uporabna INFORMATIKA

2000
ŠTEVILKA 4
OKT/NOV/DEC
LETNIK VIII
ISSN 1318-1882

tematska številka:
Objektna tehnologija

DONATORJI

ASTER

Nade Ovčakove 1, 1000 Ljubljana
Tel.: +386 01 589 42 00

gambit trade
računalništvo in informatika d.o.o.

Savska c. 3a, 1000 Ljubljana
Tel.: 01 437 63 33

Leskoškova 6, 1000 Ljubljana
Tel.: 01 585 58 00
Fax: 01 585 59 00

MAOP®
Vaš partner v informatiki
MAOP RAČUNALNIŠKI INŽENIRING D.O.O., WWW.MAOP.SI

MARAND
Napredna računalniška hiša

Cesta v Mestni log 55, 1000 Ljubljana
Tel.: 01 283 33 77

www.menea.si
menea
d.o.o., internet trgovski center

Microsoft®

PERFTECH
Perftech d.o.o., Pot na Lisice 4, 4260 Bled
Tel : 04 579 00 00 • Fax : 04 579 02 00

World Trade Center,
Dunajska 160, 1000 Ljubljana
Tel : 01 568 71 00
http://www.perftech.si • e-mail:prodaja@perftech.si

REPROMS LJUBLJANA

rrc
RRC Računalniške storitve d.d.
Jadranska 21, Ljubljana
Tel.: 01 / 4778 500, Faks: 01 / 4255 229
www.rrc.si, info@rrc.si

SIQ ISO 9001 Q-231

SIEMENS
Dunajska 22, 1511 Ljubljana, Slovenija

SMART COM d.o.o.
Brnčičeva 45, 1001 Ljubljana, Slovenija
tel: + 386 01 56 11 606

SRC.SI
Tržaška cesta 116, 1000 Ljubljana
Tel.: 01 423 32 32 • Fax: 01 423 41 73
e-mail: src@src.si • http://www.src.si

H 433748

vsebina

uporabna INFORMATIKA

uvodnik

Spoštovane bralke in bralci,

Uredništvo revije Uporabna informatika me je povabilo, da v tem letu pripravim in uredim tematsko številko Objektne tehnologije. Le-ta je zdaj pred vami. Poskušal sem zbrati prispevke, ki bi tematiko prikazali z različnih vidikov, tako da bi vsak od bralcev revije našel v njej kaj novega in zanimivega.

Objektna tehnologija kot paradigma razvoja informacijskih sistemov ni nova. Navadili smo se, da se z njo srečujemo v vseh fazah razvoja programske opreme. Pred nekaj leti smo jo v Sloveniji označevali kot novost, ki še ni prodrla v vsakdan informacijskih oddelkov in podjetij za razvoj programske opreme. Danes je drugače – objektna tehnologija se je nedvomno uveljavila kot tisti pristop k razvoju informacijskih sistemov, ki nudi konkurenčne prednosti. V fazi analize in načrtovanja omogoča veliko bolj naraven in intuitiven pristop, saj lahko entitete iz realnega sveta direktno preslikamo v programske konstrukte. Enoten pogled na podatkovni in procesni vidik ne zahteva dodatnih transformacij in omogoča enostavnejšo izvedbo iterativno inkrementalnega procesa razvoja. V fazi implementacije nam objektna tehnologija ne ponuja samo objektnih programskih jezikov. Z njo so neločljivo povezani komponentni razvoj programske opreme, porazdeljeni komponentni modeli, večnivojska arhitektura in aplikacijski strežniki. Tudi objektne in objektno relacijske podatkovne baze, objektne transakcijske storitve, deklarativni pristop k sistemskim storitvam, testiranje, vzorci in sorodna področja se tako ali drugače naslanjajo na objektno tehnologijo.

Objektna tehnologija ni sama sebi namen. Uporaba konceptov in tehnik zmanjšuje kompleksnost razvoja programske opreme, kar je ključen in najpomembnejši cilj vsake inženirske discipline. Zaradi tega se lahko programska oprema, razvita po načelih objektne orientacije, pohvali z večjo uporabnostjo, hitrejšimi razvojnimi cikli, z lažjim vzdrževanjem, višjo kakovostjo in nižjimi stroški razvoja. Po drugi strani pa prav objektna tehnologija omogoča in zagotavlja povezljivost aplikacij na visokem nivoju, to je integracijo obstoječih parcialnih informacijskih sistemov v enovit, povezan sistem, ki podpira poslovanje celotnega podjetja in integracijo vnaprej izdelanih programskih komponent.

Kot vsaka druga tehnologija pa tudi objektna tehnologija zahteva izobraževanje. Opisane prednosti lahko dosežemo le z ekipo izobraženih in izšolanih razvijalcev, ki znajo tehnologijo uporabiti na pravilen način. Uporaba objektnih orodij brez ustreznega znanja ne bo dala pričakovanih rezultatov.

V tej tematski številki obravnavamo nekatera od omenjenih področij. V prvem prispevku Architectural Design for Performances, *povabili smo tujega avtorja, se dotaknemo problema zmogljivosti oziroma prepustnosti ter časovne odzivnosti kot funkcije arhitekturnih lastnosti sistema. To področje je v praksi vse preveč zanemarjeno, povzroči pa lahko marsikatero presenečenje, če že od začetka projekta nismo bili na ta problem pozorni. Drugi prispevek* Razvoj programske opreme z uporabo jezika UML *je informativen: seznanja z jezikom UML in njegovim pomenom za objektno modeliranje. Prispevek hkrati tudi opozori na dejstvo, da pri uporabi objektne tehnologije ni enovitega razvojnega procesa. Ta se mora definirati v podjetju skladno z obstoječo kulturo dela, kot osnovno paradigmo pri tem pa lahko izkoriščamo ideje iterativno-inkrementalnega razvoja. V tretjem prispevku* Integracija – ključ do učinkovitega informacijskega sistema *avtorja naslavljata problematiko integracije pri vključevanju obstoječih informacijskih rešitev v celovit, povezan informacijski sistem podjetja. Pri tem podajata tako organizacijske, kakor tudi tehnološke vidike integracije. Za dosego zadovoljivih rezultatov razvoja programske opreme je ključnega pomena vzpostavitev ustreznega procesa in integracija le-tega v okolje. V prispevku* Oblikovanje okolja za ekstremno programiranje *avtorji prikazujejo enega od načinov razvoja, ki temelji na skupinskem delu in kodiranju v parih. Vsekakor zanimiva tema, ki ji velja posvetiti pozornost. Prispevek z naslovom* Objektna tehnologija v sistemih nadzora in vodenja produktne linije SI2000 verzije 5 *daje vpogled v celovito uporabo objektnih konceptov pri enem največjih slovenskih razvojnih projektov, povezanim s telekomunikacijami. Vsi, ki pa se že dalj časa ukvarjate z razvojem aplikacij, boste z zanimanjem prebrali prispevek* Refactoring – preoblikovanje programske kode, *v katerem kanete dobiti navdih za uporabo preoblikovanja tudi v vaših programskih rešitvah. Sledi teoretični prispevek* Obnavljanje načrtovanja s pomočjo vzorcev načrtovanja, *ki razrešuje problem kako vzorce nedvoumno zapisovati, hkrati pa predstavi analizo uporabe vzorcev nekaterih poznanih razrednih knjižnic in objektnih sistemov. Slednjič pa vas opozarjam na zanimivi prispevek* Zlepki v objektno usmerjenem okolju, *kjer boste spoznali, kakšne funkcije imajo zlepki, kakšnim namenom služijo v informatiki in kako jih predstavljamo v objektnem okolju.*

V upanju, da objektna tehnologija že rešuje vaše največje težave, vas lepo pozdravljam in vam želim zanimivo branje

prof. dr. Ivan Rozman
(gostujoči urednik)

To revijo sofinancira Ministrstvo za znanost in tehnologijo Republike Slovenije
Izhajanje revije je v letu 2000 podprla Univerza v Ljubljani, Visoka upravna šola

Zahvaljujemo se podjetju Marand d.o.o., Ljubljana, Cesta v mestni log 55, za sponzoriranje domače strani Slovenskega društva INFORMATIKA

Navodila avtorjem

Revija Uporabna informatika objavlja originalne prispevke domačih in tujih avtorjev na znanstveni, strokovni in informativni ravni. Namenjena je najširši strokovni javnosti, zato je zaželeno, da so tudi znanstveni prispevki napisani čim bolj mogoče poljudno.

Članke objavljamo v slovenskem jeziku, prispevke tujih avtorjev pa tudi v angleškem jeziku.

Vsak članek za rubriko Strokovne razprave mora za objavo prejeti dve pozitivni recenziji.

Prispevki naj bodo lektorirani, v uredništvu opravljamo samo korekturo. Po presoji se bomo posvetovali z avtorjem in članek tudi lektorirali.

Polno ime avtorja naj sledi naslovu prispevka. Imenu dodajte naslov organizacije in avtorjev elektronski naslov. Prispevki za rubriko Strokovne razprave naj imajo dolžino cca 30.000 znakov, prispevki za rubrike Rešitve, Poročila, Obvestila itd. pa so lahko krajši.

Članek naj ima v začetku Izvleček v slovenskem jeziku in Abstract v angleškem jeziku. Izvleček naj v 8 do 10 vrsticah opiše vsebino prispevka, dosežene rezultate raziskave.

Pišite v razmaku ene vrstice, brez posebnih ali poudarjenih črk, za ločilom na koncu stavka napravite samo en prazen prostor, ne uporabljajte zamika pri odstavkih.

Revijo tiskamo v črno beli tehniki s folije, zato barvne slike ali fotografije kot originali niso primerne. Objavljali tudi ne bomo slik zaslonov, razen če so nujno potrebne za razumevanje besedila. Slike, grafikoni, organizacijske sheme itd. naj imajo belo podlago. Po možnosti jih pošiljajte posebej, ne v okviru članka.

Na koncu članka navedite literaturo, ki ste jo uporabili za prispevek, po naslednjem vzorcu:

Novak, F., Bernik, S.(1999): »Naslov članka«, ime revije, letnik, štev., str. 12-15

Bernik,S.: (1999): »Naslov knjige«, založba, kraj

Novak, F.(1999): »Naslov magistrskega dela«, magistrsko delo, univerza, fakulteta

Žagar, A.: »Naslov referata«, Dnevi slovenske informatike, Zbornik posvetovanja, Slovensko društvo INFORMATIKA (1998)

V besedilu članka se sklicujte na navedeno literaturo na način (Novak 1999).

Članku dodajte kratek življenjepis avtorja (do 8 vrstic), v katerem poudarite predvsem delovne dosežke.

Z vsa vprašanja se obračajte na tehnično urednico Katarino Puc. Prispevke pošiljajte na disketi in papirju na naslov Katarina Puc, Slovensko društvo informatika, Vožarski pot 12, 1000 Ljubljana, ali samo po elektronski pošti na naslov katarina.puc@drustvo-informatika.si.

Po odločitvi uredniškega odbora, da bo članek objavljen v reviji, bo avtor prejel pogodbo, s katero bo prenesel vse materialne avtorske pravice na društvo INFORMATIKA. Po izidu revije pa bo prejel plačilo avtorskega honorarja po tedaj veljavnem ceniku ali po predlogu glavnega in odgovornega urednika.

Naslov uredništva je:

Slovensko društvo INFORMATIKA, Uredništvo revije Uporabna informatika, Vožarski pot 12, 1000 Ljubljana
www.drustvo-informatika.si/posta

© Slovensko društvo INFORMATIKA, Ljubljana

Revija Uporabna informatika bo brezplačno objavljala v rubriki Koledar prireditev datume strokovnih srečanj, posvetovanj in drugih prireditev s področja informatike. Obvestila naj vsebujejo naslednje podatke: ime srečanja, datum in kraj prireditve, naziv organizatorja, ime in telefonska številka kontaktne osebe. Pošiljajte jih na naslov: Slovensko društvo Informatika, za revijo Uporabna informatika, rubrika: Koledar prireditev, 1000 Ljubljana, Vožarski pot 12. Objavljali bomo vsa obvestila, ki bodo prispela 30 dni pred objavo revije.

ARCHITECTURAL DESIGN FOR PERFORMANCE: DETERMINING DISTRIBUTED SYSTEM SPEED FROM AN ARCHITECTURAL PERSPECTIVE

Patricia Carando*
Align360, 1430 Spring Hill Road, Suite 510, McLean, VA 22102, USA

Abstract

Few aspects of system development cause more concern to designers and frustration to users than performance. Most designers believe that they are designing with performance as a primary concern. Why then, are systems deployed that are significantly slower than anticipated, sometimes resulting in a complete design overhaul to meet performance needs? Experience suggests that the reasons are two-fold: (i) A failure to focus at an architectural level when designing for performance; (ii) An inability to gather requisite performance metrics early enough in the design cycle to affect the development outcome. This paper recommends two approaches that address these failings: 1) How to focus on total system throughput based on Use Case scenarios when designing for performance, and 2) How to create an architectural prototype that is used to gather performance metrics prior to making firm design decisions.

Keywords: Distributed system performance, architecture-centric design, architectural prototype, data-intensive Java application tuning

Izvleček

Le redki vidiki razvoja sistemov povzročajo toliko skrbi razvijalcem in toliko slabe volje uporabnikom kot zmogljivost. Razvijalci zvečine mislijo, da pri razvoju poskrbijo za največjo možno zmogljivost. Zakaj se torej uporabljajo sistemi, ki so znatno počasnejši kot je bilo pričakovano, tako da je včasih potrebno zasnovo popolnoma prenoviti, da bi dosegli ustrezno zmogljivost? Izkušnje kažejo, da sta za to dva razloga: 1) V želji, da bi imel sistem kar največjo zmogljivost, se razvijalci ne usmerjajo na določeno raven arhitekture; 2) Razvijalci nimajo možnosti, da bi že med razvojem sistema pravočasno preverili njegovo zmogljivost in tako prilagodili rezultat razvoja. Članek priporoča dva načina reševanja teh slabosti: 1) Upoštevali naj bi delovanje celotnega sistema na temelju scenarijev možne uporabe in 2) Še pred dokončno odločitvijo o njegovi zasnovi naj bi zmogljivost sistema preverjali na prototipih.

■ ■ ■

1. Introduction

Determining if a distributed architecture will meet its performance constraints is a daunting task. Often this determination is made after a considerable percentage of the system has been created; if the constraints are unmet, a performance release is planned. This performance release may involve re-architecting the system based on the newly gathered metrics.

This situation is brought about not because of a lack of concern about performance on the part of designers; rather, it is a result of the fact that one can't measure what doesn't exist. Measuring system performance just before first deployment is too late in the life cycle to impact design decisions, but this is often the first time that a sufficient amount of the architecture has been implemented to allow for metrics gathering.

How can this seemingly circular dilemma be addressed? How can metrics be gathered to validate the performance of an architecture prior to actually implementing it? One way is the creation of an archtectural performance prototype (APP). The purpose of the APP is to implement the most important design decisions relating to performance sufficiently to verify them. Failing that, the APP is an opportunity to experiment with alternate designs that can meet the system performance criteria.

* *Patricia Carando has been designing and building distributed systems for 16 years. Early research work included Distributed Artificial Intelligence systems applied to oil well exploration. For the last 10 years, Ms. Carando has been consulting on commercial, distributed system development in a variety of industries. These include telecommunications, materials provisioning, document management, and system design for fault-tolerant computing. She is currently a principal in the electronic commerce company The e4Speed Initiative in McLean, Virginia.*

This paper illustrates the following:

- How to focus on total system throughput based on Use Case scenarios when designing for performance, and
- How to create an APP that is used to gather performance metrics prior to making firm design decisions.
- Focus of the Recommendations

While applicable to n-tier distributed systems in general, the recommendations in this paper target a Java-based n-tier system, with a significant relational database aspect. A typical n-tier system architecture of this type is illustrated in Figure 1. Because many current Web applications and new electronic commerce applications are of this character, these recommendations are broadly applicable. Of particular concern is the performance of systems implemented in Java. The advent of Enterprise Java Beans (EJB) [7] has made Java server implementation very attractive because of the ease of implementing and deploying a multi-user server. Enhancing the performance of such Java-based servers is an increasingly important issue.

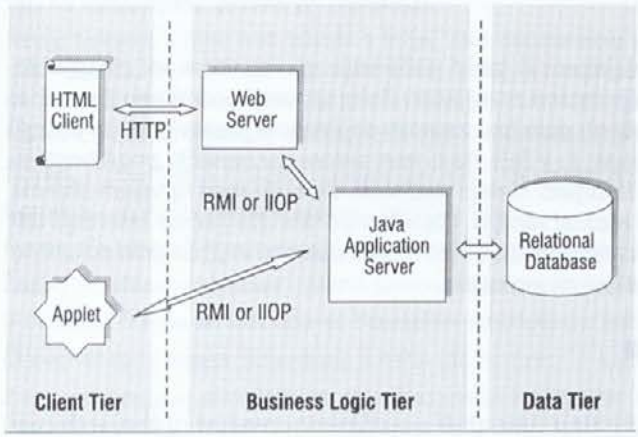


Figure 1: N-Tier Architecture

2. Creating the Architectural Performance Prototype

The body of this paper addresses how to create an APP for a system such as that shown in Figure 1. This figure illustrates a typical architecture that supports Web-based applications. In the *Data Tier*, one or more relational databases provide persistence for the application. In the *Business Logic Tier*, a Java-based Object Request Broker (ORB) or a Java application server applies business rules to data accessed and updated through the *Data Tier*. In the *Client Tier*, one or more implementations of a user interface that support the system's Use Cases direct the activities of the system. Depending on the server technology, the Java server and the Web server functionality may be combined into a single server.

The recommendations cover the following:

a) The selection of one (but not more than two) Use Cases from the system Use Case model that are most likely to represent the most data-intensive or computationally intensive activities.
b) Creation of stimulators to the systems services (databases, Java servers, Web servers) to determine their throughput under conditions of light and heavy use, given the chosen Use Cases.
c) Measurements of the servers' throughput under conditions of heavy and light load to determine lower bounds for performance, as well as typical or average throughput.
d) Suggestions for modifying design and managing expectations should preliminary performance indicators be less than optimal.

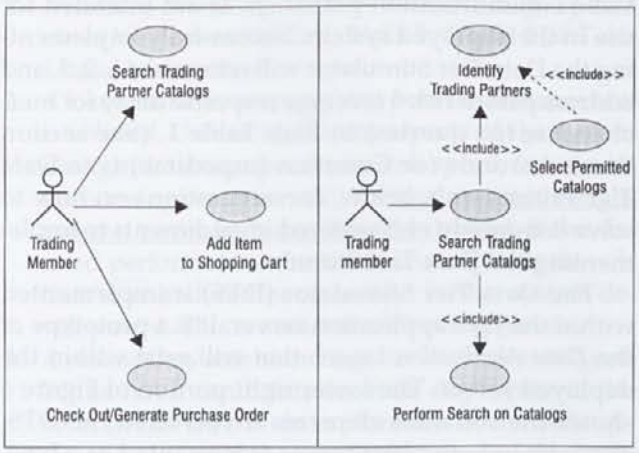
Recommendations in this paper are based on experiences in using this approach on several different projects. As is probably apparent from the context of the suggestions, the approach is best applied as part of the Rational Unified Process [2] (RUP) and can be considered one aspect of RUP's recommendation to develop an architectural prototype.

Selecting Use Cases for Prototyping

When the majority of use cases for a system have been defined, the analyst can begin to scope the use cases for risk. In this paper, we are interested in performance risk, but it is wise to include issues of criticality[1] in the choice of the use cases, as well. To illustrate this, we introduce an example.

Suppose that designers are building a business-to-business eCommerce system that allows corporate trading partners to electronically generate and transmit purchase orders for products. The system allows a trading member to browse their partners' catalogs, select merchandise for purchase, and to generate one or more purchase orders for electronic transmission to the partner corporation.[2] Such a set of capabilities is shown as use-cases in the left-hand side of Figure 2. On the right hand side of the figure is illustrated a more detailed break down of the elements in the *Search Trading Partners' Catalogs* use-case. These include *Identify Trading Partners*, possibly *Select Permitted Catalogs* (based on the Trading Member's alliances) and performing a *Search **on Catalogs*** that meets the Trading Member's *search* criteria.

1 *These are essential functions that the system must perform.*
2 *This example is simplistic, but embodies many of the issues that complex systems face: multiple data sources, inconsistent schema, and international locations that observe varied up times.*



Business-to-Bussiness eCommerce: a Simple Use Case | **Detail of the Use Case for Search Trading Partner Catalogs**

Figure 2: Use Cases for a Shopping System

Criticality and Risk in the Architectural Performance Prototype.

As the designers drill down into the descriptions of the use-cases, they note that the *Search Trading Partner Catalogs* use-case description embodies many unknowns. (These are based on the preconditions, main flow of events, exception flows, and additional notes embodied in the use case description in Figure 3.) Their risk analysis, based on this use-case, is shown in *Table 1*. The table lists a brief description of the risk and an estimate of the severity of the risk.

Risks 1 – 3 in *Table 1* address *criticality* risks—in order to access data from the data sources, logins must be created and schema must be known. These first three items are listed as risks in the table not because of their technical difficulty, but because of expected delays in setting up these capabilities due to bureaucracy and communications problems. Risk 4 is a contingency risk based on the possibility that data may be differently formatted or inconsistent amongst data sources and may need to be transformed to a common format. Risk 4 can be considered both a criticality and a performance risk. Risk 5 also is a criticality and performance risk: when data sources go offline, either intentionally or through some fault, data is not available and connection attempts may cause performance delays. Risks 6 and 7 are purely performance risks: one factor in total system throughput is the response of the data sources (Risk 6). Another factor is the ability of the server to process the data for presentation to the Trading Member.

Search Trading Partners' Catalogs Use Case Description

Search all catalogs of Trading Member's trading partner companies for the items matching the search criteria.

Preconditions:
- Login to trading partner site and access catalog.

Main flow of events
- Issue query
- Consolidate responses

Exception flows:
- Trading partner site may be down for PM
- Trading partner site may be unavailable because of network or system failure

Additional Notes:
- Trading partners are located globally;
- Access characteristics of the trading partner catalogs (schema, login authorization, network connectivity, etc.) are unknown.

Figure 3: Search Trading Partners' Catalogs Use Case Description

#	Task	Risk
1	Identify instance of each catalog	Low
2	Identify relevant query to access item list for each catalog	High
3	Identify login account and permissions for each trading partner site	Moderate
4	Transform differently formatted information into a common display format.	High
5	Determine scheduled outages of partner sites	High
6	Determine average response delay for item queries	High
7	Configure middleware for optimal performance given user load, data accessed.	High

Table 1: Risk Analysis for Search Trading Partners' Catalogs Use Case

The designers decide to prototype the *Search Trading Partners' Catalogs* use-case to validate the architecture and to ascertain that it will meet performance expectations. (Other use-cases in the system—*Select Item for Purchase, Purchase Merchandise*—are addressed similarly and deemed to be less of a performance risk: they are being implemented with known components of well-established characteristics.)

Building the Prototype

Having chosen the use-case to be prototyped, the designers must build the prototype that addresses the greatest performance risk. This prototyping effort consists of two activities. The creation of the Data Tier Stimulator and the creation of the Client Tier Stimulator. The intent of the exercise is to determine a rough estimate of system throughput for the most data intensive function.

Consider Figure 4—a variation on Figure 1—showing the Client Tier, Business Logic Tier, and Data Tier

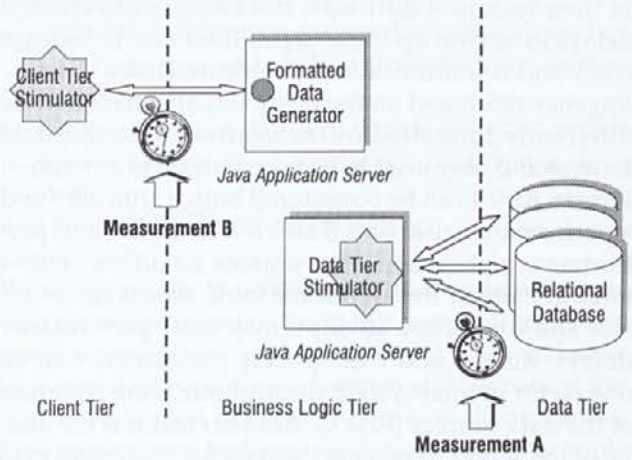


Figure 4: Client and Data Tier Stimulators with Measurement Points

for the distributed application. This figure illustrates the measurement points for the prototype. Measurement A is the Data Tier Stimulator measurement—measuring round-trip time from the server to the data sources. Measurement B is the Client Tier stimulator: a measurement of round-trip time from the client to the server. When the prototype and the measurements have been completed, adding together Measurements A and B should give a rough estimation of minimal throughput times for the most data intensive query.

Addressing the Performance Bottleneck

It is a truism that to optimize the performance of a (distributed) system one must first identify the bottlenecks to performance. Optimizing non-bottlenecks will not increase the throughput of a system. How, then, can one be assured that measuring data access will measure the real bottleneck?

Measuring the total throughput of the system is the goal of the exercise. If data access is a significant aspect of user interactions, it must be a major component of your throughput measurement. Further, data access is (probably) the major source of memory consumption in the server—a notorious source of performance degradation in Java [1], [3], [4], [5], [6]. Having eliminated this as a potential bottleneck in performance, one is free to address other areas of the server where performance issues could arise. Modeling sequence diagrams can be a good source of information on this—pointing out areas where major message activity or computation occurs.

Setting up the Data Tier Stimulator

To determine the response time of the data sources, they must be exercised from the server utilizing the same communication pathways as are intended for use in the deployed system. Successfully implementing the Data Tier Stimulator will retire risks 1, 2, 3, and address part of risk 6 (average response delay for merchandise list queries) in Risk Table 1. (See section Work-Arounds for Common Impediments to Data Tier Prototyping, below, for suggestions on how to solve commonly encountered impediments to implementing the Data Tier Stimulator.)

The Data Tier Stimulator (DTS) is implemented within the Java application server. It is a prototype of the *Data Abstraction Layer*[3] that will exist within the deployed server. The lower right portion of Figure 4 shows the software elements involved in the DTS. These include the Java server (represented as a box), the DTS classes (represented as a star), the communication pathways to the data stores (arrows), and the data stores themselves (cylinders labeled *Relational Database*).

The DTS should issue a query to each of the data stores; the query chosen should be the most data intensive query for the application. Hard-coding the query into a method of each DTS class is sufficient for the prototype. Using a single DTS class per data source is recommended. (See Appendix A for a Java example of a sample DTS class.)

The sequence diagram in *Figure 5* shows the series of events the DTS follows in exercising the data sources. The Merchandise Warehouse starts a timer (see Appendix B for a Java Timer class example) and fans out queries to the three data sources. When all query requests have returned, the timer is stopped. The elapsed time recorded by the timer is the (minimum) time needed to perform the query.

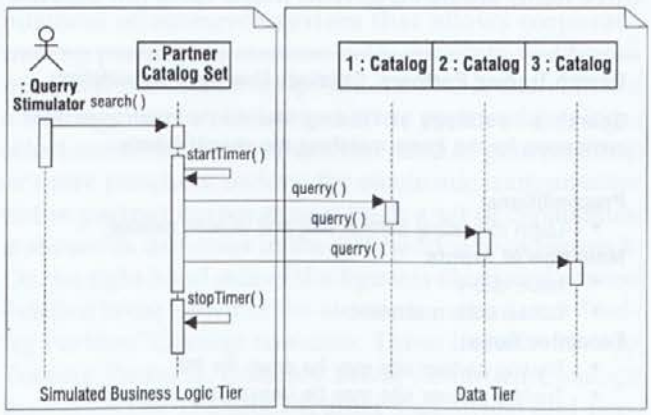


Figure 5: Sequence Diagram of Data Tier Stimulator

While this exercise is deceptively simple, a great deal is accomplished with this prototype:

3 *An architectural layer within a server that insulates the business logic layer from the details of data .*

1. *Necessary access information and knowledge of data stores has been established.* Risks 1, 2, and 3 of *Table 1* have been retired.
2. *A lower bound for accessing data from the data sources is established.* Unless significant aspects of the system change (database speed, network speed, faster multiprocessing on server, etc.) the total system throughput can never be faster than this bound. If the derived performance number is unacceptable there is time to re-architect or reconsider further system development. This is a start at addressing Risk 6.
3. *A rudimentary data abstraction tier has been prototyped.* While not a candidate for final deployment, the code in the DTS is an exploration of the data abstraction layer of the server. Pointing to the functioning prototype that accesses real data can alleviate the fears of stakeholders who may be anxious to begin implementation. This is not a technical issue, but can be very important politically.

Work-Arounds for Common Impediments to Data Tier Prototyping

Even a simple prototype like the Data Tier Stimulator can be difficult to implement. Common problems include:

a) *One or more of the data stores is unavailable.* This can occur for a number of reasons: delays in getting access because of permissions, firewalls, staff availability, etc.
Workaround. Define a test data set on a server that can be used for initial development of your performance prototype until the data store *becomes* available. Determine what the schema of the database is and what you'll have to access to satisfy your user interactions. Get a sample data set and begin your performance prototype using this same set and sample server.

b) *The schema for one or more data stores is undefined.* The general content of a data store may be known, but its full definition may not be in place when the performance prototype is being implemented
Workaround. This situation, while frustrating to performance determinations, can be an opportunity to design the data stores optimally. It is critical that the server designers and data modelers work closely together to design schema that will support queries suggested by the use-cases.

c) *The schema for one or more data stores is changing.* This situation is similar to 2 above, but has the added difficulty that the server will need to support the existing schema until the new schema is operational.
Workaround. In this situation, a very robust data abstraction layer must be implemented that has the same interface for both schema. This will help prevent data updates from rippling through to updates in the entire server.[4]

Setting up the Client Tier Stimulator.

Implementing the Client Tier Stimulator will help to determine the response time to a client request and the memory usage of the server under load. These are issues that are components of Risk 6 (average response delay for merchandise list queries) and Risk 7 (configure middleware for optimal performance given user load, data accessed) in Risk Table 1.

The CTS is implemented within a prototype client that is on a host remote from the server. This client need only issue a request to the server and receive a response. The request must elicit a response based on the same query sent to the data sources by the Data Tier Stimulator. That is, the response should be the consolidated, formatted version of the data that was returned to the Data Tier Stimulator. This is the form the data will have in the deployed system after being merged from the various sources, passed through the data logic, and readied for delivery to the client. While the raw data format and the processed data format may be similar, they are not usually identical. (Some data fields in the raw data format may be suppressed, computed fields added, data fields transformed, etc.)

The sequence diagram in Figure 6 shows the series of events the CTS follows in exercising the server. The Client starts a timer and sends a request to the server. A simulated response is generated and the data is returned to the Client. The timer is stopped at this point The elapsed time recorded by the timer is the (minimum) time needed to return data across the wire to the client. A second timer may be utilized to determine how long the client takes to format the display for the user. Both these timing figures may factor into redesign efforts should elapsed times prove too great.

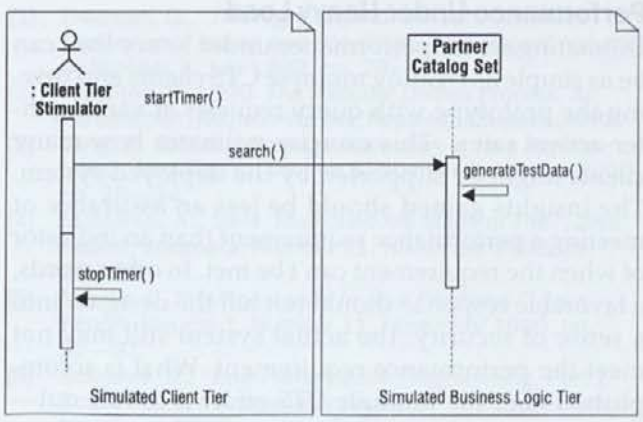


Figure 6: Sequence Diagram of Client Tier Stimulator

4 *Designing to prevent data-update-ripple-through is a discussion topic in its own right and can't be adequately addressed here.*

Determining Performance Bounds

Having completed the two parts of the prototype, the designers can now estimate minimal transit times from client-to-server-to-data–source and back by adding together the measurements from the CTS and the DTS. The minimum response time for the most data intensive query will be CTS + DTS + x, where x is the processing time for applying the business logic and data consolidation rules to the fetched data. If these numbers are "within the ballpark" of acceptable performance, the designers can expand the prototype to look at performance under load and optimal parameters for server sizing. If these numbers are off-scale, then redesign is necessary. Many possibilities for redesign exist; a few are addressed in Section 3.

Memory Configuration

In order to address Risk 7 (configure middleware for optimal performance given user load, data accessed), it is important to know how much memory the server will require to service the heaviest user queries. While there are many other factors that will need to be considered for optimal performance, determining memory usage is one of the most critical for server-side Java. Measuring memory usage for the query employed in the CTS will give an indication of how much memory is dedicated to holding client data. The memory test harness described in [4] can give an initial estimate.

Once a memory usage number is determined, double the figure to get an estimate of the real memory usage in the deployed system. Doubling the initial figure is necessary because the processed version of the data and the raw data will both be resident in memory until the response is sent. Unless the raw data can be read into the server, processed, and efficiently released as the query response is being prepared, roughly twice the memory of the formatted response may be consumed during processing.

Performance Under Heavy Load

Estimating server performance under heavy load can be as simple as creating multiple CTS clients and driving the prototype with query requests at varying inter-arrival rates. This exercise estimates how many clients might be supported by the deployed system. The insights gained should be less an assurance of meeting a performance requirement than an indicator of when the requirement can't be met. In other words, a favorable response should not lull the designer into a sense of security: the actual system still may not meet the performance requirement. What is accomplished with the multiple CTS effort is to rule out—very early—infeasible design approaches.

3. Modifying Design and Managing Expectations

If the performance prototyping of a system goes well and the early metrics indicate the design can easily meet or exceed performance requirements, the designer can happily continue with detailed design. However, should this not be the case, the designer needs to reconsider the approach.

Problems Revealed in the Client Tier Stimulator

Problems revealed by the CTS may include:

1. *The amount of data returned is causing significant delays.* The user may have an unbounded query that returns too much data. Two suggestions for avoiding this situation are:
   Suggestion 1a: Restrict the generality of the query prior to processing.
   Suggestion 1b: If large responses are mandatory, return data incrementally. This breaks the query-response into segments that give the impression of over-all better response.
2. *Response is impacted by memory management infrastructure.* Excessive memory usage has the memory allocator and garbage collector working overtime. If one can't add more memory, consider the following:
   Suggestion 2a: Prefer primitive types. Excessive object creation can cause significant memory allocation overhead [4]. Consider where primitives can replace objects in the design.
   Suggestion 2b: Use less memory by processing query responses–and the raw data that supports them–incrementally. (This is a variation of 1b above.)

Problems Revealed in the Data Tier Stimulator

Early detection of performance problems in the DTS can help in re-architecting (at best) or managing user expectation (at worst). A common problem found by the DTS is:

- *The database query response is too slow.* If the speed of the network and the database servers are not the culprits in slow database performance, enhancement may be accomplished by the following:
  Suggestion a. Change the schema. Optimizing the schema to better fit the needs of the user query can result in significantly better performance.
  Suggestion b. Modify the user interaction. If the query involves the execution of many subqueries, it is beneficial to try to simplify the use-case to minimize the complexity of queries. This approach should only be taken when the schema can't be modified.

When neither alternative is possible, the designer must inform the user community that response will

fall outside the performance boundaries. This knowledge is best transmitted before the developed system is erroneously found at fault.

4. Conclusions

Determining the performance characteristics of a distributed architecture is an anxiety-provoking task, particularly if these characteristics can only be determined after the system has been created. This paper has illustrated how to ascertain these characteristics early in the design process so as to avoid the costly effort of re-architecting after an implementation. Steps illustrated include:

How to select the critical, data intensive use-cases from a risk list for performance prototyping.

a) How to employ the use-cases in creating Data-Tier and Client-Tier-Stimulators to derive performance metrics.
b) How to modify the design and manage expectations if preliminary performance indicators are poorer than what was anticipated.

While not applicable to all distributed system development, these approaches should serve as useful techniques in optimizing for server-side Java systems.

Appendix A.

A Sample Data Store Stimulator Class

The sample class Catalog1 of a class used in the Data Tier Stimulator. It would be employed as illustrated in *Figure 5: Sequence Diagram of Data Tier Stimulator.*

1. Create the instance. Initialize the timer
2. Call the Query method
   - Invoke the startTimer method: Starts the timer (See Appendix B, below, The Java Timer class)
   - Get a connection to the database
   - Execute the hard-coded SQL query
   - Stop the timer
   - Print out the elapsed time for performing the query

```
package Performance;
    // Copyright:   Copyright (c) 2000
    // Author:      P. Carando
    // Description:  A simple data tier stimulator
    public Catalog1 = new
                    Performance.Timer ();
     }
     public void query () {
      try {
       startTimer ();
       java.sql.Connection conn =
          java.sql.DriverManager.getConnection("<db url>");
       java.sql.Statement stmt = conn.createStatement ();
       java.sql.ResultSet r = stmt.executeQuery ("<query>");
       stopTimer ();
       System.out.println ("w1: " +  aTimer.elapsedTime ());
      } catch (java.sql.SQLException ex) {
       stopTimer ();
       System.out.println ("w1 exception: " + ex);
      }
     }
```

```
     private void startTimer () {
       aTimer.reset ();  aTimer.start ();
     }
     private void stopTimer () { aTimer.stop (); }
     }
```

Appendix B.

A Java Timer Class

This simple Java timer class uses the System clock to measure elapsed time in milliseconds.

```
package Performance;
    // Title:       Timer
    // Copyright:   Copyright (c) 2000
    // Author:      P. Carando
    // Description:  A simple timer
    public class Timer {
     private long startTime, stopTime, elapsedTime;
     private boolean started, stopped;

     public Timer() { reset (); }
     public void reset () {
       started = false;   stopped = false; startTime = 0;
       stopTime  = 0;   elapsedTime = 0;
     }
     public void start () {
       startTime = System.currentTimeMillis ();
       started = true;
     }
     public void stop () {
       stopTime = System.currentTimeMillis ();
       stopped  = true;
       elapsedTime = Math.max (stopTime - startTime, 0);
     }
     public long elapsedTime () {
       if (!started) return 0;
       if (!stopped) {
         elapsedTime =
           System.currentTimeMillis () - startTime;
       }
       return elapsedTime;
     }
    }
```

References

[1] Freeman, G., *"The Right tools for the Job, Common Performance Issues and Solutions"*, Java Report, Volume 4, Number 7, July 1999, pp. 29—34.

[2] Kruchten, P., 1999. *The Rational Unified Process, An Introduction*, Addison-Wesley, Reading, Massachusetts.

[3] Long, F., *"Avoiding Garbage Collection in Java Applications"*, Java Report, Volume 5, Number 1, January 2000, pp. 28—34.

[4] McManus, A., *"Java: Memories Are Made of This"*, Java Report, Volume 3, Number 11, November 1998, pp. 39—48.

[5] Nylund, J., *"Memory Leaks in Java Programs "*, Java Report, Volume 1, Number 11, November 1999, pp. 22—31.

[6] Sosnoski, D., *"Java Performance Programming, Part 1: Smart Object Management Saves the Day"*, JavaWorld, November, 1999, www.javaworld.com.

[7] Sun Microsystems, *Java 2™ Platform Enterprise Edition Specification, v1.2*, December 17, 1999.

RAZVOJ PROGRAMSKE OPREME Z UPORABO JEZIKA UML

Aleš Živkovič, Marjan Heričko, Ivan Rozman
Fakulteta za elektrotehniko, računalništvo in informatiko
Inštitut za informatiko
e-pošta: {ales.zivkovic, marjan.hericko, i.rozman}@uni-mb.si

Izvleček

V prispevku je predstavljen jezik za dokumentiranje dela in rezultatov pri objektnem razvoju programske opreme. Predstavili bomo zgradbo jezika UML, diagramske tehnike in dodatne izrazne možnosti, vpeljane v obliki dopolnitev k osnovnim gradnikom jezika. Zelo natančen in dobro zasnovan standard pa ne zajema napotkov za uporabo jezika, zato potrebujemo še proces. Spoznali bomo, da univerzalnega procesa ni, prilagoditi ga moramo okolju, v katerem se uporablja, in ga nenehno izboljševati. Podali bomo priporočila za oblikovanje procesa razvoja programske opreme, kadar sledimo objektni paradigmi.

Abstract

In the paper language for documenting software development artifacts will be presented. First the structure, diagram elements and supplemental expressive possibilities introduced as an extension mechanisms for basic techniques will be described. Unfortunately, well-formed standard doesn't define when and how to use its elements. Therefore we need the software development process to define steps, inputs and outputs for each step. There is no single process suitable for all the environments. Process needs to be tailored to environment specifics and continuously improved. Recommendations for forming an object-oriented software development process will also be presented.

■ ■ ■

1. Uvod

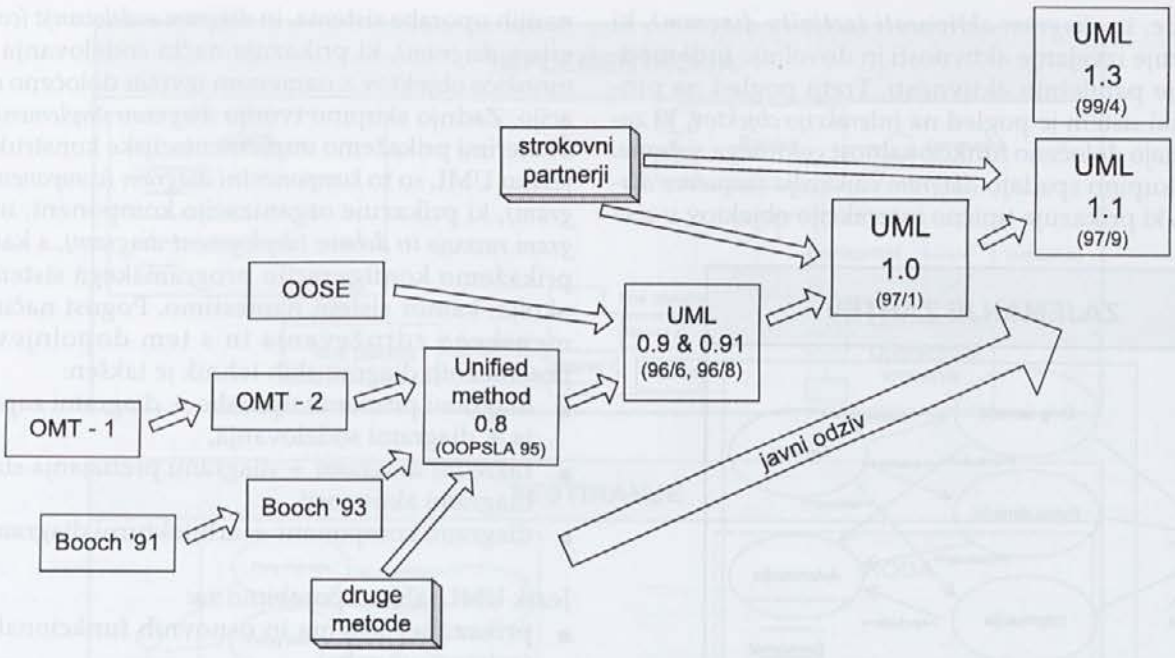
Kljub vedno boljšim integriranim okoljem za razvoj programske opreme, ki nam omogočajo enostavno delo in abstrakcijo kode, ki nastaja, je potreba po modeliranju in načrtovanju programske opreme neizmerna. Tehnološki koncepti, ki vodijo do učinkovitih rešitev, so postali kompleksni. Direkten preskok v implementacijo, brez jasno začrtane poti, je praktično nemogoč, če pa že, pa je zagotovo neučinkovit in časovno potraten. Načrte potrebujemo v takšni ali drugačni obliki. Lahko jih oblikujemo čisto samoiniciativno in na svoj način ali pa sledimo natančno določenim postopkom in korakom. Veliko je odvisno od skupine, tipa in velikosti projekta in nenazadnje organizacije in vodstva. Ne glede na vse to, se razvijalci zavedamo prednosti uporabe posameznih diagramskih tehnik, ki v nobenem pogledu niso zgolj lepe slikice. So način komuniciranja na višjem, bolj razumljivem nivoju abstrakcije in s tem neodvisni od izbranega programskega jezika. Zaradi množice notacij, ki so bile v preteklosti v uporabi, je bila komunikacija med razvijalci otežena in je pogosto povzročala zmedo. Po letu 1996 so različne notacije konvergirale v en sam, standarden nabor diagramskih tehnik, prvič predstavljen na konferenci OOPSLA'95 [3]. Kasneje so pobudo podprla številna podjetja, kot tudi skupina OMG[5]. Vpeljano je bilo ime Unified Modelling Language (UML) [5], ki še bolj poudarja, da proces v njem ni zajet. Dobili smo standarden jezik za dokumentiranje postopkov in rezultatov pri razvoju programske opreme. Značilnosti lahko strnemo v naslednjih točkah:

- je vizualno orodje za modeliranje,
- ima standardizirano notacijo,
- omogoča zajemanje poslovnega procesa,
- izboljša komunikacijo,
- omogoča obvladovati kompleksnost,
- definira arhitekturo programske opreme,
- pospešuje ponovno uporabo,
- omogoča uporabo poljubnega razvojnega procesa.

