UDC: 004.438:004.42.045 INFO M: str. 4-12

# MODELOVANJE SISTEMA VELIKIH RAZMERA PRIMENJENO U PRAKSI: RAZVOJ SLOŽENIH POSLOVNIH APLIKACIJA POMOĆU IZVRŠIVOG UML-A

# PUTTING LARGE-SCALE MODEL-DRIVEN DEVELOPMENT INTO PRACTICE: BUILDING COMPLEX BUSINESS APPLICATIONS WITH AN EXECUTABLE UML

dr Dragan Milićev, Nemanja Kojić

REZIME: Uprkos intezivnim aktivnostima istraživača i industrije u poslednjih desetak godina na primeni i razvoju discipline razvoja softvera zasnovanog na modelima (engl. *Model-Driven Development*, MDD), ova disciplina očigledno nije postala široko zastupljena u praksi razvoja velikih informacionih sistema. U ovom radu pokušavamo da identifikujemo i objasnimo glavne razloge za to. Dajemo i kratak prikaz našeg pristupa razvoju složenih poslovnih aplikacija zasnovanog na jednom izvršivom profilu jezika UML pod nazivom OOIS UML, koji je podržan razvojnim okruženjem otvorenog koda pod nazivom SOLoist. Potom iznosimo neka naša iskustva u korišćenju datog pristupa i okruženja u industrijskim projektima srednjih i velikih razmera razvijanih u poslednjih desetak godina.

KLJUČNE REČI: Objedinjeni jezik za modelovanje (engl. *Unified Modeling Language, UML*), razvoj softvera zasnovan na modelima (engl: *Model Driven Development, MDD*), brz razvoj aplikacija (engl: *Rapid Application Development*), objektni upitni jezik (engl: *Object Query Langugage, OQL*), informacioni sistemi.

ABSTRACT: Despite intensive work in academy and industry around it in the last decade, the discipline of model-driven development apparently has not become the industrial mainstream for building large-scale information systems. In this paper, we try to identify and explain the main reasons for the lackluster adoption of MDD in this field. Then we present our approach to building large-scale business applications based on an executable profile of UML, named OOIS UML, and implemented as an open-source framework named SOLoist. We also briefly report on our experiences in using the approach and the framework in medium to large industrial projects over the last decade.

KEY WORDS: Unified Modeling Language (UML), model-driven development (MDD), rapid application development, object query language (OQL), information systems.

## UVOD

Razvoj softvera zasnovan na modelima (engl. *Model-Driven Development*, MDD), kao opšta disciplina softverskog inženjerstva, zajedno sa pratećim tehnologijama i standardima ustanovljenim od strane organizacije OMG (engl. *Object Management Group*), kao što su MDA (engl. *Model-Driven Architecture*) i druga generacija jezika UML, prisutni su već više od desetak godina. Međutim, MDD do sada nije postao preovlađujuća metodologija u industriji razvoja složenih informacionih sistema. Uprkos činjenici da se i u industriji i u akademskim istraživačkim sredinama mnogo uradilo na razvoju ove paradigme, kao i tome što postoje mnogi izveštaji o iskustvima u upotrebi MDD u praksi, utisak je da ovi izveštaji mahom potiču iz akademskih i drugih istraživačkih krugova, kao i da su prvenstveno vezani za pionirske poduhvate i preliminarne provere i ispitivanja, a ne za široke industrijske primene. Doduše, u industriji postoji značajno interesovanje za unapređenja u ovom domenu. Postoje i mnoge inicijative vezane za razvoj infrastrukture, standarda, proizvoda i alata za podršku MDD. Međutim, očigledno je da velika većina programera u praksi i dalje uopšte ne koristi MDD za razvoj informacionih sistema, ili ga koristi veoma retko i u veoma ograničenom obliku.

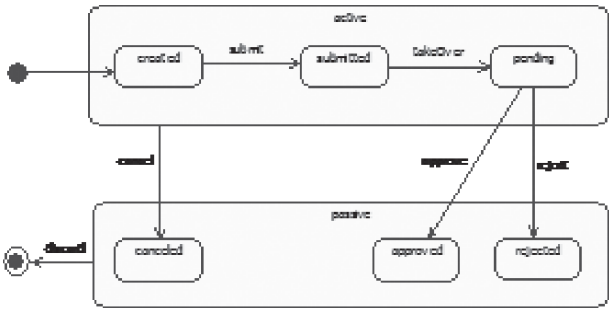
U odeljku 2 ovog rada najpre ukratko ukazujemo na glavne motive za korišćenje MDD, kao i na najvažnije prednosti koje takav pristup donosi. Pokušavamo da identifikujemo i objasnimo i glavne razloge za nedovoljan stepen prihvaćenosti ove discipline. U odeljku 3 ukratko predstavljamo naš pristup za efikasan razvoj informacionih sistema velikih razmera zasnovan na modelima. Ovaj pristup je zasnovan na jednom izvršivom profilu jezika UML pod nazivom OOIS UML, a koji je podržan razvojnim okruženjem otvorenog koda pod nazivom SOLoist. Pošto je profil OOIS UML detaljno opisan na drugom mestu [8], ovde samo u kratkim crtama navodimo neke njegove glavne elemente i ilustrujemo kako se pomoću njega rešavaju problemi identifikovani u odeljku 2. Najzad, u odeljku 4 dajemo kratak izveštaj o našem iskustvu u korišćenju datog pristupa i okruženja u srednjim i velikim industrijskim projektima tokom poslednjih desetak godina. U odeljku 5 ukratko diskutujemo o drugim sličnim rešenjima. Treba naglasiti da je naš fokus na razvoju informacionih sistema, posebno onih koji zahtevaju složen korisnički interfejs baziran na vebu (npr. poslovne aplikacije za različite domene, a koje rukuju bazama podataka).

## MDD: PREDNOSTI I ZAMKE

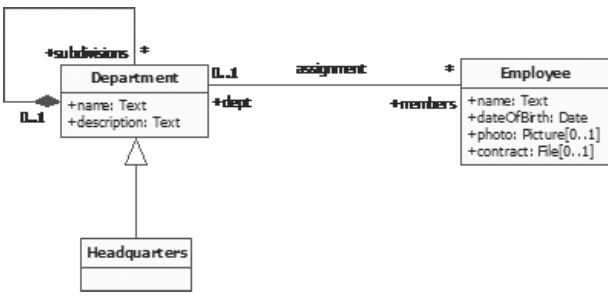
Još od vremena kada su izmišljeni programski prevodioci, unapređenja u načinu na koji se softver specifikuje svode se na stalno podizanje nivoa apstrakcije. Osnovna tendencija jeste da se za opis ideja, koncepata i projektnih odluka koriste apstraktni jezici što bliži domenu problema, a apstraktne specifikacije programa prevodilac može da prevede u predstavu na nižem nivou apstrakcije (u krajnjem slučaju u oblik koji računar može da razume). Očigledna posledica upotrebe apstraktnijih jezika jeste poboljšana izražajnost specifikacija softvera: softverski inženjer je u poziciji da sa manje "reči" kaže više (specificiraju-

UML


Slika 1a




Slika 1b




Slika 1c

Java

```
public class Department {
  public List<Employee> members =
                                  new ArrayList<Employee>;
  ... // Elided
}
public class Employee {
  public Department dept;
  ... // Elided
}
```

```
switch (event) {
  case cancel:
    switch (state) {
      case created:
      case submitted:
      case pending: state=canceled; break;
    }
... // Elided
}
```

```
public class Employee {
  public String name;
  public Date dateOfBirth;
  public Picture photo;
  public File contract;
  Department dept;
}

public class Department {
... // Elided
```

**Slika 1:** – *Jednostavni primeri koji ilustruju razliku u nivoima apstrakcije u jeziku UML (levo) i nekom tradicionalnom OO programskom jeziku (desno).*

ći ono što mašina treba da uradi). Ovo je zapravo najevidentnija dobit od korišćenja modela umesto tradicionalnih programskih jezika: MDD stimuliše upotrebu visoko apstraktnih modela arhitekture/dizajna umesto neposrednog kodiranja.

