Eksplicitna in implicitna uporaba vzorcev pri razvoju informacijskih rešitev

Marjan Heričko, Simon Beloglavec, Matjaž B. Jurič, Boštjan Kežmah
Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Inštitut za informatiko, Smetanova 17, 2000 Maribor
e-pošta: marjan.hericko@uni-mb.si

Povzetek

Uporaba komponent in ogrodij ter z njimi povezanih vzorcev je temelj sodobnega razvoja informacijskih rešitev. Prispevek pokaže uporabo nekaterih osnovnih načrtovalskih vzorcev v sklopu sodobnih programskih jezikov ter nakaže, da lahko ločimo med implicitno in eksplicitno uporabo vzorcev. Implicitna uporaba pomeni, da se razvijalec ne zaveda celovite interne strukture vzorca, a lahko kljub temu uspešno izkorišča pridobitve, ki izhajajo iz uporabe vzorca. Eksplicitna uporaba zahteva celovito razumevanje in poznavanje vzorca. Raziskava je pokazala, da apliciranі vzorci interno – po svoji strukturi – večinoma niso preprosti, tudi zato, ker jih lahko, podobno kot komponente in ostale gradnike, združujemo ter tako oblikujemo sestavljene vzorce. Ugotovitve kažejo, da lahko tudi nad vzorci uporabimo vse tri temeljne tipe abstrakcij: klasifikacijo, agregacijo in generalizacijo. Prispevek pokaže, da je za uspešen razvoj in uporabo komponent, predvsem strežniških, nujno poznavanje in razumevanje množice načrtovalskih vzorcev. Vloga in način uporabe vzorca pa določata potrebni nivo poznavanja in razumevanja vzorca.

Abstract

Explicit and Implicit Use of Software Patterns

The application of components and frameworks with corresponding design patterns is basic for contemporary software development. This article demonstrates the usage of fundamental design patterns in modern programming languages and points out the difference between implicit and explicit usage of design patterns. Implicit usage means that the developer is not aware of the whole structure of the design pattern, but he can neverthless successfully exploit its benefits. Explicit usage requires in-depth understanding of the design pattern. Our research has shown that the internal structure of used patterns is not simple. The fact is that design patterns can be combined into compound patterns in a similar way as other software components. Our findings have shown that it is possible to use all three basic types of abstractions with design patterns: classification, aggregation and generalization. The article shows that awareness and understanding of appropriate design patterns are crucial for successful development and application of software components, especially server-side components. The role and manner of the design pattern usage define the required level and depth of understanding of it.

1 Uvod

Naša zmožnost zagotavljanja ustreznih programskih in informacijskih rešitev že od samih začetkov računalništva temelji na abstrakciji – bodisi na nivoju abstrahiranja sklopov in elementov strojne opreme, abstrahiranja na nivoju sistemske ali aplikativne programske opreme ali v smislu abstrakcij na logičnem nivoju internega ustroja programskih rešitev, ki temeljijo na modularnih, objektnih in/ali komponentnih arhitekturah. Abstrakcija je tista, ki omogoča, da obvladujemo kompleksnost, saj z njeno pomočjo skrijemo oz. zanemarimo za določen kontekst nepomembne vidike. Zgodovinski razvoj – od programske logike, zapečene v krmilnih vezjih, do aktualnega stanja - od strojne in programske platforme neodvisne vmesne kode, kot jo npr. zasledimo pri tehnologijah J2EE in .NET, dokazuje, da se nivo abstrakcije nenehno viša. To pomeni, da se skozi profiliranje in specializacijo vlog večji del informatikov vse manj posveča tehničnim in vse bolj poslovnim izzivom. Nenazadnje je to pomemben dosežek in podpira naše poslanstvo – informacijske rešitve morajo biti namenjene podpori doseganja zastavljenih poslovnih ciljev.

Žal pa abstrakcijo v sklopu razvoja informacijskih rešitev večinoma apliciramo le nad konkretnimi komponentami – strojnimi in programskimi. Pri tem uporabljamo tri pomembne vrste abstrakcije, ki so:

- *klasifikacija* – primerke razvrščamo v skupine na osnovi skupnih lastnosti,
- *agregacija* – osredotočimo se na dejstvo, da deli skupaj tvorijo neko celoto, zanemarimo razlike med deli (z agregacijo je tesno povezano ograjevanje oz. skrivanje implementacije) in
- *generalizacija/specializacija* – skupine podobnih množic organiziramo v hierarhije glede na splošnost/specifičnost.

Z uporabo omenjenih abstrakcij lahko oblikujemo tri različne poglede na obravnavan subjekt (npr. problemsko področje, programsko rešitev, komponento), in sicer:

- *konceptualni pogled* - osredotočamo se na identifikacijo tistih gradnikov, ki so relevantni za identifikacijo rešitve v neki domeni, ne ukvarjamo se z nobenim izmed vidikov implementacije rešitve,
- *specifikacijski pogled* - obravnavamo vmesnike gradnike, ne pa interne realizacije rešitev in
- *implementacijski pogled* - obravnavamo vse podrobnosti implementacije oz. realizacije rešitve.

Temelj vsake inženirske discipline je vsekakor ponovna uporaba – vendar ne zgolj na nivoju izgradnje novih sistemov na osnovi obstoječih, že preizkušenih komponent, temveč tudi na osnovi ponovne uporabe na višjih nivojih abstrakcije – na nivoju ponovne uporabe izkušenj in spoznanj. Takšne abstraktne, ponovno uporabne koncepte, ki opisujejo in podajajo izkušnje oz. idejne rešitve, imenujemo vzorci (»patterns«). V sklopu tega prispevka bomo raziskali, kako lahko tudi za vzorce, uporabne v sklopu razvoja informacijskih rešitev, uporabimo in opredelimo prej omenjene vrste abstrakcije – podobno kot to počnemo za konkretne programske in računalniške komponente. Predvsem pa nas zanima širši pomen in vloga vzorcev pri sodobnem razvoju, temelječem na komponentah in ogrodjih. Raziskali smo uporabo vzorcev v osnovnih knjižnicah dveh najsodobnejših programskih jezikov (Java, C#) in ugotavljali, ali obstaja možnost, da vzorce uporabimo implicitno, kar pomeni, da se v bistvu ne zavedamo celovite interne strukture vzorca, a lahko kljub temu uspešno izkoriščamo prednosti, ki jih prinaša uporaba vzorca. Eksplicitna uporaba vzorcev prav nasprotno zahteva dobro poznavanje posameznega vzorca. Zanimalo nas je, ali se morda zahtevnost in nivo eksplicitne uporabe vzorcev veča, če razvijamo rešitve na osnovi komponentnih modelov. Prispevek zaključimo s povzetkom ugotovitev ter smernicami nadaljnjega dela.

