Fakulteta za računalništvo in informatiko
Univerza v Ljubljani

DAVID VIDRIH

MODELNO VODENI RAZVOJ: OVREDNOTENJE KONCEPTOV IN RAZVOJ PODPORNIH ORODIJ

MAGISTRSKO DELO

Mentor: prof. dr. Viljan Mahnič

Ljubljana, 2008

Zahvala

V začetku se moram zahvaliti svojim staršem in brezplačnemu visokemu šolstvu, kar mi je omogočilo študij.

Poleg študija sem veliko znanja in praktičnih izkušenj pridobil med delom v podjetju Marand d.o.o., ki je tudi sponzor mojega podiplomskega študija. V okviru dela sem se dejansko začel zanimati za obravnavano temo.

Zahvala gre v prvi vrsti mentorju prof. dr. Viljanu Mahniču za vso pomoč in nasvete ter za pregled in komentarje vsebine naloge.

Zahvaljujem se Slavku Drnovščku za pomoč pri iskanju virov.

Zahvalil bi se tudi Jerneju Ivančiču in Marku Poženelu, ki sta me navdušila nad LaTeX-om, katerega filozofija podajanja dejstev v osnovni obliki se zelo sklada z obravnavano temo.

Zahvalo si zasluži tudi lektorica Manja Krapež za hitro in kvalitetno delo v danem nemogočem roku.

Na koncu se zahvaljujem še vsem, ki so mi nudili moralno podporo, predvsem življenjski sopotnici Valeriji.

Kazalo

Slike

Kratice

Kratica	Pomen	Razlaga
API	Application Programming Interface	vmesnik do programske komponente
CIM	Computation Independent Model	procesno neodvisni model
DSL	Domain-Specific Language	jezik, specifičen za določeno domeno
DTO	Data Transfer Objects	standardni vzorec razredov za prenos podatkov
EJB	Enterprise Java Beans	strežniška komponentna arhitektura za modularno gradnjo programskih rešitev
HTML	HyperText Markup Language	označevalni jezik za izdelavo spletnih strani
JEE	Java Enterprise Edition	različica programskega jezika Java za izdelavo strežnikov
MDA	Model Driven Architecture	glavni standard za modelno vodeni razvoj
MDD	Model Driven Development	modelno vodeni razvoj programske opreme
MOF	Meta Object Factory	standard za formalno definicijo modelov in meta modelov
MVC	Model-View-Controller	standardni vzorec za ločevanje poslovne logike od uporabniškega vmesnika
OCL	Object Constraint Language	jezik za podajanje omejitev, poizvedb in postopkov za UML
OMG	Object Management Group	organizacija, ki je razvila standard MDA
PIM	Platform Independent Model	model, neodvisen od izvajalnega okolja
PSM	Platform Specific Model	model za izbrano izvajalno okolje
QVT	Query View Transformation	standard za poizvedbe, poglede in preslikave modelov
SQL	Structured Query Language	jezik za podajanje poizvedb za podatkovne baze
UML	Unified Modeling Language	splošno-namenski modelni jezik
XMI	XML Metadata Interchange	standard za izmenjavo MOF modelov in meta podatkov
XML	Extensible Markup Language	splošno-namenska osnova za označevalne jezike
XSLT	Extensible Stylesheet Language Transformations	jezik za preslikavo XML-a

Povzetek

Magistrska naloga obravnava *modelno vodeni pristop* k razvoju programske opreme kot eno najmočnejših smernic prihodnosti tega področja. Osnovna ideja pristopa je opisovanje problemskih domen z modeli in avtomatično preslikovanje teh modelov v končne programske rešitve. Pristop obljublja zmanjšanje sredstev razvoja ter naprednejši zapis podajanja navodil računalniku.

Modelno vodeni pristop je še v razvojni fazi in ga tako še ni možno uporabljati v idealni obliki izključnega risanja modelov. Tako je treba tiste dele rešitve, ki še niso podprti z obravnavanim pristopom, izdelati klasično, ali pa za te nepodprte domene najprej razviti modelno vodeno podporo, kar je precej zahtevna naloga. Delovanje v okviru podprtih zmožnosti izpolnjuje dane obljube, preseganje teh okvirov pa je precej zahtevno.

Glavna različica pristopa je *standard MDA*, ki ponuja dobro konceptualno osnovo, ne podaja pa dejanskih rešitev. Dobro dopolnitev omenjenemu standardu predstavljajo *domensko-specifični (modelni) jeziki*, ki omogočajo izdelavo lastnih jezikov za poljubne domene, vendar tudi ti ne dajejo dejanskih rešitev. Tako ugotavljamo, da je osnovni problem področja razvoja programske opreme pomanjkanje dobrih teoretičnih rešitev za podajanje navodil računalniku.

