Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Blaž Ahačič

Prikaz uporabe modelno vodenega razvoja z orodjem Eclipse Modeling Framework

diplomsko delo

visokošolski strokovni študijski program
računalništvo in informatika

Mentor: viš. pred. dr. Igor Rožanc

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Modelno voden razvoj (Model Driven Development) je sodobna paradigma razvoja programske opreme, ki z uporabo modeliranja in transformacij med modeli na različnih ravneh obeta učinkovitejši razvoj programskih rešitev. Uporaba visokonivojskih konceptov na višjih nivojih abstrakcije in avtomatsko generiranje kode naj bi pohitrilo razvoj z zmanjšanjem deleža neposrednega programiranja rešitev, obenem pa poskrbelo za vrsto drugih prednosti kot so višja kvaliteta rešitev, ponovna uporabljivost, prenosljivost v različna izvajalna okolja in usklajeno dokumentiranje.

V diplomski nalogi preverite, v kolikšni meri tovrstni obeti držijo. V ta namen najprej prestavite modelno voden pristop in vrsto njegovih izvedb (MD*), še zlasti Model Driven Architecture - MDA ter njeno konkretno implementacijo Eclipse Modeling Framework - EMF. Njeno uporabo in zmožnosti predstavite na krajšem zgledu, ki naj ima za cilj izdelati celovito aplikacijo od idejne zasnove dalje. Zadnji del naloge naj bo vaša analiza uporabe tovrstnih rešitev v primerjavi s klasičnim razvojem.

Izjava o avtorstvu zaključnega dela

Spodaj podpisani Blaž Ahačič, vpisna številka 63960001, avtor zaključnega dela z naslovom:

Prikaz uporabe modelno vodenega razvoja z orodjem Eclipse Modeling Framework (angl. *The demonstration of Model Driven Development using Eclipse Modeling Framework*)

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom viš. pred. dr. Igorja Rožanca;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil/-a vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil/-a;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal/-a v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil/-a soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 26. maja 2016 Podpis študenta/-ke:

Zahvaljujem se mentorju viš. pred. dr. Igorju Rožancu za vodenje in koristne komentarje pri izdelavi diplomskega dela. Posebna zahvala pa gre moji družini za vzpodbude in potrpežljivost. Brez vas tega diplomskega dela ne bi bilo.

Moji družini.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	programski vmesnik
CIM	Computational Independent Model	računsko neodvisni model
CWM	Common Warehouse Metamodel	jezik za modeliranje metapodatkov podatkovnih skladišč
EMF	Eclipse Modeling Framework	Eclipse modelirno ogrodje
MBE	Model Based Engineering	modelno osnovano inženirstvo
MDA	Model Driven Architecture	modelno vodena arhitektura
MDD	Model Driven Development	modelno vodeni razvoj
MDE	Model Driven Engineering	modelno vodeno inženirstvo
MOF	Meta Object Factory	standard za formalno definicijo modelov in metamodelov
OCL	Object Constraint Language	jezik za podajanje omejitev za UML
OMG	Object Management Group	organizacija, ki je razvila standard MDA
PSM	Platform Specific Model	model odvisen od računalniškega okolja
PIM	Platform Independent Model	model neodvisen od računalniškega okolja
QVT	Query View Transformation	množica jezikov za definiranje poizvedb, pogledov in preslikav modelov
UML	Unified Modeling Language	poenoteni jezik modeliranja
XMI	XML Metadata Interchange	jezik za izmenjavo metapodatkov v XML obliki
XML	Extensible Markup Language	razširljivi označevalni jezik
XP	Extreme Programming	ekstremno programiranje

Povzetek

Naslov: Prikaz uporabe modelno vodenega razvoja z orodjem Eclipse Modeling Framework

Diplomsko delo obravnava modelno voden razvoj, ki predstavlja eno izmed obetavnejših smernic v razvoju programske opreme. Z načrtovanjem modela problemske domene kot abstraktnejšim načinom podajanja navodil računalniku, pristop napoveduje povečanje učinkovitosti razvoja, saj predstavlja model hkrati tudi gradnik, na podlagi katerega se avtomatično izdela končna programska rešitev. V trenutni fazi razvoja področja modelirni jeziki žal še ne omogočajo učinkovitega opisa celotne rešitve, zato je potrebno v pristop vključevati klasične tehnike razvoja.

Obstaja več različic modelnih pristopov, najbolj specifična in razširjena pa je različica imenovana MDA, ki za glavni modelirni jezik uporablja jezik UML. Orodje EMF, ki predstavlja konkretno izvedbo standardov MDA, omogoča avtomatsko izdelavo programske rešitve na podlagi modela definiranega z razrednimi diagrami. Z ročnimi spremembami generirane kode omogoča orodje prilagoditev bolj specifičnim in podrobnejšim zahtevam.

Z orodjem EMF je bila izdelana celovita rešitev za definiranje jedilnika restavracije. Na podlagi praktičnih izkušenj, ki so bile ob tem pridobljene, je bilo ugotovljeno, da je največja učinkovitost uporabe orodja dosežena pri izdelavi podatkovno intenzivnih aplikacij, katerih glavne lastnosti so ponavljajoča koda, generični uporabniški vmesniki in ohlapno definirane zahteve. Izkazalo se je tudi, da je za podporo bolj specifičnim in podrobnejšim zahtevam še vedno potrebno veliko programerskega znanja, kar pa zahteva strmo

krivuljo učenja.

Ključne besede: modelno voden razvoj, modelno vodena arhitektura, Eclipse Modeling Framework, Ecore, Acceleo.

Abstract

Title: The demonstration of Model Driven Development using Eclipse Modeling Framework

This thesis presents model driven development that is one of more promising paradigms for software development. By modeling problem domains at a higher abstract level it promises an efficient new way of providing the computer with instructions for building a system. The model being the main artefact for automatic generation of software. Unfortunately modelling languages do not yet posses enough descriptive power to effectively describe solution in enough details so classical programming approach needs to be used as well.

There are many model driven approaches, MDA being the most specific and widely used. It uses UML as main modelling language. EMF tool implements MDA standards and uses class diagrams for defining models that are used for automatic code generation. Special and detailed requirements can be implemented by using native tool support for manual modification of generated code.

EMF tool was used for writing an application for definition of restaurant menus. It has been found, on the basis of practical experiences gained at using this tool, that EMF tool is most efficient at writing data-centric applications with a lot of repetitive code, generic user interfaces and loosely defined requirements. It has also been found that a lot of programming knowledge with steep learning curve is still needed for implementing specific and detailed requirements.

Keywords: Model Driven Development, Model Driven Architecture, Eclipse Modeling Framework, Ecore, Acceleo.

Poglavje 1

Uvod

Moderna programska oprema postaja vse bolj zahtevna, s tem pa tudi delo preslikovanja zahtev v končno rešitev predstavlja vedno večji izziv. Upravljanje zahtevnosti sistema in osredotočanje na poslovne zahteve se lahko doseže s pomočjo abstrakcije in deljenja odgovornosti (ang. separation of concern). Oboje predstavlja glavni koncept modelno vodenega razvoja (ang. Model Driven Development - MDD) [2].

MDD je pristop k razvoju programske opreme, ki kot glavni aktivnosti v življenski cikel razvoja programske opreme uvaja modeliranje in preslikave med modeli. Modeli pomagajo razvijalcem pri spopadanju z velikimi in zahtevnimi sistemi na način, ki omogoča obdelavo visoko nivojskih konceptov na pravem nivoju abstrakcije. Posledično se osredotočenost razvoja programske opreme premakne od pisanja programske kode za specifično računalniško okolje (ang. platform) k razvoju rešitve, ki uspešno zadovoljuje poslovne zahteve.

Modeli so uporabljeni za avtomatizacijo večine kodiranja aplikacij z izdelavo modelov odvisnih od računalniškega okolja (ang. Platform Specific Model - PSM) iz modelov neodvisnih od računalniškega okolja (ang. Platform Independent Model - PIM) in dejanske kode iz PSM. Na ta način so modeli generatorji kode in ne obratno. MDD daje arhitektom možnost definiranja in komuniciranja rešitve ob hkratnem ustvarjanju izdelka, ki postane

del končne rešitve.

MDD z ločevanjem med poslovnim znanjem, ki predstavlja "KAJ"del opisa sistema, in tehničnim znanjem, ki predstavlja "KAKO"del, udejanja načelo deljenja odgovornosti. S tem pa MDD doseže tri glavne cilje: prenosljivost, povezljivost in ponovno uporabo. Te lastnosti peljejo v smeri povečane učinkovitosti razvoja programske opreme.

Namen te diplomske naloge je proučiti MDD in odgovoriti na vprašanje o praktičnosti uporabe modelno vodenega razvoja programske opreme, ki je dosežena z današnjimi orodji, konkretno z orodjem Eclipse Modeling Framework, v nadaljevanju EMF [7]. V prvem delu diplomske naloge bomo najprej predstavili modelno vodeno arhitekturo (ang. Model Driven Architecture - MDA) [4] kot konkretno izpeljanko MDD konceptov. Nadaljevali bomo s predstavitvijo orodja EMF, ki predstavlja konkretno izvedbo MDA standardov in smernic. V drugem delu diplomske naloge bomo z modeliranjem realne problemske domene s pomočjo orodja EMF poskušali priti do odgovora na prej zastavljeno vprašanje praktičnosti. Zaključili bomo s pregledom naših ugotovitev in smernic za nadaljno preučevanje področja.

1.1 Akronimi MD*

Obstaja veliko akronimov s področja modelno vodenega sveta. Vsak predstavlja svojo različico pristopa k modeliranju. Vse različne pristope se pogosto označuje kar z MD* (ang. Model Driven Star) [2]. Slika 1.1 predstavlja relacije med temi različnimi pristopi.

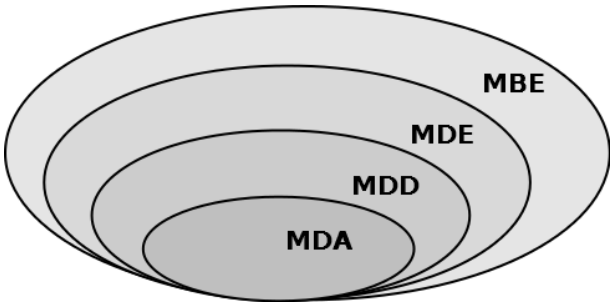
MDD je razvojna paradigma, ki uporablja modele kot glavne izdelke razvojnega procesa. Z MDD je končna rešitev ponavadi (pol)avtomatsko izdelana iz modelov.

MDA je konkretna vizija MDD, predlagana s strani organizacije OMG (ang. Object Management Group - OMG) [17], ki temelji na uporabi OMG standardov. Zato lahko gledamo na MDA kot na podmnožico MDD, kjer so modelirni in transformacijski jeziki standardizirani iz strani OMG.

Po drugi strani je modelno vodeno inženirstvo (ang. Model Driven Engineering - MDE) [2] nadmnožica MDD, ker gre MDE nad čiste razvojne aktivnosti in vključuje druge modelirne naloge s celotnega inženirskega procesa razvoja programske opreme (npr. modelno vodeno obratno inženirstvo podedovanih sistemov). V MDE spadajo tudi vse izpeljanke tipa MD*E [2]:

- MDSE - Model Driven Software Engineering,
- MDPE - Model Driven Product Engineering.

Modelno osnovano inženirstvo (ang. Model Based Engineering - MBE) [2] se nanaša na ohlapnejšo različico MDE. V MBE imajo programski modeli pomembno vlogo, vendar ne nujno kot ključni razvojni produkt. Modeli ne usmerjajo procesa. Primer bi bil razvojni proces, kjer se po fazi analize model sistema preda programerjem kot načrt za ročno kodiranje. V tem primeru modeli še vedno igrajo pomembno vlogo, vendar so manj popolni kot modeli v MDD. Vsi modelno vodeni procesi so modelno osnovani, ne velja pa tudi obratno.

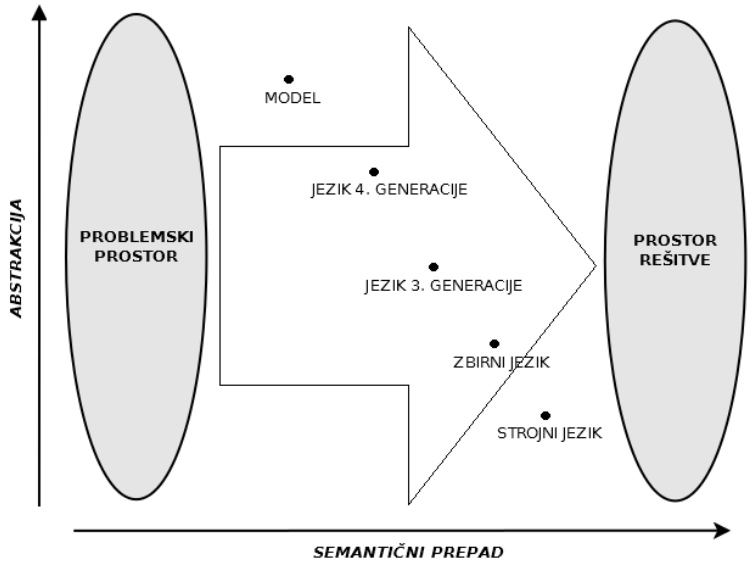


Slika 1.1: Relacije med različnimi MD* akronimi

1.2 Dvig nivoja abstrakcije

Zgodovina razvoja programske opreme je zgodovina dvigovanja nivoja abstrakcije in s tem zmanjševanja semantičnega prepada med problemskim pro-

storom in prostorom rešitve. Slika 1.2 prikazuje dvigovanje nivoja abstrakcije skozi razvoj programskih jezikov.