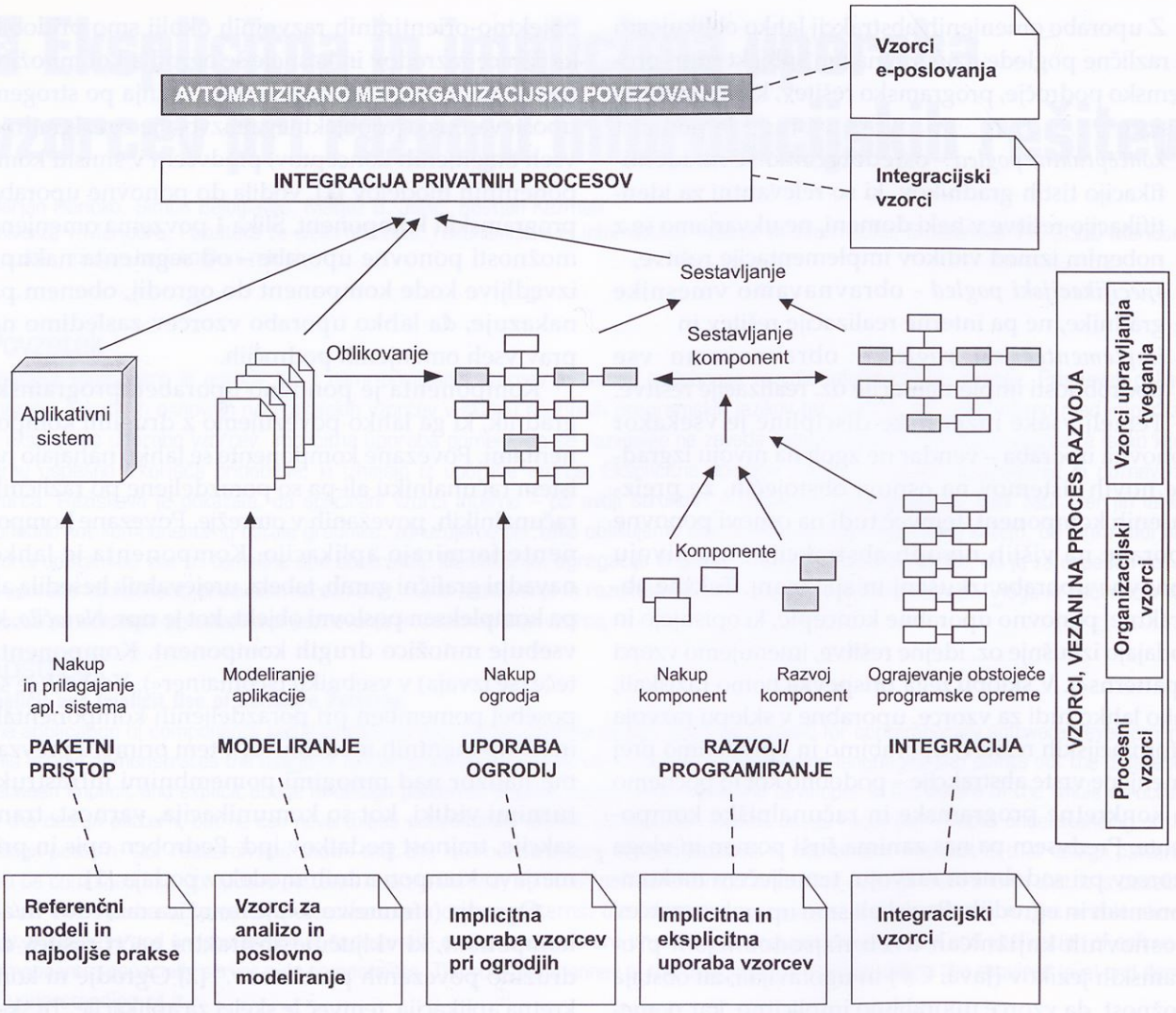
2 Stopnjevanje abstrakcije ponovne uporabe

Pri razvoju informacijskih rešitev že od samih začetkov ponovno uporabljamo celotne pakete izvedljive kode ali pa zgolj dele programske kode – skupine stavkov, ki smo jih vključevali v svoje aplikacije. Konstrukti modularnosti so olajšali delo in zvišali stopnjo ponovne uporabe na nivo metod in funkcij – uveljavile so se knjižnice funkcij in makrojev. Z nastopom objektno-orientiranih razvojnih okolij smo pridobili knjižnice razredov in kasneje še ogrodja kot množice medsebojno povezanih razredov. Težnja po strogem upoštevanju načel objektnega razvoja je z nadgradnjo vseh omenjenih konceptov, predvsem v smislu komponentnih modelov [1], vodila do ponovne uporabe programskih komponent. Slika 1 povzema omenjene možnosti ponovne uporabe – od segmenta nakupa izvedljive kode komponent do ogrodij, obenem pa nakazuje, da lahko uporabo vzorcev zasledimo na prav vseh omenjenih področjih.

Komponenta je ponovno uporaben programski gradnik, ki ga lahko povezujemo z drugimi komponentami. Povezane komponente se lahko nahajajo na istem računalniku ali pa so porazdeljene po različnih računalnikih, povezanih v omrežje. Povezane komponente formirajo aplikacijo. Komponenta je lahko navadni grafični gumb, tabela, urejevalnik besedila ali pa kompleksen poslovni objekt, kot je npr. *Naročilo*, ki vsebuje množico drugih komponent. Komponenta teče (se izvaja) v vsebniku (»container«). Vsebnik je še posebej pomemben pri porazdeljenih komponentah in komponentnih modelih, saj v tem primeru prevzame nadzor nad mnogimi pomembnimi infrastrukturnimi vidiki, kot so komunikacija, varnost, transakcije, trajnost podatkov ipd. Podroben opis in primerjavo komponentnih modelov podaja [1].

Ogrodje (»framework«) je množica razredov in/ali komponent, ki vključuje abstraktni načrt rešitev za družino povezanih problemov." [2] Ogrodje ni konkretna aplikacija, temveč le skelet za aplikacije. Ta skelet oz. ogrodje zaživi, ko razvijalci zagotovijo specifične komponente, nujne za delovanje rešitve. Sicer pa ogrodja vsebujejo več komponent in imajo bolj kompleksen vmesnik kot npr. sestavljene komponente. V vsakem primeru ogrodja zagotavljajo množico storitev, skupnih več aplikacijam. Storitve ogrodja so običajno vezane na določeno področje, npr. izgradnja grafičnih uporabniških vmesnikov, ogrodja poslovnih objektov za posamezne domene (bančništvo, zavarovalništvo, telekomunikacije) ali kot infrastruktura elektronskega poslovanja (npr. ebXML). Sistematično primerjavo in opis ogrodij podaja [3].

Vse bolj razširjena uporaba ogrodij v povezavi z definicijo, ki omenja abstraktni načrt, kaže na to, da lahko pri razvoju uporabimo ideje rešitev iz preteklosti. Predvsem tiste ideje, ki so se že večkrat izkazale in dokazale kot dobre in učinkovite. O tem, kako reševati ponavljajoče se podobne probleme na



Slika 1: **Gradniki ponovne uporabe in uporaba vzorcev pri razvoju informacijskih rešitev**

nekaterih drugih področjih, lahko razberemo iz raznih priročnikov. Nekatere med njimi smo mnogi tudi sami uporabljali, npr. matematični priročnik, fizikalni, kemijski ipd. V bistvu smo v njih poiskali idejno rešitev nekega problema, ki smo jo nato uporabili v svojem kontekstu. Tako oblikovane izkušnje in spoznanja imenujemo vzorec. Ena izmed definicij vzorca je: *Vzorec opisuje problem, ki se večkrat pojavlja v našem okolju, podaja jedro njegove rešitve na tak način, da lahko idejo rešitve uporabimo v več različnih primerih, ne da bi pot od ideje do rešitve prehodili na enak način* [4]. Vzorec torej povezuje problem z idejno rešitvijo v določenem kontekstu.

Čeprav ni razlogov, da se ne bi določene strategije uveljavile in uporabile v poljubni disciplini ali aktivnosti razvoja, pa se v praksi izkaže, da se strategije in tehnike običajno uveljavljajo od spodaj navzgor (najprej na nivoju kodiranja in načrtovanja ter šele kasneje pri poslovnem modeliranju). Podobno velja tudi za vzorce, katerih katalogi so se uveljavili najprej na področju načrtovanja in implementacije in šele kasneje še na vseh ostalih področjih, tudi na področju procesa razvoja. Še vedno pa ostajajo kot najbolj pogosto omenjani načrtovalski vzorci, ki jih poznamo pod imenom vzorci četverice oz. vzorci GoF [5]. Te vzorce lahko delimo po namenu na ustvarjalne,

strukturne in vedenjske, glede na področje uporabnosti pa na objektne in razredne. Podrobna analiza vzorcev, ki smo jo izvedli, je pokazala, da je med navedenimi triindvajsetimi vzorci v bistvu le petnajst osnovnih, dva sta variacija oz. izpeljanka ostalih, štirje so jezikovno odvisni, medtem ko sta dva v bistvu le idioma objektnega razmišljanja [6].