Razvoj jezika prikazuje slika 1.

2. Jezik UML

Jezik UML definira notacijo in metamodel jezika. Notacija je grafična predstavitev jezika, sintaksa, ki sama po sebi ne daje jasnega in nedvoumnega pomena posameznih konstruktov. Splošna uporaba notacije



Slika 1: Zgodovinski razvoj jezika UML

največkrat definira nekatere neformalne definicije, toda bolje bi bilo poiskati bolj formalno definicijo. Ideja rigoriznih specifikacij in jezikov načrtovanja prevladuje pri uporabi formalnih metod. Kadar uporabljamo tehnike s tega področja, izražamo specifikacije in načrtovanje s pomočjo izpeljank predikatnega računa. Takšne definicije so matematično natančne in ne dopuščajo dvoumnosti, a na žalost niso vsesplošno uporabne. Tudi kadar lahko dokažemo, da program zadovoljuje matematično specifikacijo, nikakor ne moremo dokazati, da te specifikacije v resnici izpolnjujejo uporabnikova pričakovanja.

Avtorji jezika za objektno modeliranje UML so jeziku zagotovili formalno podlago, ne da bi žrtvovali njegovo uporabnost. Ena od možnosti je definicija metamodela – največkrat razrednega diagrama, ki definira notacijo in natančno ter nedvoumno definira pomen posameznih gradnikov. Avtorji so jezik za objektno modeliranje UML zgradili štiriplastno:

- *Meta–metamodel*, ki definira jezik za specifikacijo metamodela in predstavlja infrastrukturo za arhitekturo metamodela.
- *Metamodel*, ki je primerek meta-metamodela in definira jezik za specifikacijo modelov.
- *Model*, ki je primerek metamodela in definira jezik za opis informacijske domene.
- *Uporabniški objekti*, ki so primerki modela in definirajo konkretno informacijsko domeno.

Takšna zasnova jezika omogoča uporabo jezika na vseh področjih, saj vsebuje gradnike, ki omogočajo modeliranje različnih sistemov. Poleg formalne zasnove jezika so avtorji dodali tudi jezik OCL (*Object Constraint Language*) – jezik za določanje omejitev, ki je popolnoma formalen in omogoča oblikovanje omejitev vseh modelov (vloge v asociacijah, številnost povezav, pred- in po-pogoji metod, ipd.). Kot kažejo izkušnje, je kombinacija formalne zasnove jezika za modeliranje prinesla prednosti na dveh področjih. Po eni strani je jezik splošno uporaben, a jasno definiran, kar omogoča uporabo jezika UML na vseh problemskih področjih, po drugi strani pa jasno definirana struktura jezika omogoča lažji razvoj podporne programske opreme.

2.1 Gradniki

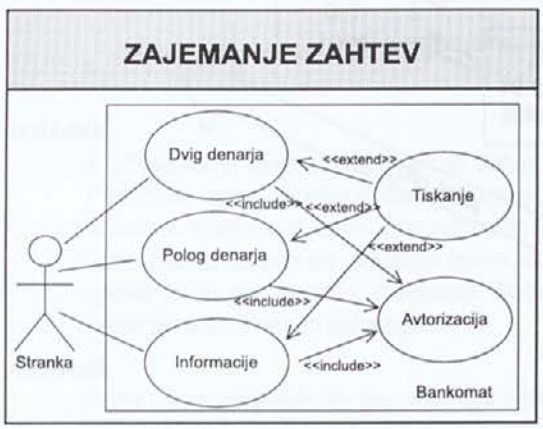
Iz diagramskih tehnik jezika UML lahko razberemo štiri različne poglede na problemsko področje. Prvi je pogled na uporabniške zahteve, kjer uporabljamo *diagram primerov uporabe (use case diagram)*, ki izvira neposredno iz Jacobsonove metodologije in je bil za potrebe jezika UML poenostavljen. V drugo skupino lahko uvrstimo *razredni diagram (class diagram)*, ki prikazuje statično sliko sistema. Razredni diagram je sestavljen iz razredov in povezav med razredi in je le nekoliko spremenjena diagramska tehnika iz metodologije OMT (*Object Modeling Technique, Rumbaugh*)[1]. Tretji pogled predstavlja skupina diagramskih tehnik, ki modelirajo *obnašanje programskega sistema*. Tu najdemo *diagram prehajanja stanj (statechart diagram)*, s katerim opisujemo prehajanja stanj za posamezne

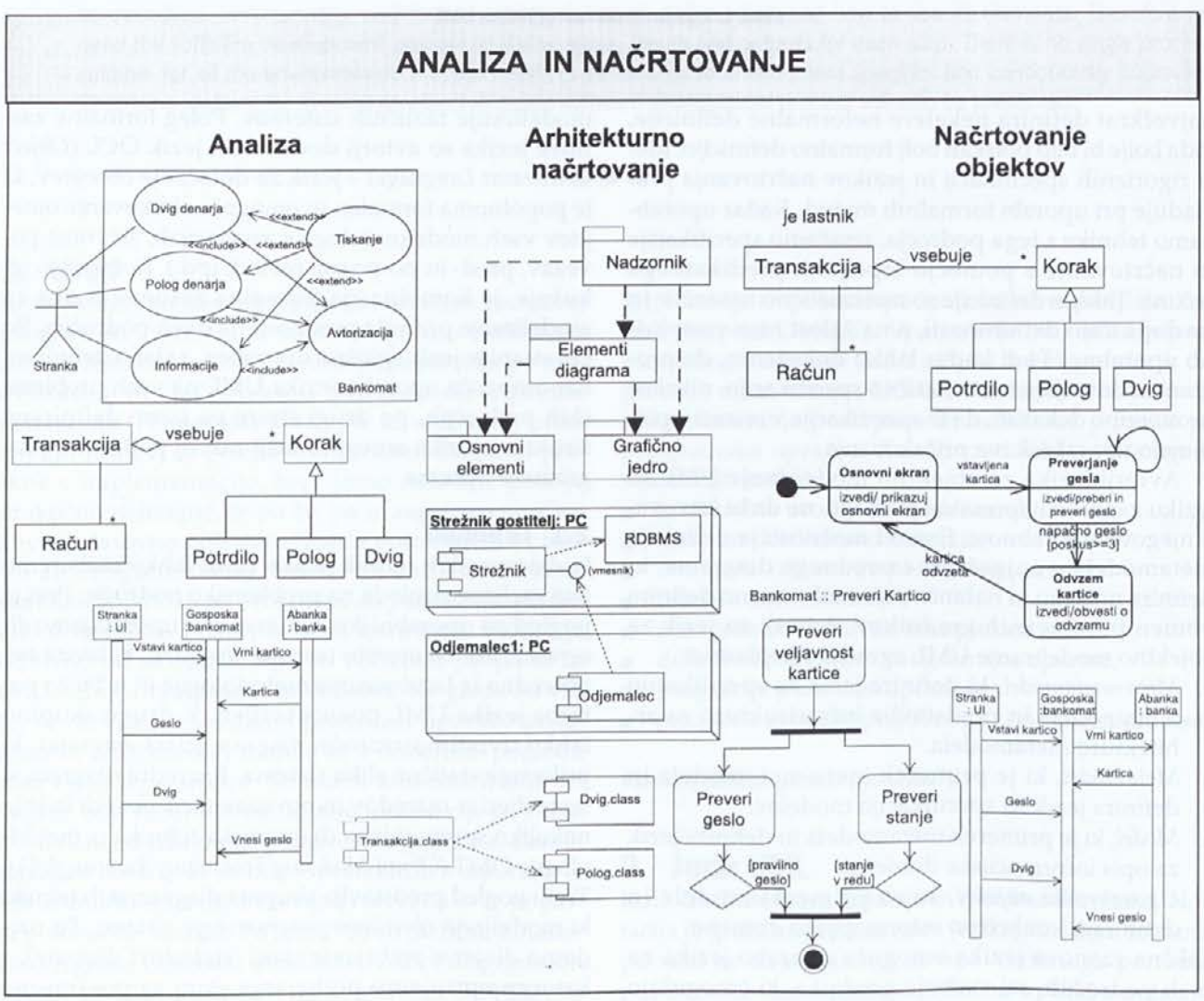
razrede, in *diagram aktivnosti (activity diagram)*, ki prikazuje izvajanje aktivnosti in dovoljuje tudi modeliranje paralelnih aktivnosti. Tretji pogled na programski sistem je pogled na *interakcijo objektov*, ki zagotavljajo določeno funkcionalnost celotnega sistema. V to skupino spadajo *diagram zaporedja (sequence diagram)*, ki prikazuje tipično interakcijo objektov v scenarijih uporabe sistema, in *diagram sodelovanja (collaboration diagram)*, ki prikazuje način sodelovanja med množico objektov z namenom izvršiti določeno operacijo. Zadnjo skupino tvorijo *diagrami implementacije*, s katerimi prikažemo implementacijske konstrukte. V jeziku UML so to *komponentni diagram (component diagram)*, ki prikazuje organizacijo komponent, in *diagram razvoja in dobave (deployment diagram)*, s katerim prikažemo konfiguracijo programskega sistema in okolja, kamor sistem namestimo. Pogost način pomenskega združevanja in s tem dopolnjevanja posameznih diagramskih tehnik je takšen:

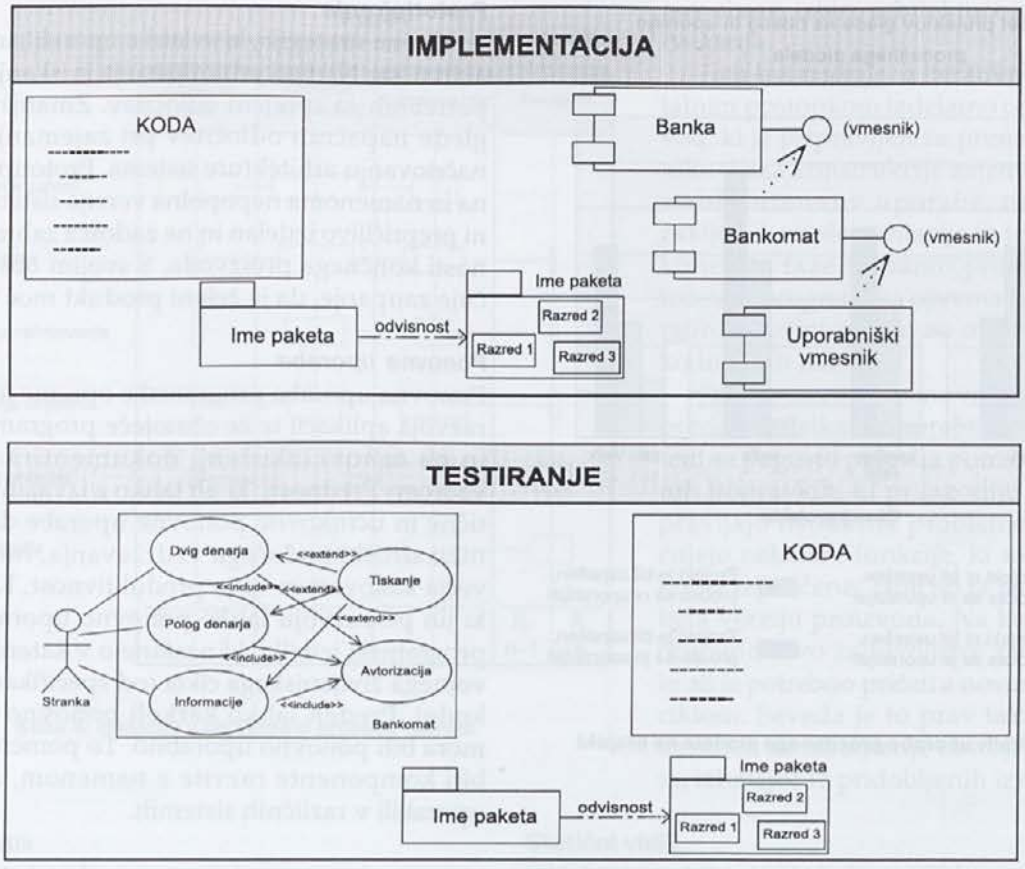
- diagrami primerov uporabe + diagrami zaporedja + diagrami sodelovanja,
- razredni diagrami + diagrami prehajanja stanj + diagrami aktivnosti,
- diagrami komponent + arhitekturni diagrami.

Jezik UML lahko uporabimo za:

- prikaz mej sistema in osnovnih funkcionalnosti (primeri uporabe),







Slika 2: Shematski prikaz uporabe diagramskih tehnik

- prikaz realizacije sistema (diagrami interakcije),
- prikaz statične strukture sistema (razredni diagrami),
- modeliranje obnašanja objektov (diagrami stanj),
- določitev fizične implementacije arhitekture (arhitekturni diagrami).

Osnovne gradnike lahko dopolnimo z naslednjimi koncepti:

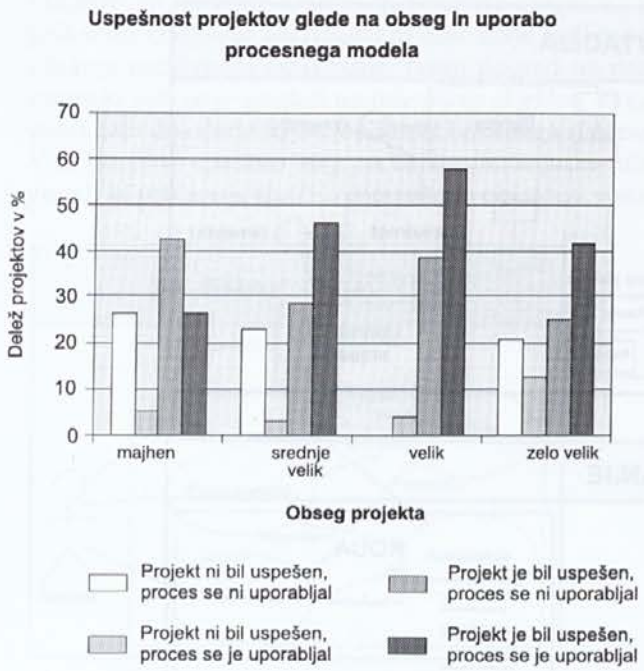
- *Stereotip* (stereotype) - poljubnemu elementu jezika damo nov pomen, oziroma pomen prilagodimo zahtevam razvojnega procesa.
- *Imenovana vrednost* (tagged values) - par oznaka, vrednost lahko dodamo kateremukoli elementu modela. Nekatera imena so že definirana.
- *Omejitve* (constraints) - diagram lahko natančneje opišemo in dodamo natančne pogoje kdaj in pod kakšnimi pogoji se bo nekaj zgodilo.

3. Razvojni proces

3.1 Vpliv procesnega modela na projekt

Iz primerjave vplivov različnih dejavnikov na projekt [7] lahko posredno sklepamo, da je urejenost postopkov in ponovljivost pri izvajanju projektov prav tako pomemben dejavnik, saj je zrelost organizacije bodisi po Zrelostno zmožnostnem modelu (Capability Maturity Model CMM) ali skladno s standardom 9000-3 [6] označena kot tretji najbolj pomemben dejavnik, takoj za človeškim faktorjem in pretokom informacij med vpletenimi v projekt. S poznavanjem CMM in ISO 9000-3 lahko ugotovimo posredno pomembnost uporabe procesnega modela za razvoj programske opreme. Predpostavko potrjuje povezava med uspešnostjo, obsegom projekta in uporabo procesnega modela. Iz slike 3 lahko razberemo, da je uporaba natančno določenih postopkov pri razvoju programske opreme bolj pomembna pri večjih projektih, kjer je odstotek uspešno zaključenih projektov bistveno večji v primerjavi s projekti, kjer se procesni model ni uporabljal.

Povedali smo že, da je jezik UML zgolj notacija in ne metoda razvoja, saj nima definiranega procesa razvoja, ki je pomemben del metode. Avtorji so že na samem začetku ugotovili, da splošnega procesa razvoja ne bo moč razviti, zato predlagajo uporabo ogrodja, kjer si vsak uporabnik proces lahko prilagodi svojim zahtevam in uporabi tiste diagramske tehnike, ki so primerne in smiselne za njegov razvojni proces.



Slika 3: Vpliv uporabe procesnega modela na projekt

3.2 Značilnost objektnega procesnega modela

Predlagamo, da objektni proces razvoja IS združuje naslednje pomembne strategije:

Iterativni razvoj

Za mnoge razvijalce programske opreme je uporaba iteracij ena izmed ključnih značilnosti njihovega dela. Praktično to pomeni odlašanje izvedbe za določeno časovno obdobje z namenom hitrejšega napredovanja in kasnejšega vračanja k istemu problemu, kjer dosežene rezultate izboljšamo oziroma popravimo. S takšnim načinom dela se lažje prebijemo skozi kritične točke dela.

Inkrementalni razvoj

Inkrement pomeni dodatek k nečemu - napredovanje. Inkrement pomeni korak bliže k izpolnitvi zadanih ciljev. Inkrementalni razvoj je strategija napredovanja v malih korakih, da bi prišli do ustreznih rezultatov. Inkrementalni pristop zahteva razdelitev problema na podprobleme tako, da jih lahko rešujemo sočasno in izmenično. Ob rešitvi vsakega podproblema le-tega testiramo in povežemo z ostalimi deli sistema. Izraza iterativni in inkrementalni se pogosto uporabljata v navezi ali izmenično in opisujeta uporabljen procesni model pri razvoju predvsem objektnih aplikacij. Strategiji sta med seboj ločeni in neodvisni.

Prototipiranje

Gre za strategijo, ki jo lahko uporabljamo pri večini aktivnosti. Namen prototipiranja je iskanje informacij, potrebnih za sprejem odločitev. Zmanjšuje tveganje glede napačnih odločitev pri zajemanju zahtev in načrtovanju arhitekture sistema. Prototip je predhodna in namenoma nepopolna verzija sistema. Običajno ni prepričljivo izdelan in ne zadošča zahtevam robustnosti končnega proizvoda. S svojim obstojem povečuje zaupanje, da je želeni produkt moč izdelati.

Ponovna uporaba

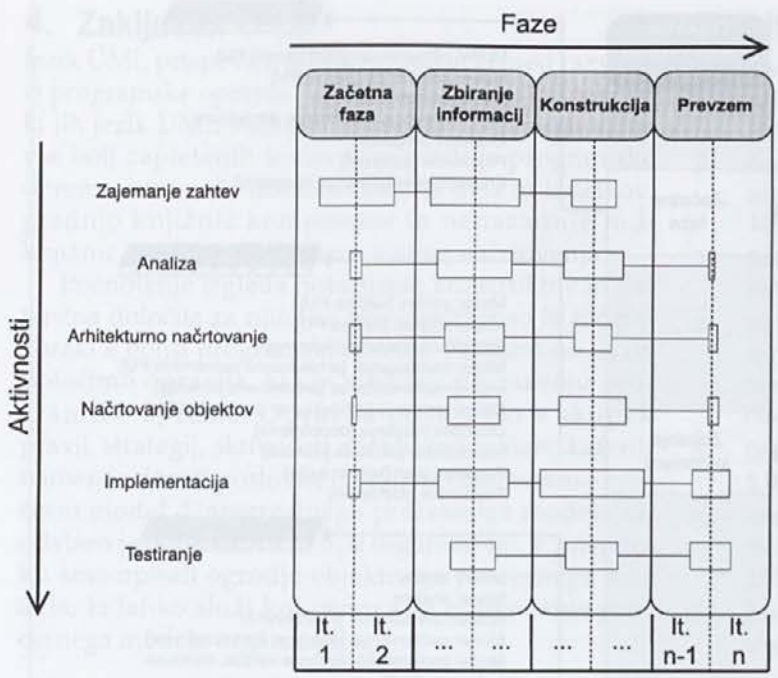
Ponovna uporaba programske opreme je sposobnost razvoja aplikacij iz že obstoječe programske opreme in na osnovi izkušenj, dokumentiranih v obliki vzorcev. Prednosti, ki jih lahko z izvajanjem sistematične in učinkovite ponovne uporabe dosežemo, so nižji stroški bodočega vzdrževanja, hitrejši razvoj, večja kakovost in višja produktivnost. Komponente, ki jih pri razvoju lahko ponovno uporabimo, so vsi programski izdelki, ki nastanejo v katerikoli fazi razvojnega življenjskega cikla (od specifikacij zahtev do kode). Preden lahko karkoli ponovno uporabimo, mora biti ponovno uporabno. To pomeni, da morajo biti komponente razvite z namenom, da jih bomo uporabili v različnih sistemih.

3.3 Zgradba

Proces razvoja programske opreme je razdeljen na posamezne faze. Pri objektnem razvoju IS ločimo štiri osnovne faze (*začetna faza, zbiranje zahtev, konstrukcija, prevzem*). Faze prikazujejo časovno delitev objektnega procesa. Iterativno inkrementalna narava objektnega pristopa narekuje predstavitev procesnega modela v dveh dimenzijah:

- po času – predstavlja življenjski cikel procesa,
- po aktivnostih – predstavlja sestavne dele procesa.

Tovrstna predstavitev v procesnem modelu poveže dva do sedaj ločena pogleda na razvoj informacijskih sistemov. Gledano iz časovne perspektive govorimo o fazah, katere delimo na razvojne cikle - iteracije in mejnike. Tak pogled je značilen za upravljalski nivo gledanja na projekt (projektno vodenje). Le-ta zajema časovno komponento, sredstva, ljudi in organizacijo dela. Gledano iz perspektive proizvodov pa razvojni proces delimo na posamezne aktivnosti. Pri tem ločimo tehnične aktivnosti in podporne aktivnosti (*upravljanje konfiguracij in sprememb, vodenje projektov, uporaba metrik, upravljanje okolja*). Slika 4 prikazuje le tehnične aktivnosti. Iz opisa objektnega procesa razvoja IS so izvzete tudi aktivnosti, ki se ne spremenijo z objektnim pristopom, npr. načrtovanje grafičnega uporabniškega vmesnika, oblikovanje uporabniške dokumentacije, oblikovanje tehnične dokumentacije.



Slika 4: Iteracije v objektnem procesu razvoja

gov obseg, izbiro arhitekture in morebitna tveganja.

Faza konstrukcije - z iterativno inkrementalnim postopkom izdelamo celoten proizvod, ki je pripravljen za prenos k uporabniku. Faza konstrukcije zajema opis preostalih primerov uporabe, načrtovanje, zaključek implementacije in testiranje. Na koncu te faze moramo presoditi, ali so izdelani programska oprema kot tudi uporabniki, pripravljeni za opravljanje vsakodnevnih nalog.

Faza prevzema - osnovni cilj te faze je predaja izdelka v uporabnikovo okolje. S tem se pogosto pojavlja potreba po dodatnih popravkih, ki prilagodijo sistem, popravljajo neodkrite probleme ali dokončujejo nekatere funkcije, ki so bile namenoma izpuščene. Tipično ta faza nastopi z beta verzijo proizvoda. Na koncu te faze ocenimo nivo zadovoljitve zadanih ciljev in ali je potrebno pričeti z novim razvojnim ciklom. Seveda je to prav tako primeren trenutek za izboljšanje obstoječega procesa, izhajajoč iz pridobljenih izkušenj.

Dinamični vidik

Življenjski cikel vsake generacije[1] programske opreme delimo na štiri faze. Vsaka faza se zaključi z definiranim časovnim mejnikom in zahteva sprejem odločitev, ki lahko v veliki meri vplivajo na uspešnost projekta. Vhode in izhode posamezne faze prikazuje slika 5.

Začetna faza - projekt definiramo s poslovnega stališča in določimo njegov obseg. V ta namen poiščemo vse zunanje akterje, s katerimi bo sistem sodeloval, ter v grobem definiramo način sodelovanja. Poiskati moramo vse primere uporabe, opišemo pa le najpomembnejše. Poslovno definiranje projekta zajema določitev kriterija uspeha oziroma neuspeha projekta, ocenitev tveganj, presojo potrebnih virov in fazni načrt, iz katerega so razvidni glavni časovni mejniki.

Faza zbiranja informacij - analiziramo problemsko področje, postavimo osnovno ogrodje arhitekture sistema, izdelamo projektni plan in razrešimo najbolj rizične elemente projekta. Arhitekturne odločitve morajo upoštevati sistem kot celoto, kar zahteva podroben opis večine primerov uporabe in upoštevanje nekaterih dodatnih omejitev. Prototipno realiziramo sistem do te mere, da lahko prikažemo glavne primere uporabe in ovrednotimo izbrano arhitekturo. Na koncu te faze pregledamo podrobne cilje sistema in nje-

1 *Generacijo programske opreme definiramo kot novo verzijo, ki gre skozi vse faze razvoja.*

Statični vidik

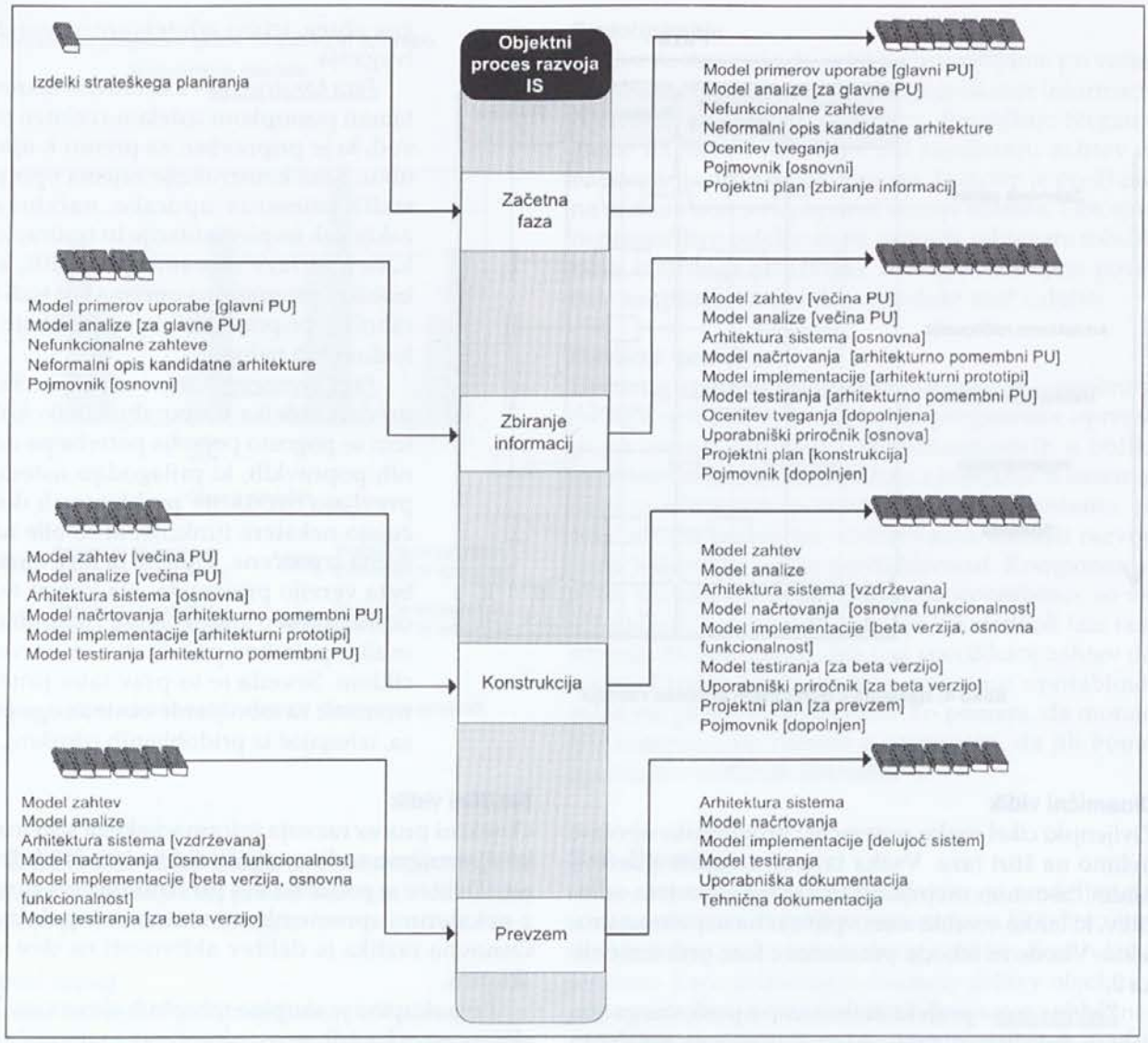
Objektni proces razvoja informacijskega sistema lahko opazujemo tudi s stališča aktivnosti, ki jih izvajamo. Delitev je podobna kot pri strukturnem pristopu z nekaterimi spremembami oziroma dopolnitvami. Osnovna razlika je delitev aktivnosti na dve veliki skupini.

Prva skupina je skupina tehničnih aktivnosti. To so aktivnosti, v katerih se osredotočimo na opravila, ki so popolnoma tehnične narave (npr. identifikacija primerov uporabe, določanje razredov, dodeljevanje odgovornosti).

Druga skupina aktivnosti pa so podporne aktivnosti, ki so namenjene vodenju tehniških aktivnosti in nadzoru projekta. Teh aktivnosti v dokumentu ne bomo opisovali, saj so skupne tako strukturnemu kot objektnemu razvoju. Gre za aktivnosti upravljanja konfiguracij in upravljanja sprememb, aktivnosti projektnega vodenja in aktivnosti upravljanja okolja.

Zajemanje zahtev - poglavitni namen je pridobiti uporabnikove zahteve za nastajajoči sistem v obliki primerov uporabe, lastnosti in ne-funkcionalnih zahtev. Izhod te aktivnosti je model primerov uporabe, scenariji primerov uporabe in opis nefunkcionalnih zahtev.

Analiza – definira, katere operacije in objekti bodo prisotni v razvijajočem se sistemu, ne oziraje se na to, kako bodo te operacije in objekti izdelani. Izhod iz te aktivnosti zajema razredni diagram in vmesnik sistema,



Slika 5: Vhodi in izhodi posameznih faz

ki se odraža v operacijah in dogodkih. Razredni diagram analize oblikujemo na podlagi razrednega diagrama problemskega področja.

Arhitekturno načrtovanje – določa strukturo sistema glede komponent in povezav med njimi. Primarni rezultat arhitekturnega načrtovanja je fizična arhitektura sistema, ki izhaja iz logične arhitekture. Logična arhitektura je prvi približek zgradbe sistema, katero dopolnjujemo in spreminjamo z namenom izdelati optimalno arhitekturno zgradbo sistema.

Načrtovanje objektov - operacije priredimo objektom in sprejmemo odločitve glede dedovanja, vidljivosti in predstavitve povezav (asociacij). Opišemo tudi pristope k sočasnemu izvajanju posameznih delov sistema in s tem v povezavi tudi sinhronizacijo. Izhodi so razredni diagram načrtovanja, diagram sodelovanja objektov, začetna konfiguracija objektov in posodobljen arhitekturni diagram.

Implementacija – rezultate načrtovanja implementiramo v enem izmed programskih jezikov. Pri tem upoštevamo lastnosti izbranega programskega jezika. Potrebno je razviti tako tehnološke kot tudi poslovne razrede. Pri razvoju je potrebno upoštevati izbrano arhitekturo sistema. Upoštevati je potrebno tudi nefunkcionalne zahteve.

Testiranje - izvedemo testiranje izdelka. V sklopu te aktivnosti je potrebno izvesti tako testiranje enot kot tudi integracijsko testiranje. Izdelek aktivnosti je poročilo o testiranju, nastane pa tudi množica testnih primerov in testnih vzorcev, s katerimi zagotovimo ponovljivost testiranja. V primeru, ko želimo testiranje avtomatizirati, razvijemo testne razrede.

4. Zaključek

Jezik UML prispeva k lažji komunikaciji med razvijalci programske opreme. Uporaba diagramskih tehnik, ki jih jezik UML vsebuje, poenostavlja razumevanje vse bolj zapletenih konceptov gradnje programske opreme, omogoča dokumentiranje dela in izdelkov, gradnjo knjižnic komponent in nenazadnje tudi knjižnic idejnih rešitev skozi vzorce načrtovanja.

Poenotenje izgleda notacijskih konstruktov in natančna določila za njihovo povezovanje so le začetni koraki k boljši programski opremi. Pomembno je, da določimo opravila, ki jim sledimo pri razvoju programske opreme. Opravila povežemo v skupek pravil, strategij, aktivnosti, metod in korakov, katerih namen je doseči poslovne cilje, in jih imenujemo procesni model. Univerzalnega procesnega modela ni, odvisen je od velikosti in tipa organizacije. V prispevku smo opisali ogrodje objektnega procesnega modela, ki lahko služi kot osnova za oblikovanje procesnega modela organizacije.

5. Literatura

1. Derr Kurt W., *Applying OMT : A practical step-by-step guide to using the object modeling technique*, SIGS Books, 1995
2. Fowler M, Kendall S, *UML Distilled – Applying the standard object modeling language*, Addison-Wesley, 1997
3. Grady Booch, James Rumbaugh, *Unified Method*, Rational Software Corporation, 1995
4. Inštitut za informatiko, *Enotna metodologija razvoja informacijskih sistemov - 4. Zvezek: Objektni razvoj IS tehnično poročilo*, FERI Maribor Institut za informatiko, november 1999
5. *OMG Unified Modeling Language Specification*, version 1.3, June 1999, http://www.omg.org/
6. Rozman et al., *PROCESSUS - Integration of SEI CMM and ISO quality models*, Software Quality Journal, Marec 1997, str. 37-63
7. Živkovič Aleš, Rozman Ivan, *Značilnosti uspešnih projektov*, OTS'99 zbornik prispevkov, str. 162 - 168

◆

Dr. Ivan Rozman je redni profesor Univerze v Mariboru, dekan Fakultete za elektrotehniko, računalništvo in informatiko v Mariboru in ustanovitelj Laboratorija za informacijske sisteme, ki ga vodi še danes. Je avtor številnih publikacij in vodja več raziskovalnih projektov. Diplomiral je na Fakulteti za elektrotehniko v Ljubljani, magistriral in doktoriral pa na Tehniški fakulteti v Mariboru.
e-pošta: i.rozman@uni-mb.si

◆

Dr. Marjan Heričko je docent na Inštitutu za informatiko na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru. Njegovo raziskovalno-razvojno delo obsega vse vidike objektne tehnologije, s poudarkom na metodologijah razvoja, orodjih CASE, razvojnih okoljih in metrikah. Svoja spoznanja in izkušnje je predstavil v številnih prispevkih na domačih in tujih konferencah ter revijah. Aktivno sodeluje pri koordiniranju aktivnosti Centra za objektno tehnologijo ter vodi organizacijo strokovnih srečanj OTS Objektna tehnologija v Sloveniji. Diplomiral, magistriral in doktoriral je na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru.
http://lisa.uni-mb.si/osebje/hericko; e-pošta: marjan.hericko@uni-mb.si

◆

Mag. Aleš Živkovič je asistent na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru in aktiven član Centra za objektno tehnologijo, v okviru katerega je sodeloval pri pripravi in izvedbi številnih seminarjev in delavnic iz področja Jave, objektnega načrtovanja, porazdeljenih objektnih sistemov in interneta. Je avtor številnih domačih in tujih člankov. Njegovo raziskovalno delo zajema javansko tehnologijo, področja objektne tehnologije, projektnega vodenja in interneta. Je ustanovitelj in koordinator skupine JUGSI (Java User Group of Slovenia). Diplomiral in magistriral je na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru.
e-pošta: ales.zivkovic@uni-mb.si

INTEGRACIJA – KLJUČ DO UČINKOVITEGA INFORMACIJSKEGA SISTEMA

Matjaž B. Jurič, Ivan Rozman
Univerza v Mariboru, Inštitut za informatiko, Fakulteta za elektrotehniko, računalništvo in informatiko
E-pošta: matjaz.juric@uni-mb.si, URL: http://lisa.uni-mb.si/~juric/

Izvleček

Parcialni, segmentirani informacijski sistemi v obliki množice nepovezanih ali delno povezanih aplikacij danes ne morejo zadovoljiti naraščajočih potreb, ki jih povzročajo elektronsko poslovanje in globalna povezljivost na eni ter nova ekonomija na drugi strani. Integracija je ključen pristop, s pomočjo katerega v časovno sprejemljivem okviru povečamo učinkovitost informacijskega sistema podjetja, hkrati pa ohranimo obstoječe rešitve. V prispevku se bomo osredotočili na organizacijsko strateški in tehnološki vidik integracije. Prikazali bomo večfazni postopek integracije, ki temelji na vzorcu integracijskega posrednika in način ovijanja obstoječih sistemov.

Abstract

Partial, segmented information systems, constituted of several non-connected or partially connected applications cannot meet the increasing demands, initiated by electronic commerce and global connectivity on one hand and new economy on the other hand. Integration is the key process, which enables us to increase the enterprise information system efficiency in a reasonable time and allows us to preserve existing (legacy) solutions. In the article we focus on organizational, strategic and technological viewpoints of integration. We present a multiphase integration process, based on the integration broker pattern. We also present a technique for wrapping of existing systems.

■■■

1. Uvod

Sodobne informacijske sisteme zaznamujejo heterogena zasnova, komponentna zgradba, lokacijska neodvisnost in povezljivost. Samostojne, monolitne aplikacije postajajo vedno bolj redke, vedno bolj redki so tudi veliki informacijski sistemi, zgrajeni po specifičnih zahtevah naročnika. Sodobna podjetja kupujejo programsko opremo v obliki komponent, posamezne komponente pa nato integrirajo v celovit informacijski sistem [4, 6]. Ravno integracija delov informacijskega sistema v celoto je eden od najtrših orehov. Za uspešno izvedbo integracije, kot ključnega procesa za izkoriščanje prednosti sodobnih informacijskih sistemov, je potrebno določiti strategijo in izbrati ustrezno tehnološko bazo.

Pomen in vloga integracije delov informacijskega sistema v celoto je razvidna iz projekcije Gartner Group: »Do leta 2004 bo 70% družb, ki so ali bodo oblikovale centralizirano službo za integracijo aplikacij, izvedlo integracijo učinkovito. 70% družb, ki bo prepustilo odgovornost za integracijo posameznim aplikacijskim sektorjem, integracije ne bo uspešno izvedlo (verjetnost 0.8).«

Zaradi tega je integracijski arhitekturi potrebno posvetiti veliko pozornost. V tem prispevku obravnavamo organizacijsko-strateška in tehnološka vprašanja.

2. Organizacija

Iz izsledkov zgoraj navedene napovedi Gartner Group in sorodnih raziskav je razvidno, da sta za podjetje najpomembnejši vpeljava projekta integracije in organizacija centralne službe za integracijo. Takšna služba ima v splošnem dve odgovornosti. Prva je operativna in se navezuje na izbiro, instalacijo in vzdrževanje konsistentne infrastrukture za integracijo, ki je enaka za celotno podjetje. Ta infrastruktura zagotavlja visoko nivojsko integracijo, logično nad lokalnim omrežjem podjetja. Pri tem uporablja eno od vrst ali kombinacijo komunikacijskega vmesnega sloja (middleware) v obliki integracijskega posrednika [7].

Druga odgovornost se veže na razvoj in vzdrževanje dokumentacije za integracijo aplikacij, ki je po navadi sestavljena v obliki množice vmesnikov.

Potrebno je zagotoviti, da posamezne projektne skupine, ki razvijajo aplikacije, upoštevajo izbrano integracijsko paradigmo in v aplikacije vgradijo ustrezno podporo.

Zagotoviti je torej potrebno, da se integracija izvaja na nivoju celotnega podjetja in se vodi in koordinira centralno. Zato naj se vpelje projekt integracije in ustvari ustrezna služba, ki bo razvijala, vpeljevala in vzdrževala integracijsko infrastrukturo.

Tipičen informacijski sistem je sestavljen iz več modelov. Z vidika integracije je ključnega pomena model, ki opisuje relacije z zunanjim svetom. Večina starejših informacijskih sistemov je največjo pozornost posvečala modelom notranje organizacije. Relacije z zunanjim svetom pa so bile pogosto zanemarjene. Informacijski sistemi tudi niso skladni s celovitim objektnim in podatkovnim modelom podjetja. V praksi ni možno modificirati aplikacij tako, da bi zagotovili, da je njihova interna struktura konsistentna. Zato je za uspešno izvedbo integracije potrebno upoštevati omejitve in realno situacijo.

Za uspešno izvedbo integracije sama tehnologija ni zadostna. *Tehnoloških ovir za integracijo heterogenih sistemov danes praktično ni.* Po drugi strani pa imamo lahko velike težave pri integraciji dveh aplikacij na istem operacijskem sistemu, napisanih v istem programskem jeziku z isto podatkovno bazo. Integracija je torej odvisna od arhitekture aplikacije in od organizacije vmesnikov, prek katerih aplikacije komunicirajo z zunanjim svetom. Noben tehnološki standard ne more zaobiti organizacijskih problemov.

Uporaba informacijskih rešitev v obliki sistemov ERP (Enterprise Resource Planning) lahko izboljša, ne more pa ukiniti potreb po integraciji. To velja tudi za podjetja, ki kupijo vse od enega dobavitelja. Po ocenah raziskav, sistemi ERP običajno pokrijejo do 30 odstotkov poslovnih funkcij podjetja. Ostale naloge opravljajo obstoječi sistemi, lastne majhne aplikacije in programska oprema drugih dobaviteljev.

Ker je integracija sistemov ERP zahtevna, dobavitelji teh sistemov dopolnjujejo svoje rešitve z vpeljavo podpore za predvsem komponentne vmesnike. S tem se strinja tudi GartnerGroup, katerega prognoza pravi: »Od 1999 do 2001 bodo vsi glavni dobavitelji ERP ponudili bistveno boljše vmesnike do posrednikov zahtev objektov in sporočilnih sistemov, kar bo omogočilo integracijo z zunanjimi aplikacijami (verjetnost 0.8).«

Ključne aplikacije običajno izkazujejo velike arhitekturne in implementacijske razlike, ki so razvidne v različnih uporabniških vmesnikih, načinu komunikacije, strukturi podatkov in dostopa do podatkovne baze. Pomanjkljivosti glede integracije so:

- heterogenost in neusklajenost uporabniških vmesnikov,
- nepovezanost delov informacijskega sistema,
- prekrivanje dela funkcionalnosti,
- redundantnost podatkov,
- klasična arhitektura odjemalec-strežnik ali celo monolitna zgradba,
- neobstoj standardnih, jasno definiranih vmesnikov za povezljivost.