Podrobneje obravnavamo možnosti izdelave in uporabe orodij za podporo modelno vodenemu pristopu: možnosti izdelave lastnih podpornih orodij od začetka, uporabo delnih rešitev in naprednih podpornih razvijalskih okolij. Obravnavana je tudi možnost uporabe idej modelno vodenega pristopa pri klasičnem razvoju. Celotna obravnava je podkrepljena s prototipi.

Ugotavljamo tudi določene slabosti obravnavanega pristopa, že omenjeno pomanjkanje dejanskih rešitev, vprašljivost absolutne idealnosti grafičnih konkretnih zapisov ter problem podvojevanja dela in zapisa v smislu specifikacije in izvedbe programske rešitve. Na podlagi teh pomanjkljivosti podajamo predlog *izvedljive specifikacije*. *Izvedljiva specifikacija* je enoten zapis za formalno specifikacijo programskih rešitev, ki je hkrati direktno izvedljiv. Gre za zapis, katerega dejanski primerki so zaradi organiziranosti, možnosti ponovne uporabe, razumljivosti in obvladljivosti razbiti na smiselne vsebinske enote, ki se lahko sklicujejo ena na drugo. Vsebina enot pa je formalni jezik, ki se zaradi intuitivnosti in minimalnosti potrebnega znanja za razumevanje bere kot naravni jezik. Zaradi obširnosti problema podpore poljubnim domenam smo razvili splošno-namenske izrazne zmožnosti ter izrazne zmožnosti za domeno grafičnega vmesnika. Razvili smo tudi orodje za direktno izvajanje tako definiranih *izvedljivih specifikacij*.

Bistvo ideje *izvedljive specifikacije* je agregacija obstoječe ogromne količine znanja in idej s področja razvoja programske opreme v kompakten visokonivojski specifikacijski jezik brez odvečnih tehničnih podrobnosti, ki ga je moč brez dodatnega dela izvajati.

Ključne besede: modelno vodeni razvoj, MDA, domensko-specifični jeziki, podporna orodja, napredna navodila računalniku, izvedljiva specifikacija.

Abstract

The master's thesis discusses a *model driven approach* to software development as one of the best future ideas in this domain. The basic idea is to describe the problem domain using models which are then automatically transformed to final software solution. The approach promises a reduction in development resources and advanced form of specifying instructions to computers.

Model driven approach is still in development phase, and thus can not yet be used in its ideal form of exclusive models drawing. The unsupported parts of developed solution must be developed using conventional approaches, the alternative being development of model driven support for unsupported domains, which can be very demanding task. Working within the supported capabilities meets the promises, but overcoming these limits is usually quite demanding.

The main version of this approach is the *MDA standard*, which offers a good conceptual basis, but does not offer concrete solutions. A good complement to MDA are *domain-specific (modeling) languages*, that can be used for designing custom languages for any domain, which also do not provide concrete solutions. Thus, we recognize the lack of good theoretical solutions for specifying instructions to computers being the basic problem of software development domain.

The thesis also contains a detailed discussion of possibilities for model driven approach supporting tools development and use: possibilities for custom support tools development from scratch, use of partial solutions and advanced development support environments. We also discuss the possibilities of using model driven approach ideas in conventional development. The whole tools discussion is supported with concrete prototypes.

We also identified certain weaknesses of model driven approach, such as already mentioned lack of concrete solutions, the questionable absolute ideality of graphic concrete syntaxes, and the problem of duplication of work and solution description by first writing specification and then implementation. Basing on these weaknesses we propose the *executable specification*; a unified language which is suitable for formal specification of software solutions and is at the same time also directly executable. For the purpose of organization, reusability, comprehensiveness and manageability, the specifications in proposed language may be broken into meaningful units, which may refer to each other. The language itself, due to intuitiveness and minimal required knowledge for its understanding, reads as a natural language. Due to the fact, that creating the support for wide variety of possible domains would be unfeasible for us, we have limited ourselves to developing the support for general-purpose specifications and for graphic interface specific domain. We have also developed a tool for direct execution of the defined *executable specifications*.

The essence of the *executable specification* idea is aggregation of existing huge amounts of knowledge and ideas of software development domain into a compact

high level specification language without unnecessary technical details, which can be executed without additional work.

Key words: model driven development, MDA, domain-specific languages, support tools, advanced computer instructions, executable specification.

