*It's a nice morning, isn't it?*"), statements, and stories not relating to the task.

Small talk, also referred to as *phatic communication* (Malinowski, 1923), is a talk in which interpersonal goals are emphasized and task goals deemphasized (e.g. social chat, conversational stories). Besides its transitional function of moving into a conversation that might otherwise be uncomfortable, it helps people establish expertise and credentials (Jaworski, 2006). Small talk avoids face threat by using safe topics, establishes common ground by topics that are clearly in the current context, increases coordination between participants by allowing them to synchronize short units of talk and nonverbal acknowledgments, and allows for reciprocal appreciation of each other's contributions, therefore maintains solidarity and increases familiarity and affect (Bickmore, 2003).

9.3.2 Modeling Social Dialogue

In the remainder of this section we describe REA's approach to modeling mixed social-and-task dialogue, some computational details are simplified due to brevity, and a full account can be found in the original work (ibid, p. 64–70).

REA's relationship model is based on the Svennevig's model described earlier, and uses three of its relational dimensions: *familiarity depth*, *familiarity breath*, and *solidarity*. Each dimension is normalized to unit scalar (ranging from zero to one), and is updated dynamically during the interaction with the user. REA maintains conversational topics that she can engage in, for each the minimum and the maximum value of social invasiveness are specified. Then, as conversation progresses (topics are introduced) familiarity depth is updated as the degree of social invasiveness of topics that were already introduced (i.e. the more intrusive topics are introduced, the higher familiarity depth gets), familiarity breadth corresponds to the ratio of how many of the available topics were already introduced, and solidarity increases gradually (linearly) with each dialogue move as the number of dialogue moves is limited and they cannot be repeated.

Each available REA's dialogue move corresponds to a speech act (e.g. story, query, statement) and is about a predefined set of topics; thus the coherency of two sets of topics can be easily computed. Then, the value of face threat of introducing a particular dialogue move is computed according to the current relational state as weighted sum of: threat due to familiarity, intrinsic face threat of a corresponding speech act, and amount of threat due to topic incoherency. The weights in this sum provide a flexible mechanism for implementing different agent personalities, normal, goal-oriented, and chatty agents were considered.

With the ability to evaluate a single step in dialogue (as described in previous paragraphs), REA's discourse planner is capable to interleave small talk and the task using an *activation network-based approach* based on Maes' *Do the Right Thing architecture* (Maes, 1989). Nodes in the network represent conversational moves and edges between them represent various enabling and disabling conditions among the moves (Figure 9-6).

Dialogue planning is seen as a spreading activation process that uses information from the current state, relational model and task goals to determine which moves are more likely to succeed. Dialogue plans correspond to paths in the network. Activation process is

two-way; energy is moved backward from task goals to moves which directly lead to their achievement, from there to moves which enable those moves, etc. Simultaneously in the forward direction, energy is pushed into moves which can be immediately performed given the conversation state and relational model, from there to moves which are enabled by those moves, etc. In this way, REA's planner achieves task goals, conversational moves obey logical preconditions, while moves expected to cause face threat to the user are deferred, and topics are introduced gradually and coherently.

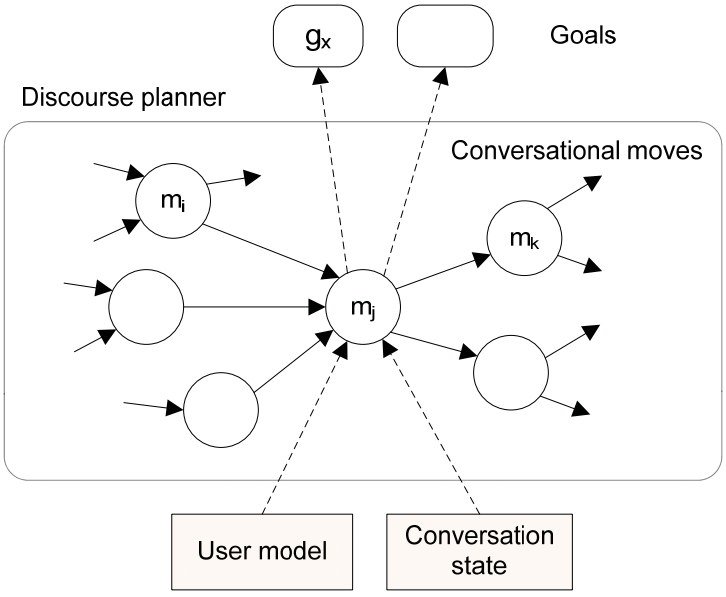


Figure 9-6. REA's conversational moves in the activation network.

Although REA's relational modeling is relatively simple, and could engage only in a system-initiative dialogue it produced interesting results. In evaluation, 4 experiments were conducted, TASK vs. SOCIAL conditions crossed with EMBODIED vs. PHONE conditions. In the TASK conditions (i.e. TASK+EMBODIED and TASK+PHONE) only task-oriented dialogue moves were used, in the SOCIAL conditions identical task-oriented dialogue moves along with social dialogue were used. In the EMBODIED conditions full embodiment capabilities as described earlier were employed, in the PHONE conditions only 3D renderings of apartments were shown and the agent conversed with subjects over phone. For the purpose of the experiment, REA speech recognition was controlled by a wizard-of-oz setup. Interestingly, only a single positive result was achieved, namely that some evidence was found that small talk can lead to an increase trust. In this case however, it was observed only for one group of users (extroverts) and in one medium (face-to-face), while it was hypothesized that it may be due to other reasons (e.g. inherent impulsivity of extroverts). Other than that, the EMBODIED conditions clearly demonstrated that REA's visual was very cold and uninviting, and her nonverbal behavior was insufficient. Subjects generally preferred to conduct small talk with her over phone, while preferring task-only talk face-to-face. Finally, the system-initiative approach was unable to carry on with user-initiated social dialogue moves, and subjects simply believed that REA has

failed to understand what they were saying, and study authors concluded that unless a general intelligence can be utilized the dialogue context must greatly limit what can be said in social dialogue, without destroying the natural and relaxed feel.

9.4 Summary and Open Problems

Collaborative learning tools currently do not really understand the underlying users' interaction. Support is predominantly on the structural level (e.g. symbolic actions within a workspace), and tools attempt to "understand" unstructured interactions such as natural language dialogues by the use of information extraction and machine learning methods, which unfortunately are rather inaccurate for high-level natural language tasks. The main differences between individual systems thus correspond to the level of analysis they perform, and thus the amount of "reasoning" they engage into.

There are several open problems in the area of collaborative support. As for group formation strategies, previous attempts on group creation focus largely on satisfying preset conditions determined before the actual collaboration occurs, or utilize only shallow statistical measures of previous collaborations' successes.

To make the whole collaborative experience for the user personalized and more sociable we have to ensure that in the group creation process we take more advantage of previous user's experiences with other collaboration partners which, in fact, devise an overall profile of what the specific user awaits from the collaboration, and thus should be looked upon in the first place. Furthermore, analyzing time-related data of how users interact with each other inside the group and with outsiders, and how the group evolves through the several developmental stages, provides interesting avenues for further research.

In the area of analyzing collaborative interaction, contemporary Web 2.0 social environments systems facilitate the creation of vast amounts of user-generated discourse. Analyzing this interaction content and extracting knowledge from it assists in building structured knowledge bases for improving the dialogue capability of artificial agents that can be further used to elicit additional interactions. Researches try to build computational models of discourse.

To extract relevant knowledge from textual conversations an integral approach for identifying both semantic and pragmatic features is required. Fully-automated annotation methods seem to fail in this respect and semi-automated methods, in which statistical and/or machine learning algorithms pre-process the annotation in the first step while users make final adjustments in the second step, are proposed. The problem here is: why, what and how to annotate to enable personalized and more sociable interactions for an effective collaboration to happen.

References

[1] Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: membership, growth, and evolution. In *Proc. of 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM Press, pp. 44–54, 2006.

[2] Baker, M., Lund, K.: Flexibly structuring the interaction in a CSCL environment. In *Proc. of the European Conference on Artificial Intelligence in Education*, pp. 401–407, 1996.

[3] Baron, N.S.: Discourse structures in instant messaging: The case of utterance breaks. In *Computer-mediated conversation*, Hampton Press, Cresskill, NJ, 2005.

[4] Berscheid, E., Reis, H.: Attraction and Close Relationships. In *The Handbook of Social Psychology*, McGraw-Hill, New York, pp. 193-281, 1998.

[5] Bickmore, T. W.: Relational Agents: Effecting Change through Human-Computer Relationships. *Doctoral dissertation*, MIT, 2003.

[6] Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, Elsevier, 1998.

[7] Burke, P. J. (Ed.): *Contemporary Social Psychological Theories*. Stanford University Press, 2006.

[8] Cavanaugh, R., Ellis, M. G., Layton, R. A., Ardis, M. A.: Automating the Process of Assigning Students to Cooperative-Learning Teams. In *Proc. of the 2004 American Society for Engineering Education Annual Conference & Exposition*, 2004.

[9] Cheng, A., Friedman, E.: Manipulability of PageRank under Sybil Strategies. In *Proc. of 1st Workshop on the Economics of Networked Systems*, USA, 2006.

[10] Christodoulopoulos, C. E., Papanikolaou, K. A.: Investigation of Group Formation Using Low Complexity Algorithms. In *Proc. of Workshop on Personalization in E-Learning Environments at Individual and Group Level of User Modeling 2007*, Corfu, Greece, pp. 57–60, 2007.

[11] Driskell, J., Radtke, P., Salas, E.: Virtual teams: Effects of technological mediation on team performance. In *Group Dynamics: Theory, Research, and Practice*, 7(4), pp. 297–323, 2003.

[12] Douceur, J.: The Sybil Attack. In *Proc. of 1st International Workshop on Peer-to-Peer Systems*. Cambridge, MA, pp. 251–260, 2002.

[13] Duck, S.: *Human Relationships*. SAGE Publications, London, 2007.

[14] Forsyth, D. R.: *Group Dynamics*. Wadsworth Publishing, 2005.

[15] Friedman, E., Resnick, R., Sami, R.: Manipulation-Resistant Reputation Systems. In *Algorithmic Game Theory*. Cambridge University Press, 2007.

[16] Gogoulou, A., et al.: Forming Homogeneous, Heterogeneous and Mixed Groups of Learners. In *Proc. of User Modeling 2007*, Greece, 2007.

[17] Graf, S., Bekele, R.: Forming Heterogeneous Groups for Intelligent Collaborative Learning Systems with Ant Colony Optimization. In *Proc. of Intelligent Tutoring Systems 2006*, Taiwan, 2006.

[18] Grudin, J.: Computer-Supported Cooperative Work: History and Focus. *Computer*, 27 (5), IEEE, pp. 19–26, 1994.

[19] Haas, S., Stafford, L.: Maintenance behaviors in same-sex and marital relationships: A matched sample comparison. *Journal of Family Communication*, *5*(1), pp. 43–60, 2005.

[20] Hartup, W.: Cooperation, close relationships, and cognitive development. In *The company they keep: Friendship in childhood and adolescence*. Cambridge University Press. pp. 213–237, 1998.

[21] Hertel, G., Geister, S., Konradt, U.: Managing virtual teams: A review of current empirical research. *Human Resource Management Review*, 15(1), pp. 69–95, 2005.

[22] Hoppe, H. U.: Using multiple student modeling to parameterize group learning. *Artificial Intelligence in Education (AIED 1995)*, AACE, Charlottesville, pp. 234–241, 1995.

[23] Houston, T. K., Cooper, L. A., Ford, D. E.: Internet support groups for depression: A 1-year prospective cohort study. *American Journal of Psychiatry*, 159, pp. 2062–2068, 2002.

[24] Inaba, A., Supnithi, T., Ikeda, M., Mizoguchi, R., Toyoda, J. I.: How Can We Form Effective Collaborative Learning Groups? In *Proc. of Intelligent Tutoring Systems 2000*, Montréal, Canada, pp. 282–291, 2000.

[25] Jarboe, S.: Procedures for enhancing group decision making. In *Communication and Group Decision Making*. Sage Publications, pp. 345–383, 1996.

[26] Jaworski, A., Coupland, N.: *The Discourse Reader*. Routledge, London, 2006.

[27] Jermann, P., Soller, A., Mühlenbrock, M.: From mirroring to guiding: A review of the state of art technology for supporting collaborative learning. *European Conference on CSCL*, pp. 324–331, 2001.

[28] Kelley, H. H., et al.: *Close Relationships.* Percheron Press, 2002.

[29] Kumar, V.: Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, Vol. 13. Issue 1, pp. 32–44, 1992.

[30] Lai, K., et al.: Incentives for cooperation in peer-to-peer systems. In *Proc. of Workshop on the Economics of Peer-to-Peer Systems*, 2003.

[31] Leone, N., et al.: The DLV system for knowledge representation and reasoning. In *ACM Transactions on Computational Logic (TOCL)*, 7(3), pp. 499–562, 2006.

[32] Levi, D.: *Group Dynamics for Teams*. Sage Publications, 2007.

[33] Maes, P.: How To Do The Right Thing. *Connection Science*, 1(3), pp. 291–323, 1998.

[34] Malinowski, B. K.: The problem of meaning in primitive languages. In: *The Meaning of Meaning*. Routledge & Kegan Paul, 1923.

[35] Moreno, M., Vivacqua, A., de Souza, J.: *An Agent Framework to Support Opportunistic Collaboration*. Springer, 2003.

[36] Mühlenbrock, M.: *Action-based Collaboration Analysis for Group Learning*. IOS Press, 2001.

[37] Mühlenbrock, M.: Analyzing collaborative learning interactions in shared workspaces. *Künstliche Intelligenz*, 1/03, pp. 37–39, 2003.

[38] Ounnas, A., Davis, H. C., Millard, D. E.: A Metrics Framework for Evaluating Group Formation. *ACM Group'07*, Florida, USA, 2007.

[39] Ounnas, A., Davis, H. C., Millard, D. E.: A Framework for Semantic Group Formation. In *roc of 8th IEEE International Conference on Advanced Learning Technologies*, Spain, 2008.

[40] Parks, C., Sanna, L.: *Group performance and interaction*. Westview, 1999.

[41] Reeves, B., Nass, C.: *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*. Center for the Study of Language and Information, 2003.

[42] Resnick, P., Friedman, E.: The social cost of cheap pseudonyms. *Journal of Economics & Management Strategy*, 10(2), pp. 173–199, 2001.

[43] Roschelle, J., Teasley, S.: The construction of shared knowledge in collaborative problem solving. *Computer-Supported Collaborative Learning*, Springer, pp. 69–97, 1995.

[44] Soh, L., Khandaker, N., Liu, X., Jiang. H.: A computer-supported cooperative learning system with multiagent intelligence. In *Proc. of Autonomous Agents and Multi-Agent Systems*, 2006.

[45] Soller, A., Cho, K. S., Lesgold, A.: Adaptive Support for Collaborative Learning on the Internet. In *Proc. of ITS Workshop on Adaptive and Intelligent Web-based Systems*, Montreal, Canada, 2000.

[46] Soller, A., Martínez-Monez, A., Jermann, P., Mühlenbrock, M.: From mirroring to guiding: A review of state of the art technology for supporting collaborative learning. *International Journal or Artificial Intelligence in Education*, 15(4), pp. 261–290, 2005.

[47] Supnithi, T., Inaba, A., Ikeda, M., Toyoda, J., Mizoguchi, R.: Learning goal ontology supported by learning theories for opportunistic group formation. *International Conference on Artificial Intelligence in Education (AIED 1999)*, France, pp. 263–272, 1999.

[48] Svennevig, J.: *Getting Acquainted in Conversation*. John Benjamins, Philadelphia, 1999.

[49] Tate, D. F., Zabinski, M. F.: Computer and Internet applications for psychological treatment: Update for clinicians. *Journal of Clinical Psychology*, 60, pp. 209–220, 2004.

[50] Tuckman, B.: Developmental sequence in small groups. *Psychological Bulletin*, 63, pp. 384–399, 1965.

[51] Tuckman, B. W., Jensen, M. A.: Stages of small-group development. *Group and Organizational Studies*, 2, pp. 419–427, 1977.

[52] Wessner, M., Pfister, H. R.: Group formation in computer-supported collaborative learning, *ACM SIGGROUP*, Colorado, USA, pp. 24–31, 2001.

10
SEMANTIC-BASED NAVIGATION IN OPEN SPACES

Michal Tvarožek

The Web is a *large open information space*, where *large* means that the information space contains many – millions or even billions of information artifacts, while *open* means that virtually anyone can modify its contents in a more or less unrestricted, syntactically correct way.

In general, navigation can be defined as movement and orientation in an information space. Web navigation can be defined more specifically as "the activity of following links and browsing web pages" (Levene & Wheeldon, 2004), where movement corresponds to the following of links, and orientation corresponds to the knowledge of one's position and selection of links to follow. In practice, the navigation problem (i.e., users getting lost) and issues concerning information overload significantly impair search and navigation experience for many users.

Due to these problems and properties of the Web, truly effective navigation means for the web environment have yet to be realized. Adaptive navigation aims to address these issues by adapting the hyperlinks and their associated visualization. For example, it may add new links to related pages, annotate links with additional information, highlight relevant links or remove broken or unimportant links.

We describe the similarities and differences between Web and Semantic web navigation also from the graph perspective, and introduce navigation as means for search. Next, we outline the concept and goals of adaptive navigation, while also providing an overview of existing navigation types, navigation and orientation tools, and visualization options and approaches. Lastly, we conclude this section with a comparative overview of existing navigation and visualization solutions.

10.1 Web Navigation vs. Semantic Web Navigation

Since the Web can be seen as a network of documents linked via hyperlinks, it can be represented as a *directed graph* where nodes represent documents (information artifacts) and edges represent hyperlinks. Similarly, the Semantic Web is a network of resources linked via relations, which can also be represented via a directed graph. Thus, web navigation is the process of moving via edges from one graph node to another.

Typical web navigation involves the presentation of a single graph node (web page) at a time. However, in the Semantic Web, the presentation of multiple resources at once seems more practical due to the different granularity of information and the availability of both data and metadata as opposed to the Web. For example, a job offer page contains all data about the specific job offer, while in the Semantic Web, the job offer would be represented as several related instances, e.g. one for the job offer, one for the employer, one for each requirement, and one for contact information.

Consequently, in Semantic Web navigation we move or modify a window, which defines the presented resources. In the trivial case this can be reduced to moving the center of the window, between graph nodes via edges. In the job offer example, the window would be centered on the job offer instance and also contain other directly associated instances (see Figure 10-1). Exploring the properties of, e.g., the employer instance would center the window on the employer instance.

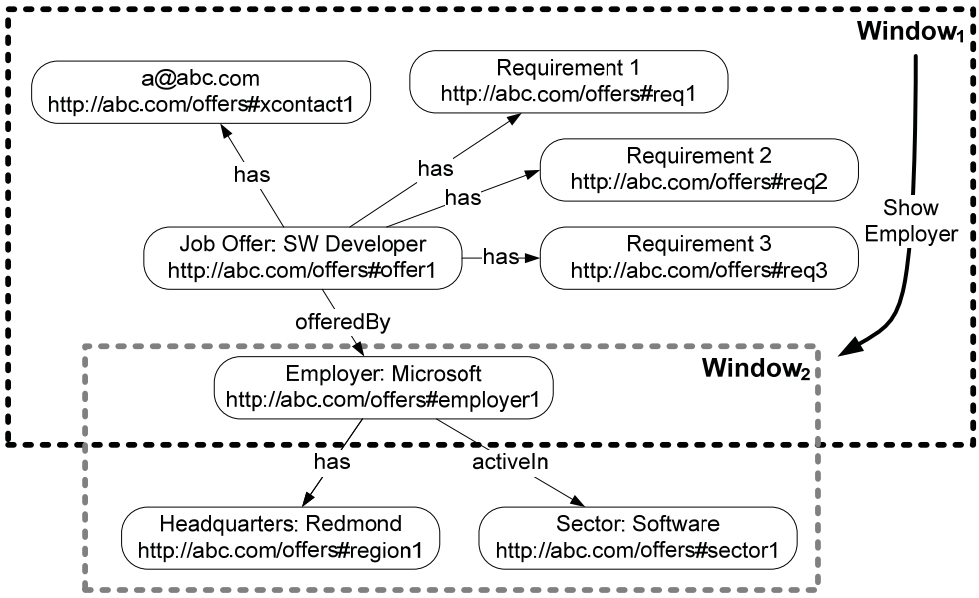


Figure 10-1. Window movement in the Semantic Web, window centers shown in grey.

If however we consider a set of job offers presented simultaneously (e.g., search results), there is no clear node, which might be the window's center (see Figure 10-2).

Furthermore, the Semantic Web effectively contains both data (e.g., job offer instances, employers, requirements) and metadata (e.g., the class JobOffer, Employer and Requirement), and a set of inference rules that can be used to reason on the available information and infer new information. Thus, relations between resources need not be explicitly asserted but can be inferred based on available metadata and rules enabling additional navigation options compared to traditional web navigation.

Hence, we define Semantic Web navigation as the movement and modification of presentation windows containing resource visualizations, based on the following of embedded links corresponding to relations between resources.

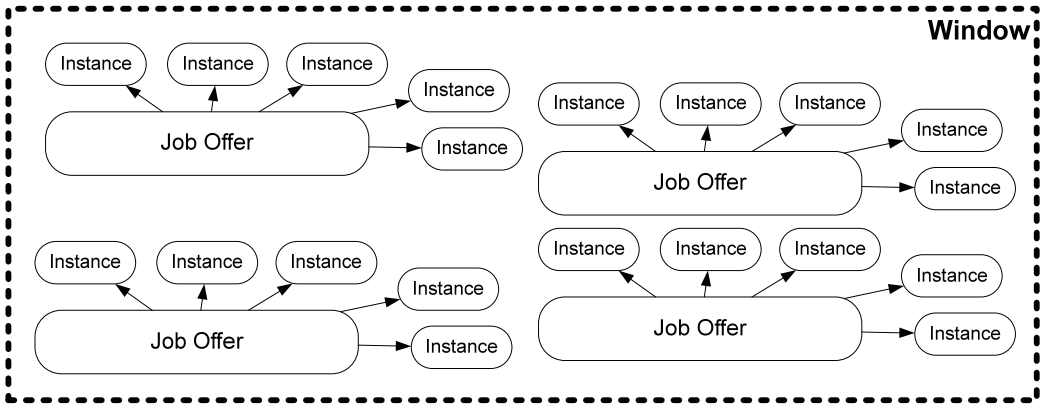


Figure 10-2. Presentation window without a specific center.

10.2 Searching by Means of Navigation

Query-by-example (Geman, 2006) and view-based approaches enable users to search by means of navigation. More specifically, they allow users to construct search queries via navigation with the immediate evaluation of the query and optional query modification and/or refinement.

Faceted browsers as examples of view-based search allow users to navigate the associated faceted classification of the information space. Their link (i.e., facet and restriction) selection is transformed into a search query effectively leading to visual query construction. This approach requires the existence of a (manually) predefined faceted classification scheme that is used to construct the faceted browser interface.

Due to the availability of metadata in a Semantic Web scenario, a faceted classification scheme can be (semi)automatically derived from the used ontology scheme(s). Moreover, even greater expressive power can be achieved by using the entire set of available metadata and its possible aggregations for view-based search. Ultimately, this allows users to visually construct and evaluate *semantic queries* (e.g., in SeRQL or SPARQL), which are otherwise difficult to write even for experienced users, thus negating one principal disadvantage of semantic search.

10.3 Navigation Models

Navigation models correspond to the specific organization of navigation in a particular information space such as a web site. Presently, different navigation models or their combinations are employed based on the application domain and the respective application goals (Rocketface Graphics, 2007).