V pogojih globalizacije tržišča, globalne povezljivosti in dostopnosti informacij in prehoda na elektronsko poslovanje bodo informacijski sistemi morali zagotoviti nove oblike uporabe in se prilagoditi novim zahtevam. Arhitekture, tehnologije in načini vzdrževanja, ki so dobro delovali za klasične, ročno izdelane in nepovezane aplikacije, se izkažejo za neučinkovite in celo kontraproduktivne, če jih apliciramo na današnje heterogene in porazdeljene sisteme.

Le če bo podjetje sposobno hitro prilagoditi obstoječe aplikacije in uporabiti njihovo funkcionalnost na nove načine, bo uspešno delovalo v nastajajoči dobi elektronskega poslovanja. Uspešnost bo odvisna od uporabe novih vzorcev načrtovanja in načinov upravljanja pri zagotavljanju integracije. Izvajanje poslovnih strategij, kot sta »podjetje brez zakasnitve« (Zero Latency Enterprise) ali »premočrtno procesiranje« (Straight Through Processing), lahko zagotovijo le sodobni informacijski sistemi. Strategija »premočrtno procesiranje« zagotavlja, da se isti podatki v informacijski sistem vnašajo samo enkrat. S tem je zagotovljena konsistentnost podatkov in minimirana njihova redundanca. Strategija »podjetje brez zakasnitve« zagotavlja, da se spremembe, narejene v delu informacijskega sistema, takoj (sinhrono), torej brez zakasnitve, odrazijo v ostalih delih celotnega informacijskega sistema, ne glede na njegovo distribuiranost. Običajno ločimo sisteme, ki implementirajo enostransko obveščanje o dogodkih, in sisteme, ki implementirajo dvosmerni kompozitni integracijski vzorec. Tradicionalni občasni paketni prenosi podatkov med aplikacijskimi sistemi in podatkovni bazami ne zadoščajo.

3. Tehnologija

V produkcijskih informacijskih sistemih najdemo danes v številnih primerih paketno komunikacijo z izmenjavo datotek. Taka komunikacija je nedvomno naprednejša od ročnega vnosa podatkov in posledično nesinhroniziranih podatkovnih baz. Kljub temu pa tak način vedno težje vzdrži pritisku naraščajočih poslovnih potreb, ki zahtevajo vedno bolj ažurne podatke [1]. Tako se dogaja, da se paketna komunikacija zaganja večkrat na uro, da bi se zagotovila konsistenca. Poleg tega je direkten prenos v podatkovne baze problematičen, saj zaobide programsko

logiko in poslovna pravila informacijskih sistemov, ki podatke sprejemajo, kar lahko vodi do nekonsistentnosti v podatkih.

Zaradi tega je za integracijo heterogenih informacijskih sistemov uveljavljen *vzorec integracijskega posrednika*. Tak posrednik je osrednji komunikacijski vmesni sloj med aplikacijskimi sistemi ne glede na njihovo tehnološko osnovo. Uporabi omenjenega vzorca govori v prid tudi projekcija Gartner Group: »Do 2001 bo imela več kot polovica velikih podjetij vpeljano eno od oblik integracijskega posrednika (verjetnost 0.8).«

Integracijski posrednik prinaša številne prednosti:

- nudi vnaprej izdelano komunikacijsko infrastrukturo, ki je neodvisna od tehnološke osnove,
- nudi preverjeno in učinkovito podlago za integracijo,
- loči direktno komunikacijo med posameznimi aplikacijami in kardinalnost N-proti-N zmanjša na 1-proti-N,
- omogoča izvedbo integracije z upravljanjem povezav med sistemi na način črne škatle.

Integracijski posredniki delimo na dve skupini, ki se razlikujeta po načinu delovanja in povezave:

- posredniki zahtev objektov (ORB – Object Request Broker) in
- sporočilni sistemi (MOM – Message Oriented Middleware).

Posredniki zahtev objektov so infrastruktura za komunikacijo med porazdeljenimi objekti oziroma komponentami [3]. Zasnovani so kot transportni sistem, na katerem temeljijo sodobni pristopi k gradnji informacijskih sistemov. Posredniki zahtev objektov so osnova za večslojno arhitekturo aplikacij in temelj komponentnih modelov. Ponujajo različne oblike komunikacije – tako sinhrono kot tudi asinhrono komunikacijo med enotami sistema [5]. Zaradi pomena posrednikov zahtev objektov, ki znatno presegajo le okvire integracije aplikacij, smo bili v zadnjih letih priča standardizaciji in danes obstajata dva pomembna standarda:

- Družina posrednikov zahtev objektov arhitekture CORBA (Common Object Request Broker Architecture), ki jo je standardiziral OMG (Object Management Group)in deluje preko protokola IIOP (Internet Inter-ORB Protocol). V to družino prištevamo poleg številnih implementacij arhitekture CORBA tudi porazdeljen objektni model platforme Java 2: RMI oz. RMI-IIOP (Remote Method Invocation).
- Posrednik zahtev objektov Microsoftove arhitekture DCOM/COM+, ki deluje na osnovi protokola ORPC, razširjenega protokola RPC modela OSF DCE (Distributed Computing Environment).

Posredniki zahtev objektov (predvsem družine CORBA) so standardizirani [13, 14]. To je posebej pomembno, saj se z njihovo uporabo odločimo za splošno sprejet industrijski standard in ne za rešitev posameznega podjetja. Poleg tega so omenjeni posredniki zahtev objektov tudi osnova, na kateri delujejo komponentni modeli druge generacije (CCM, EJB in COM+) [4, 8].

Sporočilni sistemi ponujajo le asinhroni način komunikacije med enotami sistema. Delujejo na osnovi sporočilnega kanala. Njihov pomen slabi, saj njihovo funkcionalnost praktično v celoti nudijo tudi posredniki zahtev objektov in v prihodnosti pričakujemo zlitje obeh tehnologij. Prav tako na področju sporočilnih sistemov nismo bili priča standardizaciji. Posledica je, da sistemi različnih proizvajalcev med seboj niso združljivi z vidika programske kode. Pri spremembi sporočilnega sistema moramo spreminjati tudi naše aplikacije. Edini resnejši poskus abstrakcije je bil narejen v Java 2 Enterprise Edition, kjer je definiran univerzalen vmesnik JMS (Java Messaging Service). Omenimo le dva pomembna predstavnika sporočilnih sistemov: IBM MQSeries in MSMQ.

Pomembna razlika med posredniki zahtev objektov in sporočilnimi sistemi je tudi v zmogljivostih. Odzivni časi posrednikov zahtev objektov so praviloma krajši, že zaradi načina komunikacije [10, 11, 12]. Zaradi že omenjenih potreb po podjetju brez zakasnitve in premočrtnemu procesiranju, ki nedvomno favorizirata sinhrono integracijo distribuiranih aplikacij, zaradi omenjenih boljših zmogljivosti ter dejstva, da so posredniki zahtev objektov ključna tehnologija sodobnih pristopov k gradnji informacijskih sistemov, se bomo v nadaljevanju prispevka osredotočili na posrednika zahtev objektov in porazdeljene objektne oziroma komponentne modele [15, 2].

4. Izbira posrednika zahtev objektov

Posredniki zahtev objektov so vrsta programske opreme za vmesni sloj (middleware). Omogočajo transparentno komunikacijo delov informacijskega sistema. Pri tem razvijalcem dopuščajo uporabo že poznane sintakse za komunikacijo med programskimi entitetami. Posredniki zahtev objektov skrijejo vse podrobnosti komunikacije in jo naredijo neodvisno od lokacije, programskih jezikov in orodij, operacijskih sistemov, platform ali omrežnih vmesnikov. Praviloma so sestavni del porazdeljenih objektnih oziroma komponentnih modelov.

Porazdeljeni objektni modeli omogočajo izgradnjo programske opreme z integracijo novo razvitih in ograjenih obstoječih komponent - objektov. Zagotavljajo enotno komunikacijsko infrastrukturo. Z zmožnostjo ponovne uporabe, prenosljivosti in povezljivosti v

distribuiranem, heterogenem omrežju rešujejo večino težav, s katerimi se danes srečujejo razvijalci programske opreme in uporabniki računalnikov. Poleg tega ponujajo množico storitev, kot npr. transakcijske, varnostne in relacijske storitve, storitve življenjskega cikla, trajnega stanja, poimenovanja in druge. Nudijo tudi podporo zagotavljanju razširljivosti.

Omenili smo že, da posrednike zahtev objektov delimo v družino CORBA (ki vključuje tudi Java RMI-IIOP) in Microsoft DCOM/COM+. Obe družini temeljita na podobnih osnovnih konceptih [6, 9], razlike pa obstajajo v številnih podrobnostih. Glede integracijskega posrednika je za omenjene modele najpomembnejše dejstvo podpore različnim operacijskim sistemom in programskim jezikom.

Tabela 1 prikazuje podporo operacijskim sistemom za omenjene porazdeljene objektne modele. Ker implementacije modela CORBA ponujajo različni dobavitelji, so podprti vsi relevantni operacijski sistemi. Povezljivost med različnimi implementacijami modela CORBA in z modelom RMI-IIOP je zagotovljena s podporo protokolu GIOP/IIOP (General Inter-ORB Protocol/Internet Inter-ORB Protocol).

CORBA	Windows (vse verzije, tudi 2000), Mac, Solaris, SunOS, SGI, HPUX, OS/2, SCO, Irix, Digital Unix, AIX, VMS, VxWorks, QNX, MVS, OS/400, pSOS, Cray, Linux[1]
Java RMI-IIOP	Vsi s podporo za JVM (Java Virtual Machine)
DCOM/COM+	Windows NT 4, 2000, 95, 98

Tabela 1: Podpora za operacijske sisteme

Tabela 2 prikazuje podporo za programske jezike.

CORBA C#[2]	C, C++, Java, COBOL, Smalltalk, Ada, PLI,
RMI-IIOP	Java[3]
DCOM/COM+	C, C++, C#, Java, Visual Basic; ostali jeziki preko Automation

Tabela 2: Podpora za programske jezike

Poleg omenjenih je pri izbiri potrebno posvetiti pozornost vsaj naslednjim kriterijem. Zrelost modela, ki se izraža v prisotnosti na tržišču, omogoča sklepanje o njegovi zanesljivosti. Jezikovno neodvisen protokol je pomemben za izdelavo komunikacijske hrbtenice. Zelo pomembna kriterija sta enostavnost učenja in enostavnost razvoja programske opreme in ju velikokrat ni enostavno vnaprej oceniti.

Poleg tega je pri izbiri potrebno pozornost posvetiti tudi temu, kako se arhitektura sklada z deli informacijskega sistema, ki smo ga razvili v hiši in kako arhitekturo podpirajo dobavitelji ostalih delov sistema. Predvsem je to pomembno pri velikih sistemih, kot so sistemi ERP. SAP tako že nekaj časa podpira tako CORBA, kakor tudi DCOM/COM+. Zanemariti ne smemo tudi obstoječe baze znanja zaposlenih.

V nadaljevanju si bomo ogledali načine integracije obstoječih sistemov v enovit informacijski sistem in postopek ovijanja delov informacijskega sistema.

5. Večfazni postopek integracije

Ugotovili smo že, da je v podjetjih običajno del programske opreme kupljen na tržišču, del pa samostojno razvit (ali znotraj podjetja ali skladno s specifikacijami podjetja pri zunanjih razvijalcih). Poleg tega obstajajo lokalne in ad-hoc rešitve. Ključne aplikacije običajno izkazujejo velike arhitekturne in implementacijske razlike, ki so razvidne v različnih uporabniških vmesnikih, načinu komunikacije, strukturi podatkov in v dostopu do podatkovne baze. Prav tako smo ugotovili, da je za integracijo smiselna uporaba integracijskega posrednika.

Predlagani postopek integracije informacijskih sistemov je inkrementalen. Vsebuje tri glavne faze, poimenovane kot:

- konsistentnost podatkov,
- večnivojski proces in
- kompozitni informacijski sistem.

5.1. Konsistentnost podatkov

Cilj prve faze integracije je doseči konsistentnost podatkov na logičnem nivoju. Podatki so redundantni, integracija pa zagotavlja njihovo prenašanje med posameznimi aplikacijami. Aplikacije so neodvisne, povezave se ne zavedajo in imajo ločeno in neodvisno definirane procese. Takšno strukturo prikazuje slika 1.

Značilnosti:

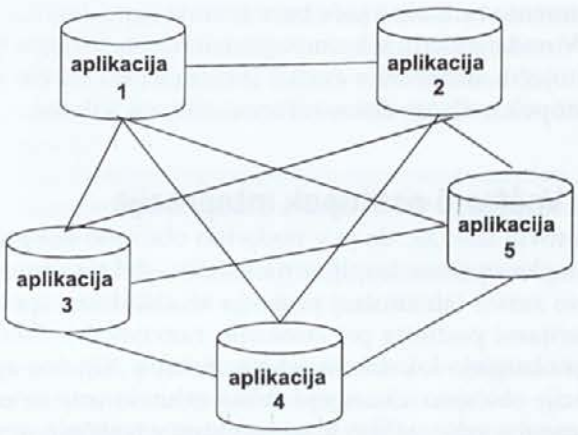
- ločeni poslovni procesi,
- potrebnih je več korakov prenosa,
- komunikacija je enosmerna in asinhrona,
- prenosi so običajno paketni,
- sistemi so fizično neodvisni,
- sistemi so logično povezani, vendar se tega ne zavedajo,
- delna skladnost s strategijo premočrtnega procesiranja.

1 *Našteti so samo najpomembnejši operacijski sistemi. Obstaja tudi podpora za druge, manj pomembne operacijske sisteme.*

2 *Našteti so le jeziki, z katere je podpora standardizirana s strani OMG. Podprti so tudi drugi jeziki, kot npr. Objective C, Perl, Delphi, Visual Basic, Eiffel, Common Lisp, Scheme [13].*

3 *Java vsebuje omejeno podporo za C++ preko vmesnika JNI (Java Native Interface).*

Integracija za dosego konsistentnosti podatkov se implementira s prenosom podatkov med aplikacijami. Pri tem je potrebno poznati podatkovne modele aplikacij in identificirati podatke, ki so skupni dvema ali več aplikacijam. Prek ustreznih API-jev ali protokolov je potrebno dostopiti do izvorne podatkovne baze in ustrezne podatke prenesti v ciljno bazo. Pri tem je potrebno posebno pozornost posvetiti konsistenci podatkov in njihovi skladnosti s poslovnimi pravili, saj poslovna logika aplikacije ne nadzoruje direktnega dostopa do podatkovne baze .



Slika 1: Konsistentnost podatkov na logičnem nivoju

Ključni problem pri takšni integraciji je identifikacija ustreznih podatkov. Slabosti direktnega prenosa med podatkovnimi bazami so predvsem v načinu povezave vsak z vsakim, kar pri večjem številu sistemov privede do kombinatorične eksplozije. Osnovni problem je vzdrževanje in nadziranje velikega števila povezav. Dodatne slabosti so potreba po intervenciji (človeški ali avtomatizirani), paketnemu prenosu in posledičnim zakasnitvam in neažurnosti podatkov. Problem je tudi možnost, da uporabniki modificirajo podatke v samo eni aplikaciji.

Del omenjenih težav rešuje izvedba integracija na osnovi konsistence podatkov z uporabo integracijskega posrednika, v našem primeru posrednika zahtev objektov. Shemo take integracije prikazuje slika 2. Osnovna prednost je definicija vmesnikov do posameznih aplikacijskih sistemov in ločitev neposredne komunikacije med aplikacijami. Način definicije vmesnikov in ovijanja aplikacij si bomo ogledali v nadaljevanju.

S tem se kardinalnost iz N-proti-N zmanjša na 1-proti-N. Komunikacija preko posrednika zahtev objektov je lahko asinhrona, sinhrona ali odložena sinhrona. Če je dogodkovno orientirana, ponuja tudi korak k zagotavljanju strategije podjetja brez zakasnitev.

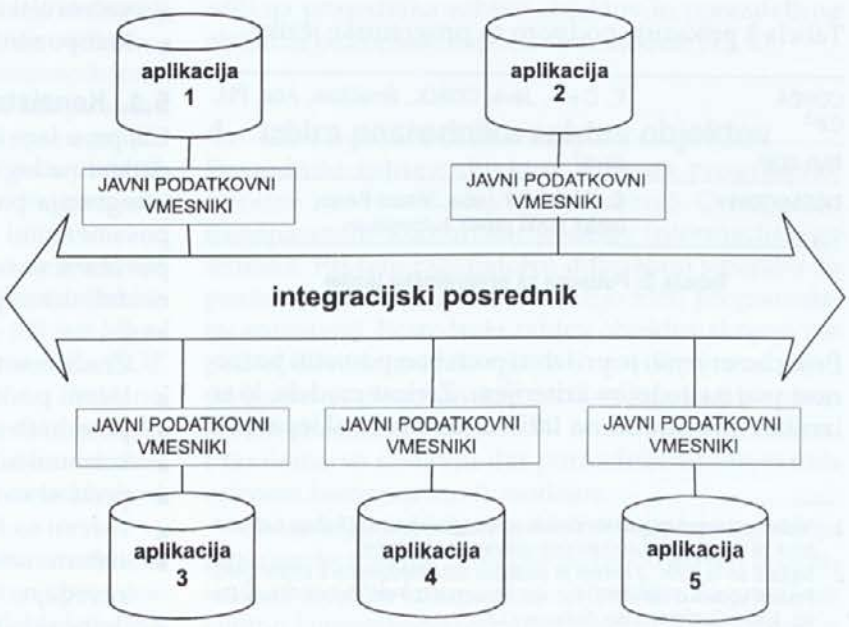
V drugem koraku prve faze izvedbe integracije tako predlagamo, da se za dele informacijskega sistema definirajo javni vmesniki, preko katerih bo možna sinhronizacija podatkov. Sam prenos se naj izvede s pomočjo posrednika zahtev podatkov. Ta način pred direktnim prenosom podatkov med aplikacijami nudi naslednje prednosti:

- oblikovati se začne infrastruktura za globalno integracijo aplikacij,
- začne se z definicijo javnih vmesnikov do podatkovne in poslovne logike aplikacij,
- olajša se nadzor in vzdrževanje komunikacije,
- komunikacija lahko temelji na dogodkih, ki jih prožijo aplikacije.

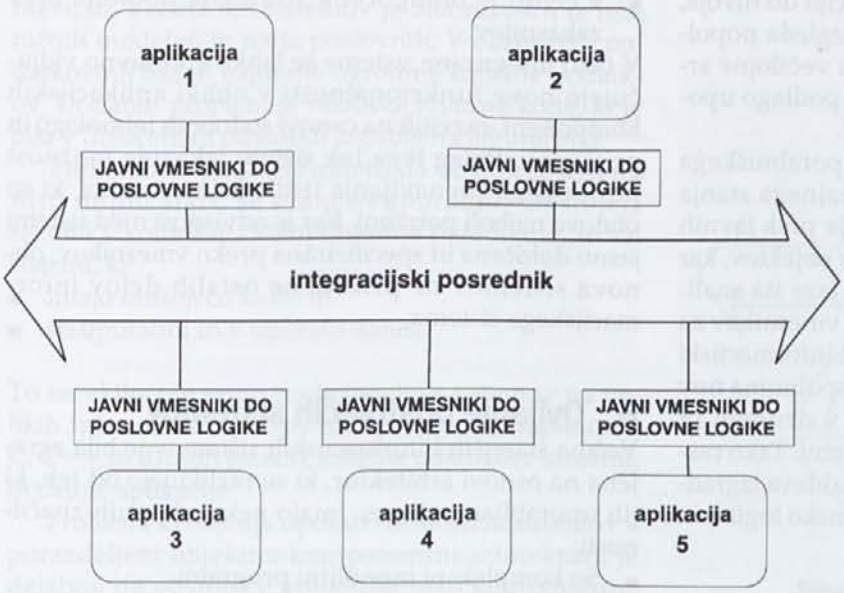
Razume se, da proces integracije ne zahteva popolne implementacije prvega koraka, ampak se lahko direktno pristopi k implementaciji arhitekture, predstavljene v drugem koraku.

5.2. Večnivojski proces

Druga faza integracije informacijskih sistemov se imenuje večnivojski proces, ki je prikazana na sliki 3. Večnivojski proces zahteva izdelavo globalnega modela načrtovanja informacijskega sistema. Tak model temelji na proučitvi scenarijev uporabe obstoječih aplikacij informacijskega sistema, na izdelavi primerov uporabe in na definiciji modela analize. Model



Slika 2: Konsistentnost podatkov z uporabo integracijskega posrednika



Slika 3: Večnivojski proces integracije

analize in analiza obstoječih aplikacij sta predpogoj za izdelavo modela načrtovanja ali globalnega objektnega modela informacijskega sistema. Pri tem je pomembno zagotoviti, da se upošteva vsa funkcionalnost obstoječih sistemov, hkrati pa se predvidijo manjkajoče funkcionalnosti, dopolnitve in spremembe. Globalni objektni model je najpomembnejše izhodišče za uspešno izvedbo druge faze integracije.

Slika 4: Oris aktivnosti za drugo fazo integracije

Na osnovi globalnega objektnega modela se definirajo javni vmesniki do obstoječih in do novo razvitih aplikacij. Tako definirani vmesniki nam skupaj z izbiro in uporabo ustrezne tehnologije omogočajo izdelavo objektnih ovojev okrog obstoječih aplikacij. S tem omogočimo dostop do poslovne logike obstoječih in novih aplikacij na enoten, univerzalen in transparenten način, ki je neodvisen od lokacije, operacijskega sistema, programskega jezika in strojne osnove. Diagram aktivnosti prikazuje slika 4.

Pri tem je potrebno poudariti, da je istočasno potrebno v proces razvoja programske opreme znotraj podjetja vnesti ustrezne spremembe in zagotoviti, da bo lastno razvita programska oprema upoštevala načela integracije in bo skladna z globalnim objektnim modelom. Zaradi tega predlagamo, da se za lastne projekte čim prej predvidijo načini in možnosti integracije, da se definirajo in implementirajo ustrezni vmesniki in se predvidi načrt podpore ustrezne integracijske tehnologije, v tem primeru posrednikov zahtev objektov. Za obstoječe kupljene komercialne sisteme je potrebno najti načine, kako s čim manj stroški čim bolj učinkovito zagotoviti vmesnike do poslovne logike.

Integracija na osnovi večnivojskega procesa zagotavlja:

- enoten poslovni proces,
- komunikacija je obojestranska, vendar običajno asinhrona,
- prenosi se lahko izvajajo v trenutku (ali po potrebi paketno),
- sistemi so fizično povezani,
- sistemi so logično povezani,
- delno je omogočena strategija premočrtnega procesiranja,
- delno je omogočena strategija podjetja brez zakasnitev.

Slabosti:

- neenoten uporabniški vmesnik,
- še vedno je potrebnih več korakov za realizacijo integracije,
- komunikacija še vedno poteka po paradigmi odjemalec-strežnik.

5.3. Kompozitni informacijski sistem

Kljub številnim prednostim druga faza integracije ni zagotovila enotnega in celovitega informacijskega sistema. Šele tretja faza integracije, imenovana kompozitni

informacijski sistem, zagotavlja integracijo do nivoja, ko informacijski sistem za uporabnika zgleda popolnoma enoten. Ta faza temelji na vzorcu večslojne arhitekture informacijskih sistemov in za podlago uporablja rezultate prejšnjih faz integracije.

Osnovni princip je stroga ločitev uporabniškega vmesnika, poslovne logike in nivoja trajnega stanja podatkov ter konsistentne komunikacije prek javnih vmesnikov na osnovi posrednika zahtev objektov, kar prikazuje slika 5. Osnovni dejavnosti te faze sta analiza in načrtovanje enotnih uporabniških vmesnikov za celovit informacijski sistem. Tako razvit informacijski sistem bo za uporabnike izgledal kot popolnoma nov sistem. V resnici pa bodo v ozadju, v drugem in tretjem sloju, delovali oviti obstoječi sistemi. Tako razvit kompozitni informacijski sistem ne zahteva izgradnje iz nič, ampak uporablja vso programsko logiko in obstoječe znanje, hkrati pa zagotavlja:

- enoten poslovni proces,
- popolno integracijo aplikacij, ki navzven delujejo kot ena velika aplikacija,
- enoten uporabniški vmesnik,
- dvosmerno, sinhrono komunikacijo,
- neposredne, takojšnje in individualne prenose podatkov,
- fizično odvisnost med sistemi,
- logično odvisnost med sistemi,
- v celoti je omogočena strategija premočrtnega procesiranja,
- v celoti je omogočena strategija podjetja brez zakasnitev.

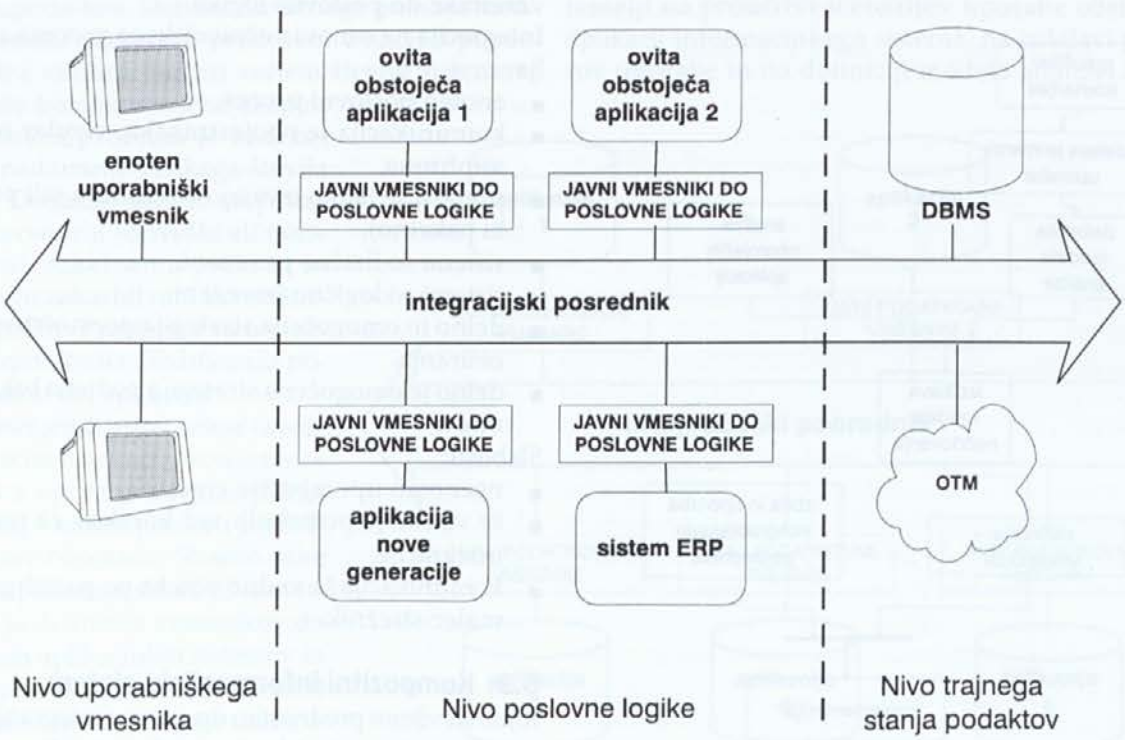
V tako integrirane sisteme se lahko enostavno vključujejo nove funkcionalnosti v obliki aplikacijskih komponent, razvitih na osnovi sodobnih tehnologij in pristopov. Poleg tega tak sistem izkazuje možnost postopnega prenavljanja tistih delov sistema, ki so obnove najbolj potrebni. Ker je odvisnost med sistemi jasno določena in specificirana preko vmesnikov, obnova sistemov ne prizadene ostalih delov informacijskega sistema.

6. Ovijanje obstoječih sistemov

Večina starejših informacijskih sistemov je bila zgrajena na osnovi arhitektur, ki se razlikujejo od teh, ki jih uporabljamo danes. Imajo nekaj skupnih značilnosti:

- So kompleksni monolitni programi.
- So ključni za opravljanje določenih nalog.
- Odvisni so od področja.
- Uporabljajo tradicionalne podatkovne baze ali pa še tega ne.
- Ponavadi delujejo na srednjih in velikih računalnikih in imajo centralizirano osebje, ki skrbi za razvoj, vzdrževanje in podporo.

Obstoječi sistemi so bili razviti s ciljem služiti posameznim poslovnim področjem, ne pa celemu podjetju. Tako imajo omejen doseg in se niso zmožni



Slika 5: Kompozitni informacijski sistem

razvijati. Večina teh sistemov je bila razvitih iz tehničnih modelov in ne iz poslovnih. V obstoječih podatkovnih bazah najdemo ogromne količine podatkov. Dodaten problem je neobstoj izvorne kode, ki je bila v določenem odstotku izgubljena ali uničena.

Pomembno pa je razumeti, da obstoječi sistemi niso nujno stari, še manj neuporabni. Obstoječi sistemi v kontekstu komponentne paradigme, so vsi sistemi, ki:

- imajo obstoječo kodo in
- so uporabni in v uporabi danes.

To ne vključuje samo tradicionalnih sistemov na velikih računalnikih pač pa tudi programe, napisane v C, C++ in drugih jezikih, sisteme odjemalec-strežnik in druge aplikacije.

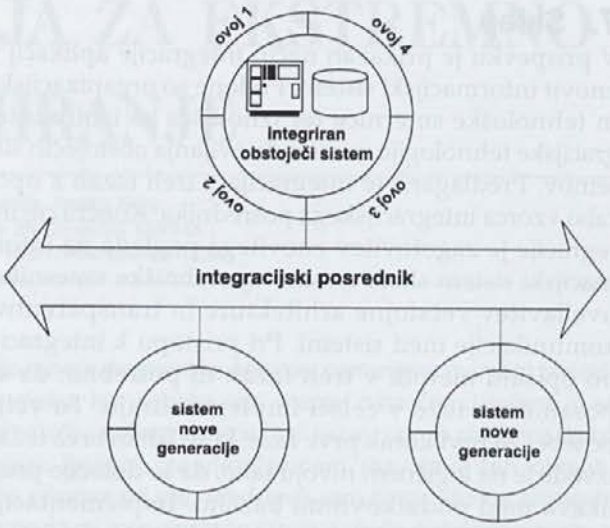
Problem, ki otežuje uporabo obstoječih sistemov v porazdeljeni objektno-komponentni arhitekturi, je dejstvo, da obstoječe aplikacije niso komponente. Izziv je najti način, kako ograditi obstoječe aplikacije tako, da bodo izgledale kot porazdeljene komponente in se bodo operacije ene komponente lahko implementirale v več aplikacijah.

Cilj integracije obstoječih sistemov je torej uporaba njihovih informacij in kode pri razvoju arhitekture informacijske podpore za podjetje. Želimo si, da bi obstoječi sistemi lahko komunicirali z drugimi obstoječimi in novo razvitimi sistemi (slika 6). Potrebujemo torej informacijsko arhitekturo, ki bi omogočila sodelovanje obstoječih sistemov in jih razbila v komponente, poslovne procese pa v ločena, kooperativna področja.

Takšno ločevanje v področja zahteva razumevanje vsebine sistemov in primerov njihove uporabe in implementacijo abstraktnih vmesnikov, ki omogočajo drugim področjem dostop do vsebine in primerov uporabe.

Integracija skrije obstoječe sisteme za konsistentnimi vmesniki, ki skrivajo implementacijske podrobnosti, so v skladu s poslovnim procesom in slovarjem in omogočajo spreminjanje ali zamenjavo implementacij brez vpliva na preostanek sistema. Abstraktni vmesnik ograjuje in skriva dejansko implementacijo sistema. Odjemalci dostopajo v sistem samo skozi ta vmesnik.

Ovijanje *(wrapping)* obstoječega sistema je proces, v katerem definiramo in omogočimo dostop do obstoječega sistema skozi abstraktni vmesnik. Na eni strani ovoj predstavlja sistem z abstraktnim vmesnikom. Na drugi strani ovoj komunicira z obstoječim sistemom,

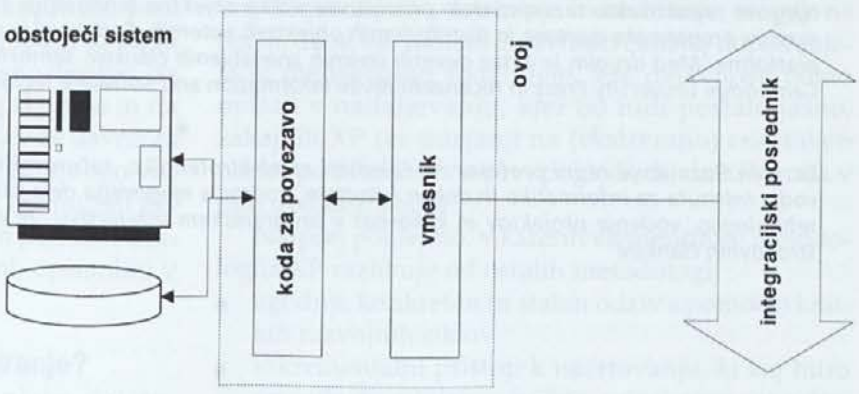


Slika 6: Integracija obstoječih sistemov

kot prikazuje slika 7. Ovoji predstavljajo sistemski pogled, neodvisen od implementacije. So naraven način za integracijo obstoječih sistemov drugega z drugim in z novimi sistemi. Enkrat ovit, se lahko obstoječi sistem vključi v porazdeljeno objektno-komponentno okolje z uporabo posrednikov zahtev objektov.

Na Univerzi v Mariboru, Inštitutu za informatiko, Centru za objektno tehnologijo imamo izkušnje z integracijo obstoječih sistemov. Definirali smo tehniko, ki natančno določa potrebne korake pri izdelavi ovoja in opozarja na področja, ki jim je potrebno posvetiti dodatno pozornost. Tehniko smo uspešno preizkusili pri ovijanju več realnih aplikacij [5, 7].

Čez čas lahko funkcionalnost obstoječih aplikacij postopoma zamenjamo s komponentami, dokler prvotna koda ne postane odvečna. Izbrane dele lahko obnavljamo korak za korakom, glede na potrebe in razpoložljiva sredstva. Tak pristop zmanjša stroške, poveča zmogljivosti in olajša vzdrževanje.



Slika 7: Ovijanje obstoječega sistema

7. Sklep

V prispevku je prikazan način integracije aplikacij v enovit informacijski sistem. Podane so organizacijske in tehnološke smernice ter izhodišča za izbiro integracijske tehnologije in metode ovijanja obstoječih sistemov. Predlagana je integracija v treh fazah z uporabo vzorca integracijskega posrednika. Končni cilj integracije je zagotovitev enovitega pogleda na informacijski sistem skozi enotne uporabniške vmesnike, uveljavitev večslojne arhitekture in transparentne komunikacije med sistemi. Pri pristopu k integraciji po opisani metodi v treh fazah ni potrebno, da se posamezne faze v celoti implementirajo. To velja posebej za prvi korak prve faze, ki se lahko brez težav izvede le na logičnem nivoju tako, da se določijo preslikave med podatkovnimi bazami. Implementacija prvega koraka ni potrebna in se lahko takoj pristopi k drugemu koraku prve faze. Končni cilj je izdelati arhitekturo, skladno s shemo, opisano v tretji fazi. Tako integriran informacijski sistem zagotavlja premočrtno procesiranje in distribucijo podatkov brez zakasnitev. Na ta način bo informacijski sistem prilagojen vstopu v globalno elektronsko ekonomijo informacijske družbe.

Viri

[1] Donald E. Rimel Jr., Ronald C. Aronica, *Leveraging Legacy Assets*, First Class, Dec 1996, OMG

[2] Juric B. Matjaz, Zivkovic Ales, Rozman Ivan, *Are Distributed Objects Fast Enough*, More Java Gems, Cambridge University Press, 2000

[3] Jurič M. B., Heričko M., Rozman I., *CORBA - objektni model porazdeljenega procesiranja*, Uporabna informatika, jan/feb/mar 1997

[4] Jurič B. Matjaž, *Nova generacija komponentnih modelov CORBA 3, COM+, EJB*, Objektna tehnologija v Sloveniji OTS'2000, 2000, str. 18-29

[5] Jurič B. Matjaž, Rozman Ivan, Heričko Marjan, *A method for integrating legacy systems within distributed object architecture*, International Conference on Enterprise Information Systems, Setúbal, Portugal, March 1999

[6] Jurič B. Matjaž, Rozman Ivan, Heričko Marjan, *CORBA komponente in pomen objektnih transakcijskih storitev*, Dnevi slovenske informatike, DSI 2000, del. 1, str. 52-60

[7] Jurič B. Matjaž, Rozman Ivan, Heričko Marjan, *Integrating legacy systems in distributed object architecture*, ACM Software Engineering Notes, March 2000, vol. 25, no. 2, str. 35-39

[8] Jurič B. Matjaž, Rozman Ivan, Heričko Marjan, *Performance comparison of CORBA and RMI*, Information and Software Technology Journal, 2000, vol. 42, no. 13, str. 915-933

[9] Jurič B. Matjaž, Rozman Ivan, *Java 2 RMI and IDL comparison*, Java Report, February 2000, vol. 5, no. 2, str. 36-48

[10] Jurič B. Matjaž, Rozman Ivan, Stevens Alan, Heričko Marjan, Nash Simon, *Java 2 distributed object models performance analysis, comparison and optimization*, International Conference on Parallel and Distributed Systems, California, IEEE Computer Society Press, 2000, str. 239-246

[11] Jurič B. Matjaž, *The efficiency of distributed object models*, OOPSLA'99, ACM, New York, 1999

[12] Jurič B. Matjaž, Rozman Ivan, Heričko Marjan, Domajnko Tomaž, *CORBA, RMI and RMI-IIOP performance analysis and optimization*, World Multiconference on Systemics, Cybernetics and Informatics, Orlando, Florida, Vol. 8, Part 2, 2000, str. 582-587, vabljeno predavanje

[13] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.4, 2000

[14] Soley R. M., Ph.D. (1995), Object Management Architecture Guide, OMG, John Wiley & Sons, Inc., Revision 3.0, Third Edition

[15] Vinoski Steve, *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments*, IEEE Communications Magazine, Vol. 14, No. 2, February, 1997

◆

Dr. Matjaž B. Jurič je docent na Inštitutu za informatiko Fakultete za elektrotehniko, računalništvo in informatiko v Mariboru. Njegovo raziskovalno razvojno delo pokriva vse vidike objektne tehnologije s posebnim poudarkom na komponentnem razvoju programske opreme in distribuiranih objektnih sistemih. Sodeloval je pri razvoju RMI-IIOP, sestavnega dela Java 2 platforme. Med drugim je avtor devetih izvirnih znanstvenih člankov, samostojnega poglavja v knjigi, izdani pri založbi Cambridge University Press in recenzent revije Information and Software Technology Journal.

◆

Dr. Ivan Rozman je redni profesor na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Je vodja Inštituta za informatiko in dekan fakultete. Področja njegovega dela vključujejo programsko inženirstvo, objektno tehnologijo, vodenje projektov in kakovost v programskem inženirstvu. Je avtor več knjig in številnih znanstvenih ter strokovnih člankov.

OBLIKOVANJE OKOLJA ZA EKSTREMNO PROGRAMIRANJE

Matevž Rostaher, Ivan Slamek, Andrej Kline
OdaTeam d.o.o., Meljska cesta 36, SI-2000 Maribor
e-mail: info@odateam.com, URL: http://www.odateam.com

Izvleček

Ekstremno programiranje je zamišljeno kot lahka metodologija razvoja programske opreme, namenjena manjšim projektnim skupinam, ki programsko podpirajo poslovne procese v okolju, kjer potrebe niso vnaprej natančno določene in se nenehno spreminjajo. Lahko rečemo, da je tovrstnih razvojnih okolij v našem prostoru kar nekaj, zato je disciplina, ki bo predstavljena v pričujočem prispevku, namenjena prav ljudem, soudeležencem razvoja v teh okoljih. Avtorji nimamo namena vsiliti lastnih pogledov in načina dela nikomur od vas. Predstavili vam bomo ključne elemente ekstremnega programiranja in vas seznanili z načinom, kako smo ga mi sprejeli v naše razvojno okolje, kako nas je njegovo spoznavanje spremenilo v obdobju enega leta od vpeljave in kakšne so naše izkušnje ob uporabi novega načina razvoja.

Abstract

Extreme programming (XP) is a lightweight methodology for developing software with main target in smaller project groups that are obliged to develop in an environment where requirements are not fully specified in advance and/or are changing constantly. Nowadays, most software projects have to deal with changing requirements, therefore the following paper is meant for everybody involved in such a developing process. The authors do not want to enforce their own views of working style to anybody. They just present the basic elements of XP and explain the way of adopting it into the development environment; how they have changed their thinking and the way they live since beginning to implement the discipline two years ago.

■ ■ ■

1. Ekstremno programiranje (XP) kot disciplina

Okrajšava XP in problem prevoda.

Najprej glede poimenovanja. Ekstremno programiranje je dobeseden prevod izvornega pojma "Extreme programming", ki v angleškem govornem področju označuje metodologijo, predstavljeno v nadaljevanju. Slovenski prevod ni enostaven, vsaj ne takšen, ki bi bil izražen z enim samim pojmom ali besedno frazo. Ob našem srečanju z ekstremnim programiranjem smo se oprijeli izraza skupinsko programiranje (skupinski proces razvoja, ipd.), ki pa ni najbolj posrečen niti ustrezen, saj je jasno dejstvo, da se programska oprema razvija znotraj skupine in da je doba t.i. «one man band« programerjev že davno za nami. Ekstremno programiranje obsega mnogo več kot le skupinsko programiranje. Zato imenujmo ekstremno programiranje preprosto XP in pod tem izrazom razumimo disciplino z lastnostmi, opisanimi v nadaljevanju.

Kaj je in kaj ni ekstremno programiranje?

XP je nizko-obremenjen, učinkovit, nizko-rizičen, predvidljiv, znanstven in svež način razvoja programske opreme, ki predlaga (ne narekuje!) nekoliko drugačne smernice v vseh fazah razvoja. Ne uvaja nobenih revolucionarnih novosti, vendar pa obravnava vsa sredstva, potrebna v razvojnih projektih (delovni prostor, ljudi, metodologija razvoja, način vodenja in upravljanja) na svojstven, precej drugačen, za nekatere celo čuden, ekstremen, način. Z obrazložitvijo, da XP skuša vsakega od svojih sestavnih elementov privesti do ekstrema, pojasnimo tudi uporabo besede « extreme » v izvornem izrazu. XP skuša do skrajnosti poenostaviti načrtovanje, razvoj, komunikacijo, testiranje in kodiranje, ampak na način, da se vse naštete dejavnosti čimbolj dopolnjujejo in medsebojno sodelujejo. Vse dejavnosti bomo opisali v nadaljevanju, kjer bo tudi postalo jasno, zakaj jih XP res udejanji na (ekstremno) enostaven način, pa četudi se na prvi pogled zdi, da XP vnaša v njih dodaten napor.

Najprej poglejmo, v katerih elementih se metodologija XP razlikuje od ostalih metodologij :

- zgodnji, konkreten in stalen odziv s pomočjo kratkih razvojnih ciklov
- inkrementalni pristop k načrtovanju, ki kaj hitro privede do celotnega načrta, pričakovanega v življenjski dobi projekta

- prilagodljivo načrtovanje implementiranja funkcionalnosti, ki prožno reagira na spremembe poslovnih potreb
- avtomatski testi, ki jih pišejo programerji in stranke za opazovanje napredka v razvoju, omogočajo sistemu, da se razvija in da zgodaj prestreže napake
- zanaša se na ustno komunikacijo, teste in na programsko kodo, ki komunicira (pojasnjuje tudi drugim programerjem) strukturo in namen
- zanaša se na evolucijski proces načrtovanja, ki traja tako dolgo, kot živi sistem
- zanaša se na tesno sodelovanje programerjev, ki imajo zgolj običajne sposobnosti
- upošteva tako kratkoročni instinkt programerjev kot dolgoročni interes projekta (drugače povedano, popolnoma podpira posebnost in inovativnost posameznih programerjev, ampak le v skladu z dobronamernostjo za razvoj celotnega projekta!).

