Integracijski okvir primjene uzoraka u razvoju programskog proizvoda temeljenom na modelima

Picek, Ruben

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE VARAŽDIN

Mr. sc. RUBEN PICEK

INTEGRACIJSKI OKVIR PRIMJENE UZORAKA U RAZVOJU PROGRAMSKOG PROIZVODA TEMELJENOM NA MODELIMA

– DOKTORSKA DISERTACIJA –

VARAŽDIN, 2008.

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE VARAŽDIN

Voditelj rada: Dr. sc. Vjeran Strahonja, redoviti profesor

PREDGOVOR

Svijest o važnosti programskih proizvoda u današnjem poslovnom okruženju sve se više razvija. Nastajanjem novih tipova programskih proizvoda raste i potreba za otkrivanjem suvremenijih, još kvalitetnijih metodoloških načina njihova razvoja. Istraživanja u softverskoj industriji u zadnjih desetak godina krenula su u različitim smjerovima. Ovaj rad temelji se na trenutno najaktualnijoj i najskeptičnijoj paradigmi razvoja programskog proizvoda poznatoj pod nazivom *razvoj temeljen na modelima*. Želja je istražiti mogućnosti primjene uzoraka u toj nadolazećoj paradigmi. Kako je u metodološkom pristupu razvoju programskog proizvoda nužno koristiti *proces razvoja*, namjera je utvrditi jesu li postojeći procesa razvoja primjenjivi za tu paradigmu. Nadalje, želi li se koristiti uzroke u kontekstu te paradigme, kako bi se omogućio automatizirani razvoj programskog proizvoda – što naglašava ta paradigma, potrebno je najprije definirati *okvir* njihove primjene te ga tada integrirati u postojeće ili buduće metodike razvoja prikladne za MDD.

Doktorska disertacije sistematizirana je u 7 poglavlja.

U uvodnom poglavlju opisani su motivi izbora teme, osnovni pojmovi, predmet istraživanja, ciljevi i hipoteze rada.

Drugo poglavlje prikazuje stanje u softverskoj industriji i analizira trenutne trendove u razvoju programskih proizvoda. Poglavlje sistematizira probleme današnjeg razvoja, inovacije koje djelomično ili u potpunosti rješavaju neke od problema ili su prisutne samo kao ideje, te će ih tek biti potrebno realizirati u budućnosti. U poglavlju se obrazlaže i pojam *krize* u programskom inženjerstvu kao i pojam *prijelaza* paradigmi koji se trenutno zbiva. S obzirom na značaj koji se u posljednje vrijeme pridaje modeliranju tijekom razvoja programskog proizvoda, razmatraju se osnovni koncepti te ističe važnost modeliranja u tom kontekstu.

Treće poglavlje opisuje paradigmu koja predstavlja jedan od dominantnih smjerova današnjeg razvoja, a poznata je pod nazivom *razvoj temeljen na modelima* (eng. Model Driven Development). U opisu paradigme navode se prednosti koje ona donosi te se analiziraju trenutni problemi njezine realizacije. Kako trenutno postoji nekoliko pogleda na tu paradigmu, poglavlje detaljno opisuje pojedine pristupe realizaciji te paradigme.

Četvrto poglavlje govori o uzorcima. U razvoju programskih proizvoda oni objedinjavaju kako opis problema tako i njihova rješenja, a temelje se na najboljoj praksi u nekom kontekstu. Nakon definiranja osnovnih pojmova i koncepata prikazuje se osnovna

klasifikacija uzoraka, opisuje organizacija uzoraka te se sistematiziraju trenutno pronađeni katalozi uzorka. Detaljno se razmatra struktura uzoraka (specifikacija i implementacija) kao i mogućnost njihove standardizacije.

Na samom početku *petog poglavlja* definiraju se uvjeti primjene softverskih uzoraka u razvoju temeljenom na modelima. Poglavlje detaljno prikazuje osmišljen metodološki okvir primjene uzoraka u kontekstu MDD paradigme. Razvijeni okvir obuhvaća cijeli životni ciklus uzorka te sadrži smjernice, upute i korake za njihovu primjenu u razvoju temeljenom na modelima.

U šestom se poglavlju govori o suvremenim metodikama razvoja kao i utjecaju MDD paradigme na njih. Kako u ovom trenutku niti jedna dostupna suvremena metodika razvoja programskog proizvoda nije usklađena za razvoj temeljen na modelima, a kamoli da objedinjava razvoj temeljen na modelima primjenom uzoraka, poglavlje prikazuje integraciju razvijenog metodološkog okvira primjene uzoraka u razvoju temeljenom na modelima s danas najistaknutijim predstavnikom definiranih skupina metodika.

U *zaključku* disertacije prikazuju se postignuti ciljevi rada, obrazlažu se postavljene hipoteze, iznosi znanstveni doprinos, korisnost rada kao i mogućnosti daljnjeg istraživanja.

Na kraju želim izraziti zahvalnost prije svega svom mentoru prof. dr. sc. Vjeranu Strahonji koji me je uvijek svojim idejama, prijedlozima i susretljivošću ohrabrivao i pružao potrebnu podršku. Ujedno zahvaljujem svim prijateljima i kolegama s Fakulteta organizacije i informatike koji su mi svojim sugestijama i poticajima pomogli u nastanku ovog rada.

Veliko HVALA mojim roditeljima.

U Varaždinu, siječanj 2008. godine

Autor

SADRŽAJ

POPIS OZNAKA I KRATICA

Agile MDD – Agile Model Driven Development

AOP – Aspect Oriented Programming

API – Application Programming Interface

BPD Business Process Diagram

BPEL4WS – Business Process Modeling Language For Web services

BPMI – Business Process Management Initiative

BPML – Business Process Modeling Language

BPMN – Business Process Modeling Notation

BPQL – Business Process Query Language

CBD – Component Based Development

CIM – Computation Independent Model

CMP – Container-Managed Persistence entity beans

CORBA – Common Object Request Broker Architecture

CRUD – Create, Read, Update, Delete

CWM – Common Warehouse Model

DAO – Data Access Object

DPTK – Design Pattern Toolkit

DSDM – Dynamic Systems Development Method

DSL's – Domain Specific Languages

DTD's – Data Type Definitions

EAA – Enterprise Application Architecture

EAR – Enterprise Archive

EJB – Enterprise JavaBeans

EofSca – Economics of Scale

EofSco – Economics of Scope

EP – Enterprise Patterns

EUP – Enterprise Unified Process

GoF – Gang of Four

GUI – Graphical User Interface

IDE – Integrated Development Environment

JET 2 – Java Emitter Templates

JMS – Java Message Service

M2C – Model to Code

M2T – Model to Text Transformation Language

MD(S)E – Model Driven (Software) Engineering

MDA – Model Driven Architecture

MDB – Message-Driven Bean

MDD – Model Driven Development

MOF –Meta Object Facility

MVC – Model-Viev-Controler

OMG – Object Management Group

PDA – Personal Digital Assistant

PIM – Platform Independent Model

POJO – Plain Old Java Object

POSA – Pattern-Oriented Software Architecture

PSM – Platform Specific Model

QVT – Query, View, Transformation Specification

RAS – Reusable Asset Specification

RSA - IBM Rational Software Architect

RTE – Round Trip Engineering

RUP – Rational Unified Process

RUP SE – Rational Unified Process Systems Engineering

SF – Software Factories

SOA – Service Oriented Architecture

SPL – Software Product Line

SQL – Structured Query Language

SW – Software

TT – Transformation Tools

UML – Unified Modeling Language

VSTS –Visual Studio Team System

WS – Web Service

WSDL – Web Services Description Language

XMI – XML Metadata Interchange

XML – Extensible Markup Language

XP – Extreme Programming

POPIS SLIKA

Broj	Naziv slike	Stranica

POPIS TABLICA

Broj	Naziv tablice	Stranica

1. UVOD

Tijekom 1990-ih godina u razvoju programskih proizvoda dominirale su dvije paradigme. U ranim 90-im bila je to paradigma koja se temeljila na CASE alatima i jezicima četvrte generacije, dok je u drugoj polovici desetljeća bila aktualna paradigma koja se temeljila na objektnoj orijentaciji. I dok softverska industrija nije bila zrela za CASE pristup, objektno orijentirana paradigma se održala, iako u potpunosti nije ispunila sva najavljivana očekivanja. No, postala je osnova za komponentni razvoj, objektno orijentirane jezike, notaciju za modeliranje – UML kao i filozofiju reverzibilnog inženjerstva. Današnja situacija u softverskoj industriji ukazuje da se scenarij ponavlja.

Iako primjena objektno orijentiranog razvoja rješava mnoge probleme koji se pojavljuju u softverskoj industriji, industrija se trenutno suočava s mnogobrojnim novim izazovima. Povećavaju se zahtjevi prema funkcionalnosti sustava, eksponencijalno raste njihova kompleksnost, a samim time njihov razvoj postaje složeniji, rizičniji i skuplji, posebno u uvjetima skraćivanja vremena razvoja. Kako bi se doskočilo rastućim izazovima, pojavio se široki spektar novih pristupa u razvoju programskih proizvoda. Najznačajniji su: razvoj temeljen na komponentama (eng. CBD), servisno orijentirana arhitektura (eng. SOA), arhitektura temeljena na modelima (eng. MDA), agilni razvoj te linije za proizvodnju programskih proizvoda (eng. SPL). Zapravo, trenutno se stanje u softverskoj industriji može nazvati prijelaznim razdobljem tj. dešava se prijelaz paradigmi (eng. Paradigm Shift). Kako je '90-tih bio prijelaz iz strukturnog pristupa u objektni, tako se sada suvremeni razvoj programskih proizvoda usmjeruje prema novoj paradigmi - razvoju temeljenom na modelima (eng. MDD). Trenutno je MDD paradigma visoko pozicionirana u softverskoj industriji. Ideja ove paradigme je iskoristi modele, ne samo za dokumentiranje programskog proizvoda nego i za transformaciju u programski kod, kako bi se omogućila automatizacija razvoja. Neki od gore navedenih pristupa predstavljaju realizaciju te paradigme. Kombinacija tehnika koje paradigma obuhvaća (meta modeliranje, analiza domene, primjena jezika za modeliranje, generiranje na temelju modela i dr.) pokazuje koliko je industrija sazrjela usprkos problema i novih izazova koji se pojavljuju. Iako je u akademskim krugovima pojam razvoja temeljenog na modelima već dobro etabliran, u softverskoj industriji konkretna realizacija MDD paradigme nailazi na probleme te je nemoguće predvidjeti njezinu evoluciju u budućnosti. Jedan od značajnih problema je automatizacija prevođenja modela, problem koji još u

potpunosti nije riješen. Mnoge organizacije koje se bave razvojem programskih proizvoda trebat će se suočit s izazovom razvoja softvera primjenom MDD paradigme kako bi ocijenile hoće li krenuti tim putem ili ne. Trenutno su organizacije u početnim fazama usvajanja MDD paradigme s velikom dozom skeptičnosti i konzervativnosti. Uz to u softverskoj industriji prisutni su i mitovi vezani uz MDD paradigmu. Neki od njih imaju dužu povijest dok se korijeni nekih mogu pratiti iz prošlih paradigmi, a neki zapravo i nisu mitovi već realnost koja se preslikava u MDD paradigmu.

Najčešće spominjani su [Bettin, 2004., str. 22-25.]:

- da se pod pojmom model uvijek misli na UML model,
- da modeliranje uvijek podrazumijeva cjelokupnu specifikaciju u obliku vizualnih modela, na istoj razini apstrakcije,
- da je programski kod generiran iz modela ružan i neprikladan razumijevanju programera,
- da je generirani kod nepouzdan i nižih performansi nego kod napisan od programera,
- da je MDD paradigma nespojiva s konceptom agilnosti,
- da je MDD paradigma isto što i CASE koji se pokazao neuspješnim,
- da ponovno pokretanje generiranja koda iz modela nije praktično te da će uvijek doći do neusklađenosti u nekoj točci razvoja,
- da definiranje transformacije traje duže nego ručno pisanje kompletnog koda,
- da je prilagođavanje generatora novonastalim promjenama mukotrpnije nego ručna promjena generiranog koda,
- da programski kod koji producira generator nije objektno orijentiran.

Već se godinama u softverskoj industriji uspješno primjenjuju *uzorci*, kao rješenja za klase problema koji se često ponavljaju u nekom kontekstu. Iako se danas prilikom primjene pojma *uzorak* prvenstveno misli na u*zorke oblikovanja*, u nekoliko posljednjih godina uzorci su po svojoj definiciji i broju evoluirali. Njihova primjena proširila se na sve faze razvoja programskih proizvoda.

Nadalje, analiziraju li se fazni i agilni procesi razvoja, koji se danas primjenjuju u industriji, može se primijetiti da njihov odnos prema MDD nije definiran.

Industrija također nameće i promatranje razvoja programskih proizvoda s ekonomskog aspekta te se sve više svi tipovi artefakata u razvoju promatraju kao ponovno iskoristivi resursi.

Sve rečeno ukazuje na mnogobrojna pitanja koja ostaju otvorena te ih je vrijedno istražiti.

Ovim radom želi se istražiti mogućnosti primjene uzoraka u razvoju temeljenom na modelima utvrđujući njihov doprinos u napretku realizacije MDD paradigme.

Pitanja koja se u ovom kontekstu nameće su:

1. *Što je uzorak koji je primjenjiv u MDD paradigmi?*
2. *Kako bi izgledala primjena uzoraka u okviru razvoja temeljenog na modelima?*
3. *Može li se ta primjena metodološki obuhvatiti i integrirati u postojeće procese razvoja programskog proizvoda?*

Istraživanje odgovora na ova pitanja čini predmet istraživanja ove doktorske disertacije.

Postavljeni ciljevi doktorske disertacije su:

- Prikazati današnje stanje u softverskoj industriji, s naglaskom na *probleme* prisutne u objektno orijentiranoj paradigmi i *prijelazno stanje* u kojem se nalazi softverska industrija koje ukazuje na novu paradigmu razvoja.
- Provesti istraživanje postojećih MDD pristupa razvoju programskog proizvoda i pripadajućih procesa razvoja.
- Napraviti osnovnu klasifikaciju uzoraka prema utvrđenim kriterijima.
- Definirati uvijete primjene uzoraka za razvoj temeljen na modelima.
- Razviti vlastiti okvir primjene uzoraka u razvoju programskog proizvoda temeljenom na modelima.
- Integrirati definirani metodološki okvir u postojeće fazne i agilne metodike razvoja.

Na temelju opisanog problema i ciljeva rada postavljene su dvije **hipoteze** koje se izradom doktorske disertacije žele dokazati. To su:

H1:

Moguće je definirati metodološki okvir primjene uzoraka u razvoju programskog proizvoda temeljenom na modelima.

H2:

Objedinjavanje predloženog metodološkog okvira primjene uzoraka s postojećim metodikama razvoja programskih proizvoda olakšava realizaciju MDD paradigme.

2. RAZVOJ PROGRAMSKOG PROIZVODA

Poglavlje prikazuje stanje u softverskoj industriji, analizira trenutne trendove razvoja programskih proizvoda te ističe važnost modeliranja i modela u kontekstu današnjeg razvoja programskih proizvoda.

2.1. Analiza današnjeg stanja u softverskoj industriji

Ekspanzija primjene programskih proizvoda u različitim segmentima sve je veća. Softver postaje središnjih faktor ljudskog društva, a izravna posljedica toga je njegova veličina i sve veća složenost. Unatoč ekonomskoj i funkcionalnoj važnosti programskih proizvoda u suvremenom okruženju, njihov razvoj i održavanje još se uvijek smatraju aktivnostima visokog rizika. Izvještaji Standish Group [The Standish Group, 1995.], [The Standish Group, 2004.] za period od 1994. do 2006.* rađeni za IT sektor u SAD-u i članci [InfoQ, 2006.], [Rubinstein, 2007.] idu u prilog tim tvrdnjama.

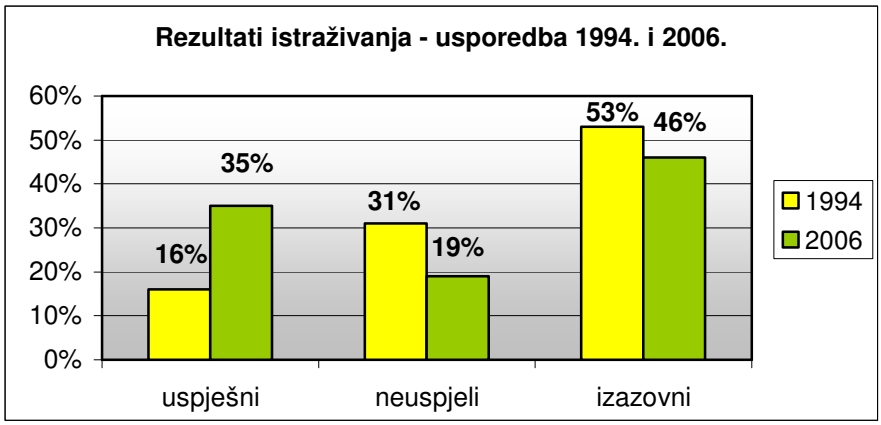
Tablica 2-1: Rezultati istraživanja Standish Group

STANDISH GROUP REPORT *CHAOS*	**1994**	**1996**	**1998**	**2000**	**2002**	**2004**	**2006***
uspješni	16%	27%	26%	28%	34%	29%	35%
neuspješni	31%	40%	28%	23%	15%	18%	19%
izazovni	53%	33%	46%	49%	51%	53%	46%
prosječan % prekoračenja budžeta	180%	142%	69%	45%	43%	56%	-
prosječan % prekoračenja trajanja	164%	131%	79%	63%	82%	84%	-

Izvor: [The Standish Group, 1995.], [The Standish Group, 2004.], [InfoQ, 2006.], [Rubinstein, 2007.]

Kratkom analizom podataka iz tablice 2-1., može se zamijetiti da u periodu od 1994. do 2002. postoji evidentan napredak u produktivnosti razvoja programskih proizvoda. No, usporede li se brojke za 2004. i 2006. primjećuju se marginalan napredak.

* Djelomični izvještaj izašao je u ožujku 2007. u časopisu SD Times. <http://www.sdtimes.com/article/story-20070301-01.html> (pristupano 26.03.2007.)



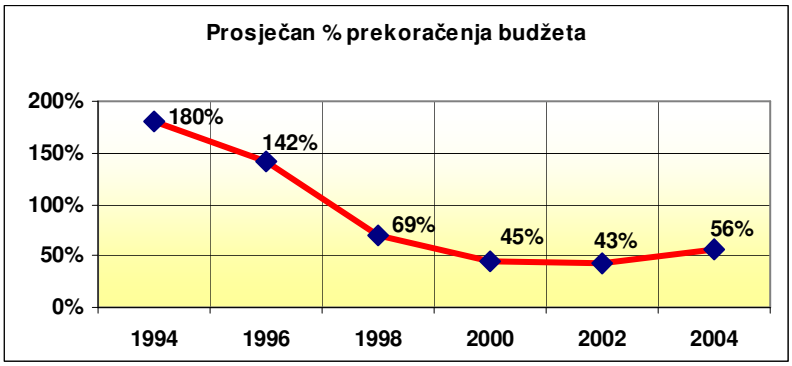
Slika 2-1. Usporedba rezultata istraživanja IT projekta u SAD-u za 1994. i 2006.

Kada je **1994.** Standish Group po prvi puta provela istraživanje za SAD, rezultati istraživanja [The Standish Group, 1995.] prikazivali su da je na razvoj softvera (cca 175.000 projekata), godišnje utrošeno $250 milijardi dolara (od čega su $140 milijardi dolara gubici) s prosječnom cijenom projekta u rasponu od $430.000 – $2.300.000 ovisno o veličini poduzeća. Kao što je vidljivo na *slici 2-1*. samo 16% tih projekata realizirano je na vrijeme i u okviru predviđenog budžeta, dok je 31% bilo neuspješnih. U 53% projekata (uključujući i neuspješne) troškovi su bili veći od planiranih i to u prosjeku za 180%. Projekti koji su bili dovršeni imali su samo 42% originalno zamišljene funkcionalnost.

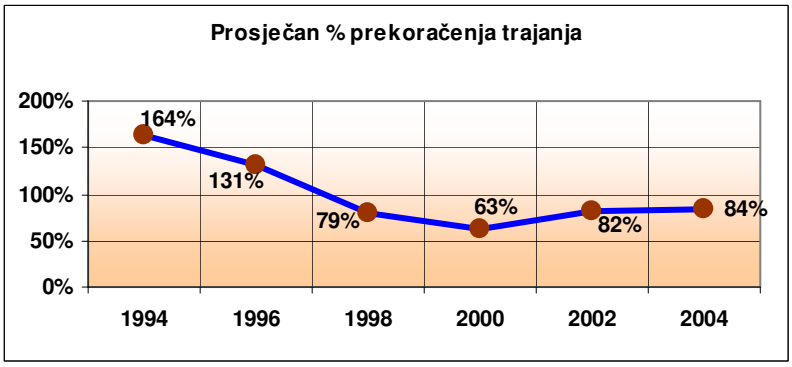
Tijekom perioda od **1994. do 2004.** analizirano je više od 40 000 projekata [InfoQ, 2006.], [The Standish Group, 2004.], a rezultati za 2004. znatno su povoljniji. Postotak uspješno realiziranih softverskih projekata bio je veći za više od 100% i iznosio je 34%. Postotak neuspješnih projekata znatno se smanjio te je iznosio svega 15% svih projekata. No postotak projekata koji su premašili vrijeme i budžet te bili smanjene funkcionalnosti iznosio je 51% (smanjio se svega za 2%). Za projekte koji su premašili planirane troškove (uključujući i neuspješne) u prosjeku je iznos bio veći za 43%. Iznos koji je utrošen, iznosio je $255 milijardi dolara od čega su $55 milijardi dolara gubici.

Iako izvještaj za **2006**., u vrijeme pisanja novog poglavlja, nije još u potpunosti dovršen, poznati su neki podaci [Rubinstein, 2007.]. Tako je broj uspješnih projekata u odnosu na 2004. porastao za 1%, i iznosio 35%. Postotak neuspješnih projekata bio je 19%. No postotak projekata koji su prekoračili vrijeme i budžet te su bili smanjene funkcionalnosti iznosio je 46%. Komentar izvještaja za 2006. [Rubinstein, 2007.] navodi, da za svaki uloženi dolar, 41 centi predstavlja gubitak, dok je 1998. gubitak za svaki dolar iznosio 76 centi.

Iako ohrabruje činjenica da raste postotak uspješnih projekata, postotak projekata koji prekoračuju vrijeme (*slika 2-2.*), budžet (*slika 2-3.*) ili su smanjene funkcionalnosti još uvijek je velik, te zabrinjava stagnacija u tom segmentu.



Slika 2-2. Prosječan postotak prekoračenja iznosa budžeta



Slika 2-3. Prosječan postotak prekoračenja vremena trajanja projekta

Na temelju iznesenih statističkih podataka može se zaključiti da je današnji razvoj spor i skup, a da softver sadrži mnogobrojne mane u upotrebljivosti, pouzdanosti, sigurnosti i performansama te njihovo otklanjanje uzrokuje prekoračenja.

Sve to ukazuje na činjenicu da je u softverskoj industriji još uvijek prisutna **softverska kriza** (eng. software crisis)[*], pojam koji je poznat koliko i samo programsko inženjerstvo.

2.1.1. Pojam softverske krize

Pojam *softverska kriza* korišten je u ranim danima programskog inženjerstva kako bi se

[*] Na inicijativu NATO-a, 1968. godine u Garmischu održana je konferencija na kojoj su u softversku industriju uvedeni pojmovi *programsko inženjerstvo* i *softverska kriza*.
<http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html>

opisao utjecaj rapidnog povećanja računalne snage i kompleksnosti problema koje je trebalo riješiti, a odnosi se na težinu pisanja ispravnih i razumljivih računalnih programa.

Crnković u [Crnković, 2007.] naziva *softversku krizu* stanjem u kojem se problemi pojavljuju brže nego što se mogu riješiti.

Prema [Strahonja *et al.*, 1992. str. 1.], *softverska kriza* očituje se u nemogućnosti pojedinca i organizacijskih sustava da:

- na odgovarajući način zadovoljavaju svoje potrebe za informacijama i znanjima,
- upravljanju informacijama i
- računalno podrže i automatiziraju sve poslovne aktivnosti.

Uzroci softverske krize povezani su s cjelovitom složenosti procesa razvoja i relativnoj nezrelosti programskog inženjerstva ako discipline. U *korijene softverske krize* ubraja se: kompleksnost, nerealna očekivanja i teško upravljanje promjenama [Greenfield *et al.*, 2004., str. 7.].

Može se reći da se kriza manifestira u:

- količini softvera,
- prekoračenju iznosa budžeta projekta,
- prekoračenju vremena razvoja,
- niskoj kvaliteti softvera,
- neispunjenju definiranih korisničkih zahtjeva,
- neupravljanju projektom razvoja i
- teškom održavanju koda.

Još neki od oblika softverske krize navedeni u [Crnković, 2007.] su:

- velik broj pogrešaka u softveru,
- visoka cijena koštanja softvera i
- teško razumijevanje softvera.

Zaključimo, u simptome koji uzrokuju softversku krizu mogu se ubrojiti:

- količina softvera,
- površno razumijevanje korisničkih zahtjeva,
- nejasna komunikacija,
- ne otkrivanje nekonzistentnosti u zahtjevima, oblikovanju i implementaciji,
- nemogućnosti reagiranja na promjene,
- nemogućnost spajanja modula,

- kompleksno održavanje ili nadogradnja rješenja,
- kasno otkrivanje ozbiljnih pogrešaka u projektu,
- niska kvaliteta softvera,
- neprihvatljive performanse,
- nedovoljan timski rad i
- nepouzdan proces razvoja.

2.1.2. Današnji oblici softverske krize – izazovi

Gore navedeni oblici softverske krize danas su lako prepoznatljivi u objektno orijentiranoj paradigmi. Najznačajniji izazovi u razvoju programskih proizvoda s kojima se IT stručnjaci, projektanti i programeri svakodnevno suočavaju su [Sommerwille, 2007., str. 13.]; [Greenfield *et al.*, 2004., str. 7.]:

- ubrzana evolucija tehnologije i platformi,
- sve veća očekivanja klijenata, koja stvaraju nove vrste aplikacija,
- naizgledna potreba za beskonačnim usvajanjem novih vještina i znanja te
- pritisak da se iz zastarjelih aplikacija izvedu nove mogućnosti.

Rezultati istraživanja navedeni u uvodu ovog poglavlja pokazuju da savladavanje uzroka softverske krize i današnjih izazova i nije tako uspješno kako se misli. Kako se današnji objektno orijentirani razvoj nosi s dva najveća korijena softverske krize – *složenosti sustava* i *upravljanja promjenama*?

Složenost sustava nije jednostavno precizno definirati, te je se može zamisliti kao mjeru problematičnosti rješavanja nekog problema [Greenfield *et al.*, 2004., str. 35]. Cilj razvoja programskog proizvoda je implementacija korisničkih zahtjeva. No, zbog razlike u razinama apstrakcije neminovno se pojavljuje jaz između zahtjeva korisnika i programskog koda koji se naziva *apstrakcijski jaz*. Kako bi se on što je moguće više smanjio, nastoji se povećati razina apstrakcije. U objektno orijentiranoj paradigma glavna tehnika kojom se rješava složenost sustava je *učahurivanje*. Učahurivanjem se povećava razina apstrakcije "pakiranjem" jednog ili više elemenata u konceptima kao što su klasa ili komponenta. Slijedeća tehnika koja se primjenjuje je *dokumentiranje dizajna* sustava vizualnim modeliranjem ili vizualizacijom koda. Ujedno, u svrhu što

kvalitetnijeg savladavanja složenosti definirani su procesi razvoja s detaljno opisanim koracima i smjernicama poznati kao formalne metodike.

Upravljanje promjenama - **(ne)mogućnost reagiranja na promjene**. Promjene koje se u ovom kontekstu pojavljuju mogu se klasificirati na:

- promjene u problemskoj domeni (promjene korisničkih zahtjeva) i
- promjene u domeni rješenja (evolucija tehnologije).

Za očekivati je da bi se programski proizvod trebao moći mijenjati kao odgovor na promjene u odnosu na zahtjeve i okruženje u kojem se nalazi. Paradoksalno, u praksi su, bilo kakve promjene na softveru skupe i nepoželjne. Ujedno, kako bi se što kvalitetnije upravljalo promjenama koriste se agilne metodike razvoja programskog proizvoda (npr. XP, DSDM, SCRUM) dok je pristup u razvoju projekta iterativan.

Jedan od problema koji se ovdje može klasificirati je i zastarijevanje programskog proizvoda kako industrija evoluira.

2.1.3. Kronični problemi i ograničenja objektno orijentirane paradigme

Neki od kroničnih problema prisutni su od početka objektno orijentirane paradigme dok su se drugi pojavili tek nedavno. Problemi se mogu klasificirati u 3 kategorije [Greenfield *et al.*, 2004., str. 109.]:

1. sastavljanje softvera,
2. prevelika općenitost i
3. nezrelost procesa razvoja.

Kratka analiza problema u svakoj kategoriji prikazana je u nastavku.

2.1.3.1. Sastavljanje softvera

Jedna od vizija pionira objektno orijentirane paradigme bila je i da se programski proizvod sastavlja od već postojećih objekata. No, nažalost, do danas nije naučeno kako sastavljati softver na temelju ponovno iskoristivih komponenata u komercijalno signifikantnom obimu. To je vrlo razočaravajuća činjenica, jer se smatralo kako će upravo objektno orijentirana paradigma riješiti taj problem. Prirodna posljedica razvoja objektno orijentirane paradigme je razvoj temeljen na sastavljanju (eng. Development by Asembly). Sada je naravno poznata

činjenica da su objekti previše zrnati elementi, koji su ujedno ovisni o kontekstu, da bi ih se primjenjivalo kao elemente za sastavljanje. Pojavom razvoja temeljenog na komponentama (eng. Component-Based Development – CBD) smatralo se da će se riješiti ovaj problem no, usprkos obećanjima CBD, softver se i dalje razvija od početka. Razvojni inženjeri* kao glavni uzrok spominju premali broj ponovno iskoristivih komponenata ili teško pronalaženje odgovarajućih komponenta koji će rješavati konkretan problem.

Istraživanja [Greenfield *et al.*, 2004., str. 111-115.] pokazuju kako su ovi problemi simptomi fundamentalnijih problema kao što su:

- **Postojanje protokola koji su specifični o platformi.** Današnje komponente previše su ovisne o platformi i tehnologiji implementacije što otežava njihovo spajanje čak i kada je u pitanju samo nova verzija.
- **Nedovoljno pakiranje.** Trenutne specifikacije komponenata i tehnologije pakiranja ne sadržavaju dovoljno informacija o svojstvima komponenta, njihovoj interakciji što otežava razvoj temeljen na sastavljanju.
- **Preveliko učahurivanje.** Trenutne tehnologije implementacije komponenata koriste mehanizme prevelikog učahurivanja što otežava njihovo prilagođavanje tijekom razvoja.
- **Sindrom "ne postoji za moj problem".** Najpodmukliji oblik sastavljanja softvera prisutan je zbog nepovjerenja prema svemu što nije razvijeno unutar poduzeća.

2.1.3.2. Prevelika općenitost

Trenutne metode i tehnologije razvoja programskih proizvoda pružaju veći stupanj slobode nego što to zahtijeva većina aplikacija.

Problemi koji spadaju u ovu kategoriju su:

Neprecizni jezici za modeliranje. Vizija da se primjenom modela poveća razina apstrakcije i automatizira razvoj prisutna je već dugi niz godina. S ciljem realizacije te vizije razvijen je UML (eng. Unified Modeling Language) - unificirani jezik za modeliranje koji je kasnije i standardiziran i proglašen temeljem današnjeg razvoja. No,

* U radu se koristi pojam *razvojni inženjeri* kako bi se jednom riječju obuhvatili svi sudionici razvoja programskog proizvoda, a to su: projektanti, analitičari, arhitekti, programeri testeri, voditelji projekta i drugi.

usprkos većem broju UML alata koji su u zadnjih desetak godina razvijeni, očekivani napredak u kontekstu povećanja razine apstrakcije kako bi se omogućila automatizacija je izostalo. Tako se za UML kaže da je: presložen, preopćenit, neprecizan u semantici i sl. Martin Fowler, Steve Mellor [Flower, 2004., str. 2.] nezavisno jedan od drugog, navode kako UML treba promatrati kroz 3 razine: za skiciranje sustava, za dokumentiranje i kao programski jezik. I dok se UML uspješno može primjenjivati na prvoj i djelomično na drugoj razini, na trećoj razini još nije dostigao cilj, koji je i zapravo temelj razvoja temeljenog na modelima. Uzrok tog neuspjeha je u njegovoj nepreciznosti, kao i činjenici da je on dizajniran za dokumentiranje sustava, a ne za generiranje koda na temelju modela [Greenfield *et al.*, 2004., str. 118].

Slabe mogućnosti generiranja koda. Primjena CASE alata koji su trebali primjenom modela automatizirati razvoj doživjela je neuspjeh, jer razvoji inženjeri nisu željeli trošiti vrijeme na modeliranje i modele za koje se obećavalo da će moći dati programski kod. Nadalje, njihova primjena obećavala je generiranje koda za različite platforme dok je u praksi kvaliteta tog koda bila slaba. Ujedno mogućnosti reverzibilnog inženjerstva (eng. Round Trip Engineering) nisu pružale dobre rezultate. Danas se pokreće novi val razvoja razvojnih okruženja za koje se tvrdi da imaju mogućnosti generiranja programskog koda na temelju modela, no mnogi su skeptični.

2.1.3.3. Nezrelost procesa razvoja

Jedan od najizazovnijih problema u današnjem razvoju programskih proizvoda je upravljanje procesom razvoja, što naročito dolazi od izražaja u situacijama kada jedan proizvod razvija više timova koji trebaju surađivati tijekom razvoja. Prema [Greenfield *et al.*, 2004., str. 121.] Booch kaže: *"proces razvoja predstavlja skup inženjerskih praksi koji su oblikovane s ciljem ostvarivanja visoko kvalitetne i efikasne alokacije resursa i efikasnog upravljanja troškovima i vremenom"*.

Iako danas gotovo sve organizacije koriste neki proces razvoja, te je na tržištu prisutan veliki broj različitih metoda i metodika, samo neke od njih uspijevaju razviti programski proizvod koji udovoljava zahtjevima i u okviru je predviđenog budžeta i trajanja projekta. To ukazuje na problem da trenutačne metode i metodike nisu dovoljno razvijene.

Većina današnjih procesa razvoja pripada jednom od dva ekstrema: ili su pretjerano formalne (kako bi se ostvarila što bolja suradnja između timova) ili odbacuju formalizam te

dopuštaju pretjeranu autonomiju u cilju postizanja agilnosti.

Drugim riječima, formalni procesi optimizirani su kako bi se nosili sa složenošću sustava, dok su neformalni (agilni) procesi optimizirani kako bi se što je moguće kvalitetnije upravljalo promjenama.

Detaljno definiranje procesa razvoja, kod *formalnih procesa*, osigurava strukturu projekta koja će sudionicima omogućiti organiziranje i razumijevanje zadataka, okruženja i organizacije. Za razliku od njih, *agilni procesi* namijenjeni su manjim projektima u kojima sudjeluju mali timovi, gdje se verbalnom komunikacijom smanjuje potreba specificiranja zahtjeva, a softver se razvija u malim iteracijama uz stalnu interakciju s klijentima.

2.1.4. Inovacije - trendovi razvoja programskog proizvoda

Kako se nositi s gore navedenim problemima i izazovima koje oni donose?

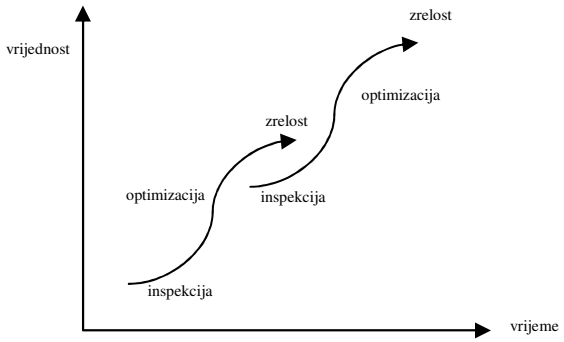
U najznačajnije trendove koji su danas prisutni u softverskoj industriji mogu se nabrojiti:

- Unificirani jezik za modeliranje (eng. UML)
- Domain Specific Language's (eng. DSL's)
- Web servisi (eng. WS)
- Servisno orijentirana arhitektura (eng. SOA)
- Linije za proizvodnju softvera (eng. SPL)
- Aspektno orijentirano programiranje (eng. AOP)
- Razvoj temeljen na modelima (eng. MDD)
    - Arhitektura temeljena na modelima – MDA (eng. Model Driven Architecture)
    - Tvornice softvera – SF (eng. Software Factories)
    - Agilni razvoj temeljen na modelima – Agile MDD (eng. Agile Model Driven Development)

Kako je predmet istraživanja ove disertacije ograničen na razvoj temeljen na modelima (MDD), detaljan prikaz MDD kao nove paradigme razvoja i njezinih realizacija analizira se u 3 poglavlju.

2.1.5. Prijelaz paradigmi

U prethodnoj točci dan je kratak pregled trenutnih inovacija u softverskoj industriji. Prema [Greenfield *et al.*, 2004., str. 32.] moguće je prepoznati uzorak ciklusa inovacije (*krivulje*) koji je prikazan na *slici 2-4*.



Slika 2-4. Krivulja inovacija.

Izvor: [Greenfield *et al.*, 2004., str. 32.]

Kao što se vidi na *slici 2-4*., napredak *krivulje inovacije* je u početku rapidan, ali kako ona postiže stabilnost i zrelost, napredak se smanjuje. Na kraju, krivulja gubi mogućnost zadržavanja napretka te postiže svoj plato. U tom trenutku pojavljuje se nova krivulja koja predstavlja novu generaciju novih tehnologija te se proces ponavlja. Prema [Greenfield *et al*, 2004., str. 32.] svaka krivulja predstavlja jednu paradigmu, a prijelaz s jedne na drugu, naziva se **prijelaz paradigmi** (eng. paradigm shift).

Prema [Greenfield *et al.*, 2004., str. 32.] nova se paradigma pojavljuje kako bi uklonila nedostatke prethodnih i kako bi istražila nove načine razmišljanja. Kako su inovacije (krivulje) diskontinuirane, prijelaz paradigmi je po prirodi razoran te često dolazi u sukob s postojećim društvenim, socijalnim i ekonomskim pogledima [Greenfield *et al.*, 2004., str. 32.].

Preslika li se dosad rečeno u današnje stanje, objektno orijentirani razvoj mnogo se promijenio od svoje inspekcije, no još uvijek ostaje trenutna paradigma. Poboljšao je strukturni pristup iako je u početku bilo mnogih problema. Kao i svaka nova paradigma inicijalno je bilo mnogih problema koji su ispravljeni nakon stečenog iskustva. No nažalost, usprkos velikom napretku, vizija koju su izložili pioniri objektno orijentirane paradigme s naglaskom na industrijski razvoj ostala je nerealizirana [Greenfield *et al.*, 2004., str. 33.]. Kada su riješeni problemi koji su se nalazili u žarištu paradigme, pažnja se posvetila onima na margini, koji su tada postali prevelike zapreke daljnjeg napretka razvoja paradigme. Kako

se navodi u [Greenfield *et al.*, 2004., str. 33.] može se zaključiti da su inovacije u objektno orijentiranoj paradigmi dostigle plato, te da je potrebna nova paradigma koja će predstavljati korak naprijed u razvoju softvera. Prema [Kuhn, 1970.] nova paradigma temeljiti će se na prednostima prethodne i riješiti će slabosti koje proizlaze iz njezinih kroničnih problema. Ta nova paradigma je razvoj temeljen na modelima - MDD.

2.2. Važnost modeliranja i modela

Sve veća dinamičnost poslovnog okruženje doprinijela je pojavi novih tipova programskih proizvoda kao i pojavi novih tehnoloških platformi poput: web servisa, mobilnih uređaja i PDA uređaja. Efektivan način savladavanja *složenosti* i *upravljanja promjenama* u razvoju takvih sustava je *modeliranje* – inženjerska tehnika koja se godinama uspješno primjenjuje u različitim industrijama.

U IT industriji modeliranje se uglavnom primjenjuje u kontekstu:

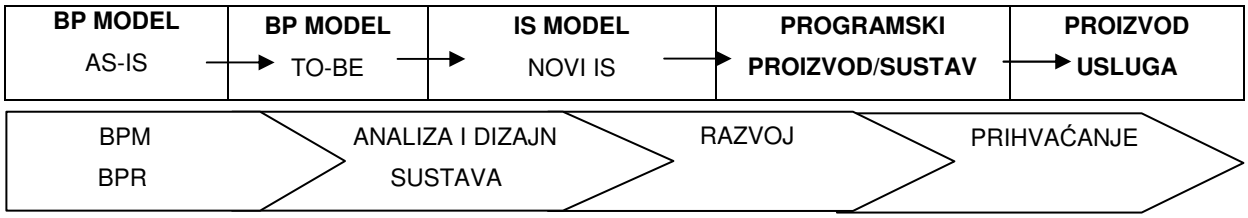
- poslovnih procesa (*eng.* bussiness modeling) i
- razvoja programskog proizvoda (sustava/softvera) (*eng.* system/SW modeling)

Opće prihvaćena razlika je da, *modeliranje poslovnih procesa* opisuje odvijanje poslovnih procesa u interakciji s interesnom skupinom (eng. stakeholers) i događajima, dok *modeliranje programskog proizvoda* obuhvaća razvoj novog softvera (definiranje i analiza zahtjeva, dizajn i implementacija na nekoj odabranoj tehnologiji).

Prema [Strahonja, 2006. str. 296.], cilj *modeliranja poslovnih procesa* je postizanje zajedničkog razumijevanja između interesne skupine (eng. stakeholera) s obzirom na to *tko* daje i razmjenjuje *što* (dobra, usluge) *s kim* i *što* se očekuje za uzvrat. Cilj modela poslovnog sustava je specificirati *kako* i *tko* izvršava procese.

Prema [Strahonja, 2006. str. 296.], cilj *modeliranja programskog proizvoda* je na strukturirani način primijeniti modeliranje na same poslovne procese razvijajući zahtjeve programskog sustava kao i naravno druge modele za aktivnosti oblikovanja i implementacije softvera koje slijede kasnije.

Kada se poslovne procese i razvoj softvera promatra u zajedničkom kontekstu može se generirati lanac vrijednosti koji je prikazan na *slici 2-5*.



Slika 2-5. Lanac vrijednosti poslovni procesi/razvoj softvera

Navedene perspektive modeliranja tradicionalno uključuju potpuno različite svjetove modeliranja te je izuzetno teško realizirati ovakav lanac vrijednosti. Stoga je sve više prisutna ideja unificiranja tih svjetova – ne nužno jednim jezikom za modeliranje ili alatom, već kroz kombinaciju višestrukih otvornih industrijskih standarda [Cernosek, Naiburg, 2004., str. 3.].

2.2.1. Opće karakteristike modeliranja

Prije nego što se istakne vrijednost modeliranja u okviru svake perspektive potrebno je istaknuti opće karakteristike modeliranja. To su:

- pojam modeliranja i modela,
- ciljevi, principi i važnost modeliranja i
- komponente modeliranja.

2.2.1.1. Pojam modeliranja i modela

Modeliranje se može promatrati i u vizualnom, ali i tekstualnom kontekstu.

Modeliranje predstavlja centralni dio svih aktivnosti koje vode do razvoja dobrog programskog proizvoda tijekom kojeg se, na osnovi odabrane metodike, oblikuju modeli [Booch].

U uskoj povezanosti s riječju modeliranje je i riječ *model*, koja dolazi od latinske riječi *"modulus"*, a znači: *mjera, pravilo, uzorak, primjer koji treba slijediti* [Ludewig, 2003.]. Od mnogobrojnih definicija izdvojena je i prikazana jedna opća i jedna iz kojom se definira softverski model:

Model predstavlja pojednostavljenje tj. apstrahiranje realnog svijeta, iz određenog kuta gledanja u svrhu što boljeg razumijevanja sustava koji se razvija [Veljović, 2002. str. 5.].

Softverski model predstavlja opis sustava iz određene perspektive zanemarujući pri tome irelevantne detalje kako bi se što jasnije vidjelo karakteristike sustava [Yusuf el al., 2006., str. 1.].

Modeli se mogu odnositi na različite perspektive poslovne domene ili programskog proizvoda koji se razvija i to na različitim razinama apstrakcije. Kako je upotrebljiv model relativno jednostavno proizvesti i jednostavnije proučavati, modeliranjem se smanjuje rizik i troškovi koji su povezani s finalnom implementacijom. Modeli se mogu koristiti i za prikazivanje ideja dizajna, za naglašavanje važnih aspekata oblikovanja, procjenu mogućnosti, otkrivanje pogrešaka i određivanje kompromisa u složenom dizajnu prije njegove realizacije [RSA, 2006.].

2.2.1.2. Ciljevi, principi i važnost modeliranja

U [Picek, 2004., str. 42.] navedeni su slijedeći ciljevi i principi modeliranja:

Ciljevi modeliranja:

- modelom vizualno prikazujemo stvarno ili željeno stanje sustava (*vizualizacija*),
- modelom definiramo strukturu i ponašanje sustava (s*pecifikacija*),
- model predstavlja predložak koji pokazuje kako sustav treba biti konstruiran (*konstrukcija*),
- model predstavlja dokumentaciju projektnih odluka (*dokumentacija*) i
- definiranjem modela upravljamo rizikom.

Principi modeliranja:

- izbor modela koji će se izrađivati ima ključan utjecaj na to kako će se problem rješavati i kako će rješenje biti oblikovano,
- svaki model može prikazivati različite razine detalja,
- najbolji modeli povezani su s realnim svijetom i
- niti jedan model nije dovoljan sam za sebe; svaki netrivijalni sustav najbolje se opisuje malim skupom gotovo nezavisnih modela.

Važnost modeliranja:

Modeliranje nekog realnog sustava prije njegove konstrukcije ili nadogradnje vrlo je značajno. Modele složenih sustava izrađujemo jer je takve sustave vrlo teško obuhvatiti i razumjeti kao cjeline. Apstrahiranjem nebitnih elemenata izrađujemo *model*, koji predstavlja određeni pogled na složeni sustav, smanjujući time složenost izučavanog sustava, ali i rizik koji donosi njegova implementacija. Kombinacijom većeg broja manjih i nezavisnih modela, možemo se učinkovito suprotstaviti složenosti realnih sustava.

Prema [Cernosek, Naiburg, 2004., str. 4.] potrebno je modelirati:

- da bi bolje razumjeli poslovnu domenu,
- kada je rješenje jako kompleksno,
- kada u razvoju sudjeluju više ljudi (potencijalno su i dislocirani) i
- kada je rješenje potrebno održavati i nadograditi.

Ne zahtijevaju svi projekti nužno modeliranje. Prema [Cernosek, Naiburg, 2004., str. 4.] **ne** modeliramo kada:

- je problemska domena dobro poznata,
- je rješenje relativno jednostavno implementirati,
- jako malo ljudi treba sudjelovati u realizaciji rješenja i
- rješenje ne zahtijeva skoro nikakvo održavanje/nadogradnju.

Modeliranje pruža mogućnost vizualizacije cjelokupnih sustava, isprobavanje raznih varijanti i diskusiju oko idejnih rješenja puno jednostavnije i jednoznačnije prije ulaženja u rizik (tehnički, financijski ili drugi) prilikom stvarne implementacije sustava.

2.2.1.3. Komponente modeliranja

Okruženje modeliranja prema [Maruna, 2006.a] može se promatrati kroz slijedeće esencijalne komponente:

- jezik za modeliranje,
- metodika,
- tehnike koje se koriste tijekom modeliranja i
- razvojni alati.

Jezik za modeliranje: je standardizirana notacija, pretežno grafička, kojom se služi neka metodika za prikaz modela. Prema [Picek, 2004., str. 43.], jezik za modeliranje mora uključivati:

- *Elemente modela* – predstavljaju temeljne koncepte i njihovu semantiku,
- *Notaciju* – način vizualizacije elemenata modela i
- *Smjernice* – upute za korištenje unutar struke.

Metodika: prema [Strahonja *et al.*, 1992. str. 24.] propisuje i normativno uređuje skup mogućih načina rješavanja posla, koji moraju biti formalizirani. Proces razvoja predstavlja krucijalnu komponentu svakog razvoja. Kako projekti postaju sve složeniji potrebo je znati *tko* (učesnik), *kada* (faza), *što* (artefakt) i *na koji način* (aktivnost) treba modelirati.

Tehnika: je skup praktičnih metoda i vještina obavljanja posla u konkretnoj situaciji [Strahonja *et al.*, 1992. str. 26.]. U razvoju programskog proizvoda najčešće se koriste dijagramske tehnike.

Razvojni alati: Mogu se koristiti na svim razinama apstrakcije. Najčešći problem koji se može javiti tijekom modeliranja vezan je uz interoperabilnost i gubitka informacija.

Ove četiri komponente razmatraju se u svakoj od navedenih perspektiva modeliranja.

2.2.2. Modeliranje poslovnih procesa

S teorijske perspektive, metamodel procesa sadrži koncepte kojima se opisuje: što se dešava, na koje događaje, kada i zašto.

Kao što je već rečeno, *modeliranje poslovnih procesa* je opisivanje poslovnih procesa poduzeća, njegovog okruženja i načina na koji ono ostvaruje veze s okruženjem. Rezultat modeliranja je model poslovnih procesa koji sadrži dijagrame koji prikazuju procese iz određenih perspektiva [Bosilj-Vukšić, 2004.].

Ključno je pitanje zašto modelirati poslovne procese?

Prema [IBM Software Group, 2005.], [Bosilj-Vukšić, 2004.] neki od važnih razloga modeliranja poslovnih procesa su:

- razumjeti *trenutačne probleme* u ciljnoj organizaciji te identificirati *potencijalna poboljšanja postojećih procesa ili uvođenje novih*,
- procijeniti utjecaj *organizacijskih promjena*,

- osigurati da svi koji su u kontaktu s organizacijom imaju *zajedničko razumijevanje organizacije*,
- izvući *korisničke zahtjeva za razvoj novog IS-a ili promjenu postojećeg* koji će pružiti podršku ciljnoj organizaciji i
- razumjeti kako će se *novi* (*eng.* To-Be) *IS uklopiti u organizaciju.*

U nastavku se razmatraju opće karakteristike modeliranja u kontekstu modeliranja poslovnih procesa.

Jezik za modeliranje

Postoji katalog svih standarda za modeliranje poslovnih procesa, definiran na Internet stranici korporacije *Cover Pages* [Coverpages, 2004.]. Dvije standardizirane grafičke notacije koje se danas najčešće primjenjuju za modeliranje poslovnih procesa su: BPMN i UML [Kalnins, Vitolins, 2006.].

I dok se BPMN notacija koristi u kontekstu poslovnih procesa, UML se koristi u razvoju programskih proizvoda. Koja je razlika u ovim notacijama? UML koristi objektno orijentirani pristup modeliranju aplikacija, dok BPMN koristi procesno orijentirani pristup modeliranju sustava. I dok BPMN notacija stavlja u fokus poslovne procese, UML je usmjeren na razvoj i implementaciju aplikacija. Stoga se ne radi o dvije međusobno konkurentne, već kompatibilne notacije. Model poslovnog sustava ne mora biti nužno implementiran preko BPEL-a (XML jezik namijenjen opisu poslovnog procesa pri čemu većina zadataka predstavlja interakciju između procesa i web servisa), već se može preslikati u koncepte kao što su slučajevi korištenja i dijagrami dinamike u UML-u.

BPMN (*eng.* Business Process Modeling Notation) je standardizirana grafička notacija za modeliranje poslovnih procesa i njihovog tijeka kreiranjem BPD dijagrama (eng. Business Process Diagram). Razvila ju je, 2004. godine, BPMI.org (eng. Business Process Management Initiative) organizacija. Kako su u lipnju 2006. kompanije BPMI.org i OMG objavile spajanje, aktivnosti BPM-a povjeravaju se tada osnovanoj grupi Business Modeling & Integration (BMI) Domain Task Force (DTF) koja standard i održava do danas. Prilikom spajanja OMG je prihvatio samu notaciju te izdao novu verziju sa svojim logotipom. [White, 2006.], [Maruna, 2006.b], [Sooksanan, Roongruangsuwan, 2005.]. U vrijeme zadnje provjere (travanj 2008.) aktualna je upravo verzija BPMN 1.0 dok je BPMN notacija verzije 2.0 u postupku prihvaćanja i se može pronaći kao prijedlog na web stranici OMG organizacije.

Primarni cilj bio je pružiti notaciju koja će biti razumljiva svim poslovnim korisnicima, od analitičara poslovnih procesa (*eng.* business analyst) koji kreira inicijalne dijagrame procesa do programera (*eng.* developers) odgovornih za implementaciju tehnologije koja će izvoditi te procese [White, 2006.], [OMG, 2006.].

BPMI.org organizacija razvila je niz i standarda za oblikovanje, implementaciju, izvršenje, održavanje i optimizaciju procesa koji su također prihvaćeni.
Prema [Maruna, 2006.b] četiri su temeljna standarda u modeliranju poslovnih procesa:

- BPMN: pruža definiranje i razumijevanje unutrašnjih i vanjskih poslovnih procedura primjenom BPD (*eng.* Business Process Diagram).
- BPML – meta -jezik za modeliranje poslovnih procesa.
- BPQL – standardizirano sučelje za nadolazeće sustave.
- BPEL4WS – notacije specijalizirana za modeliranje poslovnih procesa realiziranih kao web servisa.

Na taj način BPMN stvara most koji je povezuje jaz između modeliranja poslovnih procesa i njihove implementacije [Maruna, 2006.b]. Detaljna BPMN notacija definirana je specifikacijom [OMG, 2006.] koja se može pronaći na već spomenutoj stranici.

U odnosu na OMG kao lidera u industriji, postoji koalicija pod nazivom **WfMC** (*eng.* Workflow Management Coalition) koja je također razvila 2 standarda XPDL 2.0 i Wf-XML 2.0.

UML (*eng.* Unified Modeling Language). Iako je UML idejno razvijen za modeliranje softvera neki ga koriste kao notacija za modeliranje poslovnih procesa, preciznije rečeno primjenjuje se njegov dijagram aktivnosti. Prema [Kalnins, Vitolins, 2006.] to je najobećavajuća notacije za definiranje tijeka procesa (koja je ujedno stereotipizirana - profil) zbog precizno definirane semantike i opće popularnosti UML-a. Organizacija OMG predložila je *UML Profile for Business Process Definition*, i *Business Process Runtime Interfaces Platform Independent Model)* koji se temelje na UML notaciji.
Kako navodi [Kalnins, Vitolins, 2006.] u odnosu na UML, BPMN ima skup nedostataka, a posebno se ističu dva: neformalna semantika i nedostatak adekvatnih obilježja za definiranje podataka.

Metodika

U kontekstu modeliranja procesa najpoznatiji je *Zachman framework*. Uz ovaj okvir (eng. framework) najpoznatije metodike koje se danas još primjenjuju su i: *RUP SE* (eng. Rational Unified Process Systems Engineering), i *EUP* (eng. Enterprise Unified Process).

Tehnike

Koriste se dijagramske tehnike definirane metodikom.

Razvojni alati

Najznačajniji alati koji su na današnjem tržištu u širokoj primjeni su:

- Casewise: Corporate Modeler,
- IDS Scheer AG: ARIS tools,
- Sybase: Workspace,
- IBM: WebSphere Business Modeler.

2.2.3. Modeliranje u razvoju programskog proizvoda

Programski sustavi danas postaju sve složeniji te modeliranje pruža efikasan način boljeg razumijevanje sustava koji je potrebno razviti kao i tehniku smanjenja rizika koje razvoj donosi. Tijekom razvoja moguće je definirati različite modele ovisno o namjeri koju želimo postići njihovom primjenom.

Važnost modeliranja, kao kritične komponente u razvoju programskog proizvoda, može se prepoznati iz i članka [Booch] u kojem Gardy Booch, razvoj programskog proizvoda uspoređuje s izgradnjom *kućice za pse*, *obiteljske kuće* i *neboderom*. Vrlo se lako može prepoznati činjenica da se nitko ne bi usudio graditi neboder, a da prethodno nije razvijena i proučena projektna dokumentacija koja uključuje različite tipove modela.

Krajnji korisnici, analitičari sustava, projektanti, programeri, testeri sustava, voditelji projekata i još mnogi drugi koji sudjeluju u razvoju programskog proizvoda gledaju na sustav iz svoje perspektive, želeći informacije značajne za njihov dio razvoja. Stoga se prilikom modeliranja, programski proizvod promatra iz različitih perspektiva. Razvoj dobrih modela koji predstavljaju specifikaciju problema i/ili rješenja iz odgovarajuće perspektive pridonosi kvaliteti budućeg programskog proizvoda.

Modeliranje složenih aplikacija donosi nekoliko **koristi** [Cernosek, Naiburg, 2004., str. 4.]:

- bolje razumijevanje poslovne domene ("As-Is" model),
- kvalitetnije oblikovanje arhitekture budućeg sustava ("To-Be" model) i
- vizualizaciju programskog koda i drugih oblika implementacije.

U [Yu, 2007. str. 2.] uz ove 3 koristi, autor navodi dodatne značajke modela koje će biti važne za budući razvoj softvera, a to su:

- standardizacija,
- produktivnost - generiranje programskog koda.

Zapravo, u razvoju programskog proizvoda može se govoriti o *spektru modeliranja* [Cernosek, Naiburg, 2004., str. 4-10.], [Brown, 2004., str 2.] kojeg čine:

- pristup orijentiran samo na programski kod primjenom integriranog razvojnog okruženja - IDE (eng. Integrated Development Environment),
- vizualizacija koda,
- reverzibilno inženjerstvo i
- pristup orijentiran na model.

Svaki od navedenih oblika modeliranja nalaze se na različitim razinama apstrakcije. Naglasimo sada komponente koje se koriste u modeliranju programskih proizvoda.

Jezik za modeliranje – notacija

S obzirom na ciljeve modeliranja i čimbenike koje mora sadržavati jezik za modeliranje, a koji su gore navedeni, danas je SW industrija usvojila *UML* kao standardiziran i dobro definiran unificirani jezik koji predstavlja kulminaciju inženjerskih praksi koje su se pokazale uspješnima u modeliranju velikih i složenih sustava.