Furthermore, no clear classification of individual navigation models seems to exist, as many sources use the same names for different models. Based on different sources, we devised the following classification of navigation models:

- *Linear navigation* – the successive browsing through a list of pages, such as search results returned from a search engine.
- *Hierarchical navigation* – the browsing through a hierarchical (tree) structure of a site using the main menu in large web sites or online shops.

– *Faceted navigation* – the browsing via a view-based interface employing a faceted classification of the information space (effectively a forest), such as searching for job offers or products in an online shop.

In practice, individual navigation models correspond to different navigation types and are realized using different navigation and orientation tools (see Section 10.5), which when placed on a web page allow users to choose from a variety of navigation options.

10.3.1 Navigation Types

We distinguish different navigation types within an information space:

– *Local navigation*, which links nodes in the current information subspace (e.g., a cluster of similar nodes) and allows users to navigate in the nearby vicinity of their current position.

– *Global navigation*, which is persistent throughout all views (e.g., web pages) and links the current view with landmark nodes in the web graph thus providing quick access to all major hubs in the information space.

– *Contextual navigation*, which links related nodes throughout the information space and is often realized as contextual links within the text or as lists of "see also" links.

– *Supplemental navigation*, which includes other navigation tools such as site maps, indexes and guides. These include special nodes with high out degrees as well as specially created sequences of nodes (trails).

10.3.2 Linear Navigation

Linear navigation is a straightforward navigation approach used mainly for local navigation in unstructured information spaces, when browsing lists of data (e.g., search results) or when following (pre)defined navigation trails (e.g., guides). This translates to local navigation along a path in the web graph, which has few outgoing edges, "ideally" only one per node (or two for reciprocal navigation).

During linear navigation, users successively browse one page after another. *"Straight line" linear navigation* allows users to proceed to the next page in the sequence (see Figure 10-3), whereas *reciprocal linear navigation* also supports backward links to previously visited pages (see Figure 10-4).

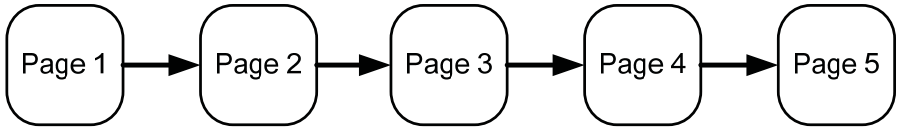


Figure 10-3. The linear "straight line" navigation model supports one-way linear movement in an information space.

The disadvantage of linear navigation is that it takes too long to examine all available data and the user usually has little information about the content of the following pages. For example, during search results browsing most users view only the first page, while only few go beyond the second or third page. One possible advantage of linear navigation is its

simplicity, universal usage and perhaps the unavailability of complex navigation options, which lessens information overload (i.e., the user is unlikely to get lost).

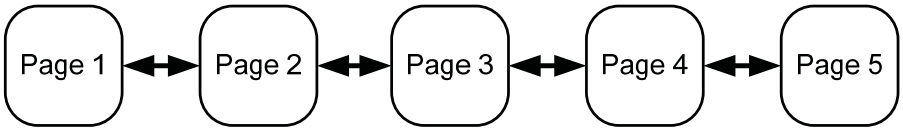


Figure 10-4. The linear reciprocal navigation model supports two-way linear movement in an information space.

10.3.3 Hierarchical Navigation

Hierarchical navigation is based on a hierarchical classification of an information space (e.g., of the pages within a web site). It is often used for global navigation around large web sites with a predefined structure, where it enables users to search on the site by selecting items from the classification. The main menus (i.e., the navigation bar) of web sites often correspond to the hierarchical structure of the sites and thus the hierarchical classification of information provided.

From the graph perspective, hierarchical navigation somewhat corresponds to navigation in hierarchic clusters – the homepage represents the entire top-level cluster, while its children and descendants represent the main divisions and subdivisions. Figure 10-5 depicts the classical *hierarchical navigation model*, which corresponds to a tree. Different sources also call this model the *database* or the *grid navigation model*.

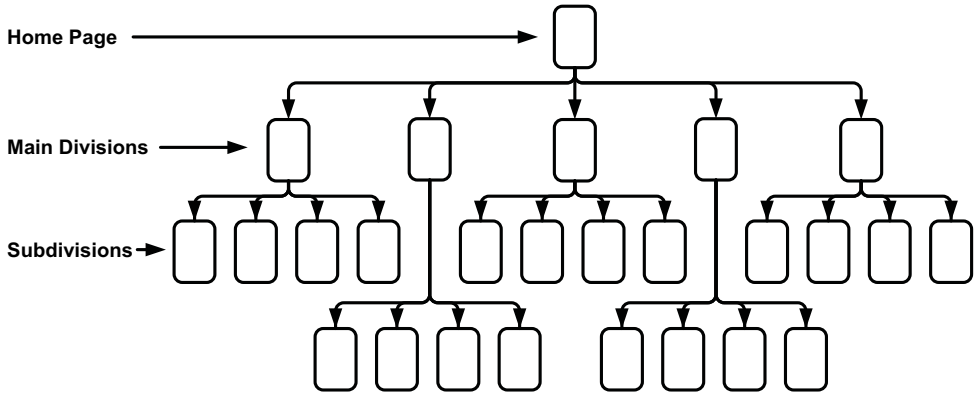


Figure 10-5. The hierarchical navigation model effectively subdivides the information space into smaller independent subspaces.

An advantage of hierarchical navigation is that it provides users with information about the content of other pages and thus enables them to find relevant information quicker. For example, by following the link "notebooks" in a hierarchical navigation model in an online shop, one might expect that the following page will contain information about notebooks.

The obvious disadvantage is the necessity of a hierarchical classification scheme and the fact that navigation paths are predefined by the respective classification. Furthermore, in the case of large or complex classifications, the understanding of what path to take and

what the content of the following pages might be is difficult as is the maintenance of the classifications themselves.

Some practical applications (e.g., larger web sites) require smoother navigation experience with both hierarchical and lateral navigation in the information space in order to make relevant nodes easily accessible from other branches of the tree. Figure 10-6 depicts the *Web navigation model*, which combines hierarchical and linear navigation into a complex interlinked model, which allows easy access from the root (i.e., homepage) to all other sections and/or databases. Some sources also reference this model as *hierarchical navigation* or *site-wide navigation*.

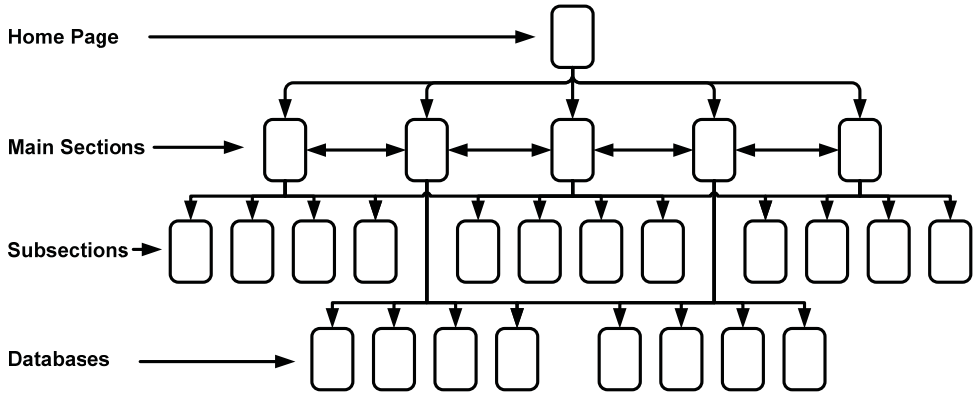


Figure 10-6. The web navigation model provides quick access by combining hierarchical and lateral navigation.

10.3.4 Faceted Navigation

The faceted navigation model is based on a faceted classification scheme (The Knowledge Management Connection, 2006) of an information space. Originating in library sciences, *"a faceted classification differs from a traditional one in that it does not assign fixed slots to subjects in sequence, but uses clearly defined, mutually exclusive, and collectively exhaustive aspects, properties, or characteristics of a class or specific subject. Such aspects, properties, or characteristics are called facets of a class or subject, a term introduced into classification theory and given this new meaning by the Indian librarian and classificationist S.R. Ranganathan and first used in his Colon Classification in the early 1930s."* (Wynar & Taylor, 1992, p. 320). For online information retrieval and navigation however, the library definition of faceted classification can be somewhat relaxed, e.g. the exhaustiveness is not strictly necessary.

Faceted navigation is widely used by faceted browsers in practical applications (Adkisson, 2005). Example applications include many online shops or information retrieval systems built around databases, e.g. for job search. As opposed to hierarchical navigation, faceted navigation is almost exclusively used for dynamic systems, which generate all views at runtime, due to the exponential number of possible facet and restriction[1] combinations.

---

[1] Each facet consists of a (hierarchical) set of its values – restrictions. For example, New York or Washington are restrictions in a facet describing location.

In practice, users can easily select the desired information by accessing one or more facets available in the used faceted classification and selecting one or more restrictions in those facets. Users actually create faceted queries by navigating and *selecting metadata* (i.e., facets and restrictions respectively), thus specifying the *data* (i.e., results) that should be retrieved (see Figure 10-7).

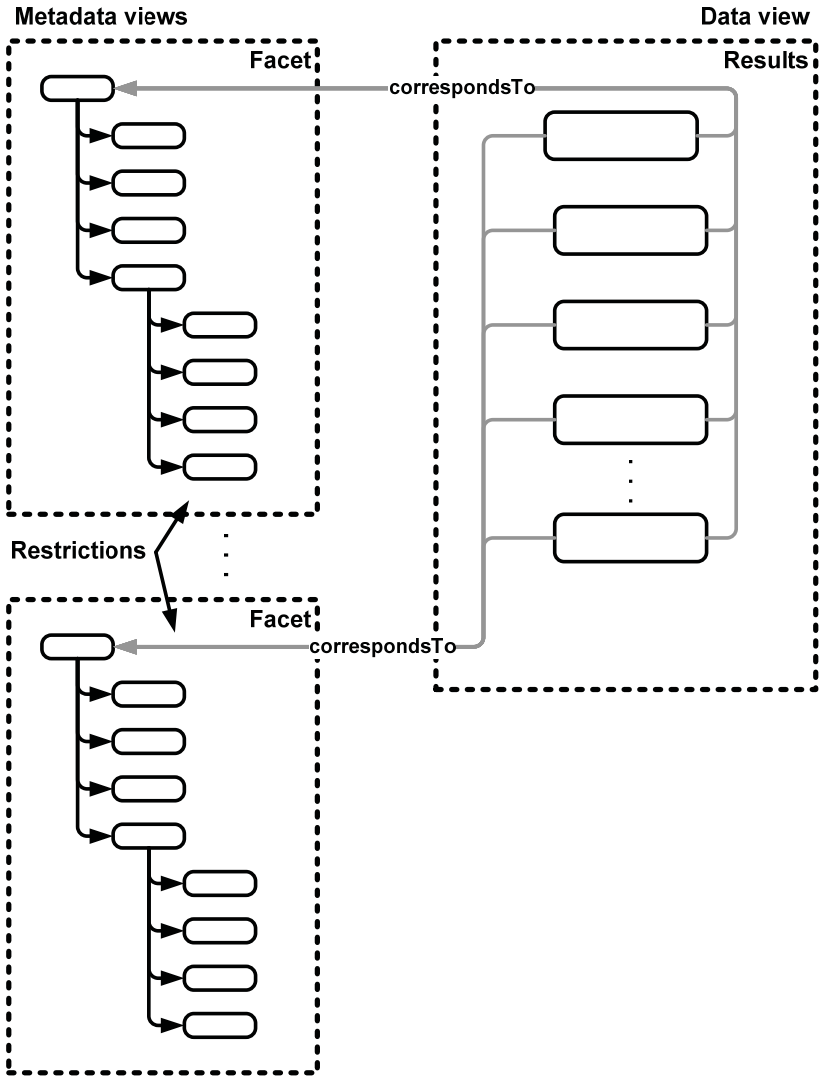


Figure 10-7. The faceted navigation model provides effective access to information via navigation in multiple metadata views (facets) whose combination describes the global browsing state.

This effectively translates into multidimensional hierarchical navigation in metadata describing a particular information domain or, in graph terms, simultaneous navigation in multiple tree hierarchies (i.e., a forest) as individual facets are often hierarchically organized. The combined navigation state from all facets then defines the global navigation state and the presentation window, which shows the faceted query results.

Advantages of faceted navigation include its flexibility and expressivity – users can navigate the information space in many different ways and combine elements from various facets to specify their information need.

Disadvantages of faceted navigation originate mostly from properties of faceted classifications, which do not provide quick access to popular topics and at first might be difficult to understand due to their scope. Furthermore, a faceted browser interface is somewhat more complex which might results in cognitive overload if too much information is available.

The true strength of *faceted navigation* lies in the fact that it *corresponds to view-based search* – it natively provides users with integrated search and navigation capabilities thus alleviating several disadvantages of traditional search approaches (e.g., difficult query construction, unsuitability for open-ended tasks).

10.4 Adaptive Navigation

The goal of "good" navigation would be to prevent users from getting lost, inform them about their current position, and suggest useful options for future navigation while protecting users from information overload. To accomplish this goal, adaptive navigation takes advantage of personalization based on user context and customizes existing or generates entirely new navigation and orientation tools according to current adaptation goals and estimated user needs.

This section provides a high level view of adaptive navigation with respect to specific adaptation goals and adaptive navigation techniques.

10.4.1 Adaptation Goals

All of the aforementioned navigation types can be adapted and/or personalized via specific adaptation techniques with respect to the adaptation goals relevant for a particular application (Brusilovsky, 1996):

- *Local guidance* which helps users with the selection of the next link to follow via link sorting and recommendation.
- *Global guidance* which guides the user towards his global information goal by recommending an optimal trail (path through the web graph) through the information space. For example, in an educational system this would mean learning only the necessary concepts omitting unnecessary distractions.
- *Local orientation support* which provides users with information about their current position and the nearest surroundings. It often employs link annotation, which describes the suitability of links for further navigation, and hides less relevant links thus preventing information overload.
- *Global orientation support* which provides the user with a global overview of his/her absolute position in the information space and its structure, mostly using link annotation.

10.4.2 Adaptation Techniques

Current adaptive navigation approaches take advantage of these techniques:

- *Direct guidance* is a technique that recommends the next link or sequence of links, which should be visited by a user, usually via a "next" button. It is often used in educational systems, where the next link to be visited is determined based on user knowledge about specific concepts.
- *Adaptive link sorting* changes the order of existing links. Until now, it was mostly used in closed information spaces but was not very well accepted by users who were confused by the constantly changing order of links, which the users expected at their original locations. However, adaptive link sorting might be employed in open information spaces, which often change by their sole definition. Moreover, in large information spaces, sorting is the only sensible means of finding the required information and is used by all search engines. Consequently, *adaptive link sorting* and *link generation* should not have adverse affects on user experience in large open information spaces.
- *Adaptive link generation* adds new links to related resources, e.g., based on the results of data mining techniques.
- *Adaptive link hiding* can be subdivided into these techniques:
  - *Link hiding*, when links are clickable but look like normal text.
  - *Link disabling*, when links are not clickable yet look like normal text; used for contextual links.
  - *Link removal*, when links are physically removed; used for non-contextual links.
- *Adaptive link annotation*, as perhaps the most used technique, adds additional information to selected links, whose purpose is to more closely describe the target web pages thus helping the user to decide which link to follow next.
- *Map adaptation* adapts the local or global sitemap respectively.

We devised a method for adaptive faceted browsing in the Semantic Web and developed a prototype adaptive faceted semantic browser – *Factic* (Tvarožek & Bieliková, 2007). Figure 10-8 illustrates examples of common adaptation techniques on a sample GUI from *Factic*. We focused on the adaptation of facets and restrictions (left), which includes

- *direct guidance* via recommended restrictions (green background).
- *link generation* – all links for restrictions are generated (left).
- *link sorting* – individual facets are sorted by relevance, restrictions are sorted alphabetically (left).
- *link annotation* via tooltips and traffic light colors for individual restrictions and background color for facets (left).
- *link hiding* by hiding irrelevant facets, which are available on demand (left).

10.5 Navigation and Orientation Tools

The most common means for navigation on the Web is the web browser, which retrieves, renders and displays web pages, and allows users to navigate the Web by means of following links.

Current browsers support a list of features which aid users during browsing. These can be divided into:

- *Browser-based tools*, which are implemented in the browser itself and are thus independent from the viewed information space and its contents (e.g., bookmarks or the forward and back buttons).
- *View-based tools*, which are contained in individual views and originate from and/or are provided by the information space itself (e.g., link annotation on a web page).

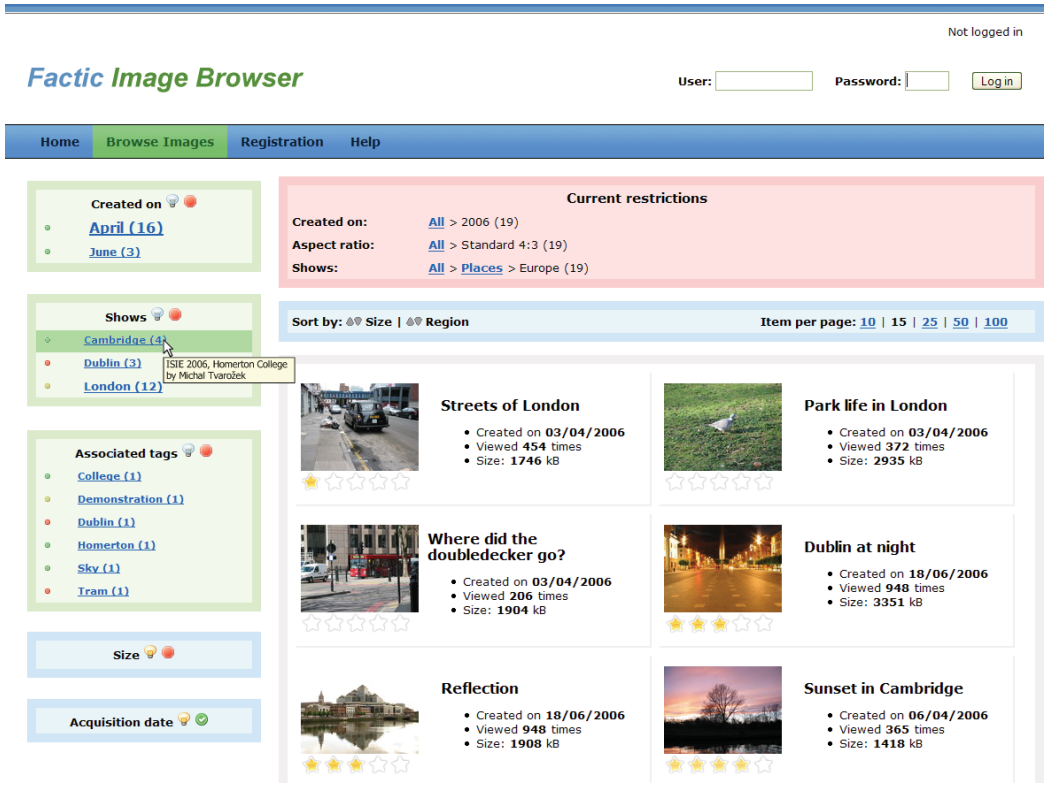


Figure 10-8. Examples of adaptation techniques used by our adaptive faceted browser Factic. Adaptation, annotation and recommendation of facets (left), adaptation and annotation of search results (center).

10.5.1 Browser-Based Navigation Tools

Several browser-based navigation tools have already been developed and successfully implemented in practical solutions. Other tools were proposed and evaluated with promising results, yet are still unavailable in mainstream web browsers:

- *Back button*, which enables users to backtrack their path through the information space. It is actually one of the most used navigation tools accounting for about 14 % of clicks, being second only to following links that account for about 43 % of clicks (Weinrich et al., 2006). Two paradigms for the back button were explored – stack based and history based, though studies indicate that there is no significant difference between the two for common users (Levene & Wheeldon, 2004).

- *Forward button*, which works together with the back button but is used only marginally in practice since users mostly return to previously visited pages.
- *Home button*, which is used only marginally in practice and enables users to return to a known location in the information space (e.g., a search engine).
- *History list*, whose use is extremely limited as it only offers a simple list of recently pages (e.g., today, last week or last month). Since it contains many pages, is not organized in any sensible way and does not support searching, it cannot be effectively used to revisit pages.
- *Bookmarks*, which allow users to mark a page for future reference and thus work as a selective history list, which can be organized into folders and subfolders. While some users use bookmarks to store already visited pages not suitable for further navigation, bookmarks are good for future page revisits as they allow users to "search" based on a hierarchical classification of folders. However, the number of bookmarks greatly increases over time thus making effective bookmark management difficult, and their overall usage and usefulness limited.
- *History tree*, which organizes recently visited pages in a tree instead of a list and provides users with a good overview of their recent browsing history. Different approaches exist, which may visualize the whole history for multiple sessions or only the history of the current navigation session or domain. In (Nadeem & Killam, 2001) the authors compare two tree based approaches (GlobalTree – shows the current session, and DomainTree – shows individual domains) with the history list approach of common web browsers and conclude that users prefer tree based history behavior instead of history lists.

10.5.2 View-Based Navigation and Orientation Tools

Typically different navigation and orientation tools are employed in order to simplify user access and improve user experience (Levene & Wheeldon, 2004):

- *Link markers* or *embedded links* are the basic means of navigation by means of inserting and highlighting links directly into the presented information via text or images thus corresponding to contextual navigation (Figure 10-9, D).
- *Navigation bars* contain a list of links and are often used for global navigation – site menus on the left/top/right side of the page (Figure 10-9, A).
- *Bread crumbs* simplify the organization of information and display the user's current position in the information space together with the path that brought him/her there (Figure 10-9, B). They improve user orientation and can be effectively used both for global and local navigation.
- *Tabs* effectively subdivide the content into multiple parts and can be used for linear local navigation (Figure 10-9, C).
- *Site maps*, as means for supplemental navigation, provide a global overview on the main link and content structure of the information space (see Figure 10-10). A fisheye view for sitemaps displays only the most relevant part of the map around the current user position.

Figure 10-9. Examples of navigation and orientation tools – the navigation bar (A), bread crumbs (B), tabs (C) and link markers (D).

- *Landmark nodes* are prominent nodes within an information space or the respective subspace the user is browsing. They are best identified by the high number of nodes that can be reached within two links from the landmark node and which can reach it within two links.
- *Guided tours* or *trails* aid users in navigation by suggesting whole trails (sequences of links) to follow usually via local linear navigation. They are often used in educational systems where teachers can author them based on the recommended order of learning for individual concepts. More advanced trails would include automatically created trails or collaborative trails.

10.5.3 Faceted Browsers

Faceted browsers provide advanced navigation and orientation means, which can be used to provide access to information spaces via faceted navigation. While all of the aforementioned navigation and orientation tools would work for an entirely static web site, faceted browsers are dynamic by their very nature and thus may only be used in web-based applications. For example, online shops often employ faceted browsers for navigation in the products they offer.