Discipliniranost XP

se kaže v nekaterih, sicer redkih principih, od katerih metodologija ne odstopa. Npr. testiranje. Kdor ne piše testov, ali bolje, kdor ne piše avtomatskih testov, ne udejanja XP! Polovičarstva ni. XP ima pravilo: Ali si zraven ali nisi.

Ali je potrebna še ena nova disciplina?

XP v osnovi ne vnaša nobenih novih prvin. Tehnike programiranja, ki jih uporablja, so znane že desetletja. Načini vodenja ljudi, ki jih postavlja v ospredje, pa celo že stoletja. Kaj je torej novega pri tej disciplini?

Razvoj programske opreme je povezan z riziki, kot so: zamik planiranih datumov izdaje programja; ukinitev projekta; stroški popravkov izdanega sistema so tako veliki, da je sistem potrebno zamenjati; odstotek pojavljanja napak v sistemu je zelo visok; programska oprema ne ustreza poslovnim zahtevam; poslovne zahteve se med implementacijo spremenijo; izdaja programske opreme, ki uporabniku ne prinaša vrednosti; programerji, ki se naveličajo projekta v agoniji, zavrnejo sodelovanje. XP se dotika teh problemov na vseh nivojih razvoja: teži h kratkim iteracijam projekta, h kratkim ciklom izdaje; uporabnike se vpraša za čim manjšo smiselno izdajo, ki bodisi postopno gradi sistem ali pa povzroči malo škode, če je implementacija neustrezna; kreira in vzdržuje obširno zbirko testov, najbolje za testiranje celotnega sistema, ki se izvajajo ob vsaki spremembi, tudi večkrat na dan; testiranje vseh implementiranih enot opravljata oba programerja v paru, testiranje funkcionalnosti pa le za to določeni programerji; najbolj nujna funkcionalnost se implementira najprej, s čimer se rešimo nevarnosti po implementiranju neuporabnega sistema; stranke so sestavni del integracije sistema, s čimer se specifikacije sistema redefinirajo vzporedno z novimi zahtevami uporabnikov.

Vidimo torej, da je XP zelo pozitivno naravnana disciplina, ki v osnovi teži k uspešnosti projektov, ki vztrajno dodajajo poslu vrednost in ki se kontrolirano razvijajo!

2. Ekonomski vidik vpeljave XP razmišljanja v razvoj programske opreme

Štiri spremenljivke.

XP kontrolira potek projektov s pomočjo štirih spremenljivk:

- stroškov,
- časa,
- kakovosti in
- obsega.

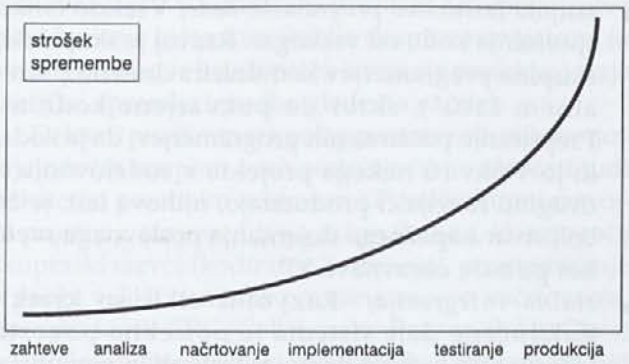
Igra razvoja programske opreme se igra po pravilu, da zunanji vplivi projekta (poslovne stranke, uprava) izberejo tri od štirih spremenljivk. Razvojna skupina pa kot rezultat izbere četrto spremenljivko.

Ponavadi uprava in poslovne stranke določijo prve tri spremenljivke; s kakšnimi vložki želimo v določenem časovnem obdobju doseči kakšno kakovost sistema. Nikakor ne smejo uprava ali stranke določati obsega sistema (kode). To je izključno v domeni razvijalcev. Nasploh XP strogo ločuje poslovne odločitve od odločitev tehnične narave. Prve naj sprejema poslovni svet (poslovne stranke), slednje pa razvojna skupina.

Poleg tega kakovost sistema praviloma ni spremenljivka, vsaj na dolgi rok ne. V zgodnjih iteracijah sicer XP svetuje načelo "naredi najenostavnejšo možno stvar, ki lahko deluje", ki ne zagotavlja popolne funkcionalnosti sistema. Ta bo dodana z naslednjimi iteracijami, ob nenehnem testiranju, ob posvetovanjih s poslovnimi strankami, kontrolirano! Kakovost sistema pa je konstantna, torej edina spremenljivka, ki je XP ne daje na izbiro. Kakovost naj bo skozi vse faze razvoja sistema največja možna.

Strošek spremembe.

Ena splošnih ugotovitev in celo predvidevanj programskega inženirstva je, da stroški sprememb v programu naraščajo eksponentno s časom (Slika 1). V zadnjih desetletjih si razvoj programske opreme, gledano kot skupnost ali kot tendenca, prizadeva zmanjšati ta strošek – z jeziki nove generacije, z boljšimi podatkovnimi bazami, z boljšo navado programiranja, z boljšimi razvojnimi okolji in orodji, z novimi notacijami. Vpeljava navedenih novosti je vsekakor dobrodošla, že za boljše počutje in ugodnejše delo razvijalcev, vendar izpostavljenega problema ne reši. Zanimivo, vendar žal resnično.



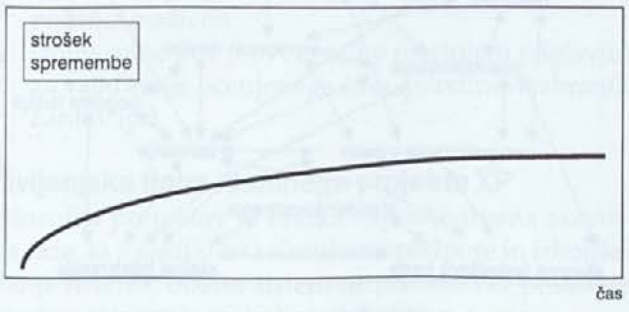
Slika 1: Strošek spremembe raste eksponentno s časom

Ena od tehničnih premis XP pa je ravno zmanjšanje rasti tega stroška, ki naj se s časom približa neki asimptotični vrednosti (Slika 2). Tako nam že sama predpostavka, da stroški sprememb v sistemu s časom ne postanejo enormni, omogoča razmišljati drugače. Velike odločitve o nadaljnji usmeritvi sistema (tudi projekta) lahko prenesemo v kasnejši čas, ko bodo zahteve popolnejše in jasnejše. Prav tako je pri slednji rasti možno načrtovati strošek spremembe, ki bo ob morebitni spremembi potreben v prihodnosti.

Tako kot XP teži k vzpostavitvi okolja, v katerem je strošek sprememb s časom relativno majhen, nikakor pa ne eksponentno velik, tako si tudi obratno discipline XP brez predpostavke, da so spremembe poceni, ne moremo zamisliti. Kako bi sicer razvijali pod načelom "naredi najenostavnejšo stvar, ki lahko deluje" zdaj, nadaljnja funkcionalnost pa bo dodana kasneje, ko bo to neznansko drago!?

Zadrževanja nizke vrednosti stroška sprememb seveda ni možno doseči kar tako. Potreben je skupek premišljenih tehnologij in razvojnih navad, ki jih prav XP skuša zbrati pod okriljem svoje discipline.

Na tehnološki strani so objekti gotovo ključna tehnologija. S sporočili, ki jih izmenjujejo in metodami, ki se izvajajo v ozadju, so pripravljeni na majhne, obvladljive spremembe, ki nimajo učinka na celoten sistem. Nadalje lahko sklepamo na tehnologijo podatkovnih baz, kjer so tiste objektno zasnovane v prednosti glede sprememb, saj je podatkom pripeta tudi koda.



Slika 2: Strošek spremembe s časom ne zraste dramatično

Nikakor pa ne trdimo, da je objektna tehnologija, katere privrženci smo, pogoj za izvajanje XP. XP se v osnovi naslanja na naslednje elemente:

- preprosto načrtovanje, brez kakršnihkoli idej o poslovnih potrebah, ki še niso uporabljane, ampak bi se utegnile pokazati kot koristne v prihodnosti
- avtomatizirani testi, ki nam vlijejo zaupanje pri spreminjanju sistema; da ne naredimo neke spremembe sistema v strahu, kaj nam bo povzročila na netestiranih delih sistema
- ogromno prakse pri spreminjanju in načrtovanju; da se ne bojimo spreminjati sistema.

3. Okolje, v katerem lahko XP zaživi

Spoznali smo že namen in ključne elemente discipline. Zdaj lahko strnemo spoznanja v glavnih vrednotah in principih, ki jih zagovarja XP.

Štiri vrednote:

- komunikacija,
- preprostost,
- odzivnost in
- pogum.

Pri komunikaciji so zajete vse udeležene strani: razvijalec-razvijalec, razvijalec-stranka, razvijalec-vodstvo... Preprostost smo že večkrat omenili. Je ena najtežje dosegljivih vrednot za razvijalca XP. Namreč pri razvoju ni enostavno odmisliti vseh zahtev, ki so trenutno res nepotrebne. Odzivnost je sposobnost strank seznaniti se z implementiranim delom sistema in reagirati nanj. XP teži k predajanju manjših enot sistema čimprej v uporabo, da bodo tudi morebitne napake čimprej ugotovljene. Pogum pa XP obravnava kot vrednoto z največjo težo, saj le osebje, ki si drzne začeti in spreminjati sistem, lahko živi in razvija v skladu s filozofijo discipline.

Vrednote se ponovno prepletajo, npr. pogum razvijalca je tem večji čim boljša je komunikacija z drugimi razvijalci, pa tudi z vodstvom. Razvijalec, ki je močno kaznovan za vsako manjšo napako, se sprememb boji, pa tudi nove stvari razvija z občutkom strahu pred grajo.

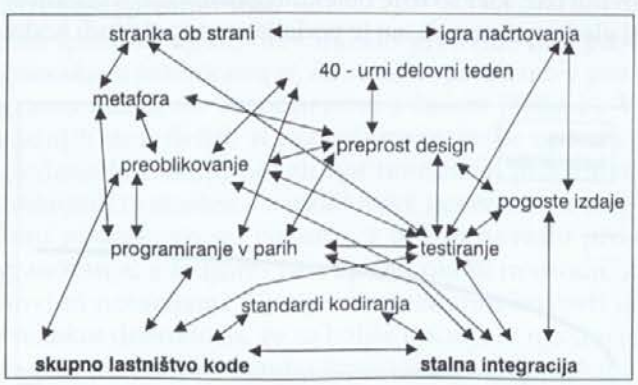
Principi XP:

Kot vsaka druga metodologija, tudi XP v končni fazi sloni na skupku principov. Teh je kar nekaj: igra načrtovanja, majhne in pogoste izdaje, metafora, preprost dizajn, testiranje, preoblikovanje, programiranje v parih, skupno lastništvo kode, stalna integracija, 40-urni delovni teden, stranka zmeraj prisotna in standardi kodiranja. Seveda se medsebojno povezujejo (Slika3), večinoma samostojno sploh ne morejo nastopati. Vsekakor pa, še preden na sliki dobite vtis, kako kompleksen in zamršen je XP, pojasnimo pomen principov in njihovo medsebojno sodelovanje.

- *igra načrtovanja*; Razvoj programske opreme je zmeraj razvijajoči se dialog obeh strani (poslovne in tehnične) glede tega, kaj je možno in kaj zaželeno. Poslovni ljudje odločajo o obsegu razvoja sistema, o prioriteti razvoja posameznih delov, o komponiranju izdaj (pod)sistema ter o datumih teh izdaj. Seveda poslovne odločitve ne morejo biti sprejete brez ustreznih tehničnih posvetovanj in odločitev glede: ocenjenega časa, potrebnega za implementacijo posameznih delov sistema; posledic, ki jih prinašajo poslovne odločitve (npr. izbira baze podatkov); načina organiziranosti razvojne skupine; podroben urnik razvoja.
- *majhne in pogoste izdaje*; Vsaka izdaja naj bo tako majhna, kot je le možno, da še zmeraj predstavlja zaključeno celoto. V poslovni svet naj prinese implementacijo najbolj vrednih zahtev (tistih z najvišjo prioriteto). Pogostost izdaj naj bo vsakih nekaj tednov.
- *metafora;* Metafore služijo za skupno poimenovanje tehničnih elementov sistema in povezav med njimi, v smislu nedvoumne komunikacije med vsemi osebami, vpletenimi v razvoj sistema. Npr. pojem zavarovalna pogodba nedvoumno določa abstrahiran element zavarovalniškega poznavanja, pa čeprav je to za uporabnike sistema viden kot en vnos v brskalniku, na katerega lahko klikajo in dobijo ustrezne povezave, za razvijalca pa predstavlja točno določen tip objekta z atributi... Metafore v XP nadomeščajo t.i. arhitekturo sistema, največkrat poznano kot "velike škatle s povezavami".
- *preprost design*; Pravilen design v vsakem trenutku razvoja ustreza naslednjim aktivnostim:
  1. zagon vseh testov
  2. preverjanje podvojenosti logike
  3. označevanje vseh namer ali idej, pomembnih za programerje
  4. uporaba čim manj razredov in metod.
- *testiranje*; XP prakticira funkcijske teste in teste enote. Oba načina testiranja sta bila podrobno predstavljena v prispevku [12]. Seveda je med razvojem ključnega pomena oblikovanje in vzdrževanje avtomatskih testov, katerih uporabo smo že omenili. Na tržišču so prisotna odlična testna ogrodja, tako za Javo kot Smalltalk: JUnit, SUnit.
- *preoblikovanje*; Vedno, ko programer naleti na situacijo, da mu preoblikovanje obstoječe kode olajša nadaljnji razvoj novih zahtev, naj uporabi tehnike preoblikovanja. Tudi v ta namen so na tržišču orodja za podporo, npr. "Refactoring browser" za Smalltalk. Podrobneje bo ta princip opisan v prispevku [11].
- *programiranje v parih*; Princip, po katerem vsak zlog nove kode ustvarjata oba programerja, združena v razvojni par. Tudi testiranje poteka v dvoje. Podrobneje je bil ta princip opisan v prispevku [13].
- *skupno lastništvo programske kode*; Vsakdo lahko spreminja kodo od vsakogar. Razvoj je skupinski, skupina programerjev komunicira dnevno ("v realnem času"), skrbi da pokvarjene kode ni. Prepričanje posameznih programerjev, da je koda, ki jo v okviru nekega projekta v sodelovanju z drugimi razvijalci producirajo, njihova last, je že bolj stvar napačnega dojemanja poslovnega sveta kot pa naše obravnave XP.
- *stalna integracija;* Razvojni cikli so kratki. Zaključene dele sistema je potrebno pogosto (včasih tudi večkrat dnevno) testirati in povezati v delujočo celoto (še preden se jo preda uporabniku kot naslednjo izdajo).
- *40-urni delovni teden*; Projektno zasnovano delo zahteva seveda velike napore vseh sodelujočih. Zato XP že ob načrtovanju časa razvoja kot svoje spremenljivke računa z 8 urami dela dnevno. Na kratki rok se sicer da delati tudi več ur dnevno, vendar sodelujoči v projektu že po nekaj tednih postanejo demotivirani za delo.
- *stranka ob strani*; Prisotnost akterjev iz poslovnega sveta, za katerega se sistem razvija, smo že omenili kot nenadomestljivo pridobitev v smislu izboljševanja zahtev, hitrega odziva, zmanjšanja rizika napačno razvitega sistema ipd.
- *standardi kodiranja;* Pravila morajo biti jasna vsem programerjem, da lahko ti komunicirajo kar z izvorno kodo, ne pa s tekstovnimi opisi. Zelo pomembno je poznavanje vzorcev in njihova dosledna uporaba.

Razvojno okolje = delovni prostor + razvojna skupina + razvojno orodje

Disciplina XP je namenjena skupinskemu razvoju do 10 razvijalcev. Sodelujočih poslovnih strank je lahko seveda več, vendar ne hkrati pri enem projektu več kot deset, šteto skupaj z razvijalci. Z večanjem števila razvijalcev principi XP izgubljajo svojo veljavo. Naj še enkrat poudarimo, da v sklopu XP vse osebe sodelujejo pri načrtovanju, kodiranju in testiranju (tehnično osebje in



Slika 3: Principi XP podpirajo drug drugega

poslovni svet). V kadrovski strukturi XP vpeljuje dve novi funkciji: inštruktor (vodja skupine razvijalcev) in sledilec (oseba, ki beleži potek izvajanja projekta; izvaja metrike, spremlja izvedbo glede na načrte).

Delovni prostor mora poleg posameznih zahtev razvijalcev (in nasploh vseh sodelujočih) zadoščati tudi nekaterim posebnim zahtevam XP: velik skupni razvojni prostor z razporeditvijo računalnikov, ki omogoča skupinski razvoj (kodiranje, testiranje), programiranje v parih; velika namizna površina za igro načrtovanja, uporabo kartic CRC (Collaborator Responsibility Class) – [13]; ločen integracijski računalnik. Vedno mora biti na razpolago dovolj hrane in pijače, ugodna pa je tudi možnost počitka ali celo spanja.

Razvojno orodje naj bo izbrano v skladu z metodologijo in prepričanjem razvijalcev. Vsekakor je pomembno, da nobenega segmenta orodja (CASE orodij, podatkovne baze, testnih ogrodij, ...) vodstvo ne vsili razvojni skupini, ampak da skupina argumentira vodstvu izbiro orodij in mu svetuje pri tovrstnih odločitvah.

4. Psihološki vidik vpeljave XP

Uvajanje tovrstne discipline vsekakor spremeni obnašanje in mišljenje marsikaterega poslovnega subjekta, ki sodeluje v razvoju. Poglejmo si še s te plati zahteve XP.

Ločitev tehnične in poslovne odgovornosti.

Tehnično osebje mora biti osredotočeno na probleme tehnične narave. Projekt mora biti voden s poslovnimi odločitvami, katerih osnova pa morajo biti tehnične odločitve glede ocene stroškov in rizika.

Poslovni svet naj izbere:
- obseg ali čas posameznih izdaj
- relativno prioriteto predlaganih potreb
- natančen obseg predlaganih potreb.

Razvojni oddelek pa mora prispevati:
- ocenjen čas, potreben za implementacijo različnih funkcionalnosti
- oceno posledic uporabe različnih tehničnih disciplin
- izbiro razvojnega procesa, ki kar najbolje ustreza osebnemu prepričanju razvijalcev in je v skladu s politiko podjetja
- izbiro principov (navedenih v prejšnjem poglavju) za validiranje ocenjenega časa, pravilnosti zbranih zahtev ipd.

Življenjska doba idealnega projekta XP

Filozofija projektov je kratka vzpostavitvena razvojna faza, ki ji sledijo leta simultane podpore in izboljševanja sistema, dokler sistem ne prinaša več poslovne vrednosti in ga je najbolje upokojiti.

Naj na tem mestu omenimo še vlogo vodstva kot izvrševalca tudi ukinitve projekta, če se ta izkaže za zgrešenega (kljub uporabi XP!).

Pravilo 20-80,

ki ga razvijalci programske opreme razumejo po načelu "80% koristi prinese 20% razvoja", XP obrne malo drugače, in pravi: producirajmo najprej 20% najbolj vredne (vredne za stranke) funkcionalnosti in jo dajmo takoj v uporabo; ostalih 80% funkcionalnosti ima za stranke nižjo prioriteto in jih zato lahko implementirano skozi kasnejše izboljševanje sistema.

Kaj bremeni XP?

Vsekakor je možnih več dejavnikov. Najbolj banalen je odpor do same ideje. Ampak pustimo ta primer ob strani.

Kot glavno breme lahko izpostavimo čustva, zlasti strah. Če se nekdo počuti pod pritiskom, ker mora povezovati vse spoznane principe XP ali pa mu je bilo dodeljenih preveč opravil (znotraj posamezne izdaje), potem dela pod pritiskom. V tem primeru se osebno počuti slabo in dela napake. Takšno delo mu je vse prej kot v veselje.

Kot nadaljnje težave lahko navedemo še zmožnost razvijanja kompleksnih sistemov na enostaven način, kot ga zagovarja XP. Težko je nekaterim posameznikom priznati neznanje ali nezadovoljivo znanje na določenem področju razvoja. Prav tako je težko podreti čustvene ovire med razvijalci ali celo znotraj njih.

Skoraj največjo nevarnost pri taki organiziranosti dela pa predstavlja dejstvo, da ima lahko majhen problem velike posledice.

Potrebne so izkušnje in pogum!!!

Kdo naj ne udejanja XP?

Disciplina nima nekih ostrih mej uporabe. Vendar pa lahko najdemo čiste primere, ko je bolje ne poskusiti z njenim udejanjanjem:
- velike razvojne skupine
- nezaupljive stranke
- uporaba tehnologije, ki ne podpira majhnih in pogostih sprememb sistema (v takšnih sistemih je vsaka pozna sprememba namreč strahotno draga)
- okolja, kjer je povratna informacija od strank počasna (reda nekaj mesecev)
- delovno okolje ni primerno; za neprimernost zadostuje že razdelitev razvoja v 2 etaži.

5. Odateam xp-erti pri delu

V preostalih poglavjih vam bomo predstavili način, na katerega smo privzeli disciplino XP v našem podjetju. Kako smo implementirali posamezne principe, kako se je s tem spremenilo naše delo in kaj smo s tem pridobili.

Delovno okolje.

S prehodom na način življenja XP smo si pred dobrim letom dni najprej poiskali nove poslovne prostore. Naš cilj je bil velik skupen prostor, kjer ustvarjajo vsi člani razvojne skupine ves čas skupaj. Delovne računalnike smo po nekajkratnem poizkušanju razporedili tako, da je delo razvojne skupine najbolj komunikativno. Našo začetno postavitev prikazuje Slika 4a, ugotovljeno optimalno razporeditev pa Slika 4b. Delo je podrejeno programiranju in testiranju v spreminjajočih se parih.

Povsem smo ločili privatno sfero «dela» od poslovnih aktivnosti. Delovni računalniki so namenjeni izključno razvoju in komunikaciji s poslovnimi strankami. Za lastno uporabo ima vsak od razvijalcev privatni, prenosni računalnik, ki mu omogoča opravljanje vseh privatnih aktivnosti bodisi doma bodisi v poslovnih prostorih, AMPAK izven prostora razvoja.

V poslovnih prostorih je še velika miza za modeliranje z uporabo kartic CRC, za posvetovanja pri načrtovanju in za pogovore s poslovnimi strankami. Spet so privatne mize ločene od te skupne, « razvojne delovne ploskve ». Stene imamo polepljene s t.i. nalogami (orig. « task ») in pravili, ki se jih pri delu držimo.

Naj na koncu opisa prostora še omenimo, da je nujno potrebna seveda tudi hrana in pijača! To je edina stvar, ki nastopa tako na privatnih kot na razvojnih mizah.

Delo razvojne skupine.

Poglavitno vlogo pri načrtovanju novih funkcij sistema ali pri preoblikovanju obstoječih delov sistema imajo naše poslovne stranke, konkretno osebe z natančnim poznavanjem zavarovalništva. Poslovne in tehnične odločitve so vidno ločene, v skladu z opisom, podanim v poglavju 4. Vodstvo postavlja grobe časovne načrte, razvojna skupina pa jih – po končanem načrtovanju in spoznavanju zahtev – precizira in postavi naloge razvoja.

Design

Za modeliranje uporabljamo kartice CRC (« Collaborator Responsibility Class »). Sodelujejo vsi razvijalci, najpogosteje skupaj s predstavnikom poslovnega sveta. Ugotovitve (ustvarjene odgovornosti in sodelovanja) se prenesejo v implementacijo. Z uporabo kartic vsem razvijalcem postane sfera, ki jo abstrahirajo, popolnoma jasna. V primeru, da kartice ne zadostujejo, se razvijalci lotimo testov. Ti zagotovo pojasnijo še vsako morebitno nejasnost, saj če znaš napisati test za del sistema, tudi razumeš, kakšne so njegove zahteve in kaj producira.

Ob igranju kartic si postavimo tudi naloge, za katere je posamezni razvijalec odgovoren (v 4. poglavju omenjena « accepted responsibility »), sestavimo tudi skupen seznam potreb po oblikovanju ali preoblikovanju (t.i. « ToDo List »). Ocenimo število idealnih programerskih dni, potrebnih za izpeljavo projekta ter z upoštevanjem obremenilnega faktorja (ki zajema vse dejavnike, ki nas pri razvoju motijo: vzdrževanje, študij, ...) ocenimo dejanski čas izpeljave projekta.

Testiranje

Tako kot programiranje tudi testiranje poteka v parih. Testiranje obsega teste enote (« Unit tests »), ki jih izvajamo razvijalci za vsako razvito enoto sistema – pogosto pa tudi pred samo implementacijo – ter funkcijske teste (« Functional tests »), ki jih napišejo sodelujoči iz poslovnega sveta, da validiramo razviti sistem.

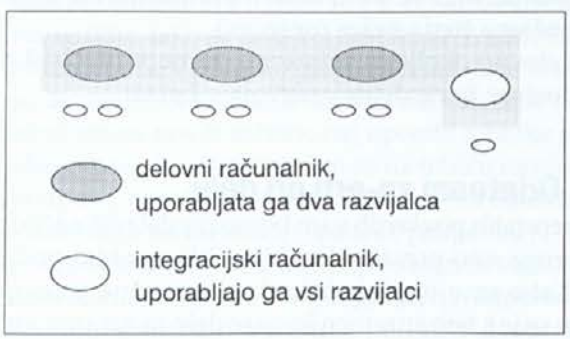
Pri testiranju si pomagamo z orodji SUnit in JUnit, predstavljenimi v lanskoletni predstavitvi [12].

Kodiranje

Kodiranje poteka zmeraj v parih, dva programerja za enim računalnikom. Dosledno udejanjamo načela, opisana v tretjem poglavju : uporaba vzorcev, metafor, pravil kodiranja. Koda mora biti tista, ki komunicira.

Proces kodiranja zahteva spremljanje izvajanja zastavljenih nalog, spoznanih bodisi v času načrtovanja bodisi v predhodnih fazah kodiranja. V ta namen imamo osebo, imenovano sledilec (tracker).

Integracija kode poteka pogosto, tudi večkrat dnevno.



Slika 4a: Prvotna zamisel razporeditve delovnih računalnikov

Slika 4b: Optimalna razporeditev delovnih računalnikov

Vzdrževanje
je dejansko lažje, če sistem pokaže v svojem delovanju manj napak. Logično. Če stremimo k zagotavljanju kakovosti sistema skozi ves razvoj in udejanjamo položno krivuljo strošek/čas, je vzdrževanja manj in je cenejše. Pa smo spet pri enem od "ekstremov".

6. Sadovi uporabe XP

Po dobrem letu dni udejanjanja discipline imamo v celotnem poslovanju (ne samo znotraj razvojne skupine) zelo pozitivne izkušnje.

Resda smo se morali privaditi na precej specifičen način dela. Morda je še največ preglavic razvijalcem povzročal princip ločitve privatnih aktivnosti od poslovnega dela. Naš način dela, izhajajoč iz objektne usmerjenosti, se ni kaj dosti spremenil. Principi XP so nam že dalj časa znani. Morda se jih nismo toliko zavedali ali jih tako dosledno uporabljali.

Pridobitve so naslednje :

- Homogena razvojna skupina, ki tesno sodeluje z vodstvom in s strankami.
- Prijateljski odnosi krepijo spoštovanje in odnos pri delu; odnos do sodelavcev, do strank.
- Uvajanje novih sodelavcev je enostavno, saj se skoraj takoj lahko vključijo v delo, seveda kot par izkušenejšemu razvijalcu.
- Razvoj je hitrejši, saj je komunikacija med razvijalci ter komunikacija razvijalec-stranka ena poglavitnih vrednot discipline.
- Avtomatski testi omogočajo razvoj z večjo samozavestjo, ni bojazni pred spreminjanjem sistema.
- Poslovne stranke so bolj zadovoljne, saj aktivno sodelujejo pri načrtovanju in nam s tem posredujejo svoje zahteve in želje po izboljšavah.
- Razvojni cikli so kratki, sistem se razvija v manjših, kontroliranih etapah.
- Naše vodstvo in razvojna skupina skupaj popolnoma zaupamo disciplini. Privadili smo se na novi način dela. Rezultati nas spodbujajo k nadaljnjemu razvijanju tega načina dela, izpopolnjevanju posameznih načel, k prilagajanju načel za naše potrebe in tudi k prilagajanju nas samih tem načelom.

Slabosti:
Težave so seveda – kot pri drugih panogah – psihološke narave. Če se razvijalci ne strinjajo z načeli discipline, je bolje, da je ne udejanjajo.

Zdaj, ko ste prebrali ta prispevek, imate dve možnosti; XP lahko vzljubite ali pa se vam bo vse skupaj zdelo nesmiselno. Če stavite na prvo možnost, vam v referencah podajamo izvrstno izhodišče za globlje spoznavanje discipline. Če se vam zdi vaš dosedanji način dela prikladnejši, pa ste dobili svež pogled na razvoj programske opreme.

REFERENCE

1. BECK Kent, "Extreme programming explained", Addison Wesley, 1999
2. BECK Kent, "Guide to Better Smalltalk", Cambridge University Press, 1999
3. BECK Kent, "Smalltalk Best Practice Patterns", Prentice Hall, 1997
4. GABRIEL Richard P., "Patterns of Software", Oxford University Press, 1996
5. GAMMA Erich et.al., "Design Patterns", Addison-Wesley, 1995
6. WIRFS-BROCK Rebecca et.al., "Designing Object-Oriented Software", Prentice-Hall, 1990
7. BROWN William J. et.al., "Anti Patterns", John Wiley & Sons, 1998
8. ROBERTS Don et.al., "Why Every Smalltalker Should Use the Refactoring Browser", The Smalltalk Report, letnik 6, številka 10, september 1997
9. WILKINSON Nancy M, "Using CRC Cards", SIGS Books, 1995
10. C3 Team, Chrysler Corp., "Case Study, Chrysler Goes to Extremes", Distributed Computing, oktober 1998
11. DeMARCO Tom, "Peopleware", Dorset House 1999
12. GRAJFONER Uroš, "Ponovna uporaba (Refactoring)", Zbornik srečanja Objektna tehnologija v Sloveniji 2000, junij 2000
13. GRAJFONER Uroš, REPINC Peter, "Testiranje OO sistemov", Zbornik srečanja Objektna tehnologija v Sloveniji 99, junij 1999
14. ROSTAHER Matevž, KLINE Andrej "Proces skupinskega razvoja programske opreme (Extreme programming)", Zbornik srečanja Objektna tehnologija v Sloveniji 99, junij 1999
15. http://www.c2.com/cgi/wiki?ExtremeProgramming, Extreme Programming Discussion
16. http://www.xprogramming.com, XP Magazine
17. Mailing lista: *extremeprogramming@egroups.com*
18. http://www.extremeprogramming.org

Matevž Rostaher je študiral telematiko na Tehnični univerzi v Grazu. Študent računalništva in informatike na Univerzi v Mariboru in od leta 1992 zunanji sodelavec podjetja OdaTeam d.o.o., kjer se je ukvarjal z analizo, načrtovanjem in s programiranjem informacijskih sistemov z uporabo objektne metodologije in razvojnega orodja Visual Smalltalk in Java. Od leta 2000 projektni vodja v podjetju FJA OdaTeam d.o.o. Sodeloval je v pri uvedbi ekstremnega programiranja v proces razvoja informacijskih sistemov.

Ivan Slamek, študent računalništva in informatike na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru. Dela kot razvijalec informacijskih sistemov v podjetju FJA OdaTeam d.o.o. Sodeloval je pri razvojnih projektih v Smalltalku in Javi ter pri uvajanju prvin ekstremnega programiranja v proces razvoja informacijskih sistemov.

Andrej Kline je diplomiral na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru, kjer je zagovarjal diplomsko nalogo »Uporaba Jave pri zagotovitvi varnosti oddaljene povezave«. Zdaj je študent podiplomskega študija na tej fakulteti in razvijalec informacijskih sistemov v podjetju FJA OdaTeam d.o.o. Sodeloval je pri razvojnih projektih v Smalltalku in Javi in pri uvajanju prvin ekstremnega programiranja v proces razvoja informacijskih sistemov.

OBJEKTNA TEHNOLOGIJA V SISTEMIH NADZORA IN VODENJA PRODUKTNE LINIJE SI2000 VERZIJE 5

Ana Robnik, Igor Šalamun
IskraTEL, d.o.o., Kranj
e-pošta: robnik@iskratel.si, salamun@iskratel.si
URL: http://www.iskratel.si

Izvleček

Iskratel je podjetje z bogato tradicijo na področju telekomunikacij, katere rezultat je lastni izdelek SI2000. Strežniki za nadzor in vodenje omrežja ter storitev so vitalna nadgradnja sistema upravljanja vozlišč iz programa produktov SI2000 verzije 5. Predstavljajo funkcionalno in komunikacijsko vez med produktno linijo vozlišč SI2000 in EWSD ter poslovno ravnjo in aplikacijami drugih tujih proizvajalcev. Opisani koncept vodenja omrežja in storitev, uporaba objektne in komponentne tehnologije ter arhitekture porazdeljenih objektov vpeljujejo visoko stopnjo modularnosti pri gradnji celotnega koncepta TMN (Telecommunications Management Network), ki ga je standardizirala mednarodna organizacija ITU-T (International Telecommunication Union - Telecommunication Standardization Sector). Aplikativni nivo, realiziran v porazdeljeni objektni tehnologiji, omogoča enostavno medsebojno funkcionalno povezovanje aplikacij in gradnje različnih grafičnih vmesnikov.

Abstract

Iskratel is a company with long tradition in the development of telecommunications equipment, whereof the result is own product - the SI2000 system. Servers for supervision and management of network and services are a vital upgrade of the Management Node from the SI2000, version 5 product line. They make a functional and communication link between SI2000 and EWSD nodes and business layer as well as 3-rd party applications. The concept of network and service management as described herein, the use of object and component technologies along with distributed objects architecture, introduces a high degree of modularity into the construction of a complete TMN (Telecommunications Management Network) concept, standardized by the International Telecommunication Union - Telecommunication Standardization Sector (ITU-T), in line with world trends in the field. The applications layer, implemented in distributed object technology, allows simple functional interoperability among applications, as well as creation of various graphical interfaces.

■■■

1 UVOD

Svetovno komunikacijsko infrastrukturo tvorita v prvi vrsti klasično telefonsko omrežje in internet. Po napovedih strokovnjakov bo internet, ki je po mnenju mnogih srce in gonilo informacijske dobe, prevladujoče omrežje v naslednjem desetletnem obdobju. Svetovni tokovi na področju medsebojnega povezovanja različnih omrežij in predvsem storitev v njih težijo k odprtim rešitvam in njihovi vsesplošni povezljivosti. Gladki in elegantni prehodi med vodovnimi in paketnimi omrežji, govornimi in podatkovnimi omrežji, brezžičnimi in fiksnimi dostopi silijo k celovitim standardiziranim rešitvam na danem področju. Zlivanje omrežij pomeni pravo revolucijo na področju vodenja in nadzora omrežja in storitev. Bogatejši nabor storitev in vse obsežnejši podatkovni in multimedijski promet ne zahtevajo le vse večje pasovne širine ter hitrih, varnih, zanesljivih in kakovostnih hrbteničnih omrežij in prehodov med njimi, ampak narekujejo tudi nove tehnološke pristope in uvajanje novih paradigem v razvoj programske opreme.

Poleg zgoraj omenjenega zlivanja različnih tipov omrežij in storitev v njih, dobiva vse izrazitejši pomen možnost elegantnega vključevanja v omrežje na različnih fizičnih mestih na zemeljski obli, s trenutno dostopnostjo do vseh podatkov, ki so vitalnega pomena za nadaljnje delo uporabnika in pridobivanje informacij. Mobilnost uporabnika na eni strani ter

prenosljivost in dostopnost njegovih osebnih nastavitev in konfiguracije virov na drugi strani, so s stališča uvajanja dodatnih, povsem novih tehnoloških rešitev, svojevrsten izziv.

Kako se na dane svetovne tokove pripravljamo v našem podjetju, kakšne izkušnje smo pridobili pri uvajanju novih konceptualnih shem razvoja programske opreme? Kaj želimo poleg že prisotnega dodatno uvesti v sistem in tako čimbolj zadostiti novim smernicam v razvoju telekomunikacijskih omrežij, storitev, mobilnosti in upoštevanju osebne note uporabnika pri svojem delu? Na dani vprašanji bomo poskušali podati odgovore v nadaljevanju.

2 UPRAVLJANJE VOZLIŠČ DRUŽINE SI2000

Iskratel je podjetje z več kot petdesetletno tradicijo na področju telekomunikacij in skoraj tridesetletnim razvojem lastnega izdelka, ki so ga v zgodnjih sedemdesetih letih poimenovali sistem SI2000. Sistem je zasnovan na programirljivih vozliščih, ki omogočajo enostavno načrtovanje in gradnjo telekomunikacijskih omrežij. Sistem SI2000 omogoča komunikacije v javnih omrežjih, velikih korporativnih omrežjih, samostojnih naročniških centralah (vozlišča PBX) in dostopovnih omrežjih. Na voljo je več različnih konfiguracij in sicer od manjšega dostopovnega vozlišča, na katerega je priključenih nekaj sto naročnikov, do velikega komutacijskega sistema z nekaj deset tisoč naročniki in nekaj tisoč prenosniki z različnimi vmesniki in signalizacijami, ki omogočajo povezovanje vozlišč med seboj ter priključevanje različnih naprav na naša vozlišča.

Sistem SI2000 zagotavlja osnovne in ISDN storitve, centreksne storitve, brezvrvični dostop z lokalno mobilnostjo (sistem DECT), računalniško podprto telefonijo (CTI) in veliko izbiro storitev PBX. Različni standardni dostopovni in omrežni vmesniki ter sistemi signalizacije po skupnem (CCS) ali pridruženem kanalu (CAS) omogočajo vključitev sistema v kakršnokoli javno ali korporativno omrežje.

Dosedanji razvoj sistemov nadzora in vodenja tako telekomunikacijskih kot tudi podatkovnih omrežij je pokazal, da imajo vsi proizvajalci opreme lastne rešitve sistemov za upravljanje omrežja in ne predvidevajo bistvenega poenotenja med ravnjo elementov omrežja in ravnjo upravljavcev mrežnih elementov. Ponavadi oboji tvorijo enovit sistem, katerega komunikacija temelji na različnih standardnih in internih protokolih. Glede na obstoječe rešitve upravljanja ne moremo tudi v prihodnje pričakovati poenotenja že na prehodu med nivojem elementov omrežja in upravljalnim elementom. Odpiranje telekomunikacijskega omrežja posameznega proizvajalca s stališča vodenja in nadzora je tako možno in nujno na ravni upravljalnega elementa, ki mora sistemom iz višjih ravni - to je omrežne, storitvene in poslovne ravni - omogočati realizacijo želenih funkcionalnosti.

Upravljani element predstavlja osnovno informacijsko entiteto sistema, na kateri bomo gradili celovit sistem upravljanja in nadzora. Smer razvoja upravljalnih vozlišč je v zadnjih letih narekovala centralizirano upravljanje mrežnih elementov posameznega proizvajalca.

V primerih, ko operaterji uporabljajo opremo samo enega dobavitelja, lahko njihove centralizirane sisteme upravljanja uporabimo na ravni upravljanja omrežja in storitve. Seveda morajo takšni sistemi vsebovati vse potrebne funkcionalnosti. V praksi so takšni primeri bolj izjema kot pravilo. Praviloma operaterji uporabljajo telekomunikacijsko opremo različnih proizvajalcev, kar vodi do uporabe dveh ali več centraliziranih sistemov upravljanja.

Informacijsko povezovanje centraliziranega sistema z celovitim sistemom upravljanja je bolj domena načrtovanja programske opreme kot iskanja standardnih protokolov za povezovanje. Pri izvedbi je vse bolj pomemben arhitekturni koncept sistema, ki omogoča dinamično povezovanje različnih aplikacij v smislu proženja akcij in prenosa podatkov.

Z novo verzijo komutacijskih (SN: Switch Node) in dostopovnih vozlišč (AN: Access Node) iz družine produktov SI2000 verzija 5 se je podjetje Iskratel odločilo za nov arhitekturni koncept upravljalcev elementov omrežja. Upravljalno vozlišče (MN: Management Node) in upravljalni terminal (MT: Management Terminal) sta z gledišča celotnega arhitekturnega koncepta vodenja telekomunikacijskih omrežij (TMN: Telecommunication Management Network) obravnavana in uvrščena na raven upravljavca elementov omrežja, pri čemer so elementi omrežja bodisi komutacijska, dostopovna vozlišča bodisi napajalni sistemi (PSN: Power-Supply Node). Upravljalno vozlišče omogoča nadzor in upravljanje hibridnih sistemov, to je poljubne kombinacije vozlišč različnih verzij, ki so med seboj v skladnem delovanju.

Uvedba arhitekturnega koncepta vodenja ter nadzora omrežja in storitev v družino produktov SI2000 V5 pomeni za Iskratel, d.o.o., Kranj naslednji korak pri razvoju programske opreme. Programska oprema upravljalnega vozlišča tako postaja funkcionalno vse bolj odprta, združljiva in nadgradljiva s programsko opremo omrežne, storitvene in poslovne ravni. Povezljivost med programskimi opremami različnih proizvajalcev je bodisi vertikalna bodisi horizontalna in ponuja celovito rešitev na danem področju.

Na sejmu TELECOM'99 so vsi večji proizvajalci telekomunikacijske opreme predstavili svojo vizijo celovitega sistema upravljanja. Ugotovitve lahko strnemo v nekaj točkah:

- komercialnih celovitih rešitev še ni na trgu
- pilotni projekti temeljijo na centraliziranemu sistemu upravljanja posameznega proizvajalca
- ohranjene so dosedanje rešitve pri komunikaciji med elementi omrežja in elementi upravljanja
- elementi upravljanja se odpirajo navzven s porazdeljenimi objekti
- realizacija omejene funkcionalnosti dobave in zagotavljanja storitev za končnega uporabnika in operaterje na segmentu skrbi za končnega kupca.

Vidimo, da vsi proizvajalci dopuščajo možnost povezovanja preko "standarda" porazdeljenih objektov. To je tudi razvidno iz rezultatov del, ki jih opravljajo različna standardizacijska telesa na različnih delih sveta. Poraja se vprašanje: kaj so porazdeljeni objekti in kaj omogočajo? Ali dana nova paradigma resnično daje celovite odgovore na tako kompleksno postavljene zahteve in visoka pričakovanja?

3 PORAZDELJENI OBJEKTI

Koncept TMN je kompleksen celovit sistem vodenja telekomunikacijskega omrežja. Sestavljen je iz različnih strojnih in programskih platform. Želja je, da bi lahko vse sodelujoče aplikacije sodelovale preko proženja različnih akcij in izmenjave podatkov. Zaradi objektne orientacije aplikacij se rešitev kaže v porazdeljenih objektih, ki omogočajo veliko fleksibilnost pri medsebojni integraciji.