Vzorce lahko torej klasificiramo na različne načine, npr. po namenu, uporabnosti in tudi po tipu aktivnosti, v sklopu katerih so uporabni. Za področje razvoja programske opreme obstajajo vzorci tako za področje implementacije kot načrtovanja, analize pa tudi projektnega vodenja, organiziranja, oblikovanja procesnega modela in modeliranja poslovnih procesov (Tabela I). Klasifikacija vzorcev je torej nujno potrebna, saj bi sicer le s težavo identificirali ustrezni katalog vzorcev in se znotraj kataloga omejili na množico tistih vzorcev, ki so relevantni za naš problem in sprejemljivi v našem kontekstu. Potrebo po klasifikaciji pogojuje tudi rast števila vzorcev, zato ne zadostuje zgolj klasičen opis vzorca, ki običajno vsebuje [5]: ime, namen, motivacijo, uporabnost, vpliv, strukturo, učinek, sorodne vzorce in znane primere uporabe. Sicer pa lahko na spletu zasledimo množico katalogov vzorcev in diskusijskih baz. Ena boljših izhodiščnih točk za področje vzorcev je npr. http://www.cetus-links.org/oo_patterns.html

3 Implicitna in eksplicitna uporaba vzorcev

Če se osredotočimo na temeljne aktivnosti razvoja programskih rešitev – zajemanje zahtev, načrtovanje sistema in implementacijo, lahko ugotovimo, da se je uporaba vzorcev uveljavila predvsem, če že ne izključno, v povezavi z objektnim razvojem. Tudi zato se bomo v nadaljevanju osredotočili na raziskavo uporabe načrtovalskih vzorcev (»design patterns«), implementiranih v sklopu Jave in C#, kot predstavnikov najbolj razširjenih sodobnih jezikov. Nenazadnje je Java s svojim nastankom sredi prejšnjega desetletja bistveno pripomogla k pospešeni vpeljavi in uveljavitvi objektne tehnologije ter komponentnega razvoja kot temeljev razvoja sodobnih informacijskih rešitev. C# te smernice le še dopolnjuje.

Raziskave o pridobitvah uporabe vzorcev večinoma temeljijo na formalizaciji opisa vzorcev, predvsem njihove interne strukture. Na ta način lahko sicer olajšamo ugotavljanje odvisnosti in povezanosti vzorcev [7], žal pa formalni opis ne rešuje temeljnega izziva – kako izboljšati ponovno uporabo izkušenj, opredeljenih kot vzorcev. Ker izkušnje iz prakse kažejo, da vzorci večinoma niso intuitivni in preprosti za razumevanje, nas je zanimalo, ali lahko morda vzorce pri razvoju v Javi in C# uporabljamo spotoma in nehote - torej implicitno, ali pa je za njihovo apliciranje potreben temeljit razmislek in zahtevano poglobljeno razumevanje oz. eksplicitna uporaba vzorcev. Preliminarna ocena stanja glede razumevanja in kompleksnosti posameznih vzorcev načrtovanja GoF, v kateri je sodelovalo dvajset razvijalcev, v povprečju več kot s tremi leti izkušenj z objektnim programiranjem, je namreč pokazala, da so vzorci različno razumljeni in uporabljani. Izmed triindvajsetih vzorcev jih anketiranci namreč kar polovico ne poznajo v podrobnosti [8] – se pa kot razvijalci v Javi oz. C# s temi vzorci nenehno srečujejo.

3.1 Implicitna uporaba vzorcev

Mnogi, ki pri razvoju uporabljajo Javo in C#, se ne zavedajo, da v teh okoljih praktično ni razvoja brez uporabe vzorcev [9]. Razlog za to je dejstvo, da razvijalci te vzorce uporabljajo implicitno, saj so ti ograjeni znotraj razredov bogatega nabora knjižnic oz. paketov, ki jih zagotavlja Java oz. ogrodje .NET. V nadaljevanju bomo omenili le nekatere vzorce, več jih zasledimo v [9,10] za Javo in v [11] za C#.

Vzorec *Edinec* zagotavlja, da lahko za nek razred kreiramo samo en primerek oz. le enega predstavnika. Da to zagotovimo, je potrebno onemogočiti splošen dostop do konstruktorja razreda, dostop do edinega primerka pa nadzorujemo v javni razredni (statični)

Tabela I: **Avtorji oz. skrbniki katalogov znanih vzorcev za posamezne aktivnosti**

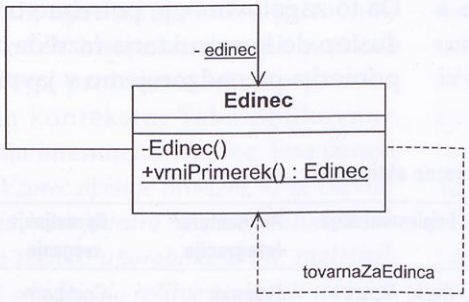
Poslovno modeliranje	Analiza	Načrtovanje	Implementacija	Arhitektura/ Integracija	Upravljanje tveganja	Procesni vzorci
Erikssson&Penker	Coad Fowler	GoF Bus Coplien, Schmidt	Beck, Alpert (Smalltalk) Cooper (C#) Grand (Java)	Schmidt Jurič	Cockburn	Ambler Coplien

metodi, ki po potrebi poskrbi tudi za kreiranje edinega primerka – če ta seveda še ne obstaja. Na sliki 2 je podan primer kode, ki kaže idejo te rešitve. Konkretna uporaba tega vzorca pomeni, da bo razvijalec, razen preimenovanja razreda, temu dodal še specifične atribute ter ostale metode, ki zagotavljajo storitve objekta kot edinega predstavnika tega razreda. V tem primeru govorimo o eksplicitni uporabi vzorca. Na takšen način so ga tudi uporabili snovalci in razvijalci razredov, kot je npr. *RunTime* v Javi oz. *Application* v .NET. V sklopu implicitne uporabe se razvijalci razen tega, da ustrezno referenco na edini primerek pridobijo preko razredne (statične) metode omenjenih razredov, niti ne zavedajo uporabljenega vzorca.

Naslednje, ne najbolj znano dejstvo je, da so v Javi in C# vsi objekti razreda *String* v bistvu nespremenljivi, kar pomeni, da niza znakov ne moremo spremeniti. Pri implementaciji razreda *String* namreč srečamo aplikacijo vzorca *Nespremenljiv*. Ta vzorec je uporaben v različnih kontekstih, vsem pa je skupno, da obstaja primerek vzorca, ki je uporabljen v več razredih, hkrati ima objekt statično naravo – praviloma se ne spreminja. Tako C# kot Java namreč za potrebe dela z objekti tipa *String* vzdržujeta fond objektov nizov, ki so bili oblikovani in so v določenem trenutku še aktualni za sistem. Vzorec povečuje robustnost objektov, ki si delijo skupno referenco na en sam objekt, in zmanjšuje kompleksnost hkratnega dostopa do objekta. To dosega s prepovedjo spremembe kateregakoli podatka, ki določa stanje objekta, potem ko je bil le-ta kreiran. Podobno idejo zasledimo tudi pri vzorcu *Zrno*. S tem ko uporabljamo objekte razreda *String*, se torej soočamo z implicitno uporabo vzorcev *Zrno* in *Nespremenljiv*. Aplikacijo vzorca *Nespremenljiv* pa npr. v jeziku C# srečamo tudi pri objektih – delegatih.