Slika 1.2: Dvig abstrakcije in zmanjševanje semantičnega prepada

Sprva so bili računalniki samo stroji za računanje, ki so podatke in navodila prejemali preko stikal v obliki 0 in 1. Obseg uporabe teh prvih strojev je bil omejen tako s stališča zmožnosti, ki so ga stroji ponujali (hitro računanje), kot tudi načina, kako so se stroju podajala navodila in podatki za računanje (programiranje).

Z razvojem zbirnikov (ang. assemblers) je bil uveden zbirni jezik, ki je z uporabo mnemonikov, specifičnih za vsako okolje strojne opreme, nadomestil dolga zaporedja 0 in 1 strojnega jezika.

Sčasoma se je s skrivanjem podrobnosti strojnega jezika dvignil nivo abstrakcije programskih jezikov in pripeljal do višjenivojskih programskih jezikov. Nadaljni razvoj je programske jezike približeval človeškim jezikom in tako pripeljal do programskih jezikov četrte generecije, ki omogočajo programerju komunikacijo s pomočjo abstraktnih pojmov na način, ki je primerljiv z načinom razmišljanja ljudi, ko rešujejo nek problem.

Vzporedno z razvojem programskih jezikov so se razvijala orodja, ki

omogočajo avtomatsko preslikavo iz enega nivoja abstrakcije v drugega. Orodji, kot sta zbirnik (ang. assembler) in prevajalnik (ang. compiler), omogočata programerjem, da pišejo v višjenivojskih programskih jezikih, ne da bi ob tem poznali ali se zavedali podrobnosti z nižjenivojskimi jeziki, v katere jih ta orodja preslikajo.

Naslednji nivo abstrakcije je premik v modelno voden razvoj, kjer razvijalci gradijo modele, ki so neodvisni od okolja programske opreme (ang. software-platform-independent). Neodvisnost od okolja programske opreme je analogna neodvisnosti od okolja strojne opreme (ang. hardware-platform independent). Od okolja strojne opreme neodvisni programski jeziki (kot so C# ali Java), omogočajo pisanje specifikacij, ki se lahko brez sprememb izvedejo na različnih okoljih strojne opreme. Podobno omogočajo jeziki, ki so neodvisni od okolja programske opreme, pisanje specifikacij, ki se lahko izvedejo na različnih okoljih programske opreme (Java [14], .Net [8], CORBA [9]).

Poglavje 2

Modelno vodena arhitektura

Modelno vodena arhitektura (ang. Model Driven Architecture - MDA) je standard predlagan iz strani organizacije OMG. MDA predstavlja pristop k specifikaciji informacijskega sistema, ki temelji na ločevanju med specifikacijo funkcionalnosti in specifikacijo izvedbe te funkcionalnosti na specifični tehnološki platformi [20]. MDA definira arhitekturo modelov z naborom smernic za strukturiranje specifikacij, ki so predstavljene kot modeli.

Pristop MDA in podpirajoči standardi omogočajo, da je isti model izdelan v različnih računalniških okoljih. To dosežemo z definiranjem različnih tipov modelov, ki predstavljajo sistem na različnih nivojih abstrakcije. Transformacije med modeli pa so ena glavnih lastnosti MDA.

V splošnem MDA predstavlja pristop k razvoju programske opreme, v katerem so modeli uporabljeni kot glavni vir za analiziranje, dokumentiranje, načrtovanje, izgradnjo, nameščanje in vzdrževanje sistema.

2.1 Model

Model je poenostavitev nečesa na tak način, da lahko opazujemo, upravljamo in razpravljamo o tem ter tako dosežemo razumevanje pripadajoče zapletenosti v stvari, ki jo proučujemo [6].

Glede na uporabo modela lahko opredelimo vsaj tri različne pomene mo-

dela [3]:

- **model kot skica**, ki predstavlja na košček papirja narisanih nekaj črt, ki nakazujejo fizikalne sile, in nekaj enačb, ki opisujejo, kako sile med seboj učinkujejo. Skica ni natančna ali celovita. Namen skice je preverjanje ali predstavljanje ideje. Skica se ne vdržuje ali dostavlja.

- **model kot načrt**, na primer fizični model ladje v bazenu. Bolj pogosto si pod načrtom predstavljamo dokument, ki predstavlja načrt za izgradnjo prave stvari.

- **izvršljivi model**, na primer, če bi model ladje v bazenu iz prejšnje točke lahko preoblikovali v pravo fizično ladjo. Kadar imamo model programske opreme, ki se lahko prevede in izvaja v okviru funkcionalnih in nefunkcionalnih zahtev, ni potrebe po razlikovanju med modelom in pravo stvarjo. To je posebnost programske opreme.

Uporaba modelov je v inženirskih panogah prisotna že dolgo. Inženir zgradi model proučevane stvari z uporabo postopkov abstrakcije, razvrščanja in posplošitve nad problemsko domeno. Zgrajen model tako omogoča cenejše proučevanje in razmišljanje o problemski domeni. Seveda pod predpostavko, da je ceneje zgraditi model kot pravo stvar. O dobrih modelih bi lahko povzeli naslednje lastnosti [6]:

- izpuščajo informacijo, da lahko gledalci vidijo problematiko bolj celostno in preverijo pravilnost tako zgrajene predstave. Dober model tako ne bo v vseh pogledih enak kot modelirana stvar. Lahko recimo modeliramo ladjo, da razumemo, kako bo plula, izpustimo pa podrobnosti o notranji opremi ladje.

- pravilno predstavljajo pravo, abstraktno ali hipotetično realnost. Kljub temu, da model izpušča informacijo, preostala informacija pravilno povzema predmet proučevanja. Če pogledamo na primeru ladje, čeprav postavitev notranje opreme morda ni pomebna za razumevanje hidrodinamičnih lastnosti ladje, lahko njena teža vpliva na hidrodinamiko.

- se morajo zgraditi ceneje kot prava stvar. Ceneje ne pomeni samo v finančnem smislu. Model ladje lahko zgradimo preden zgradimo pravo ladjo z namenom, da preverimo njeno obnašanje na vodi. Na pravi ladji bodo namreč ljudje, katerih življenje bo odvisno od pravilne izgradnje ladje.

- služijo kot sredstvo komunikacije. Dober model predstavi idejo hitro in enostavno. Standarden, dobro opredeljen modelirni jezik zmanjšuje možnost napačnega razumevanja modela s strani gledalca, vključno z računalniki.

2.2 Metamodel

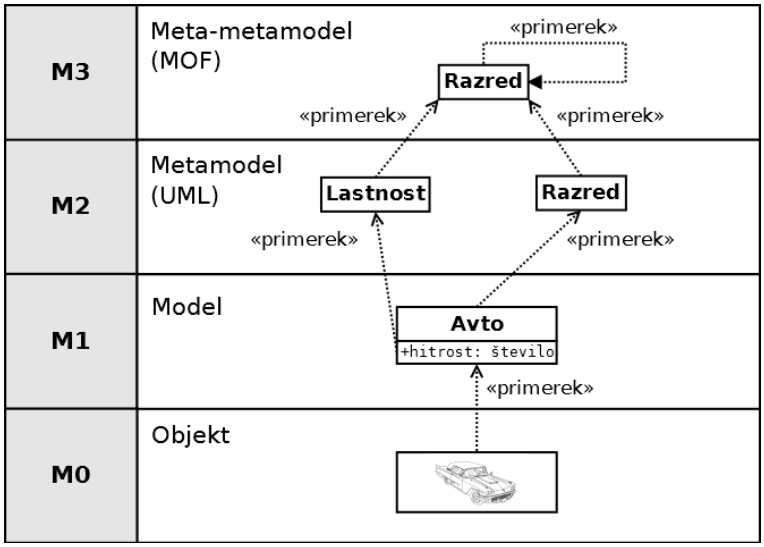
Modeli morajo biti izraženi na način, ki vsem udeleženim strankam zagotavlja splošno razumevanje modela na način, ki točno opredeli, kaj model predstavlja oziroma pomeni. Slednje omogočajo metamodeli. Ti opredelijo koncepte jezika, s katerimi je definiran model.

Metamodel je rezultat uporabe postopkov abstrakcije, razvrščanja in posplošitve nad problemsko domeno modelirnega jezika. Metamodel je pravzaprav model modelirnega jezika.

Za zgrajen model lahko zgradimo model, ki opisuje model (metamodel) in nadalje model, ki opisuje metamodel (meta-metamodel). Čeprav bi lahko v teoriji definirali neskončno nivojev metamodelov, se je v praksi pokazalo, da se meta-metamodel lahko definira z uporabo samega sebe in zato ni smiselno nadaljevati nad tem nivojem abstrakcije. MDA standardizira in poimenuje to kot štiri nivojsko arhitekturo modelov [4], ki so prikazani na sliki 2.1.

Nivoje modelov MDA poimenuje z M3, M2, M1 in M0:

- **M3** predstavlja koncepte, s pomočjo katerih je definiran metamodel na nivoju M2. Opisuje lastnosti, ki jih metamodeli lahko izrazijo. To je nivo, na katerem delujejo modelirni jeziki in metamodeli, in zagotavlja prenosljivost med orodji.



Slika 2.1: Štiri nivojska MDA arhitektura modelov

- **M2** vsebuje metamodele, ki so sestavljeni iz primerkov elementov M3 nivoja. To je nivo, na katerem delujejo orodja.
- **M1** vsebuje modele sestavljene iz primerkov elementov M2 nivoja. To je nivo, na katerem se odvija modeliranje.
- **M0** vsebuje podatke in objekte realnega sveta, ki so primerki elementov iz M1 nivoja.

2.3 Standard MOF

MOF (ang. Meta Object Facility - MOF) [15] je OMG standard za formalno definiranje modelov in metamodelov, torej jezik za definiranje modelirnih jezikov. MOF je bil razvit na predpostavki, da bo v uporabi več kot en tip modela in torej tudi več modelirnih jezikov. Z uporabo MOF definicije modelirnega jezika lahko definiramo transformacije med različnimi modelirnimi jeziki. Ker so transformacije definirane na nivoju metamodelov modelirnih jezikov, so lahko uporabljene na katerem koli modelu napisanem v enem

izmed teh modelirnih jezikov. MDA pristop bi težko realizirali brez standardnega jezika, ki opisuje metamodele modelirnih jezikov. MOF je tako eden od stebrov MDA pristopa.

S pomočjo MOF so definirani pomembni jeziki s področja modelno vodenega razvoja, kot so:

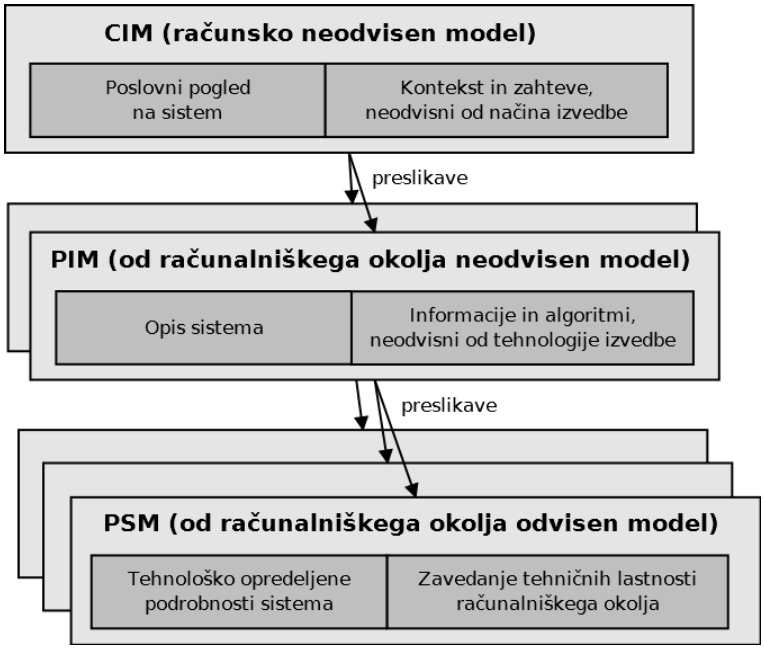
- **UML** splošno namenski modelirni jezik (ang. Unified Modeling Language),
- **QVT** množica jezikov za definiranje poizvedb, pogledov in preslikav modelov (ang. Query View Transformation),
- **XMI** jezik za izmenjavo metapodatkov v XML obliki (ang. XML Metada Interchange),
- **CWM** jezik za modeliranje metapodatkov podatkovnih skladišč (ang. Common Warehouse Metamodel).