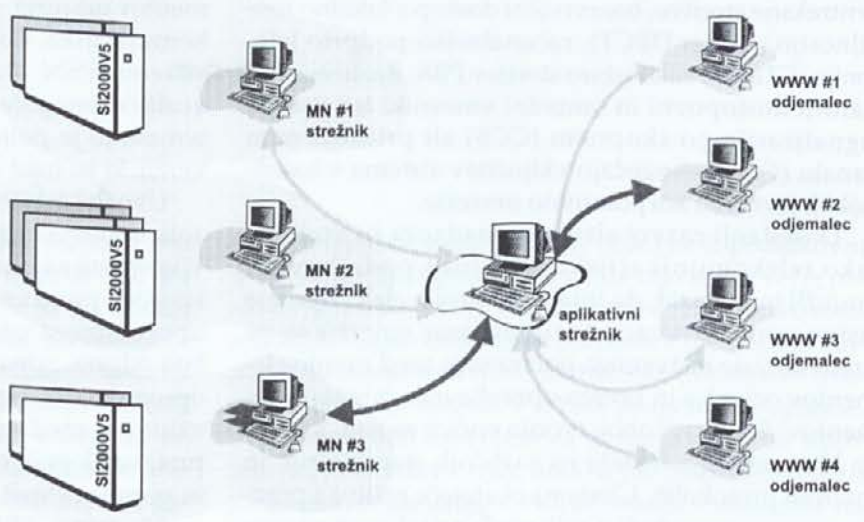
Do sedaj so se uveljavile predvsem tri izvedbe: CORBA (Common Object Request Broker Architecture), RMI (Remote Method Invocation) in DCOM (Distributed Component Object Model). Prva izvedba (CORBA) je najbolj neodvisna in splošna ter omogoča visoko stopnjo uporabnosti. Druga izvedba (RMI) je domena Jave, vendar se že kaže tendenca po povezljivosti s CORBA objekti. Zadnja izvedba (DCOM) je rešitev podjetja Microsoft, ki pa že tudi kaže namere po povezovanju s prevladujočo tehnologijo CORBA.

Moč porazdeljenih objektov je v možnosti po izvajanju na različnih računalnikih, povezanih v mrežo. Različne samostojne aplikacije si delijo porazdeljene objekte, uporabljajo njihove metode in lastnosti. Preko metod in lastnosti izmenjujejo podatke in prožijo želene akcije. Aplikacija upravljanja mrežnega elementa interno uporablja nabor objektov z določenimi funkcionalnostmi. Lastnosti takšnih objektov lahko "objavi" v zbirki IDL (Interface Definition Language) in omogoči, da jih uporabljajo tudi druge aplikacije. Aplikacija, ki skrbi za naročnika, statično ali dinamično integrira v sebi klic želenega objekta in s tem privzame njegovo funkcionalnost. Nakazana arhitektura zahteva razslojevanje običajnega debelega odjemalca v upravljalnem vozlišču.

4 TRISLOJNA ARHITEKTURA

Potrebe po hranjenju vse večje količine informacij so vpeljale v sisteme upravljanja samostojne podatkovne strežnike in enega ali več odjemalcev. Podatkovni strežniki skrbijo za hranjenje informacije in standardni način komunikacije med podatkovnim strežnikom in odjemalcem. Dvoslojna arhitektura vpeljuje debelega odjemalca, v okviru katerega je integriran dostop do podatkovne baze, aplikativna logika aplikacije in grafični vmesnik do uporabnika.

Z vpeljavo aplikacije upravljalnega elementa kot posrednika med funkcionalnostjo mrežnega elementa in aplikacijami na omrežni, storitveni in poslovni ravni se pojavi težnja po razslojitvi funkcionalnosti debelega odjemalca. Funkcionalnost debelega odjemalca predstavlja pretvornik med dvema različnima informacijskima shemama. "Pamet" sistema (komutacijska in dostopovna vozlišča), zapisana v obliki tabel v podatkovnih bazah, se v debelem odjemalcu pretvarja v informacijsko shemo, prilagojeno operaterjevemu



Slika 1. Aplikativni strežnik je most med upravljalnim strežnikom in odjemalcem.

razmišljanju, ki temelji na miselnih vzorcih. Podatki se tako grupirajo glede na lastnosti, pomembnosti in upravičenosti.

V konceptu vodenja in nadzora omrežja in storitev ter povezljivosti s poslovnimi procesi se pojavita poleg klasičnega odjemalca (operater profesionalec) še dva tipa odjemalca: aplikacija višjega nivoja in končni uporabnik. Ločitev aplikacijske logike od grafičnega vmesnika omogoči vsem trem odjemalcem uporabo iste aplikativne logike. Uvedemo trislojno arhitekturo s tremi funkcionalnimi sloji:

- shranjevanje podatkov
- aplikativna logika
- predstavitveni sloj, to je grafični vmesnik.

Za sloj shranjevanja podatkov proglasimo podatkovno bazo, ki skrbi za hranjenje podatkov in komunikacijo do aplikativne logike preko standardnih vmesnikov: ODBC, JDBC, SQL, ...

Po potrebi uvedemo še dodatne adaptivne sloje, ki omogočajo elegantne prehode in prilagajanja različnim posebnostim v sistemu, ki bi sicer naredili ostale sloje zelo specifične. S tem pridobimo na splošnosti omenjenih treh slojev.

Aplikativna logika je izvedena s porazdeljenimi objekti. Podatki in akcije se popišejo z lastnostmi in metodami objektov. Izvajajo se v okviru aplikativnih strežnikov. V njih je že koncentrirana potrebna funkcionalnost za upravljanje telekomunikacijskega omrežja. Porazdeljeni objekti so most med podatkovno vodenim elementom omrežja in odjemalcem.

Odjemalci so lahko grafični vmesniki ali aplikacije višjega nivoja. Grafični odjemalci omogočajo uporabniku (osebi) enostavno in pregledno komunikacijo s strojem (element omrežja). Zagotavljajo prijazen uporabniški vmesnik. Aplikacije višjega nivoja se sklicujejo na porazdeljene objekte aplikativne logike in uporabljajo njihove funkcionalnosti pri posredovanju in obdelavi podatkov. V takšnem primeru govorimo, da dva različna sistema neposredno komunicirata med seboj preko njunih srednjih nivojev v trislojni arhitekturi.

5 SISTEMI NADZORA IN VODENJA

Predstavljeni arhitekturni koncept trislojne arhitekture, porazdeljenih objektov in smernice, ki nam jih narekuje koncept TMN, smo realizirali v okviru upravljalnega vozlišča produktne linije SI2000 V5. Cilji, ki jih želimo izpolniti, so:

- centralizirano upravljanje celotnega sistema
- odprtost do aplikacij istega ali višjega nivoja
- možnost enostavne integracije v celovit sistem upravljanja
- prenosljivost med različnimi platformami
- centralizirano vzdrževanje
- enostavna nadgradnja
- enostavna inštalacija
- modularnost.

5.1 Upravljanje omrežja ATM

5.1.1 Uvod

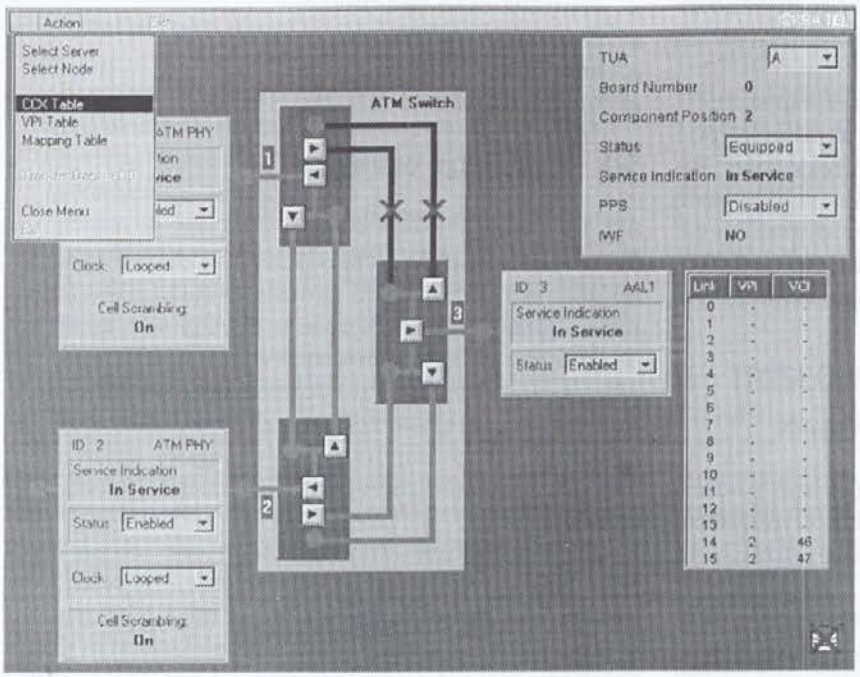
Pri izvedbi projekta je bilo potrebno dodati funkcionalnost administriranja protokola ATM (Asynchronous Transfer Mode), oziroma TUA plošče, že obstoječemu centraliziranemu upravljalnemu sistemu. Po obstoječemu sistemu smo povzeli koncept varnosti. V sistemu imamo lahko več centraliziranih sistemov upravljanja. Predstavljeni sistem omogoča upravljanje vseh, če so le dosegljivi po IP (Internet Protocol) protokolu.

Za izvedbo ustrezne aplikacije smo izbrali Javo. Njen koncept omogoča na enostaven način pokriti vse predpisane zahteve. Prenosljivost med platformami se izraža že v njenem sloganu: razvijaj enkrat in preizkušaj mnogokrat. Odprtost do aplikacij višjega nivoja je zagotovljena preko porazdeljenih objektov, ki so že vsebovani v Javi in podprti v JVM (Java Virtual Machine) z uporabo RMI, ali preko standarda objektov CORBA. Integracija javanskih razredov v spletne strani in uporaba brskalnikov omogočata enostavno integracijo v celovit sistem upravljanja. Aplikativni in programski strežnik zagotavljata centralizirano vzdrževanje sistema. Javanski koncept razredov omogoča enostavno nadgradnjo, inštalacijo in modularnost.

5.1.2 Arhitektura sistema

Pri realizaciji programske opreme v okviru spletnih strani smo se odločili za uporabo porazdeljenih objektov po tehnologiji RMI in možnost enostavne realizacije odjemalca v obliki programčkov (Applets). Takšna tehnologija zahteva na strani odjemalca samo povprečno materialno opremo in dovolj zmogljiv standardni brskalnik. Vendar pa nismo ostali samo pri brskalniku, temveč smo odjemalski del realizirali tudi v obliki samostojne aplikacije in uporabe objektov CORBA. Visok nivo modularnosti nam je omogočal enostavno realizacijo obeh načinov porazdeljenih objektov, ki pa nudijo enako funkcionalnost.

Upravljalni sistem je razdeljen na tri sloje. Najnižji sloj predstavlja podatkovni strežnik. Njegova struktura je ostala nespremenjena. Za srednji sloj smo vpeljali aplikativni strežnik (strežnik porazdeljenih objektov). Vsa aplikativna logika je realizirana s porazdeljenimi objekti. Skupina objektov nudi vso funkcionalnost, potrebno za upravljanje sistema ATM na nivoju produktov SI2000 V5. Porazdeljeni objekti se povezujejo s podatkovnim strežnikom preko standarda



Slika 2. Grafični vmesnik za administriranje sistema ATM

JDBC. Najvišji sloj pa je grafični vmesnik. Njegov namen je na prijazen način omogočiti operaterju upravljanje z sistemom ATM. S srednjim slojem se vrši komunikacija preko klica porazdeljenih objektov ter z uporabo njihovih metod in lastnosti.

5.1.3 Funkcionalnost sistema

Sistem omogoča uporabniku selektivno izbiranje mesta administriranja v omrežju ATM. Najprej je potrebno določiti fizični element administriranja. Določimo ga v treh korakih:

- izbira pripadajočega centraliziranega sistema upravljanja
- izbira želenega elementa omrežja
- izbira ustrezne TUA plošče.

Po izbiri ustreznega elementa (navigacija med elementi omrežja) imamo na voljo zmogljiv grafični vmesnik, ki nam omogoča pregledno dodajanje, brisanje in spreminjanje ustreznih parametrov (slika 2).

Izvedeno funkcionalnost grafičnega vmesnika za administriranje TUA plošče lahko povzamemo v nekaj točkah:

- prijava TUA plošče (integracija v standardno opremo upravljalnega vozlišča)
- sprememba stanja plošče in vhodov
- dodajanje, spreminjanje in brisanje križnih povezav
- določevanje navideznih poti (VPI)
- usmerjanje kanalov v navidezni poti (VPC) z E1 povezavami (Mb/sec)
- preverjanje pravilne nastavitve križnih povezav.

5.2 Dobava in zagotavljanje storitev

5.2.1 Uvod

Vodenje storitev vključuje proces dela z naročnikom in njegovimi nastavitvami storitev, torej dobavo storitev, spremlja in nadzoruje njihovo dogovorjeno kakovost in obračunavanje le-teh. Ker so dani procesi stalni in hitrost njihovega zaključevanja izjemno pomembna, jim moramo tako proizvajalci telekomunikacijske opreme kot tudi operaterji posvetiti posebno skrb. Dani procesi so tesno povezani s poslovnimi procesi operaterjev, ki morajo biti prilagojeni večoperaterskemu okolju in skrbi za končnega uporabnika. Zaželeno je, da vsi, tako operaterji kot tudi končni uporabniki, delajo z istimi informacijami, zato je izrednega pomena, da so podatki shranjeni na centralnem mestu, oziroma se sistemsko skrbi za njihovo konsistentnost. Najprimernejše mesto hranjenja je podatkovna baza. Na ta način imajo vsi subjekti dostop do njih preko grafičnih vmesnikov s pomočjo poenotene aplikativne logike.

5.3 Dobava storitev

Na sejmu TELECOM'99 smo predstavili aplikacijo in arhitekturni koncept rešitve za dobavo storitev. Operaterji na enostaven in uporabniško prijazen način dodajajo in brišejo naročnika in mu spreminjajo lastnosti. Pomembna prednost so svežnji osnovnih in dopolnilnih storitev, ki prinašajo tudi relativno enostavnost njihovih vključevanj v omrežje, tako s strani operaterja kot tudi končnega uporabnika. Na voljo je enoten obrazec za vnos novih podatkov o naročniku (Order Entry), ki pomeni enostavnost pri delu. Razvili smo tudi aplikacijo za končnega uporabnika in sicer lahko spreminja in nastavlja preko brskalnika naslednje storitve:

- brezpogojna preusmeritev klica (CFU – Call Forward Unconditional)
- preusmeritev klica v primeru zasedenosti (CFB - Call Forwarding Busy)
- preusmeritev klica, ko ni odziva (CFNR - Call Forwarding No Reply)

- opozorilni poziv, posamezni in periodični (Alarm Call Service).

Končnemu uporabniku in operaterju se ni potrebno ukvarjati z realizacijo spremljajočih procesov, ker za izvedbo skrbi ustrezna programska oprema. Uporabili in povezali smo objekte, realizirane v okolju Java in povezane preko CORBA.

5.4 Medsebojna povezljivost aplikacij upravljalnega vozlišča

Arhitektura DCOM je uporabljena za povezljivost med interno programsko opremo upravljalnega vozlišča, in sicer se povezujejo odjemalci s samostojnimi aplikacijami na strežniškem delu. S tem je zagotovljeno poenoteno beleženje vseh operacij v sistemu in elegantna rešitev koncepta lokalizacije na posameznih segmentih programske opreme. Uporaba danega koncepta se je pokazala kot primerna in učinkovita rešitev. Spodnja slika prikazuje predstavljeno arhitekturno shemo programske opreme.

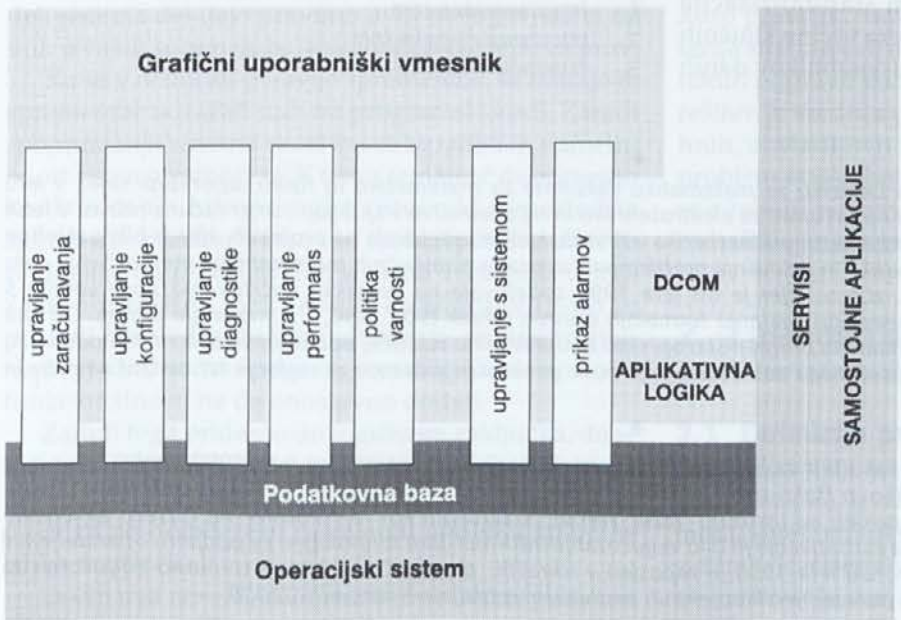
6 TEHNOLOŠKA NADGRADNJA

V zgornjih razdelkih navajamo, da odprti aplikativni vmesniki med ravnmi, tako med storitveno in poslovno kot tudi storitveno ter nižjimi ravnmi, temeljijo na arhitekturah porazdeljenih predmetov CORBA, DCOM ali RMI. CORBA se kaže kot najprimernejša tudi v svetovnem merilu, praktično vsi proizvajalci povezujejo dano rešitev v svoje aplikacije, kar pomeni veliko možnost medsebojne povezljivosti. Hkrati je na voljo enostavna nadgradnja vmesnikov in sočasno delovanje nove in stare realizacije vmesnikov, kar pomeni tudi elegantno menjavanje programske opreme in instalacije novih verzij programske opreme ter hkrati pokrivanje več verzij vmesnikov, tako starih kot tudi novejših.

Pri tehnološki nadgradnji bomo morali upoštevati še naslednje momente. Podatki se nahajajo v podatkovni bazi oziroma podatkovnih bazah v omrežju, ki so običajno med seboj povezane in se podatki tudi pretakajo med njimi. Pomembno je, da so ključni podatki shranjeni le na enem mestu (različni imeniki storitev), od koder so dostopni različnim aplikacijam, ki skrbijo za pridobivanje podatkov, npr. vodenje storitev. Za večjo količino podatkov so primerna podatkovna skladišča, ki imajo že vgrajeno programsko podporo za svetovni splet v samem sistemu za vodenje podatkovne baze. Na voljo je tudi standardni vmesnik za dostop do podatkov in njihovo spreminjanje. Za izmenjavo podatkov se uveljavlja standard XML (eXtensible Markup Language), za hranjenje podatkov se uporabljajo aktivni imeniki (Active Directory) in protokoli za dostop do podatkov kjerkoli v omrežju (podatki o podatkih), kot je na primer LDAP (Lightweight Directory Access Protocol).

Izredna lastnost danega koncepta je tudi ta, da se posamezne ravni v splošnem lahko nahajajo v oddaljenih strežnikih. Lahki odjemalec poskrbi, da se v trenutku izvajanja vsa potrebna programska koda prenese iz oddaljenih strežnikov, oziroma da se oddaljeni predmeti dinamično povezujejo. Slednja rešitev je prav gotovo še bolj učinkovita in elegantna.

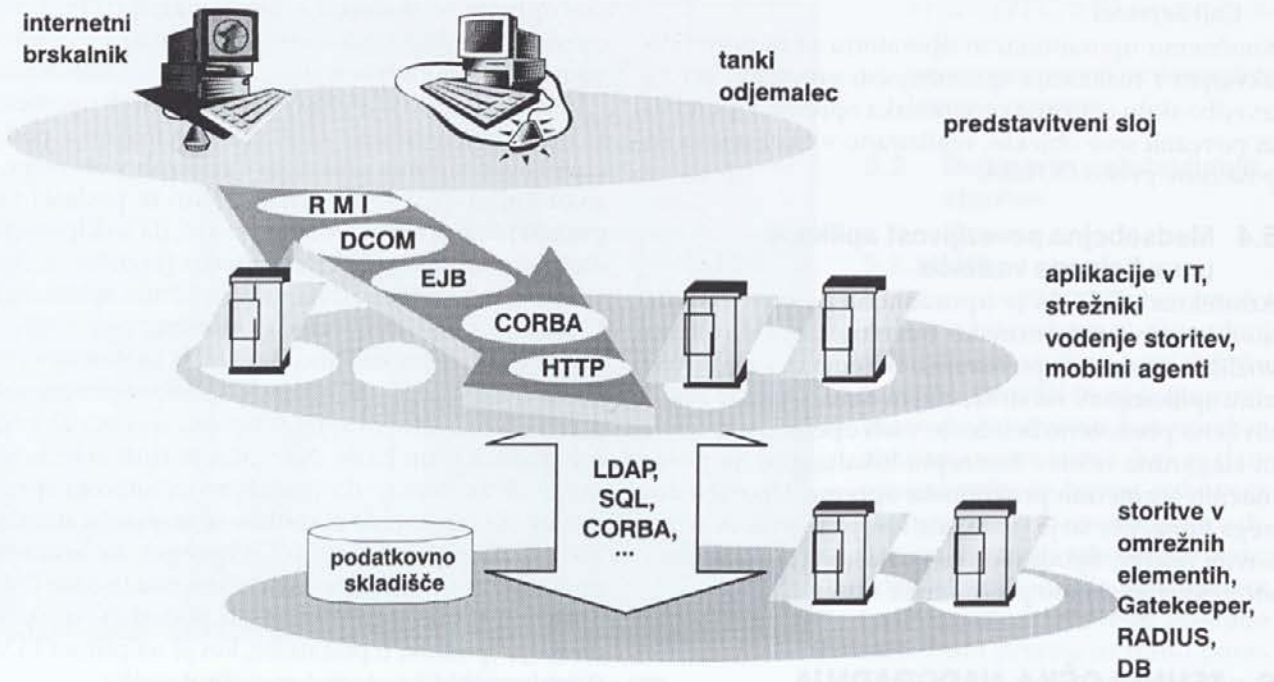
Pomembno vlogo bodo v bližnji prihodnosti odigrali mobilni agenti v omrežju, ki imajo vedenje o osebnih željah uporabnika, njegovih konfiguracijah in imajo hkrati dostop do vseh virov v omrežju, ki hranijo informacijo o dobavi, zagotavljanju in obračunu storitev, bodisi v poslovnem ali telekomunikacijskem omrežju. To je v zelo tesni povezanosti z mobilnostjo končnega uporabnika, ki bo imel možnost vključevanja v omrežje kjerkoli v okviru možnosti dostopa in dovoljenj za uporabo posameznih aplikacij ali storitev.



Slika 3. Medsebojna povezljivost aplikacij na upravljalnem vozlišču

7 ZAKLJUČEK

Opisani koncept upravljanja vpeljuje visoko stopnjo modularnosti pri gradnji celotnega sistema in povezljivosti proti višjim ravnem. Aplikativni nivo omogoča enostavno



Slika 4: Prikaz uporabe tankih odjemalcev, novih arhitektur porazdeljenih komponent in mobilnosti

medsebojno funkcionalno povezovanje aplikacij in gradnje različnih grafičnih vmesnikov, ki so prilagojeni operaterjem in ne strojni opremi. Z razširitvijo in uvedbo dodatnih tehnoloških nadgradenj bomo lahko zadostili tudi najbolj svežim zahtevam na področju vsesplošne mobilnosti in osebnim nastavitvam končnega uporabnika. Zrelost in standardizacija danih paradigem ter splošna podpora večine ključnih proizvajalcev, tako orodij kot tudi uporabnikov danih rešitev, nakazujeta jasne usmeritve v svetovnem merilu ter uporabnost in razširjenost danih konceptov. Slednje potrjujejo tudi izkušnje pri izdelavi programske opreme v podjetju Iskratel.

8 LITERATURA

1. http://www.iona.com
2. http://www.visigenic.com
3. http://www.sun.com

◆

Mag. Ana Robnik je diplomirala na Oddelku za matematiko Fakultete za matematiko in fiziko, uporabna smer, v letu 1985. Magisterski študij je zaključila na Fakulteti za elektrotehniko in računalništvo v Ljubljani, smer računalništvo. V letih 1985 do 1991 je bila zaposlena v podjetju Iskra Kibernetika v Kranju, kjer je sodelovala na projektih, kjer je bilo potrebno znanje s področja uporabne matematike in reševanja problemov z uporabo numeričnih matematičnih metod. Svoje delo je nadaljevala v podjetju IskraTEL, d.o.o., kjer je od leta 1993 sodelovala na projektu načrtovanja nove verzije 5 telekomunikacijskih sistemov, predvsem pri uvajanju formalnih opisnih tehnik (SDL, MSC, ...) v razvojni proces. Od leta 1997 vodi sektor za sistemsko in aplikativno programsko opremo za upravljalno vozlišče, podatkovno bazo in metodologijo verzije 5 in več. Pomembno področje njenega dela je uvajanje nove generacije sistemov za vodenje ter nadzor omrežja in storitev.

◆

Dr. Igor Šalamun je diplomiral na Fakulteti za elektrotehniko in računalništvo v Ljubljani leta 1990. Izobraževanje je nadaljeval na Fakulteti za matematiko in fiziko smer Jedrska tehnika, kjer je tudi magistriral (1995) in doktoriral (1998). Od leta 1991 pa do 1998 je bil zaposlen na Institutu "Jožef Stefan" v Ljubljani kot raziskovalec na Odseku za jedrsko tehniko. Glavno področje njegovega raziskovanja je bilo vključevanje informacijske tehnologije za podporo operaterjem v kontrolnih sobah jedrskih elektrarn. Od leta 1998 je zaposlen v IskraTEL, d.o.o., Kranj, kjer je trenutno odgovoren za javansko tehnologijo in tehnologijo porazdeljenih objektnih modelov v upravljalnih sistemih SI2000.

REFACTORING – PREOBLIKOVANJE PROGRAMSKE KODE

Uroš Grajfoner, Peter Repinc
OdaTeam d.o.o.
Meljska 36, SI-2000 Maribor
e-pošta: uros@odateam.com, pero@odateam.com
URL: http://www.odateam.com

Izvleček

Spreminjanje obstoječe programske kode zaradi dodajanja nove funkcionalnosti se lahko izvaja z natančno izbranimi in definiranimi koraki. Takemu procesu pravimo preoblikovanje programske kode. Izvaja se združeno z drugimi razvijalskimi aktivnostmi, pri ekstremnem programiranju pa je bistveni del razvojnega procesa. Z njim programska struktura postane razumljivejša, jasnejša, pravilnejša in fleksibilnejša za dodajanje nove funkcionalnosti. Proces je podprt z avtomatiziranim testiranjem. V najboljšem primeru se proces izvaja avtomatsko z orodji za preoblikovanje. Preoblikovanje zbija ceno spremembe obstoječega sistema in povečuje njegovo preglednost.

Abstract

Refactoring is a process of adding new functionality by changing the existing program code. It can be performed by exactly chosen and defined steps. It is practiced together with other design activities, while in extreme programming it is an essential part of the development process. With refactoring, the program structure becomes more understandable, clearer, more correct and flexible for adding new functionalities. Automatic testing by refactoring tools support the process. Refactoring reduces the cost of changing the existing system and improves its transparency.

■ ■ ■

1. UVOD

Ko se začne zapletati...

V življenjskem ciklu informacijskega sistema naletimo na dele informacijskega sistema, ki več ne odsevajo dejanskega stanja v realnosti. Dokler taki deli sistema delujejo pravilno in ni razlogov, da bi na njih izvajali spremembe, je na videz vse lepo in prav.

Ko se v realnosti pojavijo spremembe, se morajo te spremembe odražati tudi na programski kodi. Zaradi spreminjanja sistema skozi čas se zgradba in namembnost sistema zameglita. K temu še največ doprinesejo kratkoročni in hitri posegi v sistem, ki so posledica zahtev uporabnikov, časovne stiske, ipd. Zaradi tega se dogaja, da določeni deli sistema vsebujejo veliko začasnih rešitev, naknadnih dograditev in jih je posledično vedno težje vzdrževati, so vedno bolj nepregledni, vedno pogosteje se dogaja, da se nove funkcionalnosti ne da enostavno dodati.

Zaradi tega pridemo do logičnega zaključka, da je sistem treba preoblikovati, da bi lahko ob dodajanju nove funkcionalnosti še vedno ustrezno deloval, ali pa da bi ga po določenem časovnem obdobju sploh še lahko razumeli.

Dodajanja nove funkcionalnosti se lahko lotimo na različne načine. En način je, da določen podsistem zgradimo znova. Tak pristop pa je lahko zelo drag, dolgotrajen in zelo tvegan. Druga možnost je kopiranje in prirejanje delov sistema in s tem razširjanje sistemskih zmogljivosti. Vendar so te zmogljivosti le na videz velike, saj sistem s takimi posegi postane prehitro prevelik, napake se hitreje množijo, implementacija sistema več ne ustreza načrtu, stroški inkrementalnih popravil pa se pomnožijo. Najbolj sprejemljiva rešitev je srednja pot: preoblikovanje sistema v majhnih, obvladljivih korakih. Na tak način je vpogled v problem boljši, bolj osredotočen, arhitektura sistema se s takimi popravili izboljšuje, s tem pa je olajšano tudi dodajanje novosti.

2. Lastnosti preoblikovanja

Izhodiščne informacije

2.1 Definicija preoblikovanja

Preoblikovanje je proces spreminjanja programske kode tako, da se obnašanje sistema navzven ne spremeni, spremeni in izboljša(!) pa se notranja struktura sistema. Je sistematičen način »prečiščevanja« kode in zmanjševanja možnosti napak.

Namen preoblikovanja je spreminjanje sistema tako, da ga je lažje razumeti in dopolnjevati. Preoblikovanje

spreminja način gradnje informacijskih sistemov iz ustaljenega zaporedja »načrtovanje, gradnja, vzdrževanje« v delovni proces, kjer načrtovanje, gradnja in vzdrževanje niso več ločeni procesi, ampak se prepletajo in ponavljajo v celotnem življenjskem ciklu informacijskega sistema. S pomočjo preoblikovanja lahko sistem načrtujemo tudi, ko je že implementiran.

2.2. Kdaj preoblikovati

Idealno bi bilo imeti programski sistem z brezhibno arhitekturo, s čudovito strukturirano in jasno programsko kodo. Število podjetij, ki si lahko privoščijo tak sistem, ki hkrati deluje brez časovnih omejitev, limitira proti nič. Na žalost. Čas in denar sta vedno omejeni dobrini. Zaradi tega preoblikovanje ni proces, ki bi se nenehno odvijal skozi celotno življenjsko dobo nekega informacijskega sistema. Pomembno je prepoznati trenutke, ko je preoblikovanje kode najbolj smiselno.

2.2.1. Dodajanje nove funkcionalnosti

Najpogostejši vzrok za preoblikovanje obstoječe kode je dodajanje nove funkcionalnosti. Zahteve uporabnikov praviloma niso popolne in dokončne. Okolje delovanja programskega sistema se vedno spreminja. Znanje razvijalcev in načrtovalcev je različno. Vsi ti vzroki botrujejo rojevanju potrebe po novi fukcionalnosti, ki jo mora sistem vsebovati.

Pri dodajanju nove funkcionalnosti vedno naletimo na programsko kodo, ki jo je treba na neki način spremeniti. Verjetnost, da to kodo popolnoma poznamo, je majhna. Da bi kodo lahko bolje razumeli, jo lahko preoblikujemo. S tem izboljšamo njeno izraznost, kar vsekakor pride prav, ko naslednjič naletimo na isto kodo.

Drugi dober razlog za preoblikovanje je struktura kode, ki ne omogoča preprostega dodajanja nove funkcionalnosti. Seveda lahko vedno dogradimo obstoječo kodo in z določenimi posegi in izjemami dosežemo, da koda deluje tudi z novo funkcionalnostjo, vendar na ta način še povečamo nerazumljivost kode, naše življenje bo pa precej manj lepo, ko bomo spet naleteli na njo. Svet si lahko naredimo prijaznejši, če v takem primeru kodo najprej preoblikujemo tako, da je novo fukcionalnost enostavno dodati. In šele nato dodamo novo funkcionalnost.

2.2.2. Popravljanje napak

Dober razlog za preoblikovanje kode je popravljanje napak. Kodo najprej preoblikujemo, da bolje razumemo kontekst. Ko je sistem razumljivejši, je napako mnogo lažje najti in jo zatem popraviti.

Povedano drugače, napaka, ki jo popravljamo, je verjetno posledica nejasnosti kode. Preoblikovanje je torej tudi način, s katerim si zaradi boljše preglednosti zagotavljamo boljše razumevanje kode in s tem zmanjšujemo verjetnost napak.

2.2.3. »V tretje gre rado«

To smernico je dal Don Roberts Martinu Fowlerju [1]. Prvič napišemo kodo in upamo na najbolje. Ko drugič ustvarimo nekaj podobnega, se očitnega podvajanja zavedamo, morda celo ustrašimo, a večinoma zaradi različnih pritiskov kodo vseeno podvojimo. Ko ustvarimo podobno kodo že tretjič, je čas za preoblikovanje.

2.2.4. Preoblikovanje ob pregledih kode

Pregledi kode omogočajo prepoznavanje namer programerjev. Prepoznava se slabo načrtovanje, dobre ideje, odkrivajo se napake. Pri pregledih kode razvijalci z manj izkušnjami pridobivajo novo znanje. Pri tem se velikokrat zbirajo ideje in predlogi kako kodo spremeniti, da bi bolje služila svojemu namenu.

S preoblikovanjem lahko take predloge udejanjimo. Rezultat je drugačna struktura programa, ki jo lahko zavržemo, če se izkaže kot slabša alternativa, ali pa jo uporabimo kot izhodišče za naslednjo izboljšavo. Le-to bi si seveda težko predstavljali, če ne bi naredili prve spremembe. Poleg tega pri pregledu kode ostane samo pri predlogih, pri preoblikovanju pa lahko rezultate novih idej vidimo pred sabo zelo kmalu. Če je nov načrt dober, ga lahko obdržimo in sistem bo boljši. Če ne bo, se lahko vrnemo na prejšnje stanje in bogatejši bomo za novo izkušnjo.

Pri ekstremnem programiranju je tak način preoblikovanja potenciran, saj se programira v parih, kjer je preoblikovanje konstanten del razvojnega procesa. Razvijanje v parih zato deluje kot nenehen pregled kode s preoblikovanjem.

Seveda pa preoblikovanje ni univerzalen odgovor na vsako težavo. Prav tako ga je na nekaterih področjih težavno uporabiti, so tudi primeri, ko sploh ni priporočljivo. Problematično je lahko recimo preoblikovanje baze podatkov (oz. objektnega modela), kjer se moramo v večini primerov soočiti s konvertiranji v nova stanja podatkov. Prav tako lahko nastopijo težave, ko spremenimo vmesnik komponente, ki se pogosto uporablja.

V nekaterih primerih pa preoblikovanje enostavno ne pomaga in je najbolje kak del sistema odstraniti in implementirati znova. Po našem mnenju so to sistemi z resnimi napakami v samem načrtu ali sistemi, ki so preveč nestabilni za produkcijsko uporabo. Novo gradnjo kakega sistema si, če je to mogoče, lahko zamislimo tudi kot dolgotrajno globalno preoblikovanje.

2.3. Ko »zavohamo« priložnost...

V prejšnjem poglavju so opisane različne situacije, pri katerih je preoblikovanje potrebno ali priporočljivo.

Ni pa konkretnih navodil, na katerih mestih naj se lotimo dela. Prav tako nam pri samem začetku, ko se lotevamo preoblikovanja prvič, ni popolnoma razvidno, kje naj rešujemo težave najprej. Čez čas odkrijemo, da smo dobili »nos« za določene konstrukte kode, kjer imamo precej dobro zamisel, kako bi ta koda lahko delovala bolje in kako bi lahko bila jasnejša.

V naslednjih nekaj vrsticah so opisana nekatera mesta v kodi, kjer je (zelo) očitno, da je potrebno preoblikovanje. Veliko primerov je gotovo prepoznavnih iz vsakdanjega obdelovanja kode.

- *Podvojena programska koda* – najpogostejši razlog za preoblikovanje. Če se pojavi pri istem razredu, je rešitev ponavadi izsek podvojene kode in definiranje nove metode. Pri razredih v hierarhiji se skupno uporabljana koda premakne na ustrezen abstraktni razred. Če povezav med razredi ni, lahko naredimo celo nov razred, ki opravlja naloge podvojene kode.
- *Predolga metoda* – Merila za dolžino metod v objektnih programih so različna. V večini primerov pa se predolgih metod lahko rešimo z izsekom dela kode in definiranjem nove metode. V drugih primerih lahko veliko množico parametrov zamenjamo z objektom. Metodo lahko predstavimo tudi kot nov objekt ali hierarhijo objektov.
- *Prevelik razred* – Razred združuje preveč funkcionalnosti in informacij, ima »kompleks švicarskega noža«. Rešitev je dekompozicija na več manjših razredov, odgovornih za posamezne naloge.
- *Dolg nabor parametrov* – , ki jih potrebuje neka metoda, lahko nadomestimo s povpraševalnimi stavki, predajanjem objekta kot parametra itd.
- *Raznolike spremembe* - , ki jih opravljamo pri vedno istem razredu, sugerirajo kreiranje več različnih razredov z jasnimi odgovornostmi.
- *Poseg s šibrovko* - Pri določenih spremembah je potrebno spreminjati kodo na vedno enakih mestih v različnih delih sistema. Rešitev je v definiciji centralnega razreda ali metode, kjer se izvaja vsakokratna sprememba.
- *Zavidanje lastnosti* – se pojavlja pri razredih, ki opravljajo svoje delo na drugih objektih in jim »zavidajo« njihove lastnosti. Najpogostejša rešitev je premik kode ali dela kode k pravemu lastniku.
- *Kepe podatkov* – podatki, ki večinoma nastopajo skupaj, se morajo pravzaprav združiti v skupen objekt, ki kasneje pridobi še nekatere odgovornosti in operacije.
- *»IF« in »CASE« stavki* – v objektnih jezikih se pogojnih izrazov lahko lepo izognemo s polimorfizmom in hierarhijo.
- *Zavrnjena zapuščina* – podrazredi ne potrebujejo lastnosti nadrazredov. Očitno je problem v napačno zastavljeni hierarhiji (in pripadajočih metodah).
- *Komentarji* – pričajo o nejasni in nezgovorni kodi. Rešitev za tako kodo je preimenovanje imen metod in izrezovanje delov kode in primerno poimenovanje novih metod.

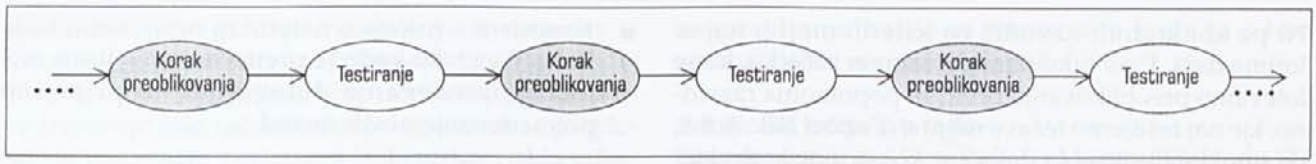
2.4. Izvajanje in načini preoblikovanja

Preoblikovanje je resen poseg v programsko kodo, ki ne spreminja njene funkcionalnosti, spreminja pa notranjo strukturo sistema. Pri vsakem preoblikovanju pa lahko pride do kake napačne konstelacije in novo ustvarjena koda več ne dela enakih stvari kot prejšnja. Zaradi varnosti preoblikovanje razdelimo na čim manjše korake, ki so točno definirani. Pri teh korakih natančno poznamo zaporedje dogodkov, ki bodo spremenili kodo. S tem zagotovimo, da ne bomo pozabili opraviti kakega pomembnega dela preoblikovanja, po drugi strani se s standardnimi operacijami varujemo pred napakami.

Taki standardni koraki se delijo na več sklopov [1], kot na primer:

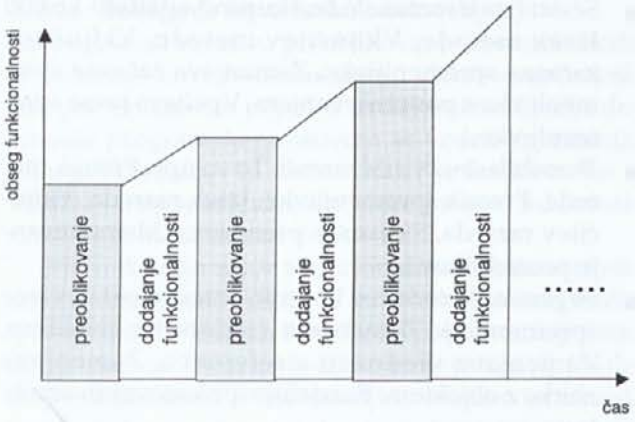
- *Sestavljanje metod.* V to skupino spadajo koraki Izsek metode, Vključitev metode, Vključitev začasne spremenljivke, Zamenjava začasne spremenljivke s povpraševanjem, Vpeljava jasne spremenljivke...
- *Premiki lastnosti med razredi.* To so npr. Premik metode, Premik spremenljivke, Izsek razreda, Vključitev razreda, Skrivanje prenosov, Odstranjevanje posrednikov...
- *Organizacija podatkov* kot npr. Interno ograjevanje spremenljivk, Zamenjava vrednosti z objektom, Zamenjava vrednosti z referenco, Zamenjava zbirke z objektom, Razdelitev prikazovanih objektov...
- *Poenostavitve pogojnih izrazov.* Razstavitev pogoja, Zamenjava pogoja z metodo, »Izpostava« podvojenih posledic pogojev, Odstranitev kontrolnih zastavic....
- *Poenostavitve klicev metod* so recimo Preimenovanje metode, Dodajanje parametra, Odstranitev parametra, Ločitev povpraševanja od spreminjanja, Parametriziranje metode, Ohranitev celega objekta...
- *Obvladovanje generalizacije* vsebuje korake kot Potegni spremenljivko gor, Potegni metodo gor, (oboje tudi dol), Premik konstruktorja gor, Izdelava nadrazreda, Izdelava podrazreda, Izdelava vmesnika, Porušenje hierarhije...

Osnova za varno izvajanje posameznih korakov preoblikovanja je avtomatizirano testiranje. Pred začetkom preoblikovanja morajo biti na voljo testni primeri, ki preverjajo pravilnost izvajanja dela sistema. Preoblikovanje se izvršuje korak za korakom, po vsakem koraku pa sledi zaganjanje testov in s tem dokazovanje, da sistem še vedno deluje enako.



Slika 1. Potek preoblikovanja

Celotna implementacija se v bistvu deli na dodajanje funkcionalnosti in preoblikovanje. Ko dodajamo funkcionalnost, dodajamo tudi nove testne primere. Ko preoblikujemo, ohranjamo enako funkcionalnost, prav tako tudi testne primere (razen v primeru, ko odkrijemo napako. Takrat se definirajo novi testni primeri, ki odkrivajo napako). Tako dodajanje funkcionalnosti kot preoblikovanje pa pomenita dodajanje določene vrednosti v programski sistem. Z dodajanjem funkcionalnosti sistem zmore več. Z preoblikovanjem je sistem razumljivejši, pravilnejši in bolj »gostoljuben« za novo funkcionalnost.