Pri razvoju v Javi in C# pogosto srečamo uporabo vzorca *Iterator*, še najbolj je uporaba iteratorjev vidna pri delu z množicami objektov pri uporabi t. i. ogrodja kolekcij (*Collection Framework*). Vmesnik *Collection* igra vlogo vmesnika do implementacije kolekcije oz. množice objektov, ki zna oblikovati iteratorje za prehod nad objekti, ki jih vsebuje. Osnovni cilj uporabe tega vzorca pa je, da se sprehodimo prek vseh primerkov oz. objektov v kolekciji, ne oziraje se na njeno implementacijo, ki je lahko npr. primerek razreda *Vector, HashTable, Set* ali poljuben drug objekt, ki zagotavlja implementacijo vmesnika. S proženjem metode *iterator()* namreč pridobimo referenco na ustrezni vmesnik, ki omogoča, da se z metodo *next()* pomaknemo na naslednji objekt, objekt zbrišemo z *remove()* in/ali prožimo metodo *hasNext()*, da preverimo, ali smo obdelali vse objekte v kolekciji. Uporaba tega vzorca zagotavlja neodvisnost razreda odjemalca od same implementacije kolekcije. Nenazadnje pa lahko prek različnih implementacij vmesnika *Iterator* zagotovimo različne poti in zaporedja obdelave/prečkanja objektov ter tako vplivamo tudi na zmogljivosti sistema.

Kot prikaz uporabnosti vzorca *Iterator* služi v Javi razred *Collections*, ki zagotavlja splošne pripomočke za urejanje in kopiranje kolekcij objektov. Še najpogosteje pa razvijalci uporabljajo izpeljanko vzorca Iterator, ki se imenuje *Enumeration*. V jeziku C# zasledimo apliciranje vzorca *Iterator* pri poljih ter pri vmesnikih kot so *ICollection, IList,* torej za razrede, ki podpirajo vmesnik *IEnumerator*. C# gre pri tem celo korak dlje – rezervirana beseda *foreach* omogoča, da na enostaven način obdelamo vse objekte v neki kolekciji (primer 1).




```
public class Edinec {
  private static Edinec edinec;
  private Edinec() { }

  public static Edinec vrniPrimerek()
  {
    if (edinec == null)
      edinec = new Edinec();
    return edinec;
  }
}
```

Slika 2: **Struktura vzorca *Edinec* in skelet implementacije v C# oz. Javi**

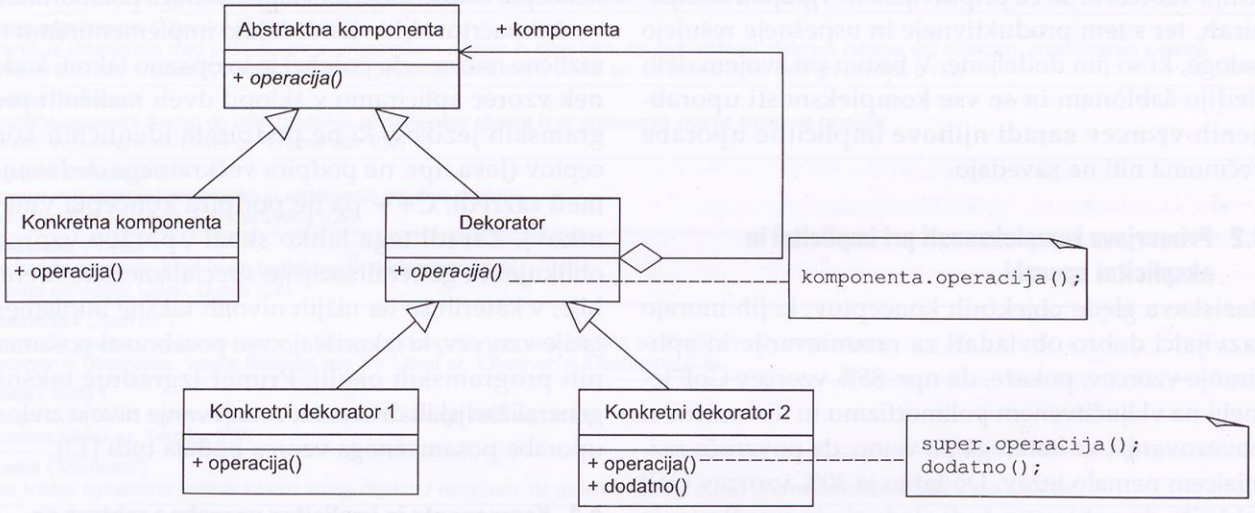
Java	C#
Iterator i = kolekcija.iterator(); while (i.hasNext()) System.out.println(i.next());	foreach (i in kolekcija) Console.WriteLine(i);

Primer 1: **Primer koriščenja vzorca *Iterator* v Javi in C#**

Pri delu z vhodno-izhodnimi tokovi preprosto ne moremo zaobiti uporabe vzorca *Dekorator* (slika 3). Ta omogoča, da objektu dinamično, v času izvajanja, dodajamo funkcionalnosti. Na osnovi hierarhije dedovanja namreč oblikujemo strukturo, ki omogoča, da pri oblikovanju z novimi storitvami obogatenih objektov v konstruktorju navedemo objekt, ki mu dodajmo funkcionalnost, veriga objektov pa prek delegiranja in uporabe vključitvenega polimorfizma rezultira v širši funkcionalnosti. Druga možnost bi bila, da kreiramo vse možne kombinacije podrazredov, za kar pa potrebujemo še podporo večkratnemu dedovanju med razredi, ki pa je Java in C# ne omogočata (večkratno dedovanje je podprto le med vmesniki).

Pri vhodno izhodnih operacijah večkrat naletimo na zahtevo, da potrebujemo funkcionalnost dveh razredov, npr. razreda za delo z datotekami in razreda, ki omogoča branje podatkov osnovnih tipov. Sama koda (primer 2) je precej enostavna, a le kot posledica dejstva, da je v ozadju oblikovana ustrezna struktura, ideja katere je prikazana na sliki 3. Omenjeni vzorec je običajno osnova tudi za ustrezno serializacijo objektov, brez katere ni mogoče zgraditi sodobnih sistemov, ki temeljijo na komunikaciji porazdeljenih objektov.

Pomemben vidik vsakega objektnega sistema in okolja je komunikacija med objekti oz. dogodkovni model, ki to komunikacijo omogoča. Dogodkovni model, ki je temeljil na dedovanju, je z verzijo 1.1 v Javi nadomestil dogodkovni model, ki temelji na delegiranju. Komponenta, ki je izvor dogodkov, mora o dogodku obvestiti vse objekte, ki so se registrirali kot poslušalci za to vrsto dogodka (pogoj je, da zagotavljajo ustrezno funkcionalnost oz. implementirajo ustrezni vmesnik). V bistvu lahko nek objekt nastopa v vlogi poslušalca različnih tipov dogodkov. Vzorec *Delegat* omogoča, da namesto oblikovanja novega



Slika 3: **Struktura vzorca *Dekorator***

Java	C#
FileReader fr = new FileReader ("a.txt"); BufferedReader br = new BufferedReader(fr); int i = br.read();	FileStream fr = new FileStream("a.txt",FileMode.Open); BufferedStream br = new BufferedStream(fr); int i = br.ReadByte();

Primer 2: **Implicitna uporaba vzorca *Dekorator* v Javi**

podrazreda, razred zgolj delegira odgovornost za izvedbo neke operacije drugemu objektu. Delegiranje je pogosto s konceptualnega vidika primernejše kot uporaba dedovanja, še posebej v primerih, ko imamo opravka z asociacijami tipa »je-vloga«.