Poglavje 1

Uvod

Računalništvo je eno izmed najhitreje razvijajočih se področij, kar velja tudi za obravnavano ožje področje razvoja programske opreme. Napredek se na področju razvoja programske opreme, kot tudi splošno, dogaja v ciklih. Ko trenutni cikel dozori, se išče ideje za reševanje slabosti trenutnega cikla. Pri tem iskanju se prej ali slej pojavi ideja, ki presega okvire trenutnega cikla in tako začne nov cikel. Osnovna ideja novega cikla se postopoma razvija in izboljšuje in tako na koncu privede do zrelosti, kjer se ugotovi nove omejitve, kar prej ali slej požene nov cikel [13, str. 32].

Trenutni cikel razvoja programske opreme je najbolj zaznamovan z objektno usmerjenim pristopom in je že v precej zrelem obdobju. V zadnjem času se je pojavilo veliko dobrih idej in praktično uporabnih pristopov, kar daje slutiti prihod novega razvojnega cikla. V tem novem ciklu bo najverjetneje v ospredju dvig abstraktnega nivoja podajanja navodil računalniku, v tem okviru pa morda tudi prehod iz klasičnega programiranja na modeliranje.

Magistrsko delo se ožje nanaša na *modelno vodeni pristop* k razvoju programske opreme [21], ki predstavlja eno najobetavnejših idej prihajajočega cikla razvoja programske opreme, v širšem smislu pa obravnava vse sodobne pristope in ideje, v katere je modelno vodeni razvoj bolj ali manj vpet.

V nadaljevanju uvoda v razdelku 1.1 najprej predstavimo razvoj programskih jezikov in pristopov do vključno objektno usmerjenega obdobja. V razdelku 1.2 predstavimo modelno vodeni razvoj z njegovimi glavnimi različicami. Sledi razdelek 1.3 s predstavitvijo drugih pomembnih sodobnih idej in pristopov. Naprej, v razdelku 1.4, podajamo cilje in pričakovane rezultate magistrske naloge. Poglavje zaključimo s krajšim pregledom nadaljnjih poglavij naloge v razdelku 1.5.

1.1 Stanje pred modelno vodenim razvojem

Programska oprema postaja iz dneva v dan bolj zapletena, obsežna, porazdeljena in na druge načine zahtevna. Zato programska industrija ves čas išče boljše načine

za njen razvoj.

Najprej so ljudje računalnikom navodila podajali v *strojnem jeziku*. Prva izboljšava je bil *zbirni jezik*, ki je predstavljal nov nivo nad strojnim jezikom. Pri zbirnem jeziku je šlo predvsem za zamenjave strojnih kod s človeku bolj prijaznimi ukazi, medtem ko so bili programi večinoma še vedno vezani na določen računalnik. V primeru spremembe strojnega jezika pa ni bilo treba popravljati vseh programov v zbirnem jeziku, ampak le prevajalnik iz zbirnega v strojni jezik. V sodobnem duhu modelno vodenega razvoja to pomeni, da je zapis navodil računalniku ostal nespremenjen, spremenila se je le preslikava na nižji nivo. Programov tako ni bilo treba čisto nič popravljati, le ponovno jih je bilo treba prevesti. Seveda je moral razvijalec prevajalnika popraviti prevajalnik, res pomembno pa je, da je na tisoče programov v zbirnem jeziku ostalo enakih.

Po zbirnem jeziku so se pojavili *programski jeziki tretje generacije*. Ta prehod se večinoma označuje kot največji napredek na področju razvoja programske opreme. Nekaj vrstic v programskem jeziku je zamenjalo tudi do več sto vrstic kode v zbirnem jeziku, kar je imelo več pozitivnih posledic: krajši čas razvoja, zmanjšanje stroškov, povečala se je razumljivost kode in s tem olajšalo vzdrževanje, razvoj programske opreme je postal zanimiv za veliko širšo množico ljudi. Na začetku so bili ti programi počasnejši od tistih v zbirnem jeziku, z razvojem prevajalnikov pa se je ta razlika zmanjšala. Na višjem nivoju napisani programi dandanes niso nujno počasnejši, saj so prevajalniki že tako dobri, da določene stvari naredijo (sistematično) bolje kot človek. Pomemben napredek programskih jezikov tretje generacije je tudi vsaj delna prenosljivost programske kode med različnimi računalniki, ali drugače rečeno neodvisnost programske kode od *izvajalnega okolja* (angl. execution platform).

Danes se strojnega in zbirnega jezika, vsaj pri razvoju klasičnih informacijskih rešitev, ne uporablja več. V veliki večini so ju nadomestili programski jeziki tretje generacije ter naprednejši pristopi. Vsi ti sodobni pristopi pa nakazujejo, da bodo tudi programski jeziki tretje generacije počasi izginili ali pa se jih vsaj ne bo uporabljalo v takšnem obsegu in na tak način kot sedaj.