Slika 2. Vloga preoblikovanja v razvojnem procesu

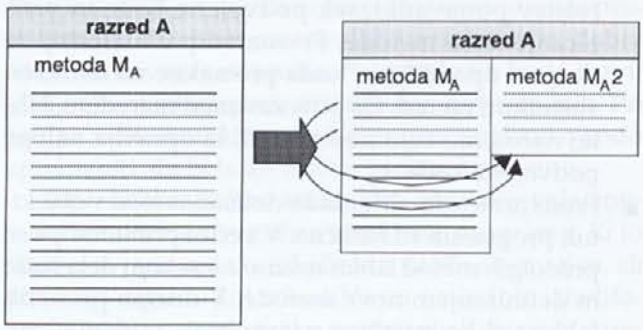
3. Primer enostavnega preoblikovanja

Preoblikovanje v praksi

Vsakdo, ki se ukvarja z razvojem programskih sistemov, je že naletel na odseke kode, ki so najprej povzročali glavobol, šele nato je sledilo njihovo razumevanje. V takih primerih si želimo, da bi koda bila jasnejša, bolje strukturirana, enostavneje zapisana itd. Z majhnimi koraki preoblikovanja lahko tako kodo prikrojimo, da bo njena izraznost občutno izboljšana.

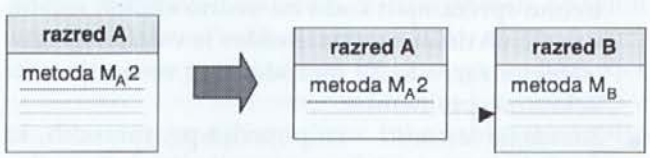
Vzemimo na primer metodo, na katero smo naleteli med dodajanjem neke funkcionalnosti. Metoda vsebuje veliko vrstic kode in je nepregledna. Zaradi neprimerno prevelike dolžine kode »zavohamo«, da bi jo lahko preoblikovali (vonj *Predolga metoda*). Privzemimo, da se metoda imenuje M_A in da pripada razredu A.

Najprej opazimo, da sta dva dela metode skoraj identična (*Podvojena programska koda*). Uporabimo korak »Izsek metode« in kodo, ki se ponavlja, izsekamo iz metode M_A ter ustvarimo novo metodo M_A2. Metoda M_A zdaj kliče metodo M_A2 na mestih, kjer je bila koda podvojena. Metoda M_A2 sedaj vsebuje podvojeno kodo. Rezultat je prikazan na sliki 3. Puščice prikazujejo klic metod.



Slika 3. Izsek podvojene kode

Na tem mestu opazimo, da metoda M_A2 pravzaprav vseskozi obdeluje le objekt razreda B, pozna objektovo notranjo strukturo in/ali celo kliče privatne metode razreda B. Večina metode M_A2 pravzaprav obdeluje samo objekt razreda B (*Zavidanje lastnosti*). Uporabimo koraka Izsek metode, kjer kodo, ki se ukvarja samo z razredom B, izsekamo iz metode M_A2, nato pa še Premik metode, s katerim novo ustvarjeno metodo premaknemo k razredu B in jo poimenujemo kot metoda M_B.



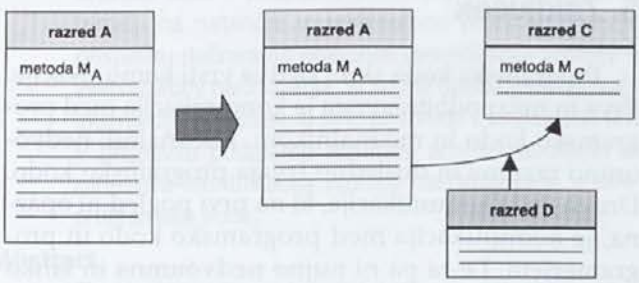
Slika 4. Premik kode k pravemu lastniku

Vrnemo se na metodo M_A, ki se še vedno zdi predolga (*Predolga metoda*). Opazimo odsek metode, ki se pretirano ukvarja z objektom razreda C (*Zavidanje lastnosti*). Omenjeni odsek izsekamo iz metode M_A (Izsek metode), nato pa ga preselimo v metodo M_C k razredu C (Premik metode), kamor pravzaprav spada.

Čeprav zdaj metoda M_A kliče metodo M_C, pa jo mora klicati z daljšim nizom parametrov (*Dolg nabor parametrov*). Poleg tega opazimo še, da ti parametri

v razredu A velikokrat nastopajo skupaj in so skupaj uporabljani še v nekaterih drugih metodah (*Kepe podatkov*).

V tem primeru uporabimo korak Izsek razreda, kjer skupine objektov, ki večinoma nastopajo skupaj, združimo v skupen objekt tako, da definiramo nov razred D s skupino objektov v njegovih instančnih spremenljivkah. Zatem spremenimo metodo M_C tako, da za svoje delovanje več ne zahteva prej omenjene množice parametrov, ampak objekt razreda D (Vpeljava objekta kot parametra). Prav tako moramo spremeniti vso kodo pri razredu A (med drugim tudi metodo M_A), kjer se nanašamo na skupino parametrov, in jo zamenjamo z uporabo objekta razreda D. Preglednost in jasnost posameznih metod se je tako neprimerno povečala.



Slika 5. Objekt namesto množice parametrov

Po izvršenih korakih preoblikovanja smo končno zadovoljni z velikostjo metode M_A. Programska koda, ki je predtem bila zgoščena na enem mestu, je zdaj razdeljena po sistemu, glede na njeno namembnost. Naredili smo kar nekaj sprememb. Oglejmo si njihov skupni učinek.

Na prvi pogled je rezultat mnogo bolj zapleten. Taka sestava programa pa ima zdaj neprimerno več dobrih strani. Metoda M_A, ki je bila naš začetni problem, je zdaj krajša in preglednejša. Prav tako so tudi vsi ostali odseki kode (nove metode) zdaj jasnejši. Nove metode so manjše in obvladljivejše. Pripadajo k tistim objektom, katerih strukturo in lastnosti najbolje poznajo. Namembnost metod je bolj razdrobljena in pravilneje porazdeljena. Vsaka metoda opravlja bolj specializirano funkcijo. Vsaka sprememba kode se zdaj lahko izvrši na mestu, ki je za njo najbolj primerno, t.j. v metodi, ki je »specializirana« za izvajanje določenega opravila. Pred izvedbo preoblikovanja bi se lahko zgodilo, da bi zaradi manjše spremembe morali spreminjati metodo M_A na več mestih. Tukaj smo verjetnost takega dogodka občutno zmanjšali.

Seveda moramo vseskozi zagotavljati, da program še vedno deluje. Po vsakem koraku je potrebo izvesti ustrezna testiranja programa, ki zagotavljajo, da program še vedno deluje.

Pri preoblikovanju lahko naletimo tudi na logične napake, ki jih prej zaradi prevelike množine podatkov nismo uspeli opaziti. Ko jih odpravimo, napišemo nove testne primere, ki take napake odkrivajo.

Prikazani način dela, izmenjujoči koraki preoblikovanja s sprotnim testiranjem, ima še eno dobro stran. Po vsakem opravljenem koraku se zavedamo, da je koda razumljivejša, čitljivejša, lepša za ponovno uporabo. Po vsakem testiranju pa si zagotovimo, da je še vedno pravilna. Lahko se zgodi, da začnemo preoblikovati šele proti koncu delovnega dne. Po nekaj izvedenih korakih spoznamo, da načrtovanih sprememb ne bomo mogli izpeljati do konca. Kljub temu opravljenega dela ni potrebno zavreči. Vsak korak je natančno definiran in preverjen s testnim orodjem, zato sistem še vedno deluje pravilno in je v vsakem primeru boljši, kot je bil pred začetkom preoblikovanja. Po vsakem koraku preoblikovanja lahko torej zaključimo z delom in nadaljujemo naslednjič.

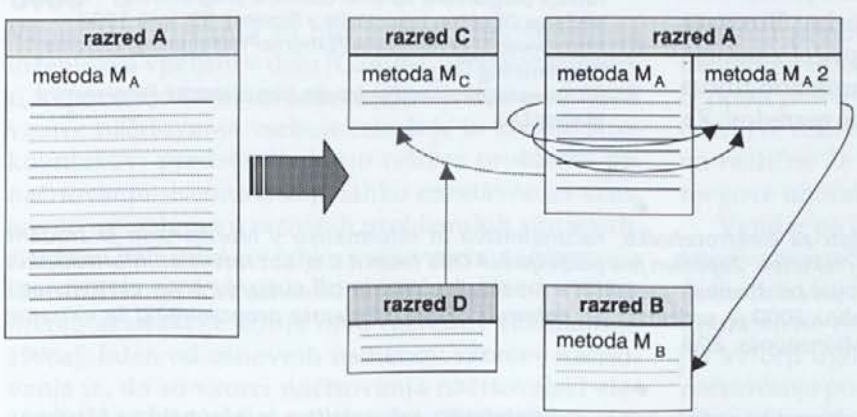
4. Orodja za preoblikovanje

Olajšanje življenja

Natančno definirani koraki preoblikovanja programske kode [1] nudijo razvijalcem gotovost, da bodo koraki izvedeni pravilno in popolno. S tem se do določene mere lahko znebimo negotovosti, ki lahko spremlja razvijalce pri preoblikovanju. Proti taki negotovosti imamo še eno orožje: avtomatizirano testiranje.

Čeprav je preoblikovanje podprto s sistematiko in testiranjem, pa se ga razvijalci vseeno pogosto izogibajo. Razlog je preprost. Preoblikovanje zahteva svoj čas. Razvijalci se ga izognejo, ker je na prvi pogled predrago.

Na tem mestu ni odveč poudariti, da je preoblikovanje proces, ki



Slika 6. Rezultat preoblikovanja

dolgoročno poceni dodajanje funkcionalnosti. Z njim odkrivamo napake in izboljšujemo programsko kodo. Brez preoblikovanja bi vsako naknadno dodajanje funkcionalnosti bilo vedno težje izvedljivo in s tem vedno dražje. Z uporabo preoblikovanja je razvoj programja na začetku navidezno upočasnjen, vendar se s časom praksa ustali, dodajanje nove funkcionalnosti ima še vedno relativno nizko ceno, napredek je zmernejši, a konstantnejši in hitrejši.

Orodja za preoblikovanje nudijo avtomatsko izvajanje korakov preoblikovanja. S tem je vklop preoblikovanja v vsakdanji razvojni proces močno olajšan. Vsak korak preoblikovanja, ki se mu razvijalec lahko izogne zaradi pomanjkanja časa, je zdaj lahko opravljen hitro in avtomatsko, z označevanjem kode in izbiro ustreznih ukazov. S tem odpadejo tudi preverjanja pravilnosti predajanja novih parametrov, osveževanja klicateljev preimenovanih metod in drugo. Proces izvedbe posameznega koraka preoblikovanja traja nekaj sekund, medtem ko bi manualno izvajanje enakega postopka lahko trajalo kar nekaj minut. Poleg tega postane vsakokratno testiranje nepotrebno, saj se vse potrebno delo izvede avtomatsko. Kar pa še ne pomeni, da s preoblikovanjem postane testiranje odvečno. Preprosto ga ne potrebujemo tako pogosto.

Z orodji se cena preoblikovanja zniža, posredno s tem pade tudi cena razvojnih napak. Zaradi tega se lahko izognemo pogosto zapletenim, vnaprej pripravljenim razvojnim načrtom, ki nastajajo zaradi pomanjkanja zahtev. Takšni razvojni načrti vsebujejo fleksibilnosti, ki se s kasnejšo uporabo izkažejo kot nepotrebne. Tako se povečuje kompleksnost programja. Brez poznavanja preoblikovanja pa bi bilo nefleksibilne sisteme zelo drago spreminjati. Z avtomatiziranim preoblikovanjem si lahko privoščimo enostavnejše načrtovanje, saj je spremembe lažje izvajati, razširjanje načrtov in dodajanje funkcionalnosti ni več tako cenovno potratno.

Na področju preoblikovalnih orodij na žalost vlada zaenkrat velika praznina. V integriranih razvojnih okoljih kot so recimo okolja za Smalltalk, je zaenkrat razvito orodje poimenovano Refactoring Browser (Preoblikovalni brskalnik), ki pa se je pričel znatneje uporabljati šele, ko je bila njegova funkcionalnost vključena v vsakdanji brskalnik knjižnic razredov. Za okolja kot je npr. Java, kjer se programska koda vpisuje tako rekoč v preprost urejevalnik besedila, se medsebojno prepletene reference na objekte in metode ne shranjujejo, zato je izdelava orodja za preoblikovanje še posebej otežena. Nekatera naprednejša okolja, kot je npr. IBM-ov VisualAge for Java, posnemajo smalltalkovsko dinamično osveževanje programskega repozitorija.

Orodje za preoblikovanje naj bi torej na neki način obvladovalo programske reference med razredi, moralo bi delovati natančno, dovolj hitro za vsakdanjo uporabo, omogočati preklice posameznih korakov preoblikovanja (»korakanje nazaj«), še najlepše pa bi bilo, ko bi bilo tako orodje integrirano v razvojno okolje samo.

5. Zaključek

Končno

Programska koda skrbi za dve vrsti komunikacije. Prva in neizpodbitna vrsta je komunikacija med programsko kodo in računalnikom. Računalnik nedvoumno razume in dosledno izvaja programsko kodo. Druga vrsta komunikacije, ki na prvi pogled ni opazna, je komunikacija med programsko kodo in programerjem. Le-ta pa ni nujno nedvoumna in lahko funkcionira zelo slabo. Z preoblikovanjem izboljšujemo predvsem komunikacijo med programsko kodo in programerjem. Hkrati pa skrbimo, da je naš informacijski sistem jasna in nedvoumna slika realnosti.

LITERATURA

1. FOWLER, Martin, "Refactoring", Addison-Wesley, Reading, 1999
2. BECK, Kent, "Extreme programming explained", Addison Wesley, Reading, 1999
3. FOWLER, Martin, KENDALL, Scott, "UML Distilled", Addison-Wesley, Reading, 1997
4. ROBERTS Don et al., "Why Every Smalltalker Should Use the Refactoring Browser", The Smalltalk Report, letnik 6, številka 10, september 1997
5. ROSTAHER, Matevž, KLINE, Andrej "Proces skupinskega razvoja programske opreme (Extreme programming)", Zbornik srečanja Objektna tehnologija v Sloveniji 99, junij 1999
6. http://www.c2.com/cgi/wiki?ExtremeProgramming, Extreme Programming Discussion
7. extremeprogramming@egroups.com, Extreme Programming Mailing List

◆

Uroš Grajfoner je diplomiral na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru in je študent podiplomskega študija informatike na tej fakulteti. Zaposlen je v podjetju FJA Oda Team d.o.o. kot razvijalec informacijskih sistemov. V zadnjih petih letih je sodeloval pri številnih projektih v Smalltalku, Javi in pri uvajanju prvin ekstremnega programiranja v proces razvoja. V oktobru 2000 je sodeloval pri delavnici "From Moderate programming to eXtreme Programming" na konferenci OOPSLA, Minneapolis, ZDA.

◆

Peter Repinc je študent računalništva in informatike na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru. Od leta 1998 je zunanji sodelavec podjetja OdaTeam d.o.o. Sodeloval je pri projektih v Smalltalku, Javi in XP.

OBNAVLJANJE NAČRTOVANJA S POMOČJO VZORCEV NAČRTOVANJA

Tomaž Domajnko, Ivan Rozman, Marjan Heričko
Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko
Smetanova ulica 17, 2000 Maribor, Slovenija
e-mail: {tomaz.domajnko, ivan.rozman, marjan.hericko}@uni-mb.si

Izvleček

Vzorci načrtovanja so deležni velikega zanimanja v svetu razvijalcev objektnih sistemov, saj tvorijo pomemben delež visokonivojskih informacij, ki lahko v veliki meri poenostavijo razumevanje in obvladovanje kompleksnih sistemov. Hkrati pa se razvijalci zavedajo, da za večino objektnih sistemov ne obstaja dokumentacija, ki bi vzorce načrtovanja eksplicitno izražala. Zato v prispevku predstavljamo pristop, ki omogoča ekstrahiranje informacij o uporabljenih vzorcih. Pristop temelji na natančni in nedvoumno predstavitvi vzorcev načrtovanja. Na podlagi predstavitve vzorcev načrtovanja v prispevku definiramo postopek identifikacije vzorcev načrtovanja v obstoječih sistemih. Pristop zagotavlja identifikacijo vseh vzorcev načrtovanja, ki so bili uporabljeni ob načrtovanju in implementaciji objektnega sistema. To pa je osnova za ekstrakcijo odločitev, ki smo jih izvedli v aktivnostih arhitekturnega načrtovanja, načrtovanja objektov in implementacije. V prispevku podajamo rezultate analize nekaterih objektnih ogrodij in knjižnic razredov, ki potrjujejo, da je možna zanesljiva identifikacija vzorcev načrtovanja in s tem ekstrakcija pomembnih, visokonivojskih informacij iz obstoječe programske kode.

Abstract

The design patterns present a very important source of high-level information about the understanding and the control of a complex object system. At the same time the development community realizes that the documentation of the current object systems does not include the information about the use of design patterns to the appropriate extent. Therefore in the article we will present the approach to the generation of the information. The approach is based on the rigorous and unambiguous presentation of the design patterns, which will also be presented in the article. The presentation enables a simple and effective implementation of the identification process for any instance of the design pattern, implemented in the object system according to the specification. We present the results of the analysis of a set of object frameworks and object systems confirming the hypothesis that the reliable identification of design pattern instances is both possible and feasible, providing high-level information about the design and implementation decisions.

■■■

Uvod

Vzorci načrtovanja so bili v svet programskega inženirstva vpeljani v delu [Gamma 1995] kot koncepti, ki povečajo razumevanje objektnih sistemov. Vsak vzorec načrtovanja vsebuje izkušnje in znanje strokovnjakov; predstavlja idejo rešitve problema pri načrtovanju sistemov, ki jo lahko enostavno in učinkovito uporabimo v različnih problemskih situacijah. Gledano bolj splošno, lahko vsak vzorec načrtovanja identificiramo kot arhitekturni element na višjem nivoju abstrakcije kot je nivo razredov [Domajnko 1998a]. Eden od osnovnih namenov vzorcev načrtovanja je, da so vzorci načrtovanja načrtovalski elementi, ki poenostavljajo aktivnost načrtovanja objektnih sistemov in dvigajo nivo komunikacije v razvojni skupini [Domajnko 1999a]. K temu dodaja tudi največkrat uporabljena predstavitev vzorcev, ki sestoji iz opisa problema, definicije ideje rešitve, lastnosti in omejitve rešitve, določitve terminologije in povezave na različne že uspešno izvedene rešitve s pomočjo njegove uporabe [Domajnko 1997].

Vendar pa lahko poenostavljeno razumevanje sistemov s pomočjo vzorcev načrtovanja dosežemo le, če dokumentacija objektnega sistema eksplicitno navaja uporabljene vzorce načrtovanja. Eden od razlogov, ki jih avtorji ugotavljajo, je ta, da s pomočjo vzorcev načrtovanja postopke vzdrževanja ali nadgraditve sistemov izvajamo na višjem nivoju abstrakcije, kar kažejo tudi rezultati v [Domajnko 1999b]. Dejansko pa

danes velika večina objektnih sistemov takšne dokumentacije ne vsebuje. Zato bi bilo zelo koristno, če bi lahko v obstoječi programski kodi identificirali primerke vzorcev načrtovanja.

Zato bomo v prispevku prikazali predstavitev vzorcev načrtovanja, ki omogoča razvoj podpornega okolja, ki lahko identificira vsak uporabljeni vzorec načrtovanja, implementiran skladno z definicijo vzorca načrtovanja. S tem pa tudi možnost, da dopolnimo dokumentacijo objektnih sistemov.

Predstavitev vzorcev načrtovanja

Na podlagi definicije vzorca načrtovanja iz [Gamma 1995]: »*Vzorec je ideja rešitve, ki se je izkazala uporabna v določenem kontekstu in bi bila najverjetneje koristna tudi v drugih kontekstih. Vzorec predstavlja večkratne odločitve strokovnjaka, ki sicer vodijo do različnih rešitev, a vsebujejo določen nivo kakovosti*« si poglejmo, kako so vzorci načrtovanja največkrat predstavljeni v literaturi [Gamma 1995, Buschmann 1996, Domajnko 1998b, Domajnko 1998c]. Tabela 1 prikazuje opisno predstavitev.

Tabela 1 Opisna predstavitev vzorcev

Sekcija	Pomen sekcije
Ime	Identifikator vzorca
Namen	Definicija problema, ki ga vzorec rešuje in rezultati uporabe vzorca
Uporabnost	Opis situacije, v kateri lahko vzorec uporabimo
Struktura	Praviloma razredni diagram, ki predstavlja statično sliko vzorca
Udeleženci	Seznam razredov, ki so vključeni v uporabo vzorca
Sodelovanja	Tekstoven opis dinamične komponente vzorca
Učinek	Podaja stanje ali konfiguracijo sistema po uporabi vzorca
Implementacija	Napotki za implementacijo vzorca
Pravila uporabe	Utemeljitev korakov uporabe vzorca
Sorodni vzorci	Opis statičnih in dinamičnih povezav med vzorci
Primeri uporabe	Opis uporabe vzorcev v obstoječih sistemih

Jasno je, da opisna predstavitev vzorcev ni primerna za računalniško podporo. To je vodilo k raziskavi razvoja jezika za opis vzorcev načrtovanja, ki bi dopolnil opis vzorcev načrtovanja in omogočil razvoj računalniške podpore. Jezik za specifikacijo vzorcev načrtovanja (poimenovali smo ga PatL) je bil zgrajen na podlagi opazovanj elementov, ki tvorijo vodilno misel kateregakoli vzorca načrtovanja. Jezik temelji na predstavitvi elementov modela abstraktne sintakse (model AST), s katero abstrahiramo implementacijske podrobnosti posameznih objektnih programskih jezikov.

Na podlagi definiranega modela AST in množice tipov udeležencev $\Re \equiv \{T_1, T_2 \ldots T_n\}$, množico relacij $\Re \equiv \{R_1, R_2 \ldots R_n\}$ in funkcijo M, ki preslika konstrukte modela AST v T in izraze AST v $\Re$, definiramo M(p) kot **model programa p v modelu AST**. Množico osnovnih konstruktov jezika PatL (element e) sestavljajo razredi (razred c) in metode (metoda m)[1]. Osnovne konstrukte združujemo v množice elementov enakega tipa - *enolične množice*. Med osnovnimi konstrukti so definirane *osnovne relacije*. Osnovne relacije smo definirali na podlagi opazovanja katalogov [Gamma 1995] in [Grand 1998]. Edina omejitev osnovnih relacij je, da morajo imeti kanonično implementacijo v več kot le enem objektnem jeziku. Tabela 2 podaja seznam osnovnih relacij med elementi, gradniki jezika PatL.

Tabela 2 Seznam osnovnih relacij

Osnovna relacija	Pomen relacije
razred (c)	delna, unarna relacija, ki izraža da je element *c* razred.
dedovanje (c_1, c_2)	relacija, ki izraža dedovanje med nadrazredom c_1 in podrazredom c_2.
metoda (m)	delna, unarna relacija, ki izraža da je element *m* metoda.
abstrakt (e)	relacija, ki določa, da je element *e* abstrakten.
definiran V (e, c)	relacija, ki izraža da je element *e* definiran v razredu *c*.
return Tip (m, c)	relacija, ki izraža da metoda *m* vrača objekte primerke razreda *c*.
konstruktor (m, c)	relacija, določa, da je metoda *m* konstruktor razreda *c*.
enakoime (m_1, m_2)	relacija, ki izraža, da sta imeni m_1 in m_2 enaki.
število argumentov (m)=n	relacija, ki definirana da ima metoda *m n* argumentov.
argument (n, m)=c	funkcijska relacija, ki za metodo *m* preslikuje zaporedno številko argumenta v tip argumenta.
proženje (m_1, m_2)	relacija, ki izraža, da v telesu metode m_1 (brez upoštevanja kontrolnih struktur) prihaja do proženja metode m_2.
enojnareferenca (c_1, c_2)	relacija, ki izraža, da obstaja objektna referenca v razredu c_1 do razreda razreda c_2 brez ločevanja med kompozicijsko, agregacijsko ali asociacijsko referenco.
multireferenca (c_1, c_2)	relacija, ki izraža, da razred c_1 v svoji definiciji vsebuje večštevno referenco do razreda c_2 brez ločevanja med kompozicijsko, agregacijsko ali asociacijsko referenco.
prirejanje (m, c_1, c_2)	relacija, ki izraža, da metoda *m* priredi referenco razreda c_1 do razreda c_2.
prirejanje po (m_1, m_2, c_1, c_2)	relacija, ki izraža, da metoda m_1 po zaključku izvajanja metode m_2 priredi referenco razreda c_1 do razreda c_2.

1 *Množico osnovnih konstruktov označimo s simbolom E, množico razredov s simbolom C, množico metod pa s simbolom* M.

Tabela 3 Seznam izpeljanih relacij

Izpeljana relacija	Definicija relacije
enak podpis(m_1, m_2)	relacija, ki izraža da imata metoda m_1 in m_2 enak podpis. $enakpodpis(m_1,m_2) \Leftrightarrow enakoime(m_1,m_2) \wedge stargumentov(m_1) = \text{stargumentov}(m_2) \wedge \prod_{i=0}^{i = stargumentov(m_1)} [\text{argument}(i,m_1) == \text{argument}(i,m_2)]$
posredovanje(m_1, m_2)	relacija, ki izraža da metoda m_1 proži metodo m_2, pri tem pa uporabi svoje argumente brez sprememb kot parametre metode m_2. $posredovanje(m_1,m_2) \Leftrightarrow proženje(m_1,m_2) \wedge enakpodpis(m_1,m_2)$
kreiranje(m_1, m_2)	relacija, ki izraža klic konstruktorske metode m_2 s strani metode m_1. $kreiranje(m_1,m_2) \Leftrightarrow proženje(m_1,m_2) \wedge konstruktor(m_2)$
produkcija(m_1, m_2, c)	relacija, ki izraža da metoda m_1 kliče metodo m_2, ta pa je konstruktor nekega razreda. Hkrati metoda m_1 tudi vrne kreiran objekt. $produkcija(m_1,m_2,c) \Leftrightarrow kreiranje(m_1,m_2) \wedge returntip(m_1,c)$

Na podlagi osnovnih relacij lahko definiramo množico izpeljanih relacij, ki imajo večjo izrazno moč, uporabljene osnovne relacije pa zagotavljajo njihovo implementacijo. Tabela 3 podaja seznam izpeljanih relacij in njihovih definicij.

Tabela 4 podaja seznam posplošitev osnovnih in izpeljanih relacij, ki dajejo matematično podlagi predstavitve vzorcev načrtovanja s pomočjo jezika PatL.

V modelu programa o imenujemo urejeno zaporedje udeležencev $\omega_1 \ldots \omega_N$ v p **kontekst**. S pomočjo kontekstov lahko opišemo strukturo poljubnega programa, zapisanega s pomočjo modela AST. Tudi opisna predstavitev vzorcev načrtovanja predstavlja vodilno misel vzorca načrtovanja (osnovno idejo) s pomočjo *udeležencev* (razredi, metode) in njihovega *sodelovanja*.

Če kontekst $\underline{\omega}$ *implementira* vzorec načrtovanja, bomo v kontekstu $\underline{\omega}$ zasledili elemente, ki implementirajo udeležence in sodelovanja, ki zagotavljajo implementacijo rešitve vzorca načrtovanja. Tako lahko definiramo, da je kontekst $\underline{\omega}$ posplošena oblika *primerka vzorca* π v programu p natanko takrat, če ima vsak $\omega_1 \ldots \omega_N$ iz $\underline{\omega}$ ustrezen tip (določen v π) in da $\omega_1 \ldots \omega_N$ sodelujejo, kot je določeno v specifikaciji vzorca π. Tako lahko vodilno misel (osnovno idejo) vzorca načrtovanja zapišemo v obliki *sheme*, kot jo definira [Computer 1990]. Shema predstavlja popolno in dokončno predstavitev vodilne misli posameznega vzorca in določa *udeležence* in *sodelovanja*. Skladno s to definicijo shema sestoji iz deklaracije *spremenljivk* (predstavljajo udeležence) in množice *predikatov*, ki izražajo relacije med spremenljivkami (izražajo sodelovanje).

V formulah uporabljamo relacije med strogo tipiziranimi spremenljivkami naslednjih tipov:

- Elementarne spremenljivke, ki predstavljajo osnovne konstrukte (razrede in metode).
- Spremenljivke višjih dimenzij, ki predstavljajo množice.
- Spremenljivke hierarhij, ki predstavljajo množico razredov v neki relaciji.

Slika 1 prikazuje dve alternativi prikaza predstavitvene sheme vzorcev načrtovanja (s pomočjo grafične sheme ali formule). V vsaki predstavitvi sheme najprej navedemo ime sheme (predstavlja

Tabela 4. Posplošitve osnovnih in izpeljanih relacij

Vrsta relacije	Definicija relacije
unarna relacija	relacija, definirana nad enim samim elementom v obliki parcialne funkcije. $r(X) \Leftrightarrow \forall x \in X : r(x)$
tranzitivna relacija	tranzitivna relacija je tranzitivno zaprtje osnovne relacije. $r^+(x,y) \Leftrightarrow r(x,y) \vee \exists z : \left[r^+(x,z) \wedge r(z,y)\right]$
totalna relacija	relacija med osnovnimi elementi ali množicami, ki prevzema obliko totalne funkcije. $r_T(X,Y) \Leftrightarrow \forall x \in X\ \exists y \in Y : r(x,y)$
popolna relacija	relacija, ki prevzema obliko izomorfne preslikave. $r_I(X,Y) \Leftrightarrow \forall x \in X\ \exists y \in Y : r(x,y) \wedge \forall y \in Y\ \exists x \in X : r(x,y)$
relacija med množicami	relacija med množicami se prevede na relacijo med elementi množic. $r(X,Y) \Leftrightarrow \forall x \in X\ \exists y \in Y : r(x,y)$ $r(x,Y) \Leftrightarrow r(\{x\},Y);\quad r(X,y) \Leftrightarrow r(X,\{y\})$
enakost relacij	enakost relacij določa, da sta relaciji r_1 in r_2 v množici S izomorfni. $enakost_{r_1,r_2}(S) \Leftrightarrow \forall p,q \in S : \left[r_1(p,q) \leftrightarrow r_2(p,q)\right]$ $enakost_{r_1,r_2}(X,Y) \Leftrightarrow \forall x \in X\ \forall y \in Y : \left[r_1(x,y) \leftrightarrow r_2(x,y)\right]$
hierarhija	relacija, ki določa množico razredov v hierarhiji dedovanja. $\text{hierarhija(h)} \Leftrightarrow \left[\begin{array}{l}\forall c_N \in h\ \exists c_0 \in h : \\ \quad \text{abstrakt}(c_0) \wedge \\ \quad \text{dedovanje}^+(c_N, c_0)\end{array}\right]$ **Relacijo *r*** nad hierarhijo $h_i \in H$ če r ni popolna relacija, definiramo kot: $r(h_i,S) \Leftrightarrow r(koren(h_i),S)$ $r(S,h_i) \Leftrightarrow r(S,listi(H_i))$ $r(h_1,h_2) \Leftrightarrow r(listi(h_1),koren(h_2))$ **Relacijo r** nad hierarhijo $h_i \in H$, kjer je je *r* popolna relacija definiramo kot: $r_P(h_i,S) \Leftrightarrow r_P(listi(h_i) \cup koren(h_i),S)$ $r_P(S,h_i) \Leftrightarrow r_P(S,listi(h_i) \cup koren(h_i))$ $r_P(h_1,h_2) \Leftrightarrow r_P(listi(h_1) \cup koren(h_1),listi(h_2) \cup koren(h_2))$
družina	je enolična množica metod z enakim podpisom, ki so definirane v množici razredov *C*. $družina(M,C) \equiv M_C^d \subseteq \left\{\begin{array}{l}M_i \mid \exists M_j : M_i \subseteq M \wedge M_j \subseteq M \\ enakvmesnik(M_i,M_j) \wedge \\ definiranv(C,M_i) \wedge definiranv(C,M_j)\end{array}\right\}$
rod	je enolična množica družin. Množica množic metod $\{F_i\}$ je *rod* v množici razredov C, tedaj in le tedaj, če je vsaka metoda $F \in \{F_i\}$ družina v podmnožici. $C\,M_C^r \Leftrightarrow \left\{M_{X_i}^d : X_i \subseteq C\right\}$

Shema

$\exists\ (x_1, x_2, \ldots x_n)$ spremenljivke

$\prod_i \Re_j(y_{i1}, y_{i2}, \ldots y_{i_n})$ predikati, povezani z operatorjem konjunkcije

$\text{shema} \equiv (x_1, x_2 \ldots x_n) \mapsto \prod_i R_i(x_1, x_2 \ldots x_m)$

Slika 1 Dve alternativni predstavitvi sheme vzorcev načrtovanja

enolično ime vzorca načrtovanja), nato navedemo seznam spremenljivk (določimo udeležence in pri tem upoštevamo strogo tipiziranost) in na koncu še množico predikatov (definiramo sodelovanje udeležencev).

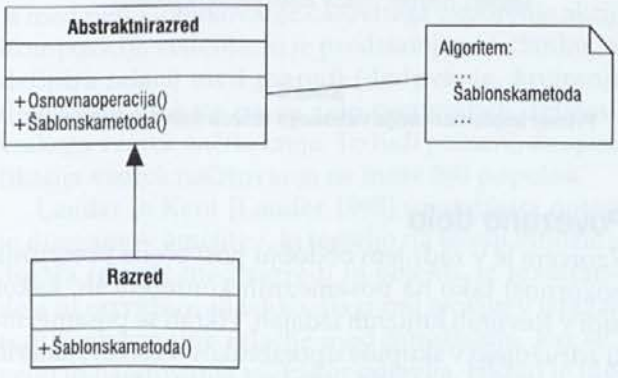
Nad shemami, ki specificirajo posamezne vzorce načrtovanja, definiramo operacije. Operacije predstavljajo uporabo in posplošitev operacij iz [IEEE 1990/5].

Tabela 5 Seznam operacij nad shemami

Operacija	Pomen operacije nad shemo
Preimenovanje komponent	zagotavlja oblikovanje nove sheme na podlagi obstoječe s pomočjo sistematičnega preimenovanja komponent. *Novashema/Osnovna [(Novakomponenta/Starakomponenta)*]*
Dekoracija sheme	zagotavlja ločevanje med spremenljivkami in predikati pred in po izvedbi določene akcije. *komponenta* - označuje komponento pred izvedbo akcije. *komponenta'* - označuje dekorirano komponento.
Vključitev sheme	zagotavlja možnost ponovne uporabe posameznih shem v kontekstu deklaracij drugih shem. Vključitev sheme u v shemo y definiramo kot[1]: $\psi \equiv (\phi_1, \phi_2 ... \phi_N) \mapsto \prod_i R_i(\phi_A ... \phi_B)$ $\upsilon \equiv (\alpha_1, \alpha_2 ... \alpha_M) \mapsto \prod_k R_k(\alpha_U ... \alpha_V)$ $\psi \oplus \upsilon \Leftrightarrow (\alpha_1, \alpha_2 .. \alpha_M, \phi_1, \phi_2 ... \phi_N) \mapsto \prod_k R_k(\alpha_U ... \alpha_V) \wedge \prod_i R_i(\phi_A ... \phi_B)$
Konjunkcija sheme	konjunkcijo dveh shem formiramo z združitvijo njunih signatur, identificiranjem skupnih spremenljivk (katerih tipi se morajo ujemati) in združitvijo predikatov. Shema ≡ ShemaA ∧ ShemaB
Disjunkcija sheme	disjunkcijo shem formiramo z združitvijo njunih signatur: identificiranjem skupnih spremenljivk (tipi se morajo ujemati) in razdruženjem njunih predikatov. Shema ≡ ShemaA ∨ ShemaB
Negacija sheme	negacijo sheme formiramo tako, da ne spreminjamo identificiranih spremenljivk, negiramo pa vse predike. Shema ≡ ¬ ShemaA
Skrivanje sheme	predstavlja mehanizem jezika shem, ki poenostavlja specifikacije. ShemaA \ (komponenta)*
Kompozicija shem	označuje relacijsko kompozicijo dveh shem. Obe shemi morata zadostiti pogojem spremenljivk v stanju po operaciji. Shema ≡ ShemaA • ShemaB

Tabela 5 prikazuje seznam operacij nad shemami.

S pomočjo zgrajene osnove jezika PatL si sedaj poglejmo primera definicij poznanih vzorcev načrtovanja. Vedenjski vzorec načrtovanja ***Šablonskametoda*** definira ogrodje algoritma v operaciji z odlaganjem nekaterih korakov v podrazrede. Vzorec dovoljuje podrazredom ponovno definirati nekatere korake algoritma, brez da bi spreminjali strukturo algoritma.



Slika 2 Razredni diagram vzorca Šablonskametoda

Slika 2 prikazuje razredni diagram vzorca. Največkrat se vzorec uporabi pri izgradnji fleksibilnih algoritmov, kjer je s pomočjo drugačne implementacije primitivnih metod moč spremeniti potek algoritma.

Slika 3 prikazuje formalno predstavitev vzorca s pomočjo jezika PatL. Osnovna razlika so dodane informacije:

- proženje metode, ki je bilo prej izraženo s pomočjo naravnega jezika, temelji sedaj na relaciji, ki ima kanonično predstavitev v objektnih jezikih,
- relacija dedovanja med razredom Abstraktnirazred in razredom Razred ni eksplicitno izražena, ker ni nujno potrebna (vpeljava koncepta vmesnika),
- proženje šablonske metode s strani osnovne operacije ni nujno neposredno, ampak je lahko posredno preko kateregakoli tranzitivnega zaprtja relacije,

2 *Vključitev sheme vsebuje še pogoj, da pri vključevanju v shemah ne obstajajo enako poimenovane komponente. V primeru, da pogoj ni izpolnjen, je potrebno komponente preimenovati.*

- definicija vzorca dopušča tudi predifiniranje osnovnih operacij, saj le te po definiciji tvorijo družino znotraj hierarhije razredov,
- razred Razred je element hierarhije razredov, njegov položaj pa je popolnoma nepomemben.
- definicija vzorca dopušča tudi definicijo več kot le ene šablonske metode in več kot le eno implementacijo le te znotraj hierarhije razredov.

Vzorec: Šablonskametoda

Šablonskametoda $\in$ M

Osnovnaoperacija $\in \{M_H^d\}$,

Razred $\in$ H

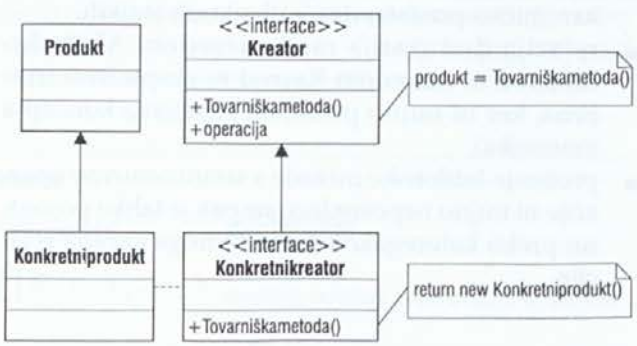
družina(Osnovnaoperacija, Razred) $\wedge$
proženje$_T$(Šablonskametoda, Osnovnaoperacija) $\wedge$
definiranv$_T$(Šablonskametoda, Razred)

Slika 3 Predstavitev vzorca Šablonskametoda v jeziku PatL

Na podlagi podanih ugotovitev lahko zaključimo, da nove informacije bolj natančno specificirajo osnovno vodilo vzorca načrtovanja (oziroma, dopolnjujejo opisno definicijo vzorca načrtovanja).

Kreatorski vzorec *Tovarniškametoda* definira vmesnik za kreiranje objekta, vendar dopušča podrazredom, da določijo, iz katerega razreda bodo izhajali. Vzorec dovoljuje, da razred odloži instanciranje na podrazrede. Slika 4 prikazuje razredni diagram vzorca, kjer lahko opazimo, da s pomočjo naravnega jezika definiramo dinamično komponento vzorca.

Slika 5 prikazuje formalno definicijo vzorca, kjer smo poleg strukture razredov (sedaj povezanih s hierarhijo in ne v neposredno dedovanje) določili tudi dinamično komponento ideje vzorca. S tem smo dodali informacije, ki lahko v veliki meri vplivajo na uporabnost samega vzorca pri rutinski uporabi.



Slika 4 Razredni diagram vzorca Tovarniškametoda

Vzorec: Tovarniškametoda

Tovarniškametoda $\{M_{H_1}^d\}$,

Kreatorji $\in H_1$,

Produkti $\in H_2$

družina(Tovarniškametoda, *Kreatorji*) $\wedge$
produkcija$_t$(Tovarniškametoda, *Produkti*) $\wedge$
returnTip$_t$(Tovarniškametoda, *Produkti*) $\wedge$
enakost $_{returnTip\ produkcija}$(Tovarniškametoda, *Produkti*)

Slika 5 Razredni diagram vzorca Tovarniškametoda

V nadaljevanju si poglejmo še enostavnost implementacije validacije vzorca. Slika 6 prikazuje implementacijo validacije vzorca Šablonskametoda, ki smo ga prej formalno definirali. Kot je moč videti iz primera, je implementacija validacije popolnoma skladna z definicijo vzorca. To kaže tudi na dejstvo, da je moč generiranje pravil za validacijo avtomatizirati s pomočjo enostavnega orodja, ki formalno specifikacijo pretvori v sintakso implementacijskega okolja (v našem primeru ekspertne lupine CLIPS in razširka JESS), samo okolje po z vgrajenimi algoritmi poskrbi za izvajanje validacije. V primeru na sliki sicer v bazo dejstev samo dodamo dejstvo, da smo našli primerek vzorca Šablonskametoda, podporno okolje pa mora dodati informacije, ki so potrebne za popolno validacijo vzorca (vsaj udeležence in njihove vloge).

```
(defrule validirajŠablonskametoda
        (razred (naziv ?razred))
        (metoda (naziv ?Šablonskametoda))
        (metoda (naziv ?Osnovnaoperacija))
        (definiranV (razred (naziv ?razred) (metoda (naziv
?Šablonskametoda)))
        (družina (metoda (naziv ?Šablonskametoda) (razred
(naziv ?razred))))
        (proženje (metoda (naziv ?Osnovnaoperacija)) (meto-
da (naziv ?Šablonskametoda)))
 =>
        (assert (vzorec (naziv Šablonskametoda)))
)
```

Slika 6
Primer implementacije validacije vzorca Šablonskametoda

Povezano delo

Vzorcem je v zadnjem obdobju posvečena precejšnja pozornost tako na posameznih konferencah, kakor tudi v številnih knjižnih izdajah. Hkrati se posamezniki združujejo v skupine uporabnikov vzorcev, katerih primarni cilj je identifikacija in zapis novih vzorcev in nadzor nad obstoječimi vzorci.

Vzorci programskega inženirstva, še posebej *vzorci načrtovanja*, so največkrat objavljeni v obliki *katalogov*, ki predstavljajo delo, v katerem so posamezni elementi neodvisni in nepovezani. Serija knjig PloPD (Pattern Languages of Program Design) predstavljajo zbirko vzorcev različnih problemskih domen in formatov, povzetih po PloP (Pattern Languages of Programming) konferencah [PloP 1995, PloP 1996, PloP 1997, PloP 1998]. Prispevke različnih avtorjev lahko razvrstimo v različne domene, kot so *vzorci načrtovanja, specializacija obstoječih vzorcev, domensko specifični vzorci, organizacijski* in *upravljalski vzorci* ter *jezikovno odvisni vzorci*.