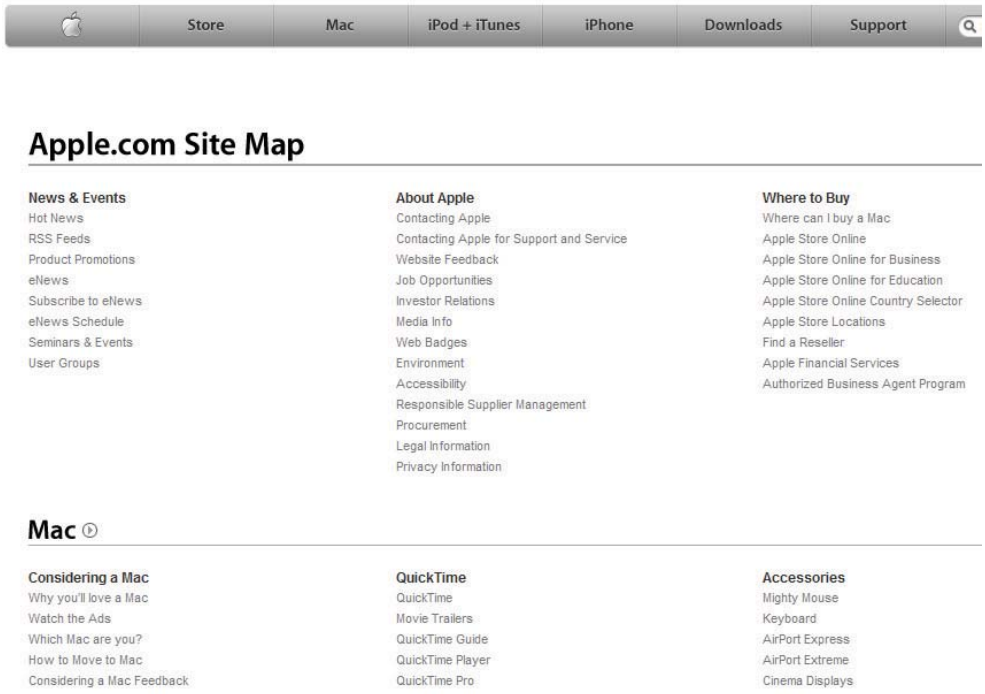


Figure 10-10. Example of a simple sitemap.

Figure 10-11 outlines the typical browsing process in a faceted browser, which corresponds to the steps performed during view-based search:

1. *Query* – users typically select facets and restrictions as long as they match their perceived (and known) information needs.
2. *Selection* – once the set of available options is exhausted or the users cannot think of any more criteria, they examine the search results and select promising results for further navigation.
3. *Navigation* – detailed information about "good" results can be retrieved and a navigation session via their properties or associated resources can be initiated (e.g., showing associated resources or comparing similar ones).
4. *Query modification* – users can relax the query by removing restrictions and repeating the process from step 1.

In addition to faceted navigation, contemporary faceted browsers support additional features for simple "processing" of the displayed data (search results):

- Simple sorting of instances based on one given attribute (e.g., name, price or weight, screen size, popularity).
- The comparison of several selected instances and their attributes in a table.
- Different views which are either more or less detailed, with or without images and with a selectable number of simultaneously displayed results.

– Different actions with search results, such as bookmarking, adding to the shopping cart or rating.

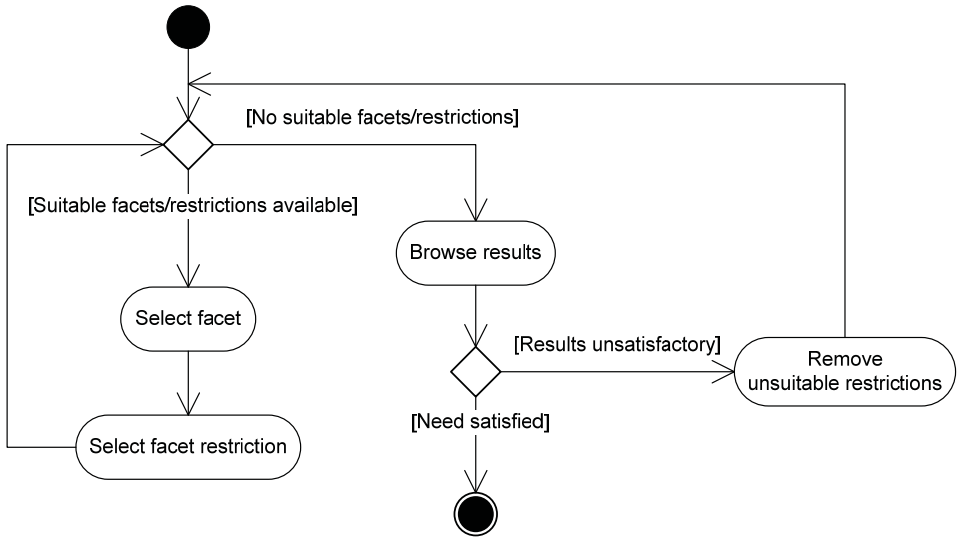


Figure 10-11. The navigation process in a simple faceted browser.

Figure 10-12 shows an example graphical user interface of our faceted browser *Factic* in the domain of job offers. As such, it is a faceted browser employing primary and secondary facets with multilevel content. It supports both nominal and ordinal facet values – enumerations and intervals respectively (Adkisson, 2005).

Individual facets for the type of the offered position, location, industry sector, start date, job term and contract type are shown on the left. The current query is shown at the top, while the results of its evaluation are displayed in the center. For each search result, the title of the job offer and its main attributes are shown. Additional operations with results include their sorting, rating and optionally editing.

10.6 Navigation Visualization and Content Presentation

The visualization and presentation aspect of web-based applications plays a major role in their success and user acceptance. This concerns the overall layout and graphical design of applications, the use of color, styles, fonts etc., which have a major impact on whether users will like or hate an application. Much can be accomplished by following "tried and true" usability guidelines (Nielsen, 2007) and taking advantage of research performed by the HCI community.

In our work however, we do not focus on the design of specific user interfaces, but rather on their conceptual functionality and behind the scenes processing with relation to navigation, user understanding and personalization of information with respect to large open information spaces.

From the navigation perspective it is important to understand how links are visualized and how the users interact with them. "Classical" navigation used only simple links, which were placed at specific locations in the presented document, whose visualization

was less important. In the Semantic Web context, the data and metadata are "more equal" and proper metadata (i.e., navigation) visualization approaches are desirable. Especially if many generated links should be presented to the user, they must be organized in a sensible and easy to understand way.

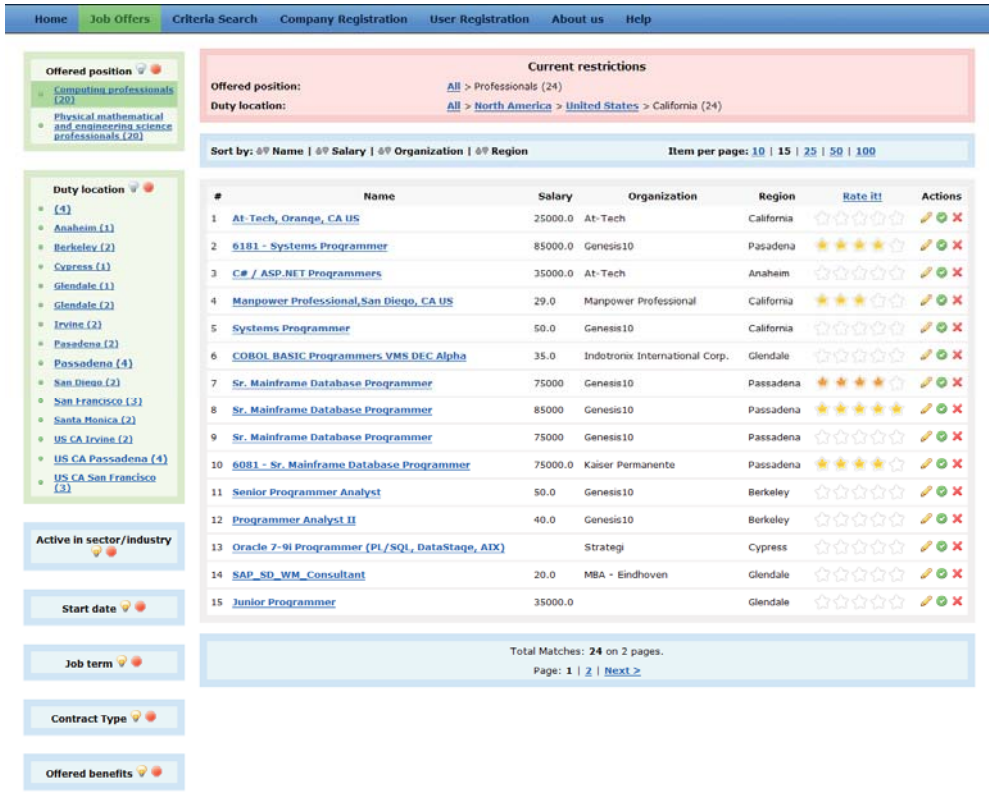


Figure 10-12. Sample GUI of our faceted browser Factic. Facets are shown on the left (green and blue backgrounds), results in the center (white background), current restrictions and available actions at the top (pink and blue background).

– Classical "text-based" visualization is most often used in simple interfaces, where links are represented via text or images (see Figure 10-13).

Figure 10-13. Example of classical text-based navigation visualization. Links are implemented as text (blue) or images respectively.

– Visual navigation uses interactive visualization approaches, which might depict the structure of the information space to provide users with a better description of the information domain (Figure 10-14). For example, graph visualization approaches (Al-

lIKnow[2]) or hierarchical clustering approaches can be implemented via Java applets, using images or via SVG. Proper algorithms for the layout and abstraction level are required as user interfaces can quickly become overloaded with information for large or complex information spaces.

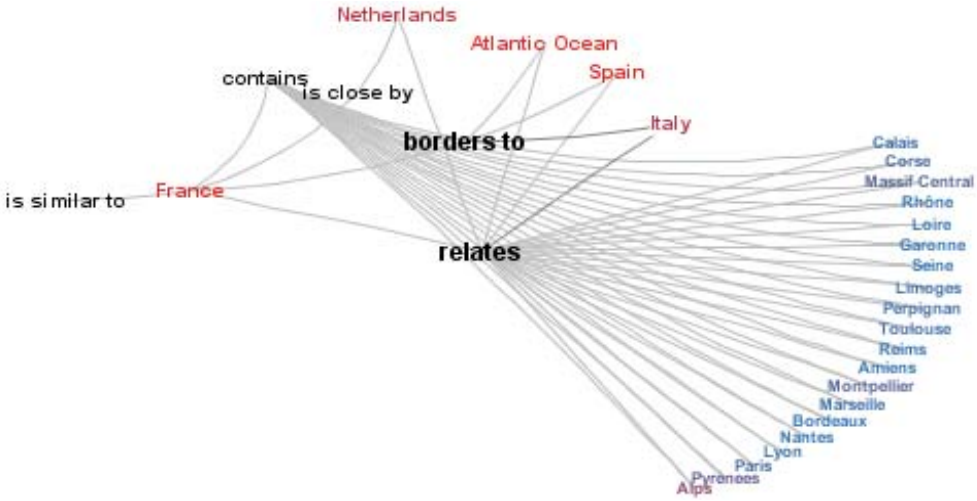


Figure 10-14. Example of visual navigation using graph visualization techniques in project AlIIKnow. Links are realized as graph nodes.

– Spatial visualization is often implemented via plug-ins, which enable users to move freely in a two- or three-dimensional space and optionally interact with the objects it contains. The most common visualization approaches are virtual maps, such as Google Maps[3] (2D) or Geonova[4] (3D) (see Figure 10-15). However, the use of spatial (3D) visualization on the Web is as of today not very common.

Other visualization approaches, which focus on the presentation of metadata can also be successfully used to visualize link structures and thus navigation.

CropCircles is a topology sensitive approach to visualization of OWL class hierarchies inspired by treemaps (Wang & Parsia, 2006). Since it visualizes (class) hierarchies, it might be ideally suited for the visualization of facets, which contain restriction hierarchies (see Figure 10-16). CropCircles provide quick overview of the topology (i.e. the size, depth and complexity of a hierarchy), while also providing a visually pleasing nested presentation of individual nodes.

[2] AllIKnow: `http://alliknow.net/` (last viewed 16.11.2006)

[3] Google Maps: `http://maps.google.com/` (last viewed 18.8.2009)

[4] Geonova: `http://www.geonova.ch/gvista/pages/ch/GVista.htm` (last viewed 18.8.2009)

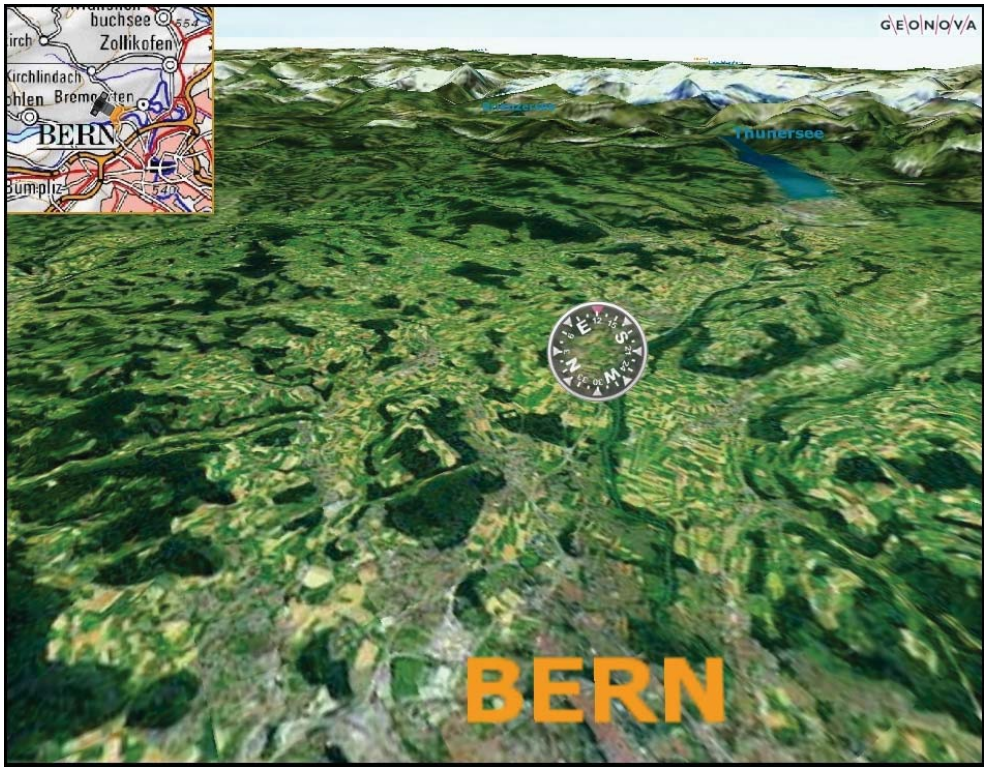


Figure 10-15. Example of spatial visualization in the GVista system by Geonova.

Circles represent nodes, their size corresponds to the size of the respective subtree rooted at a particular node. Child nodes are sorted in descending order based on their size. Different layout strategies are employed based on the size distribution of child nodes (e.g., dominant child node, equal sized children).

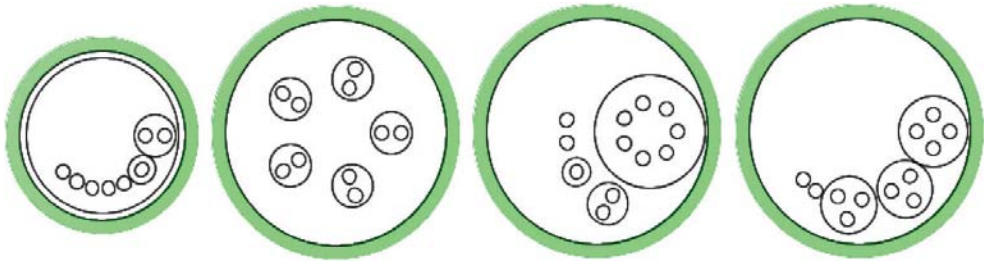


Figure 10-16. Topology sensitive visualization via CropCircles, taken from (Wang & Parsia, 2006). Layout strategies from left to right – single child, equally sized children, dominant child, no dominant child.

Figure 10-17 shows TagSphere – an approach to tag visualization for augmented content-based image retrieval using collaborative tagging (Aurnhammer, Hanappe, & Steels, 2006). The white circle in the center denotes the user's image collection yet also a query-by-example. Next, the Tag sphere – sets of search results corresponding to tags from the query are drawn. Their size denotes the number of images, distance to the white circle denotes

the number of overlapping images and the circles in their respective centers denote the overlap returned by an image classifier, which evaluates low-level image properties against the query. The outer Classifier sphere works the same way, yet describes a different set of results, which are returned by the classifier instead of a tag search. E.g., for sets *leaves* and *park* the tags seem to match the low-level image properties quite well, while having high overlap with the user's collection.

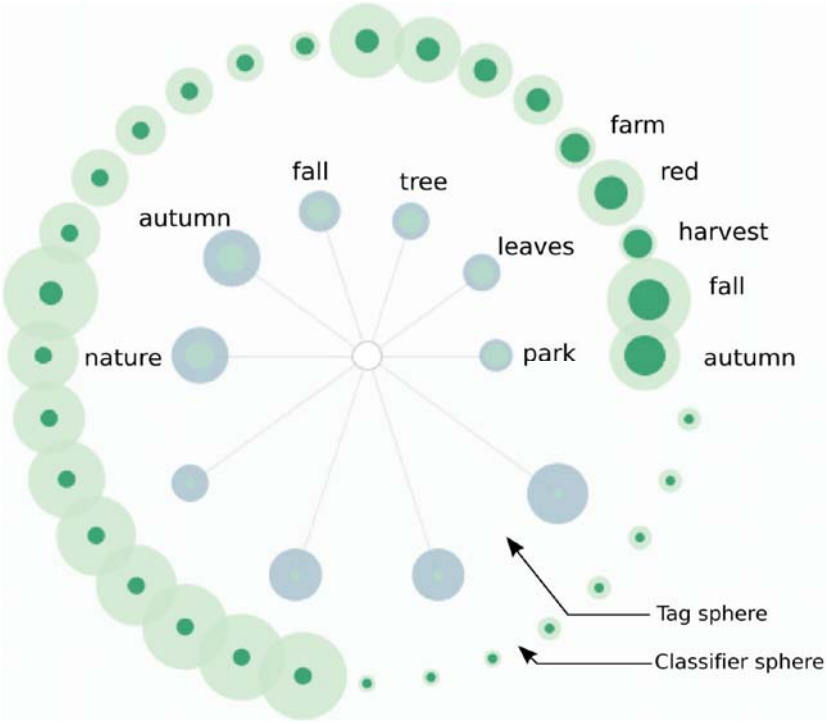


Figure 10-17. Tag visualization via TagSphere,
taken from (Aurnhammer, Hanappe, & Steels, 2006).

10.7 Existing Navigation Solutions

Based on our prior analysis we identified faceted browsers and view-based search as suitable navigation means for large information spaces based on the Semantic Web, mainly due to their user friendly GUI, high expressivity via faceted classification and overall suitability for large data collections. Consequently, we examined the properties of several existing faceted browsers:

- *Factic*: Adaptive faceted semantic browser (Tvarožek & Bieliková, Personalized Faceted Navigation for Multimedia Collections, 2007).
- *Ontoviews*: A tool for creating Semantic Web portals (Mäkelä et al., 2004).
- *BrowseRDF*: Faceted RDF browser (Oren, Delbru, & Decker, 2006).
- */facet*: Browser for heterogeneous semantic repositories (Hildebrand, van Ossenbruggen, & Hardman, 2006).
- *IGroup*: Image search engine (Wang, Jing, He, Du, & Zhang, 2007).

- *Flamenco*: FLexible information Access using MEtadata in Novel COmbinations (Yee, Swearingen, Li, & Hearst, 2003).
- *Relation Browser++* (Zhang & Marchionini, 2005).

Section 10.7.6 provides an overview of the main properties of individual solutions.

10.7.1 Factic: Adaptive Faceted Semantic Browser

We devised a method for personalized faceted navigation (Tvarožek & Bieliková, Personalized Faceted Navigation for Multimedia Collections, 2007) and implemented a prototype faceted browser Factic (Tvarožek & Bieliková, Personalized Faceted Navigation in the Semantic Web, 2007). Factic is an adaptive faceted semantic browser for OWL which supports semiautomatic configuration of the underlying faceted classification based on the used domain ontology.

The GUI of Factic copies the standard faceted browser layout with facets on the left, current query at the top and search results in the center (see Figure 10-12). Users can navigate the available facet hierarchy via view-based search and examine the details of individual instances, for which the respective properties are displayed recursively (see Figure 10-18).

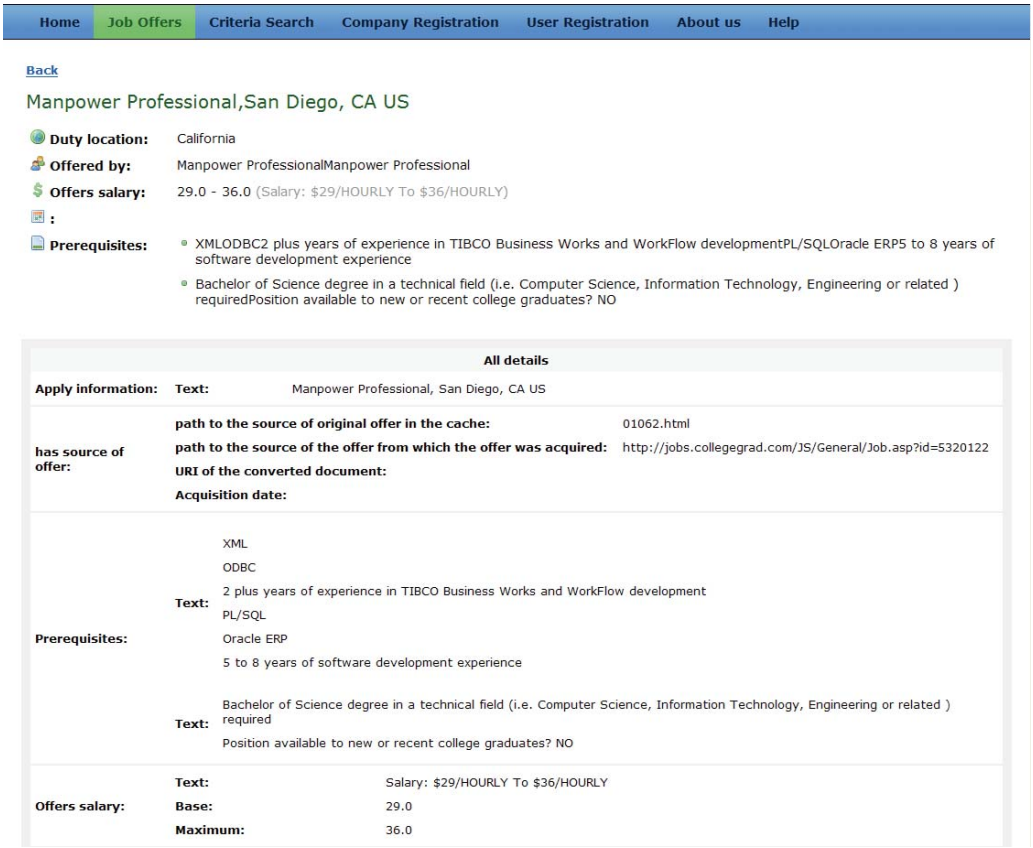


Figure 10-18. Example of the Factic GUI for presentation of instance details. Main attributes are shown at the top, other attributes are recursively expanded (bottom).

The main advantage of Factic is personalization support based on an automatically acquired user model also via external user modeling tools (Andrejko et al., 2006). Personalization includes the adaptation of facets and facet restrictions (e.g., active facets, inactive facets, disabled facets) their annotation with additional information and recommendation via ordering or background color.