Analitičari sustava, projektanti, programeri i dr. koriste UML za specificiranje, vizualizaciju, konstrukciju i dokumentiranje svih aspekata softvera [Cernosek, Naiburg, 2004., str. 7.]. UML nije samo standardizirana grafička notacija, on je jezik za modeliranje sa definiranom sintaksom (grafičkom i tekstualnom) i semantikom (značenjem koncepata). Detalji ovog unificiranog jezika za modeliranje mogu se pronaći specifikacijama na internet stranicama korporacije OMG.

Osim UML-a, industrija kreće i prema *DSL's jezicima* (eng. Domain Specific [Modeling] Languages) usmjerenih na konkretnu problemsku domenu ili pak se za te domene UML

proširuje stvaranjem UML profila za pojedine problemske domene [Cernosek, Naiburg, 2004., str. 10.].

Formalni jezici također se koriste u modeliranju programskih proizvoda. Predstavljaju stogo uređen skup koncepata kojima se deklarativno opisuju pravila i ograničenja koja je potrebno zadovoljiti tijekom razvoja. Služe za definiranje izraza u kojima bi primjena prirodnog jezika mogla izazvati dvosmislenost te kako bi se izbjegli složeni matematički izrazi. Primjer jednog takvoj jezika je OCL - (eng. Object Constraint Language). Služi za opis pravila i definiranje ograničenja, može se koristi tijekom UML modeliranja, standardiziran je i predstavlja proširenje UML-a. Ujedno predstavlja ključnu komponentu novog OMG standarda namijenjenog za definiranje transformacija modela pod nazivom **QVT** (Query/Views/Transformations), najznačajnijeg segmenta uspješne realizacije MDD paradigme.

Metodike

Tijekom posljednjih nekoliko desetljeća razvijen je velik broj procesa razvoja (formalnih i agilnih) od kojih su u današnje vrijeme najpoznatije: *RUP, MSF, XP, Scrum* i druge. Točka 6.1 detaljnije govori o metodikama koje se danas primjenjuju u industriji.

Tehnike

Tijekom modeliranja koriste se dijagramske tehnike kojima se kreira jedan ili više dijagrama koji čine model.

Razvojni alati

Kako se u kontekstu razvoja programskog proizvoda govori o spektru modeliranja tako su različiti proizvođači razvili cjelokupne obitelji alata koji podržavaju taj spektar. Tako su na današnjem tržištu u širokoj primjeni obitelji alata:

- IBM RATIONAL alati: IBM Rational Software Modeler, IBM Rational Aplication Developer, IBM Rational Software Architect (RSM+RAD),
- SYBASE alati: PowerDesigner, PowerBuilder, Workspace,
- i drugi.

3. RAZVOJ PROGRAMSKOG PROIZVODA TEMELJEN NA MODELIMA

U ovom se poglavlju prikazuje jedan od dominantnih smjerova razvoja programskih proizvoda poznat pod nazivom *razvoj temeljen na modelima*. Kako se ulažu veliki napori u savladavanje ideja na kojima se temelji ova paradigma, poglavlje će detaljnije opisivati trenutne realizacije te paradigme.

3.1. Paradigma temeljena na modelima

3.1.1. Pojam

Razvoj programskog proizvoda temeljen na modelima odnosi se na pojam koji je u literaturi poznat pod nazivima *Model Driven Develompnet* (MDD) ili *Model Driven (Software) Engineering* (MDSE). U radu će se koristi kratica iz engleskog govornog područja - MDD.

Iz dosadašnjeg razmatranja proizlazi kako SW industrija u cilju savladavanja rastućih izazova, evoluira prema novoj paradigmi razvoja. Intenzivno se istražuju mogućnosti razvoja programskih proizvoda koji su u spektru modeliranja definirani na najvišoj razini apstrakcije, a to je razvoj temeljen na modelima. Želja je primjenom modela povećati razinu apstrakcije tako da se u modelima definira dovoljno detalja kako bi se, na temelju njih, generirao programski kod za neke dijelove ili cjelokupni softver.

Slijedi nekoliko definicija paradigme temeljene na modelima.

Prema [Yusuf *et al.*, 2006., str. 1.] *MDD predstavlja stil razvoja programskih proizvoda u kojem su primarni softverski artefakti modeli, na temelju kojih se, prema najboljim inženjerskim praksama, generira kod ili neki drugi artefakti.*

U [Hailpern, Tarr, 2006., str. 452.] *MDD* se definira kao *pristup razvoju programskih proizvoda koji se sastoji od modela kojima se povećava razina apstrakcije na kojoj razvojni inženjeri stvaraju softver, s ciljem pojednostavljenja i formaliziranja različitih aktivnosti i zadataka koji se pojavljuju u razvojnom ciklusu programskih proizvoda.*

MDD, prema [Brown *et al.*, 2006., str. 468.], *predstavlja paradigmu u kojoj se na temelju modela, kod generira automatski ili poluautomatski, pri čemu se koristi standardizirani specifikacijski jezici za opis modela i transformacija između njih.*

Ove definicije jasno pokazuju kako je fokus MDD paradigme pomaknut s *programiranja* na *modeliranje*, te da *modeli* predstavljaju ključne artefakte.

Već je spomenuto da se, prilikom modeliranja, softverski modeli prikazuju u jednoj od notacija (jeziku modeliranja). Najčešće je to UML - standardizirani jezik za specifikaciju, vizualizaciju i dokumentiranje programskih sustava. UML predstavlja vizualnu notaciju kojom se prikazuje semantika softverskih modela. Tijekom dosadašnjeg razvoja, UML modeli najčešće se koriste samo kao *skice* kojima se na neformalan način prikazuje neki aspekt sustava ili kao *nacrti* kojima se detaljno opisuje dizajn sustava koji se tada ručno implementira.

Primjena modela za specifikaciju i dokumentiranje izrazito je vrijedna, međutim to zahtjeva strogu disciplinu kako bi modeli u svakom trenutku odgovarali trenutnoj implementaciji. Vrlo često se zbog nedostatka vremena tijekom razvoja mijenja samo programski kod, dok modeli ostaju ne ažurirani, prikazujući pogrešnu situaciji što može biti opasnije i od samog nepostojanja istih.

Stoga se u MDD razvoju modeli ne koriste kao *skice* ili *nacrti* već kao primarni artefakti na temelju kojih se implementacija provodi transformacijama modela u programski kod. Iz tog razloga je prilikom razvoja neke komponente ili cjelokupnog softvera, fokus primarno usmjeren na izradu modela za neku aplikacijsku domenu. Programski kod kao i drugi artefakti, karakteristični za problemsku domenu, generiraju se na temelju modela prema definiranim transformacijama modela. Može se reći da modeli predstavljaju *stepenice* na putu između opisa sustava (korisničkih zahtjeva) i programskih komponenata koje će predstavljati rješenje. Kako se povećava razumijevanje korisničkih zahtjeva, razvoj softvera evoluira - modeli postaju potpuniji, precizniji i međusobno konzistentniji. Pri tome se modele, koji se nalaze na višim razinama apstrakcije, koristi kako bi se na automatizirani način - generiranjem, definirao programski kod koji je na najnižoj razini apstrakcije.

Prema Seliću [Pierson, 2007., str. 4.] bit MDD je u dvije stvari. Prva je **apstrakcija**, tj. kako će se promišljati o problemu i kako će se tada prikazati rješenje. Druga stvar, koja se često zaboravlja, je **uključivanje automatizacije** u razvoj programskog proizvoda u što je moguće većoj mjeri, posebice primjenom razvojnih alata i cjelokupnih integriranih razvojnih

okruženja.

Prema [Yusuf *et al.*, 2006., str. 3.] MDD automatizacija ide i dalje od samog generiranja programskog koda. Kako je tijekom razvoja softvera potrebno definirati i mnoge druge artefakte (eng. non-code), neki od njih mogu se potpuno ili djelomično izvesti iz modela. Neki od njih su:

Dokumentacija: U organizacijama koje slijede neki formalni proces razvoja, izrada dokumentacije oduzima značajan dio vremena. Ujedno je teško održavati korak dokumentiranja sustava sa samom implementacijom. Primjenom MDD, dokumentacija se generira iz modela što osigurava konzistentnost, a informacije koje su razvojnim inženjerima potrebe dostupne su u svakom trenutku tijekom razvoja.

Artefakti za testiranje: Današnji MDD alati omogućavaju generiranje osnovnih testova na temelju modela. Ukoliko se provodi dodatno modeliranje specifičnih testova, primjenom UML profila za testiranje, tada je moguće generirati cjelokupno testiranje implementacije.

Skripte: Ukoliko arhitekti sustava definiraju transformacije, moguće je na temelju modela generirati različite skripte koje se tada mogu izvršavati na nekoj platformi.

Drugi modeli: Razvoj programskog proizvoda uključuje i mnogo međuovisnih modela na različitim razinama apstrakcije, pri čemu prikazuju različite dijelova sustava (GUI, baza podataka, administriranje), obuhvaćaju različita pitanja (sigurnost, performanse i mogućnosti proširenja) kao i različite zadatke koje je potrebno izvršiti. U mnogim slučajevima moguće je djelomično generirati model na temelju drugog modela.

Uzorci: Sadrže rješenja koji predstavljaju najbolju praksu problema koji se ponavlja u nekom kontekstu. Uzorci specificiraju karakteristike elemenata modela i njihovu međusobnu povezanost. Uzorke bi trebalo oblikovati na takav način da su primjenjivi u modelu te da se mogu automatizirati - kreiranjem novih elemenata, ili modificiranjem već postojećih.

MDD ujedno naglašava razvoj programskoj proizvoda neovisan o platformi, što omogućava razvojnim inženjerima koji posjeduju znanja iz neke domene, da oblikuju sustav neovisno o konceptima vezanim za određenu platformu. Ekspertiza platforme, kao i odluke o implementaciji arhitekture sustava nastoje se definirati izravno u transformacijama modela.

3.1.1.1. Elementi MDD paradigme

Što čini jezgru MDD paradigme?

Iz gornjih definicija i opisa MDD može se zaključiti da su ključni elementi:

- modeli,
- modeliranje i
- transformacije modela.

Modeli

Tijekom razvoja programskog proizvoda, uz samu aplikaciju, kreira se veliki broj drugih artefakata. Kako su temelj MDD paradigme modeli, modeli koji se definiraju tijekom razvoja su:

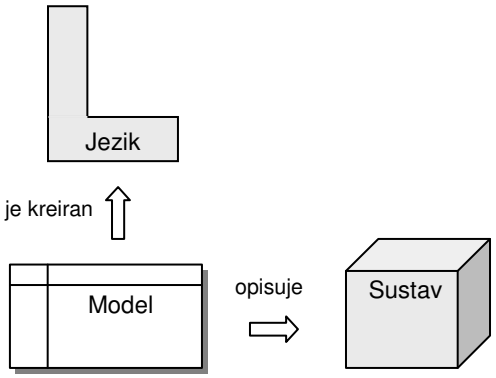
- dokumenti u kojima se definiraju korisnički zahtjevi i druga relevantna dokumentacija napisana na prirodnom jeziku,
- modeli poslovnih procesa,
- modeli analize i dizajna sustava,
- model podataka (sheme baze podataka),
- izvorni programski kod napisan u određenom programskom jeziku kao i konfiguracijske datoteke i
- definicije sučelja.

Uz ove, nastaju još i drugi artefakti, koji se ne smatraju modelima. To su:

- zahtjevi za promjenama,
- izvještaji s testiranja aplikacije i
- statistika projekta.

Iz liste gore navedenih modela jasno je da postoje različiti tipovi modela što zahtjeva detaljniji pogled na modele tj. na *jezik* (notaciju) kojom se definiraju i njihovu *primjenu*.

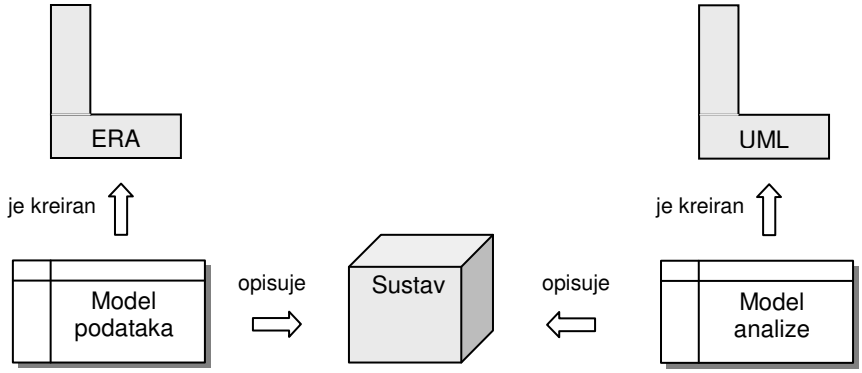
Jezik: Povezanost modela i jezika za modeliranja proizlazi iz *slike 3-1*.



Slika 3-1. Odnos modela i jezika za modeliranje

Izvor: [Kleppe *et al.*, 2003., pogl. 2., str. 3.]

Prema Conallen [DEV325] modeli se definiraju jezikom koji može biti tekstualan (npr. WSDL), vizualan (npr. UML) ili prirodni jezik (npr. engleski), a njihov opis može biti formalan ili neformalan. *Slika 3-2.* prikazuje primjer opisa istog sustava s dva različita tipa modela primjenom dva različita jezika.

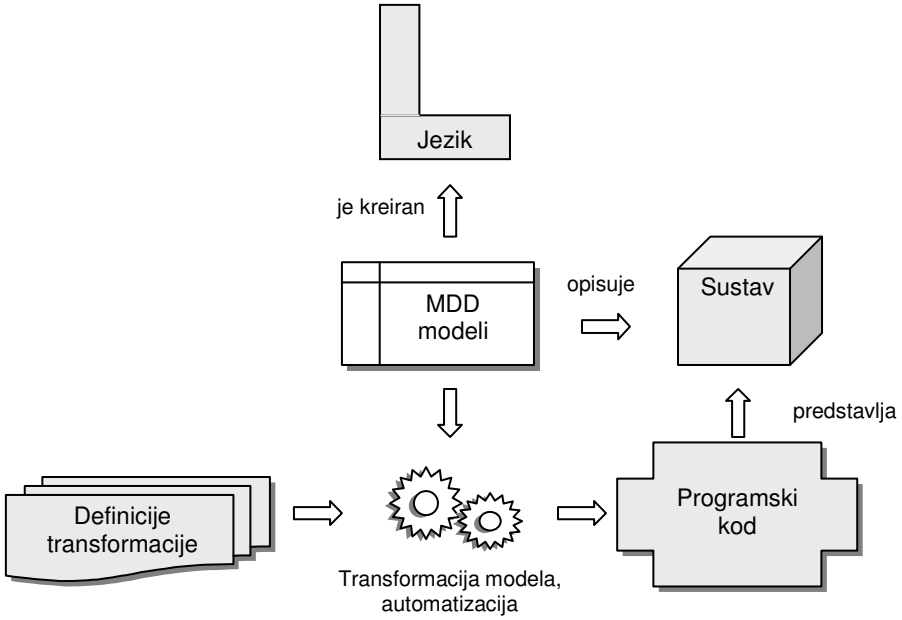


Slika 3-2. Opis istog sustava različitim modelima u kojima se primjenjuju različiti jezici za modeliranje

Izvor: [Kleppe *et al.*, 2003., pogl.2., str. 6.]

Primjena: Postoji velika razlika u onome što modeli reprezentiraju i onoga kako se ti modeli koriste. Kako bi model bio prikladan za MDD razvoj, mora zadovoljavati dodatni zahtjev, a to je da mora biti *strojno čitljiv* (eng. machine readable). Strojna čitljivost modela predstavlja glavni preduvjet generiranja artefakta na temelju modela. Stoga je tijekom modeliranja potrebno obratiti pažnju izrađuje li se model samo kao *skica* ili *nacrt* sustava ili ga se oblikuje na način prikladan za strojno čitanje tj. transformaciju.

Kroz *sliku 3-3.* prikazan je koncept primjene modela u kontekstu MDD razvoja.



Slika 3-3. Primjena modela u kontekstu MDD razvoja

Izvor: [Kleppe *et al.*, 2003., pogl.2., str. 6.]

Modeliranje

Osim klasičnih aktivnosti modeliranja u kontekstu MDD paradigme, modeliranje obuhvaća i [Brown *et al.*, 2006., str. 468-470.]:

- evoluciju i povezanost modela te
- kreiranje definicija transformacija modela,

1. **Evolucija i povezanost modela**

   Tijekom modeliranja modeli prolaze kroz evoluciju, oplemenjivanjem. Ono predstavlja proces postepene promjene modela kako bi se bolje prikazao željeni sustav. Modeli se oplemenjuju na način da se u njega nadodaju dodatne informacije kako se sustav bolje shvaća. Kako modeli evoluiraju, sve povezane modele potrebno je također revidirati.

2. **Kreiranje definicija transformacija modela**

   Definiranje i primjena transformacija modela predstavljaju kritične tehnike u kontekstu bilo kojeg pristupa u MDD paradigmi. Ulaz u transformaciju je model, dok rezultat transformacije (izlaz) može uključivati također model ili izvršni kod na različitim razinama apstrakcije. Stoga je potrebno

definirati način pretvorbe ulaza i izlaza, kreirajući definiciju transformacije koja se sastoji od skupa pravila transformacije.

Za notaciju koja se koristi prilikom modeliranja mora se znati njezina prikladnost za provođenje automatizacije modela.

Transformacije modela

Uloga modela i modeliranja u MDD svakako je iznimno velika. No transformacije modela predstavljaju ključni element realizacije MDD ideje. Transformacija modela temelji se na *definiciji transformacije* i uključuje dva ili više modela (vidi i *sliku 3-3.*). Provode se kada se iz apstraktnog modela na višoj razini apstrakcije želi dobiti apstraktni model (ili više njih) na nižoj razini apstrakcije. Transformacije ne moraju nužno biti jednosmjerne.

Postoje 3 najčešća tipa transformacija modela:

- refaktoriranje,
- model – model,
- model – kod.

Refaktoriranje modela vrši se na temelju nekog dobro definiranog kriterija. U tom slučaju rezultat transformacije je revidiran izvorni model. Npr. preimenovanje svih instanci UML entiteta u kojima se koristi *naziv* ili zamjena jedne klase skupom klasa i njihovih veza u metamodelu i svim dijagramima.

Model – model transformacija predstavlja pretvaranje jednog ili više modela u drugi model ili više njih.

Model – kod transformacija obuhvaća transformaciju elementa modela u dio programskog koda. To rade generatori aplikacija. Programski kod generira se u objektno orijentiranim jezicima kao što su npr. Java i C++. Transformacije nisu samo ograničene na programski kod već je moguće generirati i različite tipove datoteka na temelju modela definiranih u UML notaciji.

Primjeri nekih od ovdje navedenih tipova transformacija prikazuju se u kasnijim dijelovima rada.

Kako je cilj MDD paradigme provoditi automatizaciju modela u kod, transformacije se provode automatizirano, primjenom MDD razvojnog okruženja,

na temelju skupa predefiniranih *definicija transformacije*. Transformacije mogu biti *implicitne* za alate koji se koriste ili *eksplicitne* definirane na temelju znanju iz određene domene. To naravno zahtjeva da ulazni model bude sintaksno i semantički ispravan, potpun te da sadrži informacije potrebne za provođenje transformacije. Npr. u modelu je za neki atribut dovoljno reći da je on tipa *string* no dodatna informacija koja će se koristiti prilikom generiranja mora sadržavati i njegovu dužinu. To zahtjeva provođenje i konfiguriranje transformacije za razvijeni model.

3.1.2. Prednosti primjene MDD paradigme

MDD paradigma ima potencijala kojim se u velikoj mjeri može unaprijediti današnju praksu razvoja programskih proizvoda. Taj potencijal očituje se u savladavanju trenutnih izazova – smanjenju troškova razvoja te povećanju konzistentnosti i kvalitete softvera.

Prema dostupnoj literaturi analizirane su potencijalne prednosti primjene MDD paradigme. Rezultati analize, iskazani kroz tablicu 3-1., ističu elemente u kojima MDD paradigma pridonosi unapređenju u SW industriji.

Tablica 3-1. Koristi MDD paradigme u SW industriji.

Prednost primjene MDD paradigme	**Objašnjenje**
Povećanje produktivnost	Smanjenje troškova razvoja generiranjem programskog koda i drugih artefakata na temelju modela.
Održavanje	U mnogim su organizacijama softverske komponente implementirane tehnologijom koja je zastarjela i nad kojom organizacija više nema stručnost. MDD pruža arhitekturu koja omogućava konzistentnu nadogradnju i efikasnu migraciju komponenata u nove tehnologije. Modeli na najvišoj razini apstrakcije ne sadrže detalje vezane uz tehnološke platforme i njihovu tehničku arhitekturu. Promjene u pogledu novih platformi provode se promjenom pravila transformacija u definiciji i ponovnom primjenom nad originalnim modelima. To ujedno pruža mogućnost simulacije prilikom odabira platforme prije nego se donese konačna odluka.
Ponovna iskoristivost starijih sustava	Ukoliko postoji mnogo komponenata implementiranih na istoj zastarjeloj platformi mogu se razviti povratne transformacije kojima će se od komponente dobiti UML dijagram.
Adaptabilnost	Dodavanje i mijenjanje funkcionalnosti poslovanja je izravno jer su sve

	investicije vezane uz automatizaciju već provedene. Prilikom dodavanja nove funkcionalnosti razvija se samo ponašanje karakteristično za tu funkcionalnost. Sve informacije vezane uz generiranje artefakta koji će predstavljati implementaciju već su sadržane u transformacijama.
Konzistentnost	Zbog generiranja otklonjene su mnogobrojne pogreške koje su prisutne u ručnoj implementaciji.
Ponovna upotrebljivost	Svakom novom primjenom razvijenih transformacija povećava se ROI. Primjena testiranih i iskušanih transformacija povećava predvidivost razvoja novih funkcija i smanjuje rizik jer su pitanja vezana uz tehničku stranu i arhitekturu već riješena.
Unaprijeđena komunikacija svih sudionika u razvoju	Modeli apstrahiraju implementacijske detalje koji nisu važni za razumijevanje logičkog ponašanja sustava. Više su vezani uz problemsku domenu, smanjujući semantički jaz između koncepata koje razumiju sudionici u razvoju i jezika u kojem je iskazano rješenje.
Unaprijeđena komunikacija tijekom oblikovanja sustava	Modeli pridonose razumijevanju problemske domene tijekom razvoja što dovodi do kvalitetnije komunikacije i diskusije oko budućeg sustava. S obzirom da su modeli dio sustava, a ne njegova dokumentacija, oni su pouzdani te uvijek prikazuju stvarnu sliku sustava.
Učahurivanje znanja	Projekti vrlo često ovise o ekspertu koji svaki puta iznova donosi odluke koje predstavljaju najbolju praksu. U MDD to je znanje učahureno u *uzorke* i *transformacije* koji se ažuriraju i primjenjuju neovisno o prisutnosti ljudskog eksperta.
Modeli kao ponovno iskoristivi programski resursi	Modeli su važni resursi koji prikazuju što softver radi u organizaciji. Njihovom mogućnosti mijenjanja ovisno o promjenama predstavlja važan čimbeniku u razvoju softvera.
Mogućnost odgađanja odluka vezanih uz odabir tehnologije	U početnim fazama razvoja naglasak je stavljen na modeliranje problemske domene. Stoga je izbor određene tehnološke platforme moguće odgoditi dok se ne prikupe dodatne informacije. U domenama u kojima je razvojni ciklus ekstremno dugačak to je presudan čimbenik.

Izvor: [Swithinbank *et al.*, 2005., str. 9,10.], [Yusuf *et al.*, 2006., str. 6,7.]

3.1.3. Analiza trenutnih problema realizacije MDD paradigme - nedostaci

Primarni cilj MDD paradigme je povećanje razina apstrakcije na kojoj razvojni inženjeri djeluju, što bi smanjilo vrijeme uloženo u razvoj kao i složenost artefakata koje se koriste tijekom razvoja [Hailpern, Tarr, 2006., str. 457.], [Greenfield, Short, 2004. str. 1.]. Naravno potrebno je postići određeni kompromis između želje za povećanjem razine apstrakcije, s jedne strane, i uključivanja detalja potrebnih za transformacije modela s druge strane.

Problemi koji se pojavljuju, kao što se može i pretpostaviti, vezani su uz apstrakciju modela na različitim razinama tijekom razvojnog ciklusa programskog proizvoda. Otvorena pitanja koja se pojavljuju su: *Kako provesti transformaciju modela na jednoj razini apstrakcije, u model ili kod na nižoj razini apstrakcije?* U pokušaju da se odgovori na to pitanje pojavljuju se nova. *Kako koristiti modele?* Već je spomenuto da se modeli mogu koristiti kao skice ili kako nacrti sustava, a da MDD okruženje zahtjeva da se modele promatra kao programski jezik. To opet nameće pitanje: *Kojom notacijom/jezikom za modeliranje se služiti kako bi se postigla automatizacija modela?* Dakako da je neophodna standardizirana notacija za modeliranje kako bi se MDD paradigma realizirala. No vodeći standardizirani jezik za modeliranje – UML, koji se u industriji na veliko primjenjuje, izaziva mnogobrojne kritike kada ga se smještava u kontekst MDD paradigme. Smatram da je u želji da postane sveobuhvatan, UML postao presložen, dvosmislen, da sadrži neke dijagrame ili koncepte koji su redundantni ili nepotrebni jer se rijetko koriste. Zatim, UML-u nedostaje referentne implementacije kao i razumljive semantike kako bi razvojni inženjeri jednoznačno interpretirali pojedina svojstva te ih ugradili u razvojne alate. Nedostatak jasno razumljive semantike čini proizvodnju MDD razvojnih alata težim, a ona je zapravo ključan element automatizacije. Uz sve to, mnogi su problemi vezani uz učenje i primjenu notacije prilikom rada na konkretnim projektima. Sve to dovodi do razumnog pitanja: *Da li je UML prikladan kao programski jezik za definiranje modela na temelju kojih će se moći provesti automatizacija?* Kako se ovaj dio ne bi temeljio na osobnim stavovima evo kratke analize razmišljanja drugih autora. UML 2.0 kao programski jezik modela temelji se na uvjerenju da će više razine apstrakcije doprinijeti većoj produktivnosti inženjera u odnosu na trenutne programske jezike. No, da li je ovo razmišljanje opravdano?

Fowler u [Fowler], ne vjeruje da će grafička notacija u programiranju biti od velike koristi samo zato što je grafička. On navodi kako je tijekom svog iskustvu vidio slične pokušaje koji su propali, jer je njihova primjena bila sporija nego li samo pisanje programskog koda. Za usporedbu navodi primjer pisanja koda i izradu dijagrama tijeka za neki algoritam. On smatra da će, čak i ukoliko bi, UML kao programski jezik bio produktivniji od dosadašnjih programskih jezika, trebati proći dosta vremena kako bi ga se usvojilo. Nadalje, Greenfield [Greenfield *et al.*, 2004., str. 47.] kaže da, iako je UML 2.0 koristan kao jezik za modeliranje, prvenstveno misleći pri tome za dokumentiranje sustava, smatra da nije prikladan za MDD razvoj, jer je primarno UML razvijan za dokumentiranje sustava, a ne za programiranja, te smatra da nema dobre temelje da bude prikladan za MDD razvoj. On navodi kako su za MDD razvoj prikladniji jezici specifični za pojedine domene (eng. DSL's).

Frankel [Frankel, 2003. str. 73.] navodi kako postoje dijelovi UML koji se ne mogu efikasno implementirati i da samo neki alati, ako i oni postoje na tržištu, mogu implementirati UML na način definiran u specifikaciji. On navodi kako UML nije jedini jezik na kojem bi se MDD trebao temeljiti [Frankel, 2003. str. 77.].

Prema [Hailpern, Tarr, 2006., str. 460.] primjena MDD paradigme nameće i druge probleme poput:

- Redundantnosti koja se javlja zbog postojanja više prikaza modela na različitim ili istim razinama koji predstavljaju isti koncept.
- Rastući problemi reverzibilnog inženjerstva (eng. round trip) jer se povećanjem broja modela i razina apstrakcije na kojima se ti modeli nalaze povećava i broj veza između njih. Mnoge takve veze su poprilično kompleksne pa se tako otežava reverzibilno inženjerstvo. Jednako tako problemi nastaju kada se pogreške dogode na nižim razinama (npr. u kodu) pa je potrebno pomoću povratne veze mijenjati sve modele na višim razinama (kojih može biti puno). Problem se još više povećava ako su uključene i transformacije tj. ako se na nekom modelu naprave neke promjene one mogu biti izgubljene nakon reinženjeringa i ponovnog generiranja tog modela.
- Premještanje kompleksnosti umjesto njezina smanjenja.
- Potrebe za većom ekspertizom.

3.2. Pristupi razvoju u kontekstu paradigme temeljene na modelima

U okviru ove paradigme trenutno postoje dva dominantna pogleda na njezinu realizaciju. To su [MetsSoftware, 2004.]:

- arhitektura temeljena na modelima (eng. MDA) i
- tvornice softvera (eng. Software Factories)

U nastavku rada koristit će se kratice iz engleskog govornog područja: MDA (eng. Model Driven Architecture) i SF (eng. Softvare Factories).

Kako su dvije vodeće realizacije MDD paradigme: **MDA**, pristup koji razvija OMG konzorcij (eng. Object Management Group) korporacija u suradnji s firmama partnerima i **SF**, pristup koji razvija Microsoft, u nastavku rada detaljno će se opisati osnovni koncepti ta dva pristupa.

3.2.1. Arhitektura temeljena na modelima - MDA

3.2.1.1. Osnove MDA

U 2000.godini, OMG konzorcij objavljuje članak pod nazivom: *Model Driven Arhitecture* u kojem opisuje viziju razvoja programskog proizvoda temeljenu na povezivanju modela prilikom izgradnje cjelovitih sustava [Mellor et.al., 2004., str. XVI]. Kao temelj tog pristupa, bili su istaknuti tadašnji, ali i budući OMG standardi s osnovnom idejom definiranja modela koji će se transformirati u programski kod. Imajući tu ideju kao nit vodilju, 2001. godine OMG konzorcij ju je i službeno usvojio kao pristup razvoja programskih proizvoda temeljen na modelima [MDA Guide, 2003., str. 2-1.].

3.2.1.1.1. Definicija

Kako trenutno MDA, kao OMG-ova inicijativa razvoja temeljnog na modelima, ima velik utjecaj u SW industriji neophodno je istaknuti neke definicije shvaćanja tog pojma.

Prema [MDA Guide, 2003., str. 2-2.] *MDA je pristup koji se temelji na ideji odvajanja specifikacije sustava o njegovom radu od detalja vezanih za njegovu realizaciju na određenoj platformi.* To uključuje:

- specificiranje sustava neovisno o platformi na kojoj će biti realiziran,
- specificiranje platformi,
- odabir određene platforme za sustav,
- transformaciju specifikacije sustava u jednu od odabranih platformi.

Prema [Kontio, 2005., str. 1.] *MDA je pristup neovisan o platformi i proizvođačima u kojem se definira arhitektura, dizajn i implementacija sustava. Primjenjiv je na cjelokupan razvojni ciklus primjenom otvorenih standarda kao što su UML, XMI, XML i CORBA.*

Prema [Kleppe *et al.*, 2003. str. 1.] *MDA je okvir razvoja programskog proizvoda koji se temelji na unificiranom jeziku za modeliranje (UML) kao i drugim industrijskim standardima za vizualizaciju, pohranu i razmjenu modela. MDA promovira modele koji su strojno čitljivi, na najvišim razinama apstrakcije, koji su neovisni o tehnologiji implementacije i pohranjeni u standardiziranim repozitorijima.*

Iz ovih definicija proizlazi da se u MDA pristupu, sustav modelira na temelju svoje funkcionalnosti, a ne na temelju jezika, platforme ili tehnologije što znači da takav sustav može biti mijenjan ili nadograđivan bez potrebe da se zadire u jezgru infrastrukture sustava.

3.2.1.1.2. Ciljevi i načela MDA

Prilikom usvajanja definirana su 3 osnovna cilja MDA pristupa. To su [MDA Guide, 2003., str. 2-2.], [DEV325]:

1. **Prenosivost:** Omogućava da model bude realiziran na različitim platformama.
2. **Interoperabilnost:** Modeli trebaju biti povezani i primjenjivi kroz različite domene. Točke koje će predstavljati poveznice treba dokumentirati kao i definirati translaciju funkcionalnosti između dviju domena.
3. **Ponovna iskoristivost:** Kako su modeli nastali primjenom standardizirane notacije moguće ih je jednoznačno primjenjivati više puta pri čemu ti modeli mogu podržavati interoperabilnost i portabilnost.

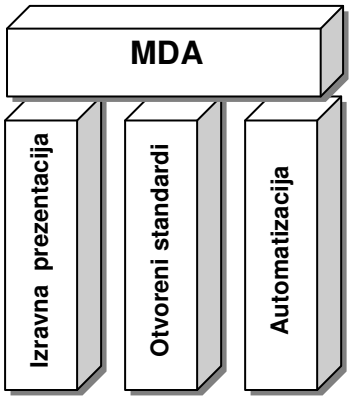
Brown [Brown, 2004., str. 2.] ističe četiri principa na kojima se temelji OMG-ov pogled na MDA. To su:

1. Modeli izraženi u dobro definiranoj notaciji predstavljaju temelj razumijevanja poslovnog sustava za koji se izrađuje aplikacijsko rješenje.
2. Izgradnja sustava može biti organizirana oko skupa modela koji će se temeljiti na znatnom broju transformacija između modela organiziranih u razine.
3. Formalni opis modela skupom metamodela olakšava smislenu integraciju i transformaciju između modela i predstavlja temelj automatizacije primjenom razvojnih alata.
4. Prihvaćanje i šira primjena MDA pristupa zahtjeva otvorenost industrijskih standarda.

3.2.1.1.3. Koncept i temeljni pojmovi MDA

Kako bi se ostvarili gore navedeni ciljevi, koncept MDA sastoji se od realizacije tri komplementarne ideje prikazane na *slici 3-4*.[DEV325]:

1. Izravno prikazivanje problemske domene.
2. Primjena otvorenih standarda.
3. Automatizacija razvoja.



Slika 3-4. Temelji MDA koncepta

Izvor: [DEV325]

1. **Izravno prikazivanje problemske domene.** Kako SW industrija sazrijeva, povećava se i razina apstrakcije u razvojnim alatima koji se koriste tijekom faza dizajna i implementacije. Svaka nova inovacija povećava razinu apstrakcije. Smanjenje semantičke razlike između problemske domene i reprezentacije omogućava izravniju povezanost između rješenja i problema što vodi do preciznijeg i kvalitetnijeg dizajna kao i do povećanja produktivnosti.

2. **Primjena otvorenih standarda.** Smanjuje velika odstupanja unutar mnoštva alata različitih proizvođača, čime se širi opće prihvaćanje arhitekture među korisnicima. Omogućavajući otvorenost različitih standarda povećava se kompatibilnost između različitih hardverskih i programskih komponenti. Standardizacijom i otvorenošću MDA inicijative stvoreno je okruženje u kojem je omogućena interoperabilnost između alata različitih proizvođača.

3. **Automatizacija razvoja.** Podrškom razvojnih alata automatiziraju se oni dijelovi razvoja koji ne zahtijevaju sudjelovanje i znanje čovjeka. Ujedno, tijekom ručnog prevođenja s više razine apstrakcije prema nižim, dolazi do izlaganja riziku u pogledu vremena isporuke i kvalitete rješenja. Jedan od glavnih ciljeva automatizacije u MDA je i smanjenje jaza između tehnologije i problemske domene na način da se eksplicitno modeliraju i problemska domena i tehnologija unutar nekog okvira koji se kasnije može iskoristiti prilikom izrade aplikacija. Automatizacija igra važnu ulogu u

smanjenju rizika jer se primjenom transformacija povećava brzina razvoja i smanjuje mogućnost pogrešaka. Pri MDA pristupu ulaz u kompajler je model, a ne programski kod.

Kako u kontekstu MDA pristupa postoji mnogo akronima i pojmova, u nastavku se prikazuju temeljni pojmovi koji čine jezgru MDA, a temelje se na: [MDA Guide, 2003., str. 2-2- 2-6.], [Mellor et.al., 2004., str. 13-19.].

Sustav. U kontekstu MDA, pojam *sustav* odnosi se na programski proizvod koji se razvija ili koji već postoji i primjenjuje se u nekoj organizaciji.

Model. Predstavlja formalnu specifikaciju funkcionalnosti, strukture i ponašanja sustava u nekom kontekstu s određene perspektive (pogleda) primjenom odgovarajuće notacije – jezika za modeliranje.

Temeljeno na modelima. Pojam kojim se opisuje pristup razvoju SW u kojem se modeli koriste kao primarni artefakti za dokumentiranje, analiziranje, oblikovanje, implementaciju i održavanje sustava.

Arhitektura sustava. Predstavlja specifikaciju dijelova i konstrukcije koja čini sustav. U kontekstu MDA, komponente konstrukcije i pravila definirana su skupom međusobno povezanih modela.

Pogled. Tehnika apstrakcije kojom se pažnja usmjeruje na određeni aspekt sustava pri čemu se svi nebitni detalji zanemaruju. Čini ga jedan ili više modela.

MDA pogledi. Tri su zadana pogleda u MDA pristupu:

- *ICT neovisan pogled* (eng. Computation Independent Viewpoint - CIV) pogled koji se fokusira na kontekst i zahtjeve sustava.
- *Pogled neovisan o platformi* (eng. Platform Independent Viewpoint - PIV) pogled koji se fokusira na mogućnosti realizacije sustava neovisno o određenoj platformi.
- *Pogled za određenu platformu* (eng. Platform Specific Viewpoint - PSV) pogled u kojem se model definiran u prošloj perspektivi proširuje detaljima realizacije na konkretnoj platformi.

Platforma. Predstavlja skup podsustava i tehnologije kojom se osigurava koherentni skup funkcionalnosti. Npr. operativni sustavi, programski jezici, sustavi za upravljanje bazama podataka, GUI i drugo.

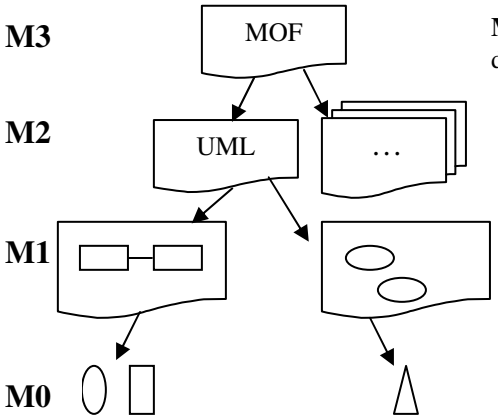
Neovisnost platforme. Predstavlja svojstvo kojim se modela može definirati neovisno o

svojstvima različitih platformi. Neovisnost predstavlja relativni indikator prilikom mjerenja razine apstrakcije kojim se jedna platforma odvaja od druge.

Transformacije modela. Predstavlja proces pretvorbe izvornog modela u ciljni model.

3.2.1.1.4. MDA i OMG standardi

Svi OMG-ovi standardi temelje se na zajedničkoj osnovi – četvero razinskoj arhitekturi prikazanoj na *slici 3-5*.



M3 razina sadrži meta-meta model→MOF. Tim se standardom definira struktura svih meta-modela.

M2 razina sadrži meta modele. Meta model definira strukturu skupa modela. UML i CWM su standardizirani meta modeli.

M1 razina sadrži modele. Njima opisujemo podatke. Primjeri modela su UML model analize, PIM ili PSM model.

M0 razina sadrži podatke koje je potrebno modelirati. Predstavljaju instance koncepata M1 razine.

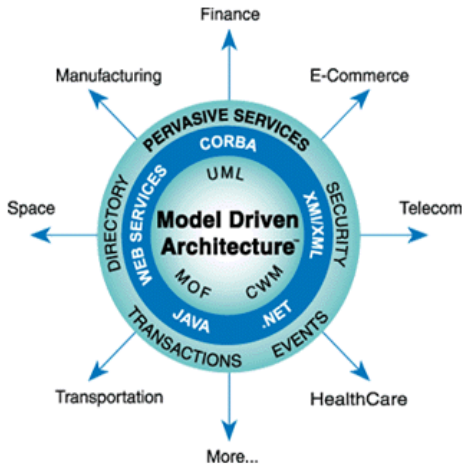
Slika 3-5. Razine arhitekture OMG standarda

Izvor: [OMG, 2002.]

Ključni čimbenik uspješne primjene MDA i upravljanja metapodacima kroz sve platforme, alate i baze podataka predstavljaju slijedeći OMG standardi:

- MOF (eng. Meta Object Facility),
- UML (eng. Unified Modeling Language),
- XMI (eng. XML Metadata Interchange).
- CWM (eng. Common Warehouse Model) i
- CORBA (eng. Common Object Request Broker Architecture).

Međusobna usklađenost MDA i navedenih standarda prikazana je na *slici 3-6.*



Slika 3-6. Prikaz MDA standarda

Izvor: [OMG, 2002.]

Ujedno, kao što je prikazano na slici 3-6., OMG je dodatno proširio te specifikacije kako bi ostvario podršku prema specifičnim granama industrije poput medicine, transporta, telekomunikacija, financija, e-poslovanja itd.

Slijedi kratko objašnjeni tih standarda temeljeno na dokumentaciji korporacije OMG [MDA Guide, 2003. str. 7-1.].

MOF (eng. Meta Object Facility) **-** OMG standard koji obuhvaća meta modeliranje - definira jednostavan, apstraktan jezik za specifikaciju svih metamodela. MOF je primjer za model metamodela odnosno tzv. meta metamodel. MOF je standard kojim se definiraju glavni objekti, sintaksa i struktura metamodela koji se koriste za izgradnju objektno orijentiranih modela. Prema MOF-u, model je samo instanca metamodela, a taj metamodel služi za opisivanje određene platforme.

CWM (eng. Common Warehouse Metamodel) definira metamodel koji predstavlja i poslovni i tehnički metamodel koji se nalaze u skladištu podataka i domenama poslovne analize. Predstavlja standard kojim se omogućuje jednostavna razmjena metapodatka iz skladišta podataka i metdapodatka poslovne inteligencije između alata i platforma skladišta podataka i repozitorija metapodataka skladišta podataka u distribuiranim heterogenim okruženjima. (npr. integrirajući skladišta podataka s poslovnom inteligencijom lanca nabave). Sustavi koji podržavaju CWM sustav međusobno razmjenjuju metapodatke u formatima koji su konzistentni s metamodelom. CWM se zapravo sastoji od više slojeva metamodela koji predstavljaju resurse podataka, analize, načine upravljanja skladištem podataka i sastavne komponente skladišta podataka. Metamodeli koji predstavljaju *resurse podataka* podržavaju i

strukturirane i nestrukturirane podatke kao što su npr. relacijske baze podataka, XML podaci, podaci temeljeni na objektima itd. *Sloj upravljanja skladištem podataka* također sadrži metamodele koji reprezentiraju standardne procese unutar skladišta podataka, omogućavaju praćenje tijeka aktivnosti i zadavanje zadataka koji će se izvršavati. *Sloj analize* definira metamodele koji su potrebni za transformaciju podataka, vizualizaciju odnosno izvješće o informacijama, poslovnu nomenklaturu te rudarenje podataka. *Osnovni sloj* odnosno sloj sa sastavnim komponentama omogućava specifikaciju različitih tipova podataka, apstraktnih ključeva i indeksa, različitih izraza, poslovnih informacija te pohranu softvera temeljenog na komponentama. CWM predstavlja pristup za razmjenu metapodataka između računalnih sustava, a koji je orijentiran na model. Skup svih metamodela unutar CWM-a je dovoljno velik da se iz njega modelira čitavo skladište podataka. Iz tog se modela pomoću nekih alata za CWM može izravno dobiti jedna instanca skladišta podataka pri čemu svaki od tih alata koristi one dijelove koji su njemu potrebni (npr. alat za relacijske baze podataka iskoristit će relacijski dio modela da iz njega napravi katalog za svoju bazu podataka).

UML (eng. Unified Modeling Language) **-** unificirani jezik za modeliranje. Pomoću vizualizacije modela primjenom UML-a smanjuje se kompleksnost struktura metamodela jer programer lakše razumije model prikazan u vizualnom obliku nego kad je on samo opisan riječima. O standardu je bilo govora i u točkama 2.2.2. i 2.2.3.

XMI (eng. XML Metadata Interchange) **–** standard za pohranu i razmjenu modela primjenom XML-a (eng. eXtensible Markup Language). XMI definira na koji način se koriste XML tagovi da se prikažu MOF modeli u XML jeziku. Metamodeli temeljeni na MOF-u se prevode u XML DTDs (eng. Data Type Definitions), a modeli se prevode u XML dokumente koji odgovaraju njima s pripadajućim definicijama tipa podataka. XMI rješava mnoge probleme povezane s pretvaranjem objekata i njihovih veza u jezik temeljen na tagovima. Osim toga XML dozvoljava da se metapodaci (izraženi pomoću tagova) i instance koje opisuju (sadržaj elemenata) zajednički upakiraju u jedan dokument čime se aplikacijama omogućava razumijevanje instanci preko njihovih metapodataka. To je vrlo korisna osobina posebno u distribuiranim, heterogenim sustavima. XMI služi razmjeni svih vrsta modela, nije ograničen samo na UML modele.

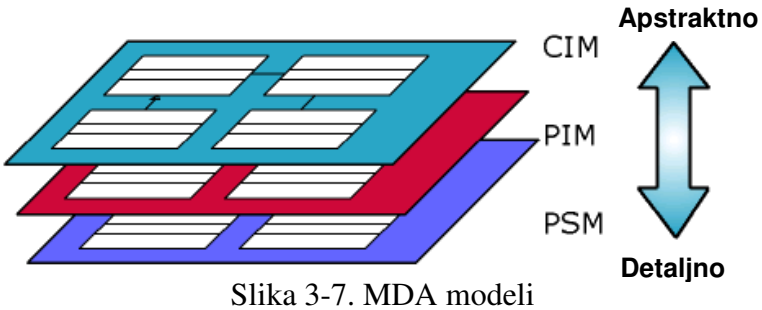
Prema [Riley, 2006., str. 2.] proizvođači MDA alata tvrde kako još uvijek u nekim dijelovima pojedinih standarda izostaju određeni detalji vezani uz njihovu implementaciju tako da na tržište dolaze alati koji evoluiraju tijekom primjene u samih projektima što uzrokuje dodatne zahtjeve i identificira nova ograničenja koja se manifestiraju proširenjima ili čak novim dijelovima standarda. Drugim riječima standardi potrebni za realizaciju MDA još

uvijek evoluiraju. Tako Schumacher [Riley, 2006., str. 2.] navodi kako je za uspješno provođenje MDA neophodno pet standarda, uz postojeća tri MOF, UML i XMI i još su dva, QVT (eng. Query, View, Transformation Specification) i M2T (eng. Model to Text Transformation Language) koja su u fazi prihvaćanja.

3.2.1.2. MDA modeli

Umjesto da se sustav opisuje samo jednim modelom, MDA prepoznaje potrebu definiranja modela na različitim razinama apstrakcije. Prema [MDA Guide, 2003., str. 2-5., 2-6.] svaka razina apstrakcije zapravo predstavlja jedan pogled (CIV, PIV i PSV pogledi) pa se u kontekstu svakog pogleda definira:

1. ICT neovisan model – CIM (eng. Computation Independent Model)
2. Model neovisan o platformi – PIM (eng. Platform Independent Model)
3. Model specifičan o platformi – PSM (eng. Platform Specific Model)



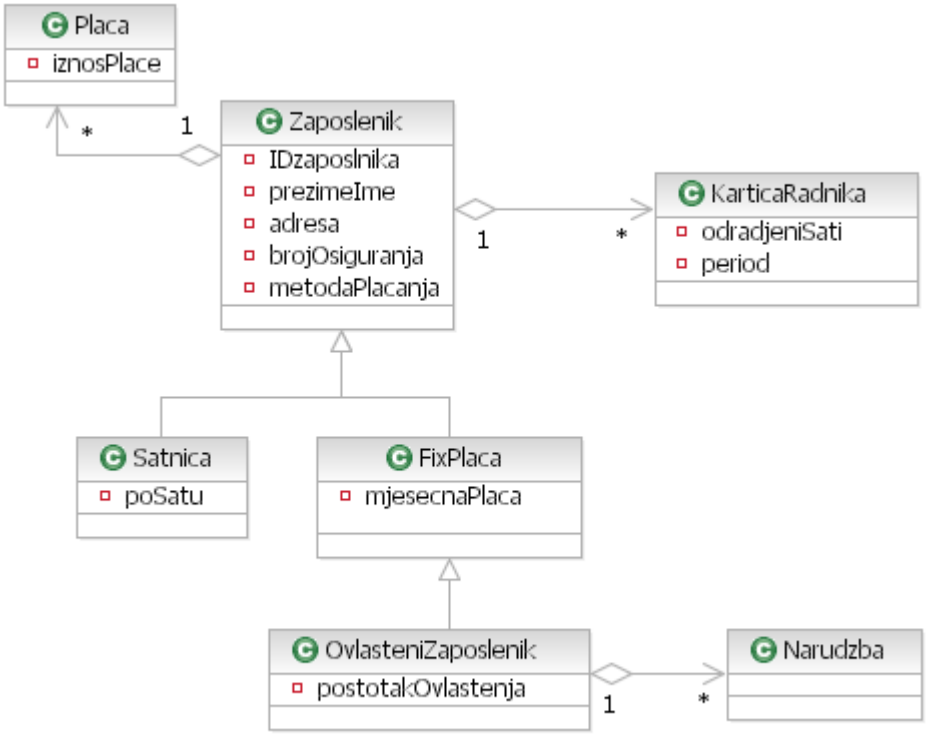
Slika 3-7. MDA modeli

Izvor: [MDA Guide, 2003., str. 2-5., 2-6.]

Ovi modeli zapravo bi se preciznije mogli opisati kao razine apstrakcije, jer se u okviru svake od njih, definira skup modela koji odgovaraju određenom pogledu na sustav.

3.2.1.2.1. ICT neovisan model – CIM

Često ga se još naziva i poslovni model ili model domene, a predstavlja model sustava s CIM pogleda [MDA Guide, 2003., str. 3-1.], [Gašević *et al.*, 2006., str. 111.]. Njegov je cilj analizirati i opisati poslovno okruženje u kojem će sustav djelovati. Prikazuje što bi poslovni sustav trebao raditi tj. što se od njega očekuje, a ne na koji način će biti implementiran [MDA Guide, 2003., str. 3-1.]. CIM modeli su najčešće izraženi u poslovnom jeziku odnosno jeziku specifičnom za domenu i ne prikazuju vezu s informacijskim sustavom. Primjer dijela CIM modela prikazan je na *slici 3-8*.

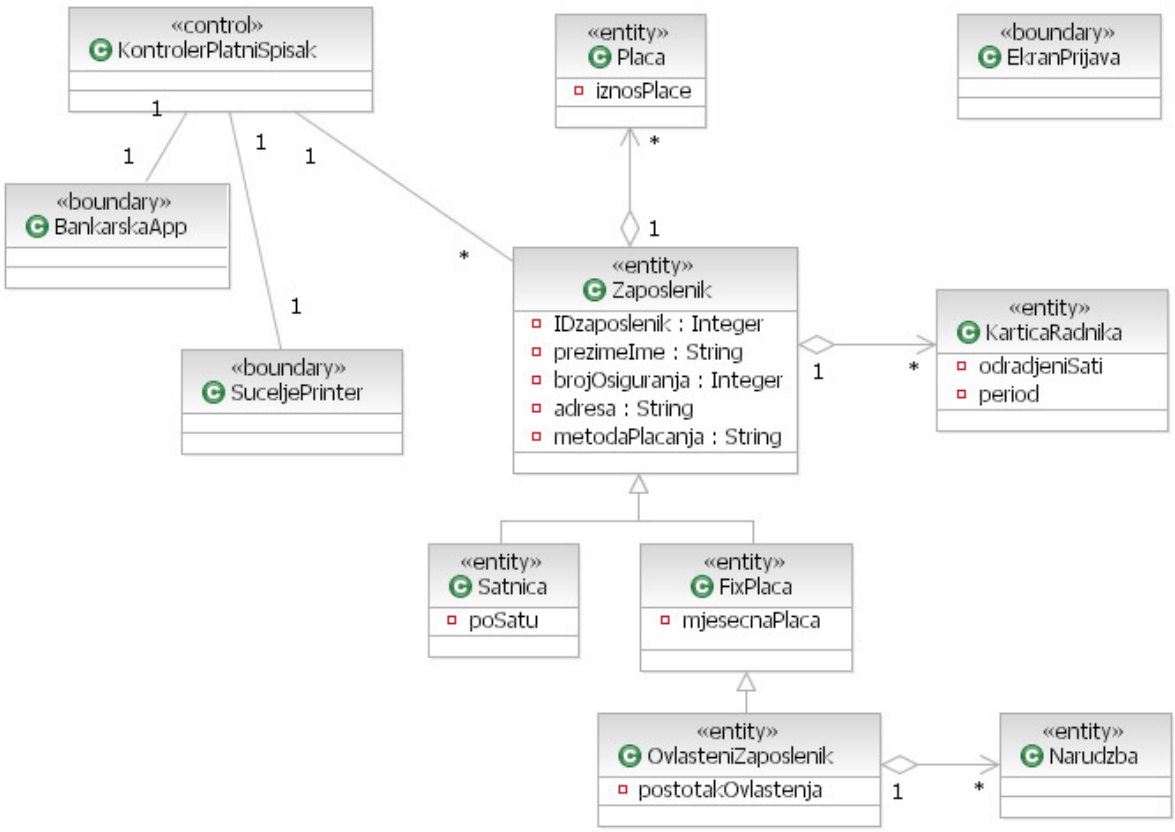


Slika 3-8. Primjer dijela CIM modela

Izvor: [DEV325]

3.2.1.2.2. Model neovisan o platformi – PIM

Predstavlja model sustava s PIM pogleda. Opisuje kako će biti izgrađen informacijski sustav (programsko rješenje) koji će pružati podršku poslovnom sustava ali bez osvrta na tehnologiju koja će biti upotrijebljena prilikom implementacije sustava [Gašević *et al.*, 2006., str. 111.]. Definiranjem PIM modela za neku domenu, arhitekt i dizajner sustava povećavaju razumijevanje domene kako bi definirali programsko rješenje koje će biti skalabilno i primjenjivo u drugim istim ili sličnim domenama. Model ne opisuje mehanizme potrebne da bi se izgradilo programsko rješenje za specifičnu platformu, ali mora biti definiran na način koji će omogućiti preslikavanje iz jedne platforme u drugu, što se često postiže definiranjem skupa servisa bez tehničkih detalja implementacije. Dolazi do izražaja kada se sustav želi implementirati na više različitih platformi. Primjer takvog modela prikazan je na *slici 3-9*.



Slika 3-9. Primjer PIM modela za CIM model

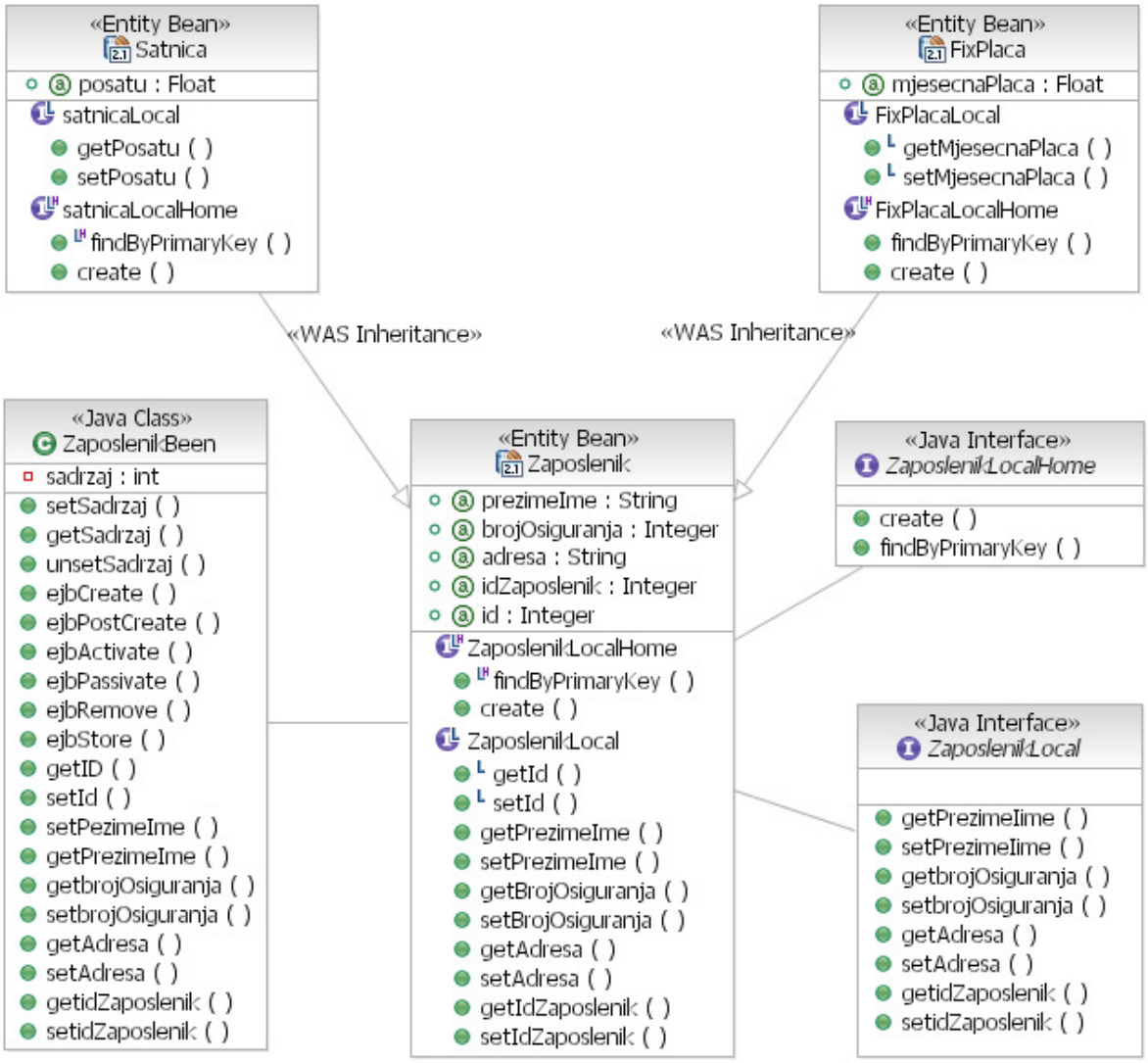
Izvor: [DEV325]

Uspoređujući PIM model s prethodno definiranim dijelom CIM modela vidi se da su:

- Atributi klasa poprimili vrijednosti tipova podataka.
- Definirani su tipovi klasa kako bi se vidjelo kako se uklapaju u programsko rješenje.
- Dodane su klase koje predstavljaju komponente programskog rješenja kao što su: sučelja za periferne jedinice, GUI ekrani i sl.