Slika 1 prikazuje nekoliko jednostavnih primera koji ilustruju razliku u nivoima apstrakcije UML modela i njihovih pandana napisanih na nekom tradicionalnom objektno orijentisanom (OO) programskom jeziku, kao što je Java.

Prvi primer na slici 1 prikazuje kako se asocijacija, kao jednostavan koncept jezika UML i inherentno bidirekciona strukturna relacija, preslikava na dve unidirekcione relacije između klasa napisanih na jeziku Java. Problem sa kôdom pisanim na Javi jeste taj što nije sasvim jasno i eksplicitno naznačeno da su članovi klasa `Department` i `Employee` konceptualno povezani (tj. logički spregnuti kao dva kraja iste asocijacije). Da bi se uspostavila veza po ovoj asocijaciji između nekog objekta klase `Department` i nekog objekta klase `Employee` korišćenjem specijalizovanog jezika akcija sa UML semantikom, dovoljno je napisati samo jednu od navedenih naredbi:

```
aDepartment.members.add(anEmployee);
```

ili

```
anEmployee.dept = aDepartment;
```

ali ne obe, jer i jedna i druga predstavljaju istu akciju uspostavljanja nove strukturne veze između dva objekta. Sa druge strane, programer na Javi mora da održava dve unidirekcione reference sinhronizovanim, tako da je potrebno napisati obe linije koda. Očigledno je da je ovaj drugi način znatno podložniji greškama. To je posledica primitivnije semantike unidirekcionih referenci u OO programskim jezicima nego bidirekcionih (strukturnih) veza u UML-u.

Drugi primer na slici 1 prikazuje jednostavnu mašinu stanja kojom je modelovan životni ciklus određenog koncepta u nekom informacionom sistemu, kao što je nekakav zahtev. Čak i za ovako jednostavnu mašinu stanja, funkcionalno ekvivalentni Java kôd postaje komplikovan i podložan greškama, kao što je prikazano na desnoj strani. Ovakav kôd je teško napisati, razumeti i održavati. Razlike još više dolaze do izražaja kada se koriste još apstraktniji koncepti mašina stanja (poput hijerarhijskih stanja ili istorije) ili kada sama mašina postane složenija.

Što je model složeniji, to je opisana razlika još veća, kao što je ilustrovano trećim primerom na slici 1. Izvorni kôd na Javi koji odgovara ovom kompaktnom i izražajnom modelu još je teži za analizu i razumevanje, jer relacije između klasa nisu eksplicitne. Osim toga, željeno značenje modela mora biti ostvareno izvršivim kôdom, a programer je taj koji mora da obezbedi taj kôd. Stoga je proizvedeni softver podložniji

greškama nego kada se koristi deklarativni pristup u kojem semantika jezika (UML u ovom slučaju) obezbeđuje željeno ponašanje. Može se jednostavno zaključiti da je skalabilnost modela mnogo bolja od skalabilnosti programskog kôda.

MDD donosi i druge sekundarne dobiti. Modeli se generalno smatraju nezavisnim od platforme ili bar mnogo manje zavisnim od implementacije nego što je slučaj sa programskim kôdom. Modeli ne moraju biti potpuno i detaljno formirani, ali i pored toga mogu da se iskoriste za dobijanje prototipova sistema u ranim fazama razvoja. To nije slučaj sa kôdom jer su prevodioci tradicionalnih programskih jezika mnogo rigorozniji i ne mogu da prevedu program koji nije u potpunosti i nedvosmisleno napisan.

Međutim, MDD ipak nije široko usvojen kao tehnološki standard za razvoj složenih informacionih sistema. Šta više, iskustvo autora ukazuje da je početni optimizam, koji je vladao na samom početku kada je UML nastao u kasnim devedesetim godinama, sada prilično raspršen, jer su mnogi razočarani njegovim korišćenjem u tim ranim fazama nastajanja. Pokušaćemo ovde da ukratko objasnimo razloge za koje mislimo da su najrelevantniji za ovaj neuspeh. Ovde se ne bavimo psihološkim, poslovnim ili sociološkim razlozima, već se isključivo držimo onih tehničkih.

U svojim prvim verzijama (1.x), UML nije imao ni približno nedvosmislenu, formalnu i izvršivu semantiku, a razlog za to leži u činjenici da je UML u svojoj izvornoj koncepciji bio prvenstveno osmišljen kao opisni jezik za specifikaciju, vizuelizaciju i dokumentovanje projektnih odluka u razvoju programa na tradicionalnim OO programskim jezicima. Osim toga, UML je zamišljen da bude veoma široko primenjiv za specificiranje programa pisanih na raznim programskim jezicima i za veoma raznolike domene aplikacija. Tako je u specifikaciji UML-a samo grubo data naznaka semantike njegovih koncepata, uglavnom izvedena uopštavanjem značenja koncepata zajedničkih za postojeće OO programske jezike. Precizna interpretacija značenja UML modela bila je gotovo potpuno prepuštena načinu na koji se ti modeli preslikavaju na implementaciju u ciljnom programskom jeziku. Takva interpretacija je očigledno veoma zavisna od semantike ciljnog programskog jezika, kao i od samog načina preslikavanja. UML modeli su zato bili semantički višeznačni. Nedvosmisleno tumačenje UML modela u nekom domenu isključivo je zavisilo od kreativnosti i discipline razvojnog tima.

Kada je reč o razvoju informacionih sistema, kao posledica navedenih činjenica, upotreba UML-a u praksi najčešće podrazumeva sledeće. Mnogi UML alati mogu lako da generišu šemu relacione baze podataka (DDL) i definicije klasa na ciljnom OO programskom jeziku iz UML modela klasa. Međutim, sprega između prostora objekata odgovarajućeg OO programskog jezika i podataka u relacionoj bazi podataka (RDBMS) se tipično ostavlja u odgovornost zasebnog okruženja za objektno-relaciono mapiranje (ORM), kao što je npr. Hibernate ili neki sličan. Takva okruženja za ORM ni na koji način nisu prilagođena UML-u i njegovoj semantici, a posebno semantici UML akcija, već samo nude spregu između OO programskog jezika i relacionih baza podataka. Takva sprega često podrazumeva da je šema baze podataka formirana odvojeno i nezavisno od strukture klasa OO programskog jezika. To znači da se šema baze podataka formira na osnovu konceptualnog modela (podataka) datog domena problema i optimizuje se za potrebne načine pristupa u skladu sa tradicionalnom praksom normalizacije i optimizacije relacione baze, dok se kôd na OO programskom jeziku projektuje u skladu sa praksom OO projektovanja. ORM ima zadatak da poveže ta dva produkta razvoja pravljena u dve različite paradigme. Kao posledica toga, okruženje za ORM programeru nameće činjenicu da objekat (u OO jeziku), koji postoji u (privremenoj) operativnoj memoriji, i njegova (perzistentna) reprezentacija u bazi podataka predstavljaju dva odvojena entiteta koje treba povezati (tipično preko identifikatora objekta) i brinuti o njihovoj sinhronizaciji pomoću odgovarajućih mehanizama datog okruženja. Ovi mehanizmi opterećuju programera obavezom da poznaje životni ciklus objekta u memoriji i da vodi računa o pozivanju odgovarajućih funkcionalnosti izvršnog okruženja koje omogućuju čuvanje stanja objekta u memoriji u bazu podataka (*save*), učitavanje stanja objekta u memoriji iz baze podataka (*load*), ili uklanjanje objekta iz memorije (*discard*). Ovo je samo jedan drastičan primer nepotrebne slučajne kompleksnosti nastale primenom nesavršene tehnologije za povezivanje dva semantički različita prostora (OO progamskog jezika i relacionih baza podataka).