2.4 Tipi modelov

MDA definira dva nivoja modeliranja:

- *poslovni nivo modeliranja*, ki ga predstavlja računsko neodvisen model (ang. Computational Independent Model - CIM) in
- *sistemki nivo modeliranja*, ki ga predstavljata od računalniškega okolja neodvisen model (ang. Platform Independent Model - PIM) in od računalniškega okolja odvisen model (ang. Platform Specific Model - PSM), kot prikazuje slika 2.2.

Model CIM se pogosto imenuje tudi poslovni model ali domenski model. Na najvišjem abstraktnem nivoju opisuje zahteve sistema in poslovne okoliščine, v katerih bo sistem uporabljen. Model tipično opisuje namen sistema in ne načina, kako bo sistem narejen. Lahko opisuje tudi vidike sistema, ki ne bodo nujno računalniško podprti. Model CIM igra pomembno vlogo pri



Slika 2.2: Nivoji modeliranja v MDA

zmanjševanju prepada, ki običajno nastopa med domenskimi strokovnjaki in informacijskimi tehnologi, ki so odgovorni za izgradnjo sistema.

Model PIM opisuje strukturo in obnašanje sistema neodvisno od računalniškega okolja, ki je uporabljeno za izgradnjo sistema. Model PIM definira samo del sistema, ki bo računalniško podprt in to na način, da izkazuje stopnjo neodvisnosti, ki omogoča preslikavo v več konkretnih računalniških okoljih.

Model PSM mora vsebovati vse informacije, ki so vezane na strukturo in obnašanje sistema in so potrebne za izgradnjo sistema ali izvajanje modela na konkretnem računalniškem okolju.

2.5 Transformacije modelov

Transformacije med modeli predstavljajo enega ključnih MDA konceptov. Transformacije med modeli na različnih nivojih abstrakcije so definirane z množico preslikovalnih pravil, ki skupaj opisujejo, kako se model iz izvornega

nivoja preslika v ciljni nivo. Tipično se model CIM lahko preslika v različne modele PIM, ki se lahko naprej posamezno preslikajo v različne modele PSM, ti pa na koncu v konkretno izvedbeno kodo.

Transformacije so definirane na nivoju metamodela in uporabljene na nivoju modela. Transformacije so izvedene med izvornim in ciljnim modelom, definirane pa nad pripadajočim metamodelom. To omogoča večkratno uporabo transformacij nad vsemi modeli, ki so v skladu z istim metamodelom. MDA načelo 'vse je model', ki je uporabljeno pri transformacijah, pripelje do definiranja transformacij v obliki modela, kar omogoča upravljanje z njihovim modelom, vključno z njihovim metamodelom.

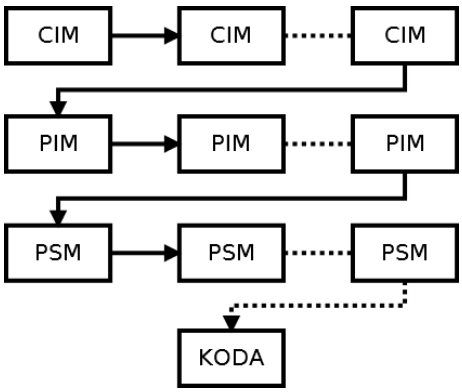
Za podporo avtomatskim transformacijam morajo biti pravila definirana v obliki, ki je primerna za računalnik. To zahteva formalizem, s pomočjo katerega so pravila izražena. OMG zato definira standard QVT (ang. Query, Views and Transformations), ki definira tri jezike za transformiranje modelov [21].

2.6 Razvojni proces

Razvojni proces se začne z definiranjem modela CIM, ki ga razvijejo poslovni analitiki v sodelovanju z poslovnimi uporabniki. Načrtovalec poslovnih sistemov (ang. enterprise architect) potem model CIM pretvori v model PIM, pri čemer dodaja načrtovalsko znanje v model CIM brez podajanja podrobnosti računalniškega okolja. Za dokončanje procesa mora specialist za konkretno računalniško okolje pretvoriti model PIM v model PSM. Znanje v vsakem koraku razvojnega procesa dodajajo različni strokovnjaki, glavni izziv procesa pa je transformacija med različnimi modeli.

Med modeli lahko obstajajo velike razlike, za katere ni mogoča direktna transformacija. V tem primeru se lahko ustvari več modelov in s horizontalno transformacijo zmanjša ta razlika, kot ponazarja slika 2.3. Ločnica med različnimi novoji abstrakcije se tako lahko zabriše. Model PIM se na primer lahko v vsakem koraku obogati z informacijami o računalniškem okolju, kar

na koncu privede do modela PSM. Model PSM pa se lahko transformira v bolj podrobne modele PSM, dokler ni dosežen nivo, iz katerega je mogoča izvedba sistema.



Slika 2.3: MDA razvojni proces

Širše gledano se razvijalci ukvarjajo z dvema glavnima aktivnostima. V prvi se formalizira znanje o dotični temi v obliki modela. V drugi fazi se za dosego končnega sistema preslika to formalizirano znanje na ciljno računalniško okolje [6].

Formaliziranje znanja nadalje vsebuje štiri aktivnosti:

- pridobivanje zahtev o željeni temi,
- abstrakcija tako pridobljenega znanja v množico konceptov,
- izražanje teh abstraktnih konceptov v obliki formalnih modelov ter
- preverjanje pravilnosti modelov (idealno z izvajanjem).

Povezovanje modelov prav tako vsebuje štiri aktivnosti:

- določanje preslikovalnih funkcij,
- označevanje modelov,
- preverjanje pravilnosti preslikav in

- transformacij modelov.

MDA predvideva, da formaliziran model za ciljno računalniško okolje že obstaja. V nasprotnem primeru je potrebno fazo formaliziranja znanja opraviti tudi za to področje.

Čeprav lahko zgradimo model česarkoli, je za učinkovitost procesa pomembno, da modele razvijamo po namembnosti področja. Kadar imamo v problemski domeni tudi področje varnosti, je primerneje to področje modelirati neodvisno. Tako je za uporabo takega modela v drugih sistemih potrebna samo faza povezovanja modelov na istem nivoju abstrakcije. Za zamenjavo modela za varnost je tako potrebna samo nova definicija povezovanja modelov.

2.7 MDA in agilne metode

V zadnjih letih so nekatere načrtovalske in razvojne metodologije pridobile veliko pozornosti in podpornikov. Eden od takih je agilni pristop k razvoju programske opreme. Primer takega pristopa je ekstremno programiranje (ang. Extreme Programming - XP) [1].

Na prvi pogled veliko ljudi domneva, da MDA in XP zavzemata nasprotni stališči v razvoju programske opreme. Ta domneva sloni na predpostavki, da so modeli samo načrtovalski izdelki in dejstvu, da XP posveča več pozornosti razvoju kot načrtovanju izdelka. Toda formalni modeli, ki poganjajo generatorje in virtualne stroje, so razvojni izdelek, ki dobro sovpada z XP pogledom. Na primer programiranje v paru se lahko prenese na ustvarjanje formalnih modelov. MDA je tudi usklajen z XP pri zavračanju slapovnega razvojnega procesa, saj predpostavlja iterativni razvoj. XP tudi zagovarja odlašanje z načrtovalskimi odločitvami kolikor dolgo je možno, saj se pogoji, ki vplivajo na načrt hitro spreminjajo. MDA na nek način ustreza tudi temu pristopu, saj načrtovanje komponente sistema z abstrahiranjem izvedbe na nek način odloži podrobne izvedbene odločitve.

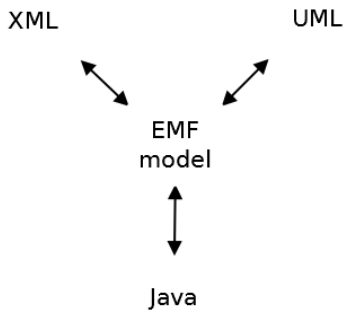
Poglavje 3

Orodje Eclipse Modeling Framework

Orodje Eclipse Modeling Framework (EMF) je konkretna izvedba MDA standardov, pristopov in konceptov v odprtokodnem razvojnem okolju Eclipse. Orodje EMF predstavlja jedro Eclipse projekta imenovanega Eclipse Modeling Project, katerega glavni namen je promovirati in razvijati modelno vodene tehnike razvoja programske opreme znotraj okolja Eclipse. Poleg orodja EMF so del projekta Eclipse Modeling Project tudi rešitve, ki omogočajo transformacije med modeli, podatkovne integracije in izdelovanje grafičnih vmesnikov za manipulacijo modelov. Projekt vključuje tudi izvedbo nekaj pomembnih OMG standardov, kot so MOF, UML, QVT in XMI.

Slika 3.1 prikazuje, kako orodje EMF združuje jezike Java, XML in UML. Pri tem osrednjo vlogo igra model kot vhod v razvojni proces in usmerjevalec izdelave kode. Model je lahko definiran v kateri koli od omenjenih tehnologij in je skupni visokonivojski predstavnik iz katerega lahko izdelamo ostale oblike tehnologij. Orodje EMF tudi združuje pojma modeliranje in programiranje. Namesto jasne ločnice med visokonivojskim inženirskim delom in modeliranjem ter nizkonivojskim izvedbenim programiranjem, orodje EMF združi oboje v dobro integrirano celoto istega postopka.

Orodje EMF sestavljajo trije osnovni deli [12]:



Slika 3.1: EMF združuje Javo, XML in UML

- **EMF jedro** vključuje metamodel imenovan Ecore [7] za opisovanje modelov, izvajalno (ang. runtime) podporo modelom, vključno z obveščanjem o spremembah, shranjevanju v XMI obliki ter odsevnim programskim vmesnikom (ang. reflective API) za generično manipulacijo EMF objektov.

- **EMF.Edit** vključuje generične razrede za izdelavo urejevalnikov EMF modelov. Zagotavlja vsebinske in predmetne ponudnike (ang. content and item providers) in druge pomožne razrede, ki omogočajo prikaz EMF modelov z uporabo standardnih namiznih (JFace) pogledov in listov z lastnostmi (ang. property sheets). Vsebuje tudi ukazno ogrodje (ang. command framework) z generičnimi izvedbenimi razredi (ang. implementation classes) za izdelavo urejevalnikov s podporo tehniki razveljavi/uveljavi.

- **EMF.Codegen** zagotavlja vso potrebno podporo za izdelavo urejevalnika EMF modelov. Vključuje grafični vmesnik za izvajanje in podajanje parametrov izdelave. EMF.Codegen znižuje vstopni prag za delo z Java Development Tools komponento orodja Eclipse.

Orodje EMF podpira samo del opisne moči jezika UML, konkretno samo razredne diagrame. Po mnenju samih snovalec predstavlja orodje srednjo pot

med dvema skrajnima pogledoma na modeliranje: "ne potrebujemo modelov"na eni strani in "vse je model"na drugi strani.

S tem so zavzeli bolj pragmatično stališče in izdelali orodje, ki je glede na trenutno stanje razvoja modelno vodenega pristopa pri razvoju programske opreme pravšnja mešanica enega in drugega pogleda na modeliranje.

3.1 Metamodel Ecore

Razvoj orodja EMF se je začel z izvedbo standarda MOF in se s časoma razvil v izvedbo podmnožice modela MOF imenovanega EMOF (ang. Essential MOF) [12]. V izogib zmedi in dejstvu, da so manjše razlike med EMF izvedbo EMOF-a in OMG-jevo definicijo EMOF-a, so model poimenovali Ecore. Ecore je torej model, ki je uporabljen za predstavitev modelov v orodju EMF in je hkrati tudi sam EMF model, kar ga postavlja na M3 nivo MDA štirinovojske arhitekture metamodelov.

Glavni Ecore gradniki za opis modelov v orodju EMF so [7]:

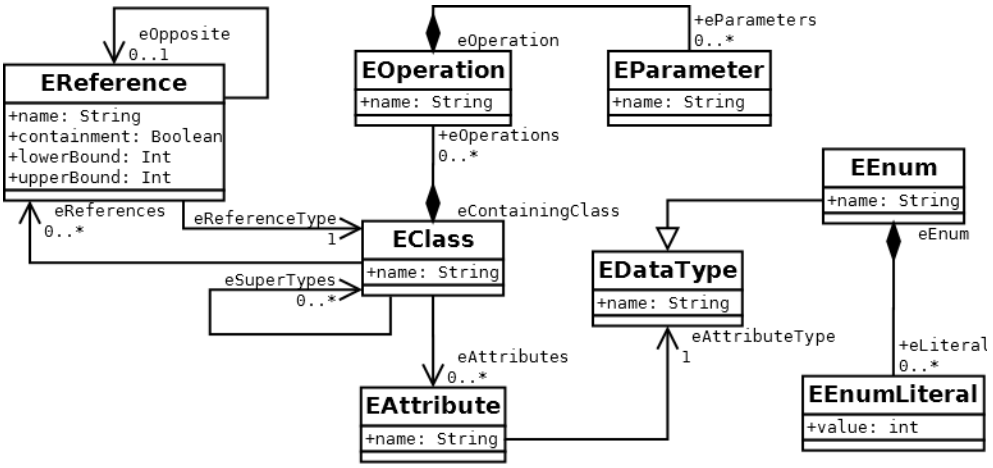
- **EClass** se uporablja za predstavitev modeliranih razredov. Zajema ime, nič ali več atributov in nič ali več povezav. Za podporo dedovanju se razred lahko sklicuje na druge razrede kot na njegove nadtipe (ang. supertype).