3.2.1.2.3. Model određene platforme – PSM

Predstavlja model sustava s PSM pogleda. Opisuje sustav odnosno programsko rješenje s gledišta specifične platforme (vidi definiciju *modela platforme*). U modelu se kombiniraju PIM specifikacije i detalji primjenjivosti sustava u odabranoj platformi kako bi se provela implementacija. Ukoliko PSM model ne uključuje sve potrebne informacije za implementaciju tada ga se smatra apstraktnim. Primjer takvog modela prikazan je na *slici 3-10*.



Slika 3-10. Primjer PSM modela za PIM model

Izvor: [DEV325]

Uz detalje koji su nadodani u PIM modelu, u PSM modelu jasno se raspoznaje konkretizacija rješenja odabirom konkretne platforme pa su tako naprimjer:

- objekti prikazani kao EJB i
- EJB koriste nasljeđivanje.

Kako u većini projekata ne postoje univerzalna rješenja tako i prilikom modeliranja u nekom projektu neće biti samo jedan CIM, jedan PIM i jedan PSM. Zapravo ne postoji pravilo koliki broj modela (CIM, PIM, PSM) je potreban na pojedinoj razini apstrakcije (pogledu).

3.2.1.3. Transformacije modela

Transformacije modela predstavljaju srce MDA pristupa. One se provode alatima za transformaciju (eng, transformation tools - TT) koji predstavljaju *plug in*-ove za pojedina razvojna okruženja (Za IBM RSA koristi se DPTK (eng. Design Pattern Toolkit) za v6.0 i JET 2 (Java Emitter Templates) za v7.0). Promatraju se kao crna kutija koja izvorni model transformira u ciljni model. Kada bi se zavirilo u TT mogli bi se identificirati elementi potrebni za realizaciju transformacije modela.

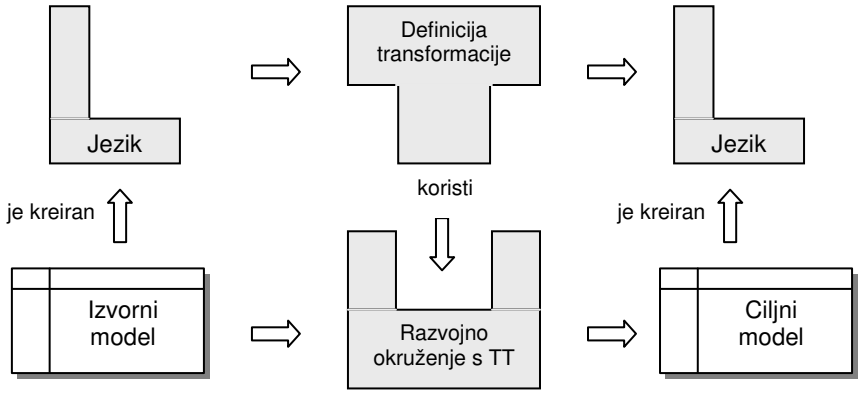
Osnovni pojmovi vezani uz transformaciju modela su [Kleppe *et al.*, 2003., pogl.2., str. 8.]:

Transformacija modela je proces pretvorbe izvornog modela (ulaz u proces) u ciljni modela (rezultat iz procesa) primjenom *definicija transformacija*.

Definicija transformacije (eng. mapping) je skup *pravila transformacija* koja zajedno jednoznačno opisuju kako se model opisan u izvornom jeziku modeliranja može transformirati u model opisan ciljnim jezikom. Naprimjer, može se napraviti definicija transformacije UML→C# kojoj će se odrediti za koje UML dijagrame se može generirati C# kod.

Pravilo transformacije (eng. mapping rules) predstavlja opis kako se jedan ili više koncepata iz izvornog jezika transformira u jedan ili više koncepata ciljanog jezika. Naprimjer: "Za svaku klasu nazvanu *ImeKlase* u izvornom modelu, u ciljnom modelu kreiraj klasu s nazivom *ImeKlase*".

Okruženje u kojem se odvijaju MDA transformacije modela prikazano je na *slici 3-11.*, a čine ga: modeli (CIM, PIM, PSM, kod), jezici za modeliranje (UML, QWT) i razvojno okruženje s TT.



Slika 3-11. Okruženje MDA transformacija modela

Izvor: [Kleppe *et al.*, 2003., pogl.2., str. 9]

U kontekstu MDA moguće su slijedeće transformacije modela [Miller, Mukerji, 2001., str. 13.]:

- CIM u PIM,
- PIM u PIM,
- PIM u PSM,
- PSM u PSM i
- PSM u programski kod.

Transformacije modela mogu se odvijati i u suprotnom smjeru pa se govori o:

- PIM u CIM,
- PSM u PIM i
- programski kod u PSM.

Primjeri nekih navedenih oblika transformacije biti će prikazani u kasnijim poglavljima (točka 5.3.2.5.2.).

Posebni oblici transformacija

Transformacija PIM u PIM: ova transformacija se koristi kada su modeli poboljšani, filtrirani ili specijalizirani tijekom razvojnog ciklusa sustava bez potrebe za podacima ovisnim o platformi. Jedna od najčešćih upotreba ove transformacije je prilikom dizajniranja transformacije modela odnosno izmjena modela.

Transformacija PSM u PSM: ova transformacija je potrebna za realizaciju komponenti i njihovu upotrebu. Te transformacije se koriste i prilikom izmjena tj. poboljšanja modela specifičnih za platformu.

Transformacija PSM u PIM: ova transformacija se koristi kada je potrebno postojeću implementaciju na nekoj platformi pretvoriti u PIM koji se zatim može koristiti da se opet izradi PSM za neku drugu platformu. Ovaj postupak je vrlo sličan postupku rudarenja podataka (eng. data mining) i teško ga je automatizirati iako se neki alati mogu iskoristiti kao pripomoć.

Iako su transformacije modela ključna komponenta MDA, mnogo toga još nije jednoznačno definirano. Primarno to se u kontekstu razmatranja transformacija modela odnosi na jezike za opis definicija transformacija i pravila koje one sadrže**.** Srž problema proizlazi iz činjenice da se trenutno definicije transformacija ne definiraju na standardiziran način

[DEV325]. OMG standard pod nazivom QVT (eng. Queries, Views, Transformations) jedan je od prijedloga načina definiranja definicija transformacija, a temelji se na MOF (eng. Meta Object Facility) standardu. Kao što je već napomenuto, standard je trenutno na usvajanju tako da postojeći MDA alati iskazuju manjkavosti u tom najkritičnijem segmentu. Kako bi pravila transformacije bila izvršiva moraju biti strojno čitljiva što zahtjeva formalizam za definiranje pravila transformacije u obliku pogodnom za strojno čitanje. Standardizacijom definicija transformacija koje bi grupa specijalista razvijala i održavala (nova uloga u razvojnom timu), omogućila bi znatno širu primjena te bi do izražaja dolazila ponovna iskoristivost.

3.2.1.3.1. Metode transformacije modela

Trenutno postoji više različitih načina provođenja transformacija modela. Razine primjene transformacija su vrlo različite, a kreću se od sasvim ručne do u potpunosti automatizirane.

Transformacije modela mogu se klasificirati u jednu od slijedećih metoda [DEV325], [MDA Guide, 2003., str. 4-1.- 4-3.], [Greenfield *et al.*, 2004., str. 146.]:

- ručno,
- automatizirano i
- polu-automatizirano.

3.2.1.3.1.1. Ručno provođenje transformacija

Ovaj način provođenja gotovo da se i ne razliku od razvoja SW koji se koristi već desetljećima. No, MDA pristup naglašava dva aspekta kojima se dodaje vrijednost dosadašnjem tradicionalnom razvoju:

- Eksplicitno razlikovanje modela koji nisu ovisni o platformi i transformiranih modela definiranih za odabranu platformu.
- Definiranje specifikacije kojom se osigurava sistematizirana transformacija između elemenata modela.

Iako danas postoje MDA alati koji interpretiraju izvorni model i na temelju *definicije transformacije* kreiraju ciljni model, taj se proces, u kontekstu MDA, može odvijati i ručno – razvojni inženjer čita izvorni model i primjenom *pravila transformacija* definiranih u *definiciji transformacije* stvara ciljni model.

3.2.1.3.1.2. Automatizirano provođenje transformacija

Postoje slučajevi (usko specijalizirani) u kojima su sve informacije potrebne za transformacije sadržane u PIM modelu te se procesom transformacije iz izvornog modela može dobiti izvršna verzija aplikacije. Za to je u kontekstu PIM modela potrebno specificirati strukturu, ponašanje - primjenom akcijskog jezika, kao i druge detalje (preduvjeti i posljedice te iznimke), kako bi model bio potpun, ali ujedno i neovisnim o platformi [DEV325], [MDA Guide, 2003., str. 4-3.]. U nekim slučajevima automatiziranom transformacijom nije potrebno kreirati PSM iako u svrhu dokumentiranja sustava daje određenu vrijednost.

3.2.1.3.1.3. Polu-automatizirano provođenje transformacija

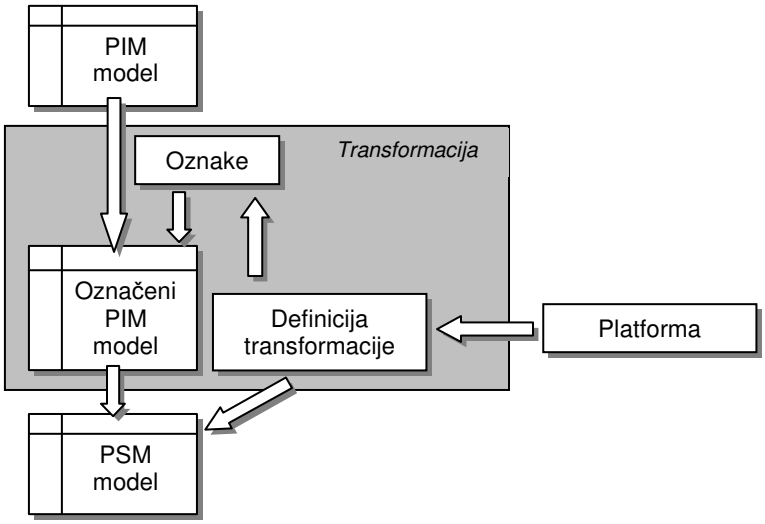
U slučajevima kada PIM model ne sadrži dovoljno detalja kako bi se provela automatizirana transformacija, potrebno je u razvojnom okruženju izvršiti niz ručnih koraka kako bi se dodale informacije i doprinijelo procesu transformacije koji se tada može pokrenuti. Jedan od načina kako se to može raditi je *označavanje* (eng. marks). Prema [MDA Guide, 2003., str. 3-3.] *označavanje* je koncept u PSM modelu (ne u PIM modelu) koji se primjenjuje na elementu PIM modela kako bi se definirale dodatne informacije potrebne za transformaciju tog elementa u PSM element. Arhitekt uzima gotov PIM model te ga *označava* (definira dodatne informacije karakteristične za određenu platformu) kako bi se tada mogao izraditi PSM.

Kada se nad PIM modelom vrši postupak *označavanja* tada do izražaja dolazi primjena [MDA Guide, 2003., str. 3-4.]:

- uzoraka,
- predložaka programskog koda,
- UML profila ili
- drugih resursa definiranih u repozitorijima.

3.2.1.3.2. Primjer primjene polu-automatiziranog provođenja transformacije modela

Kako bi se što jasnije prikazala jedna od mogućih transformacija modela u nastavku se prikazuje primjer, za sada, najčešćeg tipa provođenja transformacije. Konceptualni model prikazan je na *slici 3-12*. Odabrana je transformacija PIM modela u PSM model primjenom *označavanja*. Slike su prikazane u prethodnim odlomcima, a sada slijedi i detaljnije objašnjenje.



Slika 3-12. Primjer transformacije PIM modela u PSM model

Izvor: [DEV325]

PIM model:

Predstavlja grafički opis dijela aplikacije za obračun plaća. Model je prikazan *slikom 3-8*.

Označavanje: Označavanje koje se može primijeniti odnose se na dodjeljivanje tipova podataka za pojedine atribute i definiranje tipa klase. Pa tako klase mogu biti tipa: entitet, granična klasa i kontrolna klasa.

Definicija transformacije: Na temelju odabrane platforme i oznaka kreiraju se željene definicije transformacije. U primjeru, transformacija je definirana slijedećim *pravilima transformacije*:

1. Klasa koja je tipa entitet (stereotipizirana je oznakom "entity") treba se transformirati u *entity bean*.
2. Klasa koja je definiran kao kontrolna klasa (stereotipizirana je oznakom "control") treba se transformirati u *stateless session bean*.
3. Atributi stereotipizirana klase "entity" trebaju se transformirati u permanentna polja u *entity bean*.

PIM model s oznakama:

Slika 3-9. prikazuje dio PIM modela nakon što su nad njime primijenjene oznake.

PSM model:

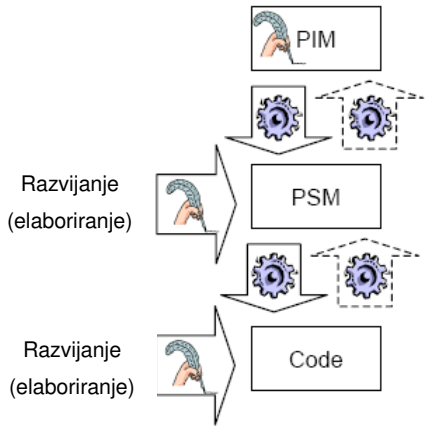
Rezultat transformacije je PSM model generiran na temelju prethodno definiranih elemenata. PSM model prikazan je na *slici 3-10*.

3.2.1.4. MDA interpretacije

S obzirom na metode transformacije modela, često se zapravo govori o dva oblika interpretacije MDA [Mc Neile, 2003., str. 2.]. Prva interpretacije naziva se *elaboracijski pristup* u kojoj se primjenjuje metoda polu-automatizirane transformacije modela. Zagovornik ove interpretacije (škole) je Kleppe o kojoj govori u svojoj knjizi [Kleppe *et al.*, 2003.]. Druga interpretacije naziva se *translacijski pristup* kojim se primjenjuje metoda automatizirane transformacije modela. Zagovornik ove interpretacije (škole) je Mellor o kojoj govori u svojoj knjizi [Mellor et.al., 2004.]. Translacijski pristup neki još nazivaju i izvršni UML ili skraćeno xUML (eng. Executable UML).

3.2.1.4.1. Elaboracijski pristup

Skica tog pristupa prikazana je na *slici 3-13*.



Slika 3-13. Elaboracijski pogled na MDA

Izvor: [Mc Neile, 2003., str. 2.]

Prema [Mc Neile, 2003., str. 2.] u elaboracijskom pogledu aplikacija se postepeno razvija: PIM - PSM - izvršni kod. Jednom kad se PIM izgradi, alat generira kostur odnosno prvu inačicu PSM-a koju programer odnosno osoba koja radi na razvoju aplikacije može elaborirati odnosno dodatno pojasniti i razviti dodavanjem dodatnih detalja ili informacija.

Jednako tako alat može generirati izvršni kod iz PSM modela te je taj kod moguće dodatno oplemeniti (doraditi). To dodatno oplemenjivanje PSM modela i izvršnog koda ovisi o samom alatu kojim je izvršena transformacija tj. da li dobiveno stanje predstavlja i željeno stanje.

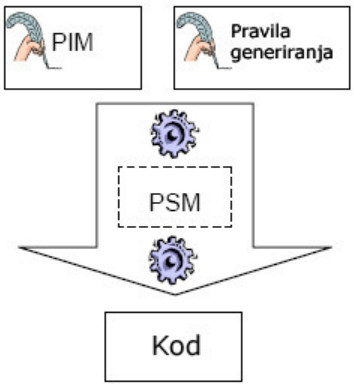
Kao rezultat te elaboracije, odnosno vlastitog razvijanja dolazi do razlikovanja modela na nižim razinama od modela na višim razinama. Zbog toga većina alata podržava mogućnost

obnove modela na višim razinama iz modela na nižim razinama. To je na *slici 3-13.* prikazano isprekidanim strelicama koje su usmjerene od modela niže razine prema modelima više razine. Taj princip gdje je moguće generirati modele prema nižim razinama, modificirati ih i ponovno sinkronizirati generiranjem prema višim razinama naziva se reverzibilno inženjerstvo (eng. round trip engineering).

Naravno u ovakvom se pogledu značaj stavlja i na razumijevanje modela od strane osobe koja se bavi razvojem jer njoj moraju biti poznati modeli da bi ih on uopće mogao dodatno elaborirati. Taj je pristup sličan tradicionalnom objektno orijentiranom pristupu (OOA, OOD, kod) s time da je on nadograđen automatizacijom modela primjenom alata.

3.2.1.4.2. Translacijski pristup

Translacijski pristup ili pogled prikazan je na *slici 3-14*.



Slika 3-14. Translacijski pogled na MDA

Izvor: [Mc Neile, 2003., str. 3.]

U ovom se pristupu, prema [Mc Neile, 2003., str. 3.], iz PIM modela direktno generira izvršni kod. To prevođenje ili translacija PIM modela u izvršni kod vrši se pomoću sofisticiranog generatora koda koji se često naziva *kompilator modela* i na *slici 3-14.* je prikazan pomoću velike strelice usmjerene prema dolje. On obavlja svoju funkciju pomoću pravila generiranja koja pobliže opisuju kako će elementi PIM modela u konačnom kodu biti prikazani.

PSM model je samo međufaza generiranja koda i nalazi se unutar generatora koda. Zbog toga što on u tom generatoru koda nije vidljiv ni promjenjiv za programera prikazan je isprekidanom crtom na *slici 3-14*.

Razlika između elaboracijskog pogleda i translacijskog pogleda je u tome što se u translacijskom pogledu ni PSM ni programski kod ne mijenjaju ručno [Mc Neile, 2003., str. 3.]. PIM i pravila generiranja predstavljaju sve resurse potrebne za generiranje čitavog sustava i nema potrebe za mijenjanjem dobivenih rezultata (kao i nakon kompilacije programskog koda). U ovom se pogledu ne upotrebljava reverzibilno inženjerstvo zbog toga jer se sve promjene rade u gornjem dijelu (dakle na PIM modelu i na pravilima generiranja) te nije potrebno usklađivati kod ili PSM sa PIM modelom [Mc Neile, 2003., str. 4.].

3.2.2. Tvornice softvera

Potpuno drugačiji koncept u odnosu na MDA, kojim se želi realizirati osnovne premise MDD (povećanje razine apstrakcije i automatizacija) je Microsoft-ov pristup poznat pod nazivom *tvornice softvera* (eng. Software Factories - SF).

3.2.2.1. Industrijski razvoj programskih proizvoda

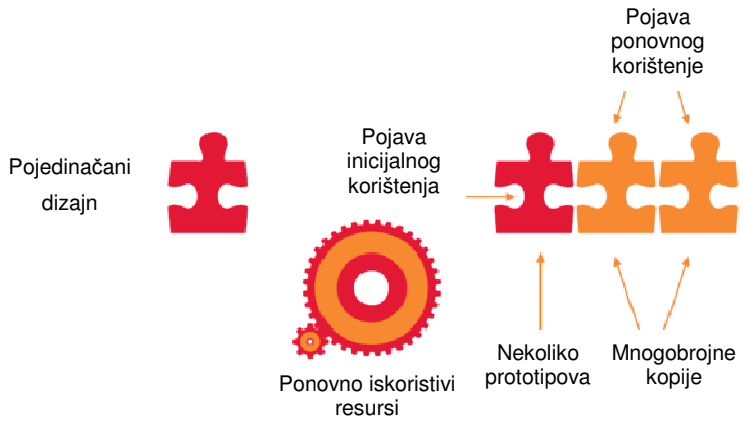
Realizacije SF temelji se na želji postizanja industrijskog razvoja programskih proizvoda (SW) po primjeru proizvodnih industrija (auto, tekstilna i druge). No je li moguće povući analogiju između razvoja programskog proizvoda i proizvodnje fizičkih dobara? Razlikuje li se razvoj programskog proizvoda od drugih industrija zbog prirode svojeg proizvoda - softvera? U [Greenfield, 2004., str. 4.] citira se Wegnera koji navodi sličnosti i razlike: *"Programski proizvodi su u nekim pogledima poput fizičkih dobara nastalih u drugim tradicionalnim industrijama. No također postoje i izvjesne značajne razlike koje razvoj programskog proizvoda čine jedinstvenim. Kako je programski proizvod više logička tvorevina nego fizička, troškovi su više usmjereni na razvoj nego na proizvodnju. Ujedno, kako se programski proizvod svojom primjenom ne troši, njegova pouzdanost ovisi o logičkim značajkama poput točnost i robusnost, a ne fizičkim kao što je čvrstoća ".* Kako bi se otklonilo miješanje krušaka i jabuka potrebno je razjasniti slijedeće vrlo bitne pojmove [Greenfield *et al.*, 2004., str. 156.], [Greenfield, 2004., str. 3.]:

Povećanje ponovnog korištenja (eng. economics of reuse). Kako bi se osigurao povrat investicija (eng. ROI), dodana vrijednost komponenata koje se koriste u razvoju programskog proizvoda mora biti veća od samo pokrivanja troškova njihova razvoja. Kako je trošak razvoja ponovno iskoristive komponente visok, prag profitabilnosti njezine ponovne iskoristivosti ne može se prepustiti slučaju. Stoga je ponovnom

korištenju potrebno pristupiti sistematski. To uključuje: identificiranje domena u kojima će se razvijati višestruki programski proizvodi, identificiranje problema koji se ponavljaju u kontekstu tih domena, razvoj ponovno iskoristivih resursa koji će predstavljati rješenje tih problema (npr. uzorci) te njihovu primjenu tijekom razvoja programskih proizvoda iz te domene [Greenfield *et al.*, 2004., str. 157.].

Povećanje broja istovrsnih proizvoda (EofSca - eng. Economics of Scale) **i povećanje raznovrsnosti proizvoda** (EofSco - eng. Economics of Scope)**.** Ovi pojmovi vrlo su značajni i dobro su poznati u drugim industrijama. Iako oba pojma, masovnom, a ne pojedinačnom proizvodnjom, ukazuju na smanjenje vremena i troškova te povećavanje kvalitete proizvoda, razlikuju se u načinu kako se te koristi ostvaruju.

Povećanje broja istovrsnih proizvoda: se pojavljuje kada se masovno proizvode višestruke identične instance jednog dizajna proizvoda (npr. proizvodnja vijaka). Dizajn proizvoda kreiran je prototipom (inicijalnom instancom) tijekom *procesa razvoja* od strane inženjera. Mnogobrojne instance proizvoda - kopije, proizvedene su drugim procesom – *proizvodnjom* od strane strojeva [Greenfield, 2004., str. 4.]. Model je prikazan na *slici 3-15*.



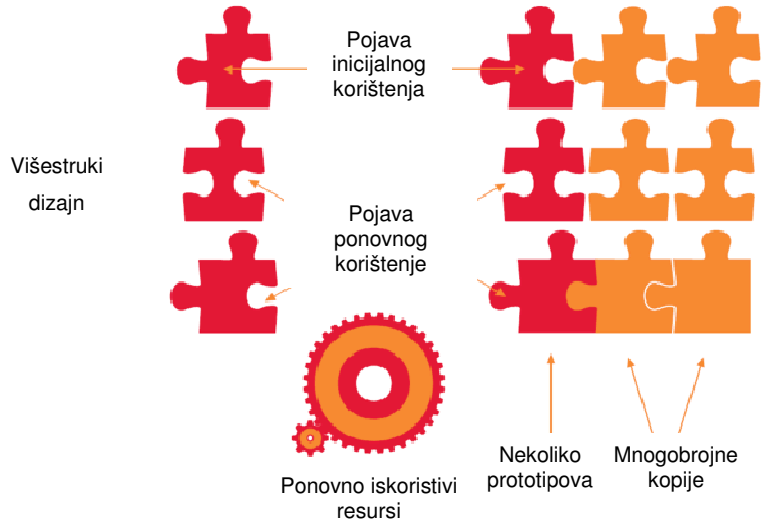
Slika 3-15. Povećanje broja istovrsnih proizvoda

Izvor: [Greenfield *et al.*, 2004., str. 158.]

Za rezultat proizvodnje se kaže da pokazuje *EofSca* ako se prosječni trošak po jedinici proizvodnje smanjuje s povećanjem proizvodnje jedinica [Maleković, 2007.]. *EofSca* najjednostavnije se može zamisliti kao smanje troškova rješavanja istog problema na isti način više puta, smanjenjem troškova proizvodnje svakog novog rješenja [Greenfield *et*

al., 2004., str. 157.].

Povećanje broja raznovrsnosti proizvoda: se pojavljuje kada se masovno proizvode višestruke slične instance različitih dizajna proizvoda i prototipova (npr. proizvodnja automobila). Ista praksa, procesi, alati i materijali služe za dizajn i izradu prototipova višestrukih sličnih ali zasebnih proizvoda [Greenfield, 2004., str. 5.]. Model je prikazan na *slici 3-16*.



Slika 3-16. Povećanje broja raznovrsnosti proizvoda

Izvor: [Greenfield *et al.*, 2004., str. 158.]

Za rezultat proizvodnje se kaže da ima *EofSco* ako je ukupan trošak proizvodnje više različitih proizvoda manji od sume troškova svakog proizvoda produciranih zasebno [Maleković, 2007.]. *EofSco* najjednostavnije se može zamisliti kao smanjenje troškova rješavanja višestrukih sličnih, ali zasebnih problema u nekoj domeni, skupnim rješavanjem njihovih zajedničkih pod problema te sastavljanje, prilagođavanje i konfiguriranje pojedinačnih rješenja u rješavanju cjelokupnog problema [Greenfield *et al.*, 2004., str. 159.].

Kao što se može zaključiti programski proizvod može se promatrati i kroz *EofSca* i *EofSco*. U [Greenfield, 2004., str. 5, 6.] se navodi kako u slučajevima (na tržištima) kada se masovno proizvodi softver za široku primjeni, npr. stolne aplikacije za obradu teksta/slika ili operativni sustavi, softver poprima *EofSca*.

U drugim slučajevima (na tržištima), kada se razvojem i primjenom programskog proizvoda želi ostvariti kompetitivna prednost u poslovanju poduzeća, te je svaki sustav jedinstven i ne rade se kopije, prisutna je *EofSco*. Upravo je to situacija prilikom mnogobrojnog razvoja programskih proizvoda.

Sada je vrlo jasno miješanje krušaka i jabuka u kontekstu industrijskog razvoja programskih proizvoda. Proces proizvodnje u industrijama fizičkih dobara naivno je preslikan i uspoređen s procesom razvoja programskih proizvoda. Softverska industrija ne može se uspoređivati s drugim industrijama po principu *EofSca*. S takvim razmišljanjem se slaže i Booch u [Booch, 2004.] kada citira Demarco-a koji u svojoj knjizi *Why Does Software Cost So Much?* kaže kako su metode koje se koriste u proizvodnji pogrešne, nerazumne i kontra produktivne budući da je razvoj programskih proizvoda R&D aktivnost, a ne aktivnost proizvodnje.

No industrijski razvoj softvera mogao bi se postići iskorištavanjem *EofSco* [Caprio, 2006., str. 3.]. Trenutno se ne može dati odgovor kako će konkretno izgledati industrijski razvoj softvera [Greenfield, Short, 2004. str. 1.], no jasno je da je prvi korak oblikovanje i "pakiranje" znanja u ponovno iskoristive resurse (eng. assets), a nakon toga njihova automatizacija. Stoga se SF promatra kao ključni element postizanja industrijskog razvoja programskih proizvoda [Greenfield *et al.*, 2004., str. 159.].

3.2.2.2. Definicija

Sažeta definicija autora ovog pristupa glasi:

SF predstavlja liniju proizvodnje softvera koja se sastoji od razvojnih alata specifičnih za pojedine domene, procesa i sadržaja (raznovrsnih artefakta) koristeći predložak tvornice softvera koji se temelji na shemi tvornice softvera u svrhu postizanja automatiziranog razvoja i održavanja različitih programskih proizvoda koji nastaju sastavljanjem, prilagodbom i konfiguriranjem već postojećih ponovno iskoristivih resursa (artefakata) [Greenfield *et al.*, 2004., str. 163.].

Pojednostavljeno, može se reći da svaka SF, predstavlja strukturirani skup različitih ponovno iskoristivih softverskih resursa, koji učahuruju znanje o poslovnoj domeni i arhitekturi programskih proizvoda, čija se primjena kombinira tijekom razvoja programskog proizvoda iz neke domene. Ujedno razvojnim inženjerima pruža podršku nizom smjernicama tijekom razvoja kao i automatizaciju (nekih) propisanih aktivnosti tijekom razvojnog ciklusa

programskog proizvoda [MSDN].

Sadržaj o kojem se govori u definiciji zapravo je skup ponovno iskoristivih programskih komponenti, predložaka, uzoraka, referentnih implementacija i sl. [MSDNTV]. *Razvojni alati* koji se koriste podržavaju čarobnjake, vizualno modeliranje te sadrže generatore koda [MSDN].

Ključna karakteristika SF je da analitičari, arhitekti i programeri mogu prilagoditi ili proširiti svaki resurs koji im stoji na raspolaganja prema jedinstvenim potrebama projekta [MSDN].

Do danas, poznate su slijedeće SF [MSDN] koje navodim u originalnom engleskom nazivlju:

- Mobile Client Software Factory,
- Web Client Software Factory,
- Web Service Software Factory i
- Application Block Software Factory.

3.2.2.3. Elementi i izgradnja tvornice softvera

Tri centralna elemenata koji čine SF su:

- shema tvornice softvera,
- predložak tvornice softvera i
- razvojno okruženje.

U nastavku slijedi njihovo detaljnije objašnjenje.

3.2.2.3.1. Shema tvornice softvera

Potreba za *shemom tvornice softvera* postala je očigledna kada je prepoznata potreba kategorizacije artefakata koji se korite u razvoju programskog proizvoda. Prema [Greenfield, Short, 2004. str. 1.] *shema tvornice softvera* može se usporediti s kulinarskim receptom. Ona sadrži popis potrebnih sastojaka – artefakta, poput XML dokumenata, modela, konfiguracijskih datoteka, programskog koda, SQL skripta kao i drugih ponovno iskoristivih resursa, uz smjernice, upute i objašnjenja kako se oni trebaju kombinirati kako bi nastao programski proizvod [MSDNTV]. Kako se SF temelji na DSL's, a ne na UML-u, u shemi se također specificira koji DSL's će se koristiti i kako će se modeli koji se temelje na tim DSL's

moći transformirati u programski kod ili druge artefakte. Ujedno opisuje se i arhitekturu sustava, povezanost svih komponenti koji će se koristiti kao i proces kojim je razvoj obuhvaćen.

Jedan od najznačajnijih aspekta svakako je konfiguriranje *sheme tvornice softvera* za potrebe konkretnog projekta budući da shema predstavlja "recept" razvoja za sve programske proizvode iz neke domene. Za usporedbu, konfiguriranje sheme jednako je prilagođavanju kulinarskog recepta zahtjevima gosta. To ukazuje da se *sheme tvornice softvera* mora sastojati od nepromjenjivih (ostaju uvijek identične za sve tipove SW u domeni) i varijabilnih (specifični za neke tipove SW iz domene) komponenata [Greenfield *et al.*, 2004., str. 172,173.].

3.2.2.3.2. Predložak tvornice softvera

U shemi tvornice softvera opisano je koji su sve artefakti potrebni u projektu razvoja programskog proizvoda. No u stvarnosti potrebno je posjedovati te artefakte. Drugim riječima potrebno je implementirati shemu tvornice softvera razvijajući DSLs, uzorke, razvojne okvire, primjere programskog koda, skripte i druge artefakte kako bi ih se moglo staviti na raspolaganje programerima za razvoj novog programskog proizvoda [Greenfield *et al.*, 2004., str. 172,173.], [MSDNTV]. Svi ti artefakti predstavljaju ponovno iskoristive resurse koji čine *predložak tvornice softvera*. Prema [Greenfield, Short, 2004. str. 1.] taj se predložak može usporediti sa vrećicom u kojoj se nalaze sve namirnice navedene u kulinarskom receptu tj. shemi tvornice softvera. Kao i shemu, i predložak tvornice softvera potrebno je konfigurirati kako bi bilo prikladan za potrebe konkretnog proizvoda.

3.2.2.3.3. Razvojno okruženje

Razvojno okruženje (eng. Integrated Development Environment - IDE) predstavlja okolinu u koju se učitava *predložak tvornice softvera* sa svim ponovno iskoristivim resursima koji se tada koriste u razvoju novog programskog proizvoda. Ako bi i dalje željeli povlačiti paralelu s kulinarstvom tada bi IDE bilo kao kuhinja u kojoj nastaje obrok [Greenfield, Short, 2004. str. 1.]. Kada se ono konfigurira s *predloškom tvornice softvera* tada IDE postaje **tvornice softvera** za sve programske proizvode iz neke domene.

Usporedba koja se provlači kroz ovu točku, a navode je autori SF-a u [Greenfield, Short, 2004. str. 1.], može se još upotpuniti, pa je tako novo razvijeni programski proizvod poput obroka u restoranu. Interesna skupina (eng. stakeholders) su gosti koji naručuju obrok (jelo) s

jelovnika. Specifikacija SW je poput narudžbe jednog gosta. Razvojni inženjeri su kao kuhari koji spremaju jelo koje je naručeno i preoblikuju ga prema potrebi gosta ili spremaju jela koja nisu na jelovniku, a želja su gosta. Voditelj projekta je poput šefa kuhinje koji definira jelovnik, sastojke koji će se upotrebljavati kao i oprema koja će se koristiti u njihovom spremanju.

Nakon ovako pojašnjenih elementa moguće je pristupiti izgradnji tvornice softvera koja uključuje [Greenfield *et al.*, 2004., str. 174,175.]:

- izgradnju sheme tvornice softvera i
- izgradnju predloška tvornice softvera.

Izgradnja sheme tvornice softvera opisuje samu tvornicu i obuhvaća dvije aktivnosti: analizu i dizajn. Analizom se definiraju koji programski proizvodi se mogu razvijati tom tvornicom. Određuje se opseg tvornice – problemska domena na koju se programski proizvod odnosi, kao i zahtjevi koji su organizirani u *poglede* koji su dio sheme tvornice softvera. Dizajnom se definira kako će se u SF razvijati programski proizvodi u okvirima definiranog opsega tj. arhitektura svih proizvoda iz te domene.
Izgradnju predloška tvornice softvera predstavlja implementaciju svih ponovno iskoristivih resursa koji će se koristiti u SF.

3.2.2.4. DSL – Jezici specifični za pojedine domene

U točci 2.2.3. spomenuto je da se, osim UML-a kao standardizirane notacije za modeliranje, mogu koristiti i neke druge notacije. Kako su u točci 3.1.3. istaknute mane koje donosi primjena UML, za koje se ujedno smatra da tako brzo neće ni biti riješene [Ambler, 2005., str. 1.], Microsoft je industriji nametno razmišljanje kako je tijekom modeliranja bolje imati različite jezike specifične za pojedine domene (DSL's) nego jedan unificirani, široko primjenjiv jezik – UML. Precizan jezik za modeliranje modela koji je nedvosmislen i precizan mora biti definiran za specifičnu namjenu želi li ga se iskoristiti za automatizaciju softverskih artefakata [Greenfield *et al.*, 2004., str. 142.].

Prema [Yusuf, Gardner, 2006., str. 3.] *DSL predstavlja jezik koji se koriti za modeliranje softvera, a razvijen je za određenu problemsku domenu.*

U [Caprio, 2006., str. 1.] *DSL* se definiraju kao *jezici usklađeni za specifične domene, omogućavajući bogatije i fleksibilnije razvojno okruženje od jezika opće namjene.*

Prema [Greenfield *et al.*, 2004., str. 254.] *DSL je jezik koji omogućava specificiranje programskog proizvoda sa specifičnog pogleda.*

DSL definira rječnik (koncepte) kojima se opisuje domena koja se promatra iz jednog pogleda i može biti vizualan ili tekstualan [Greenfield *et al.*, 2004., str. 142.]. Stoga je za opis sustava potrebno definirati višebrojne DSL jezike. Prema [MSDNTV] trenutno je razvijeno 4 DSL jezika koji se isporučuju s VSTS 2005 (eng. Visual Studio Team System) okruženjem (u alatu VS Team Architect).

Hoće li DSL jezici bi uspješni od UML-a? To pitanje za sada ostaje otvoreno. Ambler [Ambler, 2005.] navodi kako je vidljivo da UML ne odgovara potrebama razvoja temeljenog na modelima te predviđa rast popularnosti DSL jezika u slijedećih pet godina. Ujedno DSL vidi kao slijedeću stepenicu u životnom ciklusu tehnika modeliranja.

3.2.3. MDA vs SF i UML vs DSL

Kako su oba pristupa temeljena na modelima, oba u prvi plan stavljaju modele kao primarne artefakte koji bi trebali poslužiti za dobivanje programskog koda. No, kako modeli nisu prvorazredni artefakti, način realizacije te ideje znatno se razlikuje. Istaknut ću dva segmenta u kojem to dolazi do izražaja: *pogledi na sustav kroz modele* i *notacija*.

Daner u [Jacobs, 2006., str. 1.] iznosi svoje gledište kako je SF potpuniji (cjelovitiji) pristup od MDA, jer na prirodniji način uključuje automatizaciju u razvoj temeljen na modelima, te ujedno sadrži i smjernice *što* i *kako* učiniti kada se naiđe na dijelove razvoja u kojim je prikladniji tradicionalan (ručni) pristup.

Snaga SF proizlazi iz činjenice kako se tijekom razvoja, razvija skup pogleda na sustav prilagođenih problemskoj domeni za razliku od MDA koja striktno određuje tri pogleda (CIM, PIM i PSM) [Jacobs, 2006., str. 1.]. Greenfield u [Jacobs, 2006., str. 2.] naglašava kako su u praksi sustavi složeniji da bi samo ti pogledi bili dovoljni za jasno i nedvosmisleno razumijevanje sustava. Isto tako, MDA je "brand" koji zagovara primjenu OMG standarda.

I dok je MDA većim dijelom usmjerena samo na neovisnost modela o platformi i

portabilnost, a o ostalim, ne manje bitnim, ujedno složenim aspektima razvoja se govori u neznatnoj mjeri [Greenfield *et al.*, 2004., str. 569.], [Greenfield, Short, 2004. str. 1.]. Autori SF tvrde kako nije moguće razviti kompleksan SW za različite domene koji je neovisan o platformi, a da ujedno i obuhvaća važne komponente poput skalabilnosti i sigurnosti te stoga ne stavljaju naglasak na portabilnost i neovisnost platformi već SF fokusiraju na produktivnost - kako brže, s manjim troškovima i uz što manji rizik, razviti kvalitetan programski proizvod [MSDNTV], [Greenfield, Short, 2004. str. 1.].

I dok je MDA usmjeren na pojedinačni razvoj, SF nastoji kroz *sheme tvornice softvera* definirati problemske domene s ponovno iskoristivim artefaktima koji bi se mogli primjenjivati i u razvoju drugih (istih ili sličnih) programskih proizvoda iz iste domene.

Daljnja prepoznatljiva razlika ovih pristupa očituje se u notaciji kojom se modeliraju modeli. Jasno je da u MDD modeli trebaju biti strojno čitljivi te da modeli koje će procesirati računalo zahtijevaju formalni opis. Rasprava u pogledu primjene UML ili DSL jezika vrlo su česte. MDA kao notaciju koristi UML. No UML se, kao što je već i istaknuto u radu, primarno koristi za *skiciranje* te mnogi autori postavljaju pitanje prikladnosti njegove primijene za definiranje modela koji bi trebali biti strojno čitljivi i koji bi trebali poslužiti za generiranje programskog koda. Uz to, UML je razvijen kako bi se unificirao način dokumentiranja programskih proizvoda, a ne kao univerzalan jezik, te postoje domene u kojima njegova primjena nije prihvatljiva [MSDNTV]. Istina je da se UML modeli mogu proširiti s mehanizmima proširenja (stereotipovi i dodane vrijednosti), ali iskustva pokazuju da je potreban jezik s preciznijim svojstvima kako bi se pružila podrška automatiziranom razvoju.

S ovakvim se razmišljanjem ne slaže Booch koji kaže [Booch, 2004.] kako se UML dokazao korisnim tijekom razvoja te da postoji samo nekoliko situacija kada je manje prikladan. Navodi kako je njegova semantika prilično bliska onome što se zahtijeva iako je detaljniji nego što se od jezika zahtijeva. Booch smatra kako je u situacijama kada je potrebna detaljnija semantika dovoljna primjena UML profila, ali se ujedno slaže i s činjenicom da je u slučajevima kada je neke poslovne koncepte potrebno definirati u specijaliziranoj sintaksi opravdana primjena DSL jezika. S druge strane SF naglašava važnost primjene DSL jezika koji su usko specijalizirani za pojedine domene s dobro definiranom semantikom i mogućnošću implementacije u razvojnim alatima kako bi se razvili artefakti koji bi poslužili za dobivanje izvornog programskog koda. Naglasimo pritom da SF ne osporava vrijednost primjene UML tijekom razvoja SW, štoviše potiče njegovu primjenu, ali samo za

dokumentiranje [Greenfield *et al.*, 2004., str. 569.] međutim, za razvoj modela iz kojih će se generirati programski kod koriste se DSL jezici [Yusuf, Gardner, 2006., str. 5.].

Iz razmatranja oba pristupa realizaciji MDD paradigme, kao zajednička nit, može se prepoznati **potreba za stvaranjem i primjenom ponovno iskoristivih artefakata** kroz automatizirana razvojna okruženja primjenom vizualnog modeliranja. Nastavak rada bit će usmjeren na promatranje jednog tipa SW artefakta – **uzoraka** kao ponovno iskoristivih resursa u kontekstu MDD paradigme.

Stoga najprije slijedi detaljnije razmatranje uzoraka u razvoju programskih proizvoda, zatim razmatranje uzorka kao ponovno iskoristivog resursa u kontekstu MDD paradigme te kreiranje metodološkog okvira njegove primjene za MDD projekte te na kraju integracija razvijenog okvira s današnjim metodikama razvoja.

4. KONCEPT I TEORIJA UZORAKA U RAZVOJU PROGRAMSKOG PROIZVODA

U poglavlju se razmatraju najznačajniji aspekti vezani za pojavu i ulogu uzoraka u softverskoj industriji. Nakon kratke povijesti i definicije softverskog uzorka opisuju se karakteristike uzoraka, osnovna klasifikacija, njihova organizacija te sistematizacija trenutno pronađenih uzoraka primjenjivih u softverskoj industriji. U razmatranju uzoraka vrlo bitan element je njihova struktura pa se u dijelu poglavlja posvećuje pažnja i tome. Zaključak poglavlja razmatra mogućnosti standardizacije uzoraka kao nužnog preduvjeta kvalitetnije primjene.

4.1. Povijest nastanka uzoraka

Pojam *uzorak*, u svojim knjigama koje se bave urbanim planiranjem i arhitektonskim rješenjima, prvi spominje arhitekt Christopher Alexander*. Alexander kaže: *"svaki uzorak opisuje problem koji se stalno ponavlja u našoj okolini, a opisuje srž rješenja na način da se ono može koristiti n puta, a da se nikada ponovno ne izgrađuje"* [Larsen, Lane, 2006., str. 4.], [Kermek, 1998., str. 71].

Iako ideja za opis uzoraka dolazi iz arhitekture, ona je primjenjiva u mnogim drugim disciplinama pa tako i u razvoju programskog proizvoda. Temeljni blokovi gradnje u današnjem razvoju softvera uglavnom su moduli, funkcije, procedure ili potprogrami.

Uzorci u razvoju programskog proizvoda imaju kratku povijest. Pojavom objektno orijentirane paradigme, početkom 90-tih, uzorci su postali interesantno područje, jer se u njima vidjela mogućnost realizacije temeljne karakteristike te paradigme – ponovno korištenje (eng. reusability). Idejnim zagovornicima uzoraka smatraju se Erich Gamma, Richard Helm, Ralph Johnson, John Vlissedes (kasnije poznati i kao Gang of Four - GoF) koji su 1991. objavili knjigu: *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma *et al.*, 1997.] u kojoj su opisali 23 uzoraka. Time su nastali *uzroci dizajna* koji su u razvoju

* Christopher Alexander o konceptu uzorka govori u knjigama: *Notes on the Synthesis of Form,* Harvard University Press, 1964.; *The Oregon Experiment*, Oxford University Press, 1975.; *A Pattern Language: Towns, Buildings, Construction,* Oxford University Press, 1977.; *The Timeless Way of Building,* Oxford University Press, 1979.

programskog proizvoda trebali povećati produktivnost i uspješnost objektno orijentiranog razvoja. U tom se opisu uzoraka ne spominje njihova veza prema konkretnom programskom jeziku, ali se prema njihovom shvaćanju, uključivanjem konkretnog programskog koda u opis uzorka dobiva veća kvaliteta, te su stoga oni odabrali tada 2 najpopularnija jezika C++ i Smalltalk [Kermek, 1998., str. 73].

Od početka uvođenja u softverskoj industriji, razvoj, proučavanje i primjena *uzoraka oblikovanja* ima rastući trend. Njihova značajna primjena utjecala je na pridavanje sve veće pažnja razvoju uzoraka što je dovelo do stvaranja cijelog niza novih uzoraka u različitim segmenta razvoja programskih proizvoda. Njihova sistematizacija bit će prikazana u nastavku.

4.2. Definicija uzorka

Pojam *uzorak* (eng. pattern) može prizivati različita tumačenja i interpretacije najviše ovisno o kontekstu u kojem se primjenjuju. Stoga najprije slijede definicije nekih autora.

Prema [Yu, 2007.a, str. 1.], [Elssamadisy, 2006., str. 49.], [Larsen, 2006., str. 1.], [Swithinbank *et al.*, 2005. str. 7.] *uzorak opisuje određeni problem i njegovo rješenje koje se često ponavlja u nekom kontekstu.*

U [OMG, 2004., str. 2-3] *uzorak* se definira kao *koncept koji se koristi u razvoju softvera a predstavlja "know-how" za neki kontekst.*

Alexander, C. u svojoj knjizi *The Timeless Way of Building* [Appleton, 2000., str. 4.] opisuje *uzorak* kao *pravilo koje se sastoji od 3 dijela kojim se opisuje vezu između određenog konteksta, problema i rješenja.*

Coplien, J. u [Larsen, Lane, 2006., str. 4.] definira *uzorak* kao *ponavljajući strukturni oblik koji rješava problem u nekom kontekstu, pridonoseći sveobuhvatnosti neke cjeline ili sustava.*

Booch, G. [Larsen, Lane, 2006., str. 4.] kaže da *uzorak* pruža kvalitetno rješenje problema koji se najčešće pojavljuju u nekom kontekstu.

Kao što sve gornje definicije ističu, *uzorak* je usmjeren prema rješenju problema koji se često pojavljuje. No uzorak je i mnogo više. Kako se problem pojavljuje u određenom kontekstu i s mnogim pitanjima, on predstavlja dobro opisanu strukturu koja balansira ta pitanja na najprikladniji način. Primjenom forme uzorka, opis rješenja nastoji obuhvatiti sve

esencijalne čimbenike kako bi ono bilo primjenjivo u sličnim situacijama. Opis uzorka ujedno uključuje i objašnjenje zašto je to rješenje potrebno.

Preciznija definicija uzorka glasila bi:

Uzorak je imenovani grumen poučnih informacija koji sadrži neophodnu strukturu i uvid u uspješan i dokazan skup rješenja za problem koji se često pojavljuje u određenom kontekstu [Appleton, 2000., str. 3.].

Vrlo često se u razvoju softvera koristi pojam *uzorci oblikovanja,* a da se pri tome primarno ne misli samo na 23 uzorka oblikovanja koja su definirali GoF. Pojam se koristi kada se izravno želi ukazati na arhitekturu, oblikovanje ili implementaciju softvera. Stoga će se u radu koristiti pojam *softverski uzorak*.

Definicija *softverskog uzorka* koji će se koristiti u ovom radu glasi:

Softverski uzorak je strukturirani opis problema koji se često ponavlja u kontekstu razvoja softvera i njegovog rješenja koje predstavlja najbolju prasku, a može se primijeniti na svim razinama apstrakcije prilikom razvoja aplikacije.

U kontekstu razvoja programskog proizvoda, *softverski uzorak* predstavlja element ponovnog korištenja na razini apstrakcije višoj od linija koda, individualnih klasa i komponenata.

Kako definicija spominje pojavu *uzoraka* na različitim razinama apstrakcije te razina apstrakcije su: poslovni procesi, arhitektura, oblikovanje, programiranje i isporuka. Na taj je način, u svim fazama razvojnog ciklusa programskog proizvoda, moguće kreirati artefakte koji će predstavljati skup najboljih praksa.

4.3. Karakteristike i prednosti primjene uzoraka

Nije svako rješenje, algoritam, najbolja praksa ili načelo, *uzorak*. Iako je moguće da ono ima sve sastavne dijelove pravila* (kontekst, problem i rješenje) ne bi se smio smatrati *uzorkom,* ako nije verificiran kao "fenomen" koji se ponavlja tj. rješenje problema mora biti primjenjivo u različitim situacijama. Ujedno, prema [Appleton, 2000., str. 5.] uzorak mora

* U nekoj literaturi ova vrsta pravila se još naziva i trojno pravilo (eng. rule of three).

uključivati i proces i artefakte koji se kreiraju.

Mogućnost *ponavljanja* nije jedina karakteristika uzorka. Uzorak mora *odgovarati problemu* i mora biti *upotrebljiv* za kontekst za koji je namijenjen. I dok je mogućnost *ponavljanja,* kvantitativna karakteristika, koja se prikazuje korištenjem trojnog pravila, *prikladnost* i *upotrebljivost* su kvalitativne karakteristike pri čemu *prikladnost* objašnjava *kako* će *uzorak* doprinijeti rješenju problema, dok *upotrebljivost* objašnjava *zašto* će *uzorak* biti uspješan i koristan.

Prema [Appleton, 2000., str. 6.] dobar uzorak je onaj:

- koji rješava problem,
- koji predstavlja već dokazan koncept,
- čije rješenje nije očigledno i
- koji opisuje veze s ostalim elementima.

Uzorci predstavljaju moćan način poboljšanja razvoja programskog proizvoda identificirajući najbolju praksu i oblikujući ekspertizu na način prikladan za integraciju u alate kojima se tada omogućava ponovno korištenje.

Prema [IBM developerWorks, 2007.] **prednosti** koje donosi primjena uzoraka u razvoju programskog proizvoda su:

- unapređenje produktivnosti,
- smanjenje vremena potrebnog za razvoj,
- smanjenje kompleksnosti,
- bolja agilnost i upravljanje promjenama,
- razvoj novih IT vještina,
- promocija otvorenih standarda,
- smanjenje jaza između poslovnih procesa i IT-a i
- smanjenje ukupnih troškova razvoja.

Uz ove, u [RSA 6.0.1.] navode se i slijedeće prednosti:

- uzorci učahuruju znanje i vrijeme eksperta,
- korisnik uzoraka ne treba poznavati kako su oni kreirani već mu je samo potrebna dobra opisana specifikacija uzorka,
- kreiranje i primjena uzoraka promiče primjenu ponovnog korištenja i

- osiguravaju djeljivost ne samo u okviru projekta neko i kompanije čak i između kompanija.

Autori [Ackerman, Gonzalez, 2007., str. 3.] navode još dvije prednosti:
- unapređenja u upravljanju razvojem programskih proizvoda i
- pojednostavljenje arhitekture softvera.

Navedene karakteristike uzoraka kao i prednosti njihove primjene ukazuju na njihovu vrijednost u SW industriji.

4.4. Osnovna klasifikacija softverskih uzoraka

Kako se broj uzorka koji nastaju u SW industriji eksponencijalno povećava, javlja se potreba definiranja osnovne klasifikacije uzoraka koja će pridonijeti njihovom kvalitetnijem razvoju, bržem pronalaženju i primjeni.

Prema istraženoj literaturi *softverski uzorci* najčešće su klasificirani prema:
- razini apstrakcije i
- stupnju općenitosti.

Prema *razini apstrakcije* [Yu, 2007.a, str. 3.], [Swithinbank *et al.*, 2005., str. 31.], [Rosengard, Ursu, 2004., str. 1.] i [Appleton, 2000., str. 8.] uzorke klasificiraju na:
- arhitektonske uzorke,
- uzorke oblikovanja i
- uzorke implementacije – idiome.

Arhitektonski uzorci: opisuju osnovne strukture organizacije za koju se razvija softver. U njima je rješenje opisano pre-definiranim skupom podsustava, specificiranim odgovornostima, kao i pravilima i smjernicama za njihovo povezivanje. Čine ih velike komponente, globalna svojstva i mehanizmi sustava. (primjer: ISO OSI referencijalni model i Model-Viev-Controler).

Prema [Kardell, str. 8.] svrha arhitektonskih uzoraka je definirati strukturu koja će:

- se sastojati od prikladnih podsustava kojim će se riješiti problem,
- pojasniti organizaciju podsustava,
- omogućiti provjeravanja i analizu,
- pomoći u uspostavljanju održive unutrašnje konzistentnosti i
- sačuvati informacije o strukturi sustava tijekom kasnijih zadataka održavanja.

Uzorci oblikovanja: Najpoznatiji i najznačajniji uzorci u softverskoj industriji svakako su objektno orijentirani *uzorci oblikovanja* koje su definirali GoF. Opisuju najčešće pojavljivanu strukturu softverskih komponenata koje rješavaju opće probleme oblikovanja unutar nekog konteksta. Definiraju mirko-arhitekture podsustava i komponenata. No uz ove i mnogobrojni drugi aspekti koji se pojavljuju tijekom oblikovanja obuhvaćeni su uzorcima koji se mogu ubrojiti u ovu kategoriju.

Uzorci implementacije – idiomi: Predstavljaju uzorke na najnižoj razini apstrakcije specifične za neki programski jezik. Opisuju kako implementirati pojedine aspekte komponenata i njihovu povezanost primjenom programskog jezika.
Prema [Kardell, str. 9.] svrha idioma je:

- prikazati pogodne načine kombinirana osnovnih koncepata programskog jezika,
- oblikovati osnovnu standardiziranu strukturu izvornog koda i
- izbjeći probleme prilikom primjene programskih jezika.

Prema *stupnju općenitosti* [Rosengard, Ursu, 2004., str. 2.] i [Kardell, str. 7.] uzorke klasificira na:

- uzorke ovisne o aplikacijskoj domeni i
- opće uzorke.

Uzorci ovisni o aplikacijskoj domeni: Kako postoje različite aplikacijske domene jasno je da postoje različiti tipovi softvera, a samim time i različiti tipovi uzoraka, od kojih su neki u pojedinim domenama prikladniji od drugih. Takvi uzorci potječu iz: sustava u stvarnom vremenu (eng. real-time), komunikacijskih sustava ili specifičnih aplikacija kao što su kompajleri ili baze podataka.

Opći uzorci: Nisu vezani za određenu domenu i mogu se primjenjivati neovisno o njoj.

Uz ove klasifikacije smatram da bi softverske uzorke bilo prikladno klasificirati i prema:

- fazama razvoja programskog proizvoda,
- aspektu razvoja
  - *baza podataka:* uključivala bi uzorke vezane uz logičke i fizičke sheme, tablice, veze, transakcije i sl. komponente.
  - *aplikacija:* uključivala bi uzorke vezane uz način realizacije rješenja aplikacije (npr. modeli domene, dijagrami klasa, i sl.)
  - *isporuka:* uključivala bi uzorke kojima bi se aplikacija iz razvojnog okruženja postavila u radno okruženje.
  - *infrastruktura:* uključivala bi uzorke vezane uz hardver i mrežnu opremu potrebnu za uspješan rad nove aplikacije.
- proizvođaču i/ili tehnološkoj platformi.

4.5. Organizacija uzoraka – katalozi i repozitoriji uzoraka

Razvijeni uzorci grupiraju se u *kataloge uzoraka* koji se smještavaju u *repozitorij uzoraka* kako bi tijekom razvoja bili na raspolaganju. Prethodno navedeni oblici klasifikacije ujedno mogu poslužiti za organizaciju tih kataloga. I dok je uobičajeno da uzorci u katalogu budu klasificirani u jednoj dimenziji, zbog sve većeg broja uzorka kao i još učinkovitije klasifikacije (te kasnijeg pretraživanja) katalog se može organizirati u dvije dimenzije tvoreći mrežu (eng. grid) u koju se svaki uzorak može smjestiti te ujedno i definirati njegova povezanost s ostalim uzorcima. Primjer jednog takvog dvo-dimenzionalnog kataloga tj. njegove mreže mogle bi činiti dimenzije: razina apstrakcije i aspekt razvoja.

4.6. Sistematizacija trenutno pronađenih kataloga softverskih uzoraka

Iako se u posljednjim godinama broj uzoraka rapidno povećava, inicijativa za njihovom sistematizacijom ne postoji. Stoga u nastavku slijedi vlastita sistematizacija temeljena na trenutno pronađenim katalozima objavljenim u istraženim publikacijama i na Internetu.

Nazivi kataloga kao i pripadajući uzorci navedeni se u izvornom (engleskom) nazivu.

Tablica 4-1. Sistematizirani pregled kataloga softverskih uzoraka

Naziv Kataloga:	**Design Patterns (Gang Of Four) [23 uzorka]**	**Izvor:** [Gamma *et al.*, 1997.]
Opis:	Uzorci primjenjivi u oblikovanju sustava.	
Klasifikacija:	**Naziv uzoraka:**	
Creational Patterns	Abstract Factory, Builder, Factory Method, Prototype, Singleton.	
Structural Patterns	Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy.	
Behavioral Patterns	Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor.	