Situacija je veoma slična kada se radi o sprezi semantičkog prostora OO programskog jezika sa prezentacionim slojem aplikacije. Mnoga okruženja i biblioteke za pravljenje korisničkog interfejsa (engl. *User Interface*, UI) sprežu semantički prostor prezentacije (na primer, HTML za veb bazirane interfejse) sa semantičkim prostorom OO programskog jezika, što prosto znači da se komponente korisničkog interfejsa prave tako da rade sa objektima u memoriji koji imaju semantiku definisanu u OO programskom jeziku. Gotovo nijedan popularni pristup ne obezbeđuje nikakvu vezu ili ne nudi nikakvu semantičku spregu sa UML-om i modelom aplikacije. Programer stoga vrlo često mora da bude svestan samih tehnoloških detalja, kao što je podela između veb stranice i pozadinskih objekata i/ili kontrolera, klijentskog i serverskog koda, sloja poslovne logike i sloja objekata entiteta itd.

Sa svom ovom jezičkom i semantičkom heterogenošću koje vodeće tehnologije nameću, OO izvorni kôd je tipično centralni produkt razvoja. Njegovoj semantici se prilagođavaju svi ostali produkti razvoja, što važi i za UML modele. U takvim uslovima model postaje samo još jedna briga više: nema mnogo smisla niti koristi crtati UML dijagrame samo da bi se dobili skeleti klasa na OO ciljnom programskom jeziku. Tako se UML zapravo koristi samo za skiciranje izvornog koda. Semantička heterogenost je sama po sebi veliki problem čak i bez korišćenja UML modela [2] i tipičan je primer neželjene slučajne kompleksnosti.

Još jedna posledica ovakvog pristupa jeste to što su programeri često u situaciji da nastoje da što pre krenu sa pisanjem koda, što se u literaturi pominje kao sindrom "žurbe ka kodovanju" (engl. *rush-to-code syndrom*). To je sindrom "sveopšte nelagodnosti u toku ranih faza razvoja, preovlađujući stav među programerima da su specifikacija zahteva i modeli 'samo dokumentacija', i ocećaj da 'pravi posao' nije započet sve dok se ne napiše programski kod" [18]. Iako se svi slažu da pravljenje specifikacije i modela daje bolji pregled sistema koji se razvija, što se više vremena investira u pravljenje takvih modela, to je osećaj

neizvesnosti rezultata veći. Uzrok za ovakav način razmišljanja leži u nedostatku objektivnih dokaza i načina da se proveri da je dobijeni model zaista ispravan i kompletan.

Kao posledica toga, kada se u razvoju jednom krene u fazu implementacije, specifikacije zahteva i modeli ostaju samo dokumentacioni proizvodi rada, bez izvršive semantike ili bilo kakvog uticaja na konačni izvršivi sistem. Budući da je proces razvoja inkrementalan i iterativan, potrebno je često vršiti razne modifikacije. Modifikacije početnog modela sistema nisu ni neophodne ni dovoljne da bi se sistem ažurirao, i zbog toga programeri nisu ni primorani da ga ažuriraju kako bi trebalo. Umesto toga, oni ažuriraju ono što je jedino i minimalno potrebno, a to je implementacija, tj. šema baze podataka i OO izvorni kôd koji direktno utiču na funkcionisanje sistema. Ovo u krajnjem slučaju vodi ka nekonzistentnosti između modela i implementacije, pa model sistema tako postaje netačna i samim tim štetna, a u najmanju ruku beskorisna dokumentacija. Ovo često motiviše razvojne timove da odbace modele i da se u kasnijim iteracijama isključivo oslanjaju na implementaciju.

Suština datog problema leži u *semantičkom diskontinuitetu* [18, 8], tj. u nedostatku formalne sprege između reprezentacija različitih vrsta povezanih detalja, o čemu je već bilo reči. Sa druge strane, problem nedostatka formalne semantike jezika UML je odavno prepoznat i napadnut u drugoj generaciju tog jezika [12], a posebno u okviru inicijative za formiranje tzv. bazičnog podskupa za izvršive UML modele (engl. *Foundational Subset for Executable UML Models*, fUML) [13]. Cilj ove inicijative jeste da se identifikuje podskup metamodela UML 2 koji daje osnovu za koncepte UML-a na višem nivou apstrakcije, kao i da se definiše precizna izvršiva semantika datog podskupa. Ovo bi trebalo da reši opisani suštinski problem nedostatka izvršive semantike osnovnih koncepata UML-a.

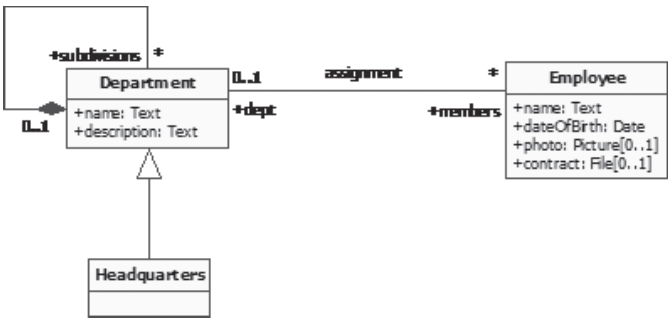
Nažalost, u praksi se još uvek ništa bitnije nije promenilo. Čini se da je pomenuta fUML inicijativa još uvek nova i bez adekvatne podrške u alatima i okruženjima. Šta više, još uvek nije ni ni poznata široj javnosti. Osim toga, fUML nudi opštenamensku platformu koju bi trebalo pažljivo profilisati za različite domene problema. Na primer, neki UML koncepti nisu toliko bitni niti neophodni za razvoj informacionih sistema, dok razvoj korisničkog interfejsa, kao jedan od najbitnijih zadataka koji obično uzima najviše resursa tokom razvoja, nije ni na koji način dotaknut u UML-u. Kao posledica toga, programeri i dalje koriste UML kao i ranije, ili ga uopšte i ne koriste. MDD kao disciplina, uz svu podršku u pratećim pomoćnim alatima, još uvek se ne čini dovoljno zrelom za praktičnu primenu u domeni razvoja informacionih sistema.

## MDD U OKRUŽENJU SOLOIST

Sada ćemo ukratko opisati kako izgleda MDD u jednom semantički homogenom i koheretnom okruženju koje čine OOIS UML i SOLoist.

### *Sprega strukture i ponašanja*

Kada se napravi UML model domena problema korišćenjem nekog alata za modelovanje, kao što je prikazano na slici 2, SOLoist automatski generiše sve neophodne produkte za aplikaciju. To uključuje šemu relacione baze podataka, kao i programski kôd na ciljnom OO jeziku koji se koristi za implementaciju (u ovom slučaju Java).