Podrobnejši razmislek pokaže, da smo vzorec *Delegat* srečali že v sklopu vzorca *Dekorator*, saj je abstraktni razred s pomočjo delegiranja odgovornost za izvedbo operacije prenesel drugemu objektu. To dokazuje, da lahko vzorce obravnavamo na podoben način kot komponente – iz manjših lahko sestavljamo večje, bolj zapletene, ki rešujejo oz. podajajo ideje rešitev kompleksnejših problemov. Uporabo vzorca *Delegat* v jeziku C# oz. okolju .NET v splošnem olajšata že vgrajena razreda *Delegat* in *Multicast-Delegat*. Jezik C# pa uporabo tega vzorca še dodatno poenostavi, saj lahko uporabimo rezervirano besedo *delegat* ter se tako izognemo neposrednemu delu z razredi.

Celovitejši seznam ter obsežnejša razlaga vzorcev, ki jih zasledimo že v osnovni Javi in ogrodju .NET, je na voljo v [9,10,11]. Vsekakor pa že nekaj preprostih primerov, ki smo jih podali, dokazuje, da lahko tudi manj izkušeni razvijalci v Javi in okolju .NET izkoristijo vzorce, ki so že pripravljeni in vgrajeni v knjižnicah, ter s tem produktivneje in uspešneje rešujejo naloge, ki so jim dodeljene. V bistvu pri svojem delu sledijo šablonam in se vse kompleksnosti uporabljenih vzorcev zaradi njihove implicitne uporabe večinoma niti ne zavedajo.

3.2 Primerjava kompleksnosti pri implicitni in eksplicitni uporabi

Raziskava glede objektnih konceptov, ki jih morajo razvijalci dobro obvladati za razumevanje in apliciranje vzorcev, pokaže, da npr. 85% vzorcev GoF temelji na vključitvenem polimorfizmu in dinamičnem povezovanju, za katerega pa vemo, da povzroča razvijalcem nemalo težav. Dodatno je 30% vzorcev GoF takšnih, da zahtevajo tudi sledenje in upoštevanje ustreznega zaporedja proženja metod. Analiza med razvijalci [8] je pokazala, da razvijalci vzorce GoF pogosto medsebojno zamenjujejo in da jih več kot polovico ne razumejo v tolikšni meri, da bi ta omogočala njihovo eksplicitno uporabo. Kot kažejo predstavljeni primeri v tem poglavju, pa se s prav istimi vzorci implicitno srečujejo pri uporabi standardnih knjižnic sodobnih jezikov. Ugotovitev dodatno potrjuje spoznanje, da eksperti v določenih domenah pri modeliranju in razvoju sistemov vzorce uporabljajo v nesistematični obliki oz. se njihove uporabe ne zavedajo, dokler jim ta ni eksplicitno predstavljena oz. dokler vzorci niso ustrezno dokumentirani [12].

Strukturo vzorcev GoF smo podrobneje analizirali tudi s pomočjo uveljavljenih objektnih metrik. Podrobni rezultati so predstavljeni v [8]. Za potrebe prispevka se osredotočimo le na že predstavljen vzorec *Dekorator*. Metrična analiza apliciranja – torej eksplicitne uporabe vzorca *Dekorator* pokaže, da imamo opravka z najmanj 4 razredi, povprečna globina dedovanja je 2, delež predefiniranih metod je 33%. Ugotovimo lahko, da pri implicitni uporabi – glej primer 2, sploh ni potrebno poznavati dedovanja in polimorfizma – zadostuje že, da sledimo ustreznima korakoma – oblikujemo osnovni objekt, ki ga nato posredujemo kot argument v konstruktor, s pomočjo katerega oblikujemo objekt dekorator. Ugotovimo lahko, da implicitna uporaba vzorcev, npr. pri tokovih, omogoča veliko stopnjo abstrakcije, razvijalec pa se sooča z bistveno manjšo kompleksnostjo, kot če bi sam oblikoval ustrezno strukturo dedovanja. V kolikor bi želeli že apliciran vzorec *Dekorator* še razširiti, pa bi bilo potrebno delno poznavanje strukture in s tem tudi konceptov dedovanja ter vključitvenega polimorfizma.

Isti načrtovalski vzorec lahko implementiramo na različne načine – še posebej je to opazno takrat, kadar nek vzorec apliciramo v sklopu dveh različnih programskih jezikov, ki ne podpirata identičnih konceptov (Java npr. ne podpira večkratnega dedovanja med razredi, C++ pa ne podpira koncepta vmesnikov). Zaradi tega lahko skozi uporabo vzorcev oblikujemo generalizacijske/specializacijske hierarhije, v katerih so na nižjih nivojih takšne implementacije vzorcev, ki izkoriščajo vse posebnosti posameznih programskih okolij. Primer izgradnje takšnih generalizacijskih dreves ter ocenjevanje nivoja zrelosti uporabe posameznega vzorca podaja tudi [13].

4.1 Komponente in implicitna uporaba z zahtevo po poznavanju vzorca

Prehod na komponentni razvoj pogojuje dodatna znanja, saj že npr. uporaba javanskega komponentnega modela (Java Beans) zahteva poznavanje in sledenje določenim vzorcem. Omenili smo, da aktualni dogodkovni modeli (tako v Javi kot v .NET) temeljijo na delegiranju. Če npr. pri oblikovanju ustreznega odziva pri uporabi grafičnega uporabniškega vmesnika, razen načina za implementacijo

Edinec ("Singleton")
zagotavlja, da ima razred en sam primerek
RunTime (Java) oz. Application (C#)

Abstraktna tovarna ("Abstract Factory")
zagotavlja vmesnik za kreiranje družin povezanih ali odvisnih objektov brez določitve njihovih konkretnih razredov
Toolkit in URLStreamHandlerFactory (Java) oz. XmlDocument (C#)

Graditelj ("Builder")
loči konstrukcijo kompleksnega objekta od njegove predstavitve – isti konstrukcijski proces lahko oblikuje različne predstavitve
ModelTreeBuilder (Java) oz. EventArgs (C#)

Prototip ("Prototype")
določa vrsto objektov z uporabo prototipnega primerka, na osnovi katerega se oblikujejo kopije
Cloneable (Java) oz. IClonable(C#)

Tovarniška metoda ("Factory Method")
definira vmesnik za kreiranje objekta, podrazredom pa prepusti odgovornost za določitev konkretnega razreda, za katerega oblikuje primerek
Component.getGraphics(), Toolkit.getImage(), Collator.getInstance() (Java) oz. PageHandlerFactory.gethandler() (C#)

Adapter ("Adapter")
preoblikuje vmesnik razreda v vmesnik, ki ga pričakuje odjemalec
JDBCDriver, Adapter-ji poslušalcev (Java) oz. RCW (Run-Time Callable Wrapper) in CCW(COM callable Wrapper) (C#)

Dekorator ("Decorator")
omogoča dinamično dodajanje odgovornosti
FilterReader ipd. pri delu z vhodno/izhodnimi tokovi v Javi in C#

Namestnik ("Proxy")
zagotavlja nadomestni objekt, ki kontrolira dostop do originalnega objekta
RMIProxy (Javi) oz. Remoting, OleDbConnection (C#)

Most ("Bridge")
omogoča, da se vmesnik in implementacija ločeno spreminjata
Component in ComponentPeer (Java) oz. IComponent in ISite (C#)

Fasada ("Façade")
zagotavlja enoten vmesnik za množico vmesnikov v podsistemu
URL (Java) oz. DataSet (C#)

Zrno ("Flyweight")
uporaba skupnega primerka za učinkovito podporo večjega števila manjših objektov
String, fond povezav, sejnih zrn pri EJB (Java) oz. FontFamily (C#)

Kompozicija ("Composite")
združuje objekte v drevesne strukture, ki prikazujejo hierarhije celota-deli
Vsebniki (Container) in grafični gradniki (Component) v Javi oz. TreeNode, Node, Frame, Control (C#)