Important is also the adaptation, annotation and recommendation of search results with support for external personalized evaluation tools. Moreover, view adaptation can customize the presentation of search results based on instance type (e.g., images in an image matrix with thumbnails, job offers in a table).

Currently one important disadvantage is the slow performance of the underlying database engine with semantic queries for larger datasets, which results in high response times. Another possibly limiting performance factor might be the use of the Apache Cocoon framework[5] and XML/XSLT transform view design pattern for presentation rendering.

10.7.2 Ontoviews: A Tool for Creating Semantic Web Portals

OntoViews (Mäkelä et al., 2004) is a comprehensive tool for the creation of Semantic Web portals based on the Apache Cocoon framework and a service oriented architecture using Ontogator as a view-based search service (Mäkelä et al., 2006). Ontoviews supports faceted navigation over RDFS ontologies and link recommendation services via Ontodella.

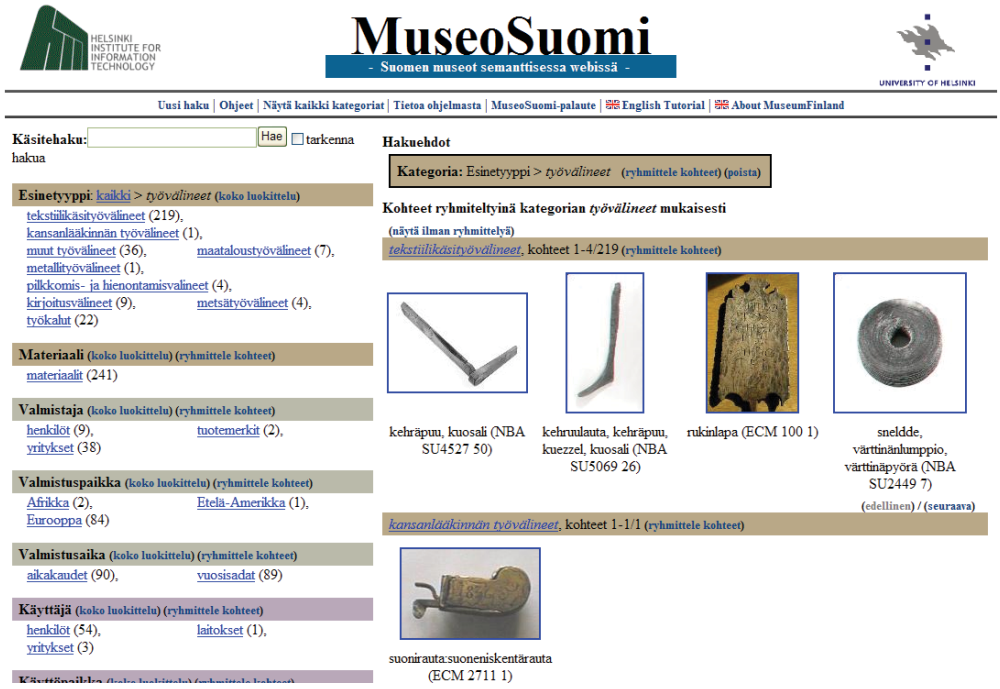


Figure 10-19. Example of the OntoViews GUI, facets shown on the left, search results shown on the right.

[5] Apache Cocoon: `http://cocoon.apache.org/` (last viewed 18.8.2009)

A demonstration application of OntoViews is publicly available in the domain of digital libraries (museums) as MuseoSuomi[6]. Figure 10-19 shows the user interface of OntoViews, which copies the typical faceted browser layout with facets on the left and content on the right. Search results are presented in groups corresponding to the last used facet. The detailed instance view (see Figure 10-20) shows instance attributes at the top, followed by a list of faceted categories to which the instance belongs. Recommended links to related instances are shown on the right.

Figure 10-20. Example of the OntoViews GUI for presentation of instance details with Instance attributes (top), other facet categories (center) and related instances (right).

Furthermore, OntoViews has a mobile user interface, which retains the functionality of the original desktop interface albeit with minimal screen size.

Link generation is based on predicates in the form *p(subjectURI, targetURI, explanation)*, which succeed when two resources (*subjectURI, targetURI*) should be linked together with label *explanation*. Individual rules/predicates are processed by the Ontodella service for link generation.

[6] MuseoSuomi: `http://www.museosuomi.fi/` (last viewed 18.8.2009)

The use of XSLT in the user interface and query transformations provide high interface flexibility, yet resulted in complicated templates that are tied to a specific RDF/XML representation. Moreover, OntoViews does not take advantage of OWL metadata and must be manually configured to use facets and link recommendation (e.g., via aforementioned rules). It also has no support for personalization based on user preferences.

10.7.3 BrowseRDF: Faceted RDF Browser

BrowseRDF (Oren, Delbru, & Decker, 2006) is a faceted browser for Semantic Web data in RDF format. BrowseRDF can automatically generate a faceted interface from arbitrary RDF data with little manual configuration.

BrowseRDF extends typical faceted queries with RDF semantics, e.g. existential selection, inverse selection, non-existential selection. Furthermore, it defines statistical metrics for automatic facet ranking and adaptation, such as predicate balance, object cardinality and predicate frequency.

Figure 10-21 shows the GUI of BrowseRDF in the domain of wanted FBI suspects. Individual facets with new selection types are shown on the left, instance details are shown in the center.

Similarly to OntoViews, BrowseRDF does not take advantage of OWL data and automatically generates facets for all available RDF predicates, even those with little sense for the end users. Moreover, it only employs statistical metrics computed from the supplied RDF data and thus supports no personalization, nor link recommendation.

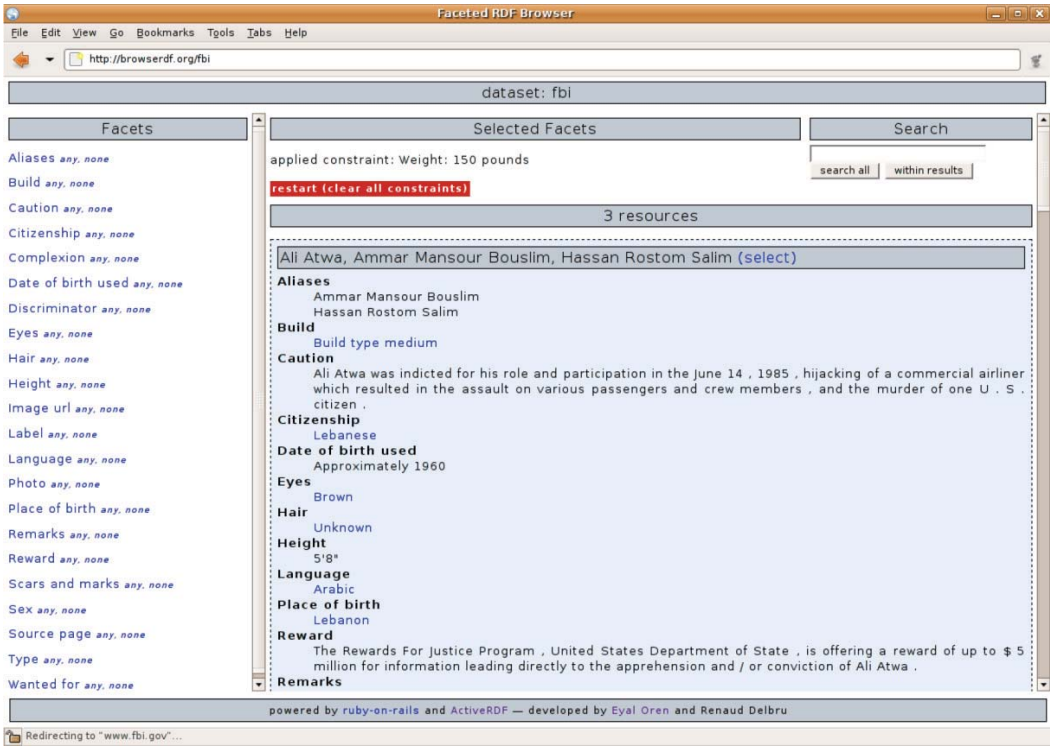


Figure 10-21. Example of the BrowseRDF GUI with existential facet restrictions (left), taken from (Oren, Delbru, & Decker, 2006).

10.7.4 /facet: Browser for Heterogeneous Semantic Repositories

/facet (Hildebrand et al., 2006) is a faceted browser for heterogeneous information spaces consisting of distributed semantic repositories represented in RDFS. It takes advantage of both the *rdfs:subClassOf* property and the *rdfs:subPropertyOf* property in order to process facet restriction hierarchies.

Furthermore, /facet supports multi-type queries and runtime facet specification thus greatly increasing flexibility and support for heterogeneous repositories. The multi-type capability effectively translates into an additional facet, which is used to specify the target data type. Based on the selection in the type facet, other facets are made available.

Figure 10-22 shows the /facet GUI. The selected type *vra:Work* corresponds to facets Creator, Date and Material.Medium. Moreover, /facet supports semantic keyword search, which allows users to perform keyword-based search on

- all instances (helps find a suitable instance type),
- individual facets (improves movement and restriction selection),
- and across all facets (improves orientation).

Lastly, /facet supports the grouping of search results based on individual properties and timeline visualization of dates. However, it does not support personalization nor advance link generation and recommendation techniques.

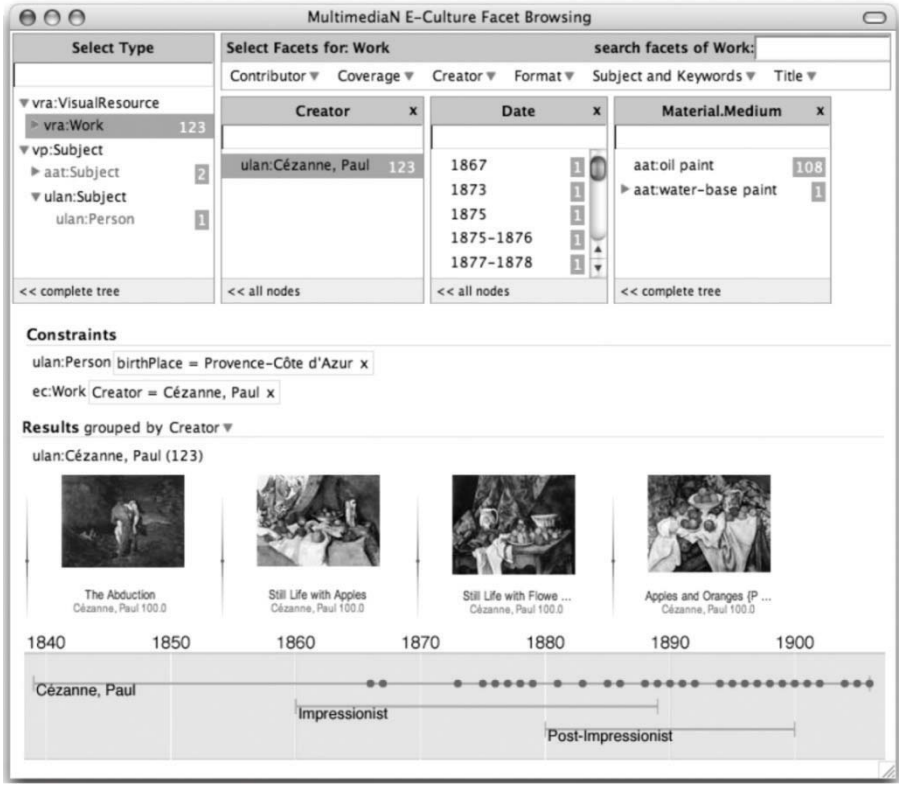


Figure 10-22. Example of the /facet GUI with multiple facets (top), constrained search results (center) and timeline display (bottom), taken from (Hildebrand, van Ossenbruggen, & Hardman, 2006).

10.7.5 IGroup: Image Search Engine

IGroup (Wang et al., 2007) is a typical keyword-based search engine in the image domain. However, it presents search results in semantic clusters that users can use for search via query-by-example thus effectively expanding their query options.

Figure 10-23 shows the IGroup GUI, with a list of identified clusters on the left. These correspond to different "tigers" identified in the use collection and allow users to select specific subspaces of the information space as in view-based search. Individual search results are presented in a matrix in the center of the GUI with some descriptive information.

The clustering algorithm takes as input the results of a standard keyword-based search and gives a list of annotated clusters as its output. It takes advantage of text, which is available for individual images and selects top-ranked phrases via n-gram analysis (phrase frequency, document frequency, phrase length, etc.).

Advantages include a wider coverage where some minor, previously hidden, subsets are now visible. Furthermore, individual clusters are annotated while allowing users to refine the query based on the displayed images instead of writing keywords.

Disadvantages include no support for personalization and link generation and no direct support for Semantic Web data as the source data results from a traditional keyword-based query to some other search engine.

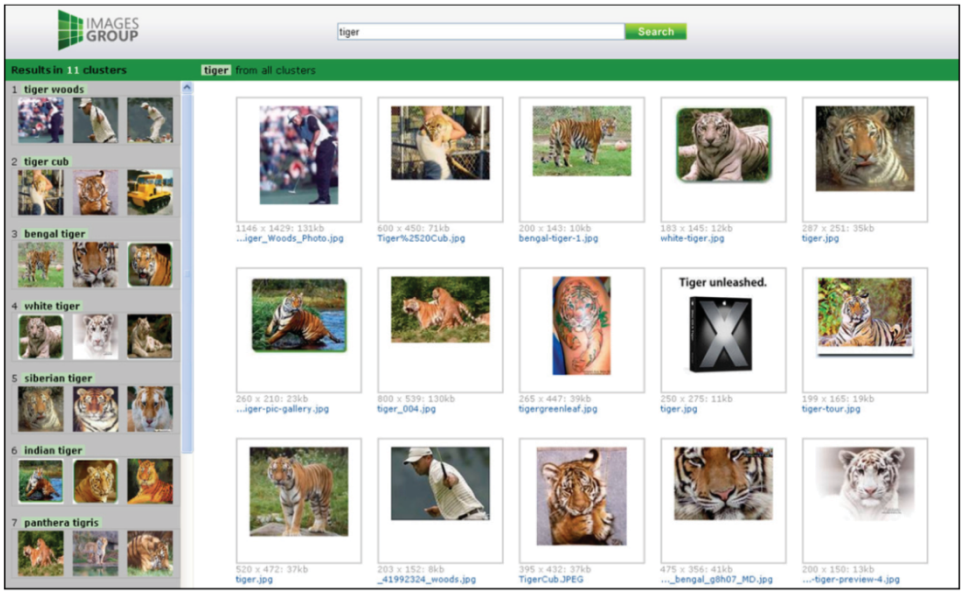


Figure 10-23. Example of the IGroup GUI with identified clusters (left), taken from (Wang et al., 2007).

10.7.6 Comparison of Navigation Solutions

We compared seven existing navigation solutions with focus on faceted navigation and view-based search (Table 10-1). Most of the examined solutions offered limited support for Semantic Web data in the form of RDF/RDFS ontologies, though several had no support at all.

Several solutions provided support for link generation though this was limited to links between instances related via predefined properties or clusters. None of the examined solutions apart from our faceted browser Factic supported personalization, although there was limited support for content-based adaptation – linking to related instances and sorting based on statistical metrics.

The most used visualization approach seems to be textual visualization in tables. No solution employed any kind of advanced visual/graph-based presentation approach for navigation, or for actual content though /facet employed timeline visualization to show temporal distribution of instances.

Table 10-1. Overview of navigation solutions.

	Semantic support	Link generation	Adaptivity	Personalization	Visualization
Factic (faceted browser)	OWL	Yes	Adaptation, annotation and recommendation of facets, restrictions, results	Yes, automatic user modeling	Text/images and tables
OntoViews (faceted browser)	RDFS	Yes, related instances	Related instances	No	Text/images and tables
BrowseRDF (faceted browser)	RDF	Yes	Statistical facet ranking	No	Test and tables
/facet (faceted browser)	RDFS	Yes	No	No	Text/images and tables, timeline
IGroup (keyword-based, query-by-example)	No	Yes, clusters	No	No	Text/images and tables
Flamenco (faceted browser)	No	Yes	No	No	Text/images and tables
RB++ (view-based search)	No	No	No	No	Text and tables

10.8 Looking Ahead

We identified several problems in the previous sections, such as the size and dynamics of open information spaces, which current information retrieval systems must deal with. While existing search engines are definitely not ideal, they are still pretty good at what they do – gathering metadata and building indices. Although, there is steady progress in keyword-based search engine improvement, we believe that this area is not interesting enough to warrant further attention and thus leave it to others. Furthermore, additional approaches already exist that can classify documents (web pages) and create metadata where they are unavailable (albeit they are not used much in practice and of somewhat questionable quality).

What current solutions seriously lack is usability and support for advanced integrated search and browsing interfaces that support typical usage scenarios (e.g., not only search, but also generic browsing or overviews). It was shown that users today cannot effectively use the possibilities offered by today's full-text search engines, which leads us to believe that the more complex semantic search engines will be even more underused (in terms of possible search options). Thus advanced approaches which combine searching and browsing with navigation appear to be a promising and very relevant field of further research, which includes the:

- Design of methods and models for search and navigation in large (open) information spaces based on the Semantic Web.
- Design of user interfaces (in terms of functionality) for these methods.
- Design of models and methods for user collaboration and communication.
- Design of methods and models of user adaptation of the respective methods to the specific requirements of users based on user contexts.
- Design, preparation and execution of experiments and case studies aimed at evaluating the proposed methods.

More specifically, visual construction of (semantic) search queries via a faceted browser seems interesting; however a more complete information retrieval interface seems necessary. This would likely include:

- A visual depiction of the available information space for visual navigation based, e.g. on a graph of clusters or relationships between concepts.
- An adaptive graph-/tree-based history browser, which might even replace the bookmark system of current web browsers, as it would be sensitive to current user requirements. History items might possibly be displayed in "full resolution" for a selected number of clicks.
- Trail navigation support in addition to links.
- Collaboration options that would make users aware of the actions or overall trends in navigation and content usage and/or credibility.

While semantic versions of faceted browsers were already proposed and experimented with, to our best knowledge, limited (if any) work was done on the possibilities of faceted browser adaptation with optional extensions of the faceted browser navigation model. Thus faceted browsers as means for integrating view-based search and navigation are another promising field of research.

Extending and formalizing the navigation process for different stages such as the opening, middle game, endgame outlined in (Yee et al., 2003) is of interest:

- Metadata navigation first identifies what metadata should be used.
- Data navigation then identifies what instances should be examined.
- Data processing and exploration ultimately facilitate the learning and understanding of the found information/knowledge.

Moreover, these facet and restriction related enhancements seem promising:

- Extending the navigation model with nested facets that enable users to define restrictions on properties of information space instances.
- Dynamic facet generation based on a domain ontology, the query and the user model.
- Adaptation, annotation and recommendation of facets and restrictions based on the global user statistics, user context (environment and device properties), social networks and long-term, short-term and in-session behavior for different user profiles (e.g., at work, at home).
- Aggregate (meta)facets based on data derived from the domain ontology. These can be based on simple aggregations (e.g., COUNT or SUM) or more complex aggregations such as hierarchical clustering based on different clustering functions (e.g., semantic similarity).
- Generation of annotations for aggregate (meta)facets and their restrictions based on data in the domain ontology and the selected aggregation function. Individual annotations can be generated based on representative instances or as enumerations/summarizations of most frequent instance properties.

Since the browsing, understanding and quick interpretation of target information is crucial, we propose improved visualization for faceted browsers and integration with concept visualization solutions displaying detailed information about individual instances using graph or domain specific visualizations:

- Dynamic generation of the entire faceted browser interface and adaptation of views for different content and user preferences.
- Graphical visualization of facets using existing approaches for e.g., geographic locations (maps) or hierarchical clusters (graph), which may improve user understanding of facets and thus orientation support.
- Graphical visualization of search results using graph visualization techniques with support for successive query refinement (e.g., by selecting a cluster of instances) instead of the table/matrix of search result attributes.

Integration with external concept browsers with support for:

- Visualization of instance neighborhoods using graph visualization of instances where graph edges denote similarity. Hierarchical clustering can be used for similarity and abstraction level evaluation.
- Visualization of instance properties with support for the browsing of associated instances and their properties, associated via graph edges corresponding to object properties of instances.
- Domain specific visualization showing multiple related instance simultaneously, or in several parts with direct navigation guidance based on the user model. For example, in the e-learning domain of programming, several concept explanations might be shown in sequence and/or simultaneously, such as a sequence of statements, branching, procedures with their respective fragments – definitions, tasks, questions and solutions.

While many information retrieval methods can be evaluated statistically, e.g., by computing precision and recall statistics, in many cases such exact evaluations cannot be performed for approaches dealing with user interaction and user interfaces for adaptive systems. The evaluation of such approaches can be done via experiments thoroughly described in case studies, such as (Yee et al., 2003) or (Wang et al., 2007).

Since evaluation is crucial, it must be well defined beforehand and performed on several variants of the test system against a baseline system which might either by an existing system suitable for benchmarking or a system made using the best or most common features of comparable systems (Yee et al., 2003).

Furthermore, layered evaluation principles should be employed to effectively separate evaluation of individual stages of the information processing process – data collection, data interpretation, user modeling, adaptation selection and adaptation application (Paramythis & Weibelzahl, 2005).

In order to counteract possible errors due to user preferences, users can be divided into multiple groups which perform the same test scenarios in different orders so that the results between the groups can later be compared to identify potential bias. Furthermore, different usage scenarios should be employed exploiting different user work styles, such as search for a specific set of items (goal oriented, structured), or search for something you like (more creative, unstructured). Individual experiments may or may not be time limited and accompanied by questionnaires aimed at gathering subjective user feedback.

References

[1] Adkisson, H. P.: Use of Faceted Classification. Retrieved October 23, 2007, from Web design practices: `http://www.webdesignpractices.com/navigation/facets.html`, 2005.

[2] Andrejko, A., Barla, M., Bieliková, M., Tvarožek, M.: Softvérové nástroje pre získavanie charakteristík používateľa. In *P. Vojtáš, & T. Skopal (Ed.), Proceedings of DATAKON '06*, pp. 139-148. Brno, Czech Republic, 2006.

[3] Aurnhammer, M., Hanappe, P., Steels, L.: Augmenting Navigation for Collaborative Tagging with Emergent Semantics. In *I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, et al. (Ed.), ISWC 2006: Proceedings of the 5th International Semantic Web Conference*. LNCS 4273, pp. 58-71. Athens, GA, USA: Springer-Verlag, Berlin Heidelberg, 2006.

[4] Brusilovsky, P.: Methods and techniques of adaptive hypermedia. In *User Modeling and User-Adapted Interaction* , 6 (2-3), 87-129, 1996.

[5] Geman, D.: Interactive image retrieval by mental matching. In *Proceedings of the 8th ACM international workshop on Multimedia information* (pp. 1-2). Santa Barbara, California, USA: ACM Press, New York, NY, USA, 2006.

[6] Hildebrand, M., van Ossenbruggen, J., Hardman, L.: /facet: A Browser for Heterogeneous Semantic Web Repositories. In *I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, et al. (Ed.), ISWC 2006: Proceedings of the 5th International Semantic Web Conference*. LNCS 4273, pp. 272-285, Athens, GA, USA: Springer-Verlag, Berlin Heidelberg, 2006.

[7] Levene, M., Wheeldon, R.: Navigating the World Wide Web. In M. Levene, & A. Poulovassilis (Eds.), *Web Dynamics – Adapting to Change in Content, Size, Topology and Use,* pp. 117-151, Springer-Verlag, Berlin Heidelberg, 2004.