**Slika 2: –** *Jednostavan UML model.*

Objekti, kao instance klasa iz modela, njihovi atributi i (strukturne) veze kao instance modelovanih asocijacija, imaju UML semantiku i inherentno su perzistentni. Objekat je logički jedinstven, koherentan entitet i ne postoji nikakva semantička razlika između njegove predstave u memoriji i one u bazi podataka, niti je na bilo koji način ova druga reprezentacija vidljiva i dostupna aplikaciji. To se manifestuje i načinom na koji se pristupa objektnom prostoru posredstvom akcija. Akcije se pišu u notaciji odabranog OO programskog jezika implementacije (Java u ovom slučaju). Na primer, nov objekat klase `Employee` pravi se pomoću akcije za pravljenje objekata (*create object*) koja se jednostavno može zapisati na sledeći način:

```
Employee anEmpl = new Employee();
```

Iako je ova notacija (namerno) potpuno ista kao ona na Javi, semantika je različita. Objekat koji se pravi je inherentno perzistentan (odmah upisan i u bazu) i živi sve dok se eksplicitno ne uništi odgovarajućom akcijom (*destroy object*). To se može desiti mnogo kasnije i nezavisno od izvršavanja programa koji je napravio dati objekat:

```
anEmpl.destroy(); // Uništi objekat na
koga ukazuje anEmpl
```

Ne postoji nikakva potreba da se objekat eksplicitno sačuva u bazi (*save*) ili učita iz nje (*load*), ili da se radi sa nekim posebnim „rukovaocem perzistencije“ (engl. *persistence manager*), entitetskim objektima, sesijama i sličnim mehanizmima koji se sreću u drugim postojećim pristupima.

Vrednostima atributa objekata rukuje se izvršavanjem akcija za čitanje i upis. Na primer:

```
anEmpl.name.set(new Text("John Doe"));
...
if (anEmpl.dateOfBirth.val().
isEqualTo(Date.today())) ... // Srećan
rođendan!
```

Kao što se može videti, čitanje i upis vrednosti atributa vrši se eksplicitnim akcijama aktiviranim kroz operacije SOLoist aplikativnog programskog interfejsa (API, `val()` i `set()` u ovom primeru). Pomenuti API nudi potpun skup operacija nad vrednostima atributa, uz podršku za UML atribute sa više

vrednosti (engl. *multivalued*), uz opciono definisanu uređenost (engl. *ordering*). Ovo se donekle razlikuje od zapisa akcija u nekim OO jezicima, gde su akcije čitanja i upisa implicitno definisane pozicijom reference na atribut objekta u izrazu: na primer, dodela vrednosti implicitno označava akciju upisa vrednosti u operand sa leve strane operatora dodele.

Slično važi i za akcije nad vezama kao instancama asocijacija. Na primer, sledeća akcija:

```
anEmpl.dept.set(aDept);
```

pravi novu vezu između objekata na koje ukazuju reference `anEmpl` i `aDept`. Nakon njenog izvršenja, objekat `anEmpl` će biti član kolekcije koja se može dobiti kao rezultat čitanja slota `aDept.members`.

Ponašanje se specificira unutar metoda operacija. Metode se pišu na programskom jeziku Java, što podrazumeva mogućnost korišćenja svih Java kontrolnih struktura, izraza i naredbi. Pristup objektnom prostoru je inkorporiran u jezik pomoću odgovarajućih akcija dostupnih kroz API, kao što je već pomenuto. Metode mogu da pozivaju operacije objekata korišćenjem uobičajene semantike poziva operacija na Javi, uključujući i prenos argumenata i polimorfizam. Reference na objekte su obične Java reference za koje važe uobičajena pravila za tipove (npr. konverzije). Finesa je u tome što reference ukazuju na proksi (engl: *Proxy*) objekte koji nude opisanu semantiku akcija i pristup podacima koji su ili u bazi podataka, ili su keširani u memoriji izvan datih proksi objekata. Međutim, sve to je samo stvar implementacije i potpuno je sakriveno od programera.

Programer ne mora ništa da konfiguriše ili da radi bilo šta drugo da bi postigao opisanu semantiku. Kao u okruženju Ruby on Rails [16], favorizovana je konvencija u odnosu na konfiguraciju. Šema relacione baze podataka se dobija na osnovu podrazumevanog skupa pravila za ORM. Za potrebe poboljšanja performansi u slučaju velikih informacionih sistema i objektnih prostora, postoje opcije za fino podešavanje OR mapiranja. Međutim, ovo fino podešavanje može da se vrši potpuno nezavisno od razvoja aplikacije i ni na koji način ne utiče na model i programski kôd metoda akcija. Celokupno objektno-relaciono mapiranje, sinhronizacija između keširanih objekata (koji su izvan pomenutih SOLoist proksi objekata) i njihovih reprezenata u bazi podataka, kao i semantika UML akcija, obavlja se u okviru izvršnog okruženja koje se sa aplikativnim kodom na Javi, dobijenim iz UML modela, povezuje kao biblioteka.

Podržane su i hijerarhijske mašine stanja (engl. *state machine*) za modelovanje životnog ciklusa objekata, ali je njihov detaljniji opis ovde izostavljen radi konciznosti.

### *Pretraživanje podataka*

Važna mogućnost bilo koje tehnologije koja se koristi za razvoj informacionih sistema jeste podrška zadavanju složenih upita u cilju pretrage podataka ili pravljenja izveštaja. Za te potrebe SOLoist podržava jednu varijantu objektnog upitnog jezika (engl. *Object Query Langugage*, OQL). Sintaksa jezika OQL je slična sintaksi jezika SQL, s tim što se OQL oslanja na objektnu paradigmu. Naša varijanta jezika OQL prilagođena je semantici jezika UML. Sintaksa i semantika ovog jezika je kompletno opisana u [8], a u ovom radu se samo ukratko demonstriraju neke osnovne mogućnosti.

Na primer, za model prikazan na slici 2, sledeći OQL upit vraća imena i datume rođenja svih zaposlenih u odeljenju sa nazivom ‘R&D’:

```
  SELECT empl.name, empl.dateOfBirth
FROM Department d, d.members empl
WHERE d.name='R&D'
```

Može se videti da OQL SELECT upit u svojoj FROM sekciji sadrži izraze za navigaciju preko objekata i veza (npr. `d.members`), umesto spajanja tabela u relacionoj algebri. Razlika je posebno primetna kada je potrebno zadati navigaciju preko asocijacija tipa više-u-više. U tom slučaju, OQL upit izgleda potpuno isto kao i navedeni, dok je u njegovom SQL ekvivalentu potrebno izvršiti spajanje tri tabele: dve za entitete na krajevima asocijacije i jednu za samu relaciju više-u-više. Što je navigacija u upitu kompleksnija, to je razlika između ekvivaletnih OQL i SQL upita veća.

OQL podržava nasleđivanje i polimorfizam. Na primer, u sledećem upitu postoji pristup nasleđenim atributima `name` i `members` klase `Headquarter`:

```
  SELECT hq, hq.name, empl, empl.name
FROM Headquarter hq, hq.members empl
```

Sledeći upit će vratiti skup svih odeljenja u kojima rade članovi koji u imenu imaju reč ‘John’; u dati skup će biti uključeni i šefovi odeljenja:

```
  SELECT dept
FROM Employee empl, empl.dept dept
WHERE empl.name LIKE '%John%'
```

OQL podržava i specijalizaciju. Ako je potrebno pronaći samo šefove odeljenja koji u imenu imaju reč ’John’, potrebno je napisati sledeći upit:

```
  SELECT hq
FROM Employee empl, empl.dept:Headquarter
hq
WHERE empl.name LIKE '%John%'
```

Rezultat nekog OQL upita je kolekcija redova kroz koju se može iterirati iz Java koda, ili se ona može prikazati u korisničkom interfejsu. Reference na objekte i vrednosti u komponentama redova koriste se na već opisan način.