Veriga odgovornosti ("Chain of Responsibility")
zmanjšuje sklopljenost med pošiljateljem in prejemnikom zahteve in sicer tako, da daje možnost obdelave zahteve več objektom, organiziranim v verigo
omejene lastnosti (VetoableChange), getFont(), getBackgroundColor() (Java) ter obravnava izjem tako pri Javi kot pri C#

Iterator ("Iterator")
omogoča zaporedni dostop do vseh gradnikov sestavljenega objekta brez poznavanja interne strukture množice
Iterator in Enumeration (Java) oz. IEnumerator (C#)

Pogajalec ("Mediator")
ograjuje in nadzoruje komunikacijo in sodelovanje množice objektov
FocusManager (Java) oz. CommonDialog (C#)

Obiskovalec ("Visitor")
predstavlja operacijo, ki se izvaja nad elementi objektne strukture
ChangeFontVisitor (Java), Queue (C#)

Opazovalec ("Observer")
vzpostavlja povezavo ena-proti-mnogo, v primeru spremembe izvornega objekta so obveščeni vsi odvisni objekti
Observer in Observable, vezane lastnosti java zrn (Java) oz. IObserver in IObservable (C#)

Stanje ("State")
omogoča, da objekt spremeni odziv v odvisnosti od svojega internega stanja
Graphics (Java) oz. TextBox (C#)

Spomin ("Memento")
brez kršitve ograjevanja zajame interno stanje objekta z namenom, da ga lahko kasneje obnovi binarna serializacija tako v Javi kot v C#

Interpreter ("Interpretor")
za dani jezik definira predstavitev slovnice in intrerpreterja stavkov tega jezika
Matcher in Patterns v javax.regex (Java) oz. XMLValidatingReader (C#)

Komanda ("Command")
ogradi zahtevo oz. ukaz kot objekt
UndoableEdit, AccessibleAction (Java) oz. SQLCommand, UndoContext(C#)

Strategija ("Strategy")
definira in ogradi družino algoritmov in tako zagotovi njihovo zamenljivost
LayoutManager, CheckInputStream (Java) oz. IRemotingFormatter, CryptoAPITransform (C#)

Metoda predloge ("Template Method")
definira skelet algoritma operacije, pri čemer odgovornost za določene korake prepusti podrazredom
Metode setter in getter pri entitetnih zrnih tipa CMP (EJB), večina uporabe virtualnih metod, tako v Javi kot v C#

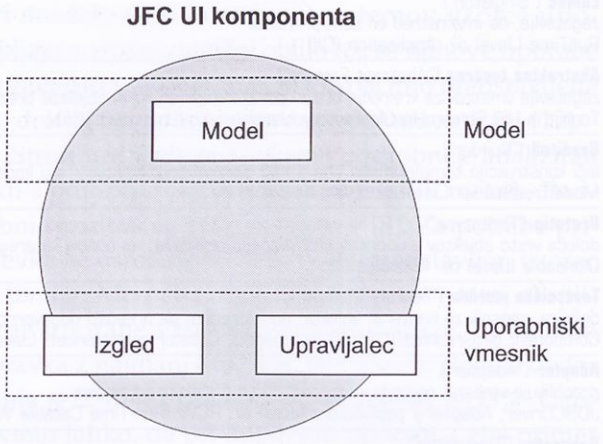
Tabela II: **Pregled vzorcev GoF s primeri apliciranja v Javi in C# oz. .NET**

vmesnika (ali uporabe adapterja) ter registriranja objekta – poslušalca, ni potrebno poznati podrobne interne zgradbe in uporabe dogodkovnega modela, pa v primeru, ko sami razvijamo komponente – zrna, moramo poznati in aplicirati tudi vse podrobnosti omenjene zasnove. Komponente namreč preko ustreznih metod za registracijo poslušalcev določajo dogodke, ki se lahko pojavijo na komponenti. Prek tega tudi razvojna okolja identificirajo dogodke posamezne komponente in omogočajo dogodkovno vodeno povezovanje komponent. Pri izvajanju komponent pa zasledimo uporabo ustvarjalnega vzorca, imenovanega *Prototip*.

Kot primer navedimo knjižnico *Swing*, ki je del JFC (Java Foundation Classes), ponuja pa nabor gradnikov (javanskih komponent oz. zrn) za izgradnjo grafičnih uporabniških vmesnikov. Razen tega, da pri izkoriščanju te knjižnice uporaba nekaterih vzorcev izhaja že iz dejstva, da gre za javanske komponente, uspešnega dela z gradniki knjižnice Swing ni mogoče pričakovati brez poznavanja arhitekture vzorca MVC (Model-View-Controller). V praksi ga zelo pogosto, ko je implementiran v določenem okolju, imenujejo kar ogrodje. Razvoj, uporaba in izvajanje Swing komponent temelji na dejstvu, da je potrebno zagotoviti tri segmente: *model, pogled* in *upravljavca*, kjer sta izgled in upravljavec zaradi močne soodvisnosti združena v enem objektu (delegat). Osnovni cilj je ločitev podatkov od njihovega prikaza, kar je ena izmed splošnih smernic pri razvoju informacijskih rešitev.

Poenostavljeno to pomeni, da je manipulacija in obravnava podatkov, ki se prikazujejo s pomočjo neke grafične komponente, ločena od samega prikaza. Implementacija razreda, ki služi kot osnova za oblikovanje objekta z modelom, pa je kljub vsemu precej odvisna od kompleksnosti komponente za prikaz. V primeru uporabe komponente *JTable* je tako npr. potrebno poznati in implementirati šest vmesnikov in devet razredov (slika 4).

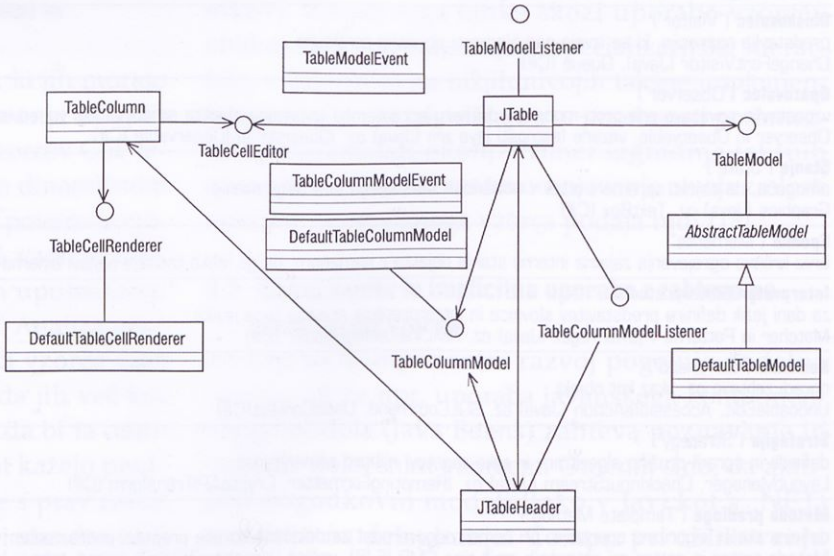
Uporabnost in prednosti uporabe arhitekturnega vzorca MVC se pokažejo v primerih, ko je potrebno modelu zamenjati izgled ali izgledu zamenjati model, zagotoviti različne poglede na iste podatke, porazdeliti



Slika 4. **Ideja vzorca MVC, uporabljena v JFC**

aplikacijo ali zaradi povečanja zmogljivosti zagotoviti dodatne objekte s poslovno logiko oz. modelom.