[8] Mäkelä, E., Hyvönen, E., Saarela, S.: Ontogator - a semantic view-based search engine service for web applications. In *I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, et al. (Ed.), ISWC 2006: Proceedings of the 5th International Semantic Web Conference*. LNCS 4273, pp. 847-860. Athens, GA, USA: Springer-Verlag, Berlin Heidelberg, 2006.

[9] Mäkelä, E., Hyvönen, E., Saarela, S., Viljanen, K.: OntoViews - A Tool for Creating Semantic Web Portals. In S. A. McIlraith, D. Plexousakis, & F. van Harmelen (Ed.), In *ISWC 2004: Proceedings of the 3rd International Semantic Web Conference*. LNCS 3298, pp. 797-811. Hiroshima, Japan: Springer-Verlag, Berlin Heidelberg, 2004.

[10] Nadeem, T., Killam, B.: A Study of Three Browser History Mechanisms for Web Navigation. In *IV 2001: Fifth International Conference on Information Visualisation,* pp. 13-21, Los Alamitos, CA, USA: IEEE Computer Society, 2001.

[11] Nielsen, J.: *Writing for the Web*. Retrieved July 1, 2007, from useit.com: Jakob Nielsen's Website: `http://www.useit.com/papers/webwriting/`, 2007.

[12] Oren, E., Delbru, R., Decker, S.: Extending Faceted Navigation for RDF Data. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, et al. (Ed.), In *ISWC 2006: Proceedings of the 5th International Semantic Web Conference*. LNCS 4273, pp. 559-572. Athens, GA, USA: Springer-Verlag, Berlin Heidelberg, 2006.

[13] Paramythis, A., Weibelzahl, S.: A Decomposition Model for the Layered Evaluation of Interactive Adaptive Systems. In L. Ardissono, P. Brna, & T. Mitrovic (Ed.), In *UM 2005: Proceedings of the 10th International Conference on User Modeling*. LNCS 3538, pp. 438-442. Edinburgh, Scotland, UK: Springer Verlag, Berlin Heidelberg, 2005.

[14] Rocketface Graphics: Website Navigation. Retrieved December 10, 2007, from *How to Design a Website – Webmasters Tutorial*: `http://www.rocketface.com/organize_website/website_navigation.html`, 2007.

[15] The Knowledge Management Connection: Faceted Classification of Information. Retrieved October 23, 2007, from *The Knowledge Management Connection*: http://www.kmconnection.com/DOC100100.htm, 2006.

[16] Tvarožek, M., Bieliková, M.: Personalized Faceted Navigation for Multimedia Collections. In *SMAP 2007: Proceedings of the 2nd International Workshop on Semantic Media Adaptation and Personalization*. London, United Kingdom: IEEE CS, 2007.

[17] Tvarožek, M., Bieliková, M.: Personalized Faceted Navigation in the Semantic Web. In L. Baresi, P. Fraternali, & G.-J. Houben (Ed.), In *ICWE 2007: Proceedings of the International Conference on Web Engineering*. LNCS 4607, pp. 511-515. Como, Italy: Springer-Verlag, Berlin Heidelberg, 2007.

[18] Wang, S., Jing, F., He, J., Du, Q., Zhang, L.: IGroup: presenting web image search results in semantic clusters. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 587-596). San Jose, California, USA: ACM Press, New York, NY, USA, 2007.

[19] Wang, T. D., Parsia, B.: CropCircles: Topology Sensitive Visualization Class Hierarchies. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, et al. (Ed.), In: *ISWC 2006: Proceedings of the 5th International Semantic Web Conference*. LNCS 4273, pp. 695-708. Athens, GA, USA: Springer-Verlag, Berlin Heidelberg, 2006.

[20] Weinrich, H., Obendorf, H., Herder, E., Mayer, M.: Off the beaten tracks: exploring three aspects of web navigation. In C. Goble, & M. Dahlin (Ed.), WWW 2006: Proceedings of

the 15th international conference on World Wide Web (pp. 133-142). Edinburgh, Scotland, UK: ACM Press, New York, NY, USA, 2006.

[21] Wynar, B. S., Taylor, A. G.: *Introduction to cataloging and classification* (8th ed.). Libraries Unlimited, 1992.

[22] Yee, K.-P., Swearingen, K., Li, K., Hearst, M.: Faceted metadata for image search and browsing. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 401-408, Ft. Lauderdale, Florida, USA: ACM Press, New York, NY, USA, 2003.

[23] Zhang, J., Marchionini, G.: Evaluation and evolution of a browse and search interface: relation browser. In *Proceedings of the 2005 national conference on Digital government research,* pp. 179-188, Atlanta, Georgia, USA: Digital Government Research Center, 2005.

11

ZÍSKAVANIE INFORMÁCIÍ Z WEBU METÓDAMI INŠPIROVANÝMI SOCIÁLNYM HMYZOM

Anna Bou Ezzeddine

Už v roku 1945 Dr. Vannevar Busch v článku As We May Think napísal o potrebe zriadenia multimediálnej digitálnej knižnice, ktorá by zhromažďovala poznatky celého ľudstva. Jednotlivé poznatky mali byť poprepájané prostredníctvom spojení. Koordinoval výskum viac ako 6000 vedcov a žiadal o vytvorenie vzťahu medzi vedcami a sumou nahromadených poznatkov.

Prešlo skoro polstoročie a Bushova futuristická predstava sa čiastočne naplnila v podobe celosvetovej pavučiny (World Wide Web ďalej web). Jeho predstava bola naplnená vznikom dokumentov v digitálnej podobe a ich vzájomným poprepájaním hypertextovými spojeniami. Predstava jednoduchého prístupu k poznatkom celého ľudstva však ešte stále nie je naplnená.

Web je v súčasnosti najväčší a najznámejší informačný zdroj. Obsahuje bilióny medzi sebou prepojených dokumentov nazývaných webové stránky, ktorých autormi sú milióny ľudí. V súčasnosti je najväčším problémom v obrovskom množstve dát nájsť kvalitnú a relevantnú informáciu. Vyhľadávanie užitočných informácií je pre používateľa internetu náročná úloha. Hoci sa používateľ dostane priamo k veľkému množstvu informácií neexistuje sprievodca, ktorý by mu pomohol vybrať najvhodnejšiu informáciu.

V poslednom období sa venuje veľká pozornosť vývoju nových inteligentných techník, ktorých cieľom je transformovať dáta na použiteľnú informáciu. Vyhľadávanie relevantných informácií je zložitý proces, pri ktorom vzniká množstvo problémov. Na riešenie vznikajúcich problémov vznikajú stále novšie a lepšie prístupy, ktoré sú založené na rôznych princípoch a teóriách.

Sociálny hmyz žijúci v kolóniách ako mravce, včely, termity, osy je známy svojimi organizačnými schopnosťami bez akéhokoľvek centrálneho riadenia (Gordon, 1996). Na organizáciu celej kolónie vplýva interakcia medzi indivíduami navzájom, interakcia medzi indivíduami a prostredím a správanie sa samotných indivíduí (Bonabeau, 1999).

V tejto kapitole sa budeme zaoberať problematikou získavania informácií z webu, pričom sa z veľkej časti sústredíme na metódy inšpirované správaním sa sociálneho hmyzu. Jej cieľom je prezentovať priebežné výsledky, ktoré sú založené na aktuálnom stave

vedeckého poznania v tejto oblasti. Opiera sa o poznatky opísané vo viacerých publikáciách a sústreďuje sa na popis možných vylepšení jednotlivých prístupov.

11.1 Získavanie informácií z webu

Web je podľa (Bing Liu, 2007) definovaný ako široká oblasť multimediálnej informáciu získavajúcej iniciatívy s cieľom poskytnúť univerzálny prístup k veľkému množstvu informácií. Jednoduchšie povedané web je počítačová sieť, ktorá umožňuje používateľom jedného počítača prístup k informáciám uloženým na inom počítači prostredníctvom celosvetovej počítačovej siete nazývanej Internet.

11.1.1 História webu

Tim Berners-Lee objavil web v roku 1989. V tom čase pracoval v CERNe (Centre European pour la Recherche Nucleaire) vo Švajčiarsku. Na začiatku sa jeho projekt, v ktorom sa využívala hierarchická organizácia informácií nestretol s podporou vedenia. V roku 1990 získal podporu vedenia pre svoj projekt, v ktorom prezentoval výhody hypertextového systému. Tento systém podľa (Bing Liu, 2007) umožňoval:

- požadovať informáciu uloženú na vzdialenom počítači cez sieť,
- vymieňať informácie v bežnom formáte,
- prepojenie dokumentov jednotlivca s inými dokumentmi prostredníctvom odkazov.

V jeho návrhu bol v podstate načrtnutý distribuovaný hypertextový systém, ktorý sa stal základnou architektúrou webu.

Vo februári roku 1993 Marc Andreesen a jeho skupina z Illionskej univerzity NCSA (National Center for Supercomputing Applications) predviedli Mosaic for X, grafický webový prehliadač (browser) pre UNIX. Ďalšie verzie systému boli včlenené do systémov Macintosh a Windows. Toto grafické prostredie umožňovalo používateľovi jednoduchú obsluhu.

V roku 1994 Jim Clark, zakladateľ Silicon Graphics, sa spojil s Marcom Andreesenom a vytvorili spoločnosť Mosaic Communitations neskôr premenovanú na Netscape Communications. Po predstavení prehliadača Netscape začal obrovský úspech webu. Internet Explorer z Microsoftu prišiel na trh v auguste 1995 a začal konkurovať Netscapu.

Objavenie celosvetového webu Timom Bernersom-Leeom nasledované predvedením prehliadača Mosaic sú často považované za dva najvýznamnejšie faktory, ktoré prispeli k úspechu a popularite webu.

Web by nemohol existovať bez Internetu, ktorý poskytuje komunikačnú sieť pre funkčnosť webu. Vývoj internetu začal počítačovou sieťou ARPANET v čase studenej vojny. Táto sieť vznikla v USA kvôli kontrole raketových striel. Prvé spojenia prostredníctvom ARPANETu boli vykonané roku 1969. V roku 1972 bol ARPANET predstavený na prvej medzinárodnej konferencii počítačov a komunikácie, ktorá sa uskutočnila vo Washingtone. Na tejto konferencii boli prepojené počítače zo 40-tich rôznych miest.

So vzrastajúcim počtom prepojených dokumentov vznikla potreba efektívneho vyhľadávania informácií. Prvý vyhľadávajúci systém EXCITE bol predstavený v roku 1993 študentmi Stanfordskej univerzity. Jerry Yang a David Filo vytvorili v roku 1994 YAHOO systém. Systém GOOGLE vznikol v roku 1998. Jeho tvorcovia sú Sergey Brin a Larry Page.

11.1.2 Neviditeľný – hlboký web

Termín hlboký web začala používať spoločnosť Bright Planet (*http://www. brightplanet.com*). V štúdii (Bergman, 2001) sa uvádza, že hlboký web je 500-krát väčší ako tzv. povrchový web. Mimoriadne dôležitá je kvalita obsahu hlbokého webu, ktorá je 1000 až 2000 krát väčšia ako pri povrchovom webe. Podľa autora sú to stovky miliárd dokumentov prístupných pomocou databáz. Priemerné webové sídlo hlbokého webu má iba za mesiac o 50 % väčšiu hustotu návštev ako sídla na povrchovom webe. Hlboký web je v súčasnosti najrýchlejšie rastúca kategória s novými informáciami na webe. Viac ako polovica obsahu hlbokého webu sa nachádza v špecializovaných predmetových databázach. Až 95 % informácií v hlbokom webe patrí k verejne prístupným informáciám, ktoré sú prístupné bez poplatkov.

Podľa (Sherman, 2001) existujú štyri druhy neviditeľného webu.

1. *Neprehľadný, nejasný web.* Nie je zverejnené do akej hĺbky vyhľadávače jednotlivé stránky indexujú a ako často indexácia prebieha. Vyhľadávače často zistia informáciu o obrovskom množstve vyhľadaných stránok, ale zobrazia iba ich časť. (V roku 2007 najznámejšie vyhľadávače zobrazili najviac 1000 stránok.). Ak na stránku neexistuje odkaz z inej stránky robot takúto stránku nedokáže nájsť a stránka sa stáva súčasťou neviditeľného webu.
2. *Súkromný web.* Tvoria stránky, ktoré úmyselne ich správcovia nezaradili do vyhľadávania a môžu byť prístupné pomocou hesla.
3. *Web prístupný za určitých podmienok.* Väčšinou sú to rozsiahle databázy, ku ktorým je obmedzený prístup napríklad digitálne knižnice. Vyžaduje sa registrácia, alebo zaplatenie poplatku, takže vyhľadávačom je celý obsah nedostupný.
4. *Skutočne neviditeľný web.* Tvoria ho stánky, ktoré sa nedajú zaindexovať z technických príčin. Vyhľadávače sa spočiatku zameriavali iba na spracovanie textových súborov. V súčasnosti mnohé spracúvajú netextové informácie ako obrázky a videá. Informácie sú často nepresné, pretože neobsahujú priamo ich obsah, ale iba informácie získané z textových dokumentov umiestnených v ich okolí.

11.1.3 Metódy na vyhľadávanie informácií

Už z predchádzajúcich charakteristík webu je zrejmé, že v súčasnosti je najväčším problémom nájsť na webe kvalitnú a relevantnú informáciu. S rastom počtu webových stránok a služieb klesá úspešnosť hľadania relevantných informácií. Objavovanie nových informácií je zložitou záležitosťou. V súčasnosti sme svedkami intenzívneho výskumu a vývoja nových indexovacích techník, vývoja nových špecializovaných informačných agentov, nových techník filtrovania, zhlukovania a klasifikácie (Bing Liu, 2007; Sherman, 2001; Baldi, 2003; Witten, 2005; Webb, 2002; Chakrabarti, 2003).

S narastajúcim množstvom informácií vznikajú nové metódy na vyhľadávanie informácií, takže od počiatočného prehliadania dokumentov sa dostávame k nástrojom umožňujúcim objavovanie znalostí. Podľa metód prístupu k informáciám delíme nástroje prístupu do týchto tried:

- nástroje umožňujúce jednoduché prezeranie dokumentov (browsing),
- vyhľadávacie, prieskumové stroje (search engines).

Tieto stroje vyhľadávajú informácie na základe kľúčových slov a ich kombinácií pomocou výrokov Boolovej algebry.

- inteligentní agenti – na základe vopred stanovených podmienok hľadajú a filtrujú informácie; k známym aplikáciám tohto agenta patrí porovnávanie cien tovarov na internete,
- nástroje na objavovanie znalostí (web mining).

11.2 Metódy inšpirované správaním sa sociálneho hmyzu

Sociálny hmyz je hmyz žijúci vo viac, alebo menej organizovaných spoločenstvách s nasledujúcimi črtami:

- spolupráca v starostlivosti o mladé pokolenie,
- reprodukčné rozdelenie práce so sterilnými jedincami pracujúcimi za jedincov zamestnaných reprodukciou,
- prekrytie aspoň dvoch generácií vývojových štádií schopných pracovať v kolónii.

Medzi hmyz, ktorý sa vyznačuje sociálnym správaním patria mravce, včely, osy a termity. Sociálny hmyz zaujíma v oblasti umelého života kľúčové postavenie najmä kvôli relatívnej jednoduchosti správania sa jedinca v spojení so zložitým kolektívnym správaním. Spoločenstvá sociálneho hmyzu sú schopné vyvinúť prostriedky na kolektívne riešenie úloh. Zložitosť týchto úloh o mnoho rádov prevyšuje schopnosti jednotlivca. Dokážu riešiť úlohy bez centrálneho vedenia bez existencie dopredu daných fixných štruktúr aj napriek existencii výrazných vnútorných šumov.

Napríklad mravce majú schopnosť vytvárať dynamicky sa meniace štruktúry a sú schopné nájsť najkratšiu cestu od zdroja potravy do hniezda bez použitia zraku. Sú schopné prispôsobiť sa na zmeny prostredia, napríklad nájsť novú najkratšiu cestu v prípade ak stará je nepoužiteľná kvôli zataraseniu prekážkou. Hlavným prostriedkom, ktoré mravce používajú na formovanie a udržanie spojenia je feromónová stopa. Mravce zanechávajú určité množstvo feromónov a uprednostňujú smer, na ktorom je ich viac. V tomto prípade mravce, ktoré si zvolili kratšiu cestu rýchlejšie zrekonštruujú novú cestu ako tie, ktoré si vybrali dlhšiu. Kratšia cesta získa vyššie množstvo feromónov za jednotku času.

Pochopenie emergentných schopností kolónií mravcov, kolektívneho správania sa spoločenstva bolo inšpiráciou pri návrhu nových metód distribuovaných výpočtov.

Včely v prípade hľadania potravy vďaka používaniu vzdušných ciest nemajú možnosť označkovať si cestu ako mravce. V tomto prípade si včely odovzdávajú informácie o vzdialenosti zdroja potravy tancom.

V roku 1973 rakúsky zoológ Karl Ritter von Frisch dostal Nobelovu cenu za objavenie jazyka tancujúcich včiel (Gadagkar, 1996). Ak sa zdroj nachádza blízko úľa, včela tancuje tzv. kruhový tanec, ktorý neobsahuje informáciu o smere potravy. Pri vzdialenejšom zdroji včely zakomponujú do tanca aj informáciu o smere a vzdialenosti potravy. Tanec sa začne rozťahovať a nadobúda tvar osmičky.

V tejto kapitole bude predstavených niekoľko prác v ktorých autori opisujú správanie sa včiel a možnosti využitia metafory včiel na získanie informácií. V týchto prácach včely predstavujú multiagentový systém. Hlavným problémom v týchto systémoch je spôsob dorozumievania sa agentov (v prípade včiel je to tanec). Agenti nemajú globálny pohľad

na riešený problém, vidia iba svojim lokálnym pohľadom. Systém by mal byť schopný vysporiadať sa s obmedzeniami, ktoré si jednotliví agenti kvôli svojmu lokálnemu pohľadu neuvedomujú. Od agentov sa vyžaduje, aby spolu rozumne kooperovali. Jednou z ciest spolupráce medzi agentmi bez explicitnej komunikácie je využitie inteligencie roja (Bonabeau, 1999; Vries, 1998).

11.2.1 Samoorganizácia v biologických systémoch

V knihe (Camazine, 2003) v kapitolách 12, 15, 16 sa autori zaoberajú popisom samoorganizácie sa včelieho spoločenstva. Spoločenstvo si pomocou jednoduchých pravidiel vyberá najlepší zdroj nektáru. Včely vylietavajú do okolia a hľadajú potravu pre spoločenstvo. Keď včela nájde potravu, vracia sa s ňou do úľa a pritom prináša pre ostatné včely aj správu o zdroji potravy. V tomto procese hľadania potravy má včela nasledujúce možnosti:

- pokračovať v zháňaní potravy bez lákania ostatných včiel,
- zdieľať informáciu o zdroji potravy tancovaním, pritom im oznamuje smer, kvalitu a vzdialenosť zdroja – na tento zdroj sa snaží zlákať aj ostatné včely,
- opustiť zdroj potravy a nechať sa zlákať inou včelou, ktorá propaguje kvalitnejší zdroj potravy.

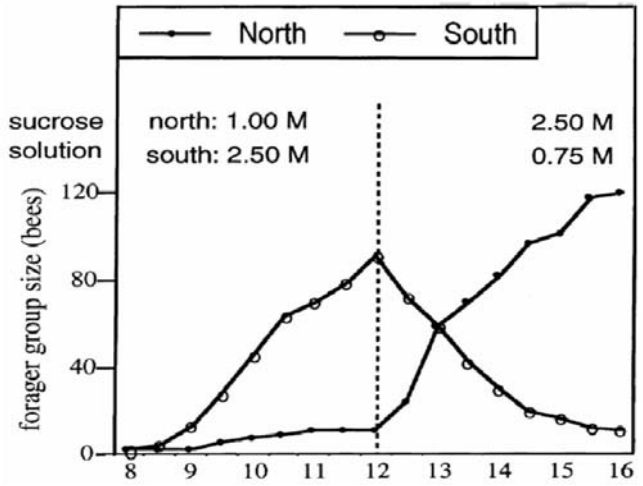
11.2.2 Ako si spoločenstvo vyberá medzi zdrojmi potravy

V práci (Selley, 1991) autori experimentálne dokázali, že rozhodovanie včely v procese hľadania potravy je založené na veľmi obmedzených informáciách získaných z ňou navštívených zdrojov. Napriek takémuto jednoduchému správaniu sa každej včely spoločenstvo si vyberie najlepší zdroj potravy. Tento zdroj si vyberá na základe miery tancovania za lepší zdroj a opúšťania nekvalitného zdroja.

Vo vyššie spomenutej práci autori vykonali experiment zameraný na preskúmanie ako si včelie spoločenstvo vyberá medzi zdrojmi potravy. Experiment vykonali na púšti. V blízkosti včelieho spoločenstva umiestnili dva zdroje potravy. Prvý bol umiestnený 400 m južne od kolónie a druhý 400 m severne od kolónie. Zdroje mali rôznu koncentráciu nektáru. Severný, menej kvalitný mal koncentráciu 1.0 jednotiek, južný bol kvalitnejší s koncentráciou 2,5 jednotiek. Experiment prebiehal v čase od 8:00 do 16:00. O 12:00 boli zdroje vymenené, bola zmenená kvalita zdrojov, takže severný mal koncentráciu 2,5 jednotiek a južný iba 0,75 jednotiek. Na začiatku experimentu bolo 12 včiel vytrénovaných, aby leteli k severnému zdroju potravy a 15 včiel aby letelo k južnému zdroju potravy.

Pri pozorovaní bolo zistené, že počet včiel znášajúcich potravu z kvalitnejšieho zdroja sa časom zväčšoval a počet včiel znášajúcich potravu z menej kvalitného zdroja sa postupne znižoval. Po zamenení zdrojov sa zmenila aj situácia. Severný, po zámene kvalitnejší zdroj navštevovalo postupne čoraz viac včiel, južný, menej kvalitný zdroj začali včely opúšťať.

Na obrázku 11-1 je znázornený výsledok tohto experimentu. Na horizontálnej osi je znázornený čas (8:00-16:00), na vertikálnej osi je znázornený počet včiel, ktoré nosia potravu z vyznačeného zdroja.



Obrázok 11-1. Experiment: Ako si včelie spoločenstvo vyberá medzi zdrojmi potravy (Selley,1991).