Pomemben prispevek k izboljšanju razvoja programske opreme predstavljajo tudi *operacijski sistemi*. Operacijski sistem je nivo nad strojno opremo, ki ponuja standardne funkcije za dostop do strojne opreme. To je poenostavilo prevajalnike za programske jezike, saj tem ni bilo treba več generirati podrobne kode za upravljanje s strojno opremo, ampak le klice standardnih funkcij operacijskega sistema. S tem se je spet povečala neodvisnost od izvajalnega okolja.

Naslednja velika stopnja v razvoju je *objektno usmerjeni pristop*, ki uvaja precej drugačen način razmišljanja od klasičnega strukturiranega načina. Dobra objektno usmerjena koda je za razliko od strukturirane veliko bližje človeškemu načinu razmišljanja. Z objektno usmerjenimi jeziki so se širše uveljavili tudi tako imenovani *navidezni stroji* (angl. virtual machines). Rezultat prevajalnika se pri navideznih strojih imenuje *vmesna koda*. Ta ni več namenjena določenemu izvajalnemu okolju oziroma procesorju, ampak navideznemu stroju. Navideznih strojev pa je za določen jezik več, po eden za vsak podprt računalniški oziroma operacijski

sistem. Z uvedbo navideznih strojev je bil dosežen nov standardni nivo, podobno kot je bil s pojavom operacijskih sistemov dosežen standardni nivo nad strojno opremo.

Od objektno usmerjenega pristopa naprej se je pojavilo kar nekaj pomembnih in uporabnih idej na teme možnosti ponovne uporabe, organizacije programske kode obsežnih projektov, standardizacije prijemov in podobno: uporaba komponent, načrtovalski in drugi vzorci, porazdeljeno procesiranje, večnivojski modeli, ogrodja, napredna orodja in podobno. Pomembna je postala tudi *vmesna programska oprema* (angl. middleware). V splošnem se danes pri razvoju programske opreme teži k uporabi že izumljenih ter v praksi preizkušenih konceptov in vnaprej izdelanih komponent. Pisanje programov od začetka je le še stvar entuziastov, resna industrija si tega ne more več privoščiti, saj se kompleksnost programske opreme povečuje, čas in dovoljeni stroški razvoja pa stalno zmanjšujejo.

Vzporedno z razvojem zapisa, v katerem računalniku dajemo navodila, je računalniška znanost napredovala tudi v drugih vidikih. Tako smo iz začetnega *strojno usmerjenega obdobja* uporabe na strojno opremo vezanega strojnega in zbirnega jezika prešli na *aplikacijsko usmerjeno obdobje* z od strojne opreme neodvisnimi programskimi jeziki tretje generacije, operacijskimi sistemi in objektno usmerjenimi jeziki, kar je skupaj omogočilo razvoj širšega kroga obsežnejših aplikacij. Naslednje veliko obdobje, katerega konec se morda približuje, je *poslovno-celovito usmerjeno obdobje*, ki veliko manjših neodvisnih aplikacij zamenjuje s konceptom celovite povezane informacijske rešitve za celotno poslovanje. Pojavi se mnogo novih dobrih idej, kot so komponentno usmerjeni razvoj, vzorci, porazdeljeno procesiranje, vmesna programska oprema ter arhitekture in drugi koncepti za podporo izgradnji celovitih informacijskih rešitev.

Nekateri vidijo naslednje obdobje razvoja programske opreme v modelno vodenem razvoju v kombinaciji z ostalimi sodobnimi pristopi.

1.2 Modelno vodeni razvoj

Modelno vodeni razvoj (angl. model driven development) oziroma *razvoj programske opreme na podlagi modelov* je ena izmed novejših smernic razvoja programske opreme. V ožjem smislu modelno vodeni razvoj pomeni izdelavo modelov, iz katerih se posredno ali neposredno dobi končne programske rešitve, kot zamenjavo za pisanje klasične programske kode. V širšem smislu pa modelno vodeni razvoj zajema več podobnih filozofij, celovitih pristopov, dejanskih orodij in raznih drugih splošnih konceptov na temo razvoja na podlagi modelov. Obstaja več različic in definicij modelno vodenega razvoja, od katerih je najbolj priznana, standardizirana in celovita različica *modelno vodene arhitekture* [30], ki bo predstavljena v nadaljevanju, druge različice pa se od te razlikujejo predvsem po priporočenih vrstah modelov.