Naziv Kataloga:	**Analysis Patterns [>70 uzorka]**	**Izvor:** [Fowler, 1997.]
Opis:	U katalogu su sistematizirani uzorci koji su primjenjivi u različitim poslovnim domenama (zdravstvo, financijsko poslovanje, računovodstvo). Svaki je uzorak opisan tekstualno uz dijagram.	
Klasifikacija:	**Naziv uzoraka:**	
Accountability	party, organisational hierarchies, organisational structure, accountability, accountability knowledge levels, party type generalizations, hierarchic accountability, operating scope, post	
Observations and Measurements	quantity, conversion ratio, compound units, measurement, observation, subtyping observation concepts, protocol, dual time record, rejected observations, active observations & hyupothesis& projection, associated observation process of observation	
Observations for Corporate Finance	enterprise segment, measurement protocol, range, phenomenon with range	
Referring to objects	name, identification scheme, object merge, object equivalence	
Inventory and Accounting	account, transaction, summary account, memo account, posting rules, individual instance method, posting rule execution, choosing entries, accounting practice, sources of entry, balance sheet & income statement, corresponding account, specialized account model, booking entries to mulitple accounts	
Planning	Proposed and implemented actions,complete and abandoned action, plan, protocol, resource alocation, outcome and start actions	
Trading	contract, portfolio, portfolio, quote, scenario	
Derivative Contracts	forward contract, option, product, subtype state machines, parallel aplication and domain hierarchies	
Trading Packages	multiple access levels to a package, mutual visibility, subtyping packages	
Layered Architectures for Information Systems	two-tier, three-tier, presenttaion and application logic, database interaction,	
Application Facades		
Patterns for Type Model Design Templates	implementing association, implementing generalization, object creation, obcjet destruction, entry point, implementing constraints, design templates for other techniques	
Association Patterns	accosiative type, keyed mapping, historic mapping	

Naziv Kataloga:	**Pattern Oriented Software Architecture (POSA) [47 uzorka]**	
Opis:	Prvi dio POSA kataloga obuhvaća širok spektar uzoraka opće namjene (GRASP- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns), drugi je usmjeren na osnovne uzorke namijenjene razvoju mrežnih aplikacija dok treći sadrži uzorke kojima se opisuje efikasno upravljanje SW resursima. Definirani su i dijelovi POSA4 [Buschmann, 2007.] i POSA5 [Buschmann, 2007.a] u kojima je opisan *jezik uzorka* i njegova primjena za distribuirano okruženje.	
Klasifikacija:	**Naziv uzoraka:**	
DIO I: POSA	**GRASP [19 uzorka]**	**Izvor:** [Buschmann *et al*., 1996.]
Architectural Patterns	Layers, Pipes and Filters, Blackboard, Broker, Model-View-Controller, Presentation-Abstraction-Control, Microkernel, Reflection	
Design Patterns	Whole-Part, Master-Slave, Proxy, Command Processor, View Handler, Forwarder-Receiver, Client-Dispatcher-Server, Publisher-Subscriber	
Idiom (C++)	Counted Pointer	
DIO II: POSA2	**Patterns for Concurrent and Networked Objects [18 uzorka]**	**Izvor:** [Schmidt *et al*., 2001.]
Service Access Patterns	Wrapper Facade, Extension Interface, Interceptor	
Event Handling Patterns	Reactor, Proactor, Asynchronous Completion Token	
Initialization Patterns	Acceptor-Connector, Activator, Component, Configurator,	
Synchronization Patterns	Scoped Locking, Strategized Locking, Thread-Safe Interface, Double Checked Locking Optimization	
Concurrency Patterns	Active Object, Monitor Object, Leader/Followers, Half Sync/Half-Async, Thread-Specific Storage	
DIO III: POSA3	**Patterns for Resource Management [10 uzorka]**	**Izvor:** [Kircher, Jain, 2004.]
Resource Acquisition	Lookup, Lazy Acquisition, Eager Acquisition, Partial Acquisition	
Resource Lifecycle	Caching, Pooling, Coordinator, Resource Lifecycle Manager,	
Resource Release	Leasing, Evictor	

Naziv Kataloga:	**Patterns of Enterprise Application Architecture (P of EAA) [25 uzorka]**	**Izvor:** [Fowler, 2003.]
Opis:	Uzorci su klasificirani prema aspektu sustava koji se razvija, koji su definirani prema slojevima arhitekture programskog sustava.	
Klasifikacija:	**Naziv uzoraka:**	
Domain Logic Patterns	Transaction Script, Domain Model, Table Module, Service Layer	
Data Source Architectural Patterns	Table Data Gateway, Row Data Gateway, Active Record, Data Mapper	
Object-Relational Behavioral Patterns	Unit of Work, Identity Map, Lazy Load	

Object-Relational Structural Patterns	Identity Field, Foreign Key Mapping, Association Table Mapping, Dependent Mapping, Embedded Value, Serialized LOB, Single Table Inheritance, Class Table Inheritance, Concrete Table Inheritance, Inheritance Mappers
Object-Relational Metadata Mapping Patterns	Metadata Mapping, Query Object , Repository
Web Presentation Patterns	Model View Controller, Page Controller, Front Controller, Template View, Transform View, Two-Step View, Application Controller
Distribution Patterns	Remote Facade, Data Transfer Object
Offline Concurrency Patterns	Optimistic Offline Lock, Pessimistic Offline Lock, Coarse Grained Lock, Implicit Lock
Session State Patterns	Client Session State, Server Session State, Database Session State
Base Patterns	Gateway, Mapper, Layer Supertype, Separated Interface, Registry, Value Object, Money, Special Case, Plugin, Service Stub, Record Set

Naziv Kataloga:	**Enterprise Integration Patterns [65 uzorka]**	**Izvor:** [Hohpe, Woolf, 2004.]
Opis:	Ovaj katalog definiran je kako bi se olakšalo oblikovanje i implementacija integracije rješenja. Opisani uzorci relevantni su za razvojne alate namijenjene integraciji i platformama kao što su: IBM WebSphere MQ, TIBCO, Vitria, SeeBeyond, WebMethods, BizTalk, JMS,WCF, MSMQ, ESB's poput Sonic, Fiorano, ili Mule, kao i SOA i Web-service rješenja.	
Klasifikacija:	**Naziv uzoraka:**	
Integration tyles	File Transfer, Shared Database, Remote Procedure Invocation, Messaging	
Messaging Systems	Message Channel, Message, Pipes and Filters, Message Router, Message Translator, Message Endpoint	
Messaging Channels	Point-to-Point Channel, Publish-Subscribe Channel, Datatype Channel, Invalid Message Channel, Dead Letter Channel, Guaranteed Delivery, Channel Adapter, Messaging Bridge, Message Bus	
Message Construction	Command Message, Document Message, Event Message, Request-Reply, Return Address, Correlation Identifier, Message Sequence, Message Expiration, Format Indicator	
Message Routing	Content-Based Router, Message Filter, Dynamic Router, Recipient List, Splitter, Aggregator, Resequencer, Composed Message Processor, Scatter-Gather, Routing Slip, Process Manager, Message Broker	
Message Transformation	Envelope Wrapper, Content Enricher, Content Filter, Claim Check, Normalizer, Canonical Data Model	
Messaging Endpoints	Messaging Gateway, Messaging Mapper, Transactional Client, Polling Consumer, Event-Driven Consumer, Competing Consumers, Message Dispatcher, Selective Consumer, Durable Subscriber, Idempotent Receiver, Service Activator	
System Management	Control Bus, Detour, Wire Tap, Message History, Message Store, Smart Proxy, Test Message, Channel Purger	

Naziv Kataloga:	**Data Model Patterns**	**Izvor:** [Hay, 2006.]
Opis:	Predstavlja katalog uzoraka namijenjen modeliranju podataka. Uzorci su razvijeni na konceptualnoj razini tako da se uz modeliranje podataka mogu koristiti i prilikom objektnog modeliranja.	

Naziv Kataloga:	**Process Patterns [20 uzorka]**	**Izvor:** [Havey, 2005.]
Opis:	Po uzoru na *uzorke oblikovanja* u kontekstu modeliranja poslovnih procesa (eng. BPM) primijenjen je sličan pristup te je definiran skup *uzoraka procesa*. Autori tih uzoraka, [Havey, 2005., str. 75.], Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski i Alistair Barros (koji se nazivaju Process Four – P4) 2003. su, objavili članak pod nazivom: *Workflow Patterns* u kojem definiraju 20 uzoraka karakterističnih za procese (točnije fokus je stavljen na jedan aspekt procesa a to je *kontrolni tijek*) klasificirani u 6 glavnih kategorija.	
Klasifikacija:	**Naziv uzoraka:**	
Basic patterns	Sequence, Parallel Split, Synchronization, Exclusive Choice, Simple Merge	
Advanced branch and join pattrens	Multiple Choice, Synchronizing Merge, Multiple Merge, Discriminator, N-out-of-M Join	
Structural pattrens	Arbitrary Cycles, Implicit Termination	
Multiple instances pattrens	MI without synchronization, MI with a priori known design time knowledge, MI with a priori known runtime knowledge, MI with no a priori runtime knowledge.	
State-based pattrens	Deferred Choice, Interleaved Parallel Routing, Milestone	
Cancellation pattrens	Cancel Activity, Cancel Case	

Naziv Kataloga:	**Core J2EE Patterns [25 uzorka]**	**Izvor:** [Alur *et al.*, 2003.]
Opis:	Katalog opisuje Enterprise Application Architecture uzorke u kontekstu Java J2EE platforme. Autori ujedno navode da iako su uzorci primarno usmjereni na J2EE platformu jednako se tako mogu primjenjivati i u konteksti drugih platformi.	
Klasifikacija:	**Naziv uzoraka:**	
Presentation Tier	Intercepting Filter, Context Object, Front Controller, Application Controller, View Helper, Composite View, Dispatcher View, Service To Worker	
Business Tier	Business Delegate, Service Locator, Session Facade, Application Service, Business Object, Composite Entity, Transfer Object, T O Assembler, Value List Handler	
Integration Tier	Data Access Object, Service Activator, Domain Store, Web Service Broker	

Naziv Kataloga:	**Java Enterprise Desing Patterns [134 uzorka]**	
Opis:	Katalog od 3 dijela u kojem su opisani uzorci oblikovanja vezani uz Javu platformu.	
Klasifikacija:	**Naziv uzoraka:**	
DIO I	**[47 uzorka]**	**Izvor:** [Grand, 2002.]
Fundamental Design Patterns	Abstract Superclass, Delegation, Immutable, Interface, Interface and Abstract Class, Marker Interface, Proxy	
Creational Patterns	Abstract Factory, Builder, Factory Method, Object Pool, Prototype, Singleton	
Partitioning Patterns	Composite, Filter, Read-Only Interface	
Structural Patterns	Adapter, Bridge, Cache Management, Decorator, Dynamic Linkage, Façade, Flyweight, Iterator, Virtual Proxy	
Behavioral Patterns	Chain of Responsibility, Command, Little Language / Interpreter, Mediator, Null Object, Observer, Snapshot, State, Strategy, Template Method, Visitor	
Concurrency Patterns	Asynchronous Processing, Balking, Double Buffering, Future, Guarded Suspension, Lock Object, Producer-Consumer, Read/Write Lock, Scheduler, Single Threaded Execution, Two-Phase Termination	
DIO II	**[50 uzorka]**	**Izvor:** [Grand, 1999.]
GRASP Patterns	Controller, Creator, Expert, Law of Demeter, Low Coupling/High, Cohesion, Polymorphism, Pure Fabrication	
GUI Design Patterns	Conversational Text, Direct Manipulation, Ephemeral Feedback, Explorable Interface, Form, Disabled Irrelevant Things, Interaction Style, Limited Selection Size, Selection, Step-by-Step Instructions, Supplementary Window, Window per Task	
Organizational Coding Patterns	Accessor Method Name, Anonymous Adapter, Checked vs. Unchecked Exceptions, Conditional Compilation, Composed Method, Convert Exceptions, Define Constants in Interfaces, Extend Super, Intention Revealing Method, ServerSocket, Client Socket, Switch, Symbolic Constant Name	
Optimization Coding Patterns	Double Checked Locking , Hashed Adapter Objects, Lazy Initialization, Lookup Table, Loop Unrolling	
Robustness Coding Patterns	Assertion Testing, Copy Mutable Parameters, Guaranteed Cleanup, Maximize Privacy, Return New Objects from Accessor	
Testing Patterns	Acceptance Testing, Black Box Testing, Clean Room Testing, Integration Testing, Regression Testing, System Testing, Unit Testing, White Box Testing	
DIO III	**[37 uzorka]**	**Izvor:** [Grand, 2002.a]
Transaction Patterns	ACID Transaction, Audit Trail, Composite Transaction, Two Phase Commit	
Distributed Architecture Patterns	Demilitarized Zone, Mobile Agent; Object Replication, Object Request Broker, Process Pair, Prompt Repair, Redundant Independent Objects, Shared Object	
Distributed Computing Patterns	Connection Multiplexing, Heartbeat ,Heavyweight/Lightweight Object, Mailbox, Object Identifier, Protection Proxy, Publish-Subscribe, Registry Retransmission	
Concurrency Patterns	Ephemeral Cache Item, Deep Transaction Nesting, Lock File, Optimistic Concurrency, Session Object, Static Locking Order, Thread Pool	
Temporal Patterns	Temporal Property, Time Server, Versioned Object	
Database Patterns	CRUD, isDirty, Lazy Retrieval, Persistence Layer, Stale Object, Type Conversion	

Naziv Kataloga:	**Enterprise Solution Patterns Using Microsoft .NET [32 uzorka] i 32 uzoraka u pripremi**	**Izvor:** [Trowbridge *et al.*, 2003.]
Opis:	Prva zbirka Microsoft-ovih uzoraka vezanih uz poslovne aplikacije.	
Klasifikacija:	**Naziv uzorka oblikovanja:**	**Naziv uzorka implementacije-idioma:**
Web Presentation	Model-View-Controller, Page Controller, Front Controller, Intercepting Filter, Page Cache, Observer	Implementing Model-View-Controller in ASP.NET, Implementing Page Controller in ASP.NET, Implementing Front Controller in ASP.NET Using HTTP Handler, Implementing Intercepting Filter in ASP.NET Using HTTP Module; Implementing Page Cache in ASP.NET Using Absolute Expiration; Implementing Observer in .NET
Deployment	Layered Application, Three-Layered Services Application, Tiered Distribution, Three-Tiered Distribution, Deployment Plan	Three-Layered Services Application, Three-Tiered Distribution
Distributed Systems	Broker, Data Transfer Object, Singleton	Implementing Broker with .NET Remoting Using Server-Activated Objects, Implementing Broker with .NET Remoting Using Client-Activated Objects, Implementing Data Transfer Object in .NET with a DataSet, Implementing Data Transfer Object in .NET with a Typed DataSet, Implementing Singleton in C#
Performance and Reliability	Server Clustering, Load-Balanced Cluster, Failover Cluster	
Services	Service Interface, Service Gateway	Implementing Service Interface in .NET with an ASP.NET Web Service, Implementing Service Gateway in .NET
32 uzoraka u pripremi *(eng. pattlets)*	Four-Tiered Distribution, Abstract Factory, Adapter, Application Controller, Application Server, Assembler, Bound Data Control, Bridge, Command(s), Decorator, Facade, Gateway, Implementing Data Transfer Object in .NET with Serialized Objects, Layer Supertype, Layers, Mapper, Mediator, MonoState, Observer, Naming Service, Page Data Caching, Page Fragment Caching, Presentation-Abstraction-Controller, Proxy, Remote Facade, Server Farm, Special Case, Strategy	

Naziv Kataloga:	**Microsoft Data Patterns and Pattlets [12 uzorka i 4 uzorka u pripremi]**	**Izvor:** [Teale *et al.*, 2003.]
Opis:	Ovaj katalog definira uzorke za rad sa RDBMS.	
Klasifikacija:	**Naziv uzoraka:**	**Naziv uzoraka u pripremi (pattlet):**
Architecture	Move Copy of Data, Data Replication,	Maintain Data Copies, Application-Managed Data Copies, Extract-Transform-Load (ETL), *Topologies for Data Copies*
Design	Master-Master Replication, Master-Slave Replication, Master-Master Row-Level, Master-Slave Snapshot Replication Synchronization, Capture Transaction Details, Master-Slave Transactional Incremental Replication, Master-Slave Cascading Replication	
Impementation	Implementing Master-Master Row-Level Synchronization Using SQL Server, Implementing Master-Slave Snapshot Replication Using SQL Server, Implementing Master-Slave Transactional Incremental Replication Using SQL Server	

Naziv Kataloga:	**IBM developerWorks Pattern Repository**		**Izvor:** [IBM developerWorks]
Opis:	Katalog je klasificiran prema fazama razvoja. Za neke od faza još nema definiranih uzoraka.		
Klasifikacija:	**Naziv klastera i /ili uzoraka:**		
Define business systems strategy	NEMA UZORAKA		
Define architectural strategy	IBM Patterns for e-business	2-dimenzionalna mreža	Application & Runtime patterns
		Business patterns	Self-Service: Stand-Alone Single Channel, Directly Integrated Single Channel, As-Is Host, Customized Presentation to Host, Router, Decomposition, Agent
			Collaboration: Point-to-Point, Store and Retrieve, Directed Collaboration, Managed Collaboration
			Information Aggregation: User Information Access, User Information Access=Write-back variation, User Information Access=Federation variation
			Extended Enterprise: Exposed Direct Connection, Exposed Broker, Exposed Router, Exposed Serial Process
		Integration	Access Integration: Client, Single Sign-On, Personalized Delivery

		patterns	Application Integration	Process Integration application patterns:	Direct Connection, Broker, Router, Serial Process, Serial Workflow variation, Parallel Process, Parallel Workflow variation,
				Data Integration application patterns	Federation, Population, Two-way Synchronization
		Composite patterns	Electronic Commerce	Web-up, Enterprise-out	
			e-Marketplace	General e-Marketplace variation, Sell-Side Hub variation	
			Portals	Access Integration::Web Single Sign-On, Access Integration::Pervasive Device Access, Access Integration::Personalized Delivery, Self-Service::Directly Integrated Single Channel, Collaboration::Store and Retrieve, Collaboration::Directed Collaboration, Information Aggregation::Population, Information Aggregation::Population=Multi Step, Information Aggregation::Population=Multi Step Gather	
			Account Access	Account Access applications	
		Custom designs	Integrating Self-Service and Collaboration using WebSphere and Domino	Domino application with WebSphere Integration platform, WebSphere loaded on a Domino Server, WebSphere application with Domino services, Single sign-on between Domino and WebSphere, Using a security server to manage authentication and connections, Domino User-to-User hybrid Runtime pattern, Sametime User-to-User hybrid Runtime pattern, Combined Sametime and Domino (single machine), Combined Sametime and Domino (separate machine), Sametime and Domino with Policy Director, Single sign-on with Sametime, Domino, and WebSphere, Combined Sametime, Domino, and WebSphere with Policy Director, Domino as a central loosely-coupled broker, Combining caching proxy and load balancer, Using Tivoli Policy Director as a load balancer, Domino app with WebSphere Application Server services: Single server, Domino app with WebSphere Application Server services: Multiple servers, Web redirector with Domino and WebSphere Application Server, WebSphere Application Server application with Domino services, WebSphere Web services with Domino, WebSphere Web services with Domino, Using both Active Directory and Domino Directory, Using both External and Domino directories, Using a security server to manage authentication and connections, Domino for collaboration - Simplest pattern, Sametime only, Sametime and	

				Domino 6, Single sign-on with Sametime, Domino, and WebSphere, Sametime and Domino with Tivoli Access Manager, Combined Sametime, Domino, and WebSphere with Access Manager, Load Balancer and Domino, Caching Proxy and Domino
			Integrating WebSphere Application Server and SAP Products	Self-Service designs for small or mid-sized manufacturing companies, Integration and Extended Enterprises designs for large manufacturing enterprises, Electronic Commerce large distributor/retailer enterprises
			Non-Functional Requirements custom designs	High Availability, High Performance
Define solutions	NEMA UZORAKA			
Build, test, and deploy		**Naziv ponovno iskoristivog resursa (eng. asset)**		
	Service-Oriented Architecture (SOA) patterns	SOA Catalog Legacy Design Model Asset, SOA Imp. and Opt. of Services Recipe Asset, SOA Inventory Enterprise IT Design Model Asset, SOA Inventory Service Design Model Asset, SOA Lookup Item Use Case Model Asset		
	Information service patterns			
	Enterprise patterns	Enterprise Patterns Asset (uključuje: Session Facade, Business Delegate, Message Facade, Data Access Object)		
	WebSphere Platform Messaging patterns	WebSphere Platform Messaging Patterns Asset		
		WebSphere Response Template Pattern Asset		
	State-Oriented Portlet patterns	State-Oriented Portlet Patterns RAS Asset		
	TSA Failover Configuration patterns	TSA Failover Configuration Pattern Asset		
	Security patterns	Security Patterns Asset (uključuje: A method level authorization pattern, A run-as identity pattern, An event pattern)		
	WebSphere Cluster	WebSphere Cluster Creation Pattern Asset		
Operate and maintain	NEMA UZORAKA			

U kontekstu uzoraka pojavljuje se i pojam *antiuzorak*. Ukoliko *uzorak* predstavlja najbolju praksu tada *antiuzorak* predstavlja lekciju koja je naučena (eng. lesons learn) i koju je

potrebno izbjegavati. Koncept je predložio A. Koenig [Appleton, 2000., str. 7.], a mogu se prikazati u 2 notacije:

- Onom kojom se opisuje loše rješenje problema koje je proizašlo iz loše situacije i koje je potrebno izbjegavati jer će njegova primjena dovesti do neuspjeha.
- Onom kojom se opisuje kako izaći iz loše situacije i kako tada nastaviti da bi se postiglo dobro rješenje.

U nastavku je prikazana i sistematizacija *antiuzoraka.*

Tablica 4-2. Sistematizirani pregled kataloga softverskih antiuzoraka

Naziv Kataloga:	**Antipatterns [112 uzorka]**	**Izvor:** [Anti Patterns Catalog, 2007.]
Opis:	Opsežna sistematizacija antiuzoraka po aspektima razvoja.	
Klasifikacija:	**Naziv antiuzoraka:**	
Architecture AntiPattern	ArchitectureAsRequirements, ArchitectureByImplication, AutogeneratedStovepipeAntiPattern, CoverYourAssets, DesignByCommittee, DoerAndKnower, ExceptionFunnel, FloatingPointCurrency, FloatingPointFractions, IntellectualViolence, ImplementationInheritance, JumbleAntipattern, ReinventTheWheel, RequirementsAsArchitecture, RollYourOwnDatabase, SpaghettiCode, StovepipeEnterprise, StovepipeSystem, SumoMarriage, SwissArmyKnife, TheGrandOldDukeOfYork, VendorLockIn, WarmBodies, WolfTicket	
Development AntiPattern	BearTrap, BoatAnchor, ContinuousObsolescence, CrciCards, CreepingFeaturitis, CopyAndPasteProgramming, DeadEnd, FireDrill, FoolTrap, FunctionalDecomposition, GoldenHammer, GrenadeMessage, HiddenRequirements, IfItIsWorkingDontChange, InputKludge, LavaFlow, MagicContainer, MushroomManagement, PathOfLeastResistance, PolterGeists, RequirementsTossedOverTheWall, SecretSociety, TheBlob, WalkingThroughaMineField, AmbiguousViewpoint, RubeGoldbergMachine, SpecifyNothing, TowerOfVoodoo, ZeroMeansNull	
GreyPattern	IfItIsWorkingDontChange	
Management AntiPattern	AnalysisParalysis, AnAthena, AppointedTeam, ArchitectsDontCode, BlowhardJamboree, CarbonCopyHisManager, CornCob, DeathByPlanning, DecisionByArithmetic, DiscordantRewardMechanisms, DryWaterhole, EgalitarianCompensation, EmailIsDangerous, EmperorsNewClothes, FalseSurrogateEndpoint, FearOfSuccess, GeographicallyDistributedDevelopment, GiveMeEstimatesNow, GlassWall, HeirApparent, HeroCulture, HiddenRequirements, IrrationalManagement, ItsNotRocketScience, LeadingRequest, ManagerControlsProcess, NetNegativeProducingProgrammer, PlugCompatibleInterchangeableEngineers, ProjectMismanagement, ScapeGoat, ShootTheMessenger, SmokeAndMirrors, SpecifyNothing, TheCustomersAreIdiots, TheFeud, ThrownOverTheWall, TrainTheTrainer, ViewgraphEngineering, WeAreIdiots, YetAnotherMeetingWillSolveIt, YetAnotherProgrammer	
Not Yet Classified AntiPattern	BigBallOfMud, CargoCult, IdiotProofProcess, KitchenSinkDesign, NationalIsm, NotInventedHere, SoftwareMerger, ThrownOverTheWall, AsynchronousUnitTesting, CascadingDialogBoxesAntiPattern, RansomNoteAntiPattern, AnalogyBreakdownAntiPattern, SingleFunctionExitPoint, PassingNullsToConstrutors,	

UserInterface	RansomNoteAntiPattern, CascadingDialogBoxesAntiPattern, PhatWareAntiPattern

Naziv Kataloga:	**J2EE Antipatterns [54 uzorka]**	**Izvor:** [Dudney *et al.*, 2003.]
Opis:	Katalog je strukturiran prema različitim tipovima J2EE razvoja kroz antiuzorke opisuje što prilikom dizajna i definiranja arhitekture može krenuti u krivom, neželjenom smjeru.	
Klasifikacija:	**Naziv antiuzorka:**	
Distribution and Scaling	Localizing Dana, Misunderstanding Data Requirements, Miscalculating Bandwidth Requirements, Overworked Hubs, The Man with the Axe	
Persistence	Dredge, Crush, DataVision, Stifle	
Service-Based Architecture	Multiservice, Tiny Service, Stovepipe Service, Client Completes Service	
JSP Use and Misuse	Ignoring Reality, Too Much Code, Embedded Navigational Information, Copy and Paste JSP, Too Much Data in Session, Ad Lib TagLibs	
Servlets	Including Common Functionality in Every Servlet, Template Text in Servlet, Using Strings for Content Generation, Not Pooling Connections, Accessing Entities Directly	
Entity Beans	Fragile Links, DTO Explosion, Surface Tension, Coarse Behavior, Liability, Mirage	
Session EJBs	Sessions A-Plenty, Bloated Session, Thin Session, Large Transaction, Transparent Façade, Data Cache	
Message-Driven Beans	Misunderstanding JMS, Overloading Destinations, Overimplementing Reliability	
Web Services	Web Services Will Fix Our Problems, When in Doubt, Make It a Web Service, God Object Web Service, Fine-Grained/Chatty Web Service, Maybe It's Not RPC, Single-Schema Dream, SOAPY Business Logic	
J2EE Services	Hard-Coded Location Identifiers, Web = HTML, Requiring Local Native Code, Overworking JNI, Choosing the Wrong Level of Detail, Not Leveraging EJB Containers	

4.7. Struktura uzorka

Strukturu svakog uzorka treba činiti njegova specifikacija (opis) i implementacija. U opisu svakog uzorka definirani su elementi koje je potrebno obuhvatiti specifikacijom, dok implementacija uključuje njegovu realizaciju primjenjivu u nekom okruženju.

4.7.1. Specifikacija uzorka

Uzorci su tradicionalno definirani dokumentacijom koja se naziva *specifikacija uzorka*, a predstavljaju preciznu definiciju i opis cjelokupnog konceptualnog znanja o problemu i rješenju tog problema koje omogućava komunikaciju razvojnih inženjera i ponovno korištenje.

Uz uobičajeni tekstualni oblik, specifikacija može biti nadopunjena i dijagramom [Rosengard, Ursu, 2004.]. Kako bi se uklonila potencijalna dvosmislenost tekstualnog opisa i

nedorečenost dijagrama uzorci se mogu opisati i formalnim pristupom [Kermek, 1998., str 89.], [Rosengard, Ursu, 2004.].
U radu, pod pojmom *specifikacija uzorka* podrazumijevat će se tekstualni oblik opisa uzorka. U nastavku slijedi opis formata i elemenata specifikacije uzoraka.

4.7.1.1. Formati opisa uzorka

Tijekom dosadašnjeg razvoja uzoraka, autori koriste različite formate (oblike) opisa uzoraka. Prema [Fowler, 2006., str. 6,7.] i [Appleton, 2000., str. 9.] za opis uzoraka najpoznatiji su:

Alexandrian format. Format koji je koristio Alexandar u svojim knjigama poznat još i pod nazivom *kanonski format*. Karakterizira ga poprilično narativni stil uz mali broj strukturiranih elemenata pri čemu su najznačajnije rečenice o problemu i rješenju naglašene.

GoF format. Format primijenjen za opis 23 *uzorka dizajna* autora GoF u njihovoj knjizi *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma *et al.*, 1997.]. Karakterizira ga vrlo strukturirani oblik s mnoštvo elemenata poput: *Svrha, Motivacija, Primjenjivost, Struktura, Sudionici, Suradnja, Posljedice, Implementacija, Primjer koda, Poznate primjene* i *Povezani uzorci*.

Portland format. Naziv formata proizlazi iz prve konferencije o uzorcima na kojoj su nekolicina autora koristila slični oblik. U potpunosti je tekstualnog i kratkog oblika.

Coplien format. Naziv formata dolazi od autora Coplien, J.. Vrlo praktičan, sažet način s ključnim elementima: *Problem, Kontekst, Okruženje* i *Rješenje*.

POSA format. Ovaj je format usvojen na temelju knjige *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* [Buschmann *et al.*, 1996.] te je sličan GoF formatu po strukturi i velikom broju elementa. No elementi kojima se opisuje uzorak razlikuju od GoF formata. To su: *Sažetak, Primjer, Sadržaj, Problem, Rješenje, Struktura, Dinamika, Implementacija, Riješeni primjeri, Varijante, Poznata primjena* i *Posljedice*. Uz sve ovo sadrži i *Sažetak* u narativnom opisu.

P of EAA format. Jedan od formata koji se rijetko koristi. Potpuno je narativan i sadrži samo nekoliko dijelova: *Kako radi, Kada se koristi* i *Primjeri*.

4.7.1.2. Elementi opisa uzorka

Unatoč primjeni različitih formata za opis uzoraka mogu se izdvojiti esencijalni elementi koji su prepoznatljivi u opisu svakog uzorka. To su [Appleton, 2000., str. 9-11.]:

Naziv. Smislenim nazivom u obliku riječi ili kratke fraze želi se naznačiti sadržaj uzorka. Ponekad uzorak može imati više od jednog najčešće korištenog ili prepoznatljivog imena te mu se tada dodjeljuju podelementi: *Alias* ili *Poznat i pod nazivom*.

Problem. Dio u kojem se opisuje problem, ciljevi te svrha koja se želi postići primjenom uzorka u rješavanju nekog problema.

Kontekst. Sadrži opis preduvjeta u kojima se pojavljuje problem kao i onih u kojima je rješenje tog problema prikladno.

Utjecaji faktori. Opisuju se relevantni utjecaji i ograničenja, načini na koji oni međusobno utječu kao i kako utječu na ciljeve koji se žele postići.

Rješenje. Sadrži prikaz u obliku teksta, skica i dijagrama kojima se prikazuje kako statička struktura tako i dinamično ponašanje realizacije željenog rješenja.

Primjeri. Jedan ili više primjera dijelova aplikacije u kojima je uzorak primijenjen kako bi se korisniku povećalo razumijevanje o primjenjivosti uzorka.

Rezultat. Opisuje stanje ili konfiguraciju sustava nakon primijene uzorka. Opis uključuje i posljedice koje donosi primjena uzorka, bile one dobre ili loše, kao i potencijalne probleme koji se mogu pojaviti u tom novom okruženju.

Povezani uzorci. Dio u kojem se definira statička i dinamička povezanost više uzorka koji se primjenjuju pri rješavanju nekog problema.

Poznata primjena. Navode se praktični primjeri primjene.

Iako nije striktno zahtijevano, kvalitetni uzorci sadrže i **Sažetak** kako bi čitatelj dobio jasno sliku o uzorku i informirao se da li je isti prikladan za njegov problem. Specifikacija uzorka predstavlja osnovu za njegovu implementaciju.

4.7.2. Implementacija uzorka

Od pojave prvih uzorka (GoF //uzorci oblikovanja) prošlo je više od 20 godina. Kao što je prikazano u točci 4.6., u kontekstu razvoja programskog proizvoda, razvijen je zaista velik broj uzorka različitih namjena. Iako postoje mnogobrojni katalozi koji obuhvaćaju velik broj uzoraka oni su zapravo definirani samo u obliku *specifikacije uzoraka* (tekstualni opis u

nekom formatu) koje je tada ručno potrebno implementirati. Kako rad naglašava evoluiranje današnjeg razvoja softvera pojavom MDD i želje za automatizacijom, i u definiranju uzoraka postižu se unaprjeđenja, pa se tako danas prilikom definiranja strukture uzoraka govori o implementaciji uzoraka.

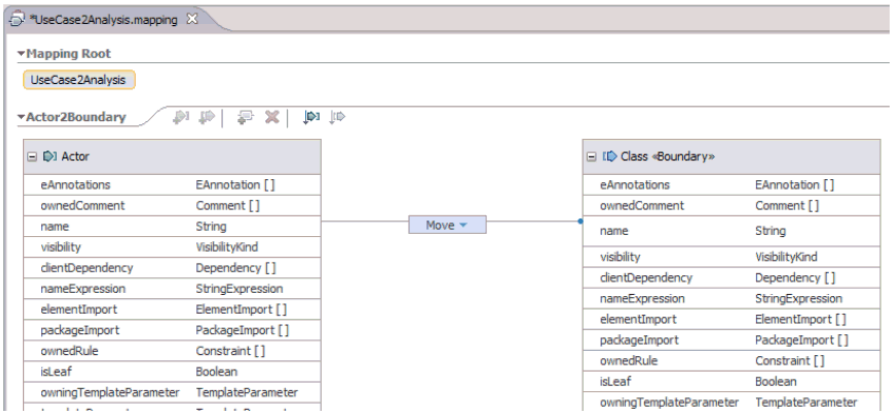
Implementacija uzoraka zapravo predstavlja artefakt kojim je specifikacija uzorka prevedena u oblik preko kojeg će se automatizirati neki dijelovi razvoja aplikacije u nekom razvojnom okruženju [Ackerman, Gonzalez, 2007., str. 1.]. Na temelju implementacije uzorka moguće je generirati različite tipove artefakata kao što su: UML modeli, programski kod, skripte ili neki drugi artefakti temeljeni na tekstu.

Implementacija uzoraka u konkretnom razvojnom okruženju (IBM Rational Software Architect) može se promatrati kao:

- UML uzorak i
- transformacije modela.

UML uzorak: predstavlja artefakt koji se primjenjuje u kontekstu modeliranja kako bi se označio (eng. *mark*) model, dodali novi elementi u model ili povezali novi ili postojeći elementi. Ovaj tip implemantacije razmatra se kasnije kroz točku 5.3.2.5.2. Primjer implementacije uzorka kroz modeliranje za jedan uzorak pod nazivom *Session Facade* u razvojnom okruženju RSA 6.0.1. prikazan je na *slici 5-7*. Slika prikazuje model koji nastaje povezivanjem postojećih klasa (Klijent i Racun) s implementacijom uzorka (UML uzorak u obliku koncepta suradnje).

Transformacije modela: Svaka transformacija prevodi element izvornog modela u element ciljnog modela primljenom definicije transformacije u kojoj se koriste pravila transformacije opisana nekim formalnim jezikom. Primjer takvog artefakta u razvojnom okruženju RSA 6.0.1. prikazan je na *slici 4-1*.



Slika 4-1. Primjer implementacije uzorka - transformacija

Izvor: Razvojno okruženje RSA 6.0.1.

Transformacijom je definirano generiranje <<Boundary>> klase za svakog učesnika u modelu analize.

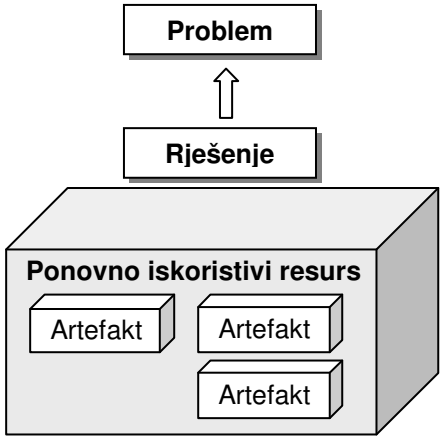
Sažeto rečeno, implementacijom uzoraka se najbolja praksa opisana u specifikaciji uzoraka kodira u oblik pogodan za automatizaciju kojom se dobiva programski kod ili neki drugi neophodni artefakti.

4.8. Standardizacija uzoraka

Ideja implementacije uzorka, u cilju automatizacije razvoja u kontekstu MDD paradigme, vrlo je privlačna. Artefakt koji nastaje tijekom implementacije uzorka, arhivira se u *repozitorij uzoraka* iz kojeg se tada učitava u razvojno okruženje omogućavajući dostupnost (djeljivost) i ponovno korištenje. U cilju realizacije tog scenarija, važan aspekt u razvoju uzoraka, predstavlja definiranje standardiziranog načina organiziranja, specificiranja, implementacije i pakiranja uzoraka.

Danas vodeće kompanije u SW industriji poput IBM, Microsoft, Component Source i drugih, traže načine višenamjenskog ulaganja u softver. Utvrđeno je da softverski entiteti trebaju biti imenovani, organizirani, ispitani te ponovno iskoristivi kako bi se povećao njihov povrat ulaganja (eng. Return On Invesment). Pojam kojim se objedinjavaju svi tipove različitih artefakata koji se primjenjuju u procesu razvoju nazvan je *ponovno iskoristivi softverski resurs* (eng. Reusable Software Asset). Prema, [OMG, 2005., str. 7.] *softverski resurs* predstavlja *skup artefakta koji doprinose rješenju problema.*

Na *slici 4-2.* prikazan je opis softverskog resursa na najvišoj razini.



Slika 4-2. Prikaz softverskog resursa

Izvor: [OMG, 2005., str. 7.]

Svaki softverski resurs mora sadržavati:

- datoteku koja se može učitati u razvojno okruženje (.ras) i
- jedan ili više artefakta.

Artefakti koje softverski resurs sadrži ovise o kontekstu ponovnog korištenja. Tako u kontekstu razvoja programskog proizvoda softverski resurs, npr. može sadržavati neke od slijedećih artefakata:

- specifikacije zahtjeva,
- modele,
- pravila transformacije,
- izvorni kod,
- skripte i
- testove i dr.

Prema [Larsen, 2003., str. 7.], [Larsen, 2006., str. 1.] u *vrste softverskih resursa* ubrajaju se:

- *softverski uzorci,*
- komponente,
- web servisi i
- predlošci.

Gore navedena opća definicija softverskog resursa može se prilagoditi na svaku od

navedenih vrsta softverskih resursa.

U nastavku ovog rada istražit će se mogućnost standardizacije softverskih uzoraka kao ponovno iskoristivih resursa čime bi se objedinila dosadašnja specifikacija i implementacija softverskih uzoraka.

4.8.1. Standard za ponovno iskoristivi softverski resurs - RAS

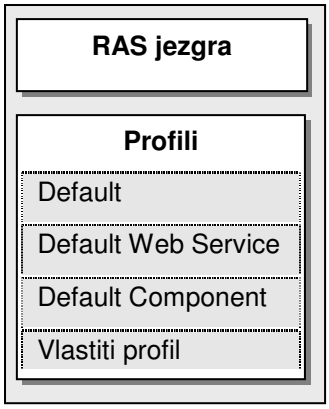
U razvoju programskog proizvoda postoji mnogo vrsta artefakta oblikovanih u različitim formama i stilovima što povećava cijenu razvoja. U cilju ponovne iskoristivosti i standardizacije artefakata koji nastaju u razvoju programskog proizvoda nastoji ih se prikazati kao *ponovno iskoristive softverske resurse*.

Razmatrajući specifikaciju uzoraka, u točci 4.7.1.1. mogu se primijetiti različite forme i stilovi njihova pisanja. Kako bi se osigurala konzistentnost uzoraka, neophodno je standardizirati način njihovog definiranja. Ujedno, u točci 4.8. razmotreno da se i uzorci mogu promatrati kao *ponovno iskoristivi softverski resursi*. Stoga, kako bi se strukturirao i organizirao svaki softverski resurs, pa tako i uzorak, i kako bi se točno znalo koje informacije treba sadržavati i kako ga je moguće pohraniti u paket za njegovo ponovno korištenje, smatram da je poželjno koristiti već definiran standard pod nazivom **RAS** (eng. Reusable Asset Specification) koji je 2005. godine prihvaćen kao OMG standard. RAS standard osigurava način arhiviranja, pretraživanja, organiziranja, dokumentiranja i dijeljenja (raspoloživosti) uzoraka.

Kao što se vidi na *slici 4-3.*, RAS je strukturiran u dva glavna dijela [OMG, 2005., str. 11.]:

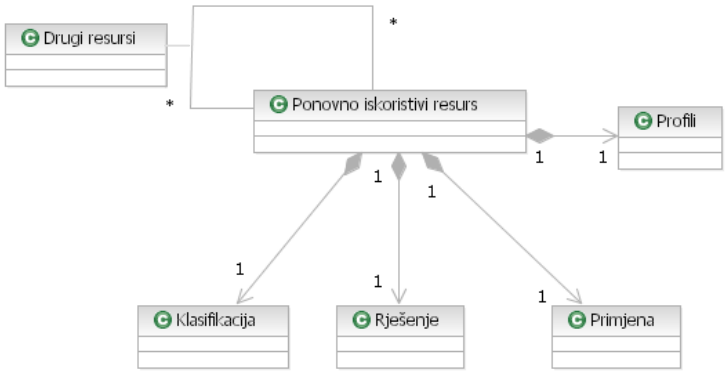
- RAS jezgra i
- profili.

RAS jezgra čini osnovni element *specifikacije* softverskog resursa (u ovom slučaju uzorka), dok se profilom mogu opisati proširenja tih osnovnih elemenata specifikacije.



Slika 4-3. Struktura softverskog resursa

Izvor: [OMG, 2005., str. 12.]

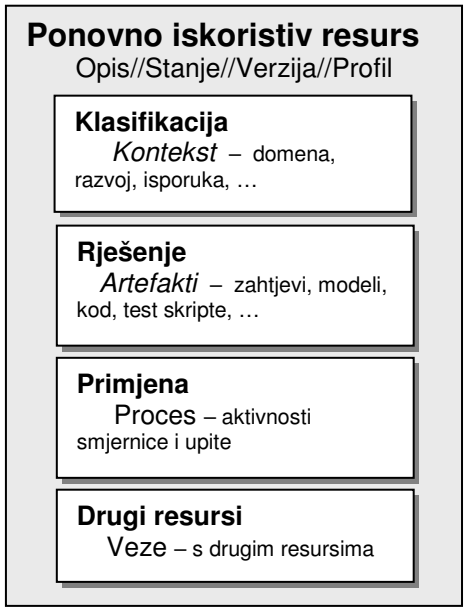


Slika 4-4. RAS meta struktura

Izvor: [OMG, 2005., str. 13.]

Slika 4-4. prikazuje meta model RAS specifikacije.

Slika 4-5. prikazuje glavne elemente jezgre RAS specifikacije kojim se na uniforman način može specificirati svaki uzorak.



Slika 4-5. Struktura RAS jezgre

Izvor: [OMG, 2005., str. 12.]

U zaglavlju svakog softverskog resursa nalaze se atributi vezani za resurs – opis, stanje, verzija i profili vezani uz resurs.

Jezgru svakog softverskog resursa čine 4 glavna dijela:

- *Klasifikacija:* skup opisa klasifikacije softverskog resursa i njegovog sadržaja.
- *Rješenje:* skup artefakata koje softverskog resursa sadrži.

- *Primjena:* aktivnosti i pravila, način primjene, prilagodba softverskog resursa.
- *Drugi resursi:* opis povezanosti softverskog resursa s drugim softverskim resursima.

Kako mogu postojati različiti tipovi softverskog resursa (web servis, komponenta, uzorak i sl..), svaki je tip definiran odgovarajućim profilom pa tako govorimo o posebnim profilima za pojedini tip resursa. Iz *slike 4-3.* vidljivo je da su trenutno na raspolaganju 3 profila uz mogućnost kreiranja vlastitog profila za pojedini tip softverskog resursa (Default, Default Web Service, Default Component i Vlasititi profil).

I dok je specifikacija uzorka sadržana u jezgri i eventualno nekim profilom, u nastavku opisujem način *implementacije i pakiranja* softverskog uzoraka u repozitorij.

Na temelju jezgre softverskog resursa, koja, kao što je rečeno, predstavlja specifikaciju uzorka, kreće se u implementaciju uzorka. Realizacija specifikacije koja nastaje implementacijom naziva se *paket*. Implementacija započinje prevođenjem pisane specifikacija u strog formalan opis koji se naziva XML Schema (`.xsd`), a definira se RAS Default profilom (ili nekim drugim - Default Web Service, Default Component i Vlastiti profil). Na temelju XML Scheme (profila) kreira se meta datoteka pod nazivom *manifest datoteka* (`.rmd`). Svaki softverski resurs mora sadržavati barem jednu manifest datoteku. Datoteka sadrži meta podatke o pakiranju softverskog resursa, sadržaju i tipu (određeno profilom) kao i meta podatke kojima su definirana i druga svojstva softverskog resursa, a to su: ključne riječi za pretraživanje, sažet opis i upute (dokumentacija) za korisnika uzorka. Toj se datoteci dodjeljuje formalni naziv `rasset.xml` i ona predstavlja ishodiše za svaki softverski resurs tj. paket. Datoteci su priložene i datoteke koje reprezentiraju artefakte koje sadrži taj resurs. Sve te datoteke zajedno se kompresiraju u **`.ras`** format i kao paket se smještavaju u RAS repozitorij iz kojeg se tada mogu importirati u neko razvojno okruženje.

Znači, svaki softverski resus (`.ras`) sadrži slijedeće tipove datoteka:

- jednu ili više XML Scheme (profila) [npr. `RASProfil.xsd`],
- jednu *manifest datoteka* (`.rmd`) u glavnom direktoriju i
- jednu ili više datoteka koje predstavljaju artefakte koje reurs sadrži.

5. METODOLOŠKI OKVIR PRIMJENE UZORAKA U RAZVOJU TEMELJENOM NA MODELIMA

U dosadašnjim poglavljima detaljno je razmotreno današnje stanje u softverskoj industriji, analizirana je MDD paradigma kao i koncept pojave, razvoja i primjene uzoraka u softverskoj industriji. Kroz slijedeća poglavlja, promatrat će se primjena *softverskih uzoraka* u kontekstu *MDD paradigme* podržana *razvojnim okruženjem (poglavlje 5)* i prikladnost primjene današnjih *procesa razvoja* za MDD razvoj u cilju provođenja integracije s razvijenim okvirom *(poglavlje 6)*.

Kako se postavljene hipoteze temelje na činjenici da primjena uzoraka, u nekom razvojnom okruženju, pridonosi realizaciji MDD paradigme, u ovom će se poglavlju najprije definirati *uvjeti primjene* softverskih uzoraka u razvoju temeljenom na modelima. U cilju dokazivanja prve hipoteze (H1), ovo će poglavlje zatim, detaljno prikazivati vlastito osmišljen i razvijen metodološki okvir, koji obuhvaća cijeli životni ciklus uzorka u kontekstu MDD razvoja te sadrži smjernice, upute i korake koji bi trebali predstavljati "najbolju praksu" primjene uzoraka u MDD razvoju. Svaki teorijski segment metodološkog okvira bit će objašnjen kroz praktičnu primjenu IBM Rational razvojnog okruženja.

5.1. Uvjeti primjene SW uzoraka u razvoju temeljenom na modelima

Od svoje pojave pa do danas, primjena uzoraka u razvoju programskih proizvoda daje značajan doprinos, što potvrđuje i njihova raširenost u različitim segmentima razvoja. No, kako se može razabrati iz četvrtog poglavlja primijećeni su i određeni nedostaci. Broj uzoraka izrazito je velik, a svakim danom nastaju novi uzorci. Oblici njihovih opisa nisu sistematizirani - ne postoji unificirani način prikazivana, interpretacije i realizacije. Primjećuje se da je većina postojećih uzoraka samo na konceptualnoj razini (tekstualni opis) što je nedovoljno za primjenu u MDD razvoju. Stoga je slijedeći logičan korak, analizirati primjenjivost softverskih uzoraka u razvoju temeljenom na modelima. Definirajući uvijete primjene, jasno bi se trebali moći identificirati uzorci koji su već sada primjenjivi u MDD razvoju te pružiti smjernice za razvoj novih.

Kroz ovu točku dat će se odgovor na slijedeća dva pitanja:

1. *Mogu li prednosti koje donosi primjena uzoraka biti prepoznate i korištene u realizaciji MDD paradigme?*
2. *Kako treba biti definiran uzorak primjenjiv u MDD okruženju?*

Odgovorimo najprije na prvo pitanje.

Mnogobrojne prednosti primjene uzorka u razvoju programskih proizvoda razmotrene su u točci 4.3. Gotovo sve navedene prednosti mogu se pronaći i u prednostima primjene MDD paradigme navedene u točci 3.1.2. To ukazuje na činjenicu da primjena uzoraka u značajnoj mjeri može pridonijeti realizaciji MDD paradigme. Kako?

Komplementaran odnos između uzorka i MDD paradigme vidljiv je u dva oblika:

1. *Softverski uzorci predstavljaju **sadržaj** za MDD razvoj.* Ekspertiza opisana uzorkom predstavlja "najbolju praksu" rješenja problema koji omogućava ponovno korištenje te su zbog toga uzorci poželjan sadržaj za MDD razvoj.
2. *Osnovna misao MDD paradigme je postizanje **automatizacije** prilikom razvoja modela i njihovih implementacija, koja se može postići primjenom uzoraka.* Ukoliko uzorak nije definiran samo u obliku teksta koji tada programer mora ručno programirati, već je definiran na standardizirani način, uz implementaciju i pakiran kao ponovno iskoristivi resurs, uzorak će na automatizirani način objediniti "najbolju praksu" od konceptualne razine do implementacije u programskom kodu.

Može se zaključiti kako je kvalitetna realizacija MDD teško održiva bez primjene uzoraka. Primjenom uzoraka u MDD razvoju, problemska domena opisana je rješenjima koje predstavljaju "najbolju prasku". Tijekom modeliranja i implementacije primjenom uzoraka, razvoj će se automatizirati, a ujedno će se i osigurati mogućnost ponovnog korištenja u sličnim situacijama.

Rezultati koje bi donijela primjena uzoraka u razvoju temeljenom na modelima vidljivi su u [Gardner, Yusuf, 2006.]:

- smanjenju vremena potrebnom za djelovanjem,
- mogućnostima dizajniranja i razvoja softvera prema zahtjevima (eng. on demand),
- smanjenju složenosti rješenja i
- povećanju produktivnosti i kvalitete programskog rješenje.

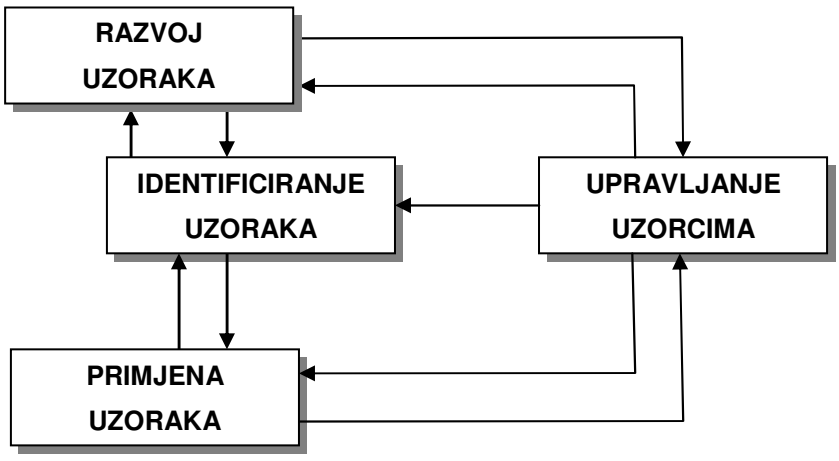
Iz dosadašnjeg razmatranja jasno se mogu definirati ***uvjeti*** koji trebaju biti zadovoljeni želi li se uzroke koristiti u razvoju temeljenom na modelima:

- **razvojno okruženje mora podržavati osnovne principe MDD razvoja,**
- **uzorci moraju biti strukturirani na standardizirani način i to u obliku ponovno iskoristivog resursa koja sadrži kako specifikaciju tako i implementaciju,**
- **uzorak mora biti implementiran kao paket, arhiviran u repozitorij uzoraka i u obliku pogodnom za importiranje u razvojno okruženje i**
- **razvojno okruženje treba imati mogućnosti primjene tako definiranih uzorka u cilju realizacije MDD paradigme - postizanja automatizacije.**

Naravno, na teorijskoj razini efekt sinergijskog djelovanja lako je uočljiv. No praksa se trenutno suočava s brojnim pitanjima na koja se još očekuju odgovori. U odnosu na broj uzoraka koji postoji u industriji, samo mali postotak zadovoljava upravo navedene uvijete. Nadalje, i za tako mali broj uzoraka pojavljuju se nova pitanja koja su vezana za način njihove primjene. Stoga se u nastavku rada definira metodološki okvir primjene uzorka koji bi trebao predstavljati doprinos u segmentu razvoja programskih proizvoda temeljenom na modelima.

5.2. Definiranje opsega okvira

KONTEKST: Kako bi razvijeni okvir bio metodološki prihvatljiv i praktično primjenjiv, neophodno je da njegovo oblikovanje obuhvaća sve segmente životnog ciklusa uzorka (tj. da bude cjelovit). Kako je jedan od definiranih uvjeta da uzorak bude ponovno iskoristiv resurs, životni ciklus uzroka, prikazan na *slici 5-1.*, upravo se temelji na životnom ciklusu ponovno iskoristivog resursa.



Slika 5-1. Životni ciklus uzorka

Ovako definirani segmenti životnog ciklusa uzorka činit će kostur razvoja okvira koji će biti prilagođen i proširen specifičnostima vezanim uz uzorke. Na taj je način određen kontekst okvira koji se želi definirati.

PRAKTIČNA PRIMJENA: Već je u uvodu ovog poglavlja spomenuto da se paralelno s *razvojem okvira* želi prikazati i praktična *primjena* tako razvijenog okvira kako bi se potkrijepila njegova vrijednost. Ideja razvoja okvira je njegova primjenjivost u svim fazama MDD razvoja. No, kako bi prikaz primjene različitih tipova kataloga uzoraka, koji zadovoljavaju gore navedene uvijete, kroz cjelokupan MDD razvoj bio preopsežan, praktična vrijednost primjene razvijenog okvira, kojom se želi potkrijepiti razvoj okvira, ograničit će se na promatranje samo jednog *tipa kataloga uzorka*, a to su *uzorci oblikovanja*. Upravo katalozi s *uzorcima oblikovanja* primjenjivi su u segmentu razvoja u kojem MDD pristup treba pružiti najznačajnije rezultate, a to su svakako faze razvoja *dizajn* i *implementacija*. Drugim riječima, parcijalni primjer(i) kojim će se potkrijepiti praktična vrijednost pojedinih segmenata razvijenog okvira bit će iz samo jednog tipa kataloga uzorka čiji katalozi se primjenjuju u fazama dizajna i implementacije.

5.3. Razvoj metodološkog okvira

Primarne intencije u razvoju okvira su:

- obuhvatiti cijeli životni ciklus uzorka,
- osigurati primjenjivost u svim fazama MDD razvoja i
- osigurati autonomnost okvira, ali s mogućnošću uske povezanosti s odabranom metodikom razvoja.

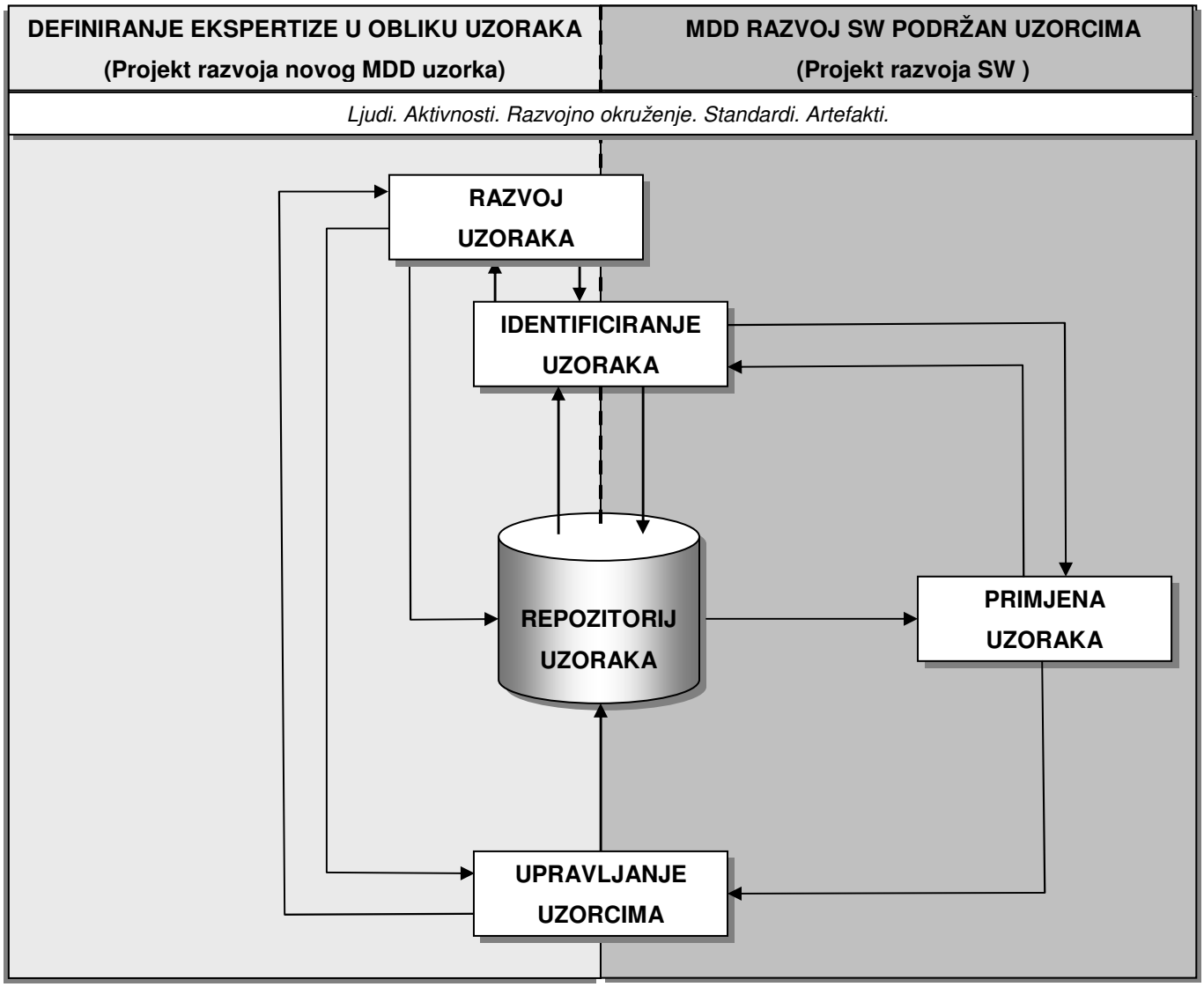
Već na prvi pogled mogu se raspoznati dvije grupe aktivnosti koje su usmjerene ili na *razvoj novih uzoraka* ili na *MDD razvoj programskog proizvoda podržan primjenom uzoraka*. Aktivnosti su isprepletene, jer se *razvoj novih uzoraka* može provoditi i tijekom *MDD razvoja programskog proizvoda podržanog uzorcima*, ali se može promatrati i izvan tog konteksta – neovisno, kao samostalan klasičan razvoj programskog proizvoda.