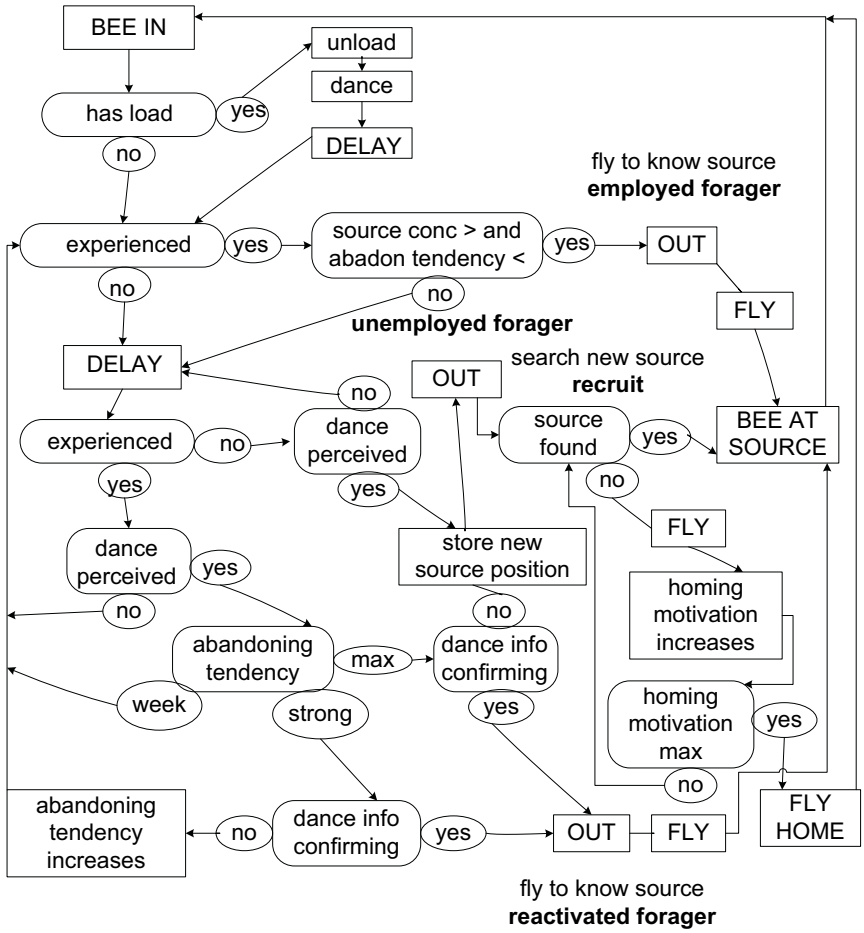
11.2.3 Modelovanie kolektívneho hľadania potravy

V práci (Vries, 1998) sa autori zaoberajú modelovaním kolektívneho zbierania potravy prostredníctvom včiel. Individuálne orientovaný model je skonštruovaný tak, aby simuloval kolektívne správanie sa včiel pri zbere. Každá včela sa riadi rovnakým súborom pravidiel správania. Cieľom autorov v tejto práci bolo vytvoriť simulačný model, ktorý by dosahoval podobné výsledky ako boli pozorované v práci (Selley, 1991). V tomto modeli existuje niekoľko kategórií včiel:

- *zamestnaná včela* – včela, ktorá pozná pozíciu výnosného zdroja, nosí z neho potravu a nenasleduje tancujúce včely,
- *nezamestnaná včela* – včela, ktorá nenosí potravu,
- *včela začiatočníčka,*
- *včela prieskumníčka* – nepozná zdroj potravy a začne ho hľadať spontánne,
- *zlákaná včela* – začne hľadať zdroj potravy potom ako pozoruje tancujúce včely; pozná približnú pozíciu zdroja, ale nepozná jeho kvalitu,
- *skúsená včela* – včela, ktorá pozná pozíciu a kvalitu zdroja potravy,
- *včela inšpektor* – spontánne sa reaktivuje, vykonáva prieskumné lety a navštívi známy zdroj, aby znovu zistila jeho kvalitu,
- *reaktivovaná včela* – reaktivuje sa potom, ako odpozoruje od tancujúcich včiel potvrdzujúcu informáciu,
- *včela prieskumníčka* – spontánne začne hľadať nový zdroj potravy potom, ako sa jej predchádzajúci vyčerpal,
- *zlákaná včela* – začne hľadať nový, doteraz neznámy zdroj potravy potom, ako pozoruje tancujúce včely.

Včela svoju kariéru začína ako nezamestnaná začiatočníčka. Nemá žiadne znalosti o zdrojoch potravy. V tomto prípade môže spontánne začať hľadať zdroj potravy a stáva sa z nej

včela prieskumníčka, alebo po pozorovaní tancujúcich včiel sa stáva zlákanou včelou, ktorá má približnú informáciu o polohe zdroja. Keď včela nájde zdroj potravy, zapamätá si jeho polohu a kvalitu a začne z neho znášať potravu. Takto sa včela stáva zamestnanou včelou. Včela sa opäť môže stať nezamestnanou včelou v prípade, že sa zdroj potravy vyčerpá. Ak si včela v pamäti uchováva informácie o zdroji nazývame ju skúsenou nezamestnanou včelou. Ak včela vykoná prieskumný let ku zdroju, ktorý kedysi opustila nazývame ju inšpektorom. Ak nezamestnaná skúsená včela odpozoruje od tancujúcich včiel pozíciu zdroja, ktorý sa podobá na ňou zapamätaný zdroj, stáva sa reaktivovanou včelou. Ak nezamestnaná skúsená včela odpozoruje od tancujúcich včiel pozíciu predtým neznámeho zdroja, stáva sa odvedenou včelou. Včela môže začať spontánne hľadať nové zdroje potravy v prípade, že nebude nasledovať žiadnu tancujúcu včelu. Tieto pravidlá správania sa včely sú schematicky znázornené na obrázku 11-2.

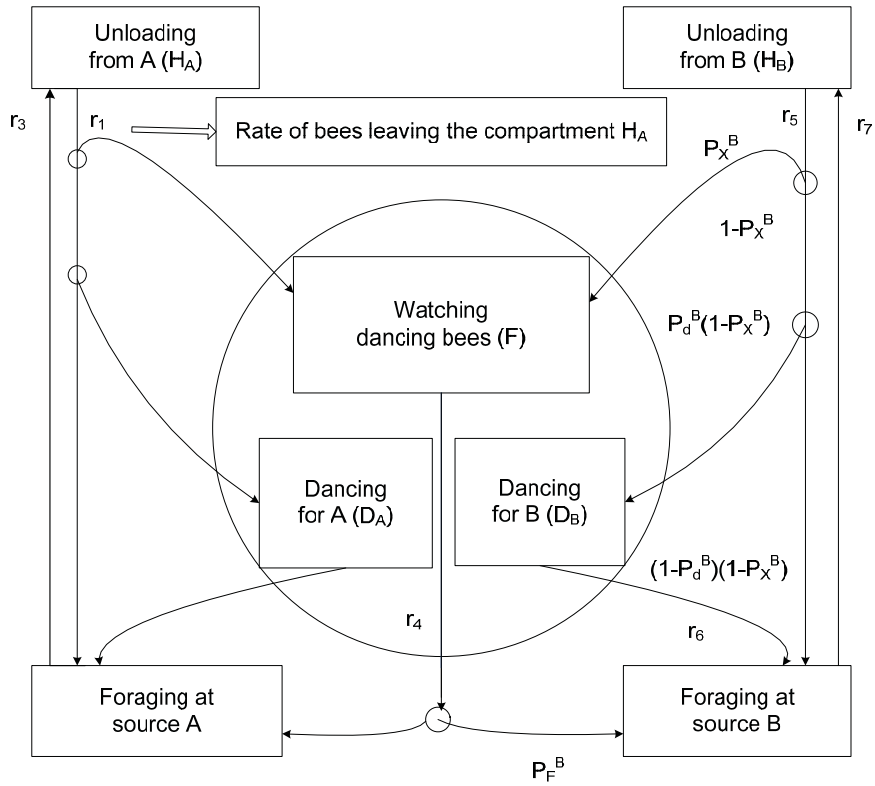


Obrázok 11-2. Správanie sa včely podľa (Vries, 1998).

11.2.4 Model získavania potravy z dvoch zdrojov

V práci (Camazine, 1991) uvedený je matematický model popisujúci dynamickú interakciu medzi včelami v procese nosenia potravy z dvoch zdrojov (zdroj A, zdroj B).

Modelovaná situácia je znázornená na obrázku 11-3, kde je vyznačených sedem rôznych častí. Každá časť predstavuje jeden z možných stavov, v ktorom sa včela môže nachádzať.



Obrázok 11-3. Matematický model získavania potravy z dvoch zdrojov (Camazine, 1991).

Predpokladajme že, `A` a `B` sú dva rôzne zdroje:

- časť H_A, časť H_B: vyloženie potravy zo zdrojov A a B,
- časť D_A, časť D_B: tancovanie za zdroje potravy A a B,
- časť F: pozorovanie tancujúcich včiel.

Na rozloženie celkového počtu včiel nosiacich potravu z dvoch zdrojov na sedem rôznych častí vplývajú dva faktory:

- itenzita, s ktorou sa včela pohybuje medzi jednotlivými časťami (r_{1-7}),
- pravdepodobnosť, s ktorou si včela vyberie jednu z dvoch ciest v rozhodovacích bodoch (na obrázku sú znázornené ako čierne body).

V nasledujúcom príklade si popíšeme správanie sa včely, ktorá priniesla potravu zo zdroja B.

Potom, ako včela vyloží v úli potravu prichádza k prvému rozhodovaciemu bodu, kedy sa rozhoduje či opustí zdroj $B\left(P_X^B\right)$, alebo bude pokračovať v zbieraní potravy zo zdroja $B\left(1-P_X^B\right)$.

Ak sa rozhodne pokračovať v nosení potravy zo zdroja $B\left(1-P_X^B\right)$, môže sa rozhodnúť, že pre tento zdroj zláka ďalšie včely $\left(P_d^B\left(1-P_X^B\right)\right)$.

Ak sa včela rozhodne nepokračovať v nosení potravy zo zdroja B, ide pozorovať tancujúce včely a rozhodne sa , ktorú z tancujúcich včiel bude nasledovať $\left(P_F^B\right)$.

Pravdepodobnosť P_F^B sa vypočíta:

$$P_F^B = \frac{D_B d_B}{D_B d_B + D_A d_A}$$

kde

- d_A a d_B sú časy tancovania včely za zdroj A alebo zdroj B; čas tancovania je priamoúmerný kvalite zdroja, teda, čím je zdroj kvalitnejší, tým je väčšia pravdepodobnosť, že pozorujúce včely odpozorujú vábny tanec a nechajú sa zlákať na zdroj potravy,
- D_A a D_B je počet včiel tancujúcich za zdroj A alebo zdroj B.

Jednotlivé pravdepodobnosti sú:

- P_X^A a P_X^B: pravdepodobnosť opustenia zdroja A alebo zdroja B; táto pravdepodobnosť sa vypočíta pri každom výlete včely,
- P_d^A a P_d^B: pravdepodobnosť tancovania za zdroj A alebo zdroj B,
- P_F^A a P_F^B: pravdepodobnosť nasledovania včely tancujúcej za zdroj A alebo zdroj B.

11.2.5 Multi-agentový odporúčajúci systém

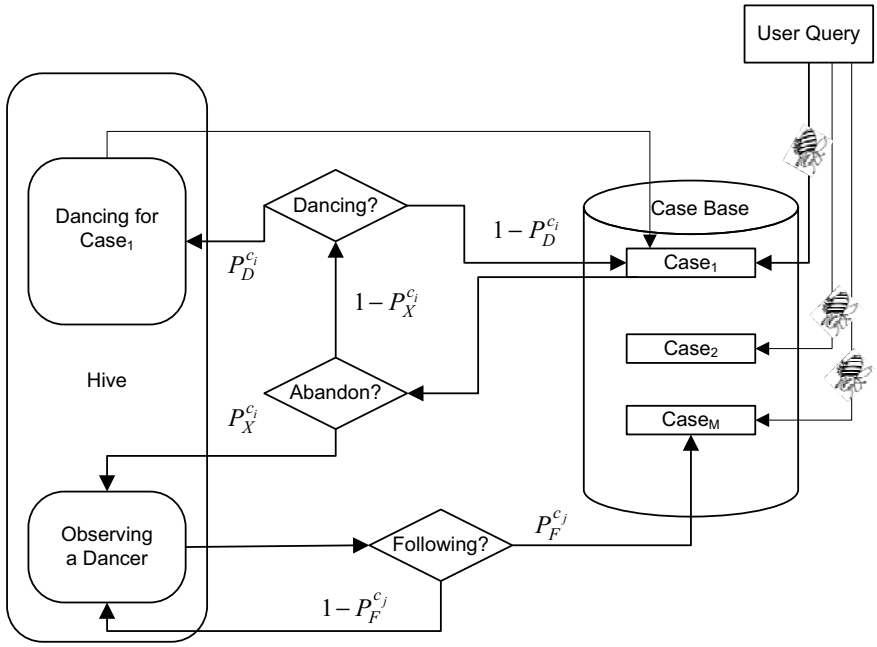
Na základe matematického modelu z (Camazine, 1991) bol v práci (Lorenzi, 2005A) navrhnutý a prezentovaný v (Lorenzi, 2005B), (Lorenzi, 2005C) prototyp multi-agentového odporúčajúceho systému, s využitím metafory správania sa spoločenstva sociálneho hmyzu - tancujúcich medonosných včiel. Vychádzali z prác (Schafer, 2001) a (Resnick, 1994), aplikujúc rozdielny prístup k riešeniu problému. Táto problematika bola spracovaná aj v (Lorenzi, 2004).

Model z (Camazine, 1991) rozšírili tak, aby bolo možné použiť viecero zdrojov potravy. Každý agent predstavuje včelu s nasledujúcimi vlastnosťami:

- $P_x^{c_i}$ pravdepodobnosť opustenia zdroja potravy c_i,
- $P_D^{c_i}$ pravdepodobnosť tancovania za zdroj c_i,
- $P_F^{c_j}$ pravdepodobnosť nasledovania iných včiel na zdroj c_j.

Pri dopyte od používateľa sa včely-agenty snažia nájsť v báze prípadov (zdrojov $c_{1\ldots m}$) najlepší zdroj potravy. Na obrázku 11-4 je znázornený model prezentovaný v (Lorenzi, 2005C).

ÚĽ je zložený z dvoch oddelení, tančiarne a pozorovateľne tančiarne. Báza prípadov reprezentuje prístupné zdroje nektáru $(1\ldots m)$. Kosoštvorce reprezentujú možnosti rozhodovania včiel.



Obrázok 11-4. Model multi-agentového systému.

V odporúčajúcom systéme je používateľova požiadavka porovnaná so všetkými prípadmi v báze prípadov. Prípad, ktorý najlepšie spĺňa požiadavku je ponúkaný používateľovi.

V tomto prípade podobnosť je použitá na to, aby sa včela rozhodla či bude pokračovať v nosení potravy zo zdroja, alebo prejde na iný. Včela navštívi bázu prípadov, porovná používateľov dopyt s navštíveným prípadom c_i na základe čoho vypočíta pravdepodobnosť $P_X^{c_i}$. Prípad s najmenšou vzdialenosťou od používateľovho dopytu je najpodobnejší prípad, teda prípad, ktorý reprezentuje najlepší zdroj. Prípad c_i je zdroj, ktorý včela momentálne navštevuje. Prípad c_j predstavuje zdroj, ktorý bol navštívený tancujúcou včelou, ktorú sleduje včela z pozorovateľne a rozhoduje sa či ju bude nasledovať, alebo nie.

Kosoštvorce predstavujú miesta, kde sa včely rozhodujú. Včela sa môže rozhodnúť, že bude pokračovať v nosení potravy zo zdroja (zostáva pri zdroji c_i), potom môže pre tento zdroj lákať iné včely (navštívi oddelenie tančiareň), alebo bude pokračovať v nosení potravy zo zdroja c_i bez pomoci iných včiel. Ak sa rozhodne že zdroj c_i opustí, navštívi oddelenie pozorovateľňa tančiarne odkiaľ pozoruje tancujúce včely a rozhoduje sa, ktorú z tancujúcich včiel bude nasledovať.

Výhoda odporúčajúceho systému založeného na opísanom princípe je, že systém vždy niečo odporučí. Niekedy je však lepšie neodporučiť nič, ako odporučiť veľmi nepresne, čo v tomto prípade predstavuje nevýhodu takéhoto systému.

Autori na rozšírenie a aktuálnosť bázy prípadov navrhujú použiť ďalšiu vrstvu agentov, ktorí by hľadali aktualizované informácie a zaraďovaním do bázy prípadov by ju rozširovali.

V experimentoch, ktoré autori s týmto systémom vykonali bolo dokázané, že čím je zdroj kvalitnejší (čím viac sa zhoduje s dopytom používateľa), tým viac včiel ho navštevu-

je. Autori nechceli poukázať iba na schopnosť zhlukovania sa včiel okolo najlepšieho zdroja, ale aj na dynamickú reakciu celého spoločenstva pri objavení lepšieho zdroja. Skúmali aj rýchlu reakciu systému pri zmene používateľovho dopytu.

11.3 Úpravy a vylepšenia modelu (Lorenzi, 2005C)

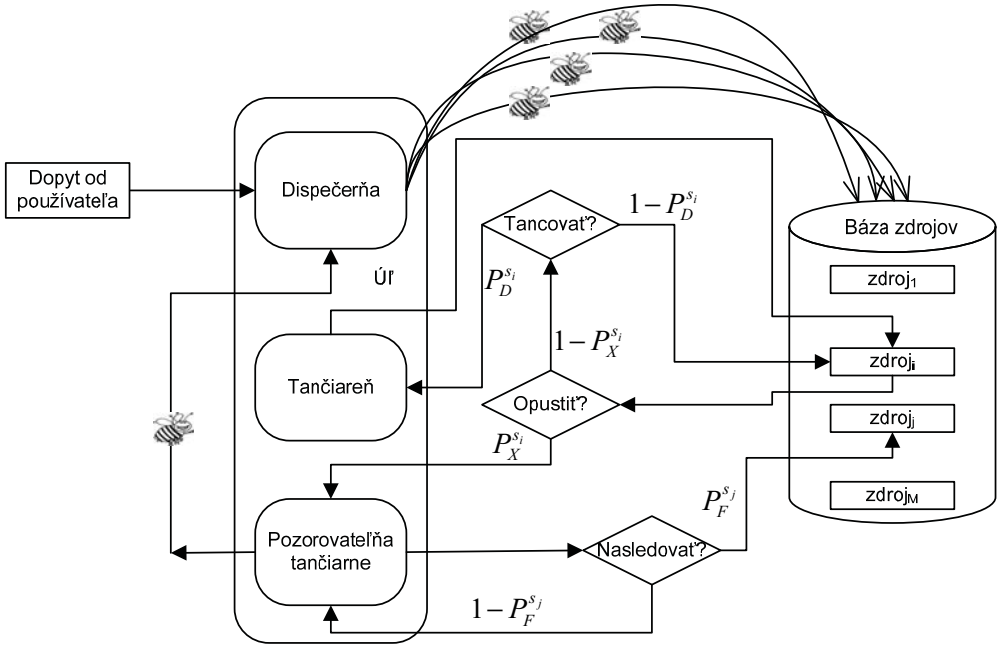
Práca (Návrat, 2006) bola inšpirovaná modelom predstaveným v (Lorenzi, 2005C). Autori zistili niektoré nedostatky modelu, z ktorého vychádzali a ich hlavným cieľom bolo ich odstránenie s prihliadnutím na to, aby sa nenarušil model včelieho spoločenstva.

K hlavným nedostatkom predtým popísaného systému bolo počiatočné priradenie jednej včely na jeden zdroj potravy. Pri takomto priradení vznikajú nasledujúce problémy:

- na dosiahnutie prijateľných výsledkov je potrebné použiť toľko včiel-agentov, koľko je zdrojov, takže pri veľkom počte zdrojov by boli nároky na systém neprijateľné,
- ak je včele na začiatku priradený zdroj, ktorého kvalita je vysoká (napríklad 95 %), napriek tomu existuje pravdepodobnosť (hoci iba 5 %), že bude zlákaná inou včelou. V tomto prípade ak včela tento zdroj opustí, neexistuje možnosť návratu k tomuto zdroju. Existujú iba dva spôsoby, ako sa včela dostane na zdroj a to na základe počiatočného priradenia, alebo v prípade zlákania inou včelou. Týmto sa môže veľmi kvalitný zdroj stať neprístupným pre celé spoločenstvo.

Hlavné požiadavky pri vytváraní modelu boli:

- zmenšenie potrebného počtu včiel,
- zmenšenie pravdepodobnosti trvalého opustenia kvalitného zdroja.



Obrázok 11-5. Model správania sa včelieho spoločenstva podľa (Návrat, 2006).

Oproti pôvodnému modelu bolo do časti úľ pridané oddelenie nazvané dispečerňa. Z tohto oddelenia včely vylietavajú náhodne do bázy zdrojov, kde hľadajú zdroje potravy. Pri používateľovom dopyte včela cez dispečerňu vyletí k náhodnému zdroju.

Ďalšou zmenou v modeli bolo usmernenie nerozhodných včiel, ktoré v predchádzajúcom modeli mohli neobmedzene dlho zostať v pozorovateľni tančiarne, čím sa zmenšoval počet výkonných včiel. V tomto modeli včely, ktoré sledujú v pozorovateľni tancujúce včely a do určitého času si nevyberú včelu, ktorú budú nasledovať, prechádzajú do dispečerne a odtiaľ na náhodný zdroj potravy.

V modeli bol pridaný aj informačný šum pri komunikácii medzi včelami (odovzdávanie informácie o zdroji tancovaním). Miera informačného šumu určuje ako nepresne tancujúca včela oznámi polohu zdroja pozorujúcej včele. Keďže včely neuvažujú priestorovú polohu zdrojov, informačný šum predstavuje pravdepodobnosť s akou tancujúca včela oznámi nepresnú adresu zdroja pozorujúcej včele. V prípade zvýšeného informačného šumu sa zvyšuje pravdepodobnosť, že včela bude poslaná na náhodný zdroj potravy.

Poslednou zmenou v modeli bolo pridanie chyby vyhodnocovania zdroja ERR. Chyba nadobúda hodnoty z intervalu $\langle 0,1 \rangle$. Táto chyba sa berie do úvahy pri výpočte podobnosti používateľovho dopytu a zdroja. Kvalita Q sa určuje zo vzťahu $Q = Q \pm Q * ERR$. V prípade, že v uvedenom vzťahu dostaneme hodnotu väčšiu ako 1, hodnota Q bude nastavená na 1.

Parametre modelu:

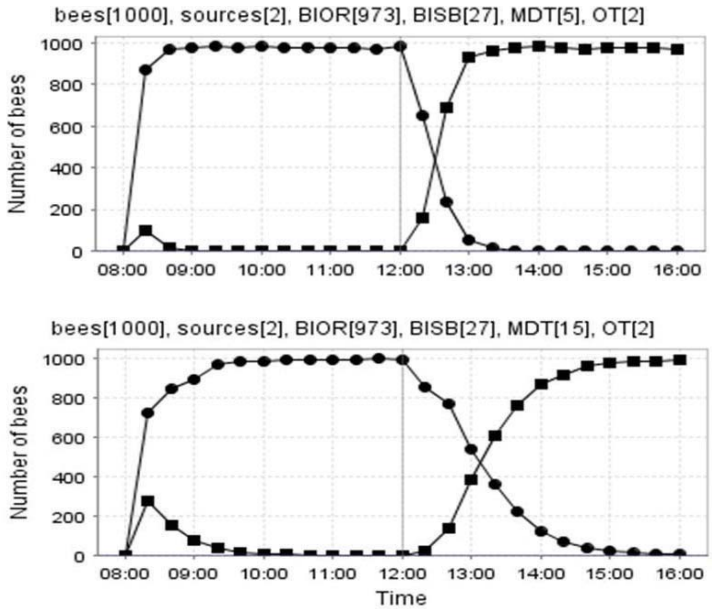
- celkový počet včiel v modeli $N(BIOR + BISB)$.
- počiatočná distribúcia včiel: $BIOR$ predstavuje počet včiel v pozorovateľni, $BISB$ počet včiel, ktoré vyletia do bázy zdrojov.
- maximálny čas tancovania za určitý zdroj potravy MDT, konkrétny čas tancovania je závislý na kvalite zdroja $(MDT * kvalita)$.
- maximálny čas, ktorý včela strávi v pozorovateľni OT, je nemenný, ale včela pred uplynutím tohto času môže byť zlákaná inou včelou.
- informačný šum $NOICE$, presnosť pri vymieňaní informácie medzi tancujúcou a pozorujúcou včelou.
- chyba vyhodnotenia kvality zdroja ERR.