Za obravnavani pristop se pojavlja več imen, najprimernejše splošno ime za označbo pristopa se nam zdi že uporabljeni izraz *modelno vodeni razvoj*. Manj

pogosto se uporablja še izraz *modelno vodeni inženiring* (angl. model driven engineering) [29]. Izraz *modelno vodena arhitektura* (angl. model driven architecture) pa je oznaka za glavno različico modelno vodenega razvoja in je tako neprimeren izraz za splošno označevanje pristopa k razvoju programske opreme na podlagi modelov.

1.2.1 Standard MDA

Object Management Group, v nadaljevanju OMG [33], je organizacija, ustanovljena z namenom zmanjšanja kompleksnosti, stroškov in časa razvoja programske opreme. Za dosego teh ciljev trenutno priporočajo uporabo njihove *modelno vodene arhitekture* (angl. model driven architecture), v nadaljevanju MDA [30]. MDA je neke vrste krovni standard, ki precej na široko in morda malo premalo konkretno opisuje OMG vizijo modelno vodenega razvoja. To svojo rešitev vidijo tudi kot določen višji nivo nad vsemi nezdružljivostmi med različno strojno in programsko opremo. Ime modelno vodena arhitektura zaradi uporabe besede 'arhitektura' morda ni najbolj idealno, saj gre za veliko več kot klasično arhitekturo programske opreme.

V svoji viziji [23] so uvodoma opozorili na smernice na področju razvoja informacijske tehnologije: prehod iz samostojnih in precej izoliranih aplikacij in podatkov na porazdeljene sisteme, povečevanje obsega in kompleksnosti programskih rešitev in na razvoj novih tehnologij, ki imajo za posledico napredek in določeno porabo energije za njihovo osvojitev. Za rešitev problemov, ki jih prinašajo te smernice, ponujajo modelno vodeni pristop. Navajajo tudi povečano zanimanje za modeliranje in splošno za razvoj na višjih abstraktnih ravneh. Prednosti modelno vodenega razvoja vidijo na vseh področjih: enostavnost razvoja, predvsem prehoda na nove tehnologije, enostavnejše povezovanje med sistemi, vzdrževanje rešitev in testiranje. Modeliranje vidijo kot naslednjo stopnjo v evoluciji razvoja programske opreme, čeprav tudi programske jezike štejejo za neke vrste modele. MDA definira tri vrste pogledov oziroma modelov [22]:

- *procesno neodvisni model* (CIM – Computation Independent Model) je model, ki se osredotoča na okolje in zahteve sistema, neodvisno od konkretnih struktur in načina procesiranja; namenjen je predvsem premostitvi vrzeli med strokovnjaki obravnavane domene in strokovnjaki za razvoj programske opreme pri zajemu zahtev,

- *model, neodvisen od izvajalnega okolja* (PIM – Platform Independent Model) je model delovanja sistema brez podrobnosti o dejanskem izvajalnem okolju, najbolje opisan kot natančen, vseobsegajoč in dobro definiran model za navidezni računalnik oziroma izvajalno okolje,

- *model za izbrano izvajalno okolje* (PSM – Platform Specific Model) je model za izbrano izvajalno okolje (na primer za EJB), ki ga dobimo s preslikavo modela, neodvisnega od izvajalnega okolja.

MDA definira tudi preslikave modelov. *Preslikava ali transformacija modela* je funkcija, ki vhodni model z morebitnimi dodatnimi vhodnimi podatki preslika v izhodni model. Vhodni in izhodni model sta lahko v splošnem poljubna modela, v MDA pa je najpomembnejša preslikava iz PIM v PSM.

Poenostavljeno je možno idejo MDA povzeti takole: strokovnjak za zbiranje zahtev programske opreme iz informacij, ki mu jih da strokovnjak za obravnavano domeno, zgradi procesno neodvisni model. Ta model, ki je na preveč visokem nivoju in premalo tehnično dodelan, se naprej preslika oziroma preoblikuje v od ciljnega izvajalnega okolja neodvisni model. Tega pa se večinoma avtomatsko preslika v model za izbrano izvajalno okolje, na primer EJB. PSM je še vedno model, zato ga je koncu treba preslikati v enega izmed klasičnih programskih jezikov in rezultat preslikave prevesti. Preslikava iz PSM v programsko kodo je ponavadi precej enostavna, saj je PIM večinoma zelo blizu konceptom ciljnega programskega jezika.

1.2.2 Modeli

Definicij modela je več; ena izmed enostavnejših [17, str. 16] se glasi: "*Model* je opis (dela) sistema v dobro definiranem jeziku."Kako model odslikuje modeliran sistem, je odvisno od namena uporabe modela. To v kontekstu modelno vodenega razvoja pomeni, da mora model zajemati vse za razvijano programsko rešitev pomembne lastnosti sistema.