Naš OQL podržava i mnoge druge mogućnosti, kao što su agregatne funkcije u SELECT delu, GROUP BY, ORDER BY i HAVING, unije, podupiti i parametrizovani upiti.

### *Sprega sa korisničkim interfejsom*

Najzahtevniji deo razvoja informacionih sistema najčešće je razvoj korisničkog interfejsa. Da bi što više smanjilo udeo slučajne kompleksnosti i razvoj učinilo što efikasnijim, naše okruženje obezbeđuje čvrstu semantičku spregu objektnog prostora i korisničkog interfejsa korišćenjem jedne nove paradigme za pravljenje korisničkog interfejsa. Ovde dajemo samo njen kratak prikaz, dok se detaljniji opis može pronaći u [8, 9, 19].

Motivacija i ideja koja leži u osnovi naše paradigme ilustrovana je na slici 3. Jedan jednostavan formular u aplikaciji, čiji je model prethodno prikazan na slici 1, radi na sledeći način. Prikaz u vidu stabla na levoj strani slike 3a je konfigurisan tako da iscrtava organizacionu hijerarhiju odeljenja u preduzeću. Tri kontrole desno od stabla prikazuju vrednosti atributa onog odeljenja koje je trenutno odabrano u prikazanom stablu. Prve dve kontrole su tekstualna polja posredstvom kojih je moguće pregledati i ažurirati vrednosti atributa `name` i `description` objekta klase `Department`. Treća kontrola je vertikalna lista koja prikazuje sve zaposlene koji pripadaju odabranom odeljenju.

Ponašanje datih kontrola može se apstraktno opisati na sledeći način. Kad god se promeni izabrani objekat klase `Department` u levom stablu, svaka od tri kontrole sa desne strane mora da prikaže vrednost odgovarajućeg slota tog objekta (`name`, `description`, `members` – kolekcija vezanih objekata klase `Employee`). Očigledno je da objekat klase `Department`, koji je odabran u stablu, treba da bude dinamički prosleđen kao ulazni parametar svakoj od kotrola na desnoj strani. Svojstvo (atribut ili asocijacioni kraj) klase `Department`, čija će vrednost biti prikazana u nekoj od tri kontrole desno od stabla, konfiguriše se u vreme razvoja samog korisničkog interfejsa. Sa druge strane, sam objekat klase `Department`, čiji slot (kao instanca konfigurisanog svojstva) treba da bude prikazan, se dinamički menja, kao odgovor na korisničke akcije.

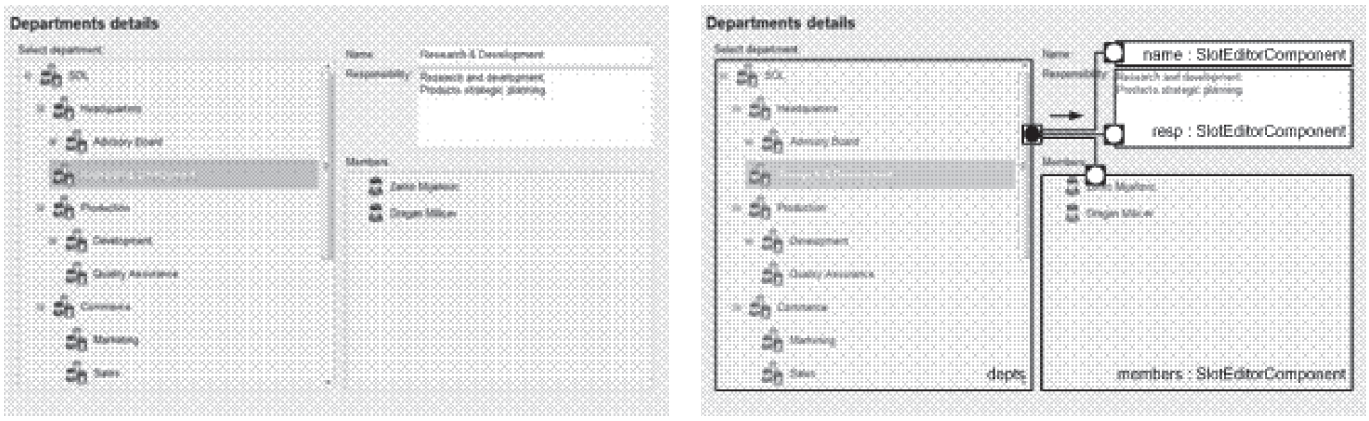
Slika 3b prikazuje način funcionisanja korisničkog interfejsa prema prethodnom objašnjenju. Kontrole na korisničkom interfejsu mogu se logički posmatrati kao komponente čiji interfejs čine tzv. *pinovi*. Pin se koristi za komunikaciju sa drugim komponentama. Kroz njega neka komponenta može da šalje ili prima signale ili podatke ka ili od drugih komponenata. Na primer, stablo kao komponenta korisničkog interfejsa ima jedan izlazni pin kroz koji šalje referencu na objekat koji je trenutno odabran u stablu, i to radi svaki put kada korisnik svojom akcijom odabere neki objekat. Svaka od tri komponente sa desne strane ima jedan ulazni pin koji prihvata referencu na objekat čiju vrednost na slotu, prema konfiguraciji, treba da prikaže ili ažurira. Interno ponašanje kontrole obezbeđuje da svaki put kada se nova vrednost pojavi na njenom ulaznom pinu, kotrola pristupa objektnom prostoru, čita odgovarajući slot objekta i prikazuje njegovu trenutnu vrednost.

Da bi sve funcionisalo na opisani način, programer samo treba da poveže izlazni pin komponente za prikaz stabla na ulazne pinove tri pomenute komponente pomoću tzv. *žica*. Žice predstavljaju veze kroz koje će teći podaci od izlaznog ka jednom ili više ulaznih pinova kad god se nova vrednost pojavi na izlaznom pinu, kao što je prikazano na slici 3b. Komponente se prave kao objekti i povezuju se kroz SOLoist API ili kroz modele, kao što je opisano u [9].

SOLoist nudi bogatu biblioteku gotovih komponenata za pravljenje veb baziranog korisničkog interfejsa. Ove komponente s jedne strane predstavljaju uobičajene kontrole korisničkog interfejsa, kao što su tekstualna polja, polja za potvrdu, padajuće liste, stabla, slike, tabele i mnoge druge, dajući korisniku standardni prikaz i ponašanje interfejsa, dok za komunikaciju sa drugim komponentama koriste svoje pinove. Sa druge strane, ove komponente podižu nivo apstrakcije i direktno se sprežu sa objektnim prostorom sa OOIS UML semantikom.