11.4 Experimenty s modelom (Návrat, 2006)

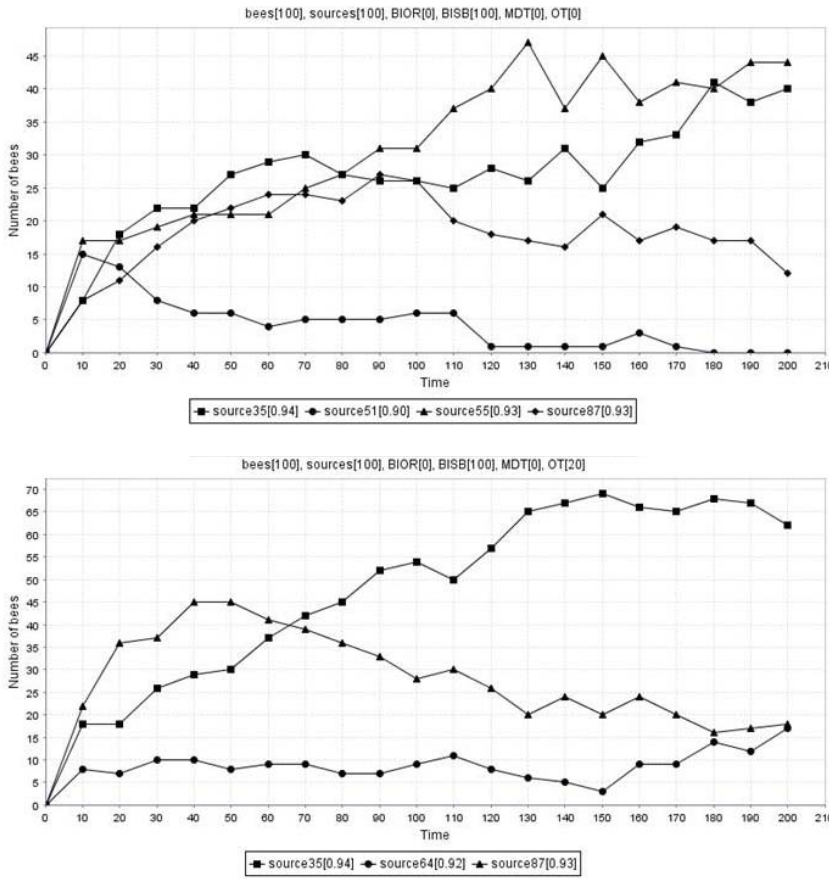
Boli vykonané viaceré experimenty podobné ako v (Selley, 1991) (výber z dvoch zdrojov potravy pri skutočnom včelom spoločenstve), s tým že sa súčasne sledovali nastavenia nových parametrov systému.

Pri zvyšovaní parametra MDT (maximálny čas tancovania) sa znižovala dynamika systému – včely dlhšie tancujú, čím sa zvyšuje pravdepodobnosť zlákania iných včiel, z ktorých následne viaceré tiež budú tancovať, teda rozhodovanie celého spoločenstva je pomalšie.

Pri znižovaní času pozorovania OT na obrázku 11-7 sa zvyšuje dynamika systému ako celku, pri hodnotách blízkych nule však nastávajú časté výkyvy správania sa spoločenstva, pretože včely často vylietavajú na náhodné zdroje.



Obrázok 11-6. Experiment s parametrom MDT.



Obrázok 11-7. Experiment s parametrom OT, x-ová os predstavuje čas a y-ová os počet včiel.

11.4.1 Experimenty s homogénnymi a nehomogénnymi včelami

Homogénne včely sú včely, ktoré vyhodnocujú kvalitu zdroja ako celok. Nehomogénne včely sú včely, ktoré vyhodnocujú kvalitu iba určitej zložky zdroja. Ak má zdroj viacero zložiek je potrebné aby existovalo viacero druhov včiel. Každý druh včiel bude vyhodnocovať kvalitu konkrétnej zložky zdroja. Za najkvalitnejší zdroj sa bude považovať ten, z ktorého na konci simulácie bude nosiť potravu najviac včiel bez ohľadu na to z akého sú druhu.

V experimente bol použitý trojzložkový zdroj, ktorý pozostával z dvoch slov a jedného čísla. Zdrojom bola hľadaná pracovná pozícia, ktorej prvou zložkou bolo mesto, druhou pracovná pozícia a treťou plat. Postupne bolo použitých 10, 100, 1000 a 2500 včiel a rôznych 1000 zdrojov.

Už pri použití 10 včiel bol vyhľadaný najlepší zdroj avšak pomalšie. Dobré výsledky sa dali dosiahnuť už použitím 10 krát menšieho počtu včiel ako bol počet zdrojov, vynikajúce výsledky boli dosiahnuté pri použití 4 krát menšieho počtu včiel ako zdrojov.

V prípade nehomogénnych včiel boli použité tie isté začiatočné podmienky ako v predchádzajúcom prípade. V tomto prípade však neboli dosiahnuté najlepšie výsledky. V prípade, že boli viaceré zdroje, ktoré obsahovali niektorú zo zložiek napríklad rovnaké mesto, včely sa rovnomerne rozdistribuovali medzi zdrojmi a postupne sa navzájom blokovali.

Výsledky experimentov boli popísané v práci (Návrat, 2007A).

11.4.2 Experiment s parametrami NOICE *a* ERR

Cieľom experimentu bolo ukázať vplyv zavedenia šumu do modelu. Boli vykonané experimenty, pri ktorých boli parametre modelu menené:

- *NOICE* nastavený na 15 % (tancujúca včela oznámi nepresne polohu zdroja pozorujúcej včele o 15 %) a *ERR* na `0`,
- *ERR* nastavený na 15 % (kvalita zdroja je ±15 % z reálnej kvality zdroja) a *NOICE* na `0`.

Zavedenie šumu sa môže javiť ako zámerné vnášanie nepresností do modelu. Pri experimentoch sa ukázalo, že zavedenie šumu môže mať priaznivý vplyv na rýchlosť objavovania nových zdrojov. Pri väčšom šume včely častejšie vylietavajú a objavujú nové zdroje.

11.4.3 Model včiel s pamäťou

Model včiel s pamäťou vychádza z predchádzajúceho modelu. Pamäť včely je veľmi jednoduchá a okrem naposledy navštíveného zdroja si pamätá pozíciu a kvalitu doteraz najkvalitnejšieho zdroja, ktorý dovtedy navštívila. Ak je kvalita nového zdroja väčšia ako kvalita zapamätaného zdroja, najkvalitnejším sa stáva nový zdroj. Oproti predchádzajúcemu modelu nastáva zmena správania sa včely v pozorovateľni tančiarne. V prípade, že včela v tančiarni propaguje kvalitnejší zdroj ako najkvalitnejší zdroj zapamätaný včelou, môže ju nasledovať, alebo sa môže rozhodnúť vrátiť sa k zapamätanému zdroju, prípadne prechádza do dispečerne.

V experimentoch sa overila predpokladaná skutočnosť, že takouto úpravou sa včely tak ako predtým sústreďovali okolo jedného zdroja potravy, avšak výsledky odporúčania

boli mierne kvalitnejšie a na kvalitné odporúčanie vystačoval menší počet včiel. Experimenty s včelami s pamäťou boli publikované v (Návrat, 2007A).

11.4.4 Použitie modelu včiel pri počítaní PageRanku

Algoritmus PageRank je založený na činnosti imaginárneho surfera, ktorý nahodne kliká na linky. Po každom kliknutí sa surfer rozhoduje či klikne znova. Pravdepodobnosť, že surfer bude pokračovať sa nazýva faktor útlmu. Viaceré štúdie sa zaoberali rôznymi faktormi útlmu, podľa (Page, 1998) tento faktor nadobúda vo všeobecnosti hodnotu približne 0,85.

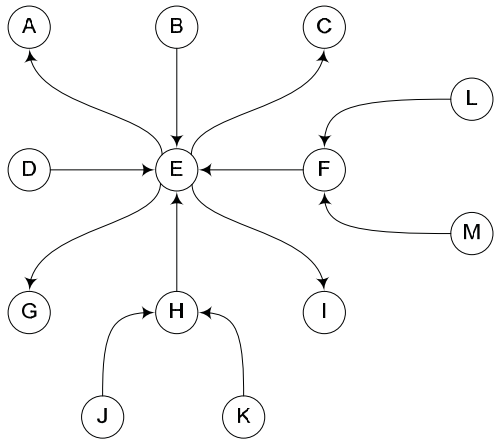
Všeobecný vzorec na výpočet PageRanku je podľa (Page, 1998) nasledujúci:

$$PR(p_i) = 1 - d + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

kde p_i je stránka, ktorej PageRank počítame, $M(p_i)$ je množina stránok, ktoré odkazujú na stránku p_i a $L(p_j)$ počet vychádzajúcich liniek zo stránky p_j.

Na výpočet dostatočne presnej hodnoty PageRanku každej stránky je potrebných niekoľko iterácií. Počas každej iterácie sa prepočíta PageRank každej stránky v kolekcii. Počet iterácií závisí od počtu stránok v kolekcii a od zložitosti prepojenia jednotlivých stránok.

Na obrázku 11-8 webové stránky predstavujú vrcholy grafu a orientované hrany grafu prepojenia medzi stránkami.



Obrázok 11-8. Jednoduchý graf prepojenia webových stránok.

Pri iteratívnom prístupe sa vypočíta PageRank stránky A, potom postupne PageRank stránok $B, C, ...M$.

Najideálnejšie by bolo vypočítať PageRank stránok, od ktorých sú ostatné stránky závislé.

Ak uvažujeme situáciu, ktorá je zobrazená na obrázku 1-8 a chceme vypočítať Page Rank stránky E. V tomto prípade potrebujeme poznať PageRank stránok B, D, H, F a následne Page Rank stránok J, K a L, M.

Uvažujme použitie včiel s mierne modifikovaným správaním. Ak včela priletí na zdroj E (vrchol E) zistí kvalitu zdroja, teda PageRank E, ale súčasne by zistila, že na presnejšie určenie PageRanku E je potrebné určiť PageRank vrcholov D, B, F, H .

Ak by bol zdroj E } kvalitný včela pri návrate do úľa by s veľkou pravdepodobnosťou išla do tančiarne, kde by však netancovala za zdroj E, ale za jeden zo zdrojov odkazujúcich na E (zdroje D, B, F, H). Výber jedného zo zdrojov je náhodný. Týmto sa včela v tančiarni snaží zlákať iné včely na zdroje, ktorých Page Rank je potrebný na presnejšie určenie Page Ranku vrcholu, ktorý navštívila. Po ukončení tancovania sa vráti včela na vrchol E a ak zlákala iné včely na odkazujúce vrcholy, môže presnejšie určiť PageRank vrcholu E . V prípade, že po aktualizácii je kvalita zdroja vysoká, existuje vysoká pravdepodobnosť, že včela bude propagovať v tančiarni jeden z odkazujúcich zdrojov. Celý cyklus sa teda opakuje a Page Rank vrcholu E sa postupne približuje k teoretickej hodnote.

Podobné pravidlá platia pre všetky zdroje.

Experimenty publikované v (Návrat, 2007B) ukázali, že použitie včiel na výpočet Page Ranku môže byť výhodné a v mnohých prípadoch rýchlejšie ako klasický iteračný prístup.

Na rozdiel od klasického prístupu metóda výpočtu PageRanku pomocou včiel neposkytuje vždy presné výsledky, čo je spôsobené povahou modelu – nie je zaručené, že včely navštívia všetky zdroje. Tento nedostatok môžeme odstrániť zvýšením počtu včiel.

Výhodou je možnosť kontinuálneho výpočtu PageRanku v prevádzke. Napríklad Google prepočítava vždy za určitý čas PageRank celého ním indexovaného webu. Pri použití modelu včiel by včely nepretržite lietali nad webovými stránkami (vrcholmi grafu) a aktualizovali by PageRank jednotlivých stránok (vrcholov grafu).

11.4.5 Vyhľadávanie webových stránok

V súčasnosti pracuje väčšina vyhľadávačov nad off-line databázou, ktorá obsahuje indexované stránky z časti Internetu. Obsah týchto stránok sa neustála aktualizuje a pribúdajú nové stránky.

Rozsah Internetu je obrovský a nie je možné prehľadať všetky stránky, preto je nutné on-line vyhľadávanie usmerniť iba na najsľubnejšie cesty. Každý používateľ má pri vyhľadávaní svoje preferencie. Tie môže vyjadriť vybratím skupiny stránok, ktoré sa venujú oblasti, ktorú preferuje. Z tejto skupiny stránok sa začína vyhľadávanie. Webová stránka teda predstavuje zdroj a dosiahnuteľnosť iného zdroja predstavuje hypertextový odkaz na inú stránku (zdroj).

Pôvodný model (Návrat, 2006) bol doplnený o správanie sa včely mimo úľa. Včela si po vyletení z úľa vyberie náhodný zdroj x, určí jeho kvalitu a s pravdepodobnosťou q_x sa vráti do úľa, alebo s pravdepodobnosťou $1-q_x$ si náhodne vyberie iný zdroj, dostupný zo zdroja x. Proces sa opakuje, pokiaľ včela nenarazí na zdroj z, z ktorého už žiadny iný zdroj nie je dostupný. V tomto prípade sa včela s pravdepodobnosťou q_z vráti do úľa s kvalitou zdroja q_z , inak s kvalitou 0. Návrat včiel s nulovou kvalitou je zavedený kvôli tomu, aby tento zdroj nebol zvýhodnený oproti ostatným, pretože z neho včely automaticky leteli do úľa a mohli tento zdroj propagovať viac oproti ostatným. Tým, že sa včely vrátia späť do úľa a propagujú kvalitné stránky, alebo ak také stránky nenájdu môžu pozorovať a nasledovať iné včely vyhľadávanie sa usmerňuje na kvalitnejšie zdroje. Za strán-

ku, ktorú úľ odporučil sa považuje stránka, ktorá bola dlhodobo najviac propagovaná v tančiarni.

Určovanie kvality zdroja

Kvalita zdroja je z intervalu $\langle 0,1 \rangle$ pričom 0 je najnižšia a 1 najvyššia kvalita.

Kvalita vzdialenosti

Vzdialenosť dvoch stránok definujme ako počet krokov, ktoré včela potrebuje, aby sa z jednej domény dostala na druhú. Čím je počet krokov menší, tým je väčšia pravdepodobnosť, že dané stránky sú obsahovo príbuzné. Na kvantifikáciu vzdialenosti boli navrhnuté vzťahy:

$$DIST_{\max} = 0 \Rightarrow q_{dist} = Q_{DIST}$$

$$d > DIST_{\max} \Rightarrow q_{dist} = 0$$

$$d \leq DIST_{\max} \Rightarrow q_{dist} = Q_{DIST} - d\frac{Q_{DIST}}{DIST_{MAX}}$$

kde $DIST_{\max}$ je maximálna vzdialenosť doletu včely, Q_{DIST} je maximálna hodnota čiastkovej kvality a d je vzdialenosť od pôvodnej stránky.

Kvalita počtu výskytov

Aby sa rozlíšil počet koľkokrát sa výraz na stránke objavil bola navrhnutá funkcia, ktorej hodnota na začiatku rýchlejšie rastie, potom sa rast spomalí a asymptoticky sa blíži k definovanej maximálnej hodnote kvality. Funkcia je tvaru:

$$q_{count} = \frac{-1}{2\left(n + \frac{1}{2Q_{COUNT}}\right)} + Q_{COUNT}$$

kde n je počet výskytov daného slova a Q_{COUNT} je maximálna hodnota definovaná pre túto čiastkovú kvalitu.

Kvalita výskytu v nadpise

Pri vyhľadávaní sa dá očakávať, že výraz vyskytujúci sa v nadpise je zaujímavejší ako ten istý výraz vyskytujúci sa v článku. V takomto prípade je dôležité sledovať ako hierarchicky vysoko sa slová v nadpisoch vyskytujú, nie celkový počet výskytov. HTML definuje nadpis celej stránky ako $\langle title \rangle$ a $\langle h_1 \rangle$ až $\langle h_6 \rangle$ pre nadpisy a podnadpisy stránok. Každej úrovni bolo priradené číslo $0(\langle title \rangle)$ až $6(\langle h_6 \rangle)$. Bola navrhnutá funkcia:

$$q_{header} = Q_{HEADER} - h * \frac{Q_{HEADER}}{HEADER_{MAX} + 1}$$

kde h je minimum zo všetkých hodnôt nadpisov, v ktorých sa hľadané slovo vyskytuje; Q_{HEADER} je maximálna hodnota tejto čiastkovej kvality; $HEADER_{MAX}$ je najväčšia hĺbka v hierarchii nadpisov zobraná pri výpočte do úvahy.

V práci (Návrat, 2007C) boli prezentované experimenty, ktoré on-line úspešne vyhľadávali stránky s podobnou tematikou ako vopred zvolené stránky.

11.4.6 Zhodnotenie analyzovaných metód

V tejto kapitole boli popísané viaceré modely včelieho spoločenstva. Vo všetkých prácach autori vykonali experiment popísaný v (Selley, 1991) na demonštráciu toho, že navrhovaný model určitým spôsobom kopíruje správanie sa včelieho spoločenstva.

Dôvody, ktoré viedli autorov k vytváraniu modelov boli rôzne. Pokiaľ v práci (Camazine, 1991) ide o snahu autorov matematicky vymodelovať a popísať experiment z (Selley, 1991), model navrhnutý v (Vries, 1998) je prepracovaný oveľa detailnejšie. Autor venuje veľkú pozornosť popisu včiel a prostredia v ktorom sa pohybujú. Tak, ako v prírode, aj v jeho modeli existuje viacero druhov včiel s rôznym vlastnosťami. Aj v tomto prípade ide o snahu vytvoriť čo najpresnejšie simuláciu biologického spoločenstva.

Ďalšie práce sú už orientované trochu inak.

Autorom už ne ide iba o simuláciu včelieho spoločenstva, ale o možnosť vhodným spôsobom použiť metaforu včelieho spoločenstva na praktické účely – odporúčanie, vyhľadávanie informácií.

Počiatočná, takto zameraná myšlienka sa objavila v prácach (Lorenzi, 2005A), (Lorenzi, 2005B), (Lorenzi, 2005C). Bol navrhnutý a prezentovaný prototyp multi-agentového odporúčajúceho systému.

Úpravy a vylepšenia tohto modelu boli prezentované v (Návrat, 2006). Autori veľmi dobre postrehli nedostatok modelu a to počiatočné priradenie jednej včely na jeden zdroj potravy. Navrhli oddelenie dispečerňa, počiatočnú distribúciu včiel, parametre *MDT*, *OT*, *NOICE*, *ERR*. Týmito úpravami dosiahli zmenšenie potrebného počtu včiel a zmenšenie pravdepodobnosti trvalého opustenia kvalitného zdroja.

V Experimentoch s takto upraveným modelom sa pokračovalo v prácach (Návrat, 2007A), (Návrat, 2007B). Bolo vykonaných viacero experimentov s parametrami modelu, vyhľadávanie relevantných informácií v databáze dokumentov, počítanie PageRanku webových stránok.

Rozšírenie modelu bolo prezentované v (Návrat, 2007C). Experimentálne bolo dokázané, že pomocou tohto modelu je možné on-line vyhľadávanie podobných webových stránok.

11.5 Otvorené problémy, možnosti optimalizácie modelu

Nasledujúca kapitola opisuje problémy, ktoré sa môžu vyskytnúť pri používaní modelu včiel na vyhľadávanie informácií na webe a náčrt možných riešení problémov.

11.5.1 Opis problému

Už v predchádzajúcej kapitole boli popísané rôzne modely, využívajúce správanie sa včiel. Z nášho pohľadu ide o vývoj týchto modelov od popisných (modely, ktorých cieľom je zachytiť a popísať správanie sa včiel) na modely slúžiace k určitému účelu (využiť charakter správania sa včiel na určitý účel).

Pri úspešnom použití uvedených modelov na vyhľadávanie na webe je potrebné venovať sa dvom základným problémom:

- *Optimalizácii vnútornej časti modelu.* Pod pojmom vnútorná časť budeme rozumieť sledovanie všetkých aktivít vykonávaných včelami vo vnútri úľa.
- *Optimalizácii vonkajšej časti modelu.* Pod pojmom vonkajšia časť modelu budeme rozumieť sledovanie všetkých aktivít, ktoré vykonávajú včely mimo úľa.

Opis vnútornej časti modelu

Prvý parameter, ktorý budeme v práci sledovať je celkový počet včiel v modeli $N(BIOR + BISB)$, počiatočná distribúcia včiel: $BIOR$ predstavuje počet včiel v pozorovateľni, $BISB$ počet včiel, ktoré vyletia do bázy zdrojov.

Jeden z hlavných cieľov v predchádzajúcom modeli bolo znižovanie počtu včiel. Na prvý pohľad je znižovanie počtu včiel výhodné, avšak nadmerné zníženie počtu včiel vedie k predlžovaniu vyhľadávania a možnosti, že najlepší zdroj potravy nebude vyhľadaný vôbec. Ideálny model by mal byť schopný reagovať na vonkajšie zmeny (množstvo a kvalita zdrojov) a dynamicky regulovať počet potrebných včiel.

V predchádzajúcej práci nebol venovaný veľký priestor skúmaniu pomeru počiatočného rozdelenia včiel. Hoci toto rozdelenie nemá na výkonnosť modelu taký vplyv ako určenie počtu včiel, bolo by vhodné určiť počiatočné optimálne rozmiestnenie. Nadmerný počet $BISB$ včiel zaistí zistenie kvality viacerých zdrojov, nedostatočný počet $BIOR$ včiel spôsobí pomalú konvergenciu k najlepšiemu zdroju. Naopak nadmerný počet $BIOR$ včiel kombinovaný s vysokou hodnotou parametra OT (maximálny čas, ktorý môže včela stráviť v pozorovateľni) spôsobí nárast množstva včiel, ktoré sú nevyužité.

V nasledujúcej práci bude potrebné overiť nutnosť použitia parametra MDT (maximálny čas tancovania). Pri optimálnom nastavení času tancovania, ktorý je priamoúmerný kvalite propagovaného zdroja sa môže javiť tento parameter ako nepotrebný, alebo nepresný, pretože môže skresľovať kvalitu propagovaného zdroja.

Použitie parametrov $NOICE$ (nepresnosť pri vymieňaní informácie medzi tancujúcou a pozorujúcou včelou) a ERR (chyba vyhodnotenia kvality zdroja) je vhodné nielen kvôli podobnosti modelu so skutočným správaním sa včiel, ale aj pre možnosť zvýšenia dynamiky systému (v prípade zlého vyhodnotenia je viacero včiel posielaných na náhodné zdroje, čím sa zvyšuje pravdepodobnosť nachádzania nových zdrojov).

Opis vonkajšej časti modelu

V tejto časti budeme venovať pozornosť zisťovaniu kvality zdrojov a návrhu, akým spôsobom nasmerovať včely, aby vyhľadávanie stránok na webe bolo organizované a dosahovalo čo najlepšie výsledky.

Zisťovanie kvality zdroja. Zisťovanie kvality zdroja je dôležitá časť modelu, od ktorej vo veľkej miere závisí relevantnosť vyhľadávania. V doteraz známych prácach autori prezentovali spôsob, podľa ktorého jedna včela zisťuje kvalitu zdroja (Kováčik, 2006).