Trenutno stanje je tako, da obstaja v svetu cela vrsta predlaganih predstavitev vzorcev v opisni obliki, kjer avtorji vzorce opisujejo z vnaprej določeno množico lastnosti, ki onemogočajo formaliziranje uporabe vzorcev. Enega prvih poskusov formalizacije predstavitev vzorcev predstavlja model LayOM [Bosch 1996], ki s pomočjo razširitve klasičnega objektnega modela s konceptoma *stanje* (state), *nivo* (layer) zagotavlja predstavitev vzorcev načrtovanja. Izraznost tako nastalega specifikacijskega jezika je prikazana z določanjem omejitev reakcij objektov in protokola sodelovanja objektov. Vendar pa tak pristop omejuje pravila na posamezne objekte, kar zagotavlja podporo omejenemu številu primerov, kjer ni potreben nadzor na višjem nivoju. Pristop zato ni primeren kot osnova za abstrahiranje načrtovalskih odločitev.

V [Ducasse 1995] avtorji definirajo nivo abstraktnega pošiljanja sporočil med objekti, kjer je nivo v obliki konektorjev sposoben prestreganja sporočil enega objekta drugemu in s tem zagotavljati neposredno specifikacijo in implementacijo *"izvedljivih konektorjev"*. Avtorji predstavijo tudi pristop k specifikaciji vzorcev načrtovanja, vendar sami podajajo omejitve pristopa, ki je primerna samo za specifikacijo dinamičnih lastnosti vzorcev, medtem ko strukurnih lastnosti ni moč določiti.

Helm, Holland in Gangopadhyay v [Helm 90] definirajo pojem *pogodba*. Pogodba predstavlja razširitev predikatne logike prvega reda z možnostjo predstavitve funkcijskih klicev, prirejanj in relacij urejanja med njimi (oblikovanje časovnega zaporedja akcij). Kompozicija vedenja, ki je predstavljena v članku, ne definira relacij med razredi (dedovanje, kreiranje, delegacija itd.), te pa so zelo pomemben dejavnik vsakega vzorca načrtovanja. To tudi pomeni, da specifikacija vzorca načrtovanja ne more biti popolna.

Lauder in Kent [Lauder 1998] uporabljata notacijo diagramov omejitev, ki temeljo na teoriji množic in določa relacije med razredi in objekti. Iz rezultatov dela ni razvidno, ali lahko diagrami omejitev izražajo tudi funkcionalne relacije med množicami, kar se v vzorcih načrtovanja vsekakor pojavlja. Hkrati je nivo abstrakcije diagramov omejitev prenizek (oziroma, diagrami so preveč podrobni) za specifikacijo zelo kompleksnih vzorcev načrtovanja, oziroma za iskanje načrtovalskih odločitev.

Budinsky, Finnie in Yu v [Budinsky 1996] predstavljajo orodje, ki podpira aplikacijo vzorcev načrtovanja in generiranje izvorne kode. Razvijalec določa postopek generacije izvorne kode s posebnim skriptnim jezikom COGENT, vendar orodje ne podpira poti nazaj – iz izvorne kode ni moč ugotoviti, kateri vzorci so bili uporabljeni. Precej podobno Quintessoft Inc v [Quintessoft 1997] opisuje orodje CASE, ki je sposobno generirati izvorno kodo v jeziku C++. Enako kot v prejšnjem primeru orodje generira izvorno kodo vzorca načrtovanja, ki je ločena od ostalega sistema, povratnih informacij pa orodj ni možno oblikovati iz obstoječe programske kode.

Kljub temu, da so danes vzorci načrtovanja konstrukti, ki so že nekaj let poznani razvijalcem programskih sistemov, pa so raziskave na področju identifikacije vzorcev načrtovanja v obstoječih sistemih precej redke. Kljub številnim raziskavam na področju povratnega inženirstva pa je namen raziskave drugačen. V raziskavi namreč ne gre za podrobno razumevanje implementacije objektnega sistema, ampak le za ekstrakcijo visokonivojskih informacij iz obstoječe kode.

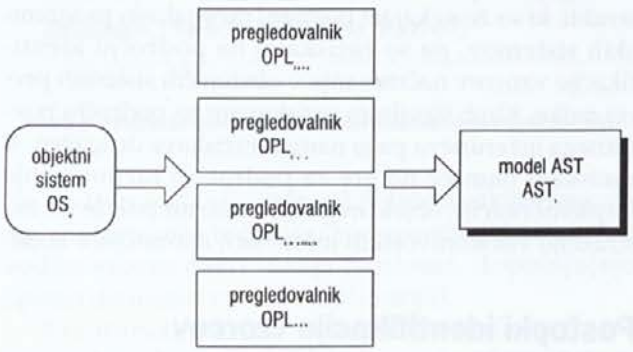
Postopki identifikacije vzorcev načrtovanja

Model predstavitve vzorcev načrtovanja zagotavlja enolično in nedvoumno predstavitev vzorcev. S pomočjo modela abstraktne sintakse (AST) lahko zagotovimo neodvisnost od programskega jezika (vse dokler lahko s pomočjo pregledovalnika izvorne kode generiramo ustrezno predstavitev objektnega sistema s pomočjo modela AST). S pomočjo jezika PatL lahko vzorce načrtovanja rigorozno definiramo. Na podlagi ugotovitev lahko definiramo naslednje aktivnosti:

- **Aplikacija vzorca** je proces, kjer je vzorec π apliciramo v kontekst $\underline{\omega}$ in konkretni elementi $\omega_1 \ldots \omega_N$ prevzamejo vloge sodelujočih iz specifikacije vzorca p. To pomeni, da se posamezni elementi spremenijo na tak način, da zadostijo zahtevam specifikacije vzorca π. Največkrat aplikacije vzorca ne izvajamo v neinicializiranem okolju temveč v določenem kontekstu. To pomeni, da prilagajamo in dopolnjujemo konstrukte, ki že obstajajo v kontekstu $\underline{\omega}$. V danem kontekstu $\underline{\omega}$ bomo večkrat identificirali, da posamezni sodelujoči specifikacije vzorca nimajo dejanskega elementa. Takrat pravimo, da je $\underline{\omega}$ *nepopolni kontekst* in implicira, da je potrebno manjkajoče vloge specifikacije vzorca kreirati. V takem primeru aplikacija vzorca rezultira v novih konstruktih, ki zadoščajo pogojem, podanih v specifikaciji vzorca načrtovanja.

- **Validacija vzorca** je proces, ki za podan kontekst ω v programu p in podan vzorec π odgovori na vprašanje ali je ω primerek vzorca π.
- **Identifikacija vzorca** je proces, ki za podan kontekst ω v programu p poda seznam validacij vzorcev π_i v kontekstu ω.
- **Identifikacija relacij med vzorci** je aktivnost, s pomočjo katere ugotavljamo binarne relacije med vzorci. Relacije praviloma klasificirajo vzorce.

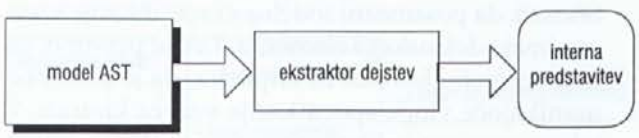
Slika 7 prikazuje prvi korak pri identifikaciji vzorcev načrtovanja, ki so bili uporabljeni pri načrtovanju (in implementaciji) objektnega sistema. S pomočjo razmeroma enostavnega pregledovalnika izvorne kode generiramo modela programa z uporabo modela AST.



Slika 7 Proces kreiranja modela objektnega sistema

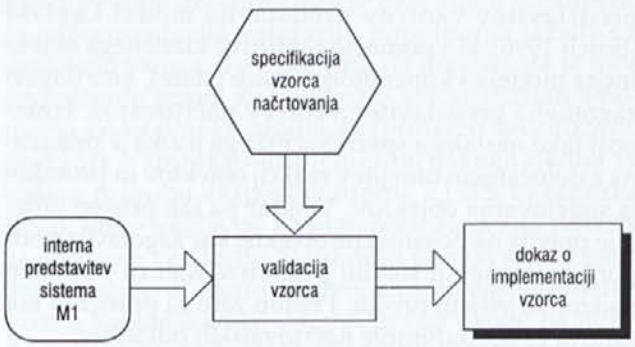
Definirani model AST lahko uporabimo za predstavitev poljubnega objektnega sistema, implementiranega v objektnem programskem jeziku, za katerega smo razvili pregledovalnik, ki ekstrahira primerne informacije. Objektni sistem, predstavljen s pomočjo modela AST v naslednjem koraku s pomočjo ekstraktorja dejstev pretvorimo v interno predstavitev, ki je skladna z jezikom PatL (interna predstavitev je odvisna od izbranega okolja, v naši raziskavi smo uporabili ekspertno lupino CLIPS, ki temelji na ogrodjih in njeno Java razširitev JESS). Zato ekstraktor dejstev na podlagi modela programa in definiranih relacij oblikuje bazo znanja v ekspertni lupini.

V tem trenutku lahko izvedemo validacijo vzorca načrtovanja. S pomočjo formalne specifikacije vzorcev



Slika 8 Proces oblikovanja interne predstavitve

in uporabe interne predstavitve lahko proces validacije izvedemo na preprost način s pomočjo enostavnih pravil, katere implementiramo v ekspertni lupini (definicija vsakega izmed vzorcev predstavlja iskalni kriterij, ki mu ekspertna lupina s pomočjo vgrajenih algoritmov poskuša zadostiti. Slika 9 prikazuje osnovo procesa. Proces validacije vzorca je deterministične in voden s principi boolove logike – ali vzorec je implementiran v objektnem sistemu ali pa ni. To je tudi edina omejitev pristopa, saj tak pristop ne zagotavlja identifikacije vzorcev, ki so skoraj v popolnosti implementirani v programski kodi. Če obstaja razlika med specifikacijo in dejansko implementacijo vzorca, proces identifikacije takšne implementacije vzorcev ne razpozna.



Slika 9 Proces validacije vzorca načrtovanja

Proces identifikacije vzorcev v objektnem sistemu predstavlja le uporabo validacije posameznega vzorca v objektnem sistemu. Prikazuje proces, kjer iz interne predstavitve poljubnega objektnega sistema pridobimo informacije o vzorcih, ki so implementirani v sistemu.

Relacije med vzorci načrtovanja

Kot smo zapisali, je klasifikacija vzorcev eden od predpogojev, da bomo tudi v času, ko bo število vzorcev načrtovanja narastlo, lahko s pridom in učinkovito izkoriščali vgrajeno strokovno znanje. S pomočjo formalne definicije vzorcev postane proces identifikacije relacij med vzorci enostaven in deterministične.

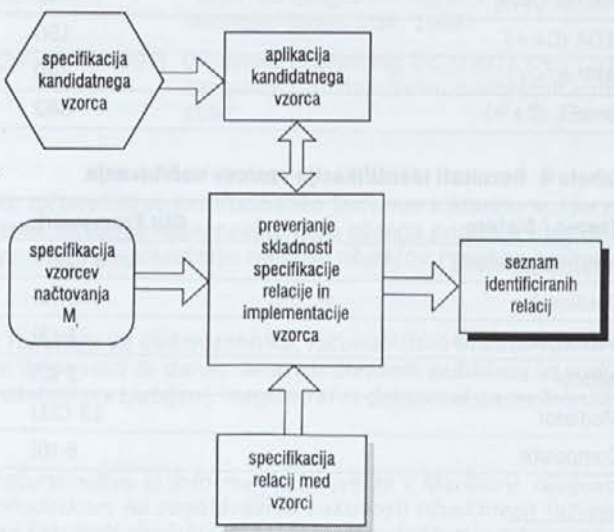
Enako kot smo definirali posamezne vzorce načrtovanja, lahko definiramo tudi množico relacij med vzorci in z enakim postopkom, kot smo izvajali validacijo vzorca načrtovanja v objektnem sistemu, lahko izvedemo tudi primerjavo vzorcev načrtovanja med seboj. Tabela 6 prikazuje definicijo nekaterih najpomembnejših relacij med vzorci načrtovanja.

Tabela 6 Opisna predstavitev vzorcev

Relacija med vzorci	Pomen relacije
specializira(π,σ)	izraža, da vzorec π specializira vzorec σ. $\pi \equiv (\pi_1, \pi_2 ... \pi_N) \mapsto \prod_j R_j(\pi_{A_j} ... \pi_{B_j})$ $\sigma \equiv (\sigma_1, \sigma_2 ... \sigma_N) \mapsto \prod_i R_i(\phi_{A_i} ... \phi_{B_i})$ $specializira(\pi, \sigma) \Leftrightarrow$ $\exists \zeta : \zeta / \sigma[\sigma_1 / \pi_1, \sigma_2 / \pi_2 ... \sigma_N / \pi_N] \wedge \{R_\zeta\} \subseteq \{R_\pi\}$
uporablja (π, σ)	izraža, da vzorec π uporablja vzorec σ. $\pi \equiv (\pi_1, \pi_2 ... \phi_N) \mapsto \prod_j R_{\pi j}(\pi_{A_j} ... \pi_{B_j})$ $\sigma \equiv (\sigma_1, \sigma_2 ... \sigma_M) \mapsto \prod_i R_{\sigma i}(\phi_{U_i} ... \phi_{V_i})$ $uporablja(\pi, \sigma) \Leftrightarrow \forall R_{\sigma i}(\sigma_{U_i} ... \sigma_{V_i})$ $\exists \zeta_{\sigma i} : \zeta_{\sigma i} / \sigma[(\sigma_{O_i} / \pi_{P_i})*] \wedge$ $R_{\zeta i}(\pi_{U_i}, \pi_{V_i}) \equiv R_{\pi k}(\pi_{U_i}, \pi_{U_i})$
specializiranv (π, σ)	izraža, da je specifikacija vzorca π specializirana v specifikaciji vzorca σ. $specializiranv(\pi, \sigma) \Leftrightarrow specializacija(\sigma, \pi)$
nivo razlikovanja $(\pi, \sigma) \mapsto \mathbf{N}\{\cup\}$	je parcialna funkcija, ki za par vzorcev <π, σ> vrne razdaljo razlikovanja. $\pi \equiv (\pi_1, \pi_2 ... \phi_N) \mapsto \prod_j R_{\pi j}(\pi_{A_j} ... \pi_{B_j})$ $\sigma \equiv (\sigma_1, \sigma_2 ... \sigma_M) \mapsto \prod_i R_{\sigma i}(\phi_{U_i} ... \phi_{V_i})$ $nivorazlikovanja(\pi, \sigma) \equiv$ $\max\left(\left\|\left\{R_{\zeta / \sigma[(\pi_M / \sigma_N)*]}\right\} - \{R_\pi\}\right\|, \left\|\{R_\pi\} - \left\{R_{\zeta / \sigma[(\pi_M / \sigma_N)*]}\right\}\right\|\right)$

Slika 10 prikazuje proces ugotavljanja relacij med vzorci. Proces izvedemo s pomočjo aktivnosti aplikacije (uporabe vzorca načrtovanja) v prazen kontekst in aktivnosti identifikacije vzorcev v nastalem kontekstu.

Kot lahko vidimo, gre v resnici za aplikacijo kandidatnega vzorca v prazen kontekst. S tem dobimo minimalno implementacijo vzorca in to implementacijo lahko nato preverimo s pomočjo enostavnih pravil – ali zadošča kriterijem kakšne izmed definiranih relacij. Proces aplikacije in preverjanja skladnosti med specifikacijo in implementacijo smo opisali v prejšnjih poglavjih.



Slika 10. Proces identifikacije relacij med vzorci načrtovanja

Rezultati analize objektnih ogrodij

Kot osnovo za preverjanje pravilnosti modela in postopka identifikacije vzorcev smo uporabili več različnih

razrednih knjižnic in objektnih sistemov različne obsega. Zaradi omejenega prostora bomo prikazali rezultate le nekaj vzorcev načrtovanja. Tabela 7 prikazuje seznam objektnih sistemov in programskih jezikov, v katerih so bili implementirani. Za prikaz velikosti sistemov je v tabelo dodanih še nekaj metrik (število razredov, skupno število podatkovnih atributov, skupno število metod, skupno število vseh relacij in število vseh dedovanj). Tabela 8 prikazuje rezultate identifikacije vzorcev načrtovanja, izvedenega s pomočjo podpornega okolja.

Kot lahko razberemo iz rezultatov, je dejansko število uporabljenih vzorcev precej majhno. Razlogov za to je več, glavni pa ta, da je veliko implementacij vzorcev nepopolnih. Na to kaže tudi dejstvo, da smo, ko smo dovolili majhno odstopanje pri implementaciji vzorca, našli večje število primerkov vzorcev (pri tem se moramo zavedati, da nismo ločevali med vrsto neskladja med definicijo in implementacijo vzorca, kar je lahko v nekaterih primerih privedlo do napačne interpretacije). Ti rezultati so v tabeli navedeni v oklepajih. Drugi razlog pa je ta, da so bili trije sistemi (LEDA, zAPP in jamaEL) zasnovani in implementirani v času, ko vzorci še niso bili splošno poznani. Zato so načrtovalci in implementatorji uporabljali svoje rešitve, ki pa vedno niso enake rešitvam, ki jih ponujajo vzorci načrtovanja. Edini sistem, ki je bil v veliki meri zasnovan s pomočjo vzorcev, je samo podporno okolje PatTool, ki izkazuje tudi največje število identificiranih primerkov vzorcev načrtovanja.

Vendarle pa rezultati analize te množice objektnih sistemov kažejo, da je s pomočjo formalne definicije vzorcev načrtovanja možno implementirati podporno okolje, ki je sposobno ekstrahirati visokonivojske podatke iz same programske kode.

Zaključek

Kot kažejo empirične raziskave tako v industrijskem kot tudi akademskem okolju, so informacije o uporabljenih vzorcih načrtovanja lahko pomembne za proces vzdrževanja in nadgradnje objektnih sistemov, saj omogočajo višjenivojski pogled na objektni sistem, z vgrajenim znanjem pa pomagajo graditi robustnejše in bolj prilagodljive sisteme.

Osnovna naloga raziskave, razvoj jezika za formalni opis vzorcev načrtovanja in njegova uporaba v procesu ekstrakcije visokonivojski podatkov iz obstoječe programske kode, je bila v celoti izpolnjena. Hkrati pa smo ugotovili še nekatere druge prednosti formalne definicije vzorcev, med katerimi je vsekakor najbolj pomembna možnost formalne definicije in deterministične identifikacije relacij med vzorci načrtovanja. Le-ta namreč zagotavlja trden temelj klasifikaciji vzorcev načrtovanja. Le s trdno in enolično določeno klasifikacijo se je namreč moč upreti vse večjemu številu zapisanih vzorcev in se izogniti redundanci in prekrivanju znanja

Kot je bilo prikazano v prispevku, je deterministična identifikacija vzorcev načrtovanja v obstoječi

Tabela 7 Seznam uporabljenih objektnih sistemov

Sistem / Merila	Razredi	Atributi	Metode	Relacije	Dedovanja
GUI Framework (Java)	45	187	3652	118	23
PatTool (Java)	72	67	2456	74	8
LEDA (C++)	150	501	4084	242	67
zAPP (C++)	240	1176	3590	298	145
jamaEL (C++)	382	2523	3845	723	212

Tabela 8 Rezultati identifikacije vzorcev načrtovanja

Vzorec / Sistem	GUI Framework	PatTool	LEDA	zAPP	jamaEL
Tovarniškametoda	9 (12)[1]	34 (34)	12 (16)	0 (14)	0 (0)
Obiskovalec	0 (0)	1 (2)	0 (0)	0 (0)	2 (2)
Šablonskametoda	1 (3)	4 (4)	0 (6)	2 (2)	6 (6)
Iterator	2 (2)	19 (19)	5 (9)	4 (8)	2 (3)
Mediator	13 (21)	2 (2)	0 (4)	0 (10)	0 (0)
Composite	6 (6)	0 (0)	4 (5)	2 (2)	4 (4)

3 *Prva števila pomeni število validiranih primerkov vzorca v programski kodi, medtem ko druga številka pomeni število validiranih vzorcev v programski kodi, kjer je nivo razlikovanja manjši kot 3.*

programski kodi možna. S pomočjo formalne predstavitve vzorcev načrtovanja je implementacija okolja, ki izkorišča prednosti uporabe vzorcev v razvoju, enostavna in učinkovita.

V sklopu nadaljnjega dela je vsekakor potrebno pomisliti na dejstvo, da je danes na voljo vedno več dokumentacije o razvoju sistema, ki bi jo lahko uporabili tudi za ekstrakcijo informacij o uporabljenih vzorcih načrtovanja. Na tem področju je zelo obetaven jezik XMI kot komunikacijski standard med CASE in razvojnimi orodji, ki s svojo izrazno močjo v veliki meri poenostavlja ekstrakcijo podatkov iz posameznega programskega jezika v obliko, potrebno za podporno okolje. Poleg tega bo naše nadaljnje delo usmerjeno k razvoju okolja, ki bi podpiralo grafično uporabo vzorcev načrtovanja. To bi po naši presoji prineslo še večjo učinkovitost uporabe vzorcev v proces razvoja in s tem pripomoglo h kakovostnejšemu razvoju programske opreme in še posebej kompleksnih objektnih sistemov.

Literatura

[Bosch 1996] Bosch J. "Language Support for Design Patterns. Proceedings TOOLS Europe '96, 1996

[Budinsky 1996] Budinsky F. J., M. A. Finnie, J. M. Vlissides, and P. S. Yu (1996). "Automatic code generation from design patterns". Object Technology Vol. 35, No. 2, 1996

[Computer 1990] Formal Methods, Computer, Vol. 23, No. 9, 1990

[Domajnko 1997] DOMAJNKO Tomaž, KOREN Silvo, CELCER Borut, DRVARIČ Ivan, "Uporaba objektnega pristopa na primeru izgradnje informacijske podpore poslovne funkcije v zavarvalništvu, zbornik srečanja Objektna tehnologija v Sloveniji OTS'97, 1997

[Domajnko 1998a] DOMAJNKO, Tomaž, ROZMAN, Ivan, HERIČKO, Marjan, GYÖRKÖS, József. "Vzorci in ponovna uporaba izkušenj", Uporabna informatika (Ljubljana), 1998

[Domajnko 1998b] DOMAJNKO, Tomaž, JURIČ, Branko-Matjaž, HERIČKO, Marjan, ROZMAN, Ivan, "Vzorci – sedanjost ali prihodnost?", Dnevi slovenske informatike, Portorož, 6. – 9. maj 1998. Zbornik posvetovanja, 1998

[Domajnko 1998c] Domajnko Tomaž, Rozman Ivan, Heričko Marjan, Jurič Branko-Matjaž, Beloglavec Simon, "Java in vzorci", Fakulteta za elektrotehniko, računalništvo in informatiko Maribor, Inštitut za informatiko, Center za objektno tehnologijo, 1998

[Domajnko 1999a] DOMAJNKO, Tomaž, JURIČ, Branko-Matjaž, HERIČKO, Marjan, ROZMAN, Ivan, "Formal based design patterns notation and support environment", 10th International Conference Information and intelligent systems IIS '99, Varaždin, 1999

[Domajnko 1999b] DOMAJNKO, Tomaž, JURIČ, Branko-Matjaž, BELOGLAVEC, Simon, ROZMAN, Ivan, "Design patterns management framework", Applied informatics : proceedings of the Seventeenth IASTED International Conference, Innsbruck, Austria, 1999

[Ducasse 1995] Ducasse Stephane, Mireille Blay-Fornarino, Anne-Marie Pinna, "A REflective Model for First Class Dependencies", Tools 1996, NY, 1995

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns - Elements of Reusable Object- Oriented Software, Addison-Wesley, Reading, MA, 1995.

[Grand 1998] M. Grand, Patterns in Java – volume 1, Wiley, 1998

[Helm 90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object- Oriented Systems", OOPSLA '90 Conference Proceedings, Ottawa, Canada, 1990

[IEEE 1990/5] Formal Methods, IEEE Software, Vol. 7, No. 5, 1990

[PloP 1996] Pattern Languages of Programs, Allerton Park, Monticello-Illinois, USA, 1996

[PloP 1997] Pattern Languages of Programs, Allerton Park, Monticello-Illinois, USA, 1997

[PloP 1998] Pattern Languages of Programs, Allerton Park, Monticello-Illinois, USA, 1998

[PloP 1999] Pattern Languages of Programs, Allerton Park, Monticello-Illinois, USA, 1999

[*Quinressoft 1997*] *Quintessoft Engineering, Inc. (1997). C++ Code Navigator 1.1.* http://www.quintessoft.com, *1997*

◆

Tomaž Domajnko je raziskovalec na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru, kjer je vpisan v doktorski študij na področju računalništva in informatike. Njegovo raziskovalno delo obsega področje objektne tehnologije s poudarkom na izkoriščanju potenciala ponovne uporabe in zagotavljanju trajnosti objektov. Posebno področje njegovih raziskav so vzorci v programskem inženirstvu.

◆

Dr. Ivan Rozman je redni profesor Univerze v Mariboru, dekan Fakultete za elektrotehniko, računalništvo in informatiko v Mariboru in ustanovitelj Laboratorija za informacijske sisteme, ki ga vodi še danes. Je avtor številnih publikacij in vodja več raziskovalnih projektov. Diplomiral je na Fakulteti za elektrotehniko v Ljubljani, magistriral in doktoriral pa na Tehniški fakulteti v Mariboru.

◆

Dr. Marjan Heričko je docent na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Njegovo raziskovalno delo obsega vse vidike objektne tehnologije s poudarkom na zagotavljanju kakovosti objektnega razvoja programskih sistemov. Diplomiral, magistriral in doktoriral je na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru.

ZLEPKI V OBJEKTNO USMERJENEM OKOLJU

Mojca Indihar Štemberger, Janez Grad
Univerza v Ljubljani, Ekonomska fakulteta,
Kardeljeva ploščad 17, 1000 Ljubljana
mojca.stemberger@uni-lj.si, janez.grad@uni-lj.si

Povzetek

Zaradi zagotavljanja konkurenčnih prednosti organizacije vse bolj uporabljajo različna orodja za analiziranje podatkov, kot so na primer orodja za rudarjenje po podatkih. Zlepki so odsekoma polinomske funkcije, ki se uporabljajo tudi pri nekaterih tovrstnih orodjih. Razen tega se jih veliko uporablja pri računalniško podprtem geometrijskem oblikovanju. Prispevek predstavlja objektni model zlepkov, ki je neodvisen od konkretne programske rešitve in lahko služi za osnovo vsake programske rešitve, ki uporablja zlepke. Tako se zmanjšuje neskladje med tovrstnimi programskimi rešitvami in povečuje njihova združljivost. Za modeliranje sva uporabila standardni jezik za objektno modeliranje UML. Predlagava tudi shranjevanje zlepkov v bazi podatkov, ki podpira standard ODMG. Opisan je tudi prototip sistema, ki sva ga razvila v programskem jeziku Java.

Abstract

Data analysis like Data Mining is getting more and more important every day because the organizations want to get their competitive advantage. Splines are piecewise polynomial functions that are used for analysis by some Data Mining tools. Besides, they are widely used in CAGD applications for designing. The article presents the object model of splines that is independent of the concrete application but can serve as basis for any application which uses splines and reduces the gap among such applications and their incompatibility. It has been modeled by standard object-oriented modeling language UML. We also propose saving splines in an ODMG compliant database. In the article the prototype of a system that has been developed in Java is presented as well.

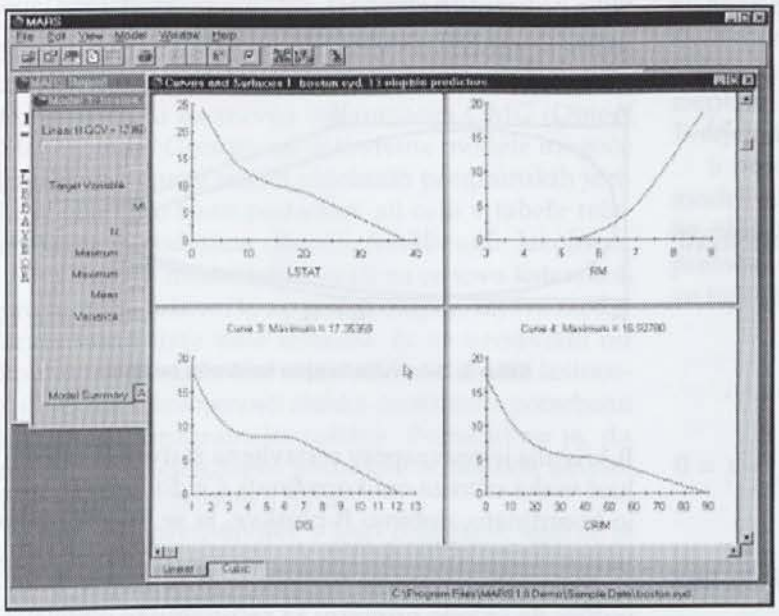
■ ■ ■

1. UVOD

Če hoče organizacija v vse težjih pogojih uspešno poslovati in si zagotavljati konkurenčne prednosti, mora s podatki upravljati kot s pomembnimi viri. To je glavni razlog vse večjega uveljavljanja različnih orodij za analiziranje podatkov, kot so na primer orodja OLAP (angl. On-Line Analitical Processing) ali pa orodja za rudarjenje po podatkih (angl. Data Mining). Ena izmed metod, ki jih orodja za rudarjenje po podatkih uporabljajo za odkrivanje povezav med velikimi količinami podatkov, je tudi aproksimacija oziroma regresija, pri čemer gre za to, da dobimo povezave med spremenljivkami v obliki matematičnih funkcij. Na ta način dobimo tudi napoved za vrednosti neke vhodne spremenljivke v točki, kjer njene vrednosti ne poznamo. Takšne metode so uporabne predvsem pri napovedovanju, ki lahko služi kot osnova za bolj kakovostno odločanje.

Razvoj na področju matematike in statistike je pokazal, da so v ta namen najprimernejše funkcije zlepki. To so odsekoma polinomske funkcije, ki omogočajo lokalnejšo aproksimacijo, v stičiščih pa gladko prehajajo ena v drugo. Orodja, ki nam omogočajo aproksimacijo in s tem povezano napovedovanje, temeljijo na teoriji aproksimacije s pomočjo zlepkov (De Boor, 1978), (Friedman, 1991), (Eubank, 1999), (Pagan, Ullah, 1999). Slika 1 prikazuje orodje za rudarjenje po podatkih MARS (kratica za angl. Multivariate Adaptive Regression Splines), ki ga razvija podjetje Salford Systems.

Razen za aproksimacijo se zlepke veliko uporablja tudi na področju računalniško podprtega geometrijskega oblikovanja (angl. Computer Aided Geometric Design - CAGD). Z zlepki so npr. v računalniku predstavljeni znaki, razen tega so del vsakega zmogljivejšega paketa za oblikovanje (npr. CorelDraw). Modernega oblikovanja si brez zlepkov pravzaprav ne moremo predstavljati. Razvoj oblikovanja z zlepki se je začel v šestdesetih letih v Franciji in ZDA zaradi potreb avtomobilske in letalske industrije (Farin, 1997). Zlepki so na kratko opisani v razdelku 2. Bralec, ki z njimi ni seznanjen, lahko o zlepkih prebere več v navedeni literaturi.



Slika1: orodje MARS za rudarjenje po podatkih, ki uporablja zlepke.

Problem, ki ga vidiva na področju uporabe zlepkov, je neskladje med različnimi programskimi rešitvami, ki zlepke uporabljajo, in njihova nezdružljivost. Predlagava uporabo enotnega objektnega modela, ki je neodvisen od konkretne programske rešitve, sistema za upravljanje baze podatkov (SUBP) in programskega jezika. Za razvoj modela sva uporabila standardni jezik za objektno modeliranje UML (Booch, Rumbaugh, Jacobson, 1999). Razviti model zajema uporabo zlepkov tako na področju aproksimacije kot na področju računalniške grafike ali na katerem drugem področju in predstavlja osnovo tovrstne programske rešitve. Zaradi lastnosti objektne usmerjenosti ga je mogoče razširiti s potrebami konkretne programske rešitve. Opisan je v razdelku 3.

Prispevek predstavlja tudi rešitev za shranjevanje zlepkov v objektni bazi podatkov (razdelek 4). Pri tem sva uporabila standard ODMG (Cattell at al., 2000), ki ga razvija istoimenska organizacija. Standard je neodvisen od uporabljenega programskega jezika in sistema za upravljanje baze podatkov. Mogoče ga je uporabiti za transparentno shranjevanje v "pravi" objektni bazi podatkov in preko posebne preslikave med objekti in relacijami tudi v relacijskih bazah podatkov.

Peti razdelek prispevka je namenjen kratki predstavitvi prototipa sistema, ki sva ga razvila s pomočjo programskega jezika Java in objektne baze podatkov Poet, šesti pa zaključku.

2. ZLEPKI

Pred razvojem, katerega rezultat so zlepki, se je za interpolacijo in aproksimacijo največ uporabljalo polinome, saj je relativno enostavno izračunati njihove koeficiente. Vendar z naraščanjem števila točk, ki jih želimo interpolirati ali aproksimirati, ni vedno mogoče najti zadovoljive funkcije v razredu polinomov. Polinomi višjega reda lahko oscilirajo (Schumaker, 1993), kar pomeni, da je bilo potrebno vpeljati nov razred funkcij. Zlepki, to so odsekoma polinomske funkcije, ki v stičiščih gladko prehajajo ena v drugo, so takšen razred funkcij. Zlepek je možno izraziti kot zaporedje polinomov definiranih na posameznih odsekih ali pa v posebni obliki kot B-zlepke ali Beziereve krivulje (de Boor, 1978), (Farin, 1997), (Schumaker, 1993).

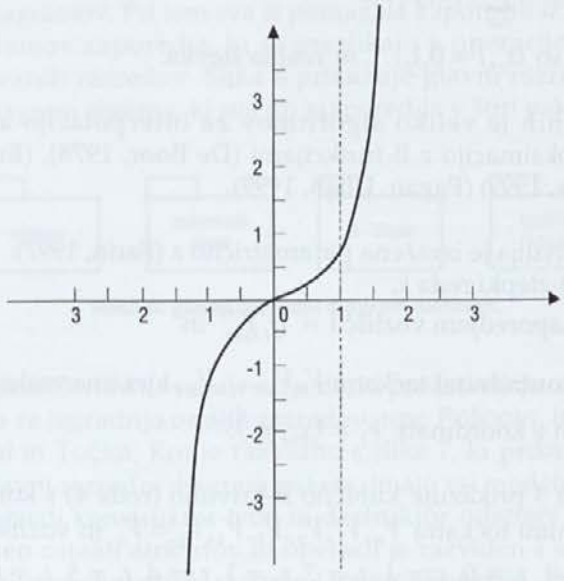
Zlepki, s katerimi se srečujemo pri orodjih za rudarjenje po podatkih, običajno sodijo v prvo skupino, saj funkcijo največkrat želimo izraziti v obliki polinomskih odsekov.

Slika 2 prikazuje kubični zlepek, sestavljen iz dveh odsekov:

$$f(x)=\begin{cases} x^3; & x\le 1 \\ 2x^3-3x^2+3x-1; & x>1 \end{cases}$$

Kot je razvidno s slike, je funkcija v točki $x = 1$ gladka.

B-zlepki, ki služijo za posebno predstavitev polinomskih zlepkov, imajo še dodatne prednosti, kot je npr. večja numerična stabilnost (Schumaker, 1993).



Slika 2: kubični zlepek.

Definiramo jih lahko na več načinov, najenostavneje pa s pomočjo rekurzivne formule:

$$B_{i,k}(x) = \frac{x - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(x) + \frac{t_{i+k} - x}{t_{i+k} - t_{i+1}} B_{i+1,k-1}(x),$$

kjer je t = $(t_i)_{i=0}^{m+k}$ nepadajoče zaporedje realnih števil, ki jih imenujemo vozlišča, izraza

$$\frac{x - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(x) \text{ in } \frac{t_{i+k} - x}{t_{i+k} - t_{i+1}} B_{i+1,k-1}(x)$$

pa imata vrednost 0 kadar je $t_{i+k-1} - t_i = 0$ ali $t_{i+k} - t_{i+1} = 0$.

B-zlepek reda k $B_{i,k}(x)$ je polinom reda k, ki ima pomembne lastnosti:

$B_{i,k}(x) \geq 0$, ko $x \in [t_i, t_{i+k}]$,

$B_{i,k}(x) > 0$, ko $x \in (t_i, t_{i+k})$,

$B_{i,k}(x) = 0$, ko $x \notin [t_i, t_{i+k}]$.

Razlog za vpeljavo B-zlepkov je definicija B-funkcij in B-krivulj. Prve se uporablja predvsem za interpolacijo ali aproksimacijo, slednje pa pri oblikovanju.

B-funkcija reda k z zaporedjem vozlišč t = $(t_i)_{i=0}^{m+k}$ je vsaka linearna kombinacija B-zlepkov reda k:

$$f(x) = \sum_{i=0}^{m} \alpha_i B_{i,k}(x),$$

kjer so $\alpha_i, i = 0,1,\ldots,m$ realna števila.

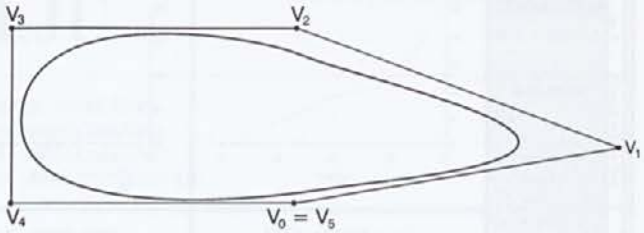
Znanih je veliko algoritmov za interpolacijo ali aproksimacijo z B-funkcijami (De Boor, 1978), (Eubank, 1999) (Pagan, Ullah, 1999).

B-krivulja je izražena parametrično z (Farin, 1997):

- B-zlepki reda k,
- zaporedjem vozlišč t = $(t_i)_{i=0}^{m+k}$ in
- kontrolnimi točkami $V_0 V_1, \ldots, V_m$, kjer ima vsaka x in y koordinati: $V_i = (x_i, y_i)$.

Slika 3 prikazuje kubično B-krivuljo (reda 4) s kontrolnimi točkami $V_0, V_1, V_2, V_3, V_4, V_5 = V_0$ in vozlišči $t_0 = -1$, $t_1 = 0$, $t_2 = 1$, $t_3 = 2$, $t_4 = 3$, $t_5 = 4$, $t_6 = 5$, $t_7 = 6$, $t_8 = 7$, $t_9 = 8$.

Slika prikazuje tudi njen kontrolni poligon, to je lomljena črta, ki povezuje kontrolne točke.



Slika 3: B-krivulja in njen kontrolni poligon.

B-krivulja je pravzaprav sestavljena iz dveh B-funkcij, kjer vsaka ustreza eni koordinati. Če dodamo še tretjo koordinato, dobimo B-ploskve, ki se jih prav tako uporablja pri oblikovanju.

Pomembna lastnost B-zlepkov, B-funkcij in B-krivulj je, da so zelo primerni za računalniške obdelave, saj je za njihovo rekonstrukcijo potrebno shraniti malo podatkov. Ena izmed njihovih glavnih prednosti je tudi, da sprememba enega odseka vpliva le na bližnje odseke.

Bezierеve krivulje so najpogosteje uporabljene pri orodjih za računalniško podprto geometrijsko modeliranje. Pravzaprav gre za posebno obliko B-krivulj, ki sta jo v šestdesetih letih vpeljala Bezier in De Casteljau. Ime nosijo po Bezieru, saj De Casteljauevo delo ni bilo objavljeno. Definirane so kot

$$B_n[V_0, V_1, \ldots, V_n; a, b; t] = \sum_{i=0}^{n} V_i B_i^n(t), a \leq t \leq b,$$

kjer so

$$B_i^n(t) = \binom{n}{i} \frac{(t-a)^i (b-t)^{n-i}}{(b-a)^n}, a \leq t \leq b, i = 0,1,\ldots,n$$

Bernsteinovi polinomi stopnje n, točke $V_0 V_1, \ldots, V_n$ pa so kontrolne točke.

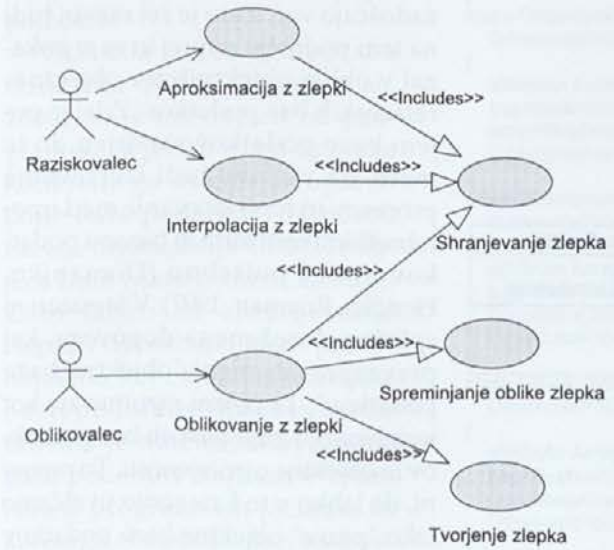
Oblikovanje s pomočjo B-krivulj ali Bezierevih krivulj (Farin, 1997) je enostavno, saj kontrolni poligon ponazarja obliko krivulje. Če premaknemo kontrolno točko, to vpliva na bližnje odseke krivulje. Operacije kot rotacija, premikanje ali spreminjanje velikosti lahko realiziramo tako, da operacijo izvedemo na kontrolnem poligonu in ponovno narišemo krivuljo. Če želene oblike ne moremo doseči s premikanjem kontrolnih točk, krivulji dodamo nove kontrolne točke s pomočjo delitvenih algoritmov (Farin, 1997), (Indihar, 1992), katerih namen je, da obstoječo krivuljo opišemo z večjim številom kontrolnih točk in tako dobimo več možnosti za spreminjanje njene oblike.

3. OBJEKTNI MODEL SISTEMA

Sistem sva modelirala s pomočjo standardnega jezika za objektno modeliranje UML (Unified Modeling Language), ki ga razvija organizacija OMG (Object Management Group), saj je tovrstne modele mogoče preslikati v enega izmed objektnih programskih jezikov, objektno bazo podatkov ali celo v tabele relacijske baze podatkov (Booch, Rumbaugh, Jacobson, 1999). Razviti model lahko služi za osnovo katerekoli programske rešitve, ki uporablja zlepke, saj sva modelirala samo tiste dele sistema, ki so neodvisni od konkretne programske rešitve. Model zaradi lastnosti objektne usmerjenosti zlahka razširimo s potrebami konkretne programske rešitve. Pomembno je, da model ne zajema samo podatkov o zlepkih, ampak tudi operacije za delo z njimi.

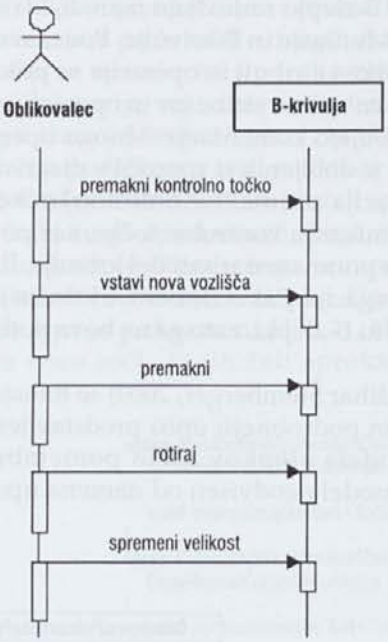
Ker je za predstavitev v bazi podatkov, tudi če je le-ta objektna, najpomembnejši statični vidik sistema, sva največ pozornosti posvetila razrednim diagramom. Te diagrame navajajo tudi v literaturi kot najpomembnejše pri modeliranju statičnega pogleda na sistem (Booch, Rumbaugh, Jacobson, 1999), (Blaha, Premerlani, 1998). Kljub temu pa sva predstavila tudi del dinamike sistema, ki vključuje zlepke. Predstavila sva jo z diagrami primerov uporabe in diagrami zaporedja, ki sestavljajo pogled na uporabniške zahteve. S pomočjo teh diagramov sva lažje določila potrebne operacije modeliranih razredov.