- **EAttribute** se uporablja za predstavitev modeliranih atributov. Atribut zajema ime in tip.

- **EReference** se uporablja za predstavitev povezave med razredoma. Zajema ime, logično zastavico, ki opredeljuje ali predstavlja vsebovanost in tip povezave, ki je drug razred. Opredeljeno ima tudi številnost relacije.

- **EDataType** se uporablja za predstavitev podatkovnih tipov atributov. Podatkovni tip je lahko enostavni (kot `int` ali `string`) ali java tip objekta (kot `java.util.Date`).

- **EEnum** se uporablja kot poseben podatkovnih tip z definirano zalogo vrednosti.
- **EOperation** omogoča definiranje vmesnikov do operacij razreda, vendar ne ponuja konstruktov za definiranje obnašanja teh operacij.



Slika 3.2: Poenostavljen diagram metamodela Ecore

V diagramu metamodela Ecore iz slike 3.2 so izpuščeni elementi, ki za uporabnika v fazi modeliranja niso vidni, so pa pomemben strukturni del, ki omogoča učinkovito izvedbo modela:

- **EModelElement** je osnovni tip za vse elemente v Ecore.
- **ENamedElement** definira osnovni element, iz katerega je izpeljanih večina razredov, da podedujejo atribut `name`.
- **ETypedElement** predstavlja osnovni razred, ki za ostale elemente združuje koncept tipa. Definira tudi številnost v smislu koliko vrednosti tega tipa dovoljuje in ali so vrednosti enolične ter urejene.
- **EStructuralFeature** združuje množico atributov, ki so lastni elementoma `EReference` in `EAttribute`. Ti atributi definirajo, kako so strukturne lastnosti elementov shranjene in na kakšen način so dostopane.

Poleg osnovnih gradnikov in strukturnih elementov je potrebno omeniti še elemente, ki omogočajo koncepte, ki so bližje programskim jezikom:

- **EPackage** združuje povezane razrede in podatkovne tipe in je zelo povezan z Java paketi. Pri serializaciji modelov predstavlja korenski element.
- **EFactory** se uporablja za tvorbo primerkov razredov in vrednosti podatkovnih tipov, ki pripadajo paketu.
- **EAnnotation** predstavlja mehanizem, s katerim lahko model označimo z dodatnimi informacijami. Vir označb `GenModel` omogoča podajanje informacij pomembnih samo za proces generinja kode, kar združeno z `EOperation` omogoča definiranje dejanske programske kode v modelu.

3.2 Izdelava modelov

Orodje EMF omogoča izdelavo Ecore modelov na podlagi:

- javanske kode,
- XML shem in
- UML razrednih diagramov.

Javanska koda predstavlja najbolj neposreden način izdelave Ecore modelov, ki zahteva najmanj vložka. Za razvijalce, ki se bolj spoznajo na programiranje kot na modeliranje, je to morda najlažji način. Zahteva pa orodje EMF, da se javanska koda označi s posebej oblikovanimi komentarji, ki označijo elemente modela in opcijsko omogočajo opredelitev informacije, ki ni direktno opisljiva z javansko kodo. Komentarji predznačijo element modela v obliki *Javadoc* komentarjev označenih z značko `@model`, kot prikazuje sintaksa:

```
/**
 * @model [lastnost='vrednost' | lastnost='vrednost'] ...
 */
```

XML sheme ponujajo pri izdelavi Ecore modelov z vidika modeliranja manjšo izrazno moč, kot jo ima Ecore. Tako recimo, ne omogoča definiranja dvosmernih povezav ali tipa elementa, na katerega kaže povezava. EMF to reši z vpeljavo XML razširitve v obliki atributov iz imenskega prostora `Ecore`[1], ki se lahko uporabijo za podajanje manjkajočih informacij. Po drugi strani je s pomočjo XML sheme možno definirati veliko podrobnosti serializacije, ki niso predstavljive v Ecore. Način za rešitev neskladja je uporaba elementa `EAnnotations`, ki omogoča podajanje informacij v modelih, za katere Ecore nima podpore.

UML diagrami ponujajo pri izdelavi Ecore modelov tri možnosti:

- Izdelavo Ecore modelov z uporabo EMF vizualnih orodij.
- Uvoz iz UML. Orodje EMF ponuja razširljivo ogrodje, v katerega se lahko vključijo vtičniki za uvoz modelov, ki so zapisani v različnih oblikah. Orodje EMF v osnovi podpira obliko za Rational Rose (`.mdl` datoteke).
- Izvoz iz UML. Podobno kot predhodna možnost, le da preslikavo podpirajo sama UML orodja.

Izdelovanje Ecore modelov z uporabo XML shem ima svoje mesto v scenarijih podatkovne integracije. Uporaba javanske kode je najbolj neposreden način, s stališča modelno usmerjenega razvoja pa je uporaba UML diagramov najbolj priporočljiv način za izdelovanje Ecore modelov.

Za ponazoritev različnih načinov izdelave Ecore modelov poglejmo enostaven primer modela osebe in njenih hobijev izdelan v:

- **javanski kodi:**

[1] http://www.eclipse.org/emf/2002/ecore

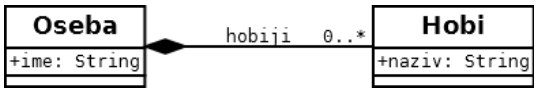
```
/**
 * @model
 */
public interface Oseba {
   /**
    * @model
    */
   String getIme();
   /**
    * @model type="Hobi" containment="true"
    */
   List getHobiji();
}

/**
 * @model
 */
public interface Hobi {
   /**
    * @model
    */
   String getNaziv();
}
```

- **XML shemi:**

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.example.org/Oseba"
   xmlns:tns="http://www.example.org/Oseba">
   <complexType name="Oseba">
      <sequence>
         <element name="ime" type="string"/>
         <element name="hobiji" type="tns:Hobi"
```

```
            minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
      </complexType>
      <complexType name="Hobi">
        <sequence>
          <element name="naziv" type="string"/>
        </sequence>
      </complexType>
    </schema>
```

- **UML diagramu:**



Ne glede na načini izdelave Ecore modela se v EMF izdelata dva modela:

- **.ecore** z informacijami o modelu in
- **.genmodel** z dodatnimi informacijami o modelu, ki služijo za generiranje kode.

Ideja generatorskega modela (.genmodel) je, da ostane Ecore model (.ecore) čist in neodvisen od informacij, ki so potrebne samo za generiranje kode. Z vidika MDA lahko torej na Ecore model gledamo kot na model PIM in na generatorski model kot na model PSM. V primeru sprememb na Ecore modelu orodje EMF zagotovi, da sta oba modela usklajena. Orodje tudi zagotovi, da je generatorski model pripravljen s privzetimi vrednostmi, ki že omogočajo generiranje kode.

Za potrebe trajnega shranjevanja orodje EMF serializira modele Ecore v obliko XMI. Ta četrta standardizirana oblika predstavitve Ecore modela omogoča tudi izmenjavo modelov med različnimi orodji. XMI oblika datoteke vsebuje samo informacije o elementih modela, grafična informacija modela

pa se izgubi (pozicija, barva, razporeditev, pisava, ...). To slabost naj bi odpravil nov OMG standard imenovan DI (ang. Diagram Interchange) [19], ki pa še ni doživel široke podpore v modelirnih orodjih.

Na primeru modela osebe in njenih hobijev izgleda XMI serializacija modela takole:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
   name="oseba" nsURI="http://www.example.org/oseba"
   nsPrefix="oseba">
   <eClassifiers xsi:type="ecore:EClass" name="Oseba">
      <eStructuralFeatures xsi:type="ecore:EAttribute"
         name="ime" eType="ecore:EDataType
         http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EReference"
         name="hobiji" upperBound="-1"
         eType="#//Hobi" containment="true"/>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass" name="Hobi">
      <eStructuralFeatures xsi:type="ecore:EAttribute"
         name="naziv" eType="ecore:EDataType
         http://www.eclipse.org/emf/2002/Ecore#//EString"/>
   </eClassifiers>
</ecore:EPackage>
```

Orodje EMF uporablja Ecore model v dveh različnih okoliščinah. Med razvojem aplikacije je Ecore model glavni vir za generator kode. Med izvajanjem aplikacije pa Ecore model uporablja orodje EMF za ugotavljanje pravilnega obnašanja modela.

3.3 Generiranje kode

Poleg pristopa k reševanju problema na višjem nivoju je glavna prednost uporabe orodja EMF avtomatsko generiranje kode. Generirana koda vsebuje veliko načrtovalskih vzorcev (ang. design patterns) [5]. Načrtovalski vzorci predstavljajo najboljšo znano izvedbo določenega problema, kar govori v prid kvaliteti generirane kode.

Na osnovi generatorskega modela se lahko generira javanska koda na treh nivojih:

- na nivoju modela,
- na nivoju prilagojevalnika in
- na nivoju urejevalnika.

Nivo modela vključuje izvedbo modela v javanski kodi z učinkovito izvedbo kreiranja, spreminjanja, shranjevanja in branja primerkov razredov modela.

Za vsak razred v modelu se izdela vmesnik in istoimenski izvedbeni razred s pripadajočimi `get/set` metodami za atribute ter izvedbami povezav med razredi. Za izvedbo povezav je uporabljen načrtovalski vzorec lenega nalaganja (ang. lazy loading design pattern) [5], ki odloži nalaganje povezanega razreda, dokler ga zares ne potrebujemo.

Za izdelavo primerkov razredov se izdela tudi tovarna (ang. factory) ter paket, ki med izvajanjem (ang. runtime) zgradi Ecore model z zagotavljanjem učinkovitega dostopa do elementov modela.

Vsi generirani razredi podpirajo tudi načrtovalski vzorec opazovalca (ang. observer design pattern), kar omogoča prijavljenim objektom spremljanje sprememb v dotičnih objektih. Ta pomembna lastnost je nujno potrebna za učinkovito predstavljanje modela na interaktivnih uporabniških vmesnikih.

Generirani razredi omogočajo tudi trajno shranjevanje primerka modela. Za učinkovito in razširljivo izvedbo te lastnosti se uporabljata načrtovalska vzorca namestnik (ang. proxy design pattern) in leno nalaganje [5].

Posebna lastnost generirane kode je tudi zmožnost uporabe odsevnega vmesnika (ang. reflective API) za generično ravnanje z EMF objekti, kar omogoča uporabo objektov modela brez generirane kode.

Izsek generirane kode na primeru modela osebe in njenih hobijev:

```
/**
 * @generated
 */
public class OsebaImpl
   extends MinimalEObjectImpl.Container
   implements Oseba {
   ...
   /**
    * @generated
    */
   public void setIme(String newIme) {
      String oldIme = ime;
      ime = newIme;
      if (eNotificationRequired())
         eNotify(new ENotificationImpl(
            this,
            Notification.SET,
            OsebaPackage.OSEBA__IME,
            oldIme,
            ime));
   }
}
```

Nivo prilagojevalnika vključuje izvedbo razredov, ki prilagajajo razrede z nivoja modela za potrebe urejanja in prikaza.

Glavni načrtovalski vzorec tega nivoja je prilagojevalnik (ang. adapter design pattern) [5]. Generirana koda predstavlja začetek interaktivnega uporabniškega vmesnika in je od njega neodvisna.

Poleg ponudnikov predmetov za vsebino (ang. content item providers) in oznako (ang. label item providers), ki omogočajo osnovo izgradnje interaktivnega vmesnika, koda na tem nivoju podpira tudi ukazno osnovano urejanje EMF objektov, vključno s podporo za funkcionalnost razveljavi/uveljavi (ang. undo/redo). Za učinkovito in razširljivo kodo generator uporablja načrtovalska vzorca ukaz (ang. command design pattern) in predloga (ang. template method design pattern) [5].

Izsek generirane kode na primeru modela osebe in njenih hobijev:

```
/**
 * @generated
 */
public class OsebaItemProvider
   extends ItemProviderAdapter
   implements
      IEditingDomainItemProvider,
      IStructuredItemContentProvider,
      ITreeItemContentProvider,
      IItemLabelProvider,
      IItemPropertySource {
    ...
}
```

Nivo urejevalnika vključuje kodo, ki združuje kodo z nivoja pregledovalnika, s kodo Eclipse uporabniškega vmesnika.