Modelov je več vrst; lahko jih ločimo glede na [17, str. 18–22] uporabljen modelni jezik oziroma tip diagrama, namen uporabe in podobno. Na temo modeliranja oziroma izdelave modelov za namene modelno vodenega razvoja je bilo že veliko napisanega. Gre za splošna pravila za vse vrste modelov o poimenovanju, vrstnih redih, modeliranju le za informacijski sistem zanimivih konceptov, organizaciji kompleksnih modelov v več podmodelov in podobno [11, pogl. 7]. Avtorji posvečajo več pozornosti diagramom za modeliranje statičnih vidikov sistema, zahtevnejši dinamični vidiki pa niso tako dobro obdelani.

1.2.3 Unified Modeling Language

Unified Modeling Language, v nadaljevanju UML, je jezik za vizualizacijo, specifikacijo, konstrukcijo in dokumentacijo programskih sistemov oziroma rešitev [5, str. 468]. UML je najbolj razširjen in poznan jezik za objektno usmerjeno modeliranje. Različni ljudje imajo različne poglede na njegovo uporabnost pri modelno vodenem razvoju.

MDA ga priporoča za PIM. Kot njegove prednosti v smislu uporabe za PIM modele se izpostavlja [11, str. 68–75] ločenost abstraktnega in dejanskega zapisa (angl. syntax), razširljivost, neodvisnost in njegovo standardiziranost[1], kot slabosti pa velik in slabo razdeljen meta model, slaba podpora pogledom (angl. viewpoints), zaostajanje s podporo za novostmi na področju razvoja programske

[1]UML je bil standardiziran s strani OMG.

opreme (vzorci, komponente in podobno), nedoločenost nekaterih osnovnih konceptov ter okornost uporabe standardnih možnosti razširjanja v praksi.

Spet drugi [17, str. 35–36] vidijo v njem več različnih uporab. Najprej kot navaden UML za PIM, precej primeren za modeliranje strukture sistema, manj za dinamičen vidik sistema. Ugotavljajo tudi, da zaradi zgoraj omenjenih slabosti ni tako primeren za generiranje PSM iz PIM. Drugo porabo pa vidijo v izvedljivem UML-ju, ki bo posebej obravnavan v nadaljevanju. Kot tretjo kombinacijo navajajo uporabo UML-ja z jezikom za določanje omejitev OCL (Object Constraint Language), ki dobro rešuje nekatere pomanjkljivosti UML-ja (za namene PIM modelov).

Za modelno vodeni razvoj je zelo pomembna možnost *razširjanja* UML-ja [11, str. 145–153]. UML se standardno razširi v UML *profil*. Za dejansko razširjanje UML-ja so na voljo trije mehanizmi. Prvi mehanizem je *stereotip* (angl. stereotype). Uporabiti ga je možno [9, str. 80] ob potrebi po novem konstruktu modelnega jezika, ki ga ni v standardnem UML-ju. Z njim je možno označiti poljuben element UML meta modela. Klasičen primer stereotipa je stereotip ≪*interface*≫, s katerim se v razrednih diagramih označuje razrede kot vmesnike. Drugi mehanizem razširjanja UML-ja so *zaznamki* (angl. tags), ki jih je možno dodajati stereotipom ali osnovnim elementom UML meta modela. To so poljubne dodatne lastnosti zaznamovanih elementov, na primer maksimalna dolžina dejanskega znakovnega polja. Tretji mehanizem razširjanja UML-ja so *omejitve* (angl. constraints). Omejitve dodajajo nov pomen ali spreminjajo obstoječa pravila [5, str. 82]. Omejitve so lahko poljubne in jih je možno dodati na poljubne elemente, lahko se nanašajo celo na več elementov skupaj. Tako lahko na primer določimo dovoljene vrednosti določenemu polju, zahtevamo urejenost ali celo določimo odvisnosti med posameznimi elementi. Kljub imenu 'omejitve' ne gre le za možnost omejevanja, ampak določanja poljubnega dejstva. Poljubnost je smiselna pri uporabi UML-ja za skiciranje, pri modelno vodenemu razvoju je treba podajati omejitve, ki jih razume preslikovalna funkcija na nižji nivo.

UML ni idealen modelni jezik, je pa res, da ni nobene bistveno boljše alternative za tako splošen jezik.

1.2.4 MDA in Meta Object Facility

Meta Object Facility, v nadaljevanju MOF, je standard za formalno definicijo modelov in meta modelov, razvit pod okriljem že omenjene organizacije OMG [11, str. 95–108].