Slika 4 prikazuje diagram primerov uporabe sistema. Kot vidimo, lahko imamo dve vrsti uporabnika sistema, ki sta prikazana kot akterja. To sta Raziskovalec in Oblikovalec. Prvi je v interakciji s primeroma uporabe Aproksimacija z zlepki in Interpolacija z zlepki, drugi pa s primerom uporabe Oblikovanje z zlepki. Vsi trije omenjeni primeri uporabe vsebujejo (oziroma uporabljajo) primer uporabe Shranjevanje zlepkov, tretji pa še Spreminjanje oblike zlepka in Tvorjenje zlepka.



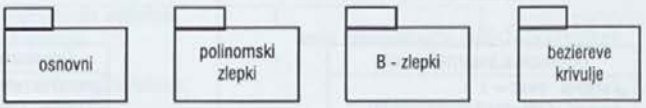
Slika 4: Diagram primerov uporabe sistema.

S pomočjo diagramov zaporedja sva podrobneje modelirala primer uporabe Spreminjanje oblike zlepka. Slika 5 prikazuje diagram zaporedja za B-krivulje, podoben diagram pa lahko narišemo tudi za Bezierove krivulje. Nahaja se v (Indihar Štemberger, 2000).



Slika 5: diagram zaporedja za B-krivulje.

Strukturo sistema sva modelirala s pomočjo razrednih diagramov. Pri tem sva si pomagala s sporočili iz diagramov zaporedja, ki se preslikajo v operacije ustreznih razredov. Slika 6 prikazuje glavni razredni diagram sistema, ki sva jih razporedila v štiri pakete.



Slika 6: glavni razredni diagram sistema.

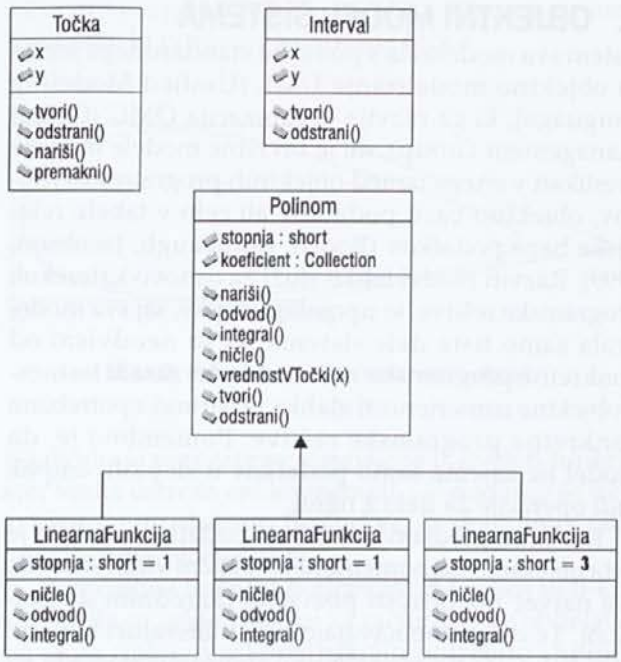
Paket Osnovni vsebuje razrede, ki predstavljajo osnovo za izgradnjo ostalih razredov, npr. Polinom, Interval in Točka. Kot je razvidno s slike 7, ki prikazuje glavni razredni diagram paketa, imajo vsi modelirani razredi konstruktor tvori in destruktor odstrani. Pomen ostalih atributov in operacij je razviden s slike. Paket vsebuje tudi razrede LinearnaFunkcija, KvadratnaFunkcija in KubičnaFunkcija, ki so najpogosteje

v rabi. Ti razredi podedujejo atribute in operacije od razreda Polinom, vendar so nekatere enostavnejše.

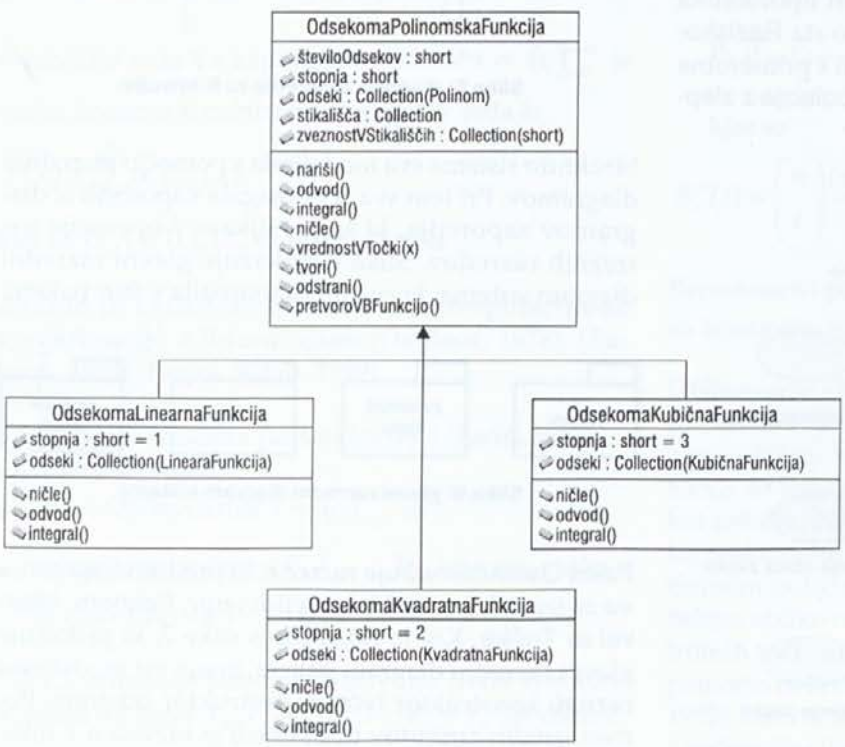
Paket Polinomski zlepki vsebuje razrede, ki so potrebni za predstavitev odsekoma polinomskih funkcij. Osrednji razred paketa se imenuje OdsekomaPolinomskaFunkcija, modeliranih pa je še več razredov, ki so vsi nasledniki tega razreda. Podrobnosti v zvezi z atributi in operacijami so razvidne s slike 8, ki prikazuje glavni razredni diagram paketa. Vsi razredi v paketu so zbirke polinomov, kjer je zbirka lahko seznam, polje ali drugega tipa.

Paket B-zlepki sestavljajo razredi, ki modelirajo B-zlepke, B-funkcije in B-krivulje. Povezave med razredi, ter njihovi atributi in operacije so prikazani na sliki 9. Pomen večine atributov in operacij je očiten, zato ne potrebujejo komentarja. Mnogo operacij razreda BKrivulja je dobljenih iz sporočil v diagramu zaporedja. Operacija premakniKontrolnoTočko na primer služi premikanju kontrolne točke, kar povzroči, da je potrebno ponovno narisati del krivulje. Razrede povezuje agregacija. Paket Bezierove krivulje je zelo podoben paketu B-zlepki, zato ga ne bova podrobneje opisala.

V (Indihar Štemberger, 2000) se nahaja še več diagramov in podrobnejši opisi predstavljenega objektnega modela zlepkov. Zelo pomembno je, da je opisani model neodvisen od namena uporabe zlepkov in lahko služi kot osnova sistema, kjer se zlepki uporablja za aproksimacijo ali interplolacijo ali pa sistema, ki omogoča oblikovanje z zlepki.



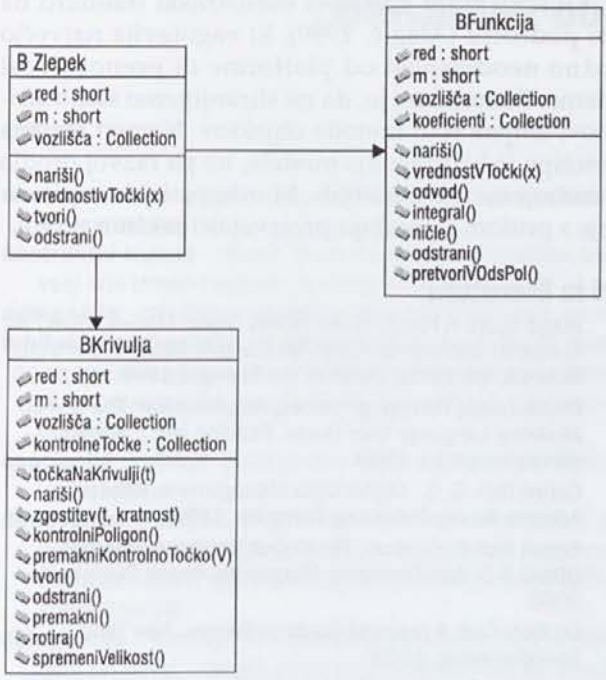
Slika 7: razredni diagram paketa Osnovni.



Slika 8: razredni diagram paketa Polinomski zlepki.

4. SHRANJEVANJE ZLEPKOV

Pri sodobnih informacijskih sistemih se srečujemo z vse kompleksnejšimi objekti, za katerih shranjevanje klasične relacijske baze podatkov ne zadoščajo več. Zato je šel razvoj tudi na tem področju naprej in se je pokazal v obliki objektnih ter objektno-relacijskih baz podatkov. Zdaj je pojem baze podatkov razširjen do te mere, da vsebuje tudi izvrševanje procesov in torej ločevanje med uporabniškimi rešitvami in bazami podatkov ni več potrebno (Domanjko, Heričko, Rozman, 1997). V literaturi ni splošno doseženega dogovora, kaj pravzaprav pomeni "objektna baza podatkov". Ta pojem razumevajo kot kombinacijo zmogljivosti baz podatkov in objektne usmerjenosti. To pomeni, da lahko v to kategorijo uvrščamo tako "prave" objektne baze podatkov kot objektne razširitve sistemov za upravljanje z relacijskimi bazami podatkov, kot je na primer Oracle.



Slika 9: razredni diagram paketa B-zlepki.

Za shranjevanje kompleksnih objektov, s katerimi se srečamo tudi pri programskih rešitvah, ki uporabljajo zlepke, so objektne baze podatkov prava rešitev. Na področju računalniške grafike se za shranjevanje nekaterih posebnih primerov zlepkov že uporabljajo objektne baze podatkov (Cattell, 1994), ne uporablja pa se jih pri orodjih za rudarjenje po podatkih.

Odločila sva se za uporabo standarda ODMG (Cattell et al., 2000), ki je neodvisen od programskega jezika in od uporabe konkretnega sistema za upravljanje baze podatkov. Standard razvija organizacija ODMG (Object Data Management Group). Zato lahko za shranjevanje zlepkov uporabimo tako "pravo" objektno kot objektno-relacijsko bazo podatkov. Po standardu ODMG je možno načrt sheme baze podatkov napisati v enem izmed programskih jezikov, za katerega obstaja jezikovna povezava (C++, Smalltalk in Java) ali v neodvisnem jeziku za definicijo objektov (angl. Object Definition Language - ODL). Odločila sva se za slednjo možnost. Slika 10 prikazuje definicijo sheme za polinomske zlepke. Definicija sheme za vse ostale razrede, ki so del sistema, se nahaja v (Indihar Štemberger, 2000).

5. PROTOTIP SISTEMA

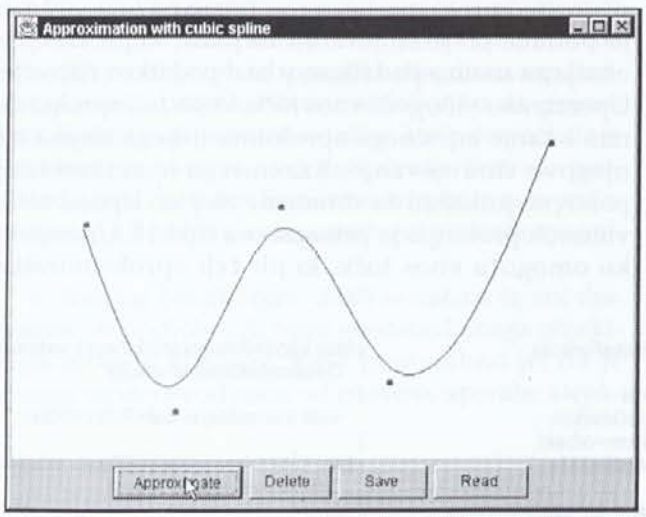
Izdelala sva tudi prototip sistema, za izgradnjo katerega sva uporabila obetavni programski jezik Java, ki se vse bolj uveljavlja. Prednost tega jezika je predvsem neodvisnost delovanja od strojne opreme in operacijskega sistema, ima pa še veliko drugih dobrih lastnosti. Prototip se nanaša na uporabo zlepkov na področju aproksimacije, saj so ravno na tem področju potrebe po shranjevanju zlepkov, ki jih kreirajo orodja za analizo podatkov, v bazi podatkov največje. Uporabniku omogoča vnos točk, ki jih želi aproksimirati, iskanje kubičnega aproksimacijskega zlepka ter njegovo shranjevanje. Razen tega je možno tudi ponovno prikazati že shranjene zlepke. Uporabniški vmesnik prototipa je prikazan na sliki 11. Uporabniku omogoča vnos točk, ki jih želi aproksimirati s

```
class OdsekomaPolinomskaFunkcija
{
   attribute short stopnja
   attribute short številoOdsekov;
   attribute Array<Polinom>odseki;
   attribute array<float>stikališča;
   void tvori();
   void odstrani()raises(ni_funkcije);
   void nariši();
   float vrednostVTočki(in floatx)raises(ni_definirana);
   bag<float>ničle();
   OdsekomaPolinomskaFunkcija odvod();
   OdsekomaPolinomskaFunkcija integral();
   BFunkcija pretvoriVBFunkcijo();
}
class OdsekomaLinearnaFunkcija extends
   OdsekomaPolinomskaFunkcija
{
   attribute Array<LinearnaFunkcija>odsek;
   bag<float>ničle();
   array<float>odvod();
   OdsekomaKvadratnaFunkcija integral();
}
class OdsekomaKvadratnaFunkcija extends
   OdsekomaPolinomskaFunkcija
{
   attribute Array<KvadratnaFunkcija>odsek;
   bag<float>ničle();
   OdsekomaLinearnaFunkcija odvod();
   OdsekomaKubičnaFunkcija integral();
}
class OdsekomaKubičnaFunkcija extends
   OdsekomaPolinomskaFunkcija
{
   attribute Array<KubičnaFunkcija>odsek;
   bag<float>ničle();
   OdsekomaKvadratnaFunkcija odvod();
   OdsekomaPolinomskaFunkcija integral();
}
class KubičniInterpolacijskiZlepek extends
   OdsekomaKubičnaFunkcija
{
   void interpoliraj(in Set<Točka>točke);
}
class KubičniAproksimacijskiZlepek extends
   OdsekomaKubičnaFunkcija
{
   void aproksimiraj(in Set<Točka>točke);
}
class NaravniKubičniZlepek extends
   KubičniInterpolacijskiZlepek
{
   void interpoliraj(in Set<Točka>točke);
}
class KubičniZlepekZOdvod extends
   KubičniInterpolacijskiZlepek
{
   void interpoliraj(in Set<Točka>točke);
     in float odvZ, in float odvK);
}
class PeriodičniKubičniZlepek extends
   KubičniInterpolacijskiZlepek
{
   void interpoliraj(in Set<Točka>točke);
}
```

Slika 10: definicija sheme za razrede OdsekomaPolinomskaFunkcija, OdsekomaLinearnaFunkcija, OdsekomaKvadratnaFunkcija in OdsekomaKubičnaFunkcija.

kubičnim zlepkom, iskanje aproksimacijskega zlepka po eni izmed metod opisanih v (Eubank, 1999), njegovo shranjevanje in branje shranjenih zlepkov. Razred, ki je rezultat aproksimacije, ustreza razredu OdsekomaKubičnaFunkcija in je naslednik tega razreda, saj je opremljen še z dodatnimi operacijami.

Odločila sva se za shranjevanje objektov v objektni bazi Poet, vendar bi bilo zaradi neodvisnosti razvitega modela od platforme, ki sva jo dosegla z uporabo standarda ODMG, mogoče uporabiti katerikoli SUBP, ki ima povezavo s tem standardom. Preko preslikave objektov v relacije (na primer Java Blend) bi bilo mogoče uporabiti tudi relacijsko bazo podatkov.



Slika 11: uporabniški vmesnik prototipa.

6. ZAKLJUČEK

Meniva, da predstavljeni model lahko služi kot "skupni imenovalec" sistema, ki uporablja zlepke. Na njem lahko temelji vsaka tovrstna programska rešitev, ki ponavadi model razširja z dodatnimi razredi, ki pa so nasledniki modeliranih razredov. Prepričana sva, da je izbira jezika UML pravilna, saj je primeren za veliko okolij. Uporaba standarda ODMG, pri razvoju katerega sodeluje večina proizvajalcev sistemov za upravljanje objektnih baz podatkov, je prav tako pravilna, saj je to edini obstoječi industrijski standard na tem področju (Alagič, 1999), ki zagotavlja največjo možno neodvisnost od platforme in prenosljivost sistema. Pomembno je, da ne shranjujemo samo atributov, ampak tudi metode objektov. Namen razvoja prototipa je bil preizkus modela, ne pa razvoj orodja za rudarjenje po podatkih. Morda pa lahko opisane ideje s pridom uporabijo proizvajalci takšnih orodij.

Viri in literatura

1. Alagič Suad: A Family of the ODMG Object Models, Third East European Conference ADBIS'99 (Lecture Notes in Computer Science, Vol 1691), Berlin et al.: Springer, 1999, str. 14-30.
2. Booch Grady, Rumbaugh James, Jacobson Ivar: The Unified Modeling Language User Guide. Reading [etc.]: Addison Wesley Longman, 1999.
3. Cattell Rick G. G.: Object Data Management. Reading: Addison-Wesley Publishing Company, 1994.
4. Cattell Rick G. G. et al.: The Object Database Standard ODMG 3.0. San Francisco: Morgan Kaufmann Publishers, 2000.
5. De Boor Carl: A practical Guide to Splines. New York: Springer-Verlag, 1978.
6. Domanjko Tomaž, Heričko Marjan, Rozman Ivan: Uporaba objektnih podatkovnih baz, COTL, 1 (1997), 2, [URL: http://lisa.uni-mb.si/cot/cotl/april97/index.shtml].
7. Eubank Randall L.: Nonparametric Regression and Spline Smoothing (Statistics, Textbooks and Monographs, V. 157), Marcel Dekker, 1999.
8. Farin Gerald: Curves and Surfaces for CAGD. Boston: Academic Press, 1997.
9. Friedman J.H.: Multivariate Adaptive Regression Splines. Annals of Statistics, 19 (1991), str. 1-141.
10. Indihar Štemberger Mojca: Zlepki v okolju objektnih baz podatkov, Doktorska disertacija, Ljubljana: Univerza v Ljubljani, Ekonomska fakulteta, 2000.
11. Kim Won, urednik: Modern Database systems – The Object Model, Interoperability and Beyond. New York: Addison-Wesley Publishing Company, 1995.
12. Khoshafian Setrag, Abnous Razmik: Object Orientation. New York: John Wiley Đ& Sons, 1995.
13. Object Data Management Group, [URL: http://www.odmg.org/].
14. Object Management Group, [URL: http://www.omg.org/].
15. Pagan Adrian, Ullah Aman: Nonparametric Econometrics, Cambridge: Cambridge University Press, 1999.
16. Poet, [URL: http://www.poet.com/].
17. Salford Systems, [URL: http://www.salford-systems.com/].
18. Schumaker Larry L.: Spline Functions: Basic Theory. Malabar, Florida: Krieger Publishing Company, 1993.

Dr. Mojca Indihar Štemberger je po osnovni izobrazbi matematik. Leta 1996 je magistrirala na Fakulteti za računalništvo in informatiko, leta 2000 pa doktorirala iz informacijsko upravljalskih ved na Ekonomski fakulteti, kjer je tudi zaposlena kot asistentka za informatiko. Ukvarja se s poslovno informatiko, optimizacijo in bazami podatkov.

Dr. Janez Grad je magistriral iz matematike na Univerzi v Birminghamu, Anglija, leta 1973 pa doktoriral iz matematičnih znanosti na Vseučilišču v Zagrebu. Od leta 1973 sodeluje kot učitelj za informatiko na Ekonomski fakulteti, najprej kot docent, od leta 1979 dalje kot izredni profesor, od 1985 pa kot redni profesor. Ukvarjal se je s programiranjem na računalniku in z reševanjem problema lastnih vrednosti in vektorjev matrik, v zadnjih letih pa se ukvarja z reševanjem problemov s področja operacijskega raziskovanja in s področja baz podatkov.

Slovarček objektne tehnologije

IZRAZI

abstraktna operacija - deklaracija operacije brez podane implementacije

abstraktni razred - razred, ki ne more imeti primerkov, saj je vsaj ena izmed njegovih operacij abstraktna

agregacija - združitev – posebna vrsta asociacije tipa »je-del«

arhitekturni diagrami - diagramska tehnika jezika za objektno modeliranje za prikaz konfiguracije izvajalnega elementa in pripadajočih programskih komponent, procesov in objektov

asociacija - relacija, soodvisnost, logična povezanost dveh ali več razredov

dedovanje - relacija, ki povezuje razrede - podrazredi podedujejo vse operacije in atribute nadrazredov; implementacija povezave, generalizacija/specializacija *(posplošitev/specializacija)*

delegiranje - koncept, ki vzpostavlja povezave med konkretnimi primerki - objekt posreduje odgovornost za izvedbo operacije drugemu objektu

diagrami aktivnosti - diagramska tehnika jezika za objektno modeliranje za opis dinamičnega obnašanja sistema z vidika aktivnosti

diagrami komponent - diagramska tehnika jezika za objektno modeliranje za prikaz komponent in relacij med njimi

diagrami objektov - poseben primer razrednih diagramov, ki prikazujejo primerke in njihove medsebojne vezi

diagrami primerov uporabe - diagramska tehnika jezika za objektno modeliranje, s katero predstavimo komunikacijo med akterji (uporabniki, drugimi sistemi, navideznimi uporabniki) in računalniškim sistemom

diagrami sodelovanja - diagramska tehnika jezika za objektno modeliranje, s katero prikažemo interakcije med objekti in njihove medsebojne povezave

diagrami stanj - diagramska tehnika jezika za objektno modeliranje za opisovanje dinamičnega obnašanja sistema v obliki zaporedja stanj, skozi katere gre objekt v času obstajanja, pripadajočih akcij in aktivnosti ter dogodkov, ki prožijo prehode med stanji

diagrami zaporedja - diagramska tehnika jezika za objektno modeliranje za prikaz časovnega sodelovanja objektov v interakciji

dinamično povezovanje - proces, ki omogoča, da se identifikacija ustrezne metode odloži vse do časa izvajanja oz. trenutka, ko se mora metoda izvesti

kompozicija - posebna vrsta agregacije z močnim lastništvom

metarazred -opis lastnosti razreda, primerek metarazreda je razred

metoda - koda, ki se izvrši za izvedbo zahtevane storitve. Implementacija operacije

objekt - združba podatkov in funkcionalne logike - operacij

ograjevanje - koncept ločevanja definicij od podrobnosti implementacije

polimorfizem – mnogoličnost - koncept, ki pogojuje, da se objekti na isto sporočilo odzivajo na različne načine - ista operacija je implementirana na več različnih načinov oz. zanjo obstaja več metod

parametrizirani razred - opis razreda z vsaj enim nedoločenim formalnim parametrom; definira družino razredov, katero dobimo z določanjem različnih vrednosti formalnim parametrom

paketi - mehanizem za abstrakcijo, združevanje in prikaz odvisnosti med elementi oz. gradniki diagramov, oblikovanih v jeziku za objektno modeliranje

primerek - predstavnik, npr. objekt je primerek razreda

primer uporabe - je zaporedje transakcij v sistemu, katerega naloga je, da akterju vrača rezultate, ki jih ta lahko ovrednoti

razred - skupen opis množice podobnih objektov, ki služi za oblikovanje več objektov z istim obnašanjem

razredna operacija - operacija, ki dostopa in deluje le nad razrednimi atributi

razredni atribut - atribut, ki je skupen vsem primerkom razreda

razredni diagrami - diagramska tehnika jezika za objektno modeliranje za prikaz statične strukture sistema s pomočjo razredov, pripadajočih atributov in operacij/metod, vmesnikov, asociacij, dedovanj in odvisnosti

vidljivost - omejitev glede možnosti dostopa do operacij in atributov – v splošnem ločimo javno (dostop dovoljen vsem), zaščiteno (dostop dovoljen le primerkom tega razreda in primerkom njegovih naslednikov) in zasebno vidljivost (dostop dovoljen le primerkom razreda)

vmesnik - tip, ki opredeli množico operacij za opis zunanje vidnega obnašanja razreda, komponente ali paketa, ki sami v celoti zagotavljajo implementacijo v vmesniku navedenih operacij

AKRONIMI

IS – informacijski sistem
COM+ - ang. component object model
CORBA – ang. common object request broker architecture
DLL – ang. dynamic link library
EJB – ang. enterprise java beans
OCL – ang. object constraint language
OO – objektno orientiran
UML – ang. unified modeling language
PU – primer uporabe
SP – strateško planiranje
SUPB – sistem za upravljanje podatkovne baze
TP – testni primer

Obširnejši slovar strokovnih izrazov za področje objektne tehnologije poglejte na spletnih straneh centra za objektno tehnologijo: http://lisa.uni-mb.si/cot/slovar.htm.

Pojmovnik računovodstva, financ in revizije

Dr. Ivan Turk:

Slovenska strokovna javnost s področja računovodstva in financ je bogatejša za novo delo profesorja Turka. Konec julija je namreč v okviru Slovenskega inštituta za revizijo izšel pojmovnik računovodstva, financ in revizije, ki bo tako kot vsi njegovi dosedanji prispevki, močno vplival na razvoj stroke.

Da je na tem področju zevala velika praznina, postane bralcu jasno že ko knjigo prime v roke, saj obsega 1083 strani, na katerih je prof. Turk podal temeljit pregled pojmov, s katerimi se srečujemo pri vsakodnevnem delu in strokovnih razpravah s področja računovodstva in financ. Avtor nam v uvodu poda nekaj svojih razmišljanj o prehojeni poti, ki je vodila do nastanka tako obsežnega dela, saj se je s tem področjem pričel ukvarjati že na samem začetku svojega akademskega udejstvovanja. V nadaljevanju je pojmovnik razdeljen na dva večja vsebinska sklopa od katerih prvi, obsežnejši, zajema opredelitev strokovnih pojmov ter slovensko slovenski in slovensko angleški slovar izrazov z obravnavanega področja, drugi del pa vsebuje angleško slovenski slovar izrazov za izbrane strokovne pojme s področja računovodstva, financ in revizije.

Kdor pozna njegov nenehni trud za pravilno rabo slovenščine v stroki in dosledno rabo pojmov v znanstvenih razpravah, ne more biti presenečen, da se je prav prof. Turk lotil tako obsežnega in zahtevnega dela. Pojmovnik vsebuje opredelitve pojmov ne samo s področja financ in računovodstva, temveč tudi z drugih področij, ki so z njima tesno povezani, zato bo koristen pripomoček strokovnjakom različnih področij, priloženi slovarji pa bodo koristili predvsem prevajalcem in bralcem angleških strokovnih del.

Malo jih je, kot je prof. Turk, ki bi s tako neizmernim žarom in zagnanostjo dajali slovenščini novo izrazno moč ter kot sam pravi, upoštevali slovenščino kot živ jezik, ki je razvoju izredno prilagodljiv. Ustvarjanje tako obsežnega dela zahteva od avtorja veliko mero natančnosti, sistematičnosti ter osebne razgledanosti, ki je prof. Turku nedvomno ne manjka. Bralec pri pregledovanju pojmovnika nikakor ne bo mogel spregledati izredno domiselne in natančne opredelitve nekaterih pojmov. Pri njihovi izbiri se je avtor usmeril predvsem na tiste, ki so v strokovnih prispevkih pogostejši, kar je samo dokaz več, kako odprte so še možnosti na tem področju.

Z izdajo pojmovnika je prof. Turk uporabnikom podal vse možnosti za bogatenje njihovega strokovnega izrazoslovja in večjo uporabnost slovenščine v strokovnih in znanstvenih delih s področja računovodstva, financ in revizije. To bo od njih zahtevalo določen trud, ki pa je neprimerljiv s trudom avtorja, ki je bil vložen v njegovo izdelavo. Prepričan sem, da se uporabi pojmovnika ne bo mogel odreči nihče, ki je vsaj malo aktiven pri razvoju računovodske, finančne ali revizijske stroke in mu je mar za lastno jezikovno identiteto pri strokovnem udejstvovanju.

Mitja Skitek

PRVI NOSILCI EVROPSKIH RAČUNALNIŠKIH SPRIČEVAL PRI NAS

Zaradi nepredvidenih težav so se pooblaščeni izvajalci programa ECDL lahko lotili certificiranja šele v septembru, to je v tekočem šolskem letu. Kljub tem težavam so bila prva evropska računalniška spričevala (European Computer Driving Licence – ECDL Certificate) že podeljena. To je uspelo družbi ISA.IT.

Družba ISA.IT, d.o.o. sodi med bolj prizadevne nosilce koncesije za izvajanje računalniškega usposabljanja po programu ECDL in za opravljanje izpitov za ECDL. Redna računalniška usposabljanja, ki so jih izvajali že v preteklih letih, so dopolnjena s pripravo za izpite in možnostjo opravljanja preizkusov in so na takem nivoju, kot zahteva standard ECDL. Spričevalo ECDL lahko pridobijo vsi tisti kandidati, ki že imajo znanje o posameznih programskih orodjih (npr. Word za urejanje besedil, Excel za delo s preglednicami, internet, elektronska pošta...) in jih nato dopolnijo ali uskladijo s programom ECDL, kot tudi začetniki, ki računalnika pri svojem delu še ne uporabljajo, pa z usposabljanjem po programu ECDL dobijo vsa potrebna znanja.

Z izvajanjem izpitov so v podjetju ISA.IT začeli v septembru 2000. Prvi udeleženci so bili posamezniki iz družb, ki so že imeli potrebno znanje in so se udeleževali samo izpitov. Kasneje so se začeli zanimati za spričevala tudi v državni upravi, saj je Center vlade za informatiko na javnem razpisu družbo izbral ISA.IT za izvajalca priprave za pridobitev spričeval ECDL in izvajalca izpitov ECDL za zaposlene v državni upravi. Istočasno je stekla šola ECDL, ki zajema sedem različnih seminarjev vključno z izpiti. Tako so udeleženci v petih tednih obvladali potrebno znanje in pridobili spričevalo. V družbi ISA.IT so pripravili tudi komplet gradiv, ki služijo za pripravo na izpite in so primerna predvsem za tiste, ki želijo potrebno znanje pridobiti sami brez udeležbe na tečajih ali pa želijo preveriti in utrditi znanje, ki ga že imajo.

Rezultati načrtnega uvajanja programa ECDL v Sloveniji so že vidni. Prvi, ki so v Sloveniji pridobili spričevala ECDL, so Matjaž Kavar (Didakta, Radovljica), Boštjan Laba (TSC Laba, Litija), Edmund Knupleš (Upravna enota Murska Sobota), Bogomir Klampfer (Upravna enota Slovenske Konjice), Peter Miklič (Ministrstvo za notranje zadeve, Upravna akademija, Ljubljana), David Drofenik in Franc Vrbančič (Šolski center Ptuj, Poklicna in tehniška elektro šola, Ptuj), Alojz Pačnik (Občina Zreče), Niko Perič (Labod Konfekcija, Novo mesto) in Helena Čretnik (Osnovna šola Moste). Prva spričevala so bila javno in slovesno podeljena oktobra na letošnjem Infosu v Cankarjevem domu v Ljubljani. V postopku pridobivanja spričevala je trenutno več kot 50 kandidatov; med njimi je vse več takih, ki se najprej udeležijo seminarjev, nato pa znanje preverijo na izpitih za posamezne module.

ECDL ima vrednost tako za imetnika spričevala, kot za delodajalca. Nosilec spričevala ima v rokah priznan izkaz znanja in usposobljenosti za delo z računalnikom. Vse več je tudi primerov, da zaposlene pošiljajo na usposabljanje njihovi delodajalci, ker je to naložba v produktivnost, spričevalo ECDL pa je za delodajalca obenem tudi zavarovanje naložbe, saj se lahko podeli le tistim, ki so usposabljanje uspešno zaključili in opravili vse izpite. Upamo, da bo mednarodno spričevalo ECDL tudi v Sloveniji kmalu dobilo tako veljavo, kot jo ima drugod po Evropi, kjer je že standard za ugotavljanje računalniške pismenosti zaposlenih.

Marjana Kajzer Nagode

Vabilo k sodelovanju na posvetovanju Dnevi slovenske informatike Portorož 2001

Vabimo vas, da se dejavno udeležite že tradicionalnega posvetovanja informatikov, ki bo v času od 18. do 21. aprila 2001 v kongresnem Centru Grand hotel Emona, Portorož. Tema posvetovanja bo »Omrežno gospodarstvo«. Referati naj podajajo različne poglede na informacijsko tehnologijo in informatiko: teoretične in raziskovalne prispevke, praktične rešitve in njihovo uvajanje. Predstavljeni bodo v eni od naslednjih sekcij:

A. Metodologija, informacijska tehnologija in pristopi (vodja sekcije Ivan Rozman)
B. Poslovna informatika in povezovanje poslovnih sistemov (vodja sekcije Marjan Krisper)
C. Internet in tehnologija elektronskega poslovanja (vodja sekcije Tomaž Gornik)
D. Informacijske rešitve in uvajanje IS (vodja sekcije Ivan Vezočnik)
E. Izobraževanje in usposabljanje na področju informatike (vodja sekcije Vladislav Rajkovič)
F. Operacijska raziskovanja (vodja sekcije Lidija Zadnik-Stirn)
G. Strokovni jezik (vodja sekcije Katarina Puc)
H. Sociološki, pravni in etični vidiki informatike (vodja sekcije Franci Pivec)
I. Študentska sekcija (vodja sekcije Mojca Indihar Štemberger)
J. Informatizirane storitve v upravi (vodja sekcije Tomaž Banovec)

Da bi zmanjšali obseg zbornika, se je programski odbor odločil, da bodo v zborniku v celoti objavljeni samo izbrani referati. Vsi drugi le v povzetku, zato naj avtorji pripravijo daljši povzetek (eno stran ali 500 besed). Vsi referati pa bodo po zaključku posvetovanja na voljo v celoti na domači strani društva. Tako bodo zainteresirani lahko dobili vsebino v elektronski obliki, vsi skupaj pa bomo prihranili nekaj papirja.

Na osnovi prvega obvestila in vabila k sodelovanju smo prejeli že številne povzetke. Vendar se je programski odbor odločil, da zbiranje prispevkov podaljša in delno spremeni pogoje njihovega objavljanja. Zato lahko tudi referenti, ki še niso poslali povzetka, pošljejo referate v pisni obliki in na disketi do 15. 1. 2001, s čimer bodo omogočili vodjem sekcij, da jih pregledajo in avtorjem predlagajo določene spremembe. Zadnji rok za oddajo referatov je 20. februar 2001. Referatov, ki bodo prispeli po tem roku, ne bomo mogli uvrstiti v posvetovanje.

Referate pošljite v predpisani obliki na papirju in na disketi na naslov Slovensko društvo Informatika, DSI 2001, Vožarski pot 12, 1000 Ljubljana. V spremnem dopisu navedite svoj polni naslov, telefon in elektronski naslov. Navedite tudi, v katero od zgoraj navedenih sekcij predlagate, da vključimo referat.

Navodila za oblikovanje referatov najdete na naslovu **www.drustvo-informatika.si.** Z vsa vprašanja se obračajte na isti naslov.

Na svidenje v Portorožu!

* *

Vsem članom Slovenskega društva INFORMATIKA, sponzorjem revije in drugim bralcem želimo

Srečno 2001

***

Uredništvo revije

*uporabna*INFORMATIKA

* *

Naslov	Datum	Kraj	Organizator	Informacije
Sodobne rešitve za javno upravo na področju informatike	30. 1. 2001	Ljubljana	Oracle Software, CVI, VUŠ	elizabeta.gruden@oracle.com faks: 01 588 88 01
Dnevi slovenske informatike	18. - 21. 4. 2001	Portorož	Slovensko društvo INFORMATIKA	www.drustvo-informatika.si
The International Conference ISACA	10. - 13. 6. 2001	Paris, FR	Information Systems Audit and Control Association	http:/www.isaca.org/international2001.htm
IFIP WG9.6/11.7 Working Conference on Security and Control of IT in Society - II	15. - 16. 6. 2001	Bratislava, SK	IFIP WG9.6/11.7 Comenius University Bratislava, Slovak Society for Computer Science	Simone Fischer-Huebner simone.fischer-huebner@kau.se
14th Bled Electronic-Commerce Conference	25. - 26. 6. 2001	Bled, SI	Univerza v Mariboru	http://eCom.fov.uni-mb.si
The 9th European Conference on Information Systems, ECIS 2001, Global Co-opeartion in the New Millennium	27. - 29. 6. 2001	Bled, SI	ECIS, UNI MB-FOV	http://www.ECIS2001.fov.uni-mb.si Gricar@uni-lj.si
Int. Conf. on Human-Computer Interaction - INTERACT 2001	9. - 13. 7. 2001	Tokyo, JP	IFIP TC13	pdf00343@nifty.ne.jp
2nd World Conf.on Information Security Education	19. - 21. 7. 2001	Perth, AU	IFIP WG11.8, E.Cowan Univ.	h.armstrong@ecu.edu.au
20th IFIP TC7 Conf. on Modelling and Optimization	23. - 27. 7. 2001	Trier, DE	IFIP TC7, University Trier	sachs@uni-trier.de
Work.Conf.on Realigning Research & Practice in Information Systems Development: The Social and Organizational Perspective	27. - 29. 7. 2001	Boise, ID, US	IFIP WG8.2, Boise State Univ., Northern Illinois Univ.	riswojt2@cobfac.BoiseState.edu http://afis.ucc.ie/ifip2001
7th IFIP World Computer Conf. on Computers in Education	29. 7. - 3. 8. 2001	Copenhagen, DK	IFIP TC3	wcce2001@sek.ddf.dk http://www.wcce2001.dk
8th IFAC/IFIP/IFORS/IEA Symposium on Analysis, Design and Evaluation of Human-Machine Systems	18. - 20. 9. 2001	Kassel, DE	IFAC, IFIP TC13	rosenzweig@vdi.de http://www.imat.maschinenbau.uni-kassel.de/hms2001/index.html
IFIP WG 6.1 Working Conference on Distributed Applications and Interoperable Systems – DAS 2001	17. - 19. 9. 2001	Krakow, PI	IFIP 6.1	http:/www.cs.agh.edu.pl/dais2001/ dais2001-info@cs.agh.edu.pl
Symposium on Information Control Problems in Manufacturing Technologies	24. - 26. 9. 2001	Vienna, AT	IFAC, IFIP TC5	e318@ihrtl.ihrt.tuwien.ac.at
4th IFIP TC-11 WG 11.5. Working Conference on Integrity and Internal Control in IS	15. - 16. 11. 2001	Brussels, Be	IFIP TC-11	http://www.ifip.tu-graz.ac.at/TC11/CONF/IICIS2001
IFIP Congress 2002	25. - 30. 8. 2002	Montreal, CA		George@cips.ca http://www.wcc2002.org

Pristopna izjava

Želim postati član Slovenskega društva Informatika

Prosim, da mi pošljete položnico za plačilo članarine SIT 5.200 (kot študentu SIT 2.400) in me sproti obveščate o aktivnostih v društvu.

(ime in priimek, s tiskanimi črkami)

(poklic)

(domači naslov in telefon)

(službeni naslov in telefon)

(elektronska pošta)

Datum: Podpis:

Včlanite se v Slovensko društvo INFORMATIKA.
Članarina SIT 5.200,- (plačljiva v dveh obrokih) vključuje tudi naročnino za revijo Uporabna informatika.
Študenti imajo posebno ugodnost: plačujejo članarino SIT 2.400,- in za to prejemajo tudi revijo.

Izpolnjeno Naročilnico ali Pristopno izjavo pošljite na naslov:
Slovensko društvo INFORMATIKA, Vožarski pot 12, 1000 Ljubljana.
Lahko pa izpolnite obrazec na domači strani društva
http://www.drustvo-informatika.si

INTERNET ■ INTERNET ■ INTERNET ■ INTERNET ■ INTERNET ■ INTERNET

Vse člane in bralce revije obveščamo,
da lahko najdete domačo stran društva na naslovu:

http://www.drustvo-informatika.si

Obiščite tudi spletne strani mednarodnih organizacij, v katere je včlanjeno naše društvo:
IFIP: **www.ifip.or.at**
ECDL: **www.ecdl.com**
CEPIS: **www.cepis.com**

INTERNET ■ INTERNET ■ INTERNET ■ INTERNET ■ INTERNET ■ INTERNET

Naročilnica

Naročam(o) revijo UPORABNA INFORMATIKA

- ☐ s plačilom letne naročnine SIT 4.600
- ☐ ..... izvodov, po pogojih za podjetja SIT 13.800 za eno letno naročnino in SIT 8.900 za vsako nadaljnjo naročnino
- ☐ po pogojih za študente letno SIT 2.000

Naročnino bom(o) poravnal(i) najkasneje v roku 8 dni po prejemu računa

(ime in priimek, s tiskanimi črkami)

(podjetje)

(ulica, hišna številka)

(pošta)

Datum: Podpis:

UPORABNA INFORMATIKA
ISSN 1318-1882

Ustanovitelj in izdajatelj:
Slovensko društvo Informatika, 1000 Ljubljana, Vožarski pot 12

Glavni in odgovorni urednik:
Mirko Vintar

Uredniški odbor:
Dušan Caf, Aljoša Domjan, Janez Grad, Andrej Kovačič, Tomaž Mohorič,
Katarina Puc, Vladislav Rajkovič, Ivan Rozman, Niko Schlamberger, Ivan Vezočnik, Mirko Vintar

Tehnična urednica: Katarina Puc

Oblikovanje: Zarja Vintar, Dušan Weiss, Ada Poklač
Naslovnica: Zarja Vintar

Tisk: Prograf
Naklada: 700 izvodov

Revija izhaja četrtletno. Cena posamezne številke je 3.500 SIT.

Letna naročnina za podjetja SIT 13.800, za vsak nadaljnji izvod SIT 8.900.
Letna naročnina za posameznika SIT 4.600, za študente SIT 2.000.

Sodobne rešitve za javno upravo na področju informatike

30. januarja 2001

Na Visoki upravni šoli, Gosarjeva ul. 5 v Ljubljana

Oracle E- Teme	
Predstavljene bodo sodobno zasnovane rešitve, ki se uporabljajo v Evropski uniji in v Sloveniji na področju državne in javne uprave.	☑
Predstavljene bodo strategije Centra vlade za informatiko s področja elektronske državne uprave.	☑
Profesor Visoke upravne šole v Ljubljani, dr. Mirko Vintar, bo predstavil strategijo razvoja elektronske javne uprave.	☑

Več informacij o seminarju dobite na naslovu: Oracle Software, d.o.o., Dunajska 156, Ljubljana, telefonska številka: 01 588 88 00. Prijave bodo možne po 5. januarju 2001 na spletnih staneh www.oracle.si ali po fax-u: 01 588 88 01.

ORACLE®

SOFTWARE POWERS THE INTERNET™

© 2000 Oracle Corporation. All rights reserved. Oracle is a registered trademark, and Software Powers the Internet is a trademark or registered trademark of Oracle Corporation. Other names may be trademarks of their respective owners.

www.oracle.si, www.oracle.com