Koda je odvisna od uporabniškega vmesnika, saj uporablja gradnike ogrodja Eclipse, kot sta JFace [11] in SWT [31]. Generirana koda predstavlja delujoč urejevalnik primerkov Ecore modela. Urejevalnik je privzeto narejen kot vtičnik za razvojno okolje Eclipse s funkcionalnostjo različnih pogledov na primerke modela in funkcionalnostjo dodajanja, brisanja, kopiranja, premikanja in urejanja primerkov modelov s podporo za razveljavi/uveljavi.

Izsek generirane kode na primeru modela osebe in njenih hobijev:

```
/**
 * @generated
 */
public class OsebaEditor
   extends MultiPageEditorPart
   implements
      IEditingDomainProvider,
      ISelectionProvider,
      IMenuListener,
      IViewerProvider,
      IGotoMarker {
   ...
}
```

Poleg javanske kode generator pripravi tudi nastavitvene datoteke, ki omogočajo uporabo generirane kode na način kot vtičniki za Eclipse. Od definicije v generatorskem modelu je odvisno število vtičnikov, v katere bo generirana koda vstavljena. Vsak nivo generirane kode je lahko v samostojnem vtičniku ali celo izpuščen iz procesa generiranja.

Generator kode proizvede kodo namenjeno ročnemu dopolnjevanju. Generirana koda je namenjena urejanju generiranih razredov, metod in spremenljivk ali dodajanju novih. Ob ponovnem generiranju se ohrani koda, ki ni predznačena s komentarjem z značko `@generated`. Predlagan vzorec označevanja ročno dodane ali spremenjene kode je zamenjava značke `@generated` z `@generated NOT`.

3.4 Izdelava primerkov modelov

Z izdelavo Ecore modela smo po MDA definiciji štiri nivojske arhitekture iz slike 2.1 ustvarili model na nivoju M1. Naslednji korak je torej izdelava modela na nivoju M0, ki predstavlja primerek modela na nivoju M1. Za izdelavo primerkov modelov orodje EMF omogoča več različnih pristopov, ki

se navezujejo na tri nivoje generirane kode.

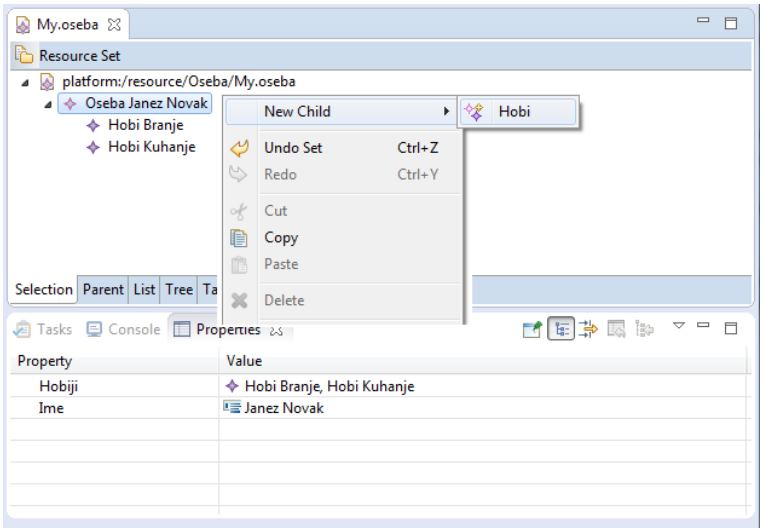
Generirana koda na nivoju modela omogoča izdelavo primerka modela kar v javanski aplikaciji. Programski vmesnik generirane kode je enostaven in razumljiv. Generirano kodo lahko preprosto uporabljajo tudi manj izkušeni programerji. Primer kode za izdelavo primerka modela osebe in njenih hobijev:

```
Oseba oseba = OsebaFactory.eINSTANCE.createOseba();
oseba.setIme("Janez Novak");
Hobi hobi = OsebaFactory.eINSTANCE.createHobi();
hobi.setNaziv("Branje");
oseba.getHobiji().add(hobi);
```

Generirana koda na nivoju prilagojevalnika predstavlja dobro izhodišče za izdelavo aplikacije z interaktivnim uporabniškim vmesnikom. Problemi, ki se pojavljajo pri vsaki taki aplikaciji, so v generirani kodi že učinkovito rešeni. Funkcionalnost prikaza in ukazno osnovano urejanje primerkov modela s podporo za razveljavi/uveljavi je tako s pomočjo generirane kode hitreje zgrajena v katerikoli javanski aplikaciji. Za razliko od enostavnega progamskega vmesnika, ki ga ponuja koda na nivoju modela, pa uporaba te kode ni mogoča brez podrobnega poznavanja in razumevanja kode orodja EMF in programskega vmesnika, ki ga ponuja.

Generirana koda na nivoju urejevalnika uporablja največ moči generirane kode. Omogoča izdelavo primerkov modela brez dodatnega programiranja, saj predstavlja generirana koda delujočo aplikacijo. Slika 3.3 prikazuje uporabniški vmesnik kot ga ponuja generirana koda na primeru modela osebe in njenih hobijev. Osnovni izgled in funkcionalnost tako generiranega uporabniškega vmesnika je možno prilagoditi in nadgraditi za različne potrebe, kar zahteva dobro poznavanje in razumevanje tako kode orodja EMF kot tudi ogrodja Eclipse. Z izjemo najbolj enostavnih prilagoditev to zahteva strmo

krivuljo učenja.



Slika 3.3: Ustvarjanje primerkov modela v EMF.Editor

3.5 Shranjevanje primerkov modelov

Orodje EMF vsebuje močno ogrodje za shranjevanje modelov. Osnovni način shranjevanja primerkov modelov je v obliki XMI. Gre za generično rešitev, ki omogoča shranjevanje katerega koli EMF modela, vključno s samim Ecore modelom. Primer serializirane oblike Ecore modela v obliki XMI smo že predstavili v poglavju 3.2. Ogrodje samo je razširljivo in omogoča transparentno povezovanje objektov shranjenih v poljubni obliki. Osnovna enota shranjevanja je vir (ang. resource), ki združuje vse objekte, ki naj bi bili shranjeni skupaj, vključujoč tudi vsebovane objekte. Viri se združujejo v množice virov (ang. resource set). Za identifikator vira, ki omogoča tudi povezovanje objektov iz različnih virov, se uporablja standard URI (ang. Unified Resource Identifier).

Na primeru modela osebe in njenih hobijev izgleda programska koda, ki shrani primerek modela v datoteko, takole:

```
ResourceSet resourceSet = new ResourceSetImpl();
URI fileUri = URI.createFileURI(
    new File("oseba.xml").getAbsolutePath());
Resource oResource = resourceSet.createResource(fileUri);
oResource.getContents().add(oseba);
oResource.save(null);
```

Serializirana oblika istega primerka modela v XMI obliki izgleda takole:

```
<?xml version="1.0" encoding="UTF-8"?>
<oseba:Oseba xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
   xmlns:oseba="http://www.example.org/oseba"
   ime="Janez Novak">
      <hobiji naziv="Branje"/>
      <hobiji naziv="Kuhanje"/>
</oseba:Oseba>
```

3.6 Ostale operacije nad modeli

Učinkovito upravljanje modelov skozi njihov celotni življenjski cikel potrebuje poleg ustvarjanja, prenosljivosti in trajnega shranjevanja modelov še nekaj z orodji podprtih področij.

Primerjava modelov je ključna operacija za upravljanje z modeli. Razlike med modeli so potreben vhod v avtomatizacijo upravljanja modelov, kot sta sorazvoj (ang. coevolution) in nadzor različic (ang. source control). EMF Compare [25] je orodje, ki ponuja generično primerjavo in grafično predstavitev razlik med EMF modeli.

Nadzor različic modelov je zelo pomemben del infrastrukture razvoja programske opreme. Omogoča pregled zgodovine razvoja modelov, hkratno delo več razvijalcev in upravljanje različnih razvojnih vej. Tradicionalni besedilno osnovani sistemi nadzora različic obravnavajo modele kot navadno besedilno datoteko, zanemarjajoč naravo modelov, ki temelji na grafih. EMF Store [29]

je orodje, ki ponuja varno shrambo EMF modelov z zmožnostjo razreševanja konfliktov v primeru sočasnega urejanja istega dela modela.

Modeli se po MDA viziji ne uporabljajo samostojno, ampak so medsebojno povezani. To pomeni, da se morajo po razvoju modela razviti tudi odvisni modeli, da bi tako ohranili medsebojno povezavo. EMF Edapt [26] je orodje za migracijo primerkov modelov na novo različico modela.

Kvaliteta modelov je široko področje, ki je pritegnilo veliko pozornosti in razvoja, vendar kljub temu ostaja velik izziv računalniške industrije. V splošnem zagotavlja orodje EMF samo pravilno strukturiranost modela (ang. well-formedness). Orodje lahko preverja, da je model pravilni primerek metamodela na podlagi katerega je izpeljan. Ne zagotavlja pa, da je možno ustvariti primerek modela, kar predstavlja znan problem zadovoljivosti (anf. satisfiability problem). Pravilna strukturiranost modelov je samo delček problema. Nekaj pristopov k testiranju pravilnosti modelov je bilo že predlaganih, noben pa še ni dosegel potrebne zrelosti za poslovno uporabo.

Poglavje 4

Izvedba na primeru

Glede na naravo orodja EMF, ki za modeliranje podpira samo razredne UML diagrame, smo se morali za primer, na katerem bi lahko ovrednotili praktičnost uporabe MDD pristopov, omejiti na podatkovno intenzivne aplikacije s podporo CRUD operacijam (ang. Create Read Update Delete - CRUD) preko grafičnega uporabniškega vmesnika. S primerom smo želeli priti do aplikacije, ki bi uporabniku nudila celovito rešitev za izbrano problemsko domeno. Za primer smo si tako izbrali problemsko domeno definiranja jedilnikov za restavracijo in pripravo jedilnikov v primerni obliki za objavo na spletu.

Za izdelavo rešitve smo uporabili razvojno okolje Eclipse (različica 4.5), imenovano tudi Mars [23]. Poleg osnovnih orodij za delo z Javo razvojno okolje Eclipse vsebuje tudi projekta EMF (različica 2.11) in Acceleo [22] (različica 3.6), ki smo ju uporabili v rešitvi.

Najprej smo s pomočjo orodja EMF zgradili model in generirali javansko kodo, ki smo jo uporabili v urejevalniku jedilnika. Nato smo s pomočjo orodja Acceleo definirali preslikavo modela v besedilno obliko in tako prišli do jedilnika v obliki, ki je primerna za objavo na spletu. Na koncu smo za izdelavo klienta, ki je neodvisen od razvojnega okolja Eclipse, uporabili še Eclipse platformo imenovano Rich Client Platform (RCP).

4.1 Opis problema

Rešitev naj restavraciji omogoča definiranje jedilnikov za obdobje enega tedna in njihovo predstavitev v obliki, ki je primerna za objavo na spletu.

Jedilnik naj se definira samo za delovne dni v tednu. Datumi jedilnika naj se določijo na podlagi zaporedne številke tedna v letu. Vse jedi, ki se pripravljajo v restavraciji, želimo definirati samo enkrat in ta seznam jedi uporabiti pri sestavi dnevnega jedilnika. Vsaka jed naj ima svojo ceno, cena menuja pa se izračuna na podlagi vsebovanih jedi. Vsak menu naj ima naziv, vsebovati pa mora natanko eno glavno jed in opcijsko juho, solato ali sladico. Za vsako jed smo dolžni določiti tudi vsebnost alergenov, ki naj bodo v objavljeni obliki jedilnika nazorno prikazani ob vsaki jedi. Poleg dnevne ponudbe nudimo tudi stalno ponudbo. Omogočeno naj bo ločeno definiranje stalne ponudbe in enostavna vključitev v tedensko ponudbo.

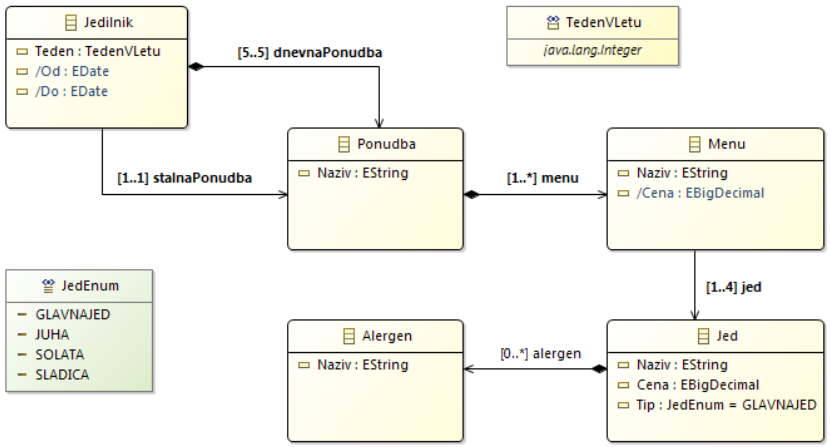
Rešitev naj omogoča trajno hranjenje vseh podatkov jedilnika. Urejanje podatkov naj poleg osnovnih tehnik, kot so izreži, kopiraj, prilepi in briši, podpira tudi tehniki razveljavi/uveljavi ter povleci/spusti. Jedilnik v primerni obliki za splet naj omogoča grafično prilagoditev, ne da bi za to potrebovali spremembe v rešitvi.