**Slika 4:** – *Jedna jednostavna kopomenta korisničkog interfejsa za ažuriranje slota kraja asocijacije.*

Kao jedan ilustrativan primer komponente koja se lako konfiguriše na visokom nivou apstrakcije, opisaćemo komponentu korisničkog interfejsa prikazanu na slici 4. Ova komponenta omogućava ažuriranje slota objekta koji se prosleđuje na njen ulazni pin `elem`. U prikazanom primeru, taj objekat je instanca klase `Employee` i u ovom kontekstu ćemo taj objekat nazivati




a) b)

**Slika 3:** – ***(a) Primer formulara sa povezanim komponentama korisničkog interfejsa. (b) Funkcionalno povezivanje komponenata korisničkog interfejsa posmatrano iz perspektive prosleđivanja poruka.***

ciljnim. Slot je instanca asocijacionog kraja iz UML modela domena. Asocijacioni kraj se konfiguriše kada se komponenta definiše. U ovom primeru, to je asocijacioni kraj `dept` kojim se zaposleni povezuje sa odeljenjem kojem pripada. Data komponenta prikazuje kolekciju objekata određene klase koji su kandidati za uspostavljanje veza sa ciljnim objektom preko konfigurisanog asocijacionog kraja (`dept`). Komponenta može biti konfigurisana tako da prikazuje listu objekata koji se dobijaju iz kolekcije prosleđene preko odgovarajućeg ulaznog pina, ili stablo sa prosleđenim korenim objektom i podstablima koja se prostiru preko veza na nekom drugom konfigurisanom kraju asocijacije (`subdivisions` u ovom slučaju). Kad god se novi ciljni objekat klase `Employee` pojavi na ulaznom pinu `elem`, komponenta dohvata objekat (ili više vezanih objekata u opštem slučaju) koji je vezan sa ciljnim objektom preko njegovog slota `dept` i ažurira polja za potvrdu da prikažu dati vezani objekat (ili više njih u opštem slučaju). Kad god korisnik promeni stanje datih polja za potvrdu, komponenta ažurira stanje u objektnom prostoru tako što pravi i/ili briše veze između objekata. Prema tome, da bi se omogućilo opisano tipično ponašanje, programer ne treba da piše nikakav programski kôd na tradicionalan način – recimo, ne treba da piše imperativni kod rutina za obradu događaja u komponenti (engl. *event handler*). Umesto toga, sve se postiže deklarativnim pristupom, samo konfigurisanjem date komponente i povezivanjem njenih pinova sa pinovima drugih komponenata u njenom okruženju.

Naša biblioteka je bogata sofisticiranim komponenatama koje funkcionišu u skladu sa opisanim principima. Njihove implementacije su prilagođene za veb. Sloj veb klijenta implementira celokupan korisnički interfejs, dok se objektni prostor nalazi na serveru. Za implementaciju klijentskog sloja koristi se GWT (*Google Web Toolkit*). Komponente iz naše biblioteke zapravo obavijaju GWT kontrole. Klijentski sloj pristupa objektnom prostoru slanjem AJAX zahteva: kad god neka komponenta treba da dovuče ili modifikuje deo objektnog prostora, ona šalje AJAX zahtev, konkuretno i nezavisno od ostalih komponenata. Domenskom objektnom prostoru pristupa se kroz UML refleksiju koju obezbeđuje izvršno okruženje.

## ISKUSTVA IZ PRAKSE

Opisani MDD pristup inicijalno je osmišljen još u ranim 2000-tim godinama. Od tada su razvijene četiri generacije prikazanog okruženja na različitim programskim jezicima (Visual Basic, C++, Java) i na različitim platformama i arhitekturama za pravljenje korisničkog interfejsa (desktop: Visual Basic, MFC, Qt; veb: GWT). Ove četiri generacije imale su veoma različite karakteristike, nivoe semantičke integracije i internu arhitekturu i dizajn. Korišćenjem ove četiri generacije tehnologije implementirano je nekoliko desetina komercijalnih industrijskih projekata različitih veličina i iz različitih domena problema. Ovde ćemo ukratko izneti iskustva iz poslednja tri velika projekta (od nacionalnog značaja), realizovana pomoću najnovije, četvrte generacije tehnologije bazirane na vebu.

U prvom projektu razvijan je sistem sa organizaciju društvenih manifestacija (*Event Management System*, EMS). Taj sistem je visoko konfigurabilan proizvod za podršku organizaciji različitih društvenih događaja, kao što su sportska takmičenja, simpozijumi, konferencije itd. Podržana je registracija učesnika, odobravanje prijava, izrada akreditacija, definisanje pristupa, konfigurisanje izgleda propusnice, štampanje i izdavanje propusnica, vođenje evidencije o prostoru, raspoređivanje događaja itd.

U drugom projektu razvijan je sistem za upravljanje ljudskim resursima vlade (*Human Resource Management System*, HRMS). Ovaj sistem podržava centralizovano upravljanje otvorenim pozicijama u svim vladinim organizacijama na nivou cele države. Razvijene su i aplikacije za obaveštavanje, pravljenje profila kandidata, testiranje sposobnosti kandidata za posao, upravljanje procesom zapošljavanja i mnoge druge.

U trećem projektu razvijan je sistem za nacionalni katastar nepokretnosti (*Real-Estate Cadastre System*, RECS). Ovaj sistem podržava sve uobičajene koncepte i funcionalnosti za jedan sistem te vrste, uključujući katastarske entitete (parcele, zgrade, stanove), kao i pravne koncepte (vlasništvo, tereti, hipoteke itd.).

U tabeli 1 prikazane su neke osnovne metrike UML modela domena za ova tri sistema. Ove metrike pokrivaju strukturne delove modela (klase, atribute i asocijacije), kao i netrivijalne funcionalnosti poslovne logike datih sistema (u vidu domenski specifičnih klasa komandi). Komande su klase čiji objekti omogućavaju obradu zahteva za uslugama sistema koji su izdati ili interaktivno kroz grafički interfejs ili poslati iz spoljašnjih sistema sa kojima dati sistem komunicira. Domenski specifične komande predstavljaju pristupne tačke do implementacije specifičnih netrivijalnih funkcionalnosti iz domena problema, koje nisu direktno podržane od strane izvršnog okruženja SOLoist ili generičkih komandi koje izvršavaju opštenamenske akcije nad objektnim prostorom. Treba napomenuti i to da date metrike pokrivaju samo domenske (perzistentne) klase iz konceptualnih modela, ali ne i one koje su vezane za korisnički interfejs. Na neki način ove metrike stoga daju informaciju o suštinskoj kompleksnosti sistema, koja je oslobođena od nepotrebne slučajne kompleksnosti.

| Metrike | *Sistem* | | |
|---|---|---|---|
| | EMS | HRMS | RECS |
| Broj domenskih klasa – ukupno/apstraktne/ konkretne | 81/13/68 | 86/5/81 | 196/17/179 |
| Broj domenskih atributa – ukupno/maksimalno/ prosečno po klasi | 218/13/2.69 | 566/52/6.58 | 280/17/1.43 |
| Broj domenskih asocijacija/asocijacionih krajeva | 79/158 | 132/264 | 254/508 |
| Broj navigabilnih domenskih krajeva asocijacije – ukupno/maksimalno/ prosečno po klasi | 139/11/1.72 | 228/18/2.65 | 336/29/1.71 |
| Broj svojstava klasa (atributi + navigabilni asocijacioni krajevi) – ukupno/maksimalno/prosečno po klasi | 357/18/4.41 | 794/54/9.23 | 616/46/3.14 |
| Dubina hijerarhije klasa – makimalna/prosečna | 4/1.95 | 3/1.24 | 5/2.36 |
| Broj domenskih enumeracija | 10 | 32 | 15 |
| Broj klasa komandi | 63 | 186 | 193 |
| Ukupan broj klasa (domenske + komande) | 144 | 272 | 389 |

**Tabela 1: –** *Metrike UML modela domena za tri industrijska projekta.*

U tabeli 2 prikazane su metrike vezane za korisnički interfejs opisana tri sistema. Date informacije odgovaraju celokupnoj strukturi korisničkog interfejsa koja pokriva ceo opseg funcionalnosti sistema.

| *Metrike* | *Sistem* | | |
|---|---|---|---|
| | EMS | HRMS | RECS |
| Ukupan broj komponenata korisničkog interfejsa | 16,883 | 6,925 | 59,448 |
| Dubina hijerarhijskog ugneždavanja komponenata – maksimalna/ prosečna | 15/8.55 | 21/11.23 | 32/16.40 |
| Broj komponenata koje sadrže druge komponente | 2,058 | 1,928 | 18,046 |
| Broj primitivnih komponenata | 14,825 | 4,997 | 41,402 |
| Broj tzv. "žica" | 12,104 | 4,255 | 31,285 |
| Maksimalan/prosečan broj komponenata koje se nalaze u sklopu neke složene komponente | 101/8.22 | 65/3.60 | 57/3.28 |

**Tabela 2: –** *Metrike korisničkog interfejsa za tri industrijska projekta.*

Najvažniji zaključak iz našeg praktičnog iskustva jeste taj da MDD zaista dobro deluje u praksi i jeste primenjiv na projekte svih veličina, ukoliko se naravno primeni na odgovarajući način, uz podršku okruženja koje obezbeđuje odgovarajuću semantičku spregu, kao što je to u našem pristupu.