Smatram da je prikladno odvojiti te grupe aktivnosti, jer *aktivnost razvoja uzorka* ima u fokusu *nove uzorke* što uključuje tim koji ih identificira, određuje njihovu isplativost, razvija ih, testira te na kraju prepušta širokoj primjeni dok je *aktivnost primjene tijekom MDD razvoja* usmjerene na *projekt* razvoja novog programskog proizvoda koji bi primjenom uzoraka i MDD paradigme trebalo ostvariti prednosti. Ujedno kako je ideja u budućnosti imati repozitorij velikog broja uzorka prikladnih za MDD razvoj, smatram da je razvoj i organizaciju potrebno odvojiti (koliko je to moguće), i posvetiti joj znatniju pažnju, kako se ne bi dešavalo da se, nakon završetka projekta razvoja programskog proizvoda, novo razvijeni uzorci stihijski stave u repozitorij. Naravno ovdje ne podrazumijevam *strogo* odvajanje aktivnosti već neku vrstu logičkog grupiranja.

Stoga, novo razvijeni okvir obuhvaća dvije dimenzije:

1. **Definiranje ekspertize u obliku uzorka.** U fokusu je razvoj novih uzoraka u kojem tim iskusnih inženjera radi na razvoju novih uzoraka.
2. **MDD razvoj podržan uzorcima.** U fokusu je projekt razvoja novog programskog proizvoda primjenom već postojećih uzoraka i koncepata MDD paradigme.

U ovako definirane dimenzije smješten je životnog ciklusa uzorka. Na *slici 5-2.* može se vidjeti vizualizacija konteksta osmišljenog okvira. Kada se segmenti *životnog ciklusa uzorka* smjeste u te dvije dimenzije vidljivo je da su i neki od njih isprepliću u obje dimenzije.



Slika 5-2. Vizualizacija konteksta osmišljenog okvira

Definirajući **tko?, što?, kada? i kako?** detaljno će se opisati realizacija svakog segmenta razvijenog okvira. Redoslijed razmatranja okvira je: identificiranje, primjena, razvoj i upravljanje.

5.3.1. Identificiranje uzoraka

Svrha: Pronaći MDD uzorke za uočene klase problema i/ili identificirati nove probleme koje je moguće definirati kao uzorke.

Nakon definiranja problemske domene, ili preciznije rečeno u svakoj fazi razvoja potrebno je postaviti pitanje:

Postoji li za probleme prepoznate u tom segmentu razvoja, MDD uzorak – rješenje koje predstavlja "najbolju praksu"?

U dobivanju odgovora prolazi se kroz slijedeće aktivnosti:

5.3.1.1. Otkrivanje i lociranje uzoraka

Za odgovarajuće klase problema, ili već postoje MDD uzorci (trenutno ih je vrlo malo) ili ih je potrebno razviti. Kako bi se ispitalo postoje li uzorci za odgovarajući problem potrebno je u repozitoriju uzoraka vršiti **pretraživanja** po određenim kriterijima (naziv, tip i sl.). Zatim je, za sve uzorke koji zadovoljavaju definirane kriterije pretraživanja, potrebno provesti detaljno **pregledavanje** specifikacije (opis) i implementacije (rješenje) kako bi se utvrdila njihova prikladnost za postojeći problem i primjenu u MDD razvoju. Ukoliko je pronađen uzorak koji odgovara problemu, prelazi se na slijedeći korak, a to je ***Primjena*** (vidi točku 5.3.2.). No, ukoliko odgovarajući uzorak nije pronađen, problem može postati kandidat za razvoj novog uzorka.

5.3.1.2. Kandidiranje problema (i rješenja) za uzorak

Kako se ovu aktivnost treba promatrati kao generiranje potencijalnih kandidata (eng. brainstorming), neće sve kandidate biti moguće oblikovati kao uzorke. Stoga je potrebno utvrditi prikladnost problema i potencijalnog ili postojećeg rješenja za uzorak. Niz pitanja na koja je potrebo odgovoriti su:

- Kolika je važnost problema za projekt? (evaluacija važnosti)
- Može li se očekivati da će se problem pojaviti u novim identičnim ili sličnim projektima? (procjena ponovne iskoristivosti)
- Da li se dobiveno rješenje može primijeniti *n* puta? (donošenje vrijednosti)

- Može li se problem kao i njegovo rješenje uopće oblikovati kao ponovno iskoristivi resurs – MDD uzorak (kao "najbolja praksa" i da zadovoljava uvijete definirane gore)?

U slučaju da kandidirani problem neće biti realiziran kao uzorak, znanje o tome potrebno je također, formalno oblikovati kako bi ga se moglo iskoristiti u budućim situacijama u kojima bi se mogao pojaviti takav problem. Način na koji bi bilo najprikladnije provesti dokumentiranje je u obliku **antiuzorka** koji bi se također smjestio u repozitorij.

No, ukoliko je odgovor na navedena pitanja potvrdan uslijeđuje inicijalna analiza isplativosti razvoja novog uzorka.

5.3.1.3. Inicijalna analiza isplativosti razvoja novog uzorka

U ovom je koraku potrebno utvrditi da li je investicija razvoja tog uzorka opravdana za buduće slične probleme. Neka od pitanja s kojima se treba suočiti su:

- Koliko su troškovi razvoja uzorka uz samu jednu primjenu?
- Koju vrijednost bi razvoj tog uzorka mogao donijeti poduzeću (eng. ROI)?
- Tko će platiti njegov razvoj?
- Tko će biti zadužen za eventualno održavanje?
- Kolika je održivost njegove primjene ovisno o domeni?

Ukoliko se analiza prihvati, te se dokaže opravdanost razvoja novog uzorka kojim bi se riješio definirani problem, prikuplja se sva dokumentacija čime nastaju *zahtjevi razvoja uzorka* te se prelazi u novi segment ***razvoja*** (vidi točku 5.3.3.). Cjelokupan proces ovog segmenta okvira prikazan je na *slici 5-19*. (na kraju poglavlja). Sažeti prikaz ovog segmenta integracijskog okvira prikazuje se kroz tablicu 5-1.

Tablica 5-1. Sažetak segmenta razvoja metodološkog okvira – identificiranje uzoraka

Korak okvira ŠTO?	Identificiranje uzoraka
Ljudi TKO?	- Zajednička aktivnost svih članova: ▪ tima za razvoj uzoraka ▪ tima za MDD razvoj SW
Faza KADA?	- Kontinuirano u svim fazama razvoja SW
Aktivnosti KAKO?	- otkrivanje i lociranje uzoraka - kandidiranje problema (i rješenja) za uzorak - inicijalna analiza isplativosti razvoja novog uzorka
Alati ČIME?	-
Standardi	-
Rezultat Artefakti	- antiuzorci - zahtjevi razvoja novog uzorka

5.3.2. Primjena uzoraka

Svrha: Definirati način primjene postojećih uzoraka prikladnih za MDD razvoj.

Primjena prikladnog uzorka definirana je slijedećim aktivnostima:

- uvoz uzorka iz kataloga i instalacija u razvojnom okruženju,
- primjena tijekom modeliranja,
- primjena tijekom implementacije i
- definiranje povratnih informacija.

5.3.2.1. Uvoz uzorka iz kataloga i instalacija u razvojnom okruženju

Kako su u točci 5.1. definirani uvjeti prema kojima, uzorak mora biti definiran kao standardiziran, ponovno iskoristiv resurs primjenom RAS standarda, a razvojno okruženje prikladno za MDD razvoj, s mogućnosti uvoza tako definiranog resursa (datoteke ekstenzije .ras), ovaj korak predstavlja inicijalnu pripremu za primjenu uzorka u kontekstu MDD paradigme. Neposredno nakon uvoza i instalacije uzorka, poželjno je napraviti i inicijalno testiranje kako bi se uvjerili da je uzorak ispravo otpakiran i instaliran u razvojno okruženje. Opcionalno, kako bi se prije same primjene dobio detaljniji uvid u uzorak, kroz razvojno

okruženje može se pregledati detaljnija dokumentacija instaliranog uzorka. Posebnu pažnju potrebno je posvetiti odabiru razvojnog okruženja koje mora podržavati osnovne karakteristike MDD razvoja.

5.3.2.2. Primjena tijekom modeliranja

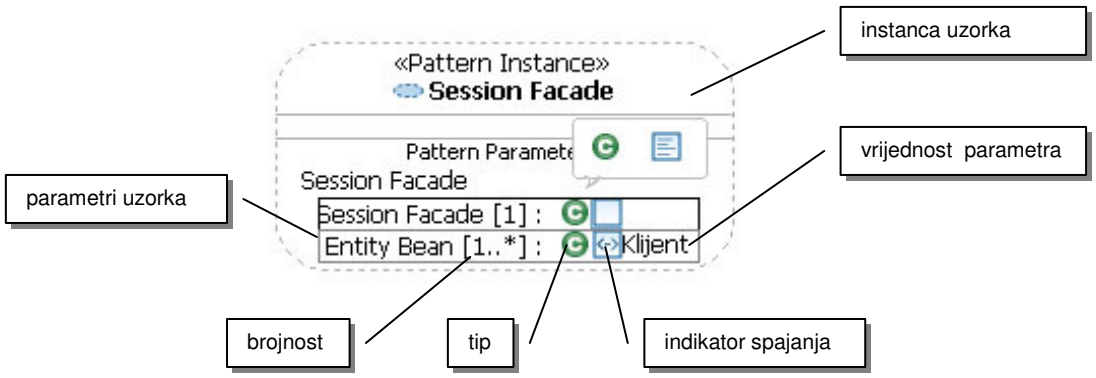
Ova aktivnost predstavlja razvoj modela koji mora biti oblikovan na način prikladan za provođenje određenih oblika automatizacije. Razvoj takvog modela podrazumijeva primjenu standardiziranog jezika za modeliranje (npr. UML).

Proces primjene uzorka tijekom razvoja modela podijeljen je na slijedeće korake:

- instanciranje uzorka,
- povezivanje instance uzorka s elementima modela,

5.3.2.2.1. Instanciranje uzorka

Instanciranje uzorka podrazumijeva "prebacivanje" uzorka u UML model koji se trenutno razvija (najčešće je to model klasa). Tom prilikom, uzorak postaje UML element - *instanca* željenog uzorka te poprima vizualnu reprezentaciju. Primjer instance jednog uzorka (iz *Enterprise Pattern* (EP) *kataloga*) zajedno sa svim elementima koje sadrži svaka instanca prikazan je na *slici 5-3*.



Slika 5-3. Primjer instance uzorka, njezinih elemenata i povezivanja parametra instance s elementom modela

Instanca uzorka prikazana je UML konceptom suradnje (eng. *collaboration*), a elementi koje sadrži nazivaju se **parametri uzorka**. Svaki parametar sadrži *naziv*, *brojnost*, *tip* i *vrijednost*. *Tipovi* su definirani pojedinim UML konceptima i najčešće su to klasa, sučelje ili

operacija. *Brojnost parametra* određuje koliko je UML koncepta istog tipa moguće spojiti na njega.

5.3.2.2.2. Povezivanje instance uzorka s elementima modela

Instanca uzorka povezuje se s elementima modela (UML konceptima) definiranjem vrijednosti parametra instance uzorka. Pri tome se na instancu mogu spojiti već postojeći elementi modela ili se preko te instance mogu kreirati novi, te će spajanje s parametrom uslijediti automatski. To se postiže "oblačićom" iznad parametra. Povezivanje jednog parametra instance uzorka (iz EP kataloga) s jednim UML elementom (klasom) prikazano je na *slici 5-3*. Na navedenoj slici klasa *Klijent* povezana je s *Entity Bean* parametrom instance uzorka *Session Facade*.

I dok su ova dva koraka vezana isključivo uz modeliranje, u nastavku se prikazuje kako tako definiran model pripremiti za njegovu automatizaciju.

5.3.2.3. Primjena tijekom implementacije

Smisao MDD razvoja je postizanje automatizacije na način da se modeli transformiraju u druge artefakte – primarno programski kod (M2C (*eng.* model to code) transformacija). Kako bi to bilo izvedivo, razvojno okruženje treba imati ili već predefinirane oblike transformacija i mogućnost kreiranja vlastitih transformacija. Oblici transformacije koje trenutno podržava IBM RSA v 6.0.1. razvojno okruženju su:

- UML – C++ transformacija,
- UML – CORBA transformacija,
- UML – EJB transformacija,
- UML – Java transformacija i
- UML – XSD transformacija.

Svaka *definicija transformacije* sastoji se od *pravila transformacije* kojima se definira kako se pojedini element iz izvornog modela, transformira u jedan ili više elementa ciljnog modela. Svim gore definiranim transformacijama zajedničko je da kao izvorni model koriste UML model dizajna (klasa) koji je kreiran u prethodnom koraku.

No, kako pokrenuti automatiziranu implementaciju, tj. transformaciju iz UML modela u programski kod i/ili neku drugu vrstu artefakta?

Proces implementacije podijeljen je na:

- kreiranje i konfiguriranje instance transformacije i
- pokretanje transformacije i dopuna programskim kodom.

5.3.2.3.1. Kreiranje i konfiguriranje instance transformacije

Kako bi se osigurala strojna čitljivost modela, nad modelom je potrebno provesti postupak označavanja (eng. mark up) s detaljima vezanim uz transformaciju modela. Da bi se proveo ovaj nadasve ključan korak, najprije je potrebno kreirati novu instancu transformacije, a zatim je i konfigurirati. Konfiguriranje instance transformacije uključuje određivanje kako osnovnih postavki poput naziva konfiguracije, izvora i ciljnog odredišta (UML modela i tipa programskog koda), tako i definiranja posebnih postavki za transformaciju vezanu uz pojedini tip realizacije. Kako je pristup razvoju iterativan, prirodno je da su model dizajna kao i njegova implementacija podložni izmjenama, te je postizanje kvalitetnog reverzibilnog inženjerstva (eng. Round Trip Engineering) ključan čimbenik MDD razvoja. Stoga je u konfiguriranju transformacije

5.3.2.3.2. Pokretanje transformacije i dopuna programskim kodom

Ukoliko su dobro definirane sve postavke transformacije, pokretanjem transformacije generira se određeni broj datoteka koje predstavljaju odabranu implementaciju UML modela. Kako za sada nije moguće 100%-no generiranje programskog koda iz modela, preostaje završni korak, a to je dopuna programskog koda (programske logike) u dijelu generiranih klasa označenih sa `TODO` (ili nekom sličnom ključnom riječi) komentarom. Kako se transformacija može pokretati *n* puta, ručno definirani kod mora se označiti posebnim tagom (oznakom) kako bi se spriječilo njegovo brisanje prilikom ponovnog pokretanja.

Nakon zadnje iteracije u implementaciji, preostaje isporuka (eng deploy) dobivene aplikacije na aplikacijski server i njezino pokretanje ili instalacija kod klijenta.

5.3.2.4. Definiranje povratnih informacija

Iako posljednji, ovo je vrlo važan korak u primjeni uzoraka, jer se njime dobiva praktično iskustvo koje obuhvaća informacije o:

- općem dojmu primjene (verziji uzorka, njegove korisnosti, jednostavnosti pronalaska i primijene, postojanju dvojbi ili zabrinutosti vezanih uz primjenu),
- greškama i nelogičnostima koje je potrebno korigirati,
- prikladnosti ponovne primjene (eng. reuse) uzorka,
- mogućnostima poboljšanja i
- kandidiranju novih uzoraka.

Ovim korakom poželjno je proizvesti *artefakte* u obliku standardiziranih predložaka za dokumentiranje gore navedenih informacija (naprimjer: Zahtjev za ispravljanje pogrešaka, Zahtjev za promjenama, Zahtjev za poboljšanjima, Kandidiranje novog uzorka i sl). Dobivene informacije predstavljat će kritičnu točku na temelju koje će se donositi nove odluke u segmentu *Upravljanja uzorcima*. Iako se ovim korakom predlaže skup predložaka s detaljnim opisom, u praksi tim vrlo često neće biti u mogućnosti pisati toliki broj dokumenata. Stoga se u nastavku prikazuje samo jedan potencijalni opći predložak kojim bi se obuhvatilo sve najvažnije povratne informacije.

Tablica 5-2. Izgled predloška za opis najvažnijih povratnih informacija.

Predložak za definiranje povratnih informacija o primijenjenom uzorku	
Naziv tipa kataloga uzorka	
Autor i naziv kataloga uzorka	
Naziv uzorka	
Verzija	
Jednostavnost pronalaska i primjene	
Prikladnost ponovnog korištenja	
Nelogičnosti i problemi	
Greške koje je potrebno ispraviti	
Mogućnosti poboljšanja	
Kandidiranje novog uzorka	

Na kraju napomenimo da se korak može provoditi na kraju faze ili paralelno s definiranim aktivnostima.

5.3.2.5. Primjer primjene uzoraka na temelju razvijenih koraka

Kroz ovu točku prikazati će se praktična primjena definiranih koraka primjene. Parametri koje primjer obuhvaća su slijedeći:

- Segment razvoja obuhvaća faze razvoja: *dizajn i implementacija*.
- Odabrani tip kataloga uzoraka: *Uzorci oblikovanja*.
- Odabrani katalog uzoraka: *Enterprise patterns*.
- Odabrano razvojno okruženje prikladno za MDD razvoj: *IBM Rational Software Architect v6.0.1* (RSA).

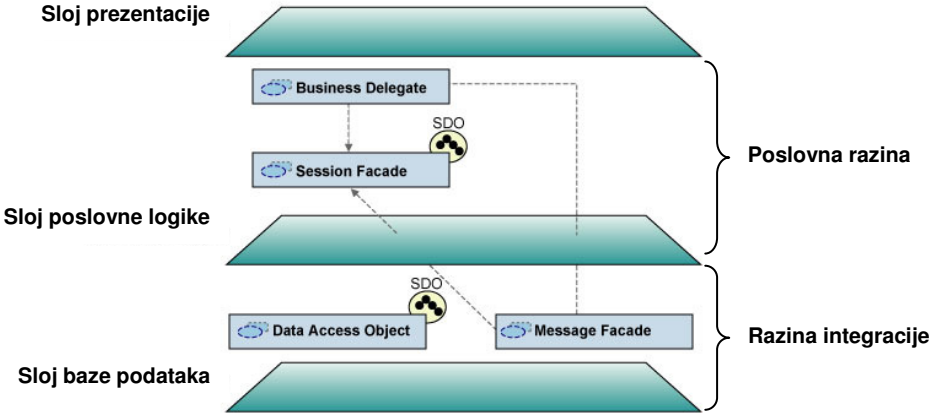
U nastavku slijedi sažet opis odabranog kataloga, a zatim njegova primjenjivost kroz definirane korake.

5.3.2.5.1. Opis Enterprise patterns kataloga

Kako postoji više kataloga koji se mogu svrstati u grupu (ili tip kataloga) *uzoraka oblikovanja* (GOF, J2EE, SOA i drugi), izdvojen je samo jedan katalog pod nazivom *Enterprise Patterns* (EP). *Enterprise Patterns* katalog namijenjen je oblikovanju i implementaciji J2EE aplikacija te uključuje četiri uzorka:

- Business Delegate (BD),
- Session Facade (SF),
- Message Facade (MF) i
- Data Access Object (DAO).

Ovi uzorci primjenjivi su prilikom: *modeliranja* za kreiranje UML modela oblikovanja i *implementacije* J2EE aplikacija tj. generiranja programskog koda na temelju UML modela primjenom UML-EJB transformacije čime se postiže automatizacija implementacije. Koncept kataloga prikazan je na *slici 5-4*.



Slika 5-4: Koncept *Enterprise patterns* kataloga

Katalog je podijeljen u dvije razine: *poslovna razina* i *razina integracije*. U poslovnoj razini nalaze se uzorci *Session Facade* i *Business Delegate*, a u razini integracije *Message Facade* i *Data Access Object*.

Uzorci u poslovnoj razini pojednostavljuju pristup poslovnim servisima i to su:

***Session Facade*:** učahuruje pristup skupu udaljenih poslovnih servisa i poslovnih objekata kao što su *entity beans*. Uzorak osigurava CRUD metode, ali i pruža mogućnost dodavanja vlastitih metoda.

***Business Delegate*:** odvaja klijentsku stranu sustava od poslovnih servisa. Karakteristike poslovnih servisa vezane uz odabranu tehnologiju sakrivene su od

klijenta. Klijentov pogled na poslovni servis realiziran je kao standardizirani java (POJO) pogled. Takav pogled omogućava klijentu fokusiranje na semantiku poslovnog servisa, a ne na sintaksu.

Uzorci u razini integracije pojednostavljuju pristup podacima i omogućuju asinkroni pristup poslovnim servisima i to su:

***Message Facade*:** koristi se za asinkroni poziv udaljenih servisa koji funkcioniraju sinkrono. Uzorak je implementiran kao *message-driven bean* (MDB). Kreirajući Java Message Service (JMS) poruku, klijent slobodno nastavlja s radom umjesto da čeka odgovor.

***Data Access Object*:** osigurava API kojim se održavaju podaci zadržani u privremenom bufferu. Predstavlja alternativu rješenju koje zahtjeva ili EJB komponentu ili privremenu pohranu sa c*ontainer-managed persistence (CMP) entity beans*. Koristi se kada se pristupa podacima koji nisu tablice u bazi podataka tj. kada su u pitanju alternativni izvori podataka npr. tekstualna datoteka ili kada je potrebno pristupati bazi sa složenim upitima.

Neophodno je napomenuti kako su tri od četiri navedena uzorka međusobno povezana, tako da funkcioniraju i razmjenjuju parametre međusobno, što i proizlazi iz *slike 5-4*.

EP je smješten u IBM *developerWorks* repozitoriju što ukazuje da je definiran kao ponovno iskoristivi resurs RAS standardom, te zadovoljava prethodno definirane uvijete primjene.

Detaljna specifikacija cijelog kataloga može se pronaći u prilogu **A**.

5.3.2.5.2. Primjena Enterprise Patterns kataloga kroz razvojno okruženje IBM Rational Software Architect v6.0.1.

Kao što se i na temelju definiranih koraka može zaključiti, razmatranje primjene odabranog kataloga u razvojnom okruženju RSA, podijeljeno je na:

- Uvoz uzorka iz kataloga i instalacija u razvojnom okruženju.
- Primjena tijekom modeliranja.
  - instanciranje uzorka i
  - povezivanje instance uzorka s elementima modela.
- Primjena tijekom implementacije.
  - kreiranje i konfiguriranje instance transformacije i
  - pokretanje transformacije i dopuna programskim kodom
- Definiranje povratnih informacija.

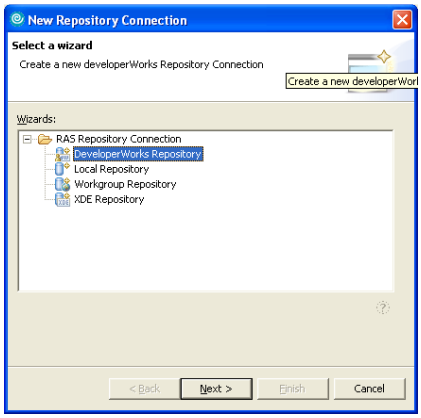
Uvoz uzorka iz kataloga i instalacija u razvojnom okruženju

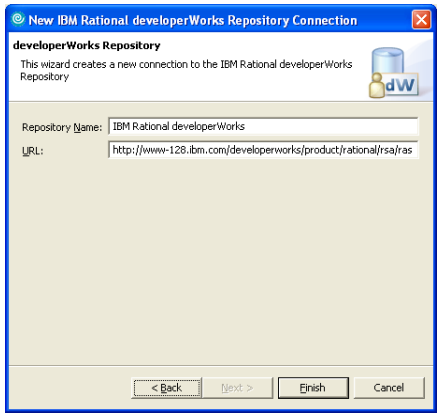
EP katalog smješten je u IBM *developerWorks* repozitoriju te ga je najprije potrebno uvesti u RSA razvojno okruženje, a zatim i instalirati. Postupak se može opisati kroz slijedeće korake:

Iz *RAS* prespektive preko *Asset Explorera* definira se nova konekcija prema *developerWorks* repozitoriju. Struktura repozitorija se učitava u RSA okolinu na temelju koje se odabire katalog koji se želi učitati u alat. Slijedi učitavanje datoteka iz repozitorija i njihova instalacija. Nakon instalacije potrebno je preklopiti na perspektivu *Modeling* kako bi katalog postao vidljiv u *Pattern Exploreru*.

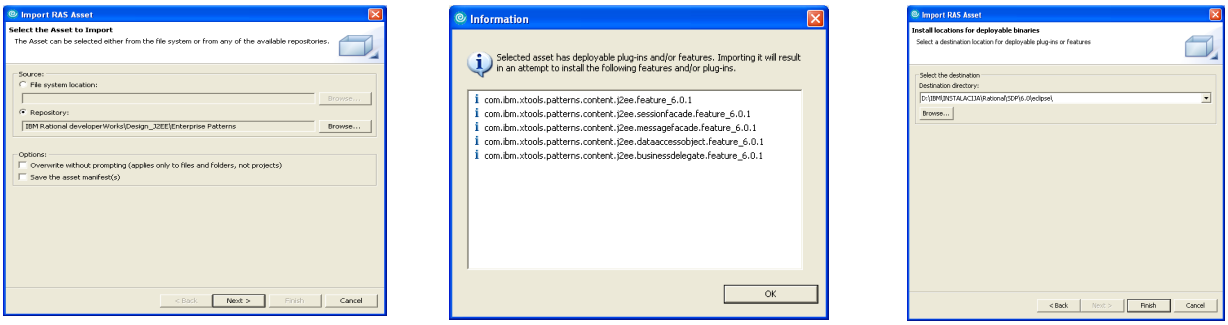
Opisani koraci mogu se vizualno pratiti kroz ekrane prikazane na *slici 5-5*.

Korak 1: Definiranje konekcije prema *developerWorks* repozitoriju.

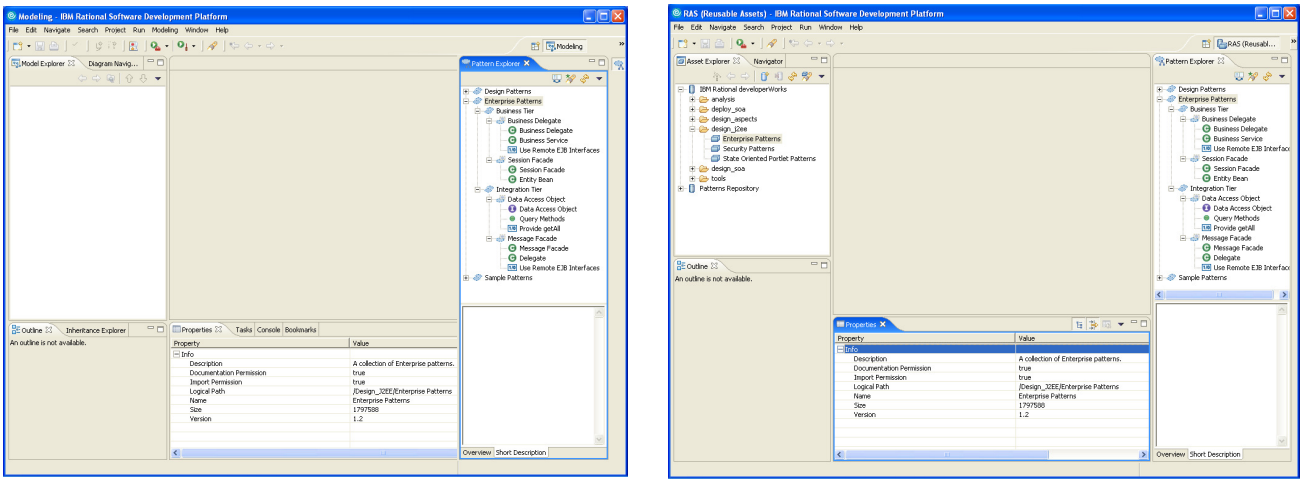




Korak 2: Uvoz i instalacija odabranog kataloga iz *developerWorks* repozitorija.



Korak 3: Izgled razvojnog okruženja iz *Modeling* i *RAS* perspektive.

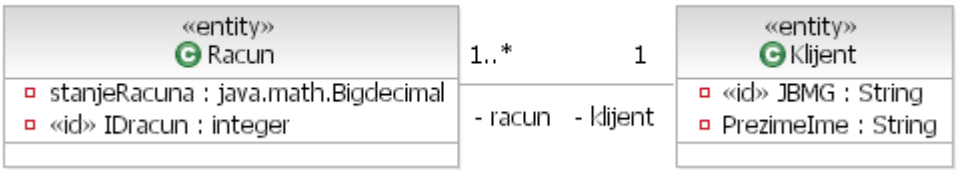


Slika 5-5. Koraci uvoza i instalacije kataloga u RSA.

Primjena tijekom modeliranja

Primjer nad kojim će se primijeniti EP katalog, predstavlja dio modela bankarskog IS-a. Poslovni objekti (klase) koji se pojavljuju u modelu biti će realizirani kao *entity beans* (stoga ih je potrebno stereotipizirati u <<entity>> oblik)* kako bi se tijekom implementacije pokazala mogućnost generiranja programskog koda na temelju definirane UML-EJB transformacije. Model je prikazan na *slici 5-6*.

* Stereotipizacija je nužna kako bi se znalo koje je tipove bean-ova potrebno kreirati prilikom transformacije.



Slika 5-6. Pojednostavljeni dio modela s dva *entity beans,* njihovim atributima i vezama.

Kako bi se definirao *model dizajna* za J2EE aplikaciju, nad gornjim modelom, primijenit će se uzorci iz EP kataloga. Izrada modela uključuje: instanciranje uzoraka u model i povezivanje svake instance uzorka s elementima modela.

- **Instanciranje uzorka**

  Iz *Pattern Explorera* u koji je prethodno instaliran cijeli EP katalog odabire se odgovarajući uzorak koji se prebacuje u gornji model čime postaje *instanca uzorka*. Povlačenjem uzorka u model (npr. *Session Facade*), nastaje njegova instanca (*slika 5-7).* Parametre uzorka će u sljedećem koraku biti potrebno povezati.

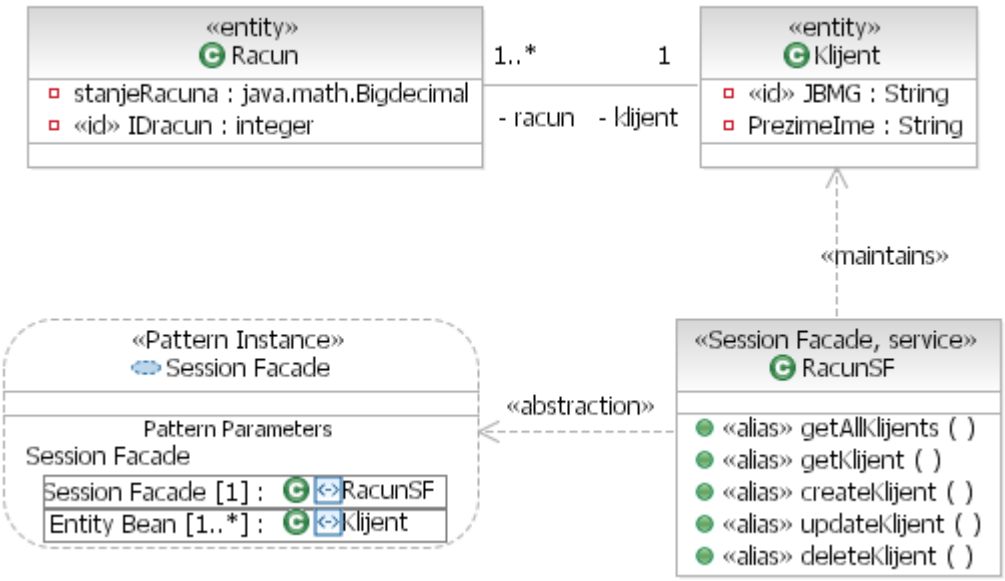
- **Povezivanje instance uzorka s elementima modela**

  Tri od četiri uzorka su povezana, što ukazuje na činjenicu da je redoslijed primjene uzoraka iz EP kataloga značajan. Najprije je potrebno primijeniti ***Session Facade***, zatim ***Business Delegat***e pa ***Message Facade***. Povezivanje se provodi da se parametar uzorka spoji s UML konceptom. U nastvaku slijedi prikaz povezivanja.

Session Facade uzorak

Kako se sada u modelu nalaze dvije stereotipizirane klase <<entity>>*Klijent* i <<entity>>*Račun* i *instanca SF uzorka* moguće se izvršiti povezivanje. Klasa *Klijent* spaja se s parametrom uzorka *Entity Bean* (*slika 5-7*). Za drugi parametar - *Session Facade,* dovoljno je upisati naziv klase, jer će *instanca SF uzorka,* upisivanjem naziva klase sama generirati novu klasu (s tim imenom) i dodijeliti joj CRUD metode klase koja je spojena s prvim parametrom - klasa *Klijent*. Instance će se također pobrinuti da nova kreirana klas bude stereotipizirana oznakom <<*service*>>. Programska logika svih tih aktivnosti učahurenu je u instanci uzorka. Korisnik jedino upisuje naziv klase - *RacunSF*.

Nakon provedenog koraka, na *slici 5-7.* može se vidjeti trenutni oblik modela.

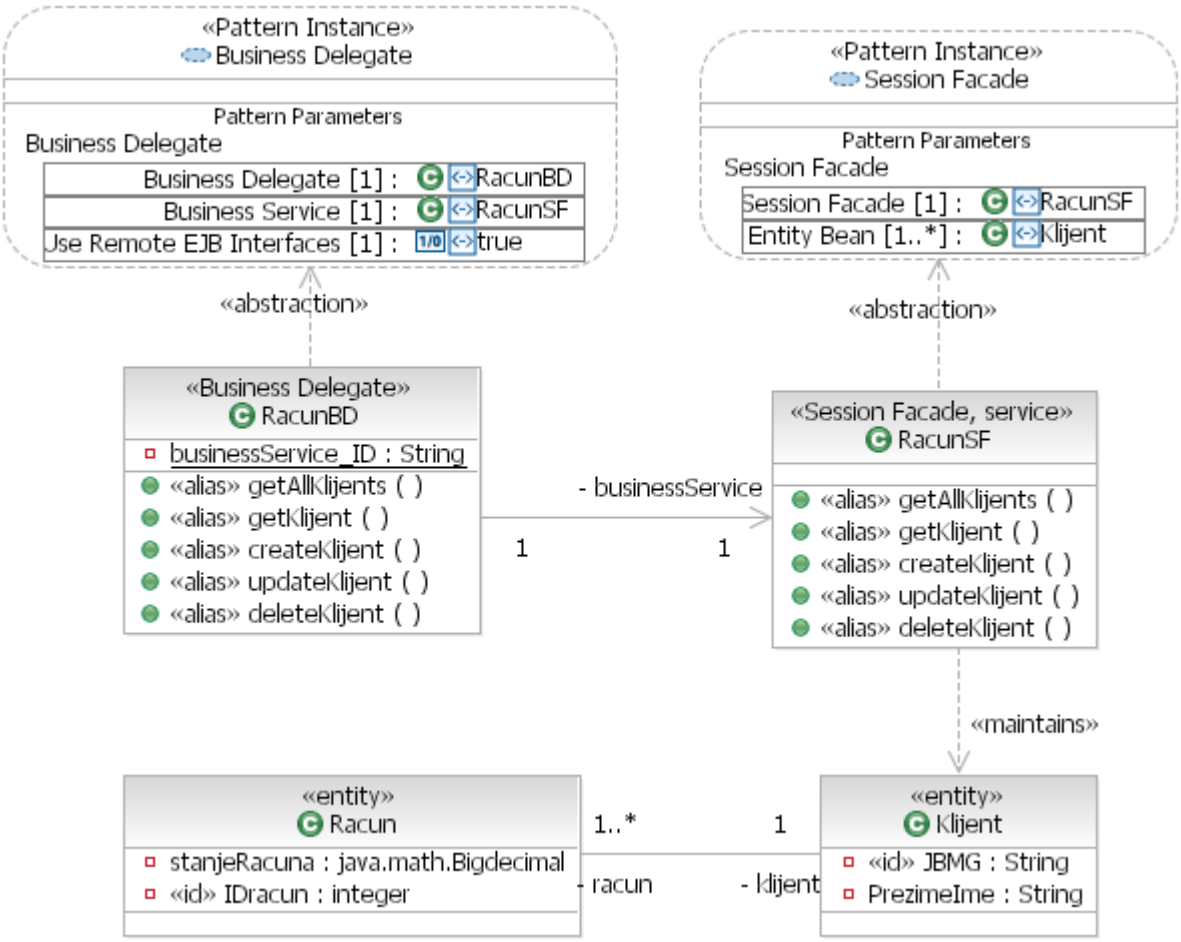


Slika 5-7. Primjena *Session Facade* uzorka

Pripomenimo da bi se prilikom transformacije modela trebala generirati .java datoteke za dva entity i jedan servis bean-a.

Business Delegate uzorak

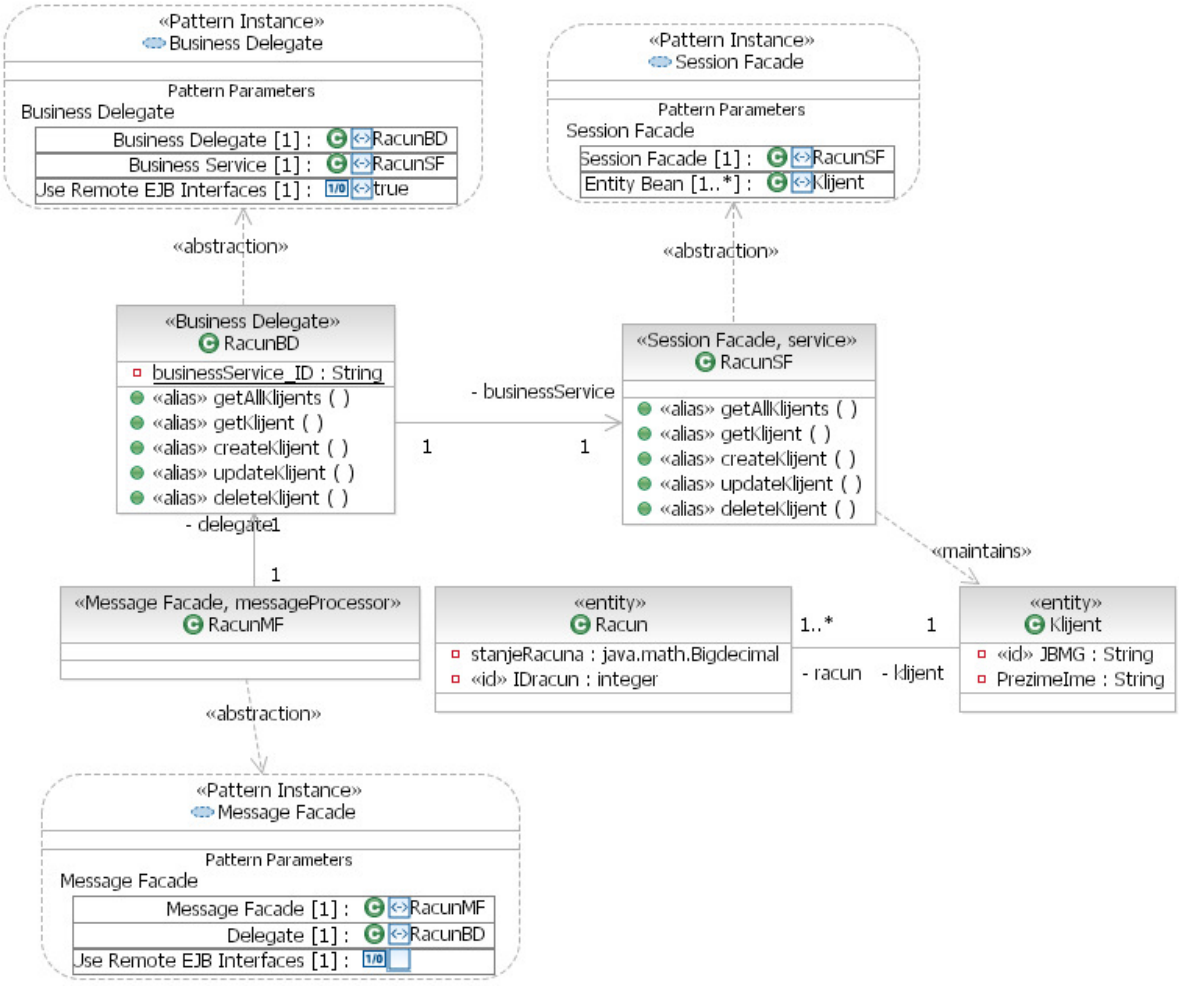
Business Delegate uzorak primjenjuje se kako bi se osigurao POJO pristup *Session Facade*-u. Sam uzorak koristi klasu <<*service*>>*RacunSF* koja je kreirana i koju "održava" prethodni uzorak, te se spaja sa *Business Service* parametrom. Kao i u prethodnom slučaju, instanca BD uzorka, kreira novu klasu pod nazivom *RacunBD* (nije stereotipizirana), a rezultat primjene je prenošenje metoda od servisa <<*service*>>*RacunSF* (stereotipizirana klasa) u klasu *RacunBD* (*slika 5-8*).



Slika 5-8. Primjena *Business Delegate* uzorka

Message Facade uzorak

Ovaj uzorak koristi klasu *RacunBD* nastalu iz prethodnog uzorka i spaja ju s *Delegate* parametrom. Upisivanje naziva klase pod parametrom *Message Facade*, *Instanca MF uzorka* kreirat će stereotipiziranu klasu <<messageProcessor>>*RacunMF*. Poželjno je da primjena MF bude povezana s BD, jer u tom slučaju MF upravlja asinkronom prirodom poruke, a BD upravlja izborom tehnologije kao i potencijalnom udaljenošću poslovnog servisa. Izgled modela nakon primjene ovog uzorka prikazan je na *slici 5-9*.



Slika 5-9. Primjena *Message Facade* uzorka.

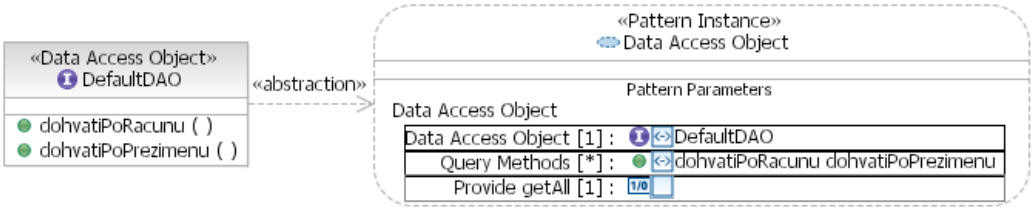
Data Access Object uzorak

Iako je u odabranom primjeru zamišljeno da poslovni objekti *Klijent* i *Racun* budu implementirani kao *entity beans* ukratko spominjem i ovaj uzorak koji predstavlja alternativnu mogućnost implementacije. DAO se prikazuje kao sučelje koje prilikom implementacije koristi drugačiji mehanizam pristupa bazi podataka.

Primjena ovog uzorka dobar je izbor kada:

- se zahtjeva alternativan pristup bazi podataka od *entity beans*. To će biti slučaj kada se vraćeni podaci ne mogu jednostavno prikazati kao *entity beans* (npr. složeni uvjeti spajanja tablica, djelomični rezultati redova tablice, baze podataka koje nisu podržane).
- EJB kontenjeri nisu poželjni

Konceptualno, DAO ima istu ulogu kao entity bean u Session Facade uzorku. Na *slici 5-10.* može se vidjeti vizualna reprezentacija instace DAO uzorka i sučelja s metodama.



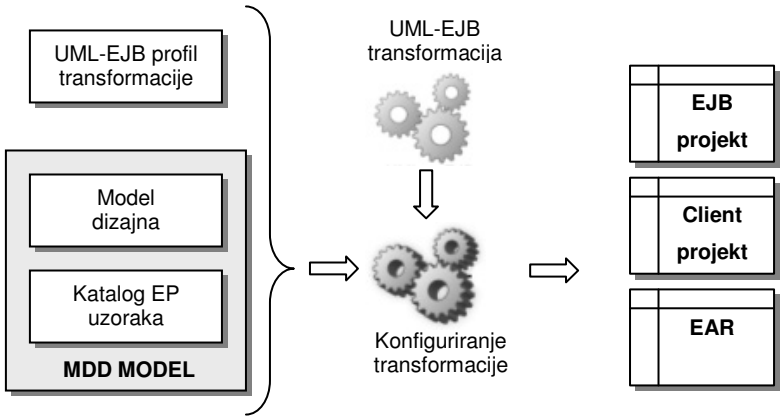
Slika 5-10. Primjena *Data Access Object* uzorka.

Definirani *model dizajna* za J2EE aplikaciju (*slika 5-9.*) čine dvije klase koje su kreirane ručno (Klijent i Račun) i klase koje su kreirane primjenom instanci uzoraka SF, BD i MF. Kroz slijedeći korak ovaj model poslužit će u provođenju automatizirane implementacije.

Primjena tijekom implementacije

Već prilikom samog modeliranja, vodilo se računa o tipu implementacije modela. Kako je odabrana primijena *UML – EJB transformacije*, koja na temelju stereotipova definiranih u pripadajućem profilu, transformira UML elemente modeliranja u Java kod i *enterprise beans*–e, klase su se stereotipizirale. Stoga će se prilikom pokretanja transformacije steteotipizirane klase transformirati u odgovarajuće *bean*-ove; <<service>> → session bean, <<entity>> → entity bean, <<MessageProcessor>> → MDBs. Rezultat provedene transformacije bit će EJB

projekt. Vrlo lijepo to je prikazano shematskim prikazom na *slici 5-11*.



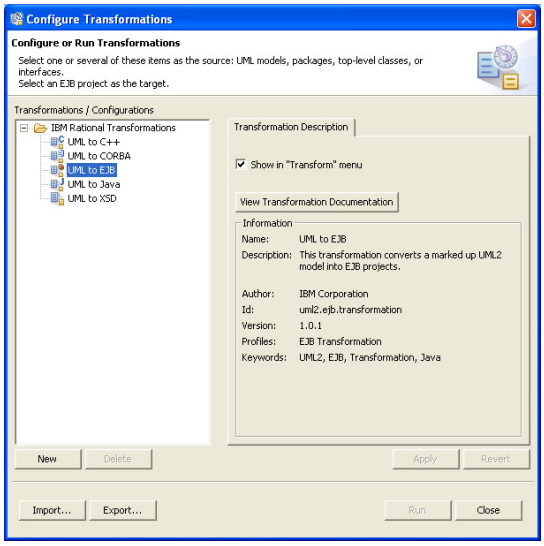
Slika 5-11. Shematski prikaz implementacije MDD modela.

Kako je ovaj korak podijeljen u dvije aktivnosti slijedi detaljnije razmatranje svake od njih.

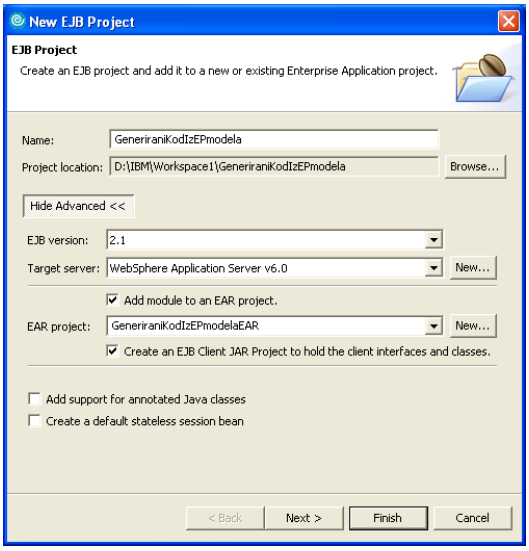
Kreiranje i konfiguriranje instance transformacije

Kao što je već bilo spomenuto, u RSA v6.0.1. moguće je odabrati nekoliko tipova transformacija te detaljno pregledati dokumentaciju svake transformacije (*slika 5-12 (a)*). Odabirom željene transformacije (UML- EJB) kreira se instanca pod nazivom *UML to EJB transformacija*. Konfiguriranje instance provodi se kroz nekoliko ekrana, a uključuje podešavanje parametara kao što su: naziv konfiguracije, izvor (model dizajna) i ciljno odredište u koji će se generirati programski kod (EJB projekt) te posebne postavke za transformaciju *session* i *entity bean*-ova. Ujedno, moguće je uključiti i opciju kojom se osigurava konzistentnost uslijed promjena na modelu ili kodu. Neke od navedenih parametara mogu se vidjeti kroz ekrane na *slici 5-12*.

Odabir tipa transformacije:



Definiranje ciljnog odredišta - kreiranje EJB projekta

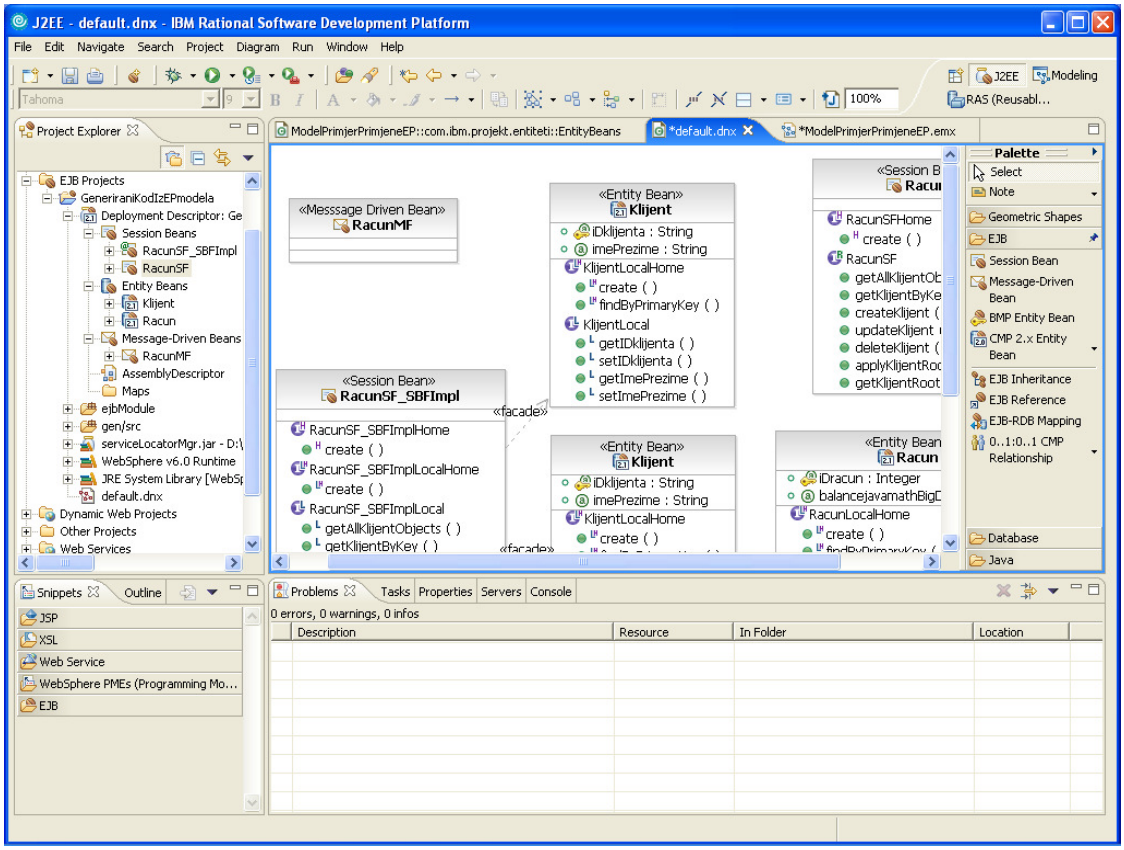


Slika 5-12. Kreiranje i konfiguriranje instance transformacije.

Nakon ovih akcija konfigurirana transformacija može se pokrenuti.

- **Pokretanje transformacije i dopuna programskim kodom**

  Kao što se može vidjeti na slici *5-13.* nakon pokretanja transformacije u ciljnom direktoriju *GeneriraniKodIzEPmodela* generirane su datoteke (.java) koje predstavljaju implementaciju definiranog modela dizajna (entity beans, session beans, message-driven beans) kao i sam vizualni model *default.dnx*.



Slika 5-13. Izgled generiranog projekta.

Prijmjer generiranog koda za npr. entity bean *Klijent* i *RacunSF_SBFImpl.* može se pronaći u **prilogu B.**

Programsku logiku sada je potrebno nadodati ručno. Pri tome je potrebno voditi računa da se transformacija može pokretati *n* put, te da će se svakim novim pokretanjem, ručno definirani kod prebrisati. Postoje dva načina sprječavanja tog scenarija. Dio programskog koda koji se generira započinje oznakom `@generated` pa je tu oznaku potrebno obrisati kako bi se spriječilo prebrisavanje napisanog koda. Drugi način da se dio koda ogradi komentarima `//begin-user-code` i `//end-user-code`.

Dobivenu aplikaciju potrebo je smjestiti na aplikacijski server. U našem primjeru to je WebSphere Application Server v6.0. Tom prilikom usljeđuje fizičko kreiranje tablica u bazi podataka kao i mogućnost testiranja dobivene aplikacije pokretanjem testnog okruženja.

Definiranje povratnih informacija

Za svaki korišteni uzorak na kraju je potrebno kreirati (vrlo često samo jedan) artefakt u obliku .doc ili .xls napisan na temelju predloženog predloška kojim bi se opisale različite povratne informacije. Primjer za ***Session Facade*** uzorak prikazan je u nastavku:

Tablica 5-3. Primjer predloška za opis povratnih informacija i primjeni uzorka

Predložak za definiranje povratnih informacija o primijenjenom uzorku	
Naziv tipa kataloga uzorka	UZORCI OBLIKOVANJA
Autor i naziv kataloga uzorka	IBM Corporation // ENTERPISE PATTERNS
Naziv uzorka	***Session Facade***
Verzija	1.0.0
Jednostavnost pronalaska i primjene	Intuitivna primjena.
Prikladnost ponovnog korištenja	Poželjno prilikom novih projekata.
Nelogičnosti i problemi	Nema
Greške koje je potrebno ispraviti	Nema
Mogućnosti poboljšanja	Nema
Kandidiranje novog uzorka	Nema

Ovime završava primjer primjene EP kataloga uzorka.

Aktivnosti provedene u segmentu primjene razvijenog okvir grafički su prikazana na *slici 5-19*. Ovaj segment razvoja metodološkog okvira zaključujem sažetim prikazom opisanih aktivnosti kroz *tablicu 5-4*.

Tablica 5-4. Sažetak segmenta razvoja metodološkog okvira – primjena uzoraka

Korak okvira ŠTO?	**Primjena uzoraka**
Ljudi TKO?	- Tim za razvoj SW
Faza KADA?	- Bilo koja faza u razvoju SW
Aktivnosti KAKO?	- Uvoz uzorka iz kataloga i instalacija u razvojnom okruženju - Primjena tijekom modeliranja ▪ Instanciranje uzorka ▪ Povezivanje instance uzorka s elementima modela - Primjena tijekom implementacije ▪ Kreiranje i konfiguriranje instance transformacije ▪ Pokretanje transformacije i dopuna programskim kodom - Definiranje povratnih informacija
Alati ČIME?	- RSA - Aplication server (WS Proces Server)

Standardi	- UML
Rezultat **Artefakti**	- model dizajna - generiran projekt s implementacijom klasa - predložak(ci) za definiranje povratnih informacija o primijenjenom uzorku

5.3.3. Razvoj novog uzorka

Svrha: Kreirati novi uzorak prikladan za MDD razvoj u obliku ponovno iskoristivog resursa koji će biti primjenjiv u *n* projekata u cilju smanjenja troškova, povećanja produktivnosti i skraćivanje vremena razvoja.

Na temelju *zahtjeva razvoja uzorka* (prikupljene dokumentacije koju čini opis problema, inicijalne analize iskoristivosti razvoja novog uzorka ili nekog drugog dodatnog dokumenta (npr. dijagram slučaja korištenja kojim je opisano lociranje ili primjena, ideja potencijalnih rješenja)) usljeđuje:

- pokretanje projekta razvoja novog uzorka,
- realizacija razvoja te
- organizacija uzorka u ponovno iskoristiv resurs.

Nakon objašnjenja sva 3 segmenta razvoja, prikazat će se i primjer razvoja jednog uzorka.

5.3.3.1. Pokretanje projekta razvoja novog uzorka

Upravljanje projektom (ima) razvoja novog uzorka

Definiranjem projekta razvoja novog uzorka određuje se niz aktivnosti koje su karakteristične za sve projekte razvoja softvera, poput: tima koji će sudjelovati u njegovoj realizaciji (voditelj, članovi, zadaci i aktivnosti), budžeta, vremena, razvojnog okruženja i procesa razvoja pa ih je tako potrebno obuhvatiti i kada se radi o razvoju uzorka. Cilj ovog koraka je definiranje opsega i okruženja projekta razvoja uzorka s detaljnom analizom realizacije projekta kako bi se kasnije što kvalitetnije upravljalo njime (ali i drugim projektima razvoja uzorka). Rezultat ovog koraka je definirani projekt s *planom razvoja novog uzorka*, i predstavlja ishodišnu točku za segment ***Razvoja***.