4.2 Izdelava modela

Za izdelavo modela smo iz opisa problema najprej izluščili glavne entitete, njihove lastnosti ter povezave med njimi. Identificirali smo pet glavnih entitet z lastnostmi in povezavami, kot jih prikazuje slika 4.1.

Vse entitete in povezave na sliki 4.1 smo definirali s pomočjo grafičnega vmesnika za urejanje Ecore modelov, tako da slika hkrati predstavlja tudi grafični diagram Ecore modela naše rešitve v orodju EMF.

Za omejitev zaloge vrednosti pri določanju številke tedna v letu smo vključili v model nov podatkovni tip `TedenVLetu`. Podatkovni tip smo izpeljali iz javanskega podatkovnega tipa za cela števila `java.lang.Integer`.



Slika 4.1: Začetni digram modela rešitve

Označili smo ga z imenskim prostorom `ExtendedMetaData`[1] in dodali ključa `minInclusive` z vrednostjo 1 ter `maxInclusive` z vrednostjo 53. Na ta način smo omejili vrednost za številko tedna v letu med vrednosti najmanj 1 in največ 53.

Zaradi zahteve rešitve, da se cena menuja izračuna na podlagi cen vsebovanih jedi, smo lastnost za ceno menuja označili kot izpeljano (ang. derived). V diagramu modela iz slike 4.1 je to ponazorjeno z znakom `/` pred imenom lastnosti. Izpeljane lastnosti se izračunajo na podlagi ostalih podatkov, zato smo za lastnost dodatno označili, da je nenastavljiva (ang. changeable), se trajno ne hrani (ang. transient) in za njo v primerku modela ne obstaja mesto za hranjenje (ang. volatile).

Izračun izpeljane lasnosti se ne da definirati v jeziku UML. Uporabiti smo morali poseben jezik OCL (ang. Object Constraint Language - OCL) [18] iz družine jezikov QVT. Izračun smo v modelu definirali s pomočjo urejevalnika OCLinEcore Editor [32], ki avtomatsko poskrbi za označbe (ang. annotations) in vključitve pravih imenskih prostorov v model. Izračun cene na entiteti `Menu` smo v jeziku OCL zapisali takole:

[1]http://org/eclipse/emf/ecore/util/ExtendedMetaData

```
initial: self.jedi.Cena->sum();
```

Za predstavitev tipa jedi smo v modelu definirali nov naštevni tip `JedEnum` z vrednostmi za vse štiri tipe jedi. Številnost povezave med entitetama `Menu` in `Jed` smo nastavili na 1..4, kar pomeni, da mora menu vsebovati vsaj eno jed in največ štiri. Za definiranje omejitev vsebovanosti jedi v menuju smo prav tako uporabili jezik OCL in urejevalnik OCLinEcore Editor. Omejitve na entiteti `Menu` smo v jeziku OCL za primer glavne jedi in sladice zapisali takole:

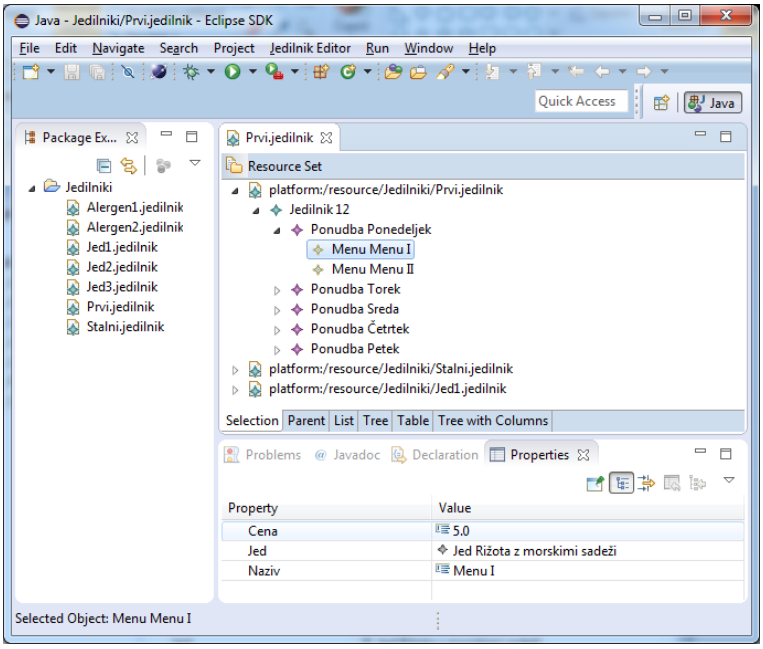
```
invariant glavnaJedTocnoEna:
   self.jedi-> select(j | j.Tip = JedEnum::GLAVNAJED)->size() = 1;
invariant sladicaSamoEna:
   self.jedi-> select(j | j.Tip = JedEnum::SLADICA)->size() <= 1;
```

Zaradi zahteve, da se jedi in njihove lastnosti določi le enkrat, smo povezavo med `Menu` in `Jed` nastavili kot nevsebujočo. V diagramu modela iz slike 4.1 je to ponazarjeno s puščico brez diamantka na začetku puščice. S tem smo dosegli, da se lahko isti primerek jedi uporabi za različne jedilnike. Enako smo storili tudi za povezavo `stalnaPonudba` med entitetama `Jedilnik` in `Ponudba` ter povezavo med entitetama `Jed` in `Alergen`.

4.3 Izdelava urejevalnika

Na podlagi modela iz slike 4.1 orodje EMF že omogoča generiranje kode urejevalnika. S privzetimi nastavitvami generatorskega modela smo s samo nekaj kliki izdelali kodo za vse tri nivoje, ki so omenjeni v prejšnjem poglavju. S tem smo prišli do urejevalnika za izdelavo primerkov modela rešitve, kot prikazuje slika 4.2.

Slika 4.2 prikazuje funkcionalno zaključeno aplikacij, ki podpira vse glavne zahteve rešitve. Za boljšo uporabniško izkušnjo pa smo morali narediti nekaj popravkov na modelu. V model smo dodali entiteto `Hrana`, ki združuje entiteti `Jed` in `Alergen`. S tem smo dosegli, da so lahko vse jedi in alergeni



Slika 4.2: Začetna oblika aplikacije

shranjeni v eni datoteki. V model smo dodali tudi entiteto `DnevnaPonudba` z lastnostjo `Dan`. S tem smo nadomestili ponavljajoče vpisovanje naziva dnevne ponudbe z avtomatskim določanjem dneva v tednu. Lastnost `Dan` smo označili kot izpeljano. V kodi na nivoju modela smo logiko za določitev dneva definirali takole:

```
/**
 * @generated NOT
 */
public Date getDan() {
   Jedilnik j = (Jedilnik)eContainer();
   int index = j.getDnevnaPonudba().indexOf(this);
   Calendar cal = Calendar.getInstance();
   cal.setTime(j.getOd());
   cal.add(Calendar.DATE, index);
   return cal.getTime();
```

```
}
```

Na koncu smo v kodi na nivoju prilagojevalnika za vsako entiteto modela popravili še besedilo in ikono, ki se prikažeta v urejevalniku. Nekaj več dela smo imeli z besedilom entitete `Jedilnik`. Delo z datumi je v Javi bolj zahtevno, kot smo pričakovali. Namesto številke tedna v letu smo prikazali začetni in končni datum jedilnika. Nekaj dodatne logike smo uporabili tudi pri izbiri ikone za entiteto `Jed` ter v kodi na nivoju prilagojevalnika za vsak tip jedi določili svojo ikono.

4.4 Izdelava primerne oblike za splet

Za izdelavo modela v obliki, ki je primerna za objavo na spletu, smo morali najprej izbrati orodje, s katerim bomo definirali preslikavo modela v besedilno obliko. V okviru projekta Eclipse Modeling Project so tri orodja [28]: JET, Acceleo in Xpand. Po pregledu njihovih lastnosti in dejstvu, da je projekt Acceleo od vseh najbolj aktiven, smo se odločili za orodje Acceleo [22].

Orodje Acceleo je odprtokodna rešitev, katere začetek sega v leto 2006. Gre za generator kode v obliki Eclipse vtičnika. Vsebuje lastni urejevalnik s podporami za dokončanje kode (ang. code completition), razhroščevanje (ang. debugging) in preurejanje (ang. refactoring). Sintaksa jezika, s katerim se v orodju definira preslikavo modela v besedilno obliko, je osnovana na OMG-jevih standardih MOFM2T [16] in OCL. Jezik s podporo predlogam omogoča modularno sestavo rešitve, možnost vključitve javanske kode pa omogoča definiranje operacij, ki v jeziku niso podprte. Neodvisnost od ogrodja Eclipse omogoča vključitev generatorja v katero koli Java aplikacijo.

Vsi pregledani primeri dokumentacije in navodil za orodje Acceleo so narejeni za generiranje kode iz Ecore modela. Za naš primer pa smo potrebovali izdelavo besedilne oblike iz primerka Ecore modela. Predloga, s katero se definira preslikavo, mora namesto entitet, kot sta `Class` in `Attribute`, uporabljati entiteti, kot sta `Jed` in `Alergen`. Z nekaj dodatnega prebiranja dokumentacije smo ugotovili, da je za naš primer potrebno posebej regi-

strirati paket. V generirano kodo, ki jo pripravi orodje za samo izvedbo preslikave modela v besedilno obliko, smo v metodo `registerPackages()` vključili registracijo naslednjega paketa:

```
EPackage.Registry.INSTANCE.put(
   jedilnik.JedilnikPackage.eNS_URI,
   jedilnik.JedilnikPackage.eINSTANCE);
```

Pri definiranju preslikave smo vse elemente preslikave obdali s sintakso jezika za označevanje nadbesedila (ang. Hyper Text Markup Language - HTML) ter tako dosegli, da je končna besedilna oblika primerna za objavo na spletu. Z uporabo zunanjih kaskadnih stilskih podlog (ang. Cascading Style Sheets - CSS) smo definicijo grafične podobe prestavili v ločeno statično datoteko ter s tem omogočili spremembo grafične podobe brez posegov v aplikacijo. Za predstavitev menuja in pripadajoče cene smo v preslikavi združili sintaksi HTML in Acceleo:

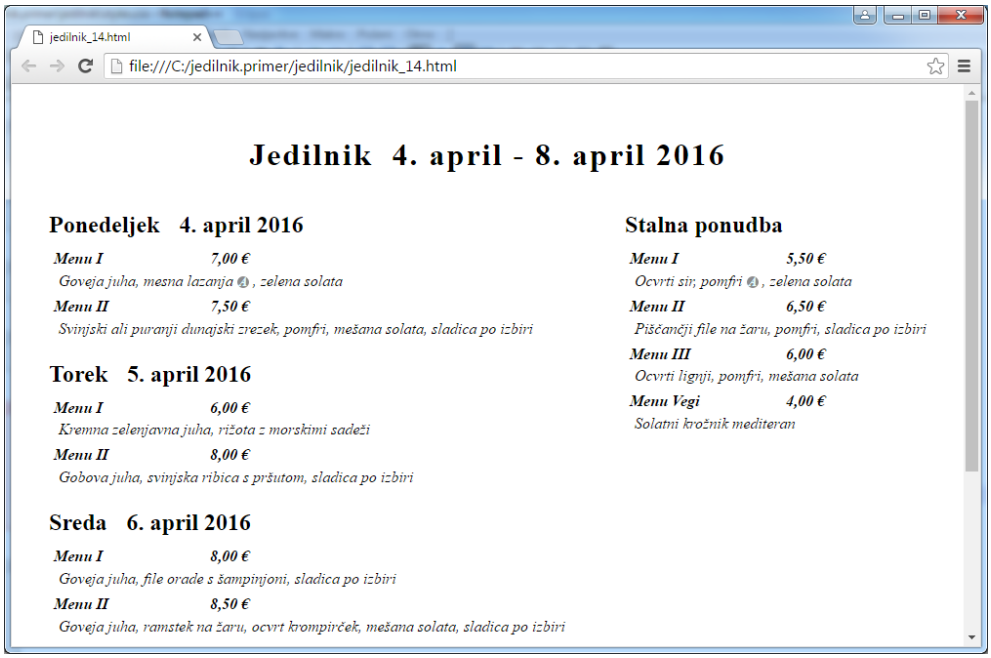
```
<p class="menu">[menu.Naziv/]
   <span class="price">[menu.Cena.Amount()/] &euro;</span>
</p>
```