Uočili smo takođe i mnoge druge dobiti od korišćenja semantički koherentnog okruženja. Slučajna kompleksnost se umnogome smanjuje u poređenju sa ostalim popularnim pristupima koji su jezički i semantički heterogeni. Takav pristup vodi ka konciznijim, ali i izražajnijim produktima u toku razvoja (model i kôd). Broj načina na koje se određeni funkcionalni i drugi zahtevi mogu uklopiti u postojeću aplikaciju je smanjen, što je dobro, jer implementacija tako postaje pravolinijska i nezavisna od stila rada programera koji treba to da uradi. Samim tim je i implementirana aplikacija lakša i za razumevanje, i za održavanje.

Na kraju, razvojni ciklus je umnogome kraći. Ovu tvrdnju nije lako dokazati i kvantifikovati, i to ovom prilikom i ne pokušavamo da uradimo. Pomenućemo samo da je za razvoj pomenuta tri sistema bilo potrebno, na osnovu grube evidencije, od 12 (EMS) do 36 (RECS) čovek-meseci. (Nezahvalno je davati izveštaj o utrošenim resursima i to činimo sa dozom rezerve, ali bismo želeli da istaknemo da je utrošak resursa i vremena bio isključivo proporcionalan suštinskoj, logičkoj i funkcionalnoj kompleksnosti domena problema.) Iako nismo eksperimentima potvrdili koliko bi vremena bilo potrebno za razvoj istih ovih sistema korišćenjem drugih pristupa, budući da smo bili koncentrisani na razvoj industrijskih projekata koji su imali veoma tesne vremenske rokove, možemo reći da su naši klijenti, kao i naši partneri koji su iskusni i dobro uvežbani u radu sa mnogim drugim popularnim okruženjima i alatima, bili iznenađeni i izuzetno zadovoljni efikasnošću razvoja kompleksnih sistema korišćenjem našeg pristupa.

Naravno, kao što to biva i sa drugim pristupima i alatima, i naš pristup je imao određene poteškoće i slabosti, posebno u svojim ranijim fazama.

Strah od novog, nepoznatog i od iznenađenja je jedna od glavnih prepreka za uvođenje novih metoda u praktičnu upotrebu, pa to važi i za MDD. Iako potpuno razumemo ovaj strah, mislimo da on ne treba da spreči napredak, pod uslovom da je potencijal novog pristupa jasno prepoznat. Da bi neka tehnologija postala dovoljno zrela, potrebno je uložiti određeni napor i strpljenje, pod uslovom da ona inherentno počiva na zdravim principima. Iskustvo nam govori da realnim problemima treba uvek pristupiti pažljivo i sa tehničkim osećajem i, ako je softver realizovan u skladu sa svim principima softverskog inženjerstva (odgovarajuća apstrakcija, lokalizacija projektnih odluka, enkapsulacija, jasna i koherentna arhitektura, itd.), praktično svaki tehnički problem ili rizik može da se prevaziđe bez ugrožavanja osnovne ideje.

Naše decenijsko iskustvo je puno takvih praktičnih primera. Jedan od najvažnijih je vezan za aspekt performansi sistema. Veoma su česte i tipične osude i sumnje da MDD, kao i bilo koji drugi visoko apstraktni pristup, unosi dodatne i neprihvatljive režijske troškove u poređenju sa tradicionalnijim pristupima, posebno u pogledu vremena izvršavanja i zauzeća memorijskog prostora. I mi smo povremeno imali probleme sa performansama, ali smo ih uspešno rešili finim podešavanjem ili uvođenjem pametnih optimizacija u ORM ili izvršno okruženje, i to bez uticaja na celokupnu koncepciju. U našem poslednjem projektu, sistem koji je radio sa objektnim prostorom od nekoliko stotina miliona objekata i korisničkim interfejsom od nekoliko hiljada komponenata, uopšte nije bio inferioran u pogledu performansi u odnosu na sisteme koji se mogu implementirati ručno. Postignute performanse su u nekim slučajevima bile čak i bolje, jer su automatski prevedeni veoma kompleksni upiti bili mnogo efikasniji zbog primenjenih optimizacija od onih koji se mogu napisati ručno. Istorija softverskog inženjerstva je puna sličnih primera u kojima se povećanje nivoa apstrakcije smatralo inferiornim u odnosu na tradicionalne i uhodane pristupe. Podsetimo se samo pojave viših programskih jezika i tvrdnji da „kôd napisan na takvim jezicima ne može nikad biti tako dobar kao što se može ručno napisati na asembleru", ili da je „Java kôd mnogo sporiji od kompajliranog koda jer se interpretira na virtuelnoj mašini." Naravno, apstrakcija unosi određene režije, ali te režije mogu biti sasvim prihvatljive (ako se svedu na razumnu meru) u odnosu na dobiti koje apstrakcija pruža [17].

## PREGLED SRODNIH REŠENJA

Ideja korišćenja jezika UML kao izvršivog programskog jezika je verovatno stara koliko i sam UML. Jedna od prvih UML virtuelnih mašina [15] bila je zasnovana na interpretiranju UML modela, umesto na generisanju kôda na ciljnom programskom jeziku. Istraživanje koje su radili Mellor i Balcer [7] dalo je jednu od prvih opštih metoda za korišćenje UML-a kao izvršivog jezika. Iako je u tom istraživanju izvršivi UML korišćen kao jezik opšte namene, modelovanje ponašanja bilo je bazirano na mašinama stanja unutar samih objekata i tako najpodesnije za sisteme pokretane događajima. Skorija inicijativa od strane OMG grupe, vezana za osnovni podskup elemenata izvršivih UML modela (fUML) [13], sa druge strane, pokriva izvršivom semantikom veoma široku paletu koncepata za modelovanje strukture i ponašanja. Međutim, nijedan od ovih pristupa nije posebno profilisan za primenu na aplikacije na koje smo mi fokusirani. Ponašanje ovih aplikacija se u principu ostvaruje kroz upravljanje složenim strukturama objekata u formi grafova definisanih konceptualnim modelima, umesto kroz način reagovanja na spoljašnje događaje u zavisnosti od internog stanja objekata. Nijedan od pomenutih pristupa pak ne nudi koncepte specifično prilagođene nekim veoma važnim aspektima razvoja informacionih sistema kao što su korisnički interfejs ili pretraživanje podataka.