Podobno kot pri vzorcu *Dekorator* je tudi vzorec MVC v bistvu sestavljen iz manjših vzorcev, imenovanih *Opazovalec, Kompozicija* in *Strategija*, kar intuitivno potrjuje potrebo po agregaciji vzorcev – formalne dokaze, da lahko vzorce združujemo v kompleksnejše vzorce podajata [6,7]. Tudi glede vzorcev je eden izmed pomembnih vidikov uspešne uporabe ustrezno dokumentiranje ponovno uporabnih gradnikov. Večinoma se raziskovalci osredotočajo na obliko dokumentiranja vzorcev za potrebe njihove eksplicitne uporabe. V [6] smo tudi sami definirali formalno



Primer 4: **Razredi in vmesniki, povezani z *JTable* v Javi**

osnovo za opis vzorcev. Pri tem smo se osredotočili predvsem na formalno predstavitev vzorcev in njihovih medsebojnih relacij, tudi z namenom definiranja mehanizmov povezovanja in integracije, torej za eksplicitno uporabo. Izkazalo se je, da lahko principe kombiniranja in sestavljanja novih komponent iz manjših komponent, v precejšnji meri izkoristimo tudi na področju vzorcev in le-te povezujemo ter tako oblikujemo sestavljene vzorce, ki predstavljajo idejo rešitev kompleksnejših nalog in problemov.

4.2 Vzorci pri razvoju strežniških poslovnih komponent

Glede na dejstvo, da že uporaba »navadnih« komponent zahteva dobro poznavanje mnogih vzorcev, je pričakovati, da bodo potrebe po razumevanju in uporabi vzorcev na nivoju strežniških komponent, npr. javanskih zrn EJB (Enterprise Java Beans) ali .NET komponent, še večje. Izkaže pa se, da strežniške tehnologije razvijalca v precejšnji meri razbremenijo nekaterih skrbi in obveznosti, kot so npr. zagotavljanje trajnosti, transakcijskega obnašanja, varnosti. Te vidike sistema določamo deklarativno, kar pa pomeni, da te mehanizme zagotovijo strežniška okolja (aplikacijski strežniki s pripadajočimi vsebniki), in sicer z apliciranjem ustreznih vzorcev, kot so npr. *Abstraktna tovarna*, *Prototip*, *Zrno* (za fonde povezav do virov), *Namestnik* (za potrebe proženja oddaljenih metod) ipd. Več tovrstnih vzorcev smo opisali v [14,15]. Na drugi strani pa lahko že z apliciranjem preprostih vzorcev rešimo nekatere izzive, ki jih prinaša razvoj porazdeljenih sistemov, npr. zagotavljanje ustrezne zmogljivosti in odzivnih časov. Kot primer navedimo le nekatere izmed »preprostih« vzorcev, ki jih lahko apliciramo v ta namen [15]:

- *Sejna fasada* (Session facade) – vzorec združuje aplikacijske funkcije v en poenostavljen vmesnik, ki je namenjen izključno odjemalcu ter s tem skrije podrobnosti implementacije poslovnega opravila;
- *Poslovni delegat* (Business Delegate) je v bistvu poseben primer vzorca *Delegat*, saj med oddaljeni poslovni objekt in odjemalca vstavi nov vmesnik, katerega namen je skriti kompleksnost oddaljene komunikacije;
- *Vrednostni objekt* (Value object) izboljšuje izmenjavo podatkov med različnimi nivoji porazdeljenega sistema – osnovna ideja je, da namesto posameznega podatka prenesemo večji nabor podatkov;
- *Dostopni podatkovni objekt* (Data Access Object) narekuje realizacijo posebnih virtualnih komponent, ki ogradijo specifično programsko kodo za dostop do posameznih zunanjih virov.

Ko ob besedi *vzorec* uporabljamo atribut »preprost« je treba upoštevati, da se pogosto izkaže, da je ideja rešitve nekega problema v bistvu enostavna – seveda takrat, ko jo poznamo. Izkaže se tudi, da zahtevni problemi pogojujejo kombiniranje in združevanje delnih rešitev, tudi idej rešitev. Ugotovili smo že, da lahko vzorce združujemo v večje gradnike. Razvijalci v vlogi implicitnega uporabnika vzorcev morajo zaradi načel komponentnega razvoja poznati in razumeti le del celotne komponentne zgradbe vzorca. Komponentni modeli spodbujajo tudi specializacijo vlog pri razvoju. Tako so tipične vloge pri razvoju strežniških komponent [16]: razvijalec komponent, sestavljalec komponent, nameščevalec, administrator zrn in ponudnik vsebnika/strežnika.

Razvijalec zgradi komponente, jih prevede in jih posreduje v obliki posebne arhivske datoteke EAR-JAR (Enterprise ARchive-Java ARchive) pri EJB oz. zbirne datoteke (»assembly«) pri ogrodju .NET. Razvijalec je odgovoren za definicijo vmesnikov komponent in implementacijo razredov s poslovno logiko. Prav tako mora priskrbeti posebne opisne datoteke – deskriptorje, ki vsebujejo parametre komponent (nanašajo se na vire, ki jih uporablja komponenta). Rezultate dela razvijalca komponent prevzame sestavljavec, ki poveže posamezne dele v eno namestitveno enoto, ki je pripravljena za namestitev na aplikacijski strežnik. V tako namestitveno enoto običajno združimo še spletne komponente (npr. JSP strani in servlete oz. ASP.NET strani). Nameščevalec nastavi pripravljeno enoto na specifičen aplikacijski strežnik, pri tem uporablja orodja, ki so predpisana s strani specifičnega aplikacijskega strežnika. Administrator komponent je odgovoren za nastavitev parametrov samega aplikacijskega strežnika, kot tudi parametrov vsebnika zrna. Specifikacija navaja ločeno še ponudnika strežnika in vsebnika, vendar sta v praksi ti dve vlogi združeni. Pri samem načrtovanju je specializiran načrtovalec nujno potreben, saj se razred s poslovno logiko iz konceptualnega modela preslika v več razredov, ko razgrajujemo model v implementacijske podrobnosti. V bistvu smo že omenili, da mora ponudnik vsebnika aplicirati mnoge vzorce (od *Abstraktne tovarne* do *Namestnika*), podobno velja za načrtovalca in razvijalca

komponent, ki vzorce eksplicitno uporabljata. Sestavljavec komponent v večini primerov shaja z razumevanjem vzorcev, potrebnih za implicitno uporabo, ob sestavljanju kompleksnejših komponent pa tudi on sledi ustreznim vzorcem, ki jih uporabi eksplicitno ali z zadostnim razumevanjem vsaj implicitno.

6 Sklep in smernice raziskav

Uporaba ogrodij komponent in z njimi povezanih vzorcev je ključ sodobnega razvoja tako na nivoju zasnove informacijskih rešitev kot tudi na nivoju definiranja in informatizacije medorganizacijskih povezav in poslovanja. Komponente, ogrodja in vzorci naj bi olajšali razvoj informacijskih rešitev. Podrobna raziskava pokaže, da vzorci interno, po svoji strukturi, niso preprosti, tudi zato, ker jih lahko podobno kot komponente in ostale gradnike združujemo in kombiniramo ter tako oblikujemo sestavljene vzorce. Z uporabo metrik kompleksnosti (upoštevanje hierarhije dedovanja, števila vpletenih razredov in asociacij med razredi) smo ugotovili, da že večina temeljnih vzorcev v principu ni preprosta. Na srečo pa sama implementacija in uporaba agregacije ter vmesnika skrije kompleksnost apliciranega vzorca.

Zato je pomembno, da tudi vzorce, podobno kot objekte in komponente, obravnavamo v luči treh, uvodoma opredeljenih abstrakcij (tabela III). Klasifikacija omogoča, da lahko določen vzorec umestimo v ustrezno kategorijo, določimo njegov namen in učinke in tako laže sprejmemo odločitev v postopku izbire ustreznega vzorca za podani problem. Uporaba agregacije podpira idejo o sestavljenih vzorcih ter delno tudi implicitno uporabo, saj nas pri tej zanima zgolj specifikacijski nivo, ne pa tudi implementacijski. Generalizacija in specializacija je pri vzorcih potrebna zaradi različnih možnosti implementacijskega nivoja vzorcev.