Posebno pozornost smo morali nameniti prikazu jedi v menuju. Jedi se v modelu pojavljajo v vrstnem redu vnosa, pri objavi pa se morajo prikazati v vrstnem redu glede na tip jedi: najprej juha, nato glavna jed, na koncu pa solata in sladica. V orodju Acceleo ukaz za razvrščanje ni deloval za naštevni tip `JedEnum`, zato smo za entiteto `Jed` v model dodali dodatno lastnost `TipAsInt`. Lastnost smo označili kot izpeljano, da smo lahko v kodi na nivoju modela definirali logiko, ki je naštevni tip preslikala v številčno vrednost. S to spremembo modela smo lahko definirali preslikavo za jedi v menuju v obliki Acceleo pomožne metode:

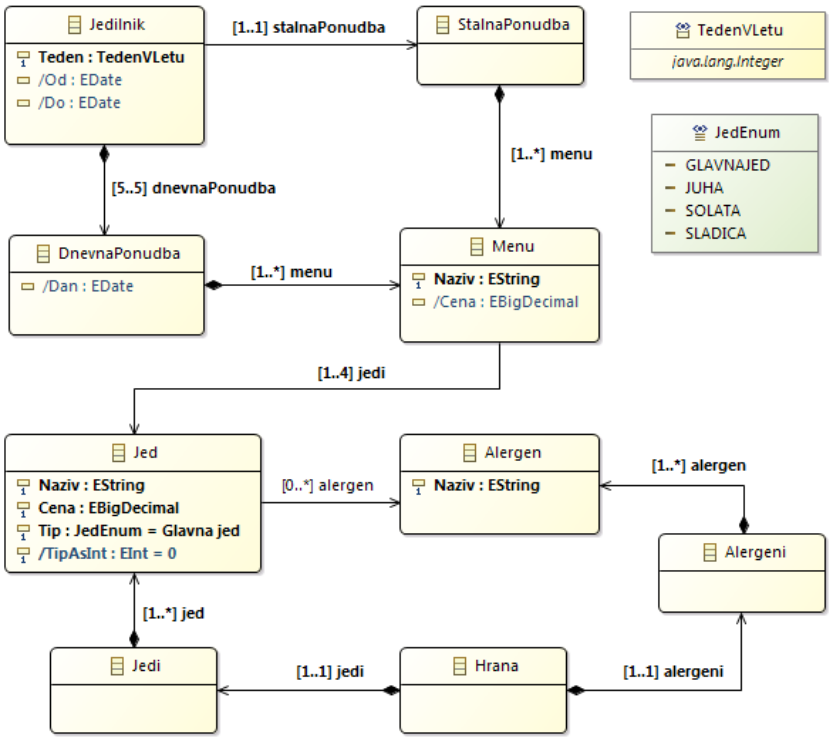
```
[template public generateMenuJedi(aMenu : Menu)]
   <p class="content">
   [for (aJed : Jed | aMenu.jedi->sortedBy(TipAsInt))
      separator(', ')]
```

```
      [if (i = 1)]
         [aJed.Naziv.toUpperFirst()/]
         [generateAlergeni(aJed)/]
      [else]
         [aJed.Naziv.toLower()/]
         [generateAlergeni(aJed)/]
      [/if]
   [/for]
   </p>
[/template]
```

S pomožno metodo smo poleg pravega vrstnega reda jedi poskrbeli še za velike začetnice jedi in vejice med njimi. Končno obliko jedilnika v spletnem brskalniku prikazuje slika 4.3. Z dodano lastnostjo `TipAsInt` smo prišli tudi do končne oblike modela, kot ga prikazuje slika 4.4.



Slika 4.3: Prikaz jedilnika v spletnem brskalniku



Slika 4.4: Končni digram modela rešitve

4.5 Izdelava samostojne rešitve

Aplikacija iz slike 4.2 je izvedena kot vtičnik za razvojno okolje Eclipse. Uporaba tega ogrodja je namenjena razvijalcem rešitev in ne uporabnikom le teh. Zato smo se odločili, da aplikacijo prenesemo v samostojno rešitev, ki za delovanje ne potrebuje razvojnega okolja Eclipse. V orodju EMF smo izkoristili podporo za aplikacije tipa Rich Client Platform (RCP) [10], ki so namenjene takšnim primerom.

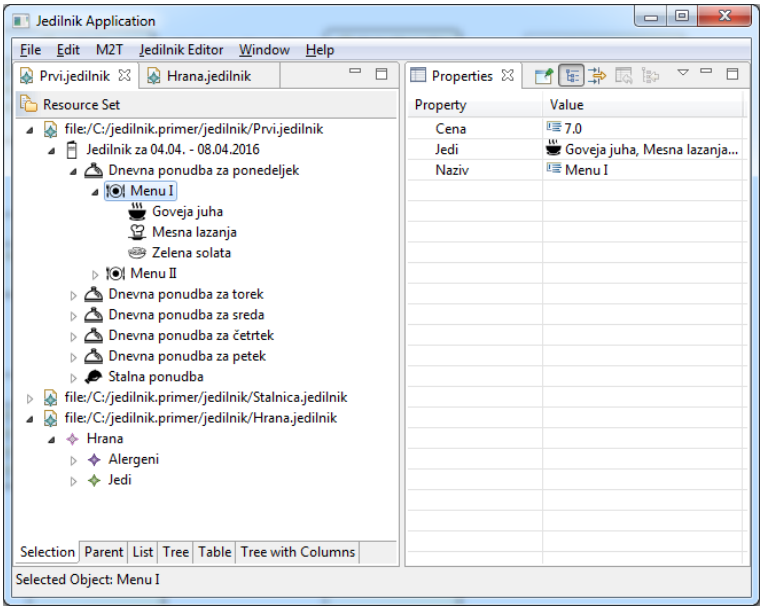
Pri izdelavi RCP aplikacije smo morali v generatorskem modelu spremeniti vrednost nastavitve *Editor.Rich Client Platform* na vrednost *true* in ponovno izvesti generiranje kode urejevalnika. Ob testiranju RCP aplikacije se je izkazalo, da podpora jeziku OCL ne deluje. Sledenje navodilom za uporabo jezika OCL v samostojnih rešitvah [32] nas je pri iskanju rešitve zavedlo. Izkazalo se je, da navodila niso bila prilagojena zadnji različici ogrodja Eclips.

V samostojni rešitvi smo tudi poenostavili postopek preslikave jedilnika v HTML obliko. Namesto definiranja parametrov preslikave v izvajalni postavitvi vtičnika (ang. plugin runtime configuration) smo vse avtomatizirali v ukaznem obdelovalniku (ang. command handler), ki je vezan na element menija v aplikaciji. Želeli smo, da klik na element menija preslika trenutno izbrani model v besedilno datoteko na lokacijo izvornega modela v datotečnem sistemu. Za določanje izvornega modela in ciljne mape smo morali raziskati zmožnosti programskega vmesnika Eclipse. Z uporabo tudi nekaj bolj osnovnih javanskih knjižnic, kot sta `java.io.File` in `java.util.Collections`, smo prišli do izvedbe ukaznega obdelovalnika:

```
public Object execute(ExecutionEvent event)
 throws ExecutionException {
   IEditorPart editor = HandlerUtil.getActiveEditor(event);
   URIEditorInput input = (URIEditorInput)editor.getEditorInput();
   String segment = input.getURI().lastSegment();
   String targetPath =
       input.getURI().devicePath().replaceAll(segment, "");
   File targetFolder = new File(targetPath);
   Generate g = new Generate(input.getURI(), targetFolder,
       Collections.emptyList());
   g.doGenerate(null);
   return null;
}
```

Na koncu smo RCP aplikacijo izvozili še v samostojni paket, v katerem so vključene vse potrebne javanske knjižnice in lastni sprožilec (ang. native launcher). Izvoz v samostojni paket je zahteval tudi razrešitev razlik v različicah odvisnih javanskih knjižnic. Za knjižnice, ki so prisotne v ogrodju Eclipse v različnih različicah, smo morali v nastavitvah izvoza navesti različico knjižnice, ki naj bo uporabljena v izvozu. Tako izdelan paket nima zunanjih odvisnosti in se lahko namesti z enostavnim kopiranjem v datotečni sistem ciljnega računalnika.

Izvajanje lastnega sprožilca odpre aplikacijo za urejajanje jedilnika, kot je prikazana na sliki 4.5.



Slika 4.5: Končna oblika aplikacije

Poglavje 5

Analiza uporabe pristopa

Ob začetnem branju literature na temo modelno vodenega razvoja programske opreme smo dobili občutek, da se da obsežne in zapletene aplikacije narediti dokaj hitro le z risanjem modelov. Z nadaljnim proučevanjem in praktičnim preizkušanjem obravnavanega področja smo ugotovili, da gre v teh trditvah predvsem za oglaševalske trditve.

Realne zmožnosti modelno vodenega pristopa so vsaj v trenutni fazi razvoja področja še precej odvisne od klasičnega programiranja. To še toliko bolj velja za orodje EMF, ki že po besedah snovalcev tega orodja predstavlja srednjo pot med MDD in klasičnim programiranjem.

V nadaljevanju bomo predstavili ugotovitve in spoznanja o modelno vodenem razvoju s pomočjo orodja EMF, do katerih smo prišli med izdelavo diplomskega dela.

5.1 Področja primerna za uporabo pristopa

Tenutno razvojno stanje modelirnih jezikov ne omogoča prikladnega načina modeliranja vseh različnih vidikov problemske domene. Tukaj mislimo na modeliranje na nivoju, ki bi zajel vse potrebne informacije za preslikavo modela v izvedljivo rešitev na način, ki bi bil učinkovitejši od klasičnega programiranja.

Orodje EMF za glavni modelirni jezik uporablja jezik UML, katerega izrazna moč je definirana z devetimi diagrami razdeljenimi v dve skupini [3]. Prva skupina pokriva statične vidike problemske domene, druga pa dinamične vidike oziroma vidike obnašanja sistema.

Za modeliranje statičnega vidika sistema se zelo dobro obnesejo razredni diagrami. Pri modeliranju dinamičnega vidika sistema pa se zaplete. Prva izbira pri modeliranju dinamičnega vidika so diagrami prehajanja stanj, ki pa niso dobri pri opisovanju obnašanja večjega številka sodelujočih objektov [3]. Avtorji orodja EMF so se tako pragmatično odločili, da v orodju podprejo samo razredne diagrame. S tem so omejili področja, ki so primerna za uporabo orodja na podatkovno intenzivne (ang. data-centric) aplikacije v domeni podpore poslovanju podjetij. Lastnost teh aplikacij je veliko število povezanih entitet in njihovih lastnosti, za katere je potrebno v aplikaciji omogočiti osnovne operacije urejanja, zagotoviti pravilnost vnosa in trajno shranjevanje. Veliko ponavljajoče kode je glavna značilnost takih aplikacij. Z orodji za podporo izdelavi na obrazcih zasnovanega grafičnega uporabniškega vmesnika, kot sta orodji EMF Cliet Platform [24] in EMF Forms [27], se to področje še tesneje prilega orodju EMF. To seveda ne pomeni, da s pomočjo orodja EMF ni možno zgraditi različnih tipov aplikacij ali orodij, po našem mnenju so podatkovno intenzivne aplikacije samo najprimernejše za uporabo tega pristopa.

5.2 Dvig nivoja abstrakcije

Ena od glavnih lastnosti modelno vodenega pristopa pri razvoju programske opreme je opis rešitve na višjem abstraktnem nivoju. Z izbiro razrednih diagramov kot edinih UML diagramov, ki so podprti v orodju EMF, pa ta lastnost v primerjavi z višjimi programskimi jeziki tretje generacije ne predstavlja praktično nobenega napredka. Preslikava razrednih diagramov na nivo objektno usmerjenih programskih jezikov je dokaj enostavna operacija, ki razrede UML diagrama, njihove atribute in operacij preslika v objektne ra-

zrede z atributi in operacijami. V dodatne atribute razredov pa preslika tudi asociacije med razredi. Osnovni koncepti, s katerimi modeliramo v UML, so tako enaki konceptom objektnih programskih jezikov, kar ni presenetljivo, če upoštevamo, da je bil jezik UML v osnovi razvit z namenom poenotenja več objektno usmerjenih grafičnih modelirnih jezikov [3]. Prednost uporabe orodja EMF pri razvoj programske opreme v primerjavi z višjimi programskimi jeziki je tako iz stališča abstrakcije omejena na prednosti grafičnega zapisa, ki ga prinaša UML.