Naš sloj za ORM, kao i naša verzija jezika OQL, svakako imaju dosta sličnosti sa drugim okruženjima, kao što su na primer Hibernate sa svojim upitnim jezikom HQL ili JEE JPA sa svojim JPQL [5]. Svi ovi upitni jezici ima sintaksu sličnu SQL-u, ali su u osnovi bazirani na objektno orijentisanoj umesto na relacionoj semantici. Međutim, ova druga rešenja baziraju se na čistoj semantici Java objekata, dok je naše bazirano na UML-u. Naš pristup ne zahteva nikakvu vrstu anotacija u izvornom kodu ili konfiguracione fajlove da bi se definisalo objektno-relaciono mapiranje. U našem pristupu jedino semantičko okruženje jeste izvršivi UML, dok u drugim pristupima programer mora da obezbedi spregu između dva semantički različita okruženja. Najzad, naše mapiranje nudi neke jedinstvene optimizacione tehnike, kao što su redundantne kopije atributa u tabelama izvedenih klasa za efikasnije izvršavanje složenih upita, kao i za specijalizaciju (konverzija tipa nadole), što nije podržano u drugim pristupima.

Koncept pinova komponenata korisničkog interfejsa je veoma blizak konceptima signala i slotova u okruženju Qt [11]. Međutim, Qt ne sprečava upotrebu svih drugih mehanizama sprege ponašanja i interakcije među komponentama. U Qt-u slotovi su implementirani kao obične operacije C++ klasa, dok objekti tih klasa mogu da interaguju preko svih drugih uobičajenih C++ mehanizama. Naš pristup obezbeđuje veoma strogu enkapsulaciju zato što interfejs komponenata čine samo pinovi, i jedini način na koji neka komponenta može da interaguje sa svojim okruženjem jeste preko poruka koje se šalju ili primaju kroz pinove. Naša tehnika razvoja korisničkog interfejsa je inspirisana metodologijom ROOM [18] umesto pristupom Qt, dok su naši koncepti semantički kompatibilni sa konceptima strukturiranih klasa, portova i konektora iz jezika UML 2. Naša paradigma razvoja korisničkog interfejsa je generalizovana u apstraktniju tehniku modelovanja koja olakšava ponovnu upotrebu delova korisničkog interfejsa u veoma složenim aplikacijama [9].

Sve pomenute tehnologije su ili suviše opšte i neprilagođene oblasti informacionih sistema, ili suviše parcijalne, jer pokrivaju samo neke aspekte razvoja informacionih sistema (npr., samo ORM ili samo UI). Naše rešenje predstavlja celovito i semantički koherentno okruženje koje pokriva sve najvažnije aspekte razvoja informacionih sistema. U tom smislu, naš pristup je bliži rešenjima kao što su Intelliun Visual Enterprise [4], ManyDesign Portofino [6] ili WebRatio [23]. Naš pristup, međutim, nudi novu paradigmu razvoja korisničkog interfejsa koja se dobro skalira za velike aplikacije.

## ZAKLJUČAK

U ovom radu smo prikazali jedno semantički homogeno i koherentno okruženje za razvoj informacionih sistema zasnovano na modelima i jednom izvršivom profilu jezika UML. Ukratko smo izneli i neka naša najvažnija iskustva iz korišćenja opisanog rešenja u velikim industrijskim projektima. Naše iskustvo je nedvosmisleno pokazalo da se MDD može koristiti kao veoma efikasno sredstvo za savladavanje suštinske složenosti velikih informacionih sistema, pod uslovom da je primenjen na adekvatan način. Adekvatan način, po našem mišljenju, podrazumeva korišćenje modela kao centralnih i izvršivih produkata rada, kao što je to izvorni kôd u tradicionalnim pristupima. U takvom pristupu, modeli više nisu samo skice, već postaju precizne i formalne specifikacije projektnih odluka, kao i autoritativne specifikacije softvera. Na ovaj način se sprečava sindrom preuranjenog pisanja kôda, uz istovremeno ostvarivanje svih prednosti korišćenja jezika visokog nivoa apstrakcije.

## REFERENCE


[1] Google Web Toolkit, http://code.google.com/webtoolkit/, očitano maja 2012.
[2] Groenewegen, D., Hemel, Z., Visser, E., "Separation of Concerns and Linguistic Integration in WebDSL," *IEEE Software*, Vol. 27, No. 5, Sept./Oct. 2010, pp. 31-37
[3] Hibernate, http://www.hibernate.org, očitano maja 2012.
[4] Intelliun Visual Enterprise, http://www.intelliun.com, očitano maja 2012.
[5] Java Persitence API, http://download.oracle.com/javaee/6/tutorial/doc/bnbpy.html, očitano juna 2011.
[6] ManyDesigns Portofino, http://www.manydesigns.com, očitano maja 2012.
[7] Mellor, S. J., Balcer, M., *Executable UML: A Foundation for Model-Driven Architectures*, Addison-Wesley Longman, 2002
[8] Milićev, D., *Model-Driven Development with Executable UML*, John Wiley & Sons (WROX), 2009
[9] Milićev, D., Mijailović, Ž., "Effective Modeling of User Interfaces for Large-Scale Applications," *submitted for publication*
[10] Nunes, N. J., E Cunha, J. F., "Wisdom - A UML Based Architecture for Interactive Systems," *Proc. DSV-IS*, 2000, pp. 191-205
[11] Nokia Qt, http://qt.nokia.com/, očitano juna 2011.
[12] Object Management Group, *UML 2.2 Superstructure Specification*, http://www.omg.org, February 2009
[13] Object Management Group, *Semantics of a Foundational Subset for Executable UML Models (fUML) V1.0*, http://www.omg.org, February 2011
[14] Olivé, A., *Conceptual Modeling of Information Systems*, Springer, 2007
[15] Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N., "The architecture of a UML virtual machine," *Proc. 16th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '01), 2001
[16] Ruby on Rails, http://rubyonrails.org, očitano juna 2011.
[17] Selic, B., "The Pragmatics of Model-Driven Development," *IEEE Software*, Vol. 20, No. 5, Sept./Oct. 2003, pp. 19-25
[18] Selic, B., Gullekson, G., Ward, P. T., *Real-Time Object-Oriented Modeling*, John Wiley and Sons, 1994
[19] SOLoist, http://www.soloist4uml.com, očitano maja 2012.
[20] Viswanathan, V., "Rapid Web Applications Development: Ruby on Rails Tutorial," *IEEE Software*, Vol. 25, No. 6, Nov./Dec. 2008, pp. 98-106
[21] Wilson, S., Johnson, P., Kelly, C., Cunningham, J., Markopoulos, P., "Beyond Hacking: a Model Based Approach to User Interface Design", *Proc. HCI*, 1993, p. 217
[22] Woods, E., Emery, D., Selic, B., "Point/Counterpoint," *IEEE Software*, Vol. 27, No. 6, Nov./Dec. 2010, pp. 54-57
[23] WebRatio, www.webratio.com, očitano maja 2012.



prof. dr Dragan Milićev,
Elektrotehnički fakultet Univerziteta u Beogradu,
dmilicev@etf.bg.ac.rs
Oblasti iteresovanja: Softversko inženjerstvo zasnovano na modelima, razvoj softvera pomoću modela, modelovanje u specifičnim domenim, metamodelovanje, Model-Driven Architecture, transformacije modela, UML, Brzi razvoj aplikacija i softverski alati za razvoj i vizuelizaciju softvera, Informacioni sistemi, Konkurentno i distribuirano procesiranje i programiranje, paralelizacija koda, instrukcijski nivo paralelizma, Objektno orijentisane tehnologije, programiranje i jezici



master-inž. Nemanja Kojić,
Elektrotehnički fakultet Univerziteta u Beogradu,
nemanja.kojic@etf.bg.ac.rs
Oblasti interesovanja: Softversko inženjerstvo zasnovano na modelima, razvoj softvera pomoću modela, Informacioni sistemi, Konkurentno i distribuirano programiranje, Objektno orijentisane tehnologije, programiranje i jezici, Veštačka inteligencija i pronalaženje skirvenog znanja, Modelovanje performansi računarskih sistema