5.3.3.2. Realizacija razvoja

Prilikom razvoja uzorka ključna komponenta svakako je proces razvoja uzorka. U pravilu, taj proces bit će iterativni *"home made"* proces koji predstavlja rezultat uočenih najboljih praksi iz dosadašnjih projekata razvoja uzoraka, koji svakim novim uzorkom evoluira. Moguće je taj proces kombinirati i s nekim postojećim procesom razvoja koji predstavlja "najbolju praksu". Faze koje se ovdje mogu identificirati u pravilu se ne razliku od klasičnih faza razvoja SW. Stoga predlažem realizaciju razvoja novog uzorka kroz slijedeće korake:

- analiza,
- definiranje arhitekture rješenja – dizajn uzorka,
- implementacija – konstrukcija uzorka i
- validiranje i verificiranje – testiranje uzorka.

5.3.3.2.1. Analiza

Na temelju *zahtjeva razvoja uzorka* potrebno je jasno definirati *što* će uzorak raditi - za koji problem je uzorak namijenjen i što će biti njegovo rješenje.

5.3.3.2.2. Definiranje arhitekture rješenja – dizajn uzorka

Ključne aktivnosti u okviru ovog koraka obuhvaćaju:

Oblikovanje rješenja: Nakon što se osmisli rješenje definiranog problema, potrebno je na jasan i nedvosmislen način definirati *kako* će se uzorak razviti tj. kako će se moći realizirati u implementaciji, a da su pri tome zadovoljeni svi kriteriji MDD paradigme i definicije uzorka. Najvažnija aktivnost svakako je *određivanje parametara* uzorka, što uključuje definiranje njihovog broja, tipa, brojnosti i eventualne međusobne povezanosti. Ujedno je potrebno razmisliti i o postojanju eventualnih ograničenja za pojedine parametre. U svrhu povećanja razumijevanja realizacije uzorka poželjno je kreirati i neku vrstu konceptualnog UML dijagrama (dijagram klasa). Ovaj će korak sigurno imati nekoliko iteracija prije nego poprimi formu rješenja koje će se implementirati u nastavku.

Specificiranje uzorka (dokumentiranje): kako je primaran cilj MDD paradigme automatizacija, upravo se aktivnosti, koje su provedene u prethodnom koraku, žele automatizirati na način da se na temelju njih generira cjelokupna dokumentacije (.doc, .html) koja će činiti specifikaciju uzorka. Znači, dokumentacija je artefakt koji nastaje razvojem, a ne nakon njega.

Pisanje testova: Završni korak svakako bi trebao obuhvaćati i definiranje testnih skripata kako bi se prije same organizacije uzorka, kao ponovno iskoristivog resursa, one izvršile te se time provjerila uspješnost implementacije uzorka.

5.3.3.2.3. Implementacija – konstrukcija uzorka

Nakon što se kroz prethodni korak definiraju osnovni elementi budućeg uzorka usljeđuje njegova implementacija. Pod tim je korakom zamišljeno dodavanje "ponašanja" kojim će se opisati što će se dogoditi kada se uzorku doda ili ukloni vrijednost svakog parametra uzorka. Zapravo to je *znanje* koje je potrebno učahuriti u uzorak, jer ono predstavlja onaj dio posla koji programer svaki puta ponovno ručno mora napisati i koji se želi automatizirati. Na temelju specificiranih osnovnih elemenata uzorka, razvojno okruženje trebalo bi generirati kostur programskog koda kojeg čine klase (parametri) s metodama dodavanja i uklanjanja vrijednosti parametara kao i metodama kojima se opisuje i eventualna povezanost dva parametra. U svaku metodu upisuje se programska logika u programskom jeziku odabranog razvojnog okruženja.

5.3.3.2.4. Validiranje i verificiranje – testiranje uzorka

Testiranje predstavlja vrlo važnu komponentu razvoja kojom se procjenjuje kvaliteta implementacije te ju je nemoguće preskočiti. Kako je osnovna namjena svakog razvijenog uzorka, ponovna iskoristivost i automatizacija, zahtjeva se detaljan i studiozan pristup ispitivanja: da li uzorak ispunjava očekivanja - validiranje (rješava li problem?) i da li se uzorak *"ponaša"* na način na koji se to očekuje na temelju definirane specifikacije - verificiranje. Testiranjem bi svakako trebao provesti provjeravanje testnih skripata napisanih prilikom dizajna uzorka kako bi se pronašle i opisale greške te provjerila i kroz konkretnu primjenu dokazala ispravnost razvijenog uzorka. Rezultat provedenih aktivnosti su povratne informacije koje razvojnom timu interno osiguravaju informacije važne za unapređenja u razvoju uzorku.

5.3.3.2.5. Organizacija uzorka u ponovno iskoristiv resurs

Za uspješno razvijen te testiran uzorak preostaju dodatne aktivnosti koje su karakteristične za MDD razvoj, a to je da tako razvijen uzorak bude definiran kao ponovno iskoristivi resurs.

Aktivnosti koje je potrebo provesti su:

- standardizacija i pakiranje te
- objavljivanje u repozitorij.

5.3.3.2.6. Standardizacija i pakiranje

Za standardizaciju i pakiranje ponovno iskoristivih resursa, a našem slučaju uzorka, koristi se RAS standard. O RAS standardu detaljno se govorilo u točci 4.8.1. Napomenimo samo kako svaki ponovno iskoristivi resurs mora sadržavati jednu manifest datoteku i više artefakta. Primjenom RAS standarda osigurava se infrastruktura za ponovnu iskoristivost.

Smisao pakiranja je objedinjavanje svih artefakata (dokumentacije, modela, koda) koji će ponovno iskoristiv resurs činiti upotrebljivim od pretraživanja do primjene. Paket (ponovno iskoristivi resurs) je definiran verzijom i ekstenzije je *.ras*.

5.3.3.2.7. Objavljivanje u repozitorij

Predstavlja korak u kojem se dobiveni resurs (.ras) smještava u repozitorij čime započinje njegova široka primjena.

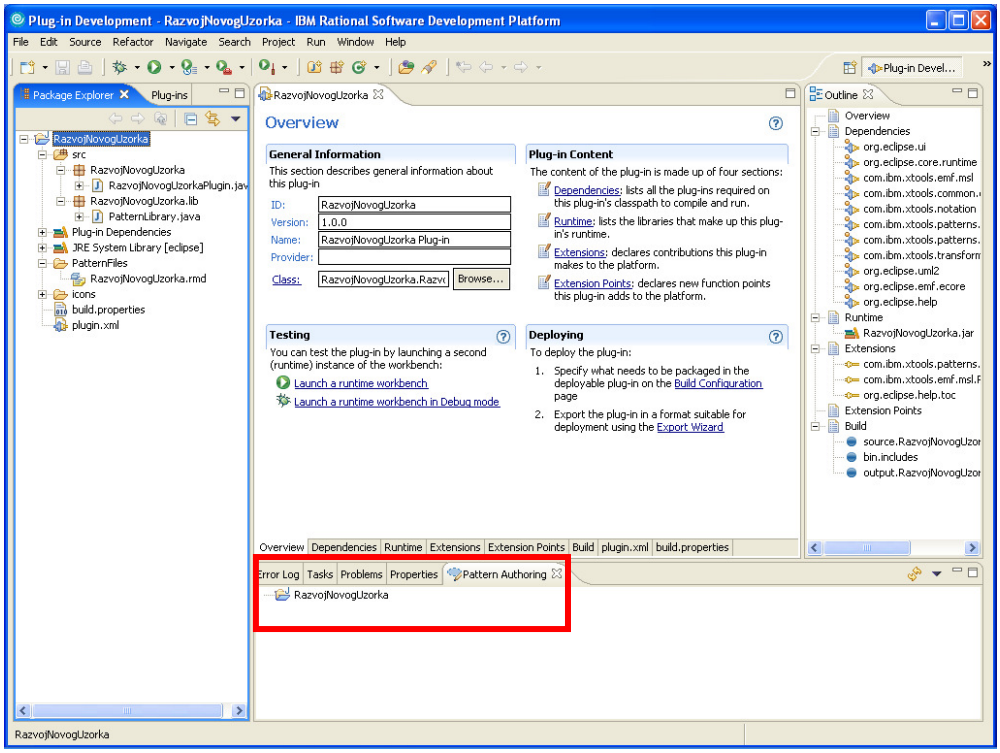
5.3.3.3. Primjer razvoja novog uzorka

Na temelju prethodno opisanih razvijenih koraka kroz razvojno okruženje IBM Rational Software Architect v6.0.1. slijedi primjer razvoja novog uzorka.

Pokretanje projekta razvoja novog uzorka

Razvoj je dodijeljen jednom članu tima koji će slijediti korake razvijenog metodološkog okvira.

Slijedi pokretanje projekta u RSA razvojnom okruženju te se nizom koraka s *čarobnjakom za razvoj uzoraka* kreira okruženje za njegov razvoj. Uključivanje Plug-in Development perspektive, odabir Plug-in projekta te kreiranje novog pod nazivom *RazvojNovogUzorka* te pokretanje čarobnjaka za razvoj uzorka. Izgled ovako kreiranog projekta vidi se na *slici 5-14*.



Slika 5-14. Izgled inicijalno definiranog projekta razvoja uzorka.

Realizacija razvoja

- **Analiza**

Kako je identificirana potreba za novim uzorkom te su oblikovani zahtjevi razvoja novog uzorka jasno je definirano *što* će uzorak raditi. Novi uzorak *svakoj će klasi, za svaki njezin odabrani atribut, nadodati operacije get i set.*

- **Definiranje arhitekture rješenja – dizajn uzorka**

Oblikovanje rješenja:

Kreira se uzorak, pod nazivom *NoviUzorak*, koji će imati dva parametra naziva **Klasa** i **Atributi**. Parametar **Klasa**, koja je tipa *class* s brojnošću *1*, definira se ciljna klasa u koju će uzorak dodavati željene metode. Drugi parametar **Atributi**, je tipa *property* s brojnošću *0..n* kroz koji se definiraju atributi za koje se želi generirati metode *get* i *set*. Ujedno uzorak mora imati opisanu programsku logiku – "ponašanje" prilikom dodavanja ili uklanjanja vrijednosti parametra. Kako je postignuto razumijevanja rješenja, nije potrebno crtati dodatne dijagrame.

Specificiranje uzroka (dokumentiranje):

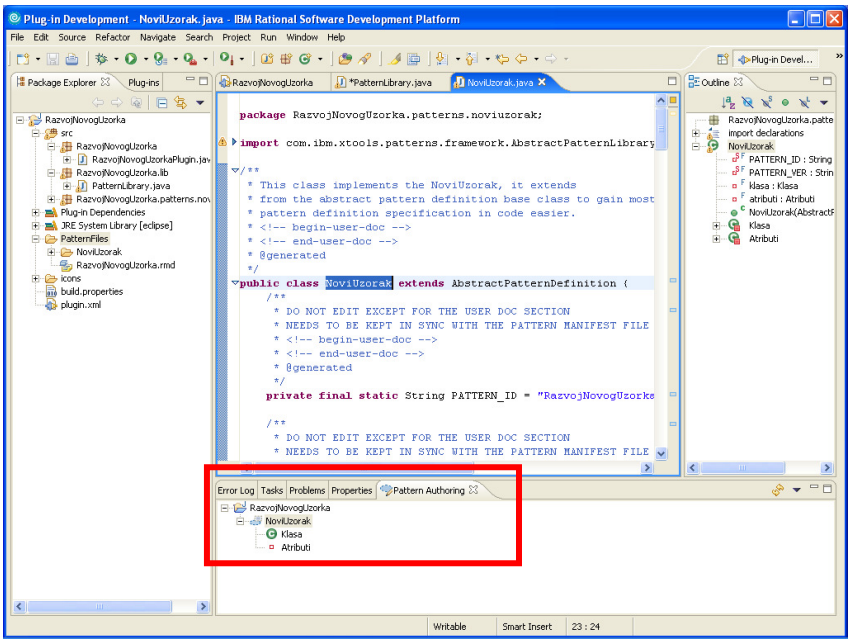
Kako alat ima mogućnost generiranja cjelokupne specifikacije u direktorij *PatternHelp* (skup .html datoteka vidljivih u pogledu *Package Explorer*) nakon konstrukcije pokrenut će se generiranje dokumentacije.

Pisanje testova:

Test se može napisati kroz razvojno okruženje IBM Rational Manual Tester i IBM Rational Funcitional Tester. Ova aktivnost ovdje se posebno nije razmatrala.

- **Implementacija – konstrukcija uzorka**

Implementacija je prikazana na *slici 5-15.*



Slika 5-15. Izgled implementacije uzorka.

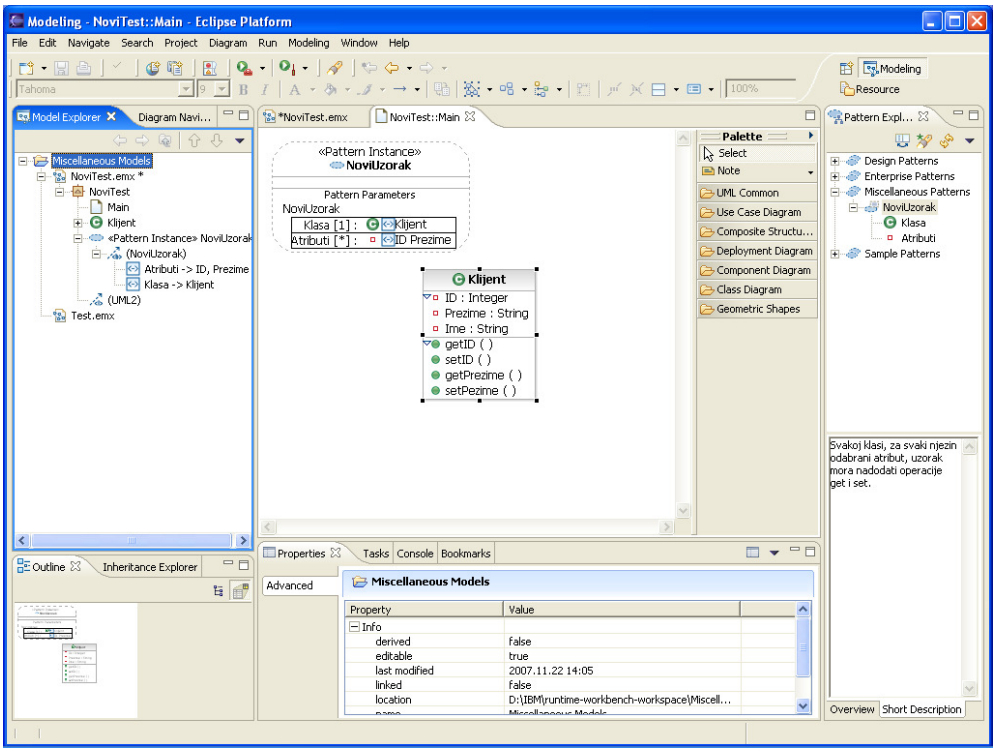
Uz *PatternLibrary.java* datoteku, RSA generira i predložak java koda, pod nazivom *NoviUzorak.java*, s već definiranim metodama za dodavanje ili uklanjanje vrijednosti parametra u koje je potrebno opisati "ponašanje" uzorka.

Programski kod datoteka može se pronaći u prilogu **C**.

- **Validiranje i verificiranje – testiranje uzorka**

Kako je testiranje razvijenog uzorka nezaobilazni korak, RSA razvojno okruženje prilikom kreiranja projekta kreira komponentu pod nazivom *Lunch a runtime workbench* (vidi *sliku 5-14.*) preko koje će se izvršiti testiranje. Preklapanjem na *Modeling* perspektivu i uključivanjem *Pattern Explorera* može se pronaći upravo razvijeni uzorak *NoviUzorak*. Slijedi kreiranje UML

modela klasa i primjena uzorka. Način primjene već je opisan kroz točku 5.3.2. U modelu se kreira nova klasa *Klijent* s nekoliko atributa (ID, Prezime, Ime) te se instancira uzorak. Zatim usljeđuje pridruživanje vrijednosti parametara na što uzorak izvršava definirano "ponašanje" – dodaje *get* i *set* metode za odabrane atribute u klasi *Klijent*. *Slika 5-16.* prikazuje opisan scenarij.



Slika 5-16. Izgled testiranja implementiranog uzorka.

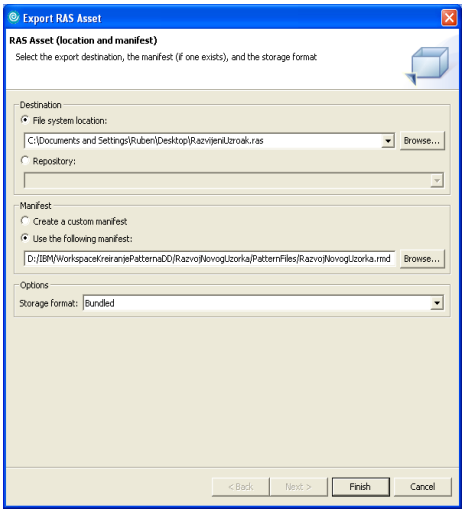
Iz rezultata kako uspješnog tako i neuspješnog testiranja oblikuju se u povratne informacije na temelju predloška prikazanog u točci 5.3.2.4.

Ovaj postupak može se provesti i izvršavanjem testova kroz IBM Rational Manual Tester i IBM Rational Funcitional Tester alate ukoliko su testovi prethodno napisati.

Organizacija uzorka u ponovno iskoristiv resurs

- **Standardizacija i pakiranje**

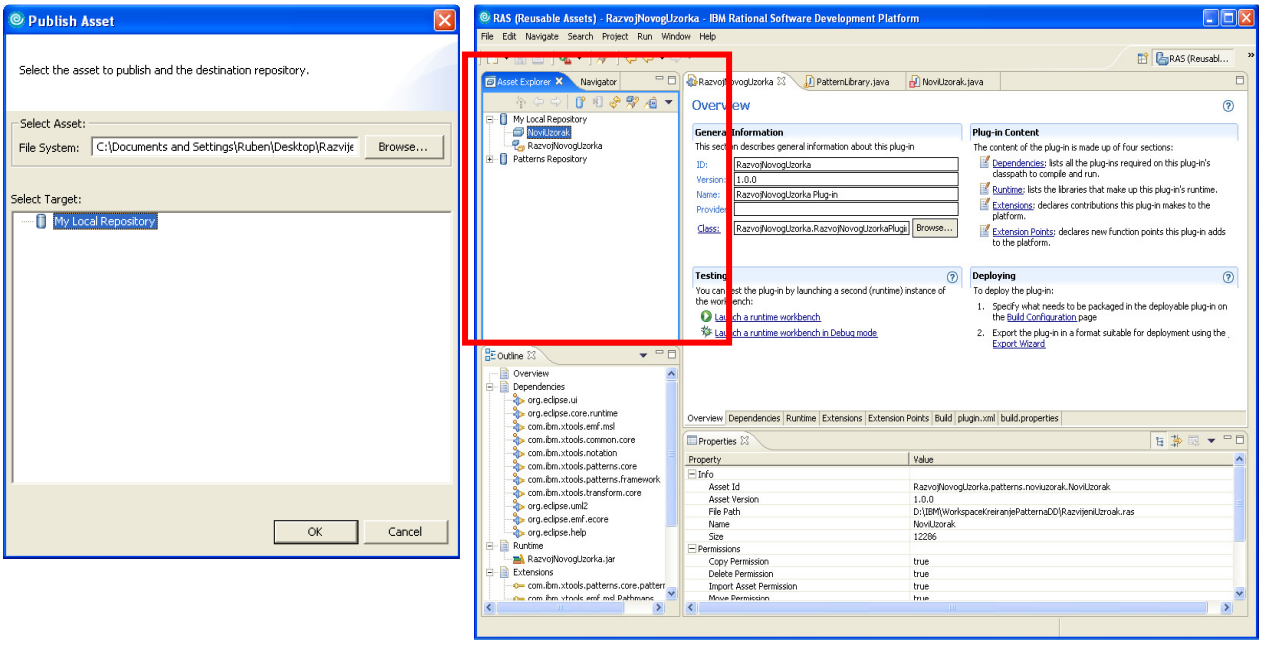
Kako RSA razvojno okruženje svako pakiranje temelji na RAS standardu nije potrebno provoditi neke posebne dodatne aktivnosti. Sve se provodi na automatiziran način odabirom *Export RAS Asset čarobnjaka*. Okruženje će samo generirati ponovno iskoristivi resurs pod nazivom *RazvijeniUzorak.ras* kao što je prikazano na *slici 5-17.*



Slika 5-17. Čarobnjak za izvoz implementiranog uzorka u ponovno iskoristivi resurs - *.ras*.

- **Objavljivanje u repozitorij**

Provodi se na način da se preklopi na RAS perspektivu u kojoj se prikazuju svi raspoloživi repozitoriji. Odabirom *opcije Publish RAS Asset* bira se *.ras* datoteka i ciljni repozitorij, kao što proizlazi iz *slike 5-18*.



Slika 5-18. Objavljivanje u repozitorij.

Ovim korakom završava segment razvoja novog uzorka.

U zaključku ove točke, kroz *tablicu 5-6.*, sažeto se prikazuje ovaj segmenta razvoja metodološkog okvira. Sam slijed aktivnosti može se vidjeti na *slici 5- 19*.

Tablica 5-6. Sažetak segmenta razvoja metodološkog okvira – razvoj novog uzorka

Korak okvira **ŠTO?**	**Razvoj novog uzorka**
Ljudi **TKO?**	- Tim za razvoj uzorka
Faza **KADA?**	- Bilo koja faza u razvoju SW
Aktivnosti **KAKO?**	- Pokretanje projekta razvoja novog uzorka, - Realizacija razvoja ▪ Analiza ▪ Definirane arhitekture rješenja – dizajn uzorka • Oblikovanje rješenja • Specificiranje uzorka (dokumentiranje) • Pisanje testova ▪ Implementacija – konstrukcija uzorka ▪ Validiranje i verificiranje – testiranje uzorka - Organizacija uzorka u ponovno iskoristiv resurs ▪ Standardizacija i pakiranje ▪ Objavljivanje u repozitorij
Alati **ČIME?**	- IBM Rational Software Architect - IBM Rational Manual Tester - IBM Rational Funcitional Tester
Standardi	- UML - RAS
Rezultat **Artefakti**	- Specifikacija uzroka (dokumentacija) - Testovi - Ponovno iskoristivi resurs (*.ras* datoteka)

5.3.4. Upravljanje uzorcima

Upravljanje uzorcima može se promatrati u širem i užem smislu.

U *širem smislu* ono predstavlja okruženje u kojem se balansiraju različiti zahtjevi, potrebe, zadaci u cilju savladavanja ograničenja i rizika te osiguravanja uspješnog razvoja, isporuke i primjene uzorka u kontekstu MDD razvoja. U *užem smislu* odnosi se na same uzorke (i antiuzorke) tj. na njihov smještaj u repozitorij, verzije te promjene koje je potrebno izvršiti.

U nastavku, segment upravljanja uzorcima promatrat će se u užem smislu kroz:

- upravljanje zahtjevima za promjenama u uzorcima i
- upravljanje uzorcima u repozitoriju.

5.3.4.1. Upravljanje zahtjevima za promjenama u uzorcima

Na temelju prikupljenih povratnih informacija (definiranih u segmentu ***primjene***) formira se *zahtjev za promjenom* kako bi se: otklonile nelogičnosti, ispravile pogreške nastale tijekom razvoja, opisale dodatne mogućnosti uzorka te definirala poboljšanja koja je potrebno provesti za novu verziju. *Zahtjev za promjenom* predstavlja artefakt kojim se formalno opisuju, prate i dokumentiraju svi zahtjevi tijekom životnog ciklusa uzorka. Naglasak je potrebno staviti i na format opisa tih zahtjeva koji bi, baš kao što je to slučaj i s predlošcima za opisa povratih informacijama, trebao biti standardiziran. Kako postoji velik broj već gotovih predložaka *zahtjeva za promjenama* potrebno je odabrati onaj koji se smatra prikladnim.

Prilikom definiranja zahtjeva korisno je voditi računa i o:

- **Obimu promjene.** Koliko toga što je do sada napravljeno će biti potrebo promijeniti i koliko dodatnog posla će biti potrebno obaviti?
- **Alternative.** Postoje li alternative?
- **Složenost promjene.** Da li će predloženu promjenu biti jednostavno izvršiti?
- **Težina promjene.** Što će se desiti ako se promjena ne provede?
- **Vremenski raspored.** Kada će se izmjena provoditi i da li je to izvedivo?
- **Utjecaj promjene.** Koje su posljedice provođenja/ne provođenja promjene?
- **Trošak.**
- **Testiranje.** Je li potrebno provesti dodatno testiranje kako bi se utvrdilo da je promjena bila uspješna?

5.3.4.2. Upravljanje uzorcima u repozitoriju

Iako je trenutno na raspolaganju jedan *IBM Rational developerWorks* repozitorij iz kojeg se mogu učitavati do sada razvijeni uzorci za široku primjenu, organizacija može odlučiti definirati jedan vlastiti repozitorij. Kako je ideja da takav repozitorij ima što veći broj MDD uzoraka jasno je da je prvi korak, jasna organizacija i klasifikacija repozitorija na grupe (logičke cjeline) kataloga uzoraka prema odabranom kriteriju, kao što je npr. tehnologija, faze razvoja, namjena ili neko drugo obilježje. Svaka grupa kataloga sadržavat će kataloge u kojima se nalazi jedan ili više uzorka koji predstavljaju rješenje određenog problema, pri čemu je na razini cijelog repozitorija potrebno voditi računa o nazivima kataloga i uzoraka kako se ne bi pojavljivali sinonimi. Kako nakon niza iteracija poboljšanja pojedine verzije uzorka dolazi do pojave nove, potreban je i mehanizam upravljanja verzijama uzoraka i antiuzoraka.

Sažetak ovog segmenta razvoja metodološkog okvira prikazan je u *tablici 5-7.*

Tablica 5-7. Sažetak segmenta razvoja metodološkog okvira – upravljanje uzorcima

Korak okvira ŠTO?	**Upravljanje uzorcima**
Ljudi TKO?	- Tim za razvoj uzorcima
Faza KADA?	- Bilo kada tijekom razvoja uzorka ili MDD razvoja
Aktivnosti KAKO?	- Upravljanje zahtjevima za promjenama u uzorcima - Upravljanje uzorcima u repozitoriju
Alati ČIME?	- IBM Rational Software Architect
Standardi	-
Rezultat Artefakti	- Zahtjev za promjenom (standardizirani)

Razvijeni metodološki okvir, podijeljen u 4 segmenta, pokriva cijeli životni ciklus uzorka. Kada se aktivnosti iz svih segmenata povežu, dobiva se dijagram koji prikazujem na *slici 5-19.*

6. INTEGRACIJA RAZVIJENOG OKVIRA SA SUVREMENIM METODIKAMA RAZVOJA PROGRAMSKOG PROIZVODA

Razvoju ni jednog programskog proizvoda ne pristupa se stihijski ili adhoc, već se on temelji na odabranoj metodici razvoja. No, današnje metodike vrlo malo ili bolje rečeno gotovo ništa ne govore u kontekstu MDD razvoja (djelomičan uzrok leži u činjenice da se pristup još razvija). Situacije nije nimalo drugačija ni s uzorcima, iako su oni već duže prisutni u SW industriji. Prema istražnoj literaturi trenutno ne postoji metodika razvoja koja je prilagođena za MDD, a kamoli da objedinjava razvoj temeljen na modelima primjenom uzoraka. Ideja razvoja ovog okvira nije njegova primjena u kontekstu neke nove metodike ili da okvir bude sam sebi dovoljan, već pokušaj da ga se oblikuje na način prikladan za integraciju u već postojeće metodike razvoja. Stoga, ovo poglavlje prikazuje u kojem je obliku to moguće provesti i koliko uspješno.

6.1. Suvremene metodike razvoja

U posljednjih 30-ak godina isproban je velik broj različitih pristupa razvoju programskih proizvoda. Danas se u SW industriji, metodike razvoja programskih proizvoda klasificiraju u dvije glavne **skupine** na:

- formalne (eng. heavyweight) i
- agilne (eng. lightweight).

I dok je u *formalnim metodikama* naglasak na detaljnom planiranju, modeliranju i dokumentiranju sustava koji će se razvijati, *agilne metodike* naglašavaju da je zbog današnjeg okruženja u kojem nastaje programski proizvod, potrebno brzo i bez suvišne dokumentacije razviti i isporučiti programski proizvod koji će zadovoljiti zahtjeve klijenta koji se ionako učestalo mijenjaju. Iako ne postoje jasno definirane granice između ove dvije skupine metodika naglasak formalnih je u dosljednom slijeđenju procesa razvoja, dok je u agilnim metodikama naglasak na vrijednostima i definiranim principima. U nastavku poglavlja opisat će se karakteristike svake **skupine** metodika, nabrojat će se metodike koje spadaju u nju te će se sažeto opisati jedan predstavnik (najzastupljenija) svake skupine metodika.

6.1.1. Formalne metodike

Svim metodika ove skupine, zajedničko je dobro upravljanje *složenošću sustava* - jednim od dva glavna izazova u razvoju softvera. Kako to postižu?

Glavne karakteristike formalnih metodika su:

- Temelje se na opsežnom planiranju.
- Sastoje se od velikog broj artefakata i strogo formalno opisanih aktivnosti kojih se je potrebno pridržavati tijekom razvoja softvera.
- Zahtijevaju vrijeme, disciplinu i veliku količinu dokumentacije koja mora pratiti cijeli razvojni ciklus.

Primjenjuju se u razvoju velikih softverskih projekata na kojem sudjeluje projektni timovi s velikim brojem ljudi.

Dvije najčešće spominjane metodike iz te skupine su:

- RUP (eng. Rational Unified Process) i
- MSF (eng. Microsoft Solution Framework).

Kako bi detaljno objašnjavanje ovih metodika izašlo iz okvira ovog rada u nastavku se sažeto prikazuje samo RUP metodika za koju će se kasnije prikazati integracija sa razvijem okvirom.

6.1.1.1. RUP metodika

Za RUP (eng. Rational Unified Process) metodiku kaže se da predstavlja najbolju praksu u današnjem programskom inženjerstvu, jer nastoji objediniti dokazane pristupe, metode i tehnike iz dosadašnjeg razvoja SW. Šest fundamentalnih *najboljih praksi* RUP metodike su [Kruchten, 2003., str. 5.] ; [RSA v6.0.1]:

- SW je potrebno razvijati iterativno,
- neophodno je upravljanje zahtjevima,
- u razvoju je potrebno primjenjivati SW komponente,
- koristiti vizualno modeliranje SW-a (UML notacija),
- kontinuirano provjeravati kvalitetu SW-a,
- kontrolirati upravljanje promjenama.

Zapravo RUP predstavlja skup široko definiranih smjernica i predložaka, koje je moguće

prilagođavati ovisno o stvarnom projektu, tako da je ponekad za ostvarenje pojedinih ciljeva moguće pronaći i nekoliko mogućih smjerova razvoja.

Temeljna četiri svojstva koja karakteriziraju RUP metodiku su:

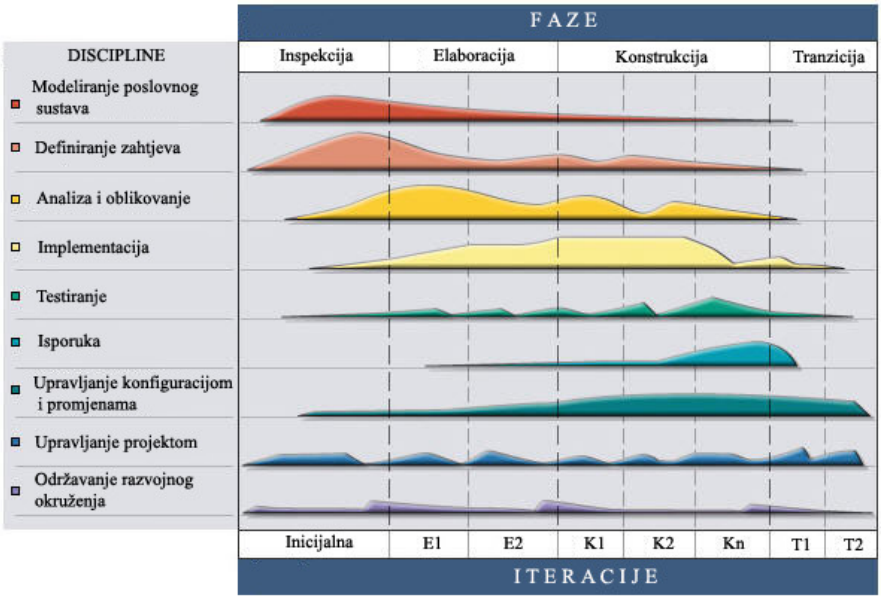
- razvoj temeljen na *slučajevima korištenja*,
- usmjerenost arhitekturi sustava,
- iterativnost i
- inkrementalnost.

6.1.1.1.1. Arhitektura RUP metodike

I dok svaka metodika svoj proces razvoja promatra samo kroz jedan pogled, RUP proces razvoja opisan je kroz dvije dimenzije:

- dinamička perspektiva (horizontalna dimenzija) prikazuju *faze* kroz vrijeme i
- statička perspektiva (vertikalna dimenzija) prikazuje *discipline* koje su definirane aktivnostima koje je potrebno provesti.

Međusobni odnos tih dimenzija grafički je prikazan *slikom 6-1*.



Slika 6-1. Arhitektura RUP metodike.

Izvor: RSA v6.0.1

Horizontalna dimenzija, kroz vrijeme, prezentira dinamičke aspekte koji su izraženi fazama, iteracijama i kontrolnim točkama (eng. milestone).

Vertikalna dimenzija, discipline, prezentira statičke aspekte koji su izraženi aktivnostima, disciplinama, artefaktima i ulogama. Uz svaku disciplinu definiran je i dijagram kojim se

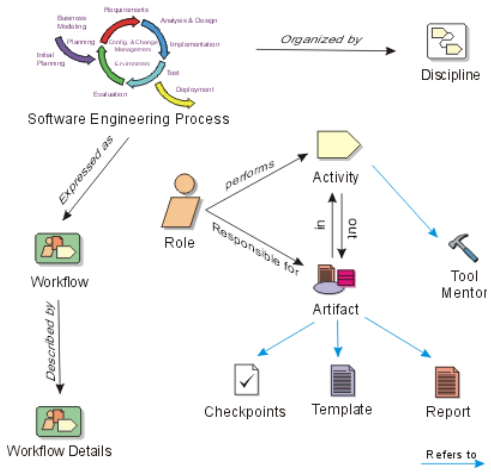
prikazuje tijek definiranih aktivnosti (eng. workflow).

Brežuljci u grafu prikazuju intenzitet rada u pojedinoj disciplini. S obzirom na položaj unutar razvojnog ciklusa neke discipline su više zastupljene od drugih. Uočljivo je da su sve discipline prisutne u svakom vremenskom razdoblju uz promjene u intenzitetu. Na samom početku rada na projektu najviše vremena se provodi u analizi zahtjeva i izradi modela poslovanja. U kasnijim iteracijama fokus polako prelazi na discipline implementacije i testiranja.

Na temelju objašnjenja ove dvije dimenzije, mogu se identificirati osnovni elementi RUP metodike:

- Glavni
  - disciplina,
  - uloga: tko?,
  - aktivnost: kako?,
  - artefakt: što? i
  - tijek: kada?
- Dodatni
  - smjernice,
  - predlošci i
  - alati.

Njihov međusobni odnos proizlazi iz *slike 6-2*.



Slika 6-2. Osnovni elementi RUP metodike.

Izvor: RSA v6.0.1

U nastavku se kratko prikazuje svaka dimenzija.

6.1.1.1.1.1. Horizontala dimenzija

Predstavlja dimenziju koja kroz vrijeme opisuje razvoj programskog proizvoda kroz *faze*, *iteracije* i *kontrolne točke*.

Faze RUP metodike su:

- inspekcija,
- elaboracija,
- konstrukcija i
- tranzicija.

Inspekcija: Osnovni cilj ove faze je određivanje izvodljivosti projekta, tj. određivanje da li je moguće ispuniti zahtjeve korisnika, te da li se to uopće isplati ili ne (što ukazuje na činjenicu da je projekt po završetku ove faze moguće i otkazati). Da bi se uspješno odredila izvodljivost projekta potrebno je upoznati se s poslovnim sustavom, te odrediti zahtjeve sustava. Nakon što se odrede zahtjevi sustava potrebno je uočiti potencijalno najkritičnije rizike, te pronaći način kako ih izbjeći, možda već u idućoj iteraciji. Potrebno je pronaći i barem jednu idejnu arhitekturu sustava koja će osigurati da je sustav uistinu moguće i fizički ostvariti. Posljednji korak je izrada plana razvoja sustava. Faza se najčešće provodi kroz jednu, ponekad i dvije iteracije. Ovo je jedina faza u kojoj iteracije ne završavaju s prototipom. U sklopu ove faze obavljaju se i pripremne aktivnosti u obliku: pripreme radnog okruženja, edukacija članova razvojnog tima i sl. Tijek aktivnosti koje proizlaze ovom fazom može se pronaći u RSA v6.0.1 (Process Browser).

Kroz *tablicu 6-1* sažeto se daje odgovor na pitanja tko?, što?, te očekivani rezultat.

Tablica 6-1. Sažetak faze inspekcije

Tko?	Što?	Artefakt
Voditelj projekta Analitičar poslovnih procesa	Novi projekt	Poslovni slučajevi korištenja
	Priprema okruženja projekta	Projektna specifikacija
Analitičar sustava Arhitekt softvera Testeri	Projektni plan	Plan razvoja softvera
		Plan iteracija
	Kontrola projekta	Procjena rizika
	Analiza problema	Definiranje vizije
		Korisnički zahtjevi
	Razumijevanje potreba naručioca	Definiranje vizije
	Upravljanje ciljem sustava	Korisnički zahtjevi
	Definicija sustava	Korisnički zahtjevi
	Arhitekturna sinteza	Idejna arhitektura
	Definicija evaluacije	Strategija testiranja
	Upravljanje iteracijama	Procjena iteracija
	Planiranje iteracija	Plan iteracija

Izvor: RSA v6.0.1

Elaboracija: Nakon što su opisani zahtjevi, te je utvrđeno da je moguće razviti sustav koji ih ispunjava, nastupa faza elaboracije. Osnovni ciljevi ove faze su uspostavljanje stabilne arhitekture sustava i izrada detaljnog plana daljnjeg tijeka razvoja projekta. Potrebno je uočiti najvažnije slučajeve korištenja sustava, te ih detaljno analizirati. U prethodnoj fazi su samo površno opisani. Definira se osnovna arhitektura koja se kasnije razrađuje često i na nekoliko slojeva. Dijelovi sustava određeni njegovom arhitekturom se zatim i fizički realiziraju, te testiraju. Tako dolazimo do kostura našeg sustava kojem u fazi konstrukcije dodajemo ostale dijelove. Pažljivo treba promatrati razvoj liste potencijalnih rizika. Oni rizici koji dovode u pitanje uspješnost projekta trebaju biti eliminirani do kraja ove faze. Faza se izvodi često u nekoliko iteracija. Provodimo ih sve dok ne utvrdimo da je arhitektura sustava stvarno stabilna. Tijek aktivnosti koje proizlaze ovom fazom također se mogu pronaći u RSA v6.0.1 (Process Browser). Kroz *tablicu 6-2.* sažeto se daje odgovor na pitanja tko?, što?, te očekivani rezultat provođenja ove faze.

Tablica 6-2. Sažetak faze elaboracije

Tko?	Što?	Artefakt
Analitičar poslovnih procesa	Priprema okruženja za izvođenje iteracija	Sredstva rada
Voditelj projekta Analitičar sustava Arhitekt softvera Programer Tester Ostali	Potpora okruženju	Razvoj infrastrukture
	Upravljanje iteracijama	Procjena iteracije
	Kontrola projekta	Procjena rizika
	Upravljanje upravljanjem zahtjeva	Promjena zahtjeva
	Analiza sustava	Korisnički zahtjevi
	Definicija predloška arhitekture	Model arhitekture
	Poboljšanje arhitekture	Unaprijeđeni model arhitekture
	Razvoj komponenti	Ponovno iskoristivi resursi
	Testiranje i procjena	Sažetak testiranja i evaluacije
	Pregled testiranja	
	Upravljanje promjenama zahtjeva	Promjena zahtjeva
	Planiranje iteracija	Plan iteracija

Izvor: RSA v6.0.1

Konstrukcija: Faza obuhvaća dizajn sustava, programiranje i testiranje. Kako se dijelovi sustava razvijaju paralelno tijekom te faze usljeđuje i integracija već razvijenih dijelova te sveobuhvatno testiranje. Na kraju faze potrebno je raspolagati s funkcionalnim programskim proizvodom i pripadajućom dokumentacijom koja je spremna za isporuku. Kako će se beta verzije sustava testirati u okolini korisnika, tijekom ove faze nužno je pripremiti korisnika i opremu na kojoj će sustav biti instaliran za fazu tranzicije. Ova faza se redovito izvodi u nekoliko iteracija. Tijek aktivnosti koje proizlaze ovom fazom također se mogu pronaći u RSA v6.0.1 (Process Browser). Kroz *tablicu 6-3.* sažeto se daje odgovor na pitanja tko?, što?, te očekivani rezultat provođenja ove faze.

Tablica 6-3. Sažetak faze konstrukcije

Tko?	Što?	Artefakt
Analitičar poslovnih procesa	Priprema okruženja za iteracije	Alati
Voditelj projekta Menadžer isporuke Analitičar sustava Programeri Testeri Ostali	Podrška okruženju	Razvoj infrastrukture
	Upravljanje iteracijom	Procjena iteracije
	Kontrola projekta	Procjena rizika
	Upravljanje promjenama zahtjeva	Promjena zahtjeva
	Razvoj komponenti	Ponovno iskoristivi resursi
	Testiranje i evaluacija	Zapis o procjeni i testiranju
	Razvoj uputa za korisnike	Upute za korisnike
	Planiranje iteracija	Plan iteracija
	Priprema isporuke	Programski proizvod

Izvor: RSA v6.0.1

Tranzicija: Cilj ove faze je omogućiti rad i korištenje sustava u radnom okruženju, što uključuje uvođenje sustava kod korisnika, te završna testiranja. Na kraju ove faze ciljevi projekta bi trebali biti ispunjeni. U nekim slučajevima završetak ove faze pokreče cijeli novi životni ciklus u kojem nastaje nova generacija izgrađenog sustava. Faza tranzicije može biti provedena u jednoj, a u pojedinim slučajevima i u više iteracija (uvođenje novog sustava u nuklearnu elektranu, sustav navođenja međunarodnog aerodroma, ERP sustav). Tijek aktivnosti koje proizlaze ovom fazom također se mogu pronaći u RSA v6.0.1 (Process Browser). Kroz *tablicu 6-4.* sažeto se daje odgovor na pitanja tko?, što?, te očekivani rezultat provođenja ove faze.

Tablica 6-4. Sažetak faze tranzicije

Tko?	Što?	Artefakt
Analitičar poslovnih procesa Voditelja projekta Integrator Programeri Tester Ostali	Priprema okruženja za iteracije	Alati
	Podrška okruženju	Razvoj infrastrukture
	Planiranje isporuke	Plan isporuke
	Upravljanje iteracijom	Procjena iteracije
	Upravljanje promjenama zahtjeva	Promjena zahtjeva
	Implementacija komponenti	Implementacija
	Testiranje i procjena	Zapisnik o procjeni i testiranju
	Integracija sustava	Skripte za testiranje
	Pisanje korisničkih uputa	Upute za korisnike
	Planiranje iteracija	Plan iteracije
	Objedinjavanje SW	Programski proizvod
	Završavanje projekta	Procjena stanja

Izvor: RSA v6.0.1

Iteracije su vremenski određeni interval s definiranim specifičnim ciljeva. Drže se što je vremenski moguće kraćima, ali dovoljno dugima kako bi se implementirali zahtjevi korisnika. Mogu se promatrati u kontekstu:

- *Svake faze.* Faza se odvija iterativno (kroz jednu ili više iteracija) pri čemu rezultat nastaje inkrementalno.
- *Svih faza.* Programski proizvod se inkrementalno razvija kroz niz iteracija od kojih svaka ima oblik malog vodopada, te sadrži sve discipline od analize do testiranja.

Na kraju svake iteracije prilagođavaju se planovi budućih iteracija na temelju rezultata trenutne iteracije.

Svaka faza završava **kontrolnom točkom** (eng. milestone) kako bi se provjerilo je li postignut željeni učinak provođenja određene faze. Projekt ne može prijeći u sljedeću fazu ukoliko ne zadovoljava uvijete postavljene u **kontrolnoj točci.** Za svaku pojedinu fazu uvjeti se unaprijed određuju. Npr. kontrolna točka na kraju Inspekcije može biti *Provjeravanje definiranih ciljeva projekta*, na kraju Elaboracije *Provjeravanje definirane arhitekture*, na kraju Konstrukcije *Provjeravanje beta izdanja*, a na kraju Tranzicije *Provjeravanje konačnog izdanja*.

6.1.1.1.1.2. Vertikalna dimenzija

Vertikalna dimenzija predstavlja statički pogled na RUP s fokusom na aktivnosti koje se izvode tijekom svih faza u razvojnom ciklusu, a grupirane su u **discipline**. Discipline predstavljaju podjelu svih uloga, aktivnosti i tijeka procesa na logičke grupe prema području važnosti. RUP-om je definirano šest glavnih i tri potporne discipline.

Glavne discipline u RUP-u su:

- Modeliranje poslovnog sustava (eng. Business Modeling),
- Definiranje zahtjeva (eng. Requirements),
- Analiza i oblikovanje (eng. Analysis & Design),
- Implementacija (eng. Implementation),
- Testiranje (eng. Test) i
- Isporuka (eng. Deployment).

Potporne discipline u RUP-u su:

- Upravljanje konfiguracijom i promjenama (eng. Configuration & Change Management,
- Upravljanje projektom (eng. Project Management) i
- Održavanje razvojne okoline (eng. Environment).

Modeliranje poslovnog sustava: Tijekom ove discipline razvojni tim se upoznaje s logikom poslovnog sustava. Kod malih projekta ono obuhvaća razvoj modela domene, dok je kod velik projekata nužno detaljno modeliranje poslovnih procesa pa čak i njihovo preoblikovanje prije definiranja modela. Kako je temelj RUP metodike UML notacija, i za modeliranje poslovnih procesa koristi se ista notacija. Iako se na prvi pogled to ne čini s(p)retnim rješenjem, jer postoje notacije koje su za modeliranje poslovnih procesa prikladnije od UML, ideja je ukloniti dvosmislenost i osigurati bolju komunikaciju primjenom jedne notacije. Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Svrha discipline je:

- razumijevanje organizacijske strukture i bolji uvid u njezina moguća poboljšanja,
- procjena utjecaja organizacijske promjene,
- prikazivanje organizacijska struktura svim interesnim skupinama,
- osmišljavanje softverske podloge koja će biti potpora organizaciji i
- razumijevanje načina implementacije novog sustava.

Definiranje zahtjeva: Disciplina kojom se određuju granice budućeg sustava i detaljno specificiraju korisnički zahtjevi (funkcionalni i nefunkcionalni) – što se očekuje od sustava. Ključni artefakt je *model slučaja korištenja* čiji razvoj se provlači kroz *inspekciju* - tijekom koje se nastoje uočiti svi potrebni slučajevi korištenja i učesnici koji sudjeluju u interaciji i *elaboraciju* – u kojoj se bira nekoliko najznačajnijih slučaja korištenja koji se tada detaljno opisuju i za koje se gradi arhitektura sustava koja će ih podržavati. Aktivnosti ove discipline ponavljaju se u nekoliko iteracija. Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Svrha discipline je:

- ustanoviti i održati dogovor o tome što sustav treba raditi,
- pomoći razvojnom timu u razumijevanju zahtjeva sustava,
- postaviti granice sustava,
- omogućiti planiranje tehničkih sadržaja,
- procijeniti cijenu i vrijeme potrebno za razvoj sustava i
- definirati grafičko sučelje sustava u odnosu na potrebe korisnika.

Analiza i oblikovanje: Na temelju prethodno definiranog *modela slučaja korištenja* slijedi analiza zahtjeva te oblikovanje arhitekture koja će biti sposobna zadovoljiti sve navedene zahtjeve. Već u fazi *inspekcije* započinje se s grubom slikom arhitekture. No najznačajniji dio poslova obavlja se tijekom *elaboracije* izgradnjom stabilne arhitekture novog sustava. Primjenom UML notacije nastoji se analizirati i opisati statika i dinamika budućeg sustava. Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Svrha discipline je:

- preoblikovanje zahtjeva u ciljani sustav,
- razviti stabilnu arhitekturu sustava i
- prilagoditi dizajn za implementaciju sustava u okolinu uz zadržavanje performansi.

Implementacija: Prethodno definirane modele potrebno je transformirati u programski proizvod. Aktivnosti se izvode tijekom faze *elaboracije* kao podrška analize i oblikovanja u želji da se izgradi stabilna arhitektura. U fazi *konstrukcije* ova disciplina postaje centralna aktivnosti u cilju razvoja izvršne verzije softvera uz obvezno testiranje modula te manja testiranja kako bi se uklonile sitne pogreške vezane uz sintaksu. Kostur arhitekture potrebno je ispuniti cjelokupnom definiranom funkcionalnošću. Poželjna je primjena komponenata,

predložaka i **uzorka** te alata za automatiziran razvoj (generatora aplikacija) kako bi se ubrzale aktivnosti u ovoj disciplini. Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Svrha implementacije je:

- ustanoviti način na koji je organiziran programski kod i omogućiti slojevitost koda,
- programiranje,
- pojedinačno testiranje modula i
- integracija pojedinih modula u svrsishodnu softversku cjelinu.

Testiranje: Testiranje predstavlja iterativan proces provjeravanja sustava (V&V) koji se provodi s disciplinom implementacije. No, testiranje cijelog sustava i testiranje integracije vrši se unutar discipline testiranje. Kako je to naporna i zahtjevna aktivnost provjeravaju se mogućnosti sustava te njegova (ne)željena reakcija u svim mogućim scenarijima korištenja sustava. Testiranje se proteže i kroz sve faze razvoja. Tako se tijekom *inspekcije* testiranje planira, u *elaboraciji* se pišu testovi, u *konstrukciji* se testiraju svi moduli i sustav kao cjelina dok se u *tranziciji* provjerava rad sustava kod korisnika. Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Svrha discipline testiranje je procjena kvalitete proizvoda koja se ostvaruje pomoću sljedećih principa:

- pronaći i opisati greške u kvaliteti softvera,
- preporučiti rješenje zapaženih problema u kvaliteti softvera,
- provjeriti i dokazati kroz konkretne primjere specifikacije napravljene u disciplinama zahtjevi i dizajn,
- provjeriti da li proizvod radi kao što je to i dizajnirano i
- provjeriti da li su zahtjevi pravilno implementirani.

Isporuka: Obuhvaća aktivnosti vezane uz postavljanje programskog sutava u radno okruženje klijenta. Iako se proces prebacivanja odvija u fazi *tranzicije*, aktivnosti ovog područja potrebno je započeti ranije. Prije samog čina isporuke potrebno je detaljno ispitati korisnikovo okruženje, te napraviti detaljan plan isporuke i ostale potrebne pripreme. Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Upravljanje konfiguracijom i promjenama: Ova disciplina obuhvaća prilagođavanje softvera krajnjem korisniku i upravljanje zahtjevima za promjenama. S porastom složenosti

današnjih sustava raste i složenost procesa njihove izrade. Posljedica toga je i povećanje količine raznih dokumenata i modela proizvedenih tijekom razvojnog procesa kao i broj stručnjaka koji sudjeluju u njihovoj izradi. Kako bi se održao red neophodno je upravljanje konfiguracijom i promjenama. Osnovna svrha ove discipline je održati stabilno razvojno okruženje, te osigurati da se promjene ne događaju slučajno, već da se promjene prate od trenutka nastanka uzroka promjene i zahtjeva za promjenom pa sve do konačnog izvještaja o uspješno obavljenoj promjeni. Upravljanje konfiguracijom prolazi kroz četiri koraka:

- raspoznavanje predmeta konfiguracije,
- ograničavanje promjena nad predmetom konfiguracije,
- revizija promjena napravljenih nad predmetom konfiguracije i
- definicija upravljanja konfiguracijom predmeta.

Aktivnosti upravljanja konfiguracijom kao i upravljanje promjenama predstavljaju podršku procesu razvoja, te nikada ne zauzima centralno mjesto u samom procesu. Dio je svakodnevnog rada svih sudionika tima, te se obim zadataka povećava s povećanjem broja sudionika i povećanjem dokumentacije koju je potrebno pratiti. Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Upravljanje projektom: Predstavlja disciplinu koja ima za cilj uspostaviti ravnotežu između managerskih sposobnosti, rizika, troškova, rokova i svih potrebnih resursa u razvoju SW projekata.

Svrha discipline upravljanja projektima je:

- pružiti okruženje za upravljanje zahtjevnim softverskim projektima,
- pružati praktične smjernice za planiranje, upravljanje kadrovima, izvršavanje i nadziranje projekta i
- pružati okolinu za upravljanje rizikom.

Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Održavanje razvojnog okruženja: Svrha discipline je pružiti okruženje pogodno za razvoj softvera. Disciplina se fokusira na tri elementa:

- prilagodba RUP metodike,
- osiguranje prikladnih razvojnih alata koji trebaju pružiti podršku cjelokupnom procesu razvoja i
- smjernice za prilagodbu organizacije potrebama projekta.

Tijek aktivnosti ove discipline može se pronaći u RSA v6.0.1 (Process Browser).

Ovaj kratak opis istaknuo je ključne elemente RUP metodike. Iako ne postoji najbolja metodika, i svakako RUP nije metodika koja je prikladna za sve tipove razvoja programskog proizvoda, ona sadrži inovacije koje su danas u SW industriji priznate kao skup najboljih praski u razvoju SW [Sommerwille, 2007., str. 85.].

6.1.2. Agilne metodike

Kako tradicionalne (formalne) metodike znatnu količinu vremena posvećuju definiranju *kako*? će se razviti softver, a tek tada programiranju i testiranju, agilne metodike razvoja softvera u fokus stavljaju sam programski proizvod, te pokušavaju ponuditi odgovor na želju za manje opsežnim i ne toliko detaljnim metodikama razvoja softvera, koje sa sobom donose brže, pokretljivije i aktivnije procese razvoja softvera kako bi se što kvalitetnije upravljalo *promjenama sustava* - drugim od dva glavna izazova u razvoju SW. S tim ciljem sredinom 1990-tih godina započinje razvijanje manje opsežnih metodika razvoja softvera, koje obično sadrže tek nekoliko pravila i načina postupanja koji su lagani za slijeđenje, a objavom manifesta pod naslovom *"Agile Software Development Manifesto"* 2001.g i formalno se usvaja pojam agilnog razvoja [Beck, *et al.*, 2001.].

Od 12 principa agilnog razvoja objavljenih u manifestu, u 4 glavna principa ubrajaju se [Beck, *et al.*, 2001.]:

- **Individualnost i interakcije važniji su od procesa i alata.** Agilni pokret naglašava vezu i zajedništvo programera i ljudskih uloga u suprotnosti s institucionaliziranim procesima i razvojnim alatima. U postojećim agilnim praksama, to se manifestira u bliskosti projektnog tima i dobrom timskom duhu.
- **Softver koji radi važniji je od opsežne dokumentacije.** Vitalni zadatak projektnog tima je kontinuirana isporuka testiranog softvera koji radi. Nove verzije se isporučuju u pravilnim razmacima (npr. nekoliko puta mjesečno). Programeri su dužni održavati programski kod jednostavnim i tehnički dorađenim što je više moguće. Naglasak nije na količini proizvedene dokumentacije.
- **Suradnja s klijentom važnije je od striktnih ugovora.** Veza i suradnja između programera i naručitelja je u prednosti nad striktnim ugovorima, iako važnost dobro sročenih ugovora raste istim tempom kao i veličina softverskih projekata. Iz poslovnog

perspektive, agilni razvoj je fokusiran na što bržu isporuku, ubrzo nakon što je projekt startan, čime se smanjuju brojni rizici (promjene zahtjeva, odustajanje od projekta i sl.).

- **Reakcija na promjene važnije je od slijeđenja plana.** Kako će tijekom projekta razvoja sigurno doći do prilagodbi i promjena, razvojni tim koji sadrži i naručitelja treba biti dobro informiran, kompetentan i autoritativan što je važnije od skupa planova koji ne odgovaraju novonastaloj situaciji.

Kao rezultat agilnog pristupa u razvoju softvera, mogu se izdvojiti slijedeće agilne metodike [Sommerwille, 2007., str. 85.], [Abrahamsson, P. *et al.*, 2002., str. 18.]:

- **XP - Extreme Programming** (Beck, 1999. godina),
- Scrum (Schwaber 1995. godine, Schwaber i Beedle 2002. godina),
- Crystal skupina metodika (Cockburn 2002. godina),
- Feature Driven Development (Palmer i Felsing 2002. godina),
- Dynamic System Development Method (Stapleteon 1997. godina),
- Adaptive Software Development (Highsmith 2000. godina),
- Open Source Software Development (O'Reilly 1999. godina),
- Agile Modeling (Ambler, 2002. godina) (nije službeno metodika) i
- Lean Software Development.

Ishodišna točka razvoja agilnog pristupa te danas najraširenija i najpoznatija agilna metodika je XP. Slijedi kratko razmatranje te metodike jer će se kasnije za nju prikazati integracija s razvijem okvirom.

6.1.2.1. XP metodika

XP ili *ekstremno programiranje* je predstavnik lakših i manje opsežnih metodika razvoja softvera namijenjen prvenstveno malim timovima koji razvijaju softver i suočavaju se s zahtjevima česte dinamike promjena. Prema [Beck, 2002., str. XVII] smatra je se laganom, efikasnom, predvidivom, fleksibilnom i metodikom s malim rizikom. Osnovne osobine XP metodike su [Baird, 2003., str. 32.]:

- konkretna povratna informacija,
- postepeno planiranje,
- fleksibilnost isporuke,
- automatizirano testiranje,

- verbalna komunikacija i
- evolucijski pristup.

6.1.2.1.1. Organizacija XP metodike

Metodika je teorijski oblikovana kroz *vrijednosti* koje podržavaju *principi*, a ostvaruju se *aktivnostima* definiranim u skupu *najboljih praksi* [Beck, Andres, 2004., pogl. 3.], [Baird, 2003., str. 26.].