5.3 Produktivnost pristopa

Glavni namen uporabe modelno vodenega pristopa je povečanje učinkovitosti razvoja programske opreme. Za eno od glavnih lastnosti orodja EMF, to je opis rešitve na višjem nivoju abstakcije, smo v prejšnjem podpoglavju pokazali, da ima le manjši doprinos k povečani učinkovitosti. Ostane torej še del orodja, ki iz modela generira kodo, ki je uporabljena v končni rešitvi. Na praktičnem primeru iz diplomske naloge smo po izdelavi začetnega modela s pomočjo generatorja kode avtomatsko izdelali več kot 6000 vrstic kode rešitve. S podporo prikazu, urejanju in shranjevanju je tako aplikacija v grobem že ustrezala glavnim zahtevam rešitve. Količina ročnega dela, ki je bila z avtomatsko izdelavo nadomeščena, se lahko po naši oceni za naš primer meri v dnevih.

Za kratek projekt, kot je naš primer, je to velik skok v produktivnosti v primerjavi s klasičnim programiranjem. Generirana koda je zelo berljiva, kar je bil tudi sam cilj snovalcev orodja in predpogoj k ideji, da je avtomatsko izdelana aplikacija samo osnova, na kateri se izvede dodatne prilagoditve bolj specifičnim in podrobnim zahtevam rešitve. Očitno je, da manj specifičnih in podrobnih zahtev pomeni manj ročnega dopolnjevanja generirane programske kode. Tako je orodje EMF idealno za izdelavo rešitev z ohlapno definiranimi zahtevami. Primer tega bi lahko bile prototipne aplikacije, katerih izdelava je z orodjem hitra in ne zahteva programerskega znanja. Ko bi

se naročnik na podlagi preučitve prototipa opredelil glede podrobnejših zahtev, bi lahko razvoj aplikacije prevzeli programerji in v avtomatsko izdelano osnovo vključili še specifične zahteve.

5.4 Uvajanje pristopa

Slika o prototipnem razvoju, ki smo jo predstavili v prejšnjem podpoglavju, je žal malce idealistična. Po naših izkušnjah se do delujoče aplikacije s pomočjo orodja EMF res pride hitro in enostavno. Zaplete pa se z vpeljavo specifičnih in podrobnih zahtev v rešitev. Količina potrebnega znanja, da se v generirano kodo vključijo dodatne zahteve, je res velika. Poleg poznavanja in razumevanja generirane kode je potrebno poznati in razumeti tudi programski vmesnik, ki ga ponuja orodje EMF za delo z modeli. To znanjen pa se mora navezovati na celotno razvojno ogrodje Eclipse, na katerem sloni izdelana rešitev. Množica komponent in vtičnikov, ki sestavljajo Eclipse, otežuje izbiro najprimernejšega za konkretno nalogo. Dokumentacija in navodila največkrat ne razrešijo težav. V splošnem manjka kvalitetne dokumentacije in navodil[1]. Pri izdelavi specifičnih zahtev za našo rešitev ali razrešitvi nerazumljivih napak v kodi smo se tako morali večkrat poslužiti kar preiskovanja spletnih forumov, na katerih si razvijalci delijo svoje izkušnje pri delu z ogrodjem. Pridobivanje znanja na tak način je počasno in največkrat pripelje tudi do neoptimalne rešitve. Kot celota zahteva orodje EMF precej strmo krivuljo učenja.

Dejstvo, da je orodje EMF in tudi večina orodij, ki so zrasla okrog njega, dostopna brezplačno, zmanjšuje stroške razvoja programske opreme. Veliko podjetij, kot je IBM ter aktivna množica ostalih podjetij in posameznikov, ki stojijo za orodjem, pa zagotavlja, da začetni vložek v pridobitev potrebnega znanja za obvladovanje razvoja z orodjem ne bo imel kratkega roka uporabnosti.

[1]Edina knjiga o EMF je drugo in zadnjo izdajo doživela v začetku leta 2009

5.5 Življenjski cikel pristopa

Spremembe programske opreme so del njihovega življenjskega cikla. Snovalci orodja EMF so zato za podporo spremembam vgradili možnost spreminjanja modela in ponovno generiranje kode, ki obdrži celotno ročno spremenjeno ali dodano kodo. Izgubljena koda nas je občasno spomnila, da kot razvijalci ne smemo pozabiti označiti ročno spremenjene ali dodane kode. Z uporabo orodja za nadzor različic bi se temu lahko izognili, vendar obseg našega primera ni upravičeval uporabe takšnega orodja.

Pomemben korak v procesu vpeljave sprememb v aplikacijo, ki zahtevajo spremembo modela, je preslikava že obstoječih primerkov modelov v novo različico modela. Za naš primer aplikacije smo po spremembah modela večkrat zavrgli nedelujoče različice primerkov modelov. V primeru pomembnejših podatkov ali večje količine podatkov bi morali z uporabo M2M (ang. Model to Model) orodij preslikati obstoječe primerke v novo različico modela.

V primeru sprememb v modelu je bila za naš primer izguba obstoječih primerkov modela edina težava, ki bi lahko zahtevala dodatno delo. Spremembe v modelu so bile namreč v grafičnem vmesniku avtomatično vključene. Pri tem je seveda pomembno, da generičnost prikaza ustreza zahtevam rešitve oziroma da le te niso preveč specifične. Podpora specifičnim zahtevam zmanjša učinkovitost uporabe orodja EMF, saj to pomeni ročni poseg v generirano kodo.

Poglavje 6

Zaključek

V okviru diplomske naloge smo najprej proučili področje MDD oziroma OMG različico imenovano MDA. Nadaljevali smo s preučevanjem orodja EMF, ki predstavlja konkretno izvedbo MDA smernic in standardov. Na koncu smo pridobljeno znanje uporabili pri izgradnji enostavne aplikacije za končnega uporabnika.

Ugotovili smo, da z orodjem EMF hitro in enostavno pridemo do generično izdelane aplikacije, ki ustreza ohlapno definiranim zahtevam. Za podporo specifičnim in podrobnim zahtevam v aplikaciji pa je potrebnega veliko več dela in predvsem znanja. Tudi samo ogrodje Eclipse dodatno zaplete prilagajanje generične rešitve specifičnim zahtevam. Poznavanje delovanja in interakcije zapletenih sestavnih delov ogrodja predstavlja precejšen izziv in strmo krivuljo učenja. Obseg potrebnega znanja za izdelavo aplikacij s pomočjo orodja EMF se s tem zelo poveča in hkrati nagne na stran programiranja. Tako je končni občutek pri delu z orodjem EMF, da je za izdelavo specifične rešitve s podrobnimi zahtevami še vedno potrebno večinoma programersko znanje in rešitve.

Za izboljšanje obstoječe rešitve bi lahko razbili model aplikacije na dva dela. En del bi modeliral koncepte hrane, drugi del pa koncepte jedilnika in restavracije. S tem bi lahko dosegli ponovno uporabo modela za hrano v drugih aplikacijah, kjer bi ga nadgradili recimo v aplikacijo za podporo izdaji

kuharskih receptov. Večjo izboljšavo bi lahko dosegli z zamenjavo avtomatsko generirane kode za grafični vmesnik z orodji, ki omogočajo modelni pristop k izdelavi grafičnih vmesnikov. Z uporabo orodja EMF Forms [27] bi tako lahko dobili definicijo grafičnega vmesnika aplikacije z možnostjo avtomatske izdelave vmesnika za namizje, splet ali mobilne naprave.

Pri proučitvi orodja EMF nismo pokrili dveh pomembnih področij orodja. Tako bi v nadalje veljalo proučiti zmogljivost orodja pri obvladovanju večje količine podatkov. Privzet mehanizem v orodju uporablja za shranjevanje modelov besedilne datoteke. Potrebno bi bilo ugotoviti, kakšne so omejitve v primeru uporabe datotečnega sistema za shrambo modelov ter tudi kakšno izboljšavo (in morebitne zaplete ali omejitve) predstavljajo orodja za podporo shranjevanja modelov v relacijske baze v primeru spreminjanja modelov. Primer takšnega orodja je EMF Teneo [30]. Drugo področje, ki ga nismo preučili, predstavljajo orodja za nadzor različic modelov. Brez orodij za nadzor različic si ne predstavljamo faze vzdrževanja aplikacije. Potrebno bi bilo ugotoviti, kakšno razvojno zrelost so orodja, kot na primer EMF Store [29], že dosegla v primerjavi s podobnimi dobro razvitimi orodji za nadzor različic klasične programske kode.

Literatura

[1] Kent Beck, Cynthia Andres, "Extreme Programming Explained (2nd Edition)", *Addison-Wesley*, 2005.

[2] Marco Brambilla, Jordi Cabot, Manuel Wimmer, "Model-Driven Software Engineering in Practice", *Morgan & Claypool*, 2012.

[3] Martin Fowler, "UML Distilled (3rd Edition): A Brief Guide to the Standard Object Modeling Language", *Addison-Wesley*, 2004.

[4] David S. Frankel, "Model Driven Architecture: Applying MDA to Enterprise Computing", *Wiley Publishing, Inc.*, 2003.

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides "Design Patterns: Element of Reusable Object-Orientated Software", *Addison-Wesley*, 1995.

[6] Stephen J. Mello, Kendall Scott, Axel Uhl, Dirk Weise, "MDA Distilled: Principles of Model-Driven Architecture", *Addison-Wesley*, 2004.

[7] Dave Steinberg, Frank Budinsky Marcelo Paternostro, Ed Merks, "EMF: Eclipse Modeling Framwork (2nd Edition)", *Pearson Education, Inc.*, 2009.

[8] .NET [Online]. Dosegljivo: `http://www.microsoft.com/net/`. [Dostopano 27. 4. 2016].

[9] CORBA [Online]. Dosegljivo: `http://www.corba.org/`. [Dostopano 27. 4. 2016].

[10] Eclipse Rich Client Platform [Online]. Dosegljivo: https://wiki.eclipse.org/Rich_Client_Platform. [Dostopano 4. 4. 2016].

[11] Eclipse JFace [Online]. Dosegljivo: https://wiki.eclipse.org/JFace. [Dostopano 2. 4. 2016].

[12] EMF [Online]. Dosegljivo: http://www.eclipse.org/modeling/emf. [Dostopano 2. 4. 2016].

[13] EMF pregled [Online]. Dosegljivo: http://help.eclipse.org/mars/topic/org.eclipse.emf.doc/references/overview/EMF.html. [Dostopano 2. 4. 2016].

[14] Java [Online]. Dosegljivo: http://www.java.com/. [Dostopano 27. 4. 2016].

[15] MOF standard [Online]. Dosegljivo: http://www.omg.org/spec/MOF/. [Dostopano 27. 4. 2016].

[16] MOF Model To Text Transformation standard [Online]. Dosegljivo: http://www.omg.org/spec/MOFM2T/1.0/. [Dostopano 27. 4. 2016].

[17] OMG [Online]. Dosegljivo: http://www.omg.org. [Dostopano 2. 4. 2016].

[18] OMG specifikacija jezika OCL [Online]. Dosegljivo: http://www.omg.org/spec/OCL/. [Dostopano 2. 4. 2016].

[19] OMG specifikacija standarda DI [Online]. Dosegljivo: http://www.omg.org/cgi-bin/doc?ptc/2003-09-01. [Dostopano 12. 5. 2016].

[20] OMG specifikacija standarda MDA [Online]. Dosegljivo: http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf. [Dostopano 2. 4. 2016].

[21] OMG specifikacija jezikov QVT [Online]. Dosegljivo: `http://www.omg.org/spec/QVT/1.1/`. [Dostopano 12. 5. 2016].

[22] Orodje Acceleo [Online]. Dosegljivo: `https://eclipse.org/acceleo`. [Dostopano 4. 4. 2016].

[23] Orodje Eclipse, različica Mars [Online]. Dosegljivo: `https://projects.eclipse.org/releases/mars`. [Dostopano 4. 4. 2016].

[24] Orodje EMF Client Platform [Online]. Dosegljivo: `https://eclipse.org/ecp/`. [Dostopano 12. 5. 2016].

[25] Orodje EMF Compare [Online]. Dosegljivo: `https://www.eclipse.org/emf/compare/`. [Dostopano 12. 5. 2016].

[26] Orodje EMF Edapt [Online]. Dosegljivo: `https://www.eclipse.org/edapt/`. [Dostopano 12. 5. 2016].

[27] Orodje EMF Forms [Online]. Dosegljivo: `http://www.eclipse.org/ecp/emfforms/`. [Dostopano 27. 4. 2016].

[28] Orodja EMF M2T [Online]. Dosegljivo: `http://www.eclipse.org/modeling/m2t/`. [Dostopano 2. 4. 2016].

[29] Orodje EMF Store [Online]. Dosegljivo: `http://www.eclipse.org/emfstore/`. [Dostopano 12. 5. 2016].

[30] Orodje EMF Teneo [Online]. Dosegljivo: `http://wiki.eclipse.org/Teneo/`. [Dostopano 12. 5. 2016].

[31] SWT [Online]. Dosegljivo: `https://www.eclipse.org/swt/`. [Dostopano 12. 5. 2016].

[32] Urejevalnik OCLinEcore Editor [Online]. Dosegljivo: `https://wiki.eclipse.org/OCL/OCLinEcore`. [Dostopano 4. 4. 2016].