Obsega štiri nivoje, z oznakami M0, M1, M2 in M3. Najvišji nivo M3 je meta meta nivo, namenjen definiranju meta modelov na nivoju M2. Meta modeli nivoja M2 so meta modeli v klasičnem pomenu besede in definirajo elemente modelov na nivoju M1. M1 modeli so modeli, ki jih dejansko največ uporabljamo pri modelno vodenem razvoju. Nivo M0 pa predstavlja dejanske primerke modela nivoja M1 s konkretnimi vrednostmi.

Dober primer za predstavitev teh nivojev je UML. Meta model UML z defini-

cijo diagramov in njihovih elementov je nivo M2, dejanski UML modeli so M1 nivo, na primer razredni diagram z razredom Oseba, M0 nivo pa dejanski objektni diagram s konkretnimi vrednostmi, na primer objektni diagram z objektom, ki opisuje dejansko osebo.

Obstaja več standardov za izmenjavo modelov in meta podatkov z MOF, najbolj poznan je XMI (XML Metadata Interchange) standard za izmenjavo meta podatkov v XML obliki [17, str. 133].

Uporaba standarda MOF pri modelno vodenem razvoju ni obvezna, večina resnih rešitev pa ga uporablja za standarden repozitorij za modele in celo preslikave. Zanimivo in poučno pa ga je spoznati že zaradi konceptov, ki jih uvaja, predvsem meta nivojev.

1.2.5 Izvedljivi UML

Izvedljivi UML (angl. executable UML) je kljub ozko usmerjenemu imenu ena izmed podrazličic modelno vodenega razvoja. Njegovi pristaši zagovarjajo idejo izvedljivih UML modelov [20]. Z izvedljivimi UML modeli predvsem mislijo na UML modele, ki so dovolj popolni oziroma nosijo dovolj informacije, da se jih da direktno interpretirati oziroma se iz njih da generirati končne programske rešitve. Pristop se ne osredotoča le na izvedljiv UML, podaja tudi smernice za oblikovanje procesa razvoja, po katerem se da priti do izvedljivih modelov. Izvedljivi UML opisuje sistem iz treh vidikov:

- *podatkovni vidik* – opisuje statično strukturo sistema, identificira razrede objektov in povezave med njimi; za to se uporabljajo UML razredni diagrami,
- *kontrolni vidik* – opisuje življenjski cikel razredov podatkovnega vidika, za to se uporabljajo UML diagrami stanj,
- *postopkovni vidik* oziroma vidik obnašanja (angl. behavior) – opisuje obnašanje oziroma akcije razredov pri prehodih med različnimi stanji; za ta vidik se pri izvedljivem UML pristopu uporablja *akcijski jezik* (angl. action language).

Ti trije modeli skupaj predstavljajo *procesno popolno* (angl. computationaly complete) specifikacijo sistema, skupaj s preslikavo oziroma prevajalnikom pa so tudi izvedljivi. Procesna celovitost je zelo pomembna, saj pove, da je možno celoten sistem opisati z modeli in iz teh modelov brez dodatnih podatkov generirati programsko opremo.

V praksi pa se z modeli dejansko opiše do 80, največ 90 odstotkov poslovne logike. Preostali del poslovne logike se seveda tudi da izraziti z izvedljivimi UML modeli, vendar se tega ponavadi v praksi ne dela, saj modeli izvedljivega UML-ja za opis nekaterih stvari niso tako primerni in je bolje preostali del napisati kar na klasičen način. Predvsem se problemi pojavljajo pri postopkovnem vidiku. Ta problem se pojavlja tudi pri klasični in drugih vrstah modelno vodenega razvoja.

Avtorji izvedljivega UML-ja imajo precej praktičen in zelo razdelan pristop k modelno vodenemu razvoju, s smernicami za pristop k modelno vodenem razvoju na splošno pa tudi podrobno za vse tri tipe modelov. Po tem se precej razlikujejo od organizacije OMG z različico MDA, ki zelo dobro in celovito pokriva vsa področja modelno vodenega razvoja, manj pa se posveča dejanskemu razvoju.

Morda najpomembnejša ugotovitev obravnavane različice modelno vodenega razvoja je postavitev modelov v središče pozornosti, pred preslikave na nižje abstraktne nivoje. Za razvijalce so najbolj pomembni izdelki, ki jih pri svojem delu dejansko razvijajo, tako imenovani *izdelki prvega reda* (angl. first class objects). To so pri modelno vodenem razvoju nedvomno modeli, podobno kot je za klasične razvijalce pomemben uporabljan programski jezik, ne zanima pa jih prav dosti kako prevajalnik njihovo programsko kodo prevede. Idejo izvedljivega UML-ja je seveda možno posplošiti na izvedljive modele.