6.1.2.1.1.1. Vrijednosti

Vrijednosti koje prožimaju XP su [Beck, Andres, 2004., pogl. 4.]:

- **Jednostavnost.** Ocrtava se u činjenici da se implementira samo ono što je stvarno potrebno, a ne nešto će eventualno možda trebati u budućnosti.
- **Komunikacija.** Izravna komunikacija s klijentom i kolegama važnija je od dokumenata i izvještaja.
- **Osiguranje povratnih informacija.** Klasificiraju se na: *povratne informacije od sustava* – uvid u stanje nakon provedenih promjena, *povratne informacije od klijenta* – osiguravanje validacije i potrebnih promjena te zadovoljstva korisnika te *povratne informacije od tima* – procjena vremenskog učinka pojedinih aktivnosti.
- **Hrabrost.** Ukazuje na spremnost da se programira brzo i da se već razvijeni programski kod refaktorira bez obzira na utrošeno vrijeme.
- **Poštovanje.** Odnosi se na rad drugih članova u timu čime se postiže visoka motiviranost i odanost timu u cilju postizanja cilja.

6.1.2.1.1.2. Principi

Iz osnovnih vrijednosti proizlazi pet glavnih principa XP metodike koji utječu na razvoj programskog proizvoda [Beck, Andres, 2004., pogl. 5.]; [Baird, 2003., str. 29.]:

- **Brza povratna informacija** – kroz kratke iteracije na kraju koje se raspolaže s povratnim informacijama programer treba saznati da li njegov rad zadovoljava zahtjeve klijenta.
- **Pretpostavljena jednostavnost** – svaki se problem promatra kao da ga se može riješiti na jednostavan način.

- **Postepene promjene** – svi problemi, bilo da se radi o planiranju, modeliranju ili razvoju, rješavaju se preko niza postepenih malih promjena.
- **Opcionalnost tijekom razvoja** – tijekom razvoja prihvaćaju se samo oni smjerovi razvoja koji uvijek sadrže i moguće opcije.
- **Kvalitetan rad** – u pogledu kvalitete ne smije biti kompromisa. Programski kod i njegovo testiranje dodatno dobivaju na važnosti.

6.1.2.1.1.3. Aktivnosti

Tijekom primjene XP metodike, svi postupci mogu se klasificirati u slijedeće 4 grupe aktivnosti [Baird, 2003., str. 30.]:

- **Slušanje.** Kako se metodika manje oslanja na formalnu, pisanu dokumentaciju verbalna komunikacija i aktivno slušanje predstavljaju važne aktivnosti.
- **Testiranje.** Ne predstavlja postupak koji se provodi nakon što je sustav razvijen, već se testovi pišu i implementiraju prije same realizacije sustava. To zahtjeva promjenu u načinu razmišljanja, ali rezultira kvalitetnijim programskim kodom.
- **Kodiranje.** Uključuje primjenu novih tehnika kao što su refaktoriranje i programiranje u paru.
- **Dizajn.** Ideja XP je da se dizajn razvija tijekom samog projekta razvoja te da u njemu sudjeluju svi članovi tima. Takva ideja ima za promjenu da se dizajn s razvojem dinamički mijenja - evoluira.

6.1.2.1.1.4. Najbolja praska

Srž XP metodike čini skup smjernica, pravila i aktivnosti obuhvaćenih u najbolju praksu. Jezgru najbolje prakse čine [Baird, 2003., str. 31.]; [Beck, 2002., str. 53]:

1. **Planiranje igre.** Zapravo je sastanak između naručitelja i voditelja projekta kojim se nastoje sastaviti korisnički zahtjevi u obliku korisničkih priča (izrađuje ih sam naručitelj). Korisničke priče trebaju osigurati dovoljno detalja kako bi se načinila razumna procjena relativno niskog rizika koja prikazuje koliko dugo će trajati implementacija te korisničke priče. Korisničke priče vode kreiranju tzv. testa prihvaćenosti softvera kojeg potvrđuje naručitelj.

2. **Male i česte verzije.** Zadaća razvojnog tima je što češće isporučiti iterativne verzije sustava naručitelju. Kod planiranja isporuke potrebno je definirati male jedinice funkcionalnosti koje su pogodne za isporuku i koje mogu biti isporučene u okruženje naručitelja u ranim fazama projekta. Kritično je dobiti vrijednu i kvalitetnu povratnu informaciju od korisnika kako bi se na vrijeme imao bolji utjecaj na daljnji razvoj sustava. Što se više čeka s isporukom važnih obilježja korisnicima, bit će manje vremena za njihov popravak.
3. **Metafora.** Metafora sustava predstavlja pojednostavljenu sliku sustava koji se razvoja. Važno je da ta slika sustava bude čim više pojednostavljena kako bi je svi razumjeli. Glavna ideja ove pojednostavljene slike je da programeri koji sudjeluju u razvoju imaju jasnu sliku gdje se njihov dio nalazi u sustavu, tj. kako se dio koji oni razvijaju uklapa u sustav. Metafora definira zajednički jezik, tj. terminologiju koja se upotrebljava na projektu i koju svi razumiju.
4. **Jednostavan dizajn.** Kako je jednostavnost jedan od vrijednosti XP, ona se može preslikati na dizajn sustava koja bi tada uključivala: izvršavanje svih testova, uklanjanje redundantnog koda, jasno iznošenje namjere koja se želi postići kodom i smanjiti broj klasa i metoda. Glavno pitanje koje si je pri tome učestalo potrebno postavljati jest: *Može li se isto izvesti i na jednostavniji način?*
5. **Testiranje.** Kao ključni korak, testiranje više nije pojava koja se događa nakon implementacije već prije. Najprije se pišu *jedinični testovi* koji se zatim implementiraju pa se tek onda pristupa pisanju koda određene funkcionalnosti. Vrijeme koje je potrebno da se implementira prvo test, a zatim programski kod koji zadovoljava test, slično je vremenu koje je potrebno za implementiranje funkcionalnosti bez implementacije testa. Implementacija testa, dakle, ne odnosi neko dodatno vrijeme, no čini implementaciju funkcionalnosti za koju se piše test dosta lakšom. Ako već postoji jedinični test, nije ga potrebno kreirati nakon implementacije funkcionalnosti, čime se štedi vrijeme. Razine testiranja su: jedinično testiranje, integracijsko testiranje, testiranje sustava, testiranje integracije sustava i testiranje prihvaćanja.
6. **Refaktoriranje.** Je tehnika poboljšavanja programskog kod (restrukturiranja postojećeg tijela koda) bez promjene funkcionalnosti, tj. vanjskog ponašanja programskog koda. Tehnika refaktoriranja uključuje uklanjanje redundancija programskog koda, eliminiranje nekorištene funkcionalnosti te "pomlađivanje" zastarjelog dizajna što programski kod čini lakšim za razumijevanje. Refaktoriranje

tijekom čitavog životnog ciklusa razvojnog projekta štedi vrijeme i povećava kvalitetu.

7. **Programiranje u paru:** Sav programski kod koji će biti uključen u produkciju kreiraju dva programera koji rade u paru za jednim računalom. Budući da programiranje ne predstavlja samo tipkanje programskog koda, već je kreativan proces koji obuhvaća slušanje, promišljanje, razgovor, testiranje, oblikovanje i samo kodiranje, brojne su prednosti zajedničkog rada. Neke od njih su: sav kod provjerile su dvije osobe, najmanje dvoje ljudi poznaje isti dio sustava, manja je mogućnost izostavljanja testova, povećava se kompetencija svakog programera, kod se nadgleda, broj evidentiranih neispravnosti je manji, povećava se kvaliteta softvera bez utjecaja na vrijeme isporuke i drugo. Dvije su uloge (eng. Driver i Navigator) koje se izmjenjuju:
    - programer koji koristi tipkovnicu računala i piše programski kod i
    - programer koji provjerava napisani programski kod.
8. **Zajedničko vlasništvo nad kodom.** Predstavlja koncept koji se daje na znanja da svaki člana tima postaje vlasnik svakog dijela koda. Time se gubi ovisnost (važnost) pojedinca uslijed okolnosti koje bi mogle znatno usporiti ili zaustaviti razvoj. Naglašena je zajednička i pozitivna ali i negativna odgovornost za cjelokupan kod.
9. **Stalna integracija.** Kako timski rad podrazumijeva paralelan rad postavlja se pitanje integracije dijelova programskog koda koji su razvili programeri. Rješenje je da se sav izvršni kod isporučuje u zajednički kontrolni repozitorij, te samo jedan par programera radi promjene nad posljednjom verzijom koda i vrši integraciju. Snažan naglasak na učestalosti proizlazi iz činjenice da preveliki period neusklađivanja vodi do problema (rasula) i vrlo vjerojatnog dodatnog posla za usklađivanje datoteka.
10. **Radni tjedan od 40 sati.** Osnovna ideja je da se ne radi više od 40 sati tjedno. Prekovremeni rad se promatra i negativnom kontekstu te dugoročno gledano smanjuje produktivnost. Također je potrebno voditi računa kako je u radu u "paru" nemoguće raditi produktivno više od 6 sati u razvoju SW.
11. **Stalna prisutnost naručitelja u timu.** Stalna dostupnost naručitelja jedan je od osnovnih zahtjeva XP metodike. Postoje dva osnovna načina kako naručitelj može sudjelovati u timu:
    - da nešto radi u projektu (ima dodijeljene zadatke) i
    - da pomaže timu u izradi programskog proizvoda, odgovarajući na nejasnoće u trenutku kada su nastupili problemi.

Aktivna uloga naručitelja obuhvaća razgovor, aktivno slušanje, postavljanje pitanja, pisanje korisničkih priča, pomaganje prilikom definiranja testova prihvaćenosti i pružanja povratnih informacija tijekom cjelokupnog razvoja.

12. **Standardi u kodiranju.** Kako svaki programer ima svoj stil, potrebno je dogovoriti pravila ili standard pisanja programskog koda, kako se u budućnosti ne bi previše vremena trošilo na razumijevanje tuđeg koda. Cilj je imati jedan konzistentan stil kodiranja koji se proteže kroz cijelu aplikaciju u svrhu lakšeg čitanja, ispravljanja i održavanja. Među standarde u kodiranju spadaju formatiranje koda, struktura koda, konvencije imenovanja, komentari i upravljanje greškama i iznimkama.

6.1.2.1.1.5. Projektne uloge i odgovornosti

Navedimo još kratko **uloge i odgovornosti** koje obavljaju članovi tima primjenjujući XP metodiku. Prema [Abrahamsson, P. *et al.*, 2002., str. 21.] to su:

- **Programer.** Čini srce XP-a. Glavni mu je zadatak rad s programskim kodom čineći ga većim, jednostavnijim i bržim. Zadaci kodiranja mogu se rezimirati u: *testiraj-kodiraj-refaktoriraj*, što je u suprotnosti s tradicionalnim pristupom i *dizajniraj-kodiraj-testiraj*. Uz razvoj koda, u kojem do izražaja dolaze vještine kao što su osjećaj za jednostavnost i definiranje algoritama, zadaća programera je i dobra komunikacija s ostalim članovima tima što naročito dolazi do izražaja tijekom programiranja u paru.
- **Naručitelj (budući korisnik).** Najbolje poznaje *što* je potrebno programirati. Uz osnovne komunikacijske vještine mora razviti sposobnost pisanja dobrih korisničkih priča, određivanja njihovih prioriteta i dobro definiranje povrtnih informacija.
- **Tester.** Njegova je odgovornost definiranje, pokretanje i obrađivanje rezultata testova. Predstavlja poveznicu između programera i naručitelja.
- **Kontrolor.** Osoba koja vodi računa o dogovorenim i završenim zadacima u pojedinoj iteraciji. Mjeri količinu obavljenog posla XP tima. Prati projekt prema nekoliko definiranih metrika.
- **Trener.** Osoba odgovorna za cjelokupan proces – vođa i mentor. Treba "dublje" razumjeti svaku ulogu tijekom procesa razvoja, znati koje alternativne prakse mogu pomoći riješiti određene probleme, kako drugi primjenjuju XP, na kojim idejama se temelji XP i kako ih povezati s trenutnom situacijom.

- **Konzultant.** XP tim s vremena na vrijeme treba tehničkog savjetnika u vidu konzultanta, unatoč fleksibilnosti i znanju koje posjeduje. Uloga konzultanta je pomoći timu riješiti određeni tehnički problem ili mu razjasniti nejasnoće iz domene kojom se tim bavi.
- **Manager.** Manager je osoba izvan tima koja predstavlja tim vanjskom svijetu. On oformljuje tim i pribavlja sve potrebne resurse, upravlja timom kroz organiziranje sastanaka, bilježenje napretka.

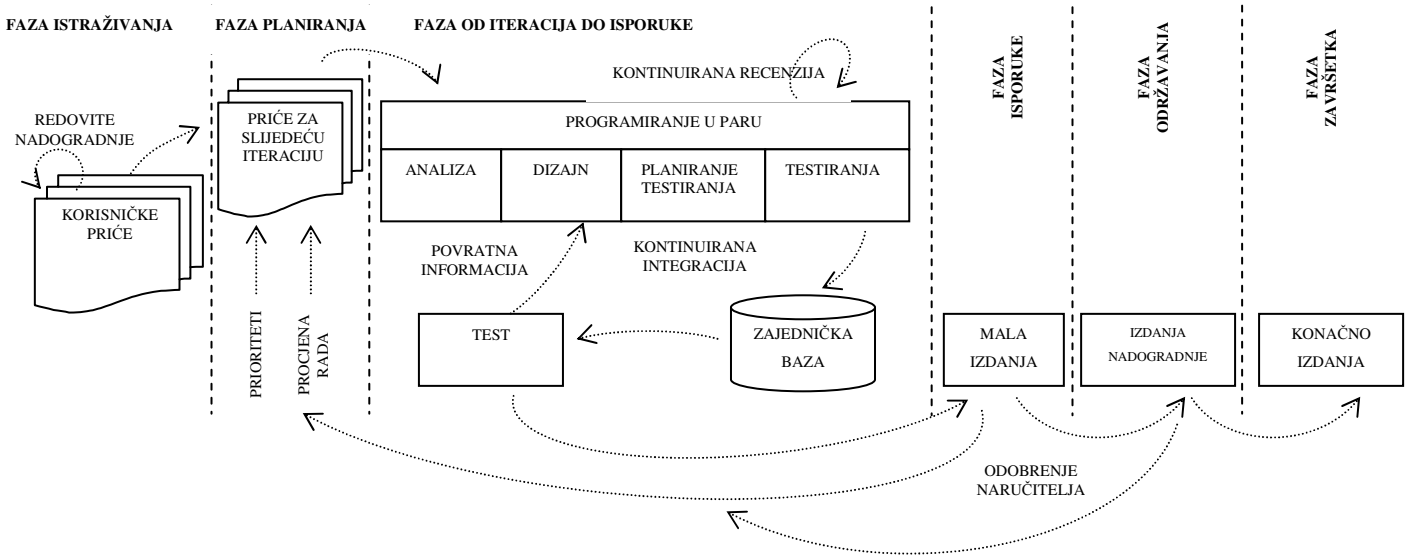
Vrlo često, posljednje tri uloge fizički obavlja jedna osoba.

6.1.2.1.2. Životni ciklus XP metodike

Životni ciklus XP metodike definiran je fazama [Abrahamsson, P. *et al.*, 2002., str. 19-21.]:

- istraživanje (eng. *Exploration Phase*),
- planiranje (eng. *Planning Phase*),
- od iteracije do isporuke (eng. *Iterations to Release Phase*),
- isporuka (eng. *Productionizing Phase*),
- održavanje (eng. *Maintenance Phase*) i
- završetak (eng. *Death Phase*).

Slika 6-3. prikazuje faze razvoja XP metodike koje će se u nastavku kratko obrazložiti.



Slika 6-3. Faze XP metodike.

Izvor: [Abrahamsson, P. *et al.*, 2002., str. 19.]

Faza istraživanja. Obuhvaća definiranje osnovne funkcionalnosti opisane u obliku *korisničkih priča* i istraživanje inicijalne arhitekture sustava oblikovanjem inicijalne *arhitekturne oštrice* – jednostavan prikaz ili model koji služi za istraživanje potencijalnih problema i rješenja. Ujedno, nužno je i definiranje opsega projekta kao i aktivno upoznavanje projektnog tima s alatima, tehnologijom i praksama koje će biti korištene u razvoju. Tehnologija koja se želi koristiti, testirat će se u određenoj mjeri kako bi se vidjelo da li svojim mogućnostima zadovoljava zahtjeve. Istraživanje mogućnosti realizacije željene arhitekture započinje identificiranjem *metafore sustava*. *Metafora* predstavlja konceptualnu skicu u kojoj su identificirani ključni objekti i opisuje kako se sustav namjerava razviti (kao predložak može poslužiti i neko već od prije poznato rješenje koje se revidira). Metafora se definira prilikom oblikovanja *arhitekturne oštrice* koje se provodi na početku projekta, najčešće prije prve iteracije (nulta/pre iteracija) ili na samom početku prve iteracije u cilju dobivanja kostura buduće arhitekture sustava. Kako naručitelj treba imati aktivnu ulogu, nastoji ga se osposobiti za kvalitetno pisanje korisničkih priča. Ključno pitanje ove faze je: "*Mogu li programeri sigurno procijeniti vrijeme potrebno za implementaciju korisničke priče?*". Trajanje faze najviše ovisi o obimu funkcionalnosti i tome koliko je tehnologija koja će se koristiti poznata programerima. Rizici ili problemi koji se mogu pojaviti u ovoj fazi su: nedovoljno znanje ili iskustvo kako bi započelo s implementacijom i uspješno razvio programski proizvod, nedovoljno istražene mogućnosti arhitekture sustava i nedovoljno poznavanje razvojnih tehnologija.

Faza planiranja. Cilj faze je na temelju definiranog opsega projekta i korisničkih priča provesti *planiranje izdanja*. Definiranim *planom izdanja* određuje se koje će se korisničke priče realizirati u pojedinoj iteraciji, kao i redoslijed koji se utvrđuje prema prioritetu pojedine korisničke priče unutar iteracije. Zatim se pristupa procjenjivanju vremena potrebnog za implementaciju pojedine priče te se konsenzusom naručitelj i projektnog tim određuje datum izdavanja prve verzije koja će obuhvaćati samo skup najprioritetnijih korisničkih priča; one koje predstavljaju kostur funkcionalnosti. Faza planiranja obično traje od par sati do maksimalno nekoliko dana. Neki od rizika koji se mogu pojaviti u ovoj fazi su: nerealno određivanje datuma izdavanja najprioritetnijeg skupa korisničkih priča i pogrešno određeni prioriteti implementacije korisničkih priča.

Faza od iteracije do isporuke. U nizu malih iteracija cilj je razviti dio funkcionalnosti sustava definiran korisničkim pričama određenim u *planu izdanja*. Stoga se najprije razvija *plan iteracije* kojem su u fokusu korisničke priče koje su odabrane za tu iteraciju te se za njih

planiraju aktivnosti (modeliranje, programiranje, testiranje i integracija) iteracije do prve isporuke. Stoga se postavlja pitanje: "*Koja je najvažnija stvar koja se mora implementirati u ovoj iteraciji?*". Iteracija započinje *sastankom s nogu* tijekom kojeg se modelira uz raspravljanje o aktivnostima i problemima vezanim uz tekuću realizaciju. Nakon sastanka započinje uparivanje članova tima i dodjeljivanja zadatka. Slijedi kratak inicijalni dizajn para u kojem se definira strategija razvoja pa testiranje, kodiranje i refaktoriranje. Na kraju svake iteracije pokreće se funkcijski test (obično test prihvatljivosti) koji kreira naručitelj. Prilikom implementacije novih testova i nove funkcionalnosti, potrebno je neprekidno izvršavati integraciju. Kod se treba nalaziti u zajedničkom repozitoriju. Na kraju posljednje iteracije sustav je spreman za isporuku. U *n* iteraciji mogu se pronaći i nove korisničke priče koje je potrebno analizirati, odrediti kako se one odnose s već postojećim te na temelju toga napraviti potrebu korekciju i revidiranje plana izdanja. Implementacija svake iteracije obično traje od jednog do četiri tjedna. U ovoj fazi rizici koji se mogu pojaviti su: problemi s razvojnim alatima koji se koriste ili nedovoljno dobro izgrađena arhitektura sustava.

Faza isporuke. Prikazuje nastajanje malih, ali čestih isporuka. Prije isporuke, važno je da svi testovi iz faze *Od iteracije do isporuke* budu uspješno provedeni te da se provjere performanse sustava prije nego što sustav bude isporučen kupcu. Iz ove faze postoje i povratne veze prema fazi planiranja gdje se kreiraju i planiraju nove korisničke priče. Male isporuke povezane su i s isporukama tijekom nadogradnje u fazi održavanja. Za vrijeme faze isporuke, iteracije trebaju biti skraćene od tri tjedna na jedan tjedan uz obavezno održavanje dnevnih sastanaka. Problemi u testiranju i s performansama sustava potencijalni su nositelji rizika u ovoj fazi.

Faza održavanja. Nakon prve isporuke, svakom novom iteracijom tj. dodavanjem funkcionalnosti, tim mora održavati implementaciju sustavu u konzistentnom stanju. Razvoj sustava koji se svako malo isporučuje s novom funkcionalnošću nije isti kao i razvoj sustava koji se isporučuje tek kada je razvijena cjelokupna funkcionalnost. Integracija novih funkcionalnosti sa sustavom koji je u radu predstavlja potencijalni rizik ove faze.

Faza završetka. Predstavlja trenutak kada je konačna verzija u radnom okruženju i više ne postoji novi funkcionalni zahtjev koje bi bilo potrebno implementirati. U toj fazi neophodno je napisati i isporučiti naručitelju prikladnu dokumentaciju sustava. Faza nastupa i u dva slučaja prije planiranog kraja i to: kada sustav ne pruža željene rezultate ili kada daljnji razvoj

sustava i dodavanje novih funkcionalnosti postaje preskupo što su uz mogućnost da sustav ne ispunjava željene rezultate prije planiranog kraja glavi rizici ove faze.

XP metodika razvoja softvera nije primjenjiva u svim projektima. XP se teško implementira u slučajevima kada:

- postoji jak otpor prema XP načinu programiranja,
- projekti zahtijevaju veliku količinu dokumentacije,
- je prekovremeni rad u okruženu uobičajena praksa,
- ljudi u timu ne mogu ili ne žele međusobno komunicirati iz različitih razloga (objektivnih ili subjektivnih),
- postoje veliki timovi i
- se nalazimo u okolini u kojoj se povratne informacije dugo čekaju.

U zaključku napominjem kako sve metodike imaju određena ograničenja, pa ih tako imaju i agilne metodike. Ujedno ne postoji najbolja ili univerzalna metodika primjenjiva u svim situacijama. Agilne metodike su najprikladnije u razvoju manjih ili srednje velikih projekata kao i za softver osobne upotrebe. Nisu prikladne u projektima s velikim timovima ili timovima čiji su članovi dislocirani kao i u onim projektima u kojima se zahtjeva složena interakcija s drugim sustavima i opsežna dokumentacija [Sommerwille, 2007., str. 398.].

6.2. Utjecaj MDD paradigme na današnje metodike razvoja programskog proizvoda

Detaljno razmatranje današnjih skupina metodika kao i njezinih vodećih predstavnika neminovno vodi do pitanja: *Kako se MDD paradigma razvoja uklapa s postojećim metodikama?*

Obje skupine metodika primjenjuju se u današnjem, *tradicionalnom* (klasičnom) *razvoju,* koji se temelji na generičkim fazama razvoja: planiranje, analiza, oblikovanje, kodiranje, testiranje i isporuka. U praksi, tijekom razvoja novog sustava, najviše vremena i rizika otpada na ručno kodiranje.

No, MDD paradigma mijenja pogled na razvoj programskog proizvoda. Ona podiže razinu apstrakcije, stavljajući naglasak na početne faze razvoja, naročito fazu analize tijekom koje se

razvijaju *modeli*, na temelju kojih bi se, primjenom razvojnog okruženja, dio ili čak cijeli programski kod trebao generirati. Modeli koji nastaju trebaju biti točni, konzistentni s dovoljno detalja kako bi se mogla provesti automatizacija primjenom definiranih transformacija. MDD paradigma još je uvijek u razvoju te neki od problema nisu riješeni (ograničene mogućnosti razvojnih alata (provođenje transformacija), notacije za definiranje modela, jezici za opis transformacija modela i sl.).

Uspoređujući ova dva "svijeta" i uzimajući u obzir temeljne ideje, mogu se primijetiti razlike. Neke od klasičnih faza nastoje se automatizirati, pomaknut je naglasak s nižih (implementacije) u više (modeliranje) razine apstrakcije, pojavljuju se *nove uloge* u timu koje traže nove oblike znanja kao i *nove aktivnosti* (definiranje transformacije, provođenje transformacija nad modelima, modeliranje u nekom DSL jeziku). No, navedene razlike ne isključuju primjenu današnjih metodika za MDD razvoj već upravo suprotno. One ukazuje na elemente kojima je te metodike moguće unaprijediti, prilagoditi i po potrebi obogatiti za MDD razvoj. Stoga u nastavku slijedi detaljnije razmatranje utjecaja MDD paradigme na svaku skupinu metodika.

6.2.1. MDD i formalne metodike

Kako se MDD paradigma može promatrati kao nova stepenica u evoluciji razvoja programskog proizvoda, koji se uglavnom temelji na formalnim metodikama (agilne su se pojavile tek sredinom '90-tih), sve gore navedene razlike primarno se odnose na formalne metodike. U kojem obimu MDD paradigma trenutno zadovoljava glavne karakteristike formalnih metodika, navedene u točci 6.1.1., vidljivo je u *tablici 6-5*.

Tablica 6-5. Uklapanje MDD paradigme s glavnim karakteristikama formalnih metodika

Karakteristike formalnih metodika	**MDD paradigma**
Temelje se na opsežnom planiranju i detaljnom opisu problemske domene.	U planiranju naglasak se stavlja na postizanje razumijevanja problemske domene kako bi se definirali upotrebljivi modeli.
Velik broj artefakata i strogo formalno opisanih aktivnosti kojih se je potrebno pridržavati tijekom razvoja SW.	Glavni tipovi artefakta su modeli i transformacije modela. Redoslijed provođenja aktivnosti potrebno je poštovati kako bi se uspješno moga provesti generiranje.
Zahtijevaju vrijeme, disciplinu i veliku količinu dokumentacije koja mora pratiti cijeli razvojni ciklus.	- Vrijeme razvoja i izrade dokumentacije se smanjuje primjenom generatora aplikacije. - Prilikom definiranja modela i M2M i M2C transformacija zahtjeva se naročita disciplina. - Pisanje dokumentacije nije naknadna aktivnost, nego dio procesa specifikacije modela koja se *generira* ta temelju specifikacije modela.

Segmenti koje je potrebno prilagoditi, promijeniti ili proširiti kako bi formalne metodike bila prikladne za MDD razvoj odnose se na:

- *Tim*: prisutne su nove *uloge* i potreba za dodatnim *znanjem* karakterističnim za MDD razvoj.
- *Proces razvoja*: promijenjen je značaj i opsega aktivnost u pojedinim fazama, ali su i uvedene nove aktivnosti kojima se treba podržati automatizacija. Analiza i razvoj modela s pripadajućim transformacijama imaju veći značaj od kodiranja.
- *Tehnologiju*: razvojno okruženje i njegove mogućnosti predstavlja kritičnu

komponentu uspjeha razvoja.

- *Modeliranje*: prepoznata je važnost modela koji se žele iskoristiti kao gradivni blokovi.
- *Problemska domena*: neke domene prikladnije su za MDD razvoj od drugih. Zbog problema koji su još uvijek prisutni pokušavaju se razviti jezici za modeliranje prikladni za pojedine domene.

6.2.2. MDD i agilne metodike

Može li se MDD paradigmu promatrati u kontekstu agilnih metodika, kad na prvi pogled naglašavaju različite aspekte razvoja? Namjera MDD paradigme je kompatibilnost i s principima agilnog razvoja [Bettin, 2004., str. 13.]. Upravo s tom idejom skupina istraživača odlučila je pokrenuti i pristup pod nazivom **agilna MDD paradigma** [Ambler, 2007., str. 1.].

U nastavku, usredotočit ću se na prikaz razlika i sličnosti ova dva koncepta.

Razlike koje proizlaze, temelje se na činjenici da agilne metodike stavljaju naglasak na ljude, dok se MDD oslanja na naprednu tehnologiju kojom se definiraju tehnološki neovisni modeli i generira programski kod. Druga temeljna razlika vidljiva je u činjenici da *agilni razvoj* ističe programiranje, a *MDD razvoj* modeliranje. Kroz *tablicu 6-6.* navodim neke aspekte u kojima je razlika jasno uočljiva.

Tablica 6-6. Razlike agilnih metodika i MDD paradigme

Aspekt	Agilne metodika	MDD paradigma
Ljudi	Predstavljaju najznačajniji faktor s najvišim prioritetom tijekom razvoja programskog proizvoda. Socijalno je obojena.	Ljudi se promatraju s tehnološke perspektive kroz *uloge* u definiranim procesima. Kroz paradigmu se pojavljuju nove uloge. Socijalni aspekt se zanemaruje.
Proces razvoja	Sve aktivnosti tijekom procesa nisu detaljno definirane. Naglasak je u aktivnostima testiranja i kodiranja. Primjenjuje se iterativan i inkrementalan pristup.	U odnosu na tradicionalni proces razvoja neke od faza (dizajn, kodiranje, testiranje) nastoje se automatizirati, a neke aktivnosti su nadodane kako bi se postigao naglasak na definiranju modela i generiranju koda. Ključna faza je analiza. Definiranje i provođenje transformacija kritičan je dio procesa.
Tehnologija i	Ima najniži prioritet. Štoviše	Najveći značaj pridaje se tehnologiji. Ovisno o

razvojna okruženja	poželjno je da alati budu što jednostavniji kako bi se ostavio dojam potpune kontrole čovjeka nad razvojem.	mogućnostima alata procjenjuje se uspješnost provođenja paradigme.
Modeliranje	Ima marginalan značaj. Razumijevanje sustava postiže se izradom prototipa.	Centralna aktivnost. O njezinoj kvaliteti ovisi daljnji razvoj (uspjeh, propast).
Kodiranje	Ručno.	Nastoji se provesti što je moguće više (za sada parcijalnih) generiranja programskog koda. Težnja je za cjelokupnim generiranjem programskog koda iz dobro definiranog modela. Ponovno generiranje zbog promjena u zahtjevima ne predstavlja problem ukoliko je promjena provedena nad modelom.
Problemska domena	Okruženje podložno dinamičnoj promjeni zahtjeva.	Okruženje stabilnih zahtjeva.

Iako se na prvi pogled može zaključiti da se MDD paradigma bolje uklapa s tradicionalnim metodikama pri čemu su neke faze nastoje automatizirati, moguće je pronaći zajedničke točke i s agilnim pristupom. Najlakše se to može uočiti kroz principe koje naglašava agilni razvoj, a navedeni su u točci 6.1.2. *Tablica 6-7.* prikazuje neke sličnosti.

Tablica 6-7. Sličnosti agilnih metodika i MDD paradigme

Princip	Agilne metodika	MDD paradigma
Individualnost i interakcije važniji su od procesa i alata	Naglašavaju vezu i zajedništvo programera i ljudskih uloga u suprotnosti s institucionaliziranim procesima i razvojnim alatima.	Struktura i aktivnosti tima nisu strogo definirani. Iako generatori aplikacije igraju ključnu ulogu alati su individualni tj. razvijaju se na temelju specifičnih domena.
Softver koji radi važniji je od opsežne dokumentacije	Ažuriraju se ključni artefakti na svim razinama apstrakcije. Nastoji se izbjeći nekonzistentnost.	Uvijek se ažuriraju modeli na najvišoj razini apstrakcije. Konzistentnost se postiže generiranjem iz tog modela. Usmjerena na SW koji će korisnik validirati.
Suradnja s klijentom važnije je od striktnih	Prisutan je stalni angažman naručitelja projekta.	Ne posvećuje neku posebnu pažnju, ali otvara mogućnost uključivanja naručitelja i to

ugovora		intenzivno za testiranje aplikacije. Kako MDD omogućava brzu transformaciju iz modela u aplikaciju ne gubi se vrijeme ukoliko naručitelj nije zadovoljan s realizacijom nekog zahtjeva. No napomenimo kako ovo za sad nalazi uporište u na teorijskoj razini MDD paradigme.
Reakcija na promjene važnije je od slijeđenja plana	Upravljanje promjena važnije je od slijeđenja plana koji ne odgovara novo definiranim (promijenjenim) zahtjevima.	Generiranje programskog koda, omogućava brzu reakcija na novi zahtjev ili promjenu u već postojećim. Dovoljno je promijeniti model i ponovno pokrenuti generiranje, što je znatno jednostavnije nego mijenjati programski kod.

Izvor: [Bettin, 2004., str. 13, 14]

Na temelju ovih principa agilnih metodika vidimo kako je ipak moguće uklopiti MDD razvoj u kontekst agilnog razvoja. Dok su principi općeniti, i u konkretnoj primjeni MDD paradigme u kontekstu agilnog razvoja mogu se pronaći dodirne točke. Primjena razvojnih alata (MDD generatora) prirodno dovodi do realizacije nekih agilnih praksi, jer donosi mogućnost brze reakcije na promjene u korisničkim zahtjevima, smanjenju vrijeme trajanja iteracije, ubrzava dobivanje povratne informacije od korisnika, štedi vrijeme u aktivnostima koje se ponavljaju i koje su podložne pogreškama (eng. *error prone*) te smanjuje rizik. Stoga, usprkos razlika koje postoje, smatram kako se MDD paradigma može promatrati i u kontekstu agilnih metodika i da se oni međusobno ne isključuju. Kako agilnost naglašava općenite vrijednosti i principe te ne određuje niz koraka koje je potrebno primjenjivati, ostaje otvoreno pitanje stupanja prikladnosti i kompatibilnosti agilnih metodika u kontekstu MDD razvoja koji najviše ovisi o nizu čimbenika karakterističnih za individualan projekt. Jasno je da ne postoji univerzalno najbolji pristup te se tijekom projekta MDD razvoja mogu uzeti i nastojati primijeniti neki aspekti agilnog razvoja.

Iz ove kratke analize može se vidjeti da je današnje metodike uz odgovarajuće prilagodbe i promjene moguće koristiti za projekte MDD razvoja. No, trenutno se u literaturi ne mogu pronaći primjeri s konkretnim procesom primjene MDD u okviru današnjih metodika. Djelomičan uzrok leži u činjenici da se MDD paradigma još razvija, kao i da postoje mnogi problemi koji šire skeptičnost ili suzdržanost primjene MDD paradigme. Osobno smatram kako MDD paradigma ima više smisla u kontekstu tradicionalnih metodika, a da će u kontekstu agilnih metodika njezin doprinos biti skroman.

6.3. Integracija

Predmet istraživanja ove disertacije je prikazivanje prikladnosti i primjenjivosti jednog tipa artefakta – SW uzorka, u MDD paradigmi. Kroz peto poglavlja razvijen je metodološki okvir koji podržava cijeli životni ciklus uzoraka u MDD razvoju. Kako bi zaokružio cjelokupno razmatranje, u nastavku slijedi prikaz mogućnosti integracije razvijenog okvira u kontekstu najzastupljenije formalne i agilne metodike.

Integracija će se provesti na način da će se u proces razvoja (faze razvoja) odbrane metodike "spustiti" segmenti razvijenog okvira s aktivnostima vezanim uz uzorke koji podržavaju MDD paradigmu. Napominjem da će se integracija promatrati samo u kontekstu aktivnosti koje podržavaju MDD paradigmu, a vezane su uz uzorke. Prilagođavanje spektra ostalih aktivnosti koje se mogu provesti kako bi odabrani proces u potpunosti podržavao sve koncepte MDD paradigme izlazi iz okvira ove disertacije te se oni neće razmatrati u kontekstu integracije.

Znači, ideja nije "prekrojiti" postojeći proces razvoja u proces prikladan za MDD nego u današnje procese dodati aktivnosti vezane uz uzorke kako bi se njima proveo MDD razvoj.

Ukratko da ponovim: razvijeni okvir se proteže u dvije dimenzije (definiranje ekspertize u obliku uzoraka i MDD razvoj SW podržan uzorcima) te kroz *životni ciklus uzorka* obuhvaća segmente: identificiranje, primjena, razvoj i upravljanje.
Napominjem da je okvir definiran i za promatranje u užem kontekstu – izvan MDD projekta razvoja, tj. da ne ovisi o samom MDD razvoju SW. No u nastavku će se prikazati integracija razvijenog okvira u odabranoj metodici u kontekstu MDD projekta razvoja podržanog uzorcima.

6.3.1. Integracija razvijenog okvira s tradicionalnim metodikama - RUP

Najzastupljenija metodika, koja podržava skup najboljih praksi u tradicionalnom razvoju, i detaljno je opisana kroz točku 6.1.1.1. je RUP.

Kako integrirati razvijeni okvir primjene uzoraka u MDD projektu razvoja SW s RUP metodikom?

Integracijski okvir promatrat će se u kontekstu RUP faza razvoja i definiranih disciplina. U okviru svake RUP faze (*inspekcija, elaboracija, konstrukcija, tranzicija*) analizirat će se aktivnosti i zadaci svake discipline te će se nastojati integrirati s osmišljenim aktivnostima koje su grupirane po segmentima razvijenog okvira (*identificiranje*, *primjena*, *razvoj*, *upravljanje*). Potrebno je imati na umu da se svaka faze može odvijati kroz nekoliko iteracija pa će se neke od definiranih aktivnosti izvoditi više puta.

6.3.1.1. Integracija razvijenog okvira s fazom *Inspekcije*

Ključni *cilj* faze inspekcije u kontekstu MDD razvoja i uzoraka je, na temelju analiziranja problema i razumijevanja potreba naručitelja, utvrditi izvodljivost MDD razvoja. Drugim riječima, potrebno je identificirati mogućnosti automatizacije i utvrditi mogu li se potrebe naručitelja opisati kroz funkcionalne i nefunkcionalne zahtjeve koji bi se realizirali MDD razvojem.

Razmotrimo što to detaljno podrazumijeva u kontekstu svake discipline.

Modeliranje poslovnog sustava:

Ovu se disciplinu promatra kada se u širem kontekstu želi detaljno razumjeti, analizirati i unaprijediti poslovne procese. Ono se može izvoditi kao dio projekta razvoja ili kao odvojeni projekt na kojem će se temeljiti nekoliko projekata razvoja SW. Prilikom modeliranja mogu se identificirati procesi koji su već prepoznati i oblikovani u obliku *uzorka procesa* (vidi katalog *Process Patterns*) kao najbolja praksa. Kako se nalaze na najvišoj razini apstrakcije, realizirani su u obliku tekstualne specifikacije, u kojoj se može navesti s kojim MDD uzorcima na nižoj razini apstrakcije je moguća njihova kasnija realizacija. Ukoliko se provodi ova disciplina, aktivnosti se odnose na *Odabir i lociranje uzoraka* i *Kandidiranje problema za uzorka* iz segmenta ***Identificiranje uzoraka***.

Definiranje zahtjeva:

Tijekom ove discipline najprije se prikupljaju informacije o problemu kako bi se na temelju njih definirali kriteriji pretraživanja repozitorija potrebni za traženje prikladnih uzoraka. Identificiranim kriterijima započinje aktivnost *Odabir i lociranje uzoraka* iz segmenta ***Identificiranje uzoraka*** koji se proteže i u slijedećoj disciplini. Uz ovu aktivnost

odvija se i aktivnost *Kandidiranje problema za uzorka*.

Analiza i oblikovanje:

U ovaj fazi tijekom ove discipline naglasak je na pretraživanju kataloga i provođenju aktivnosti započetih u prethodnoj disciplini (odabir i lociranje, kandidiranje). Po neki se puta može desiti da se za isti problem pronađe više uzoraka koji zadovoljavaju postavljene kriterije. Stoga je u cilju pronalaska uzorka koji će biti prikladniji za oblikovanje buduće arhitekture potrebno u sklopu tih aktivnosti provesti i detaljnu analizu specifikacije tih uzorka. Tijekom ove discipline provodi se i aktivnost *Inicijalna analiza isplativosti razvoja novog uzorka* kojom se utvrđuju rizici nastanka novih uzoraka. Analiziraju se uzorci koji zadovoljavaju postavljene kriterije i određuje se koji od njih će se najbolje uklapali u model buduće arhitekture.

Implementacija:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Testiranje:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Isporuka:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Upravljanje konfiguracijom i promjenama:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Upravljanje projektom:

Na temelju provedene aktivnosti *Inicijalna analiza isplativosti razvoja novog uzorka*, kojom su utvrđeni rizici nastanka novih uzoraka, provjerava se da li se i kako to uklapa u resurse cjelokupnog projekta. Ukoliko analiza zadovoljava odobrava se pokretanje slijedeće aktivnosti iz segmenta ***Identificiranje uzoraka*** kojim se prelazi u novu fazu *Elaboraciju*.

Ujedno treba podržavati provođenje aktivnosti *Upravljanje uzorcima u repozitoriju* ukoliko je to potrebno.

Održavanje razvojnog okruženja:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

6.3.1.2. Integracija razvijenog okvira s fazom *Elaboracije*

Rezultat faze inspekcije su stabilni korisnički zahtjevi i skup uzorka koji bi se mogli primijeniti tijekom razvoja ili bi se na temelju uočenih problema trebali razviti. U fazi elaboracije u kontekstu MDD razvoja i uzoraka *cilj* je definiranje arhitekture budućeg sustava što u kontekstu MDD razvoja u velikoj mjeri otpada na modeliranje i definiranje MDD modela kojima će se opisati ta arhitektura. Najznačajnije je da modeli budu definirani s dovoljno detalja kako bi se mogla pokrenuti njihova automatizacija. U cilju izgradnje, modeliranja takve arhitekture, primjena uzoraka imat će vrlo izraženu ulogu. Drugim riječima ovo je ključna faza za MDD razvoj u kojoj će do izražaja doći aktivnosti iz sva četiri segmenata razvijenog okvira (identificiranje, primjena, razvoj i upravljanje).

Aktivnosti u disciplinama opisane su u nastavku:

Definiranje zahtjeva:

Kako se povećava razumijevanje i stupanj detaljizacije korisničkih zahtjeva nastavlja se s identificiranjem potencijalnih uzoraka iz prošle faze. Nadalje, za sve opisane probleme koji su predstavljali kandidate za novi uzorak, a nisu odabrani za razvoj, pokreće se aktivnost *Oblikovanje problema u antiuzorak* kako bi se u slijedećim projektima uklonile nedoumice o mogućoj realizaciji tih problema kao uzorka. Nad kandidatima problema za koje se smatra opravdanim razvijati uzorak provodi se aktivnost *Definiranje zahtjeva za razvoj novog uzorka*.

Analiza i oblikovanje:

Teži se kreiranju globalne slike o budućoj arhitekturi te se svi odabrani uzorci stavljaju u međusobni kontekst. Kako je pokrenut razvoj novi uzoraka započinje se sa segmentom ***Razvoj uzoraka*** i to odvijanjem aktivnosti: *Analiza*, *Oblikovanje rješenja* i *Specificiranje uzorka*.

Implementacija:

Iako se pod implementaciju misli na pisanje programskog koda ovdje se odnosi na glavnu aktivnost MDD razvoja, a to je **modeliranje i nastanak MDD modela.** Teži se da dijelovi modela što je moguće više budu strojno čitljivi (izvršivi) što se postiže primjenom uzoraka. Tako za identificirane probleme za koje su pronađeni prikladni uzorci započinju aktivnosti koje su vezne za segment ***Primjena uzoraka***. To su: *Uvoz i instalacija uzorka*, *Instanciranje uzoraka* i *Povezivanje instance uzorka s elementima modela*. Na prijelazu faze elaboracije u

konstrukciju paralelno se mogu odvijati aktivnost *Kreiranje i konfiguriranje instance transformacije* iz segmenta ***Primjena uzoraka***, te *Implementacija* iz segmenta ***Razvoj uzoraka***. Pažnju je potrebno usmjeriti na usklađivanje tih aktivnosti. Ako se razvija novi uzorak za MDD projekt onda aktivnosti *Primjene* mogu uslijediti tek nakon njegove realizacije.

Testiranje:

Za uzorke koji će se razvijati potrebno je definirati testove koji će se provesti nakon što se isti razviju u kasnijoj fazi. Stoga se pokreće aktivnost *Pisanje testova*.

Isporuka:

Pred kraj faze u kontekstu MDD razvoja i uzoraka potrebno je raspolagati s početnim verzijama MDD modela.

Upravljanje konfiguracijom i promjenama:

Tijekom primjene moguće su potrebe za prilagođavanje već odabranog uzorka. Ujedno se prilikom razvoja novog uzorka mogu pojaviti potreba za izmjenom nekih već definiranih zahtjevima ili dodavanjem nove funkcionalnosti. Stoga se pokreće aktivnost *Upravljanje zahtjevima za promjenama*.

Upravljanje projektom:

Prvenstveno se odnosi na upravljanje projektom razvoja novog uzorka te se dodjeljuju uloge i zadaci, definira se plan realizacije i vrijeme potrebno da se uzorak razvije. Ako se projekt odvija u sklopu MDD razvoja SW, uključuje i finije terminiranje i usklađivanje svih projektnih aktivnosti kao i detaljniju procjenu rizika koji su povezani s njim. Aktivnost koja se pokreče je *Pokretanje projekta razvoja novog uzorka*.

Ujedno treba podržavati provođenje aktivnosti *Upravljanje uzorcima u repozitoriju* ukoliko je to potrebno.

Održavanje razvojnog okruženja:

Podrazumijeva sve potporne aktivnosti vezane uz rad razvojnog okruženja koje će se koristiti.

6.3.1.3. Integracija razvijenog okvira s fazom *Kodiranja*

Kako je ideja MDD razvoja generiranje programskog koda iz modela, opseg posla u ovoj fazi trebao bi biti znatno smanjen. No kako danas, razvojna okruženja još nisu u mogućnosti to ostvariti i tijekom ove faze odvijaju se aktivnosti vezne uz kodiranje, ali se velik naglasak stavlja na razvoj transformacija modela. Aktivnosti povezane s uzorcima objašnjene su u nastavku, a primarno se odnose se na dio aktivnosti definiranih u segmentima ***Primjena*** i ***Razvoj uzoraka***.

Definiranje zahtjeva:

I u ovoj fazi, zbog promjena u zahtjevima, pojedini uzorci mogu postati neprikladni ili su uočeni dijelovi koji su do tada bili previđeni, pa se može pojaviti potreba za traženjem novih uzoraka. Stoga će možda biti potrebno pokrenuti aktivnost *Odabir i lociranje uzoraka* (ili *Kandidiranje problema za uzorka* radi li se o razvoju novog uzorka) iz segmenta ***Identificiranje uzoraka***.

Analiza i oblikovanje:

Zbog eventualnih izmjena u arhitekturi potrebno je analizirati novopronađene uzorke i uskladiti ih s do sada razvijenim modelima. U segmentu ***Razvoj uzoraka*** provode se aktivnosti: *Analiza*, *Oblikovanje rješenja* i *Specificiranje uzorka*.

Implementacija:

Nastavlja se s započetim aktivnosti iz prošle faze izvođenjem aktivnosti u segmentima ***Primjena i Razvoj uzoraka*** koje se kako je već spomenuto mogu odvijati istovremeno. Pri tome je potrebno voditi računa o usklađenosti tih aktivnosti. Ako se ne razvijaju novi uzorci u sklopu MDD projekta, onda se s aktivnostima ***Primjene*** započinje na početku te faze (ili na prijelazu). U suprotnom je potrebno pričekati da se dovrše aktivnosti ***Razvoja***.

Za definirane zahtjeve razvoja novog uzorka primarno se odvija aktivnost vezna za segment ***Razvoj uzoraka***, a to je *Implementacija*, dok se u segmentu ***Primjena uzoraka*** odvijaju aktivnosti: *Kreiranje i konfiguriranje instance transformacije*, *Pokretanje instance i dopuna programskim kodom* i *Definiranje povratnih informacija*.

Ukoliko su usvojeni novi uzorci potrebno je izvršiti i aktivnosti: *Uvoz i instalacija uzorka*, *Instanciranje uzoraka* i *Povezivanje instance uzorka s elementima modela*.

Testiranje:

U sklopu segmenta ***Razvoj uzoraka*** provodi se aktivnost *Validiranje i verificiranje* uzoraka.

Isporuka:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Upravljanje konfiguracijom i promjenama:

Na temelju povratnih informacija na kraju segmenta ***Primjene uzoraka*** kao i na temelju završetka svake iteracije te dobivenih rezultata testiranja tijekom segmenta ***Razvoja uzoraka*** provodi se aktivnost *Upravljanje zahtjevima za promjenama*. Nakon što se promjena definira pokreće se aktivnost *Realizacija zahtjeva za promjenom*.

Upravljanje projektom:

Uključuje provođenje aktivnosti *Upravljanje uzorcima u repozitoriju*.

Održavanje razvojnog okruženja:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

6.3.1.4. Integracija razvijenog okvira s fazom *Tranzicije*

U fazi tranzicije u kontekstu MDD razvoja i uzoraka *cilj* je novonastali uzorak dodati u repozitorij u omogućiti široku primjenu. U kontekstu MDD projekta nema posebnih aktivnosti vezanih uz MDD uzorke.

Definiranje zahtjeva:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Analiza i oblikovanje:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Implementacija:

Novonastali uzorak potrebno je oblikovati u standardizirani ponovno iskoristiv resurs pa se provodi aktivnost *Standardizacija uzorka u oblik ponovno iskoristivog resursa*.

Testiranje:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Isporuka:

Nakon uspješno provedene standardizacije preostaje samo aktivnost *Objavljivanje u repozitorij*.

Upravljanje konfiguracijom i promjenama:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Upravljanje projektom:

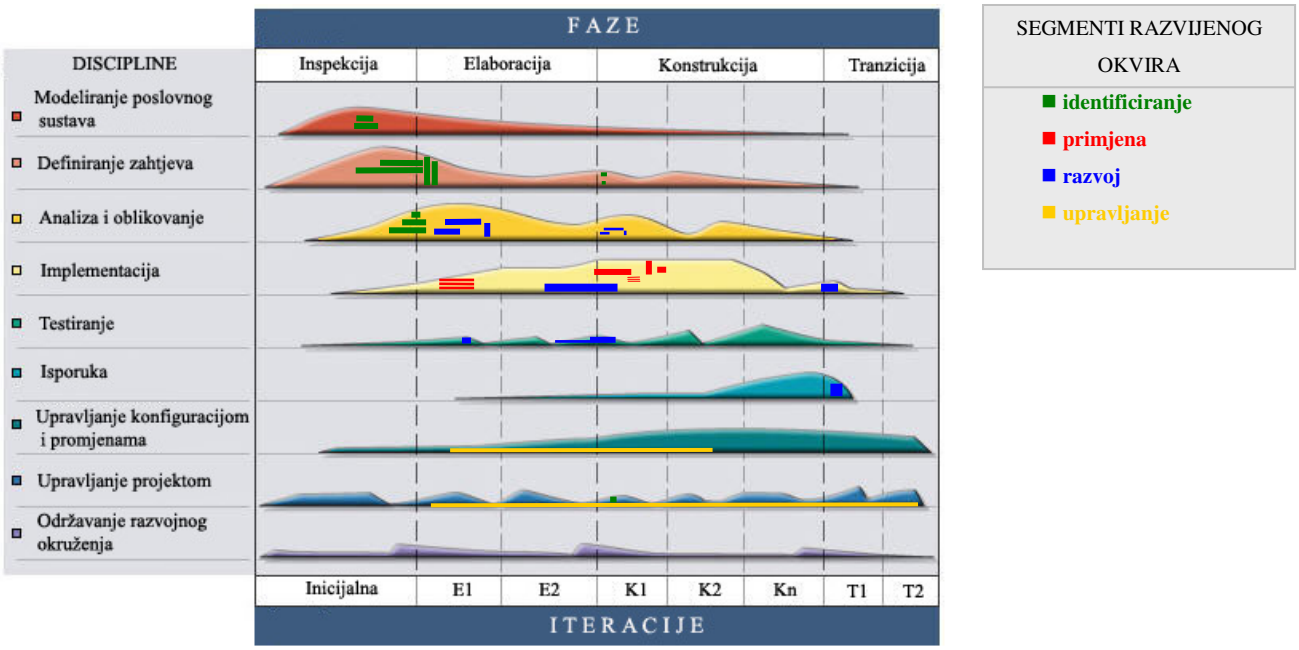
Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

Održavanje razvojnog okruženja:

Nema aktivnosti u kontekstu MDD razvoja i uzoraka.

6.3.1.5. Osvrt provedene integracije

Provedenom integracijom promatran je MDD projekt razvoja podržan RUP metodikom u koju su dodane aktivnosti kojima je podržana primjena uzorka u MDD razvoju. Opisana integracija razvijenog okvira i RUP metodike može se grafički prikazati i *slikom 6-4.* kroz koju se vide aktivnosti razvijenog okvira koje se izvode u sklopu metodike.



Slika 6-4. Integracija segmenata razvijenog okvira s RUP metodikom

Svaki segment razvijenog okvira posebne je boje, a kvadratić reprezentira aktivnost koju je potrebno provesti u kontekstu uzoraka. Najznačajnije vidljive primjedbe, svakako su vezane uz fazu *Elaboracije* i *Kodiranja*. To su faze u kojima je detaljno potrebno promijeniti intenzitet aktivnosti kao i definirane granice među fazama koje donosi MDD razvoj. Kako velik dio vremena faze *Elaboracije* otpada na modeliranje što predstavlja kritičnu komponentu MDD razvoja, brežuljak kojim se opisuje intenzitet aktivnosti svakako bi se trebao povećati. Nadalje, dosta značajnih aktivnosti odvija se na prijelazu iz faze *Elaboracije* u *Kodiranje* koji nije striktno definiran. Štoviše, u kontekstu MDD, te bi se faze mogle promatrati zajedno i to kao faze s prikladnijim nazivom *Modeliranje* i *Generiranje*. Faza *Kodiranje*, u odnosu na tradicionalan razvoj, nema toliki značaj te se u trenutku pokretanja definicija transformacije opseg posla rapidno smanjuje i usmjeruje se na dopunjavanje programskog koda. Pisanje transformacija modela predstavlja najkritičniju komponentu ove faze. Vrijeme ove faze, potrebno je smanjiti što je i jedan od ključnih doprinosa MDD paradigme. Kako su se promatrale *samo* aktivnosti definirane razvijenim okvirom, integracija je u potpunosti provedena s naglaskom na upravo navedene promjene.

6.3.2. Integracija razvijenog okvira s agilnim metodikama –XP

U nastavku donosim prikaz integracije razvijenog okvira s najzastupljenijom agilnom metodikom – XP.

Kao i u slučaju prošle skupine metodika, neće se razmatrati prilagodba XP metodike za potrebe MDD projekta, već samo integracija razvijenog okvir u kontekstu životnog ciklusa XP metodike koja bi se trebala koristi *u projektu MDD razvoja*. U okviru svake XP faze (*Istraživanje*, *Planiranje*, *Od iteracije od isporuke*, *Isporuka, Održavanje* i *Završetak*) osmišljene aktivnosti, grupirane po segmentima razvijenog okvira (*Identificiranje, Primjena, Razvoj, Upravljanje*), nastojat će se integrirati u XP metodiku.

6.3.2.1. Integracija razvijenog okvira s fazom *Istraživanje*

Od tri ključne aktivnosti koje se izvode u ovoj fazi: *definiranje opsega*, *zahtjeva* i *inicijalne arhitekture sustava*, posljednje dvije mogu se promatrati u kontekstu modeliranja. Kako se promatra samo integracija razvijenog okvira u nastavku su prikazane aktivnosti definiranih segmenata okvira.

Tijekom *oblikovanja inicijale arhitekturne oštrice* mogu se primijeniti aktivnosti iz

segmenta ***Identificiranje uzoraka***: *Odabir i lociranje uzoraka* i *Kandidiranje problema za uzorak*. Cilj je pronaći ili otkriti uzorke, koji bi se primijenili ili razvili prilikom oblikovanja modela – *arhitekturne oštrice*. Pri tome bi naglasak trebao biti da razvijeni modeli budu strojno čitljivi (izvršivi). Za probleme koji su identificirani kao kandidati za uzorak, provode se i ostale aktivnosti iz tog segmentima, a to su: *Inicijalna analiza isplativosti razvoja novog uzorka*, *Oblikovanje problema u antiuzorak* te *Definiranje zahtjeva za razvoj novog uzorka*. Ova posljednja aktivnost zapravo će biti definiranje nove *arhitekturne oštrice* u kojoj će se detaljnije razmotriti problem i potencijalno rješenje u obliku uzorka.

6.3.2.2. Integracija razvijenog okvira s fazom *Planiranje*

Glavna aktivnost ove faze je provođenje *planiranja izdanja* na temelju korisničkih priča i arhitekturnih oštrica. U kontekstu uzoraka, glavne aktivnosti su utvrđivanje prioriteta tj. koji uzorci će se razvijati u kojoj iteraciji, kao i procjenjivanje njihove problematičnosti realizacije.

6.3.2.3. Integracija razvijenog okvira s fazom *Od iteracije od isporuke*

Ovo je faza koja za projekt MDD razvoja zahtjeva značajne izmjene. U kontekstu MDD razvoja *cilj* bi trebao biti razviti model(e) i transformacije modela na temelju kojih bi se uz minimalan rad programera trebala dobiti novo malo izdanje koje zadovoljava skup definiranih korisničkih priča u iteraciji određenih *planom iteracije*.

Kako se razvija samo dio funkcionalnosti, na *sastanku s nogu*, na temelju arhitekturnih oštrica, utvrđuje se da li se u trenutnoj iteraciji uopće koriste uzorci i ako je odgovor pozitivan, utvrđuje se da li oni već postoje pa će se samo primijeniti u buduće rješenje, ili ih je potrebno razviti. Nakon toga utvrđuju se parovi koji će raditi na razvoju novih uzoraka primjenom segmenta ***Razvoj uzoraka*** koji se u potpunosti može odvijati kao XP projekt. Parovi koji rade na realizaciji modela primjenom već postojećih uzoraka slijedit će aktivnosti iz segmenta ***Primjena uzoraka***. Aktivnosti tih segmenata bit će isprepletene.

Kada se uzorak razvije i zadovolji sve testove on se sprema u repozitorij i uključuje u model. Kako uzorci reprezentirani u modelu "drže znanje", nakon provedenog generiranja, opseg aktivnosti koje programer mora provesti bit će znatno smanjen. Do izražaja dolazi reverzibilno inženjerstvo.

U slučaju kada u iteraciji test ne zadovoljava, pojavljuje se i aktivnosti iz segmenta

Upravljanja uzorcima. Primano bi se sve izmjene trebale odnositi na promjenu u modelu, a ne u nastalom kodu.