Autor prezentoval myšlienku homogénnych a nehomogénnych včiel. Homogénna včela bola včela, ktorá vyhodnocovala kvalitu zdroja ako celku, nehomogénna vyhodnocovala čiastkovú kvalitu zdroja. Pri nehomogénnych včelách existovalo toľko typov včiel, koľko bolo sledovaných zložiek zdroja a ako najkvalitnejší bol vyhodnotený zdroj, ktorý navštívilo najviac včiel bez ohľadu na typ včely. Experimentálne sa dokázalo, že pri ta-

komto vyhodnocovaní zdrojov sa dosahovali horšie výsledky ako pri použití homogénnych včiel.

V práci (Návrat, 2007C) je popísaný spôsob určovania kvality zdroja. Hoci je kvalita zdroja vyhodnocovaná podľa troch parametrov, všetky tieto parametre vyhodnocuje jedna včela. Ako doplňujúci spôsob zisťovania kvality zdroja môže byť použitá metóda z (Návrat, 2008).

Pohyb včiel medzi zdrojmi. V predchádzajúcich častiach boli uvedené dva podobné spôsoby, akými sa včely pohybujú medzi zdrojmi. V nasledujúcej práci budeme vychádzať z týchto prác a keďže v prípade prepojenia webových stránok ide o obrovský priestor, navrhnúť možnosti vyhľadávania pomocou viacerých úľov a ich centrálne riadenie pomocou hlavného úľa.

11.5.2 Matematický model

Modelovanie je jedna z metód poznávania. Rôznymi prostriedkami – slovným opisom, graficky, pomocou matematickej symboliky, fyzikálnymi a technicky realizovanými modelmi človek opisoval javy, ktoré vo svojom okolí pozoroval.

Dôležitým aspektom pri vývoji modelu je, aby sa s modelom dalo experimentovať. Proces experimentovania s modelom nazývame simulácia. Simulačné experimenty nám umožňujú hľadanie alternatív a vhodné nastavenie parametrov modelu. Pokiaľ nepoznáme realizovateľný algoritmus nájdenia optima, simulácia neumožňuje priame určenie optimálneho riešenia – optimálnych parametrov. Tie musíme hľadať metódami optimalizácie, ako sú lineárne alebo dynamické programovanie. Pri definovaní akéhokoľvek systému rozlišujeme v ňom dva druhy množín:

1. množinu objektov prvkov, častí,
2. množinu relácií, zobrazení, funkcií.

Systém S je dvojica $S = (A, R)$, kde A je množina prvkov systému a R je množina relácií medzi nimi.

Často sa systém znázorňuje v tvare orientovaného grafu, v ktorom vrcholy grafu znázorňujú prvky systému a hrany grafu znázorňujú interakcie v systéme. Dynamický systém (Neuschl, 1998) je definovaný pomocou symbolov:

$$S = (T, U, Y, V, Z, X, f, g)$$

kde:

T – množina časových okamihov t,
U – množina hodnôt vstupných veličín u,
Y – množina vstupných veličín (funkcií $y(t)$),
V – množina stavových veličín,
Z – množina hodnôt výstupných veličín,
X – množina výstupných veličín (funkcií $x(t)$),
F – prechodová funkcia,
g – výstupné zobrazenie.

Veľké množstvo modelov vychádza zo systémov, ktorých hlavnými zložkami sú: vstupný prúd, rad, obslužný kanál, výstupný prúd. Takéto systémy sa nazývajú systémy hromadnej obsluhy SHO.

Vstupný prúd je postupnosť príchodov požiadaviek, ktoré nasledujú za sebou v nejakých časových okamihoch. Časové intervaly medzi po sebe nasledujúcimi príchodmi môžu byť pravidelné, alebo náhodné. V prípade náhodných príchodov potrebujeme pravdepodobnostné charakteristiky vstupného prúdu.

Pravidlo, podľa ktorého vyberáme požiadavky z radu na obsluhu sa nazýva disciplína čakania (front, inverzný front, prioritný front).

Obsluha v SHO môže byť poskytovaná jedným, alebo viacerými obslužnými kanálmi. Čas trvania obsluhy môže byť pevný, alebo náhodný.

Výstupný prúd je tvorený postupnosťou okamihov odchodov požiadaviek zo systému. Výstupný prúd sledujeme hlavne v prípade, že je vstupným prúdom do ďalšieho SHO. Ak použijeme na modelovanie SHO, vo všeobecnosti nás zaujíma efektívnosť činnosti systému.

Vstupný prúd

Počet príchodov požiadaviek do systému od začiatku sledovania po nejaký pevný čas t je náhodná veličina (Andel, 2003), ktorú označíme $N(t)$. Pri náhodných veličinách je dôležité sledovať ich pravdepodobnostné charakteristiky, teda rozdelenie, distribučnú funkciu, hustotu rozdelenia.

Homogénny proces

Príchod požiadaviek do systému je pravidelný. Pravdepodobnosť toho, že v intervale dĺžky t príde k požiadaviek nezávisí od začiatku intervalu, ale iba od jeho dĺžky

$$P(N(s+t)-N(s)=k)=P(N(t)=k)$$

pre všetky $s \geq 0$ a $t > 0$. Hodnota veličiny $N(t)$ je teda počet požiadaviek, ktoré prišli do systému v intervale $(0, t>$, pričom $N(0)=0$. Vstupný prúd popisujeme náhodnými veličinami $N(t)$ pre $t \geq 0$, preto tento proces nazývame náhodný (stochastický).

Nezávislé prírastky príchodov

Uvažujme intervaly $(a_1, b_1>, (a_2, b_2>, ..., (a_n, b_n>$. Náhodný proces $\{N(t)\}_{t\geq 0}$ nazývame procesom s nezávislými prírastkami $(N(b_1)-N(a_1), N(b_2)-N(a_2), ..., N(b_n)-N(a_n))$, ak intervaly $(a_1, b_1>, (a_2, b_2>, ..., (a_n, b_n>$ sú disjunktné.

Ordinárny proces

Ordinárnosť znamená, že vo veľmi krátkom časovom intervale príde viac ako jeden zákazník so zanedbateľnou pravdepodobnosťou, rádovo menšou ako je dĺžka tohto intervalu. Je teda nepravdepodobné, že súčasne prídu viacerí zákazníci.

Môžeme dokázať, že homogénny ordinárny proces s nezávislými prírastkami je nevyhnutne Poissonovým procesom.

Kendallova klasifikácia

Úlohy teórie hromadnej obsluhy vykazujú viacero podobností. Pre ich vzájomné rozlíšenie bola Kendellom zavedená symbolika popísaná (Unčovský, 1980). Typ úlohy sa označuje symbolmi:

$$X / Y / n / m$$

kde

- X – označuje typ náhodného procesu príchodu požiadaviek,
- Y – zákon rozloženia dĺžky doby obsluhy,
- n – počet obslužných miest – kanálov,
- m – maximálny počet zákazníkov v systéme (nie vždy uvedené).

V praxi často používame za symboly X, Y označenie M - Poissonov proces, alebo D – deterministický príchod, alebo obsluha. Za n a m dosadzujeme prirodzené čísla, v prípade m aj ∞ (nekonečný front).

Stavy systému, intenzity prechodov

Nakoľko problematika *SHO* je pomerne obsiahla budeme sa zaoberať a detailne analyzovať iba stavy konkrétneho modelu. Popísanie stavov a intenzity prechodov medzi jednotlivými stavmi bude matematicky zapísané. V tejto fáze riešenia uvedieme hrubý návrh modelu.

Vnútorná časť modelu (úľ) bude pozostávať zo štyroch častí (stavov, v ktorých sa každá včela môže nachádzať):

- sklad – včely prichádzajú náhodne do úľa s informáciou o polohe a kvalite zdroja,
- tančiareň – miesto, kde včely propagujú posledný nájdený zdroj a tancom lákajú na tento zdroj iné včely,
- pozorovateľňa tančiarne – miesto, kde včely pozorujú tancujúce včely, pri inicializácii modelu obsahuje informáciu o počte pozorujúcich včiel,
- dispečerňa – miesto, kde prichádzajú včely, ktoré neboli zlákané na zdroje propagované v tančiarni, pri inicializácii obsahuje informáciu o počte včiel, ktoré odchádzajú na náhodné zdroje.

Ako je vidieť na obrázku 11-3 medzi týmito stavmi existujú rozhodovacie body, v ktorých sa včely na základe kvality navštíveného zdroja rozhodujú. Intenzita prechodov medzi jednotlivými stavmi bude závislá od kvality navštívených zdrojov. Stavy si môžeme predstaviť ako vrcholy a interakcie medzi nimi ako hrany orientovaného grafu.

11.6 Optimálne riadenie

11.6.1 Stochastické dynamické programovanie

V reálnom živote sa často vyskytujú úlohy, v ktorých na chod systému pôsobia náhodné vonkajšie vplyvy (v našom prípade kvalita zdrojov). Hodnoty týchto vplyvov vopred nepoznáme, napriek tomu potrebujeme optimálny výkon celého systému s minimálnymi stratami. Teória, ktorá sa zaoberá problematikou riešenia týchto systémov je uvedená v (Brunovsky, 1980).

Predstavme si v našom prípade situáciu: Včela prichádza do úľa so zdrojom potravy, ktorý má určitú kvalitu. Pri vstupe do úľa máme včelu nazvime kontrolór, ktorá kontroluje kvalitu prineseného zdroja a určuje, ktorá včela vstúpi do úľa a ktorá sa vráti na ľubovoľný zdroj (v reálnom úli takéto včely existujú). Naším cieľom bude optimalizovať počet včiel vzhľadom na kvalitu prinášaných zdrojov a možnosti ďalšieho spracovania zdrojov. Stratou v tomto prípade bude nadmerný počet včiel, ktoré sú posielané späť na zdroje, alebo nadmerný počet včiel vchádzajúcich do úľa, ktorý ich kapacitne nestíha obsluhovať a predlžuje sa čas obsluhy.

Úloha nájsť riadenie tohto systému tak, aby straty boli minimálne nemá v tomto prípade zmysel, pretože hodnota počtu využitých včiel, (teda aj straty) závisia od kvality zdrojov, čo je náhodná premenná. Správanie systému je popísané rovnicami:

$$x_{i+1} = F_i(x_i, u_i, z_i)$$

pre $i = 0,...,k-1$, kde x_i charakterizuje stav systému, u_i riadenie a z_i navzájom nezávislé náhodné premenné. Zadefinujme účelovú funkciu, ktorá predstavuje celkovú stratu za k jednotiek času:

$$J(U,Z) = \sum_{i=1}^{k-1} f_i^0(x_i, u_i, z_i)$$

Na prvý pohľad sa javí najprirodzenejšie dodefinovať optimálne riadenie tak, aby minimalizovalo strednú hodnotu účelovej funkcie vzhľadom na náhodnú premennú (zdroje).

V prípade náhodného systému je situácia iná. Riadenie definujeme v tvare spätnej väzby $V = (v_0,..., v_{k-1})$ (stratégia) a účelovú funkciu vyjadríme vzťahom:

$$J(V,Z) = \sum_{i=1}^{k-1} f_i^0(x_i, v_i(x_i), z_i)$$

Má teda zmysel definovať optimálnu stratégiu ako stratégiu, ktorá minimalizuje strednú hodnotu účelovej funkcie $J(V,Z)$ pri daných ohraničeniach.

V prípade prepojenia viacerých úľov by sa informácie o riadení úľa prenášali do centrálneho úľa, kde by sa na základe vyhodnotenia efektívnosti jednotlivých úľov rozhodlo o premiestnení úľov, v prípade celkove nízkeho alebo vysokého nachádzania kvalitných zdrojov menil spôsob vyhodnocovania zdrojov (menili by sa hranice kvality zdrojov).

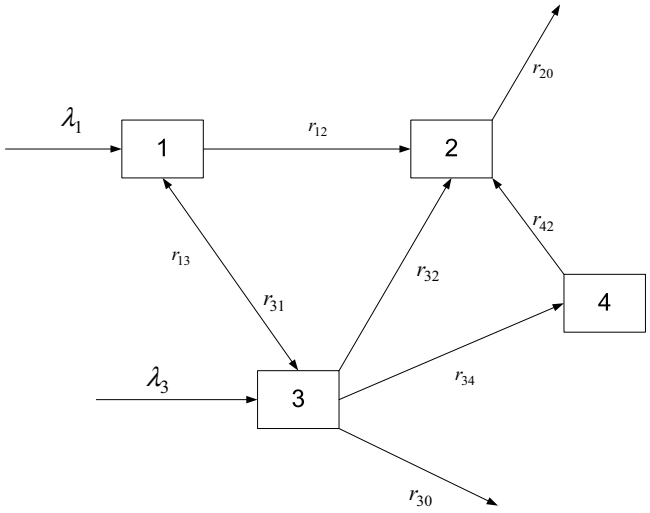
11.7 Sieťové spojenie modelov

11.7.1 Obslužné siete

Siete systémov hromadnej obsluhy sú tvorené systémami hromadnej obsluhy, pričom požiadavka, ktorej obsluha skončila v jednom systéme môže vyžadovať ďalšiu obsluhu.

Ak požiadavka vyžaduje obsluhu v systéme v ktorom už bola, hovoríme o systéme so slučkou. Siete, do ktorých môžu požiadavky prichádzať z ich okolia a odchádzať z nich nazývame otvorenými. V opačnom prípade sú to zatvorené siete.

Na obrázku 11-9 je znázornená sieť štyroch počítačov. Počítače 1 a 3 prijímajú úlohy, počítače 2 a 3 odosielajú výsledky spracovania úloh.



Obrázok 11-9. Sieť štyroch počítačov.

Obojstranná komunikácia je iba medzi počítačmi 1 a 3. Priemerná dĺžka medzier medzi príchodmi úloh do počítača 1 je $\frac{1}{\lambda_1}$, do počítača 3 je $\frac{1}{\lambda_3}$. Priemerná doba spracovania úlohy na počítači j je $\frac{1}{\mu_j}$. Pre pravdepodobnosti prechodu úloh medzi počítačmi platia vzťahy:

$$t_{12} + r_{13} = 1$$

$$r_{20} = 1$$

$$r_{30} + r_{31} + r_{32} + r_{34} = 1$$

$$r_{42} = 1$$

Hoci tento príklad je iba ilustračný čiastočne ukazuje možnosť prepojenia počítačov pri spracovaní veľkého množstva požiadaviek.

V našom prípade by malo ísť o sieť, v ktorej by jednotlivé počítače na nižšej úrovni pracovali nezávisle – vyhodnocovali kvalitu zdrojov získaných z webových stránok a zisťovali najlepší zdroj podľa optimalizovaného modelu (Návrat, 2006). Každý tento počítač by predstavoval úľ a následne by najlepšie zdroje z každého z nich tancom propagovali v centrálnom počítači.

Ak si centrálny počítač predstavíme tiež ako úľ, v ktorom okrem ukladania najlepších zdrojov môžu iné počítače zisťovať kvalitu a polohu iných kvalitných zdrojov, môže centrálny počítač ovplyvňovať hĺbku a šírku prehľadávania a okolie najlepších zdrojov bude detailnejšie prehľadané v kratšom čase.

11.8 Zhodnotenie

V tejto práci vychádzame z poznatkov získaných z analýz o problematike vývoja modelov správania sa sociálneho hmyzu. Sledujeme spôsoby a možnosti určitého napodobenia

správania sa tohto hmyzu za účelom získavania informácií z webových stránok. Vyhľadávanie informácií z webových stránok predstavuje nový pohľad na možnosti a spôsoby vyhľadávania. Zaujala nás jednoduchosť celého princípu vyhľadávania a aj napriek tomu boli experimentálne dosiahnuté úspešné výsledky vyhľadávania.

K zlepšeniu výsledkov vyhľadávania bude potrebné v budúcnosti realizovať nasledujúce úlohy:

- Matematický popis modelu (Návrat, 2006).

  Na optimalizáciu modelu je nevyhnutné poznať a matematicky popísať používaný model. Východiskovým modelom bude model z práce (Návrat, 2006). Na popis jednotlivých stavov modelu (objektov) a relácií medzi nimi nám poslúži teória SHO.

- Optimalizácia modelu, návrh na zmenu v modeli (Návrat, 2006).

  V predchádzajúcich modeloch bola zaujímavá ich jednoduchosť. Navrhnuté myšlienky sa môžu na prvý pohľad javiť ako komplikovanie predchádzajúceho modelu, na druhej strane môžu zvýšiť dynamiku a výkonnosť systému. Je dôležité uvedomiť si rozdiel medzi modelom skutočných včiel a nami navrhnutým modelom. V prírode včela po nájdení výhodného zdroja prichádza do úľa pre pomoc, aby spoločne s inými včelami odniesli množstvo potravy z jedného zdroja do úľa. Pri kvalitnom zdroji je potrebné veľké množstvo včiel, teda zdroj sa považuje za kvalitný, ak ho navštevuje veľa včiel. V našom modeli informáciu o kvalite potravy je schopná preniesť jedna včela. V prípade kvalitného zdroja sa však dá predpokladať, že aj okolie zdroja bude kvalitné a včela v tančiarni pri vábnom tanci (dĺžka tancovania je priamoúmerná kvalite propagovaného zdroja) zvábi viac včiel, ktoré budú skúmať okolie kvalitného zdroja.

- Návrh použitia a prepojenia viacerých modelov (úľov) s cieľom kvalitnejšieho vyhľadávania informácií na webe.

  Efektívne vyhľadávanie v obrovskom priestore ako je webový priestor vyžaduje použitie systému, ktorý spracuje informácie na viacerých počítačoch a centrálne riadi vyhľadávanie.

Použitá literatúra

[1] Andel, J.: Statistické metódy. *matfyzpress*, Praha, 1998.

[2] Baldi, P., Frasconi, P., Smyth, P.: *Modeling the Internet and the Web Probabilistic Methods and Algorithms*. John Wiley Sons Ltd. England, 2003.

[3] Bergman, M. K.: The Deep Web: Surfacing Hidden Value. *Journal of Electronic Publishing*, 2001.

[4] Bing Liu: *Web Data Mining: Exploring Hyperlinks,Contents and Usage Data*. Springer-Verlag Berlin Heidelberg 2007, ISBN-10 3-540-37881-2, 2007.

[5] Bonabeau, E., Thraulaz, G., Dorigo, M.: *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[6] Brunovský, P.: *Matematická teória optimálneho riadenia*. ALFA, Bratislava, 1980.

[7] Camazine, S., Sneyd, J.: A model of collective nectar source selection by honey bees: Self-organization through simple rules. *Journal of Theoretical Biology*, 149(4):547-571, 1991.

[8] Camazine, S., Deneubourg, J. L., Franks, N. R., et al.: *Self-organization in Biological Systems.* Princeton University Press, 2003.

[9] Chakrabarti, S.: *Mining the Web, Discovering Knowledge from Hypertext Data.* Morgan Kaufmann publishers, 2003.

[10] Dorigo, M., Maniezzo, V., Colorni, A.: Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1):29-41, 1996.

[11] Duda, R. O., Hart, P. E., Stork, D. G.: *Pattern Classification*. John Wiley\&Sons Ltd. Canada, 2001.

[12] Gadagkar, R.: *The Honeybee Dance-Language Controversy: Robot Bee Comes to the Rescue*. Resonance, January 1996.

[13] Gordon, D.: The organization of work in social insect colonies. *Nature*, 380:121-124, 1996.

[14] Kováčik, M.: Odporúčanie webových stránok so zaujímavou informáciou pomocou včiel. *Diplomová práca*, STU FIIT v Bratislave, december 2006.

[15] Lorenzi, F., Ricci, F.: Case-based reasoning and recommender systems. *Technical report*, IRST, 2004.

[16] Lorenzi, F., Sherer dos Santos, D., Bazzan, A.L.C.: Case-based Recommender Systems: a unifying view. In *LNCS/LNAI State-of-the-Art Survey book on Intelligent Techniques in Web Personalization*, 2005.

[17] Lorenzi, F., Sherer dos Santos, D., Bazzan, A.L.C.: Case-based recommender systems inspired by social insects. In *XXV Congresso da Sociedade Brasileira de Computacao*, 22-29 júl, 2005.

[18] Lorenzi, F., Sherer dos Santos, D., Bazzan, A.L.C.: Negotiation for task allocation among agents in case-based reccommender systems: a swarm-intelligence approach. In *IJCAI-05 Workshop on Multi-Agent Information Retrieval and Recomander Systems*. Edinburg, pp. 23-27, 2005.

[19] Návrat, P., Kováčik, M.: Web Search Engine as a Bee Hive. In: *2006 IEEE/WIC/ACM International Conference on Web Intelligence*, Hong Kong, Los Alamitos, IEEE Computer Society, pp. 694-701, 2006.

[20] Návrat, P., Kováčik, M., Bou Ezzeddine, A., Rozinajová, V.: Vyhľadávanie informácií pomocou včiel. In *Znalosti 2007*, Ostrava, ISBN 978-80-248-1279-3, pp. 63-74, 2007.

[21] Návrat, P., Kováčik, M., Bou Ezzeddine, A., Rozinajová, V.: Metafora včelieho úľa - model vyhľadávania a odporúčania informácií. *Kognícia a umelý život 2007*, Smolenice, pp. 249-258, 2007.

[22] Návrat, P., Jastrzembská, L., Jelínek, T., Bou Ezzeddine, A., Rozinajová, V.: Exploring Social Behaviour of Honey Bees Searching on the Web. In *Y. Li, V.V. Raghavan, A. Broder, H. Ho: 2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology (Workshops)*, Silicon Valley, USA, 2-5 November 2007, Los Alamitos USA : IEEE Computer Society, ISBN 0-7695-3028-1, pp. 21-25, 2007.

[23] Návrat, P., Taraba, T., Bou Ezzeddine, A., Chudá, D.: (2008). Context Search Enhanced by Readability Index. In *IFIP World Computer Congress*, Milano, Italy, 7-10 September, 2008.

[24] Neuschl, Š., et al.: *Modelovanie a simulácia*. SNTL, Nakladatelství technické litaratury, Praha, 1998.

[25] Page, L., Brin, S.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30 (1-7):107-117, 1998.

[26] Resnick, P., et al.: An open architecture for collaboretive filtering of netnews. In *Proceedings ACM Conference on Computer-Supported Cooperative Work*, pp. 175-186, 1994.

[27] Schafer, J. B., Konstan, J. A., Riedl, J.: E-commerce recommendation applications. *Data Mining and Knowledge Discovery*, 5(1/2):115-153, 2001.

[28] Selley, D., Camazine, S., Sneyd, J.: Collective decision-marking in heney bees: How colonies choose nectar sources. *Behavioral Ecology and Sociobiology*, 28:277-290, 1991.

[29] Sherman, CH., Price, G.: *The Invisible Web: Uncovering Information Sources Search Engines Can´t See*. ISBN 0-910965-51-X-2001, 2001.

[30] Unčovský, L.: *Stochastické modely operačnej analýzy*. Alfa, Bratislava, 1980.

[31] Vreis, H., Biesmeijer, J. C.: Modelling collective foraging by means of individual behaviour rules in honey-bees. *Behav. Ecol. Sociobiol. 44*, pp. 109-124, 1998.

[32] Webb, A.: *Statistical Pattern Recognition*. John Wiley Sons Ltd. England, 2002.

[33] Witten, I. H., Eibe, F.: *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier ISBN 0-12-088407-0, Canada, 2005.

EDÍCIA VÝSKUMNÝCH TEXTOV
INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Mária Bieliková, Pavol Návrat (editori)

Štúdie vybraných tém programových a informačných systémov (4)

Pokročilé metódy navrhovania softvéru
Pokročilé metódy získavania, vyhľadávania, reprezentácie
a prezentácie informácií

1. vydanie
Tlač Nakladateľstvo STU v Bratislave

346 strán

2009

ISBN 978-80-227-3139-3