Na osnovi pridobljenih spoznanj in izhodišč, ki jih podaja pričujoči prispevek, ugotavljamo, da je vzorce zagotovo treba obravnavati vsaj z dveh vidikov, ki sta pogojena predvsem z vlogami razvijalcev, ki se z vzorci srečujejo in jih uporabljajo. Apliciranje vzorcev pri razvoju informacijskih rešitev pogojuje kompleksnejša znanja in razumevanje, medtem ko uporaba že integriranih vzorcev zahteva nekoliko drugačna, občasno tudi delno poenostavljena znanja. Menimo, da bomo lahko na osnovi že definiranega formalnega zapisa interne zgradbe in relacij med vzorci zasnovali in izpeljali tudi t. i. zunanji pogled na vzorce, ki ga je potrebno formalizirati, če želimo, da bodo razvojna orodja, razen eksplicitne uporabe temeljnih vzorcev, podprla tudi sestavljanje vzorcev in razvijalcem olajšala iz tega izhajajočo implicitno uporabo vzorcev. Slednja sicer dopušča določen nivo abstrakcije, predvsem zaradi komponentnega pristopa k integraciji vzorcev, na drugi strani pa kljub vsemu prinaša med razvijalce precejšnjo kompleksnost in zahtevnost razumevanja, ki jo lahko olajša le primerna oblika ograjevanja in višji nivo njene avtomatizacije.

V prispevku smo prikazali, da predstavlja uporaba vzorcev enega od osnovnih načinov uspešnega razvoja sodobnih informacijskih rešitev. Pravzaprav edini način. Spoznali smo, da so že v osnovni Javi in ogrodju .NET uporabljeni različni tipi vzorcev, npr. ustvarjalni vzorci (npr. *Edinec, Prototip, Abstraktna tovarna*), strukturni vzorci (npr. *Dekorator*) kot tudi vzorci obnašanja (npr. *Iterator, Opazovalec*). Uporaba teh, že vgrajenih vzorcev, predstavlja neke vrste implicitno uporabo vzorcev, saj razvijalcu ni potrebno poznati vseh podrobnosti uporabljenega vzorca, temveč le osnove apliciranja. Izkaže pa se, da je uporaba in podrobno poznavanje vzorcev pri zasnovi kompleksnejših, komponentno zasnovanih rešitev nujna. Zaradi delitve vlog pri komponentnem razvoju, še posebej v sklopu strežniških tehnologij, kot sta J2EE (Java 2 Enterprise Edition) in .NET, pa lahko ugotovimo, da je delitev na eksplicitno in implicitno

Tabela III: **Uporaba treh osnovnih abstrakcij pri sodobnem razvoju**

	objektni pristop	komponentni razvoj	uporaba vzorcev
Klasifikacija	objektov v razrede	nabor storitev v vmesnikih	klasifikacije vzorcev po področjih, aktivnost
Agregacija	sestavljeni (kompleksni) objekti kot agregati in kompoziti	sestavljene komponente (nivo granularnosti)	sestavljanje vzorcev
Generalizacija	dedovanje med razredi	vmesnikov komponent	družine implementacij vzorcev

uporabo vzorcev smiselna in koristna. Temu je potrebno prilagoditi tudi program izobraževanja posameznih razvijalcev in pri tem najprej opredeliti stopnjo uporabe vzorcev, ki je lahko:

- implicitna brez zavedanja o prisotnosti vzorcev,
- implicitna z zavedanjem prisotnosti vzorcev in potrebo po njihovem poznavanju ter
- eksplicitna s poznavanjem strukture vzorca.

Literatura

1. Jurič M. B., "Nova generacija komponentenih modelov CORBA 3, COM+, EJB", zbornik pete konference *OTS'2000 Objektna tehnologija v Sloveniji*, str. 18–29.
2. Johnson R. , Foote B., "Designing Reusable Classes", *Journal of Object-Oriented Programming*, let. 1, št. 2, 1988, str. 22–35.
3. Krajnc A., Štok B., "Uporaba ogrodij v objektno-orientiranih aplikacijah", zbornik sedme konference *OTS'2001 Objektna tehnologija v Sloveniji*, str. 63–75.
4. Alexander C. et al., *A Pattern Language*, New York, Oxford University Press, 1977.
5. Gamma E., Helm R., Johnson R., Vlissides R., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
6. Domajnko T., Rozman I., Heričko M., "Analiza vzorcev načrtovanja", *Elektrotehniški vestnik*, let. 67, št. 5, str. 261–267.
7. Domajnko T., *Formalizacija opisa vzorcev*, FERI Maribor, doktorska disertacija, 2001.
8. Heričko et al., *Analiza razumevanja, kompleksnosti in uporabnosti vzorcev načrtovanja GoF*, FERI Maribor, Inštitut za informatiko, poročilo, februar 2003, 25 strani.
9. Domajnko T., Heričko M., "Vzorci in Java", zbornik četrte konference *OTS'99 Objektna tehnologija v Sloveniji*, FERI Maribor, str. 185–198.
10. Grand M., *Patterns in Java – volume 1*, Wiley, 1998.
11. Cooper J.W., *C# Design Patterns – A Tutorial*, Addison-Wesley, 2002.
12. J. Eckstein, "Architekturen für das E-business", *JavaSpektrum*, let. 8, št.1, januar 2003, str. 37–39.
13. Zhao Y., "Developing Pattern Implementation Knowledge for Reinforcing Software Design Patterns", Proceedings of the 21st IASTED *Int. Conf. Applied Informatics*, str. 967–972.
14. Jurič M. B. et al., *J2EE Design Patterns Applied*, Wrox Press Inc., 2002.
15. Rozman I. et al., *Integracijska arhitektura: strategije, postopki in metode integracije*, FERI Maribor, Inštitut za informatiko, tehnično poročilo, julij 2002, 133 strani.
16. Beloglavec S., Rozman I., "Vloge pri razvoju strežniških komponent", zbornik konference *IS'2002 Sodelovanje in informacijska družba*, str. 53–56.

Marjan Heričko je docent na Inštitutu za informatiko na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Njegovo raziskovalno-razvojno delo obsega vse vidike objektne tehnologije in komponentnega razvoja, s poudarkom na metodologijah razvoja, metrikah in razvojnih okoljih. Svoje izkušnje je predstavil v številnih prispevkih na domačih in tujih konferencah ter revijah. Je tehnični koordinator aktivnosti Centra za objektno tehnologijo in predsednik konference OTS Objektna tehnologija v Sloveniji. Diplomiral, magistriral in doktoriral je na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru.

Simon Beloglavec je asistent na Inštitutu za informatiko na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru in aktivni član Centra za objektno tehnologijo, v okviru katerega je sodeloval pri pripravi in izvedbi številnih seminarskih delavnic iz področja jave, objektnega načrtovanja in vzorcev. Njegovo raziskovalno delo zajema java tehnologijo, komponentni razvoj, aplikacijske strežnike in vzorce. Diplomiral je na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru, kjer pripravlja tudi svojo doktorsko disertacijo.

Matjaž B. Jurič je docent na Inštitutu za informatiko na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Njegovo raziskovalno-razvojno delo obsega porazdeljne objekte, komponentne modele, vzorce, metode integracije, spletne storitve in tehnologije medorganizacijskega povezovanja. Je avtor več knjig, izdanih pri založbi Wrox Press. Diplomiral, magistriral in doktoriral je na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru.

Boštjan Kežmah je asistent na Inštitutu za informatiko na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Njegovo raziskovalno-razvojno delo obsega celovite informacijske rešitve, sodobna razvojna okolja, metode integracije in spletne storitve. Diplomiral je na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru, magistriral pa na Ekonomsko poslovni fakulteti v Mariboru.