Može se zaključiti da se kroz svaku iteraciju protežu svi segmenti razvijenog okvira.

6.3.2.4. Integracija razvijenog okvira s fazom Isporuke, Održavanja i Završetka

U ovim fazama nema posebnih aktivnosti vezanih uz razvijen okvir. No prije faze *Isporuke* može se provoditi generiranje potrebne dokumentacije ili detaljnije dokumentiranje razvijenih uzoraka i usklađivanje sa standardom kako bi bili što prikladniji za repozitorij.

6.3.2.5. Osvrt provedene integracije

Kako metodika ostavlja dovoljno slobodnog prostora, način primjene XP-a tijekom MDD razvoja ostavlja mnoge dvojbe. Glavna je, da XP zagovara programiranje ako glavnu tehniku, a ne modeliranje. Sama tehnika modeliranja (još) i nije baš prihvaćena, a ako se i koristi onda je to samo za skiciranje sustava. Upravo je tu otvorena mogućnost unaprjeđenja metodike, tj prilagođavanja XP za MDD razvoj, jer se neke aktivnosti programiranja sigurno mogu unaprijediti automatizacijom modela.

Doprinos MDD paradigme u kontekstu agilnog pristupa očitovao bi se u:

- povećanju razumijevanja primjenom modeliranja u kojoj nastali modeli trebaju biti više od *skice,*
- primjeni razvojnih alata s mogućnošću generiranja programskog koda,
- mogućnosti brže reakcije na promjene u korisničkim zahtjevima,
- smanjenju vrijeme trajanja iteracije,
- uštedi na vremenu u aktivnostima koje se ponavljaju i koje su podložne pogreškama (eng. *error prone*) te
- smanjenju rizika.

Kada se uvaži ideja da se MDD razvoj odvija agilno, tj. da se prilikom svake iteracije pažnja usmjeri na definiranje ključnih modela iz kojih će se provoditi generiranje programskog koda, tada do izražaja dolazi i vrijednost primjene uzoraka kao i primjena razvijenog okvira. Kako agilni pristup naglašava što raniji razvoj programskog koda, za očekivati je značajnu primjenu naročito uzoraka oblikovanja. Podignu li se oni na razinu

kojom će se zadovoljiti definirani *uvijete primjene u razvoju temeljenom na modelima* (točka 5.1.) ostvarit će se navedena poboljšanja. Tada primjena razvijenog okvira postaje prirodnom.

Razlaganjem segmenata razvijenog okvira nedvojbeno se može uočiti kako se glavni dio svih aktivnosti provodi samo u jednoj fazi i to onoj ključnoj *Od iteracije do isporuke*, dok su ostale aktivnosti realiziraju na početku razvoja, u fazi *Istraživanje*, što smatram dobrim.

Najvećom manom tijekom integracije pokazuje se što, ukoliko u iteraciji uzorak ne postoji i mora se razvijati to mora biti što prije što će kao posljedicu imati da cijela iteracija (novo izdanje) trpi. Tada se problem neće željeti, ili se neće imati vremena razvijati kao uzorak nego po načelu "bitno je da radi, i to čim prije".

Iako iz ovoga proizlazi kako je integracija moguća, osobno smatram da se XP kao metodika po filozofiji kvalitetno ne uklapa u MDD paradigmu i trenutno zahtjeva previše prilagodbe. No ona donosi jednu prednost za razvijeni integracijski okvir. Njezina primjena pokazala bi se izuzetno pogodna tijekom segmenta ***Razvoj uzoraka***, neovisno o metodici koja se koristi za projekt MDD razvoja.

7. ZAKLJUČAK

Ovim radom obuhvaćen je samo jedan aspekt u razmatranju razvoja temeljenog na modelima. Glavna nit rada, bila je istraživanje mogućnosti primjene softverskih uzoraka u kontekstu MDD paradigme.

Primano, fokus rada bio je usmjeren na:

- kreiranje okvira koji bi s metodološke strane trebao razvojnim inženjerima pomoći u primjeni uzoraka tijekom MDD projekata razvoja SW i
- integraciju razvijenog okvira s današnjim metodikama.

Ciljevi postavljeni na početku istraživanja su realizirani. Slijedi kratko obrazloženje njihove realizacije:

- Kroz analizu današnjeg stanja u SW industriji jasno su navedeni prisutni izazovi i inovacije kojima se nastoji doskočiti novonastalim problemima te je prepoznata krivulja u inovacijama koja kao izravnu posljedicu ima nastanak novih paradigmi razvoja kako SW industrija evoluira i sazrijeva.

- Kako je SW industriji nametnuta nova paradigma – razvoj temeljen na modelima, u radu je detaljno objašnjena ideja i koncept cjelokupne paradigme. Provedeno je istraživanje postojećih MDD pristupa koji su također detaljno objašnjeni.

- Kako se disertacija temelji na jednom tipu artefakta - SW uzorka, objašnjeni su relevantni koncepti vezani uz njih. Kako u industriji na području uzoraka vlada velika neorganiziranost, provedena je osnovna klasifikacija i definirana sistematizacija trenutno poznatih uzoraka. Time je postavljen temelj razmatranja uzorka u kontekstu MDD paradigme.

- Definirani su kriteriji primjene uzoraka u MDD paradigmi.

- Uspješno je razvijen okvir primjene uzoraka u razvoju temeljnom na modelima koji se sastoji od četiri segmenta kojima je pokriven cijeli životni ciklus uzoraka.

- Kako je u metodološkom pristupu razvoju programskih proizvoda esencijalna komponenta proces razvoja, istražen je odnos između MDD paradigme i suvremenih procesa razvoja.
- Provedena je integracija razvijenog okvira u kontekstu najzastupljenijeg predstavnika svake skupine metodika te je u zaključku dan osvrt na uspješnost provedene integracije.

U nastavku slijedi opis dokazivanja postavljenih hipoteza.

H1: Moguće je definirati metodološki okvir primjene uzoraka u razvoju programskog proizvoda temeljenom na modelima.

Dokazivanje hipoteze:

Ključni aspekt MDD paradigme je definiranje i primjena ponovno iskoristivih resursa te njihova mogućnost ponovnog (automatiziranog) korištenja. Upravo su uzorci u kontekstu MDD paradigme promatrani kao ponovno iskoristivi resursi kojima bi se to moglo postići. Smatram kako tijekom modeliranja, neovisno o razini apstrakcije, mogu biti identificirani problemi za koje već postoji rješenje u obliku modela ili elemenata modela u kojima je učahureno ponašanje kojim će se ubrzati modeliranje, a ujedno i omogućiti provođenje generiranja – a to su *uzorci*. Ukoliko za neki problem ne postoje takvi uzorci, ideja je da se provodi postupak modeliranja prema odabranoj metodici sve dok se (i ako se) dio problema ne identificira kao uzorak (već postojeći ili kandidat za stvaranje novog). Svaki uzorak (postojeći ili novi) treba biti jasno definiran standardiziranom specifikacijom na temelju koje će, na nižim razinama apstrakcije, primjenom predefiniranih ili razvijenih transformacija modela doći do jedne ili više implementacija u obliku modela ili teksta (kod, skripta i sl.). Ujedno kreiranje repozitorija uzoraka omogućit će primjenu istih na svim razinama apstrakcije prilikom MDD razvoja programskog proizvoda, što će pridonijeti bržem razvoju, povećanju produktivnosti, smanjenju repetitivnog programiranja i vremena kao i većoj konzistentnosti i kvaliteti rješenja. Iz tih razloga odabrana metodika treba sadržavati niz koraka i smjernica kojim će se definirati primjena uzoraka u MDD paradigmi.

Disertacija objašnjava cjelokupan razvoj okvira i sadrži primjere kako definirane korake provesti u pojedinim segmentima razvoja. Na početku razvoja metodološkog okvira definirani su uvjeti primjene uzoraka u kontekstu razvoja temeljenog na modelima, jer svi današnji uzorci ne zadovoljavaju principe MDD paradigme. Osmišljen je kontekst kao i primarne intencije koje se žele dostići u razvoju okvira. Kako okvir treba pokrivati cijeli životni ciklus uzorka definirani su segmenti primjene uzoraka u kontekstu MDD paradigme. Iz razvijenog okvira jasno proizlaze smjernice koje daju odgovor na pitanja: tko?, što?, kada i kako?

Dakle, jednoznačno i nedvosmisleno je definiran metodološki okvir primjene uzorka u razvoju programskog proizvoda temeljenog na modelima čime je dokazana hipoteza H1.

H2: Objedinjavanje predloženog metodološkog okvira primjene uzoraka s postojećim metodikama razvoja programskih proizvoda olakšava realizaciju MDD paradigme.

Dokazivanje hipoteze:

Industrijski razvoj programskih proizvoda zahtijeva odgovarajući metodološki pristup. Trenutno u industriji postoji niz faznih i agilnih metodika razvoja programskih proizvoda. Kroz rad je analiziran pozitivan i negativan utjecaj MDD paradigme na svaku skupinu metodika. Jasno je da one nisu prilagođene za MDD paradigmu, a još manje da govore o korištenju uzoraka, što se vidi u po njihovim procesima, ulaznim i izlaznim rezultatima aktivnosti i sličnim elementima. Radom je pokazano da se one uz odgovarajuće prilagodbe i promjene mogu koristiti za MDD razvoj. Stoga je predloženi metodološki okvir primjene uzoraka bilo moguće integrirati u predstavnika svake skupine metodika. Kada bi se promatrala cjelokupna integracija smatram kako je ona bila uspješnija za formalne metodike usprkos činjenici kako je obje skupine metodika potrebno doraditi za MDD razvoj, dok bi agilni pristup bio izuzetno pogodan tijekom jednog segmenta razvijenog okvira (razvoja).

Dakle, za svaku skupinu metodika bilo je moguće provesti objedinjavanje okvira u postojeći kontekst metodike.

Iz postignutih rezultata jasno proizlaze daljnji smjerovi istraživanja koje bi trebalo usmjeriti na primjenu okvira u nekom pilot projektu i definiranje cjelokupnog procesa razvoja koji bi predstavljao najbolju praksu za MDD projekte razvoja u kojem bi razvijeni okvir iz disertacije postao dio MDD procesa razvoja.

8. LITERATURA

[Abrahamsson *et al.*, 2002.] Abrahamsson, P. *et al.*: Agile Software Development Methods- Review and Analysis, VTT Publications, 2002.

[Ackerman, Gonzalez, 2007.] Ackerman, L., Gonzalez, C.: The Value of Pattern Implementations, DrDobb's Portal, 2007., <http://www.ddj.com/cpp/199204017>, (pristupano: 27.07.2007.)

[Alur et. al., 2003.] Alur, D., Crupi, J., Malks,D.: Core J2EE Patterns: Best Practices and Design Strategies, 2003, Prentice Hall / Sun Microsystems Press, 2 izdanje

[Ambler, 2005.] Ambler, S.: Microsoft Takes On the OMG- Is my model specific enough for you?, 2005., <http://www.ddj.com/dept/architect/184415791?cid=Ambysoft>, (pristupano 15.05.2007.)

[Ambler, 2007.] Ambler, W. S: Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development, 2007., <http://www.agilemodeling.com/essays/amdd.htm >, (pristupano: 15.12.2007.)

[Anti Patterns Catalog, 2007.] ***: Anti Patterns Catalog, 2007., <http://c2.com/cgi/wiki?AntiPatternsCatalog>, (pristupano 08. 08. 2007.)

[Appleton, 2000.] Appleton, B.: Patterns and Sotfware: Essential Concepts and Terminology, 2000., <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html#PatternDefinition> (pristupano: 05.04.2007.)

[Baird, 2003.] Baird, S.: Ekstremno programiranje, Sams Publishing, Kompjuter biblioteka, 2003.

[Beck, 2002.] Beck, K.: Extreme Programming Explained: Embrace Change, 1 izdanje, 2002.

[Beck, Andres, 2004.] Beck, K.; Andres, C: Extreme Programming explained: Embrace Cgange, 2 izdanje, Addison Wesley Professional, 2004.

[Beck, *et al.*, 2001.] Beck, K., *et al.*: Agile Software Development Manifesto, 2001.,

<http://agilemanifesto.org/>, (pristupano 13.12.2007)

[Bettin, 2004.] Bettin, J: Model-Driven Software Development, SoftMetaWare, 2004. <http://www.softmetaware.com/whitepapers.html> (pristupano: 15.12.2007.)

[Booch, 2004.] Booch, G.: Software architecture, software engineering, and Renaissance Jazz, 2004., <http://www-03.ibm.com/developerworks/blogs/page/gradybooch?entry=microsoft_and_domain_specific_languages>, (pristupano: 27.06.2007.)

[Booch] Booch, G.: The Visual Modeling of Software Architecture for the Enterprise, MSDN Library Visual Studio 6.0

[Bosilj-Vukšić, 2004.] Bosilj Vukšić, V.: Upravljanje poslovnim procesima, Sinergija nakladništvo d.o.o., ZAGREB, 2004.

[Brown *et al.*, 2006.] Brown, A. W., Iyengar, S., Johnston, S.: A Rational Approach To Model-Driven Development, IBM System Yournal, Vol 45, No 3. 2006., str. 463-480., <http://www.research.ibm.com/journal/sj/453/brown.html> (pristupano: 05.01.2007.)

[Brown, 2004.] Brown, A.: An Introduction to Model Driven Architecture, 2004, http://www-128.ibm.com/developerworks/rational/library/3100.html, (pristupano: 04.09.2006)

[Buschmann *et al.*, 1996.] Buschmann, F., Meunier,R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns, Volume 1, John Wiley and Sons Ltd, UK, 1996.

[Buschmann, 2007.] Buschmann, F., Henney, K., Schmidt,D.: Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Wiley & Sons, 2007.

[Buschmann, 2007.a] Buschmann, F., Henney, K., Schmidt,D.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages, Wiley & Sons, 2007.

[Caprio, 2006.] Caprio, G.: Domain-Specific Languages & DSL Workbench, 2006.

[Cernosek, Naiburg, 2004.] Cernosek,G.; Naiburg,E.: The Value of Modeling, 2004, <http://www-128.ibm.com/developerworks/rational/library/6007.html#main> (pristupano: 04.09.2006)

[Coverpages, 2004.] Coverpages: Standards for Business Process Modeling, Collaboration,

and Choreography, <http://xml.coverpages.org/bpm.html#omg> (pristupano: 04.01.2007)

[Crnković, 2007.] Crnković, I.: The Challenges Of Software Development – How Can Component Software Help?, 2007., <http://www.idt.mdh.se/kurser/cd5490/2007/varazdin/> (pristupano 22.03.2007.)

[DEV325] ***: IBM Rational University's Online Training Program: Essentials of Model Driven Architecture, DEV 325, https://www6.software.ibm.com/developerworks/dw/rational_entitled/rational_dev325/ru_startcourse.html>, (pristupano: 26.04.2007.).

[Dudney *et al.*, 2003.] Dudney, B., Asbury, S., Krozak, J., Wittkopf, K.: J2EE AntiPatterns, 2003.

[Elssamadisy, 2006.] Elssamadisy,A.: Patterns of Agile Practice Adoption, InfoQ, 2006.

[Flower, 2004.] Flower, M.: UML Distilled Third edition, Object Technology Series, 2004.

[Fowler, 1997.] Fowler, M.: Analysis Patterns: Reusable Object Models, Addison Wesley, 1997.

[Fowler, 2003.] Fowler, M.: Patterns of Enterprise Application Architecture, 2003.

[Fowler, 2006.] [Fowler, M.: Writing Software Patterns, 2006., <http://www.martinfowler.com/articles/writingPatterns.html> (pristupano 06.04.2007.)

[Fowler] Fowler, M.: UML As Programming Language, http://martinfowler.com/bliki/UmlAsProgrammingLanguage.html, (pristupano: 25.02.2007.)

[Frankel, 2003.] Frankel, S. D.: Model Driven Architecture - Applying MDA to Enterprise Computing, John Wiley & Sons, 2003.

[Gamma *et al.*, 1997.] [Gamma, E. *et al.*: Design Patterns: Elements of Reusable Object-Oriented Software, 1997., 13 izdanje.

[Gardner, Yusuf, 2006.] Gardner, T., Yusuf, L.: Combine patterns and modeling to implement architecture-driven development, 2006., <http://www.ibm.com/developerworks/ibm/library/ar-mdd2/> (pristupano: 05.01.2007.)

[Gašević *et al.*, 2006.] Gašević, D., Djurić, D., Devedžić, V.: Model Driven Architecture and

Ontology Development, Springer, 2006.

[Grand, 1999.] Grand, M.: Patterns in Java Volume 2, Wiley Publishing, Inc., 1999.

[Grand, 2002.] Grand, M.: Patterns in Java Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, second edition, Wiley Publishing, Inc., 2002.

[Grand, 2002.a] Grand, M.: Java Enterprise Design Patterns: Patterns in Java Volume 3, Wiley Publishing, Inc., 2002.

[Greenfield *et al.*, 2004.] Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories - Assembling Application with Patterns, Models, Frameworks and Tools, Wiley Publishing, 2004.

[Greenfield, J., Short, K., 2004.] Greenfield, J., Short, K.: Moving to Software Factories, 2004., http://blogs.msdn.com/askburton/archive/2004/09/20/232065.aspx, (pristupano: 25.05.2007.)

[Greenfield, 2004.] Greenfield, J.: The Case for Software Factories, Microsoft Architect Journal, 2004., <http://msdn2.microsoft.com/en-us/library/aa480032.aspx> (pristupano: 25.01.2006.)

[Hailpern, Tarr, 2006.] Hailpern, B., Tarr, P.: Model-Driven Development: The Good, the Bad, and the Ugly, IBM System Yournal, Vol 45, No 3. 2006., str. 451-461., <http://www.research.ibm.com/journal/sj/453/hailpern.html>, (pristupano: 05.01.2007.)

[Havey, 2005.] Havey, M.: Essential Business Process Modeling, O'Reilly, 2005.

[Hay, 2006.] Hay, D.: Data Model Patterns, Morgan Kaufmann, 2006.

[Hohpe, Woolf, 2004.] Hohpe G., Woolf, B.: Enterprise Integration Patterns, Addison-Wesley, 2004.

[IBM developerWorks, 2007.] IBM developerWorks: Pattern Solution, 2007., <http://www-128.ibm.com/developerworks/rational/products/patternsolutions/>, (pristupano: 08.01.2007.)

[IBM developerWorks] IBM developerWorks: Pattern Solutions, <http://www.ibm.com/developerworks/rational/products/patternsolutions/index.html?S_TACT=105AGX15&S_CMP=LP>, (pristupano: 10.03.2007.)

[IBM Software Group, 2005.] IBM Software Group: From Idea to Realization or Building the Bridge between Business Modeling and Systems Engineering, IBM Rational

Day, 2005.

[InfoQ, 2006.] InfoQ: Interview: Jim Johnson of the Standish Group, 2006., <http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS> (pristupano 26.03.2007.)

[Jacobs, 2006.] Jacobs, R.: ARCast - MDA vs. Software Factories, 2006., <http://channel9.msdn.com/Showpost.aspx?postid=156291>, (pristupano:24.6.2007.)

[Kalnins, Vitolins, 2006.] Kalnins, A.; Vitolins, V.: Use of UML and Model Transformations for Workflow Process Definitions, 2006, <http://arxiv.org/ftp/cs/papers/0607/0607044.pdf > (pristupano: 20.12.2006)

[Kardell] Kardell, M.: A Classification of Object-Oriented Design Patterns, Umeå University, master's thesis

[Kircher, Jain, 2004.] Kircher, M., Jain, P.: Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management, John Wiley and Sons Ltd, UK, 2004.

[Kleppe *et al.*, 2003.] Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture Activities: Practice and Promise, Addison Wesley, 2003.

[Kontio, 2005.] Kontio, M.: Architectural manifesto: MDA for the entrprise, 2005., <http://www-128.ibm.com/developerworks/library/wi-arch16/> (pristupano: 05.01.2007.)

[Kermek, 1998.] Kermek, D.; Mjerenje objektno orijentirane programske opreme primjenom metrika uzoraka oblikovanja, Fakultet organizacije i informatike Varaždin, doktorska disertacija, 1998.

[Kruchten, 2003.] Kruchten, P.: What Is the Rational Unified Process?, The Rational Edge, 2003., http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jan01/WhatIstheRationalUnifiedProcessJan01.pdf>, (pristupano: 02.07. 2006.)

[Kuhn, 1970.] Kuhn, S. T.: The Structure of Scientific Revolutions, 1970., The Univesity Of Chicago Press.

[Larsen, 2003.] Larsen, G.: Asset Based Development, Rational software, 2003., <http://xml.coverpages.org/Larsen-RAS200311.pdf.>, (pristupano:

02.07. 2006.)

[Larsen, 2006.] Larsen, G.: Model-Driven Development: assets and reuse, IBM System journal Vol 45, No 3, 2006., <http://www.research.ibm.com/journal/sj/453/larsen.html>, (pristupano: 05.01.2007.)

[Larsen, Lane, 2006.] Larsen, G., Lane, E.: Building SOA applications with reusable assets, Part 1: Reusable assests, recipies, and patterns, 2006., <http://www-128.ibm.com/developerworks/webservices/library/ws-soa-reuse1/>, (pristupano: 05.01.2007.)

[Ludewig, 2003.] Ludewig, J.: Models in software engineering – an introduction, Softw Syst Model (2003) 2, p.5-14, Springer-Verlag 2003, published online: 27 February 2003.

[Maleković, 2007.] Maleković, M.: Organizacijski utjecaji KM-a, 2007., <http://www.foi.hr/CMS_library/kolegiji/uz/P05OrgUtjecajiKM-a.ppt>, (pristupano: 24.07.2007.)

[Maruna, 2006.a] Maruna, V.: Knowledge Transfer: Modeling, FER 2006.

[Maruna, 2006.b] Maruna, V.: Knowledge Transfer: Fast Track to BPM, FER, 2006.

[Mc Neile, 2003.] Mc Neile: The Vision with the Hole?, Metamaxim, 2003., <http://www.metamaxim.com/download/documents/MDAv1.pdf>, (pristupano: 03.10.2006.)

[MDA Guide, 2003.] ***: MDA Guide Version 1.0.1, OMG, 2003.

[Mellor *et.al.*, 2004.] Mellor S. J., Scott, K. S., Uhl, A., Weise, A.: MDA Distilled, Addison Wesley, 2004.

[MetsSoftware, 2004.] MetsSoftware: Software & Models, 2004., <http://merlingenerator.sourceforge.net/technology/modeling.php>, (pristupano: 27.07.2007.)

[Miller, Mukerji, 2001.] Miller, J., Mukerji, J.: Model Driven Architecture, Architecture Board ORMSC, 2001., < http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01>, (pristupano: 30.06.2007.)

[MSDN] MSDN: Microsoft patterns & practies: Software Factories, <http://msdn2.microsoft.com/en-us/practices/bb190387.aspx>, (pristupano: 24.01.2007.)

[MSDNTV] MSDNTV: Software Factories: Assembling Applications Using Models,

Patterns, Frameworks, and Tools, Interview, <http://msdn.microsoft.com/seminar/shared/asp/view.asp?url=/msdntv/episodes/en/20041118softfact/manifest.xml&rate=2&WMPVer=10.0.0.4036>, (pristupano: 25.01.2007.)

[OMG, 2004.] OMG: UML Profile for Patterns Specification, OMG, 2/2004., V1.0, (formal/ 04-02-04)

[OMG, 2005.] OMG: Reusable Asset Specification OMG Available Specification Version 2.2 (formal/05-11-02), November 2005.

[OMG, 2006.] OMG: Business Process Modeling Notation Specification (dtc/06-02-01), <http://www.bpmn.org/> (pristupano: 20.12.2006)

[OMG, 2002.] Object Management Group: OMG Model Driven Architecture, 2002, www.omg.org/mda, (pristupano: 21.04.2007.)

[Picek, 2004.] Picek, R.: Modeliranje grafičkog korisničkog sučelja u objektno orijentiranom razvoju programskog proizvoda, magistarski rad, FOI., 2004.

[Pierson, 2007.] Pierson, H.: ARCast #5, 2007., <http://channel9. msdn.com/ Showpost.aspx?postid=132943>, (pristupano: 09.02.2007.)

[Riley, 2006.] Riley, M: A Special Guide-MDA and UML Tools: CASE 2.0-or the Developer's Dream, Software Development, 2006., <http://www.ddj.com/dept/architect/184415500>, (pristupano: 20.4.2007.)

[Rosengard, Ursu, 2004.] Rosengard, J.M., Ursu, M. F.: Ontological Representation of Software Patterns, 2004.

[RSA 6.0.1.] RSA 6.0.1.: Benefits of Using Patterns

[RSA, 2006.] RSA: Introduction to Modeling, (pristupano: 30.12.2006.)

[Rubinstein, 2007.] Rubinstein, D.: The Standish Group Report: There's Less Development Chaos Today, 2007. <http://www.sdtimes.com/article/story-20070301-01.html> (pristupano 26.03.2007.)

[Schmidt *et al.*, 2001.] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.,: Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects, John Wiley and Sons Ltd, UK, 2001.

[Sommerwille, 2007.] Sommerwille, I.: Software Engineering 8, Addison-Wesley, 2007.

[Sooksanan, Sooksanan,P.; Roongruangsuwan,S.: Validation Rules for Exporting

Roongruangsuwan, 2005.]	Business Process Diagram to Business Process Execution Language, Proceedings of the Fourth International Conference on eBusiness, November 19-20, 2005, Bangkok, Thailand, <http://www.ijcim.th.org/v13nSP3/pdf/p15-1-6-Validation%20Rules2.pdf>, (pristupano: 04.01.2007)
[Strahonja *et al.*, 1992.]	Strahonja, V., Varga, M., Pavić, M.: Projektiranje informacijskih sustava, Zavod za informatičku djelatnost Hrvatske i INA-INFO, 1992. Zagreb
[Strahonja, 2006.]	Strahonja, V.: Model-based validation and verification of anomalies in legislation, JIOS, Vol.30, No.2, 2006.
[Swithinbank *et al.*, 2005.]	Swithinbank, P., Chessell, M., Gardner, T., Griffin, C., Man, J., Wylie, H., Yusuf, L.: Patterns: Model-Driven Development Using IBM Rational Software Architect, IBM Redbooks, 2005.
[Teale *et al.*, 2003.]	Teale, P., Etz, C., Kiel, M., Zeitz, C.: Data Patterns, 2003., <http://msdn2.microsoft.com/en-us/library/ms998446.aspx>, (pristupano: 20.07.2007.)
[The Standish Group, 1995.]	The Standish Group Report: Chaos,1995., <http://www.projectsmart.co.uk/docs/chaos_report.pdf>, (pristupano 22.03.2007.)
[The Standish Group, 2004.]	The Standish Group Report: Project Success Rates Improved Over 10 Years, SoftwareMag, 2004., <http://www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish>, (pristupano 22.03.2007.)
[Trowbridge et.al., 2003.]	Trowbridge, D., Mancini, D., Quick,D., Hohpe, G., Newkirk,J., Lavigne, D.: Enterprise Solution Patterns Using Microsoft .NET, v2.0, 2003., <http://msdn2.microsoft.com/en-us/library/ms998469.aspx>, (pristupano: 20.07.2007.)
[Veljović, 2002.]	Veljović, A.: Osnove objektnog modeliranja – UML, Kompjuter biblioteka, Čačak, 2002.
[White, 2006.]	White, S. A.: Introduction to BPMN, 2006.
[Yu, 2007.]	Yu, C.: Model-driven and pattern-based development using Rational Software Architect, Part 2: Model-driven development tooling support in IBM Rational software Architect, 2007., <http://www-

128.ibm.com/developerworks/rational/library/07/0116_yu/>, (pristupano: 02.02.2007.)

[Yu, 2007.a] Yu, C.: Model-driven and pattern-based development using Rational Software Architect, Part 1: Overview of the model-driven development paradigm with patterns, 2007., <http://www-128.ibm.com/developerworks/rational/library/06/1121_yu/>, (pristupano: 02.02.2007.)

[Yusuf *et al.,* 2006.] Yusuf, L., Chessel, M., Gardner,T.: Implement Model-Driven Development to Increase the Business Value of Your IT System, <http://www-128.ibm.com/developerworks/library/ar-mdd1/>, (pristupano: 14.04.2006.)

[Yusuf, Gardner, 2006.] Yusuf, L., Gardner,T.: Explore model-driven development (MDD) and related approaches: A closer look at model-driven development and other industry initiatives, 2006., <http://www-128.ibm.com/developerworks/library/ar-mdd3/>, (pristupano: 14.04.2006.)

PRILOG A

Izvorna detaljna specifikacija Enterprise Patterns kataloga

Main

Short Description:

A collection of Enterprise patterns.

Version:

1.2

Last Modified Date:

2005-11-07

Profile Name:

Default Profile

A collection of Enterprise patterns. Business Delegate Reduce the coupling between application clients and remote business services. Data Access Object Isolates the persistence storage details from client logic. Message Façade Used to receive asynchronous client messages. Session Façade Provides remote access to a collection of entity beans. This asset requires Rational Software Architect 6.0.0.1

Classification

The classification section describes the asset.

Descriptor Group Name:Default

Author:

IBM Corporation

Benefit:

Keyword:

J2EE, Enterprise Patterns, Business Tier, Integration Tier, Business Delegate, Data Access Object, Message Facade, Session Facade

Known Uses:

Liability:

Problem Solved:

Descriptor Group Name:Support

Site:

http://www.ibm.com/contact/us

Product:

Rational SW Architect/SW Modeler

Component:

RA_Pattern Content

SubComponent:

Enterprise Patterns

Solution

The solution section describes the artifacts that comprise the asset and provide the solution to one or more problems.

General:

File Name	File Type	Description	Reference	Category
BusinessDelegatePattern.ras	RAS Bundled Asset		BusinessDelegatePattern.ras	Artifact
DataAccessObjectPattern.ras	RAS Bundled Asset		DataAccessObjectPattern.ras	Artifact
MessageFacadePattern.ras	RAS Bundled Asset		MessageFacadePattern.ras	Artifact
SessionFacadePattern.ras	RAS Bundled Asset		SessionFacadePattern.ras	Artifact
Buildable	Buildable			Artifact
com.ibm.xtools.patterns.content.j2ee.feature	Project		com.ibm.xtools.patterns.content.j2ee.feature	Artifact
Deployable Eclipse Artifact	Deployable Eclipse Artifact		Deployable Eclipse Artifact/	Artifact
features	Folder		Deployable Eclipse Artifact/features/	Artifact
com.ibm.xtools.patterns.content.j2ee.feature_6.0.1	Folder		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/	Artifact
copyright.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/copyright.html	Artifact
description.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/description.html	Artifact
feature.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature.properties	Artifact
feature.xml	XML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature.xml	Artifact
feature_cs.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_cs.properties	Artifact
feature_de.properties	Java Property File		Deployable Eclipse	Artifact

			Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_de.properties	
feature_es.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_es.properties	Artifact
feature_fr.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_fr.properties	Artifact
feature_it.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_it.properties	Artifact
feature_ja.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_ja.properties	Artifact
feature_ko.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_ko.properties	Artifact
feature_pl.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_pl.properties	Artifact
feature_pt_BR.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_pt_BR.properties	Artifact
feature_tr.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_tr.properties	Artifact
feature_zh.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_zh.properties	Artifact
feature_zh_HG.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/feature_zh_HG.properties	Artifact
feature_zh_TW.properties	Java Property File		Deployable Eclipse Artifact/features/com.ibm.xtools.patt	Artifact

			erns.content.j2ee.feature_6.0.1/feature_zh_TW.properties	
license.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license.html	Artifact
license_cs.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_cs.html	Artifact
license_de.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_de.html	Artifact
license_es.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_es.html	Artifact
license_fr.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_fr.html	Artifact
license_it.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_it.html	Artifact
license_ja.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_ja.html	Artifact
license_ko.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_ko.html	Artifact
license_pl.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_pl.html	Artifact
license_pt_BR.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_pt_BR.html	Artifact
license_tr.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/lice	Artifact

			nse_tr.html	
license_zh.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_zh.html	Artifact
license_zh_HG.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_zh_HG.html	Artifact
license_zh_TW.html	HTML		Deployable Eclipse Artifact/features/com.ibm.xtools.patterns.content.j2ee.feature_6.0.1/license_zh_TW.html	Artifact
plugins	Folder		Deployable Eclipse Artifact/plugins/	Artifact
com.ibm.xtools.patterns.content.j2ee_1.0.0.1	Folder		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content.j2ee_1.0.0.1/	Artifact
j2eepatterns.jar	Jar		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content.j2ee_1.0.0.1/j2eepatterns.jar	Artifact
j2eepatternssrc.zip	Zip		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content.j2ee_1.0.0.1/j2eepatternssrc.zip	Artifact
plugin.xml	XML		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content.j2ee_1.0.0.1/plugin.xml	Artifact
templates	Folder		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content.j2ee_1.0.0.1/templates/	Artifact
j2ee-jmerge-rules.xml	XML		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content.j2ee_1.0.0.1/templates/j2ee-jmerge-rules.xml	Artifact
merge.xml	XML		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content.j2ee_1.0.0.1/templates/merge.xml	Artifact
com.ibm.xtools.patterns.content_6.0.0	Folder		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content_6.0.0/	Artifact
patternsContent.jar	Jar		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patte	Artifact

			rns.content_6.0.0/patternsContent.jar	
patternsContentsrc.zip	Zip		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content_6.0.0/patternsContentsrc.zip	Artifact
plugin.properties	Java Property File		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content_6.0.0/plugin.properties	Artifact
plugin.xml	XML		Deployable Eclipse Artifact/plugins/com.ibm.xtools.patterns.content_6.0.0/plugin.xml	Artifact

Usage

The usage section describes the activities needed to consume the asset.

Artifact Activities:

Task	Task Type	Description
Do Not Extract	com.ibm.xtools.ras.import.artifact.IGNORE_FOR_IMPORT	
Do Not Extract	com.ibm.xtools.ras.import.artifact.IGNORE_FOR_IMPORT	
Do Not Extract	com.ibm.xtools.ras.import.artifact.IGNORE_FOR_IMPORT	
Do Not Extract	com.ibm.xtools.ras.import.artifact.IGNORE_FOR_IMPORT	
Build Deployable Project com.ibm.xtools.patterns.content.j2ee.feature	com.ibm.xtools.ras.BuildDeployableProjectWithSource	

Related Assets

The related assets section describes any related assets and their relationship with this asset. The following assets are related to this one:

- com.ibm.xtools.patterns.content.j2ee.businessdelegate
- com.ibm.xtools.patterns.content.j2ee.dataaccessobject
- com.ibm.xtools.patterns.content.j2ee.messagefacade
- com.ibm.xtools.patterns.content.j2ee.sessionfacade

PRILOG B

Programski kod generiran za entity bean Klijent *i* RacunSF_SBFImpl.

KlijentBean.java

```
package ejbs;

/**
 * @ws.sbf.session-facade
 *   name="KlijentSF_SBFImpl"
 *   value-objects="Klijent"
 * @ws.sdo.value-object
 *   name="Klijent"
 *   read-only="false"
 * Bean implementation class for Enterprise Bean: Klijent
 */
public abstract class KlijentBean implements javax.ejb.EntityBean {
    private javax.ejb.EntityContext myEntityCtx;

    /**
     * setEntityContext
     */
    public void setEntityContext(javax.ejb.EntityContext ctx) {
           myEntityCtx = ctx;
    }

    /**
     * getEntityContext
     */
    public javax.ejb.EntityContext getEntityContext() {
           return myEntityCtx;
    }

    /**
     * unsetEntityContext
     */
    public void unsetEntityContext() {
           myEntityCtx = null;
    }

    /**
     * ejbCreate
     */
    public ejbs.KlijentKey ejbCreate(java.lang.String jBMG)
                  throws javax.ejb.CreateException {
           setJBMG(jBMG);
           return null;
    }

    /**
     * ejbPostCreate
     */
    public void ejbPostCreate(java.lang.String jBMG)
                  throws javax.ejb.CreateException {
    }

    /**
     * ejbActivate
     */
    public void ejbActivate() {
    }

    /**
     * ejbLoad
     */
    public void ejbLoad() {
    }
```

```
        /**
         * ejbPassivate
         */
        public void ejbPassivate() {
        }

        /**
         * ejbRemove
         */
        public void ejbRemove() throws javax.ejb.RemoveException {
        }

        /**
         * ejbStore
         */
        public void ejbStore() {
        }

        /**
         * Get accessor for persistent attribute: jBMG
         */
        public abstract java.lang.String getJBMG();

        /**
         * Set accessor for persistent attribute: jBMG
         */
        public abstract void setJBMG(java.lang.String newJBMG);

        /**
         * Get accessor for persistent attribute: prezimeIme
         */
        public abstract java.lang.String getPrezimeIme();

        /**
         * Set accessor for persistent attribute: prezimeIme
         */
        public abstract void setPrezimeIme(java.lang.String newPrezimeIme);
    }
```

KlijentSF_SBFImplBean.java

```
    /**
     * This is a derived file. In order to preserve any customizations to this file, mark the
     * file as 'underived' in the file properties. Failure to do so, will discard any
modifications
     * if the project is cleaned.
     * Note that the '@generated' annotation attribute should not be present to preserve any
     * changes to a generated or added method/property.
     */

    package ejbs;

    import java.rmi.RemoteException;

    import javax.ejb.EJBException;
    import javax.ejb.SessionBean;

    import org.eclipse.emf.ecore.EClassifier;

    import com.ibm.websphere.ejbquery.QueryException;
    import com.ibm.websphere.sbf.MediatorMetaData;
    import com.ibm.websphere.sbf.MediatorMetaDataCache;
    import com.ibm.websphere.sbf.exceptions.ApplyRootChangesFailedException;
    import commonj.sdo.DataObject;
    import com.ibm.websphere.sbf.exceptions.CreateException;
    import com.ibm.websphere.sbf.exceptions.DeleteException;
    import com.ibm.websphere.sbf.exceptions.UpdateException;
    import com.ibm.websphere.sbf.exceptions.FindException;
    import com.ibm.websphere.sbf.exceptions.MediatorMetaDataNotFoundException;
    import com.ibm.websphere.sdo.mediator.ejb.Mediator;
    import com.ibm.websphere.sdo.mediator.ejb.MediatorFactory;

    import ejbs.sdo.SdoPackage;

    import ejbs.sdo.Klijent;
```

```
import ejbs.sdo.KlijentRoot;

/**
 * @ejb.bean
 *   name="KlijentSF_SBFImpl"
 *   view-type="both"
 *   type="Stateless"
 *  transaction-type="Container"
 *<!-- begin-user-doc -->
 *<!-- end-user-doc -->
 * This is a system generated session bean facade source file.
 * Any modifications to the tags should be done in the
 * primary source Entity Bean file for this facade.
 * @see KlijentBean
 * generated
 */
public class KlijentSF_SBFImplBean implements SessionBean {
    private javax.ejb.SessionContext mySessionCtx;

    /**
     * getSessionContext
     */
    public javax.ejb.SessionContext getSessionContext() {
           return mySessionCtx;
    }

    /**
     * setSessionContext
     */
    public void setSessionContext(javax.ejb.SessionContext ctx) {
           mySessionCtx = ctx;
    }

    /*
     * (non-Javadoc)
     *
     * @see javax.ejb.SessionBean#ejbCreate()
     */
    public void ejbCreate() throws javax.ejb.CreateException {
    }

    /*
     * (non-Javadoc)
     *
     * @see javax.ejb.SessionBean#ejbRemove()
     */
    public void ejbRemove() throws EJBException, RemoteException {
    }

    /*
     * (non-Javadoc)
     *
     * @see javax.ejb.SessionBean#ejbActivate()
     */
    public void ejbActivate() throws EJBException, RemoteException {
    }

    /*
     * (non-Javadoc)
     *
     * @see javax.ejb.SessionBean#ejbPassivate()
     */
    public void ejbPassivate() throws EJBException, RemoteException {
    }

    /**
     * @generated
     */
    private MediatorMetaDataCache mediatorCache = new MediatorMetaDataCache() {
           protected MediatorMetaData createMediatorMetaData(EClassifier type) {
                  switch (type.getClassifierID()) {

                  case SdoPackage.KLIJENT_ROOT:
                         return createKlijentRootMediatorMetaData();
                  }
                  return null;
           }
```

```
};

/**
 * @generated
 */
private void doApplyChanges(DataObject dataObject) throws QueryException,
              MediatorMetaDataNotFoundException {
       mediatorCache.createMediator(dataObject).applyChanges(dataObject);
}

/**
 * Delete a value-object.
 * @generated
 */
private void doDeleteDataObject(DataObject data)
              throws MediatorMetaDataNotFoundException, QueryException {
       DataObject root = data.getDataGraph().getRootObject();
       data.delete();
       doApplyChanges(root);
}

/**
 * @generated
 */
private MediatorMetaData createKlijentRootMediatorMetaData() {

       String[] queryStrings = new String[1];

       int pattern = MediatorFactory.ROOT_CONTAINS_ALL;

       queryStrings[0] = "select k.*  from Klijent as k";
       SdoPackage pack = SdoPackage.eINSTANCE;
       MediatorMetaData md = new MediatorMetaData(pack.getKlijentRoot(),
                     queryStrings, pattern);
       md.addAbstractSchemaNameMapping("Klijent", pack.getKlijent());
       return md;
}

/**
 * Return an array of "Klijent" value-objects from the database.
 * @ejb.interface-method
 *  view-type="both"
 * @generated
 */
public Klijent[] getAllKlijentObjects() throws FindException {
       try {
              KlijentRoot root = doGetKlijentRoot(null, null);
              return root.getKlijentAsArray();
       } catch (Exception ex) {
              throw new FindException(
                            "System error while finding all \"Klijent\".", ex);
       }
}

/**
 * @ejb.interface-method
 *  view-type="both"
 * Get list of "Klijent" value-objects.
 * @generated
 */
public Klijent getKlijentByKey(KlijentKey primaryKey) throws FindException {
       try {

              String[] queryStrings = new String[1];

              queryStrings[0] = "select k.*  from Klijent as k where k.jBMG = ?1";

              Object[] pkValues = getKlijentPKeyObjectArray(primaryKey);

              KlijentRoot root = doGetKlijentRoot(queryStrings, pkValues);
              Klijent[] array = root.getKlijentAsArray();
              return (array == null || array.length == 0) ? null : array[0];
       } catch (Exception ex) {
              throw new FindException(
                            "System error while finding \"Klijent\" by key.", ex);
       }
}
```

```
	/**
	 * Persist a new "Klijent" value-object.
	 * @ejb.interface-method
	 *  view-type="both"
	 * @generated
	 */
	public void createKlijent(Klijent data) throws CreateException {
		try {
			doApplyChanges(data);
		} catch (Exception ex) {
			throw new CreateException(
					"System error while creating \"Klijent\".", ex);
		}
	}

	/**
	 * Update a "Klijent" value-object.
	 * @ejb.interface-method
	 *  view-type="both"
	 * @generated
	 */
	public void updateKlijent(Klijent data) throws UpdateException {
		try {
			doApplyChanges(data);
		} catch (Exception ex) {
			throw new UpdateException(
					"System error while updating \"Klijent\".", ex);
		}
	}

	/**
	 * Delete a "Klijent" value-object.
	 * @ejb.interface-method
	 *  view-type="both"
	 * @generated
	 */
	public void deleteKlijent(Klijent data) throws DeleteException {
		try {
			doDeleteDataObject(data);
		} catch (Exception ex) {
			throw new DeleteException(
					"System error while deleting \"Klijent\".", ex);
		}
	}

	/**
	 * Apply any and all changes to every SDO change contained by the passed
	 * KlijentRoot.
	 * @ejb.interface-method
	 *  view-type="both"
	 * @generated
	 */
	public void applyKlijentRootChanges(KlijentRoot root)
			throws ApplyRootChangesFailedException {
		try {
			doApplyChanges(root);
		} catch (Exception e) {
			throw new ApplyRootChangesFailedException(
					"System    error    while    applying    changes    to
\"KlijentRoot\".",
					e);
		}
	}

	/**
	 * Return a KlijentRoot which is the root object of the query.
	 * The KlijentRoot object will contain references to all SDOs
	 * returned in the query.
	 * @ejb.interface-method
	 *  view-type="both"
	 * @generated
	 */
	public KlijentRoot getKlijentRoot() throws FindException {
		try {
			return doGetKlijentRoot(null, null);
		} catch (Exception ex) {
```

```
                        throw new FindException(
                                        "System   error   while   executing   query   to   populate
\"KlijentRoot\".",
                                        ex);
                }
        }

        /**
         * @generated
         */
        private KlijentRoot doGetKlijentRoot(
                        String[] queryStringOverridesWithWhereClause,
                        Object[] whereClauseValues) throws QueryException,
                        MediatorMetaDataNotFoundException {
                Mediator mediator = mediatorCache.createMediator(SdoPackage.eINSTANCE
                                .getKlijentRoot(), queryStringOverridesWithWhereClause,
                                whereClauseValues);
                return (KlijentRoot) mediator.getGraph();
        }

        /**
         * @generated
         */
        private Object[] getKlijentPKeyObjectArray(KlijentKey primaryKey) {
                return new Object[] { primaryKey.jBMG };
        }
   }
```

PRILOG C

Programski kod za datoteke PatternLibrary.java *i* NoviUzorak.java

PatternLibrary.java

```
package RazvojNovogUzorka.lib;

import com.ibm.xtools.patterns.framework.AbstractPatternDefinition;
import com.ibm.xtools.patterns.framework.AbstractPatternLibrary;

/**
* This class represents the pattern library and extends from a base class that provides
some of the required library functionality.  There is a conceptual one-to-one relationship
between a pattern
 * plug-in and a pattern library.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * generated
 */
public final class PatternLibrary extends AbstractPatternLibrary {
     /**
      * <!-- begin-user-doc -->
      * <!-- end-user-doc -->
      * @generated
      */
     public PatternLibrary() {
            super(RazvojNovogUzorka.RazvojNovogUzorkaPlugin.getDefault());// DO NOT EDIT
     }

     /**
      * DO NOT EDIT EXCEPT FOR THE USER DOC SECTION
      * NEEDS TO BE KEPT IN SYNC WITH THE LIBRARY MANIFEST FILE
      * <!-- begin-user-doc -->
      * <!-- end-user-doc -->
      * @generated
      */
     protected AbstractPatternDefinition[] getAvailablePatternDefinitions() {
            return new AbstractPatternDefinition[] {

            new RazvojNovogUzorka.patterns.noviuzorak.NoviUzorak(this) };
     }

}
```

NoviUzorak.java

```
package RazvojNovogUzorka.patterns.noviuzorak;

import com.ibm.xtools.patterns.framework.AbstractPatternLibrary;
import com.ibm.xtools.patterns.framework.PatternIdentity;
import com.ibm.xtools.patterns.framework.uml2.AbstractPatternDefinition;
import com.ibm.xtools.patterns.framework.uml2.AbstractPatternInstance;

import com.ibm.xtools.patterns.framework.PatternParameterIdentity;
import com.ibm.xtools.patterns.framework.uml2.AbstractPatternParameter;
import com.ibm.xtools.patterns.framework.AbstractPatternDependency;
import com.ibm.xtools.patterns.framework.PatternParameterValue;
import org.eclipse.uml2.*;

/**
* This class implements the NoviUzorak, it extends
* from the abstract pattern definition base class to gain most of the functionality needed
to make pattern definition specification in code easier.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
```

```
   public class NoviUzorak extends AbstractPatternDefinition {
        /**
         * DO NOT EDIT EXCEPT FOR THE USER DOC SECTION
         * NEEDS TO BE KEPT IN SYNC WITH THE PATTERN MANIFEST FILE
         * <!-- begin-user-doc -->
         * <!-- end-user-doc -->
         * @generated
         */
        private final static String PATTERN_ID =
"RazvojNovogUzorka.patterns.noviuzorak.NoviUzorak"; //$NON-NLS-1$

        /**
         * DO NOT EDIT EXCEPT FOR THE USER DOC SECTION
         * NEEDS TO BE KEPT IN SYNC WITH THE PATTERN MANIFEST FILE
         * <!-- begin-user-doc -->
         * <!-- end-user-doc -->
         * @generated
         */
        private final static String PATTERN_VER = "1.0.0"; //$NON-NLS-1$

        /**
         * DO NOT EDIT EXCEPT FOR THE USER DOC SECTION
         * NEEDS TO BE KEPT IN SYNC WITH THE PATTERN MANIFEST FILE
         * <!-- begin-user-doc -->
         * <!-- end-user-doc -->
         * @generated
         */
        private final Klasa klasa = new Klasa();

        /**
         * DO NOT EDIT EXCEPT FOR THE USER DOC SECTION
         * NEEDS TO BE KEPT IN SYNC WITH THE PATTERN MANIFEST FILE
         * <!-- begin-user-doc -->
         * <!-- end-user-doc -->
         * @generated
         */
        private final Atributi atributi = new Atributi(this.klasa);

        /**
         * <!-- begin-user-doc -->
         * <!-- end-user-doc -->
         * @generated
         */
        public NoviUzorak(AbstractPatternLibrary owningPatternLibrary) {
                super(
                                new PatternIdentity(owningPatternLibrary, PATTERN_ID,
                                                PATTERN_VER)); // DO NOT EDIT
        }

        /**
        * @author Ruben
        OVO JE DEFINICIJA PRVOG PARAMETRA klasa I SADRZI 2 expand METODE
        JEDNA ZA DODAVANJE VRIJEDNOSTI PARAMETRA A DRUGA ZA UKLANJANJE
         * */
        /**
         *
         * TODO To change the template for this generated type comment go to
         * Window - Preferences - Java - Code Style - Code Templates
         */
        /**
        * This class implements the Klasa, it extends from the abstract pattern parameter base
        class to gain most of the functionality needed to make
         * pattern parameter specification in code easier.
         * <!-- begin-user-doc -->
         * <!-- end-user-doc -->
         * @generated
         */
        private class Klasa extends AbstractPatternParameter {
                /**
                 * DO NOT EDIT EXCEPT FOR THE USER DOC SECTION
                 * NEEDS TO BE KEPT IN SYNC WITH THE PATTERN MANIFEST FILE
                 * <!-- begin-user-doc -->
                 * <!-- end-user-doc -->
                 * @generated
                 */
                private final static String PARAMETER_ID = "Klasa"; //$NON-NLS-1$
```

```
		/**
		 * <!-- begin-user-doc -->
		 * <!-- end-user-doc -->
		 * @generated
		 */
		Klasa() {
			super(NoviUzorak.this, new PatternParameterIdentity(PARAMETER_ID));
		// DO NOT EDIT

		}

		/**
		 * This is the default general expand method that is typically implemented
		 * by concrete pattern parameters for handling the added and maintained
		 * expand variants which are usually similar.
		 *
		 * This method gets called during a pattern expansion for each new parameter
		 * value that has been bound since last expansion, or for each exisitng value
		 * in a reapply scenerio.
		 *
		 * The default behavior is to return true.
		 * <!-- begin-user-doc -->
		 * <!-- end-user-doc -->
		 * @generated
		 */
		public boolean expand(PatternParameterValue value) {
			//TODO : implement the parameter's expand method --DODAVANJE
			return true;
		}

		/**
		 * The default behavior is to return true since removed value expansion is
		 * optional but it is recommended that pattern parameters support this for
		 * consistency with the patterns shipped in the product and a better user
		 * experience for the patter applying user.
		 *
		 * This method gets called during a pattern expansion for each parameter
		 * value that has been unbound since last expansion.
		 *
		 * <!-- begin-user-doc -->
		 * <!-- end-user-doc -->
		 * @generated
		 */
		public boolean expand(PatternParameterValue.Removed value) {
			//TODO : implement the parameter's expand method  --UKLANJANJE
			return true;
		}

	}

	/**
	 * This class implements the Atributi, it extends
	 * from the abstract pattern parameter base class to gain most of the functionality
needed to make
	 * pattern parameter specification in code easier.
	 * <!-- begin-user-doc -->
	 * <!-- end-user-doc -->
	 * @generated
	 */
	private class Atributi extends AbstractPatternParameter {
		/* (non-Javadoc)
		 * @see
com.ibm.xtools.patterns.framework.AbstractPatternParameter#isValid(com.ibm.xtools.patterns.fra
mework.PatternParameterValue.Proposed)
		 */
		protected boolean isValid(Proposed arg0) {
			boolean isValid=super.isValid(arg0);
			if (isValid){
				Property proposedProperty=(Property)arg0.getValue();
				if		(proposedProperty.getType()==null		||
proposedProperty.getType().getName().equals("void")){
					isValid=false;
				}
			}

			return isValid;
		}
```

```
/**
 * DO NOT EDIT EXCEPT FOR THE USER DOC SECTION
 * NEEDS TO BE KEPT IN SYNC WITH THE PATTERN MANIFEST FILE
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
private final static String PARAMETER_ID = "Atributi"; //$NON-NLS-1$

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
Atributi(Klasa klasa) {
	super(NoviUzorak.this, new PatternParameterIdentity(PARAMETER_ID));
// DO NOT EDIT

	new Atributi_KlasaDependency(klasa);

}

/**
 * This is the default general expand method that is typically implemented
 * by concrete pattern parameters for handling the added and maintained
 * expand variants which are usually similar.
 *
 * This method gets called during a pattern expansion for each new parameter
 * value that has been bound since last expansion, or for each exisitng value
 * in a reapply scenerio.
 *
 * The default behavior is to return true.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public boolean expand(PatternParameterValue value) {
	//TODO : implement the parameter's expand method
	return true;
}

/**
 * The default behavior is to return true since removed value expansion is
 * optional but it is recommended that pattern parameters support this for
 * consistency with the patterns shipped in the product and a better user
 * experience for the patter applying user.
 *
 * This method gets called during a pattern expansion for each parameter
 * value that has been unbound since last expansion.
 *
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public boolean expand(PatternParameterValue.Removed value) {
	//TODO : implement the parameter's expand method
	return true;
}

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */

/**
 * NASTAVKU SLIJEDI KLASA KOJOM SE OPISUJE POVEZANOST PARAM2 SA PARAM1
 *KLASA JE UGNJEŽĐENA U PARAM 2 KLASI
 **/

private class Atributi_KlasaDependency extends
		AbstractPatternDependency {
	/**
	 * <!-- begin-user-doc -->
	 * <!-- end-user-doc -->
	 * @generated
	 */
```

```
                        private Atributi_KlasaDependency(AbstractPatternParameter dependency) {
                                super(Atributi.this, dependency);
                        }

                        /**
                         * This method gets called during a pattern expansion for each pair
                         * of new values that has been added to the dependent and dependency
                        * parameter since last expansion as a result of binding a value to
                        either the dependent or dependency parameter. In a reapply scenerio, it
                        gets
                         * called for each pair of existing values bound to the parameters.
                         *
                         * Update the dependency using the dependent and dependency values as
                         * provided. This is a hot-spot method and the default behavior if not
                         * overridden is to do nothing more than return true.
                         * <!-- begin-user-doc -->
                         * <!-- end-user-doc -->
                         * @generated
                         */
                        public boolean update(PatternParameterValue value,
                                        PatternParameterValue dependencyValue) {

                                //dodjeljivanje vrijednosti parametara
                                Property property=(Property)value.getValue();
                                Class parameter1 =(Class)dependencyValue.getValue();

                                //kreiranje Getter operacije
                                Operation
getterOperation=parameter1.createOwnedOperation(UML2Package.eINSTANCE.getOperation());
                                getterOperation.setName ("get"+property.getName());
                                ParameterDirectionKind parameterDirection=
ParameterDirectionKind.get(ParameterDirectionKind.RETURN);
                                Parameter
returnParameter=getterOperation.createReturnResult(UML2Package.eINSTANCE.getParameter());
                                returnParameter.setDirection(parameterDirection);
                                returnParameter.setType(property.getType());

                                //kreiranje Setter operacije
                                Operation
setterOperation=parameter1.createOwnedOperation(UML2Package.eINSTANCE.getOperation());
                                setterOperation.setName ("set"+property.getName());
                                Parameter
inParameter=setterOperation.createOwnedParameter(UML2Package.eINSTANCE.getParameter());
                                inParameter.setName(property.getName());
                                inParameter.setType(property.getType());

                                return true;
                        }

                        /**
                         * This method gets called during a pattern expansion for each pair
                         * of values that has been removed from the dependent and dependency
                         * parameter since last expansion as a result of unbinding a value from
                         * the dependency parameter.
                         *
                         * Update the dependency using the dependent and dependency values as
                         * provided. This is a hot-spot method and the default behavior if not
                         * overridden is to do nothing more than return true. To support remove
                         * behavior in pattern implementations this method would be overridden.
                        * In most cases the implementation of this method would be very much
                        like the method variant with the types transposed.
                         * <!-- begin-user-doc -->
                         * <!-- end-user-doc -->
                         * @generated
                         */
                        public boolean update(PatternParameterValue.Maintained value,
                                        PatternParameterValue.Removed dependencyValue) {
                                //TODO: implement the dependency's update method
                                return true;
                        }

                        /**
                         * This method gets called during a pattern expansion for each pair
                         * of values that has been removed from the dependent and dependency
                         * parameter since last expansion as a result of unbinding a value from
                         * the dependent parameter.
                         *
```

```
                         * Update the dependency using the dependent and dependency values as
                         * provided. This is a hot-spot method and the default behavior if not
                         * overridden is to do nothing more than return true. To support remove
                        * behavior in pattern implementations this method woul be overridden. In
                        * most cases the implementation of this method would be very much like
                         * the method variant with the types transposed.
                         * <!-- begin-user-doc -->
                         * <!-- end-user-doc -->
                         * @generated
                         */
                        public boolean update(PatternParameterValue.Removed value,
                                        PatternParameterValue.Maintained dependencyValue) {
                                //dodjeljivanje vrijednosti parametara
                                Property property=(Property)value.getValue();
                                Class Klasa =(Class)dependencyValue.getValue();


                                // uklanjanje Setter operacije
                                Operation setterOperation=
   Klasa.getOwnedOpetarion("set"+property.getName());
                                if (setterOperation != null){
                                        EList parameters =setterOperation.getParameters();
                                        if(parameters.size()== 1 &&
((Parameter)parameters.get(0).getType().equals(property.getType())))

       Klasa.getOwnedOperations().remove(setterOperation);

                                }

                                // uklanjanje Getter operacije
                                Operation getterOperation=
Klasa.getOwnedOpetarion("get"+property.getName());
                                if (setterOperation != null){
                                        EList returnResults =getterOperation.getReturnResults();
                                        if(returnResults.size()== 1 &&
((Parameter)returnResults.get(0).getType().equals(property.getType())))

       Klasa.getOwnedOperations().remove(getterOperation);
                                }
                                return true;
                        }
                }

       }

   }
```