1.2.6 Naklonjenost modelno vodenemu razvoju

Poznavalci so različno naklonjeni ideji modelno vodenega razvoja. Naklonjenost ideji modelno vodenega razvoja se razteza od gorečih zagovornikov do popolnih nasprotnikov.

Zagovorniki se večinoma strinjajo z osnovno idejo; razlike so bolj v zagovarjanju različnih pristopov in prijemov pri uporabi modelno vodenega razvoja.

Nasprotniki pa proti modelno vodenemu razvoju navajajo različne argumente, na primer okornost določenih konceptov, pomanjkanje dobrih celovitih orodij, nezmožnost opisa sistema v celoti, pomanjkanje standardizacije in podobno. Nekateri izmed teh argumentov so upravičeni, teža drugih pa z napredkom počasi izginja. Dejstvo je, da je modelno vodeni pristop v razvojni fazi in bo tako le čas pokazal vse njegove potenciale.

1.3 Sorodni sodobni pristopi, ideje in spoznanja

V tem poglavju bodo predstavljene druge sodobne smeri, ideje, tehnologije in spoznanja s področja razvoja programske opreme, ki so se pojavila po objektno usmerjenem pristopu in skupaj z modelno vodenim razvojem nakazujejo prihodnost razvoja programske opreme.

1.3.1 Dvigovanje nivoja abstrakcije in izguba namena

Pri vseh teh prehodih od strojnega jezika preko vmesnih faz do objektno usmerjenega pristopa oziroma modelno vodenega pristopa gre predvsem za približevanje načina podajanja ukazov računalniku v obliki, v kateri je problem najbolj idealno opisan. Za to približevanje se je z modelno vodenim razvojem širše prijel izraz *dvigovanje nivoja abstrakcije* (angl. raising the level of abstraction) [17, str. 8]. Računalniku navodil ne podajamo več v njemu najbolj prikladnem načinu, ampak na višjem, abstraktnejšem, idealnejšem nivoju. O tem prepadu med računalniku

razumljivim in med najbolj primernim zapisom je veliko govora tudi pri tovarnah programske opreme, kjer se uporablja izraz *abstraktni prepad* (angl. abstraction gap). Več o tem pri predstavitvi tovarn programske opreme.

Zaradi abstraktnega prepada oziroma pretvorbe idealnega zapisa v računalniku primeren zapis prihaja do tako imenovane *izgube namena* (angl. loss of intent) [13, str. 194–197]. Ta pretvorba pomeni zapis s primitivnejšimi elementi, predvsem pa zapis, kjer so osnovna visokonivojska dejstva porazdeljeno zapisana. Izguba namena pomeni izgubo osnovnega idealnega visokonivojskega zapisa zaradi porazdelitve zapisa teh dejstev na nižji nivo, ali drugače, pomeni, da je iz obsežne programske kode zelo težko razbrati kaj programska rešitev počne. Za ponazoritev izgube namena lahko na primer služi izjava 'aplikacija mora podpirati vnos in prikazovanje imena, naslova in telefonske številke strank', ki se v kodo klasičnega programskega jezika preslika porazdeljeno med veliko datotek izvorne kode. Pri objektnih jezikih bi tako dobili več razredov: za hrambo podatkov, grafični vmesnik, kontrolo izvajanja in podobno. Pri tej preslikavi se tako izgubi namen, saj iz programske kode ni več možno izluščiti originalne zahteve, zaradi katere je programska koda nastala. Podana izjava je precej preprosta; iz podane neobsežne programske rešitve bi se še dalo podati približno originalno zahtevo, a resnost problema se pokaže šele pri bolj obsežnih in zapletenih zahtevah.

Abstraktni prepad oziroma izguba namena s preoblikovanjem zapisa na računalniku primeren nivo skupaj tvorita jedro vseh problemov pri razvoju programske opreme. Težave povzročata v celotnem življenjskem ciklu razvoja, od analize do vzdrževanja in čez vse vidike razvoja, tudi pri testiranju, dokumentaciji in podobnem.

Pri obravnavanih temah se večkrat omenja težnjo k prehodu iz računalniku primernega zapisa k človeku primernemu zapisu. Treba je poudariti, da podana izjava ni najbolj pravilna oziroma relevantna. Res je, da je zapis v klasičnem programskem jeziku in primitivnejših oblikah človeku manj primeren, vendar je glavni cilj dvigovanja nivoja opisa doseganje vsebinsko najbolj točnega, razumljivega, kompaktnega in po ostalih drugih kriterijih najbolj idealnega zapisa, z namenom izničenja ali vsaj zmanjšanja izgube namena. Dejstvo, da je ta zapis bliže človeku, je zelo pomembno, ampak striktno gledano le posledica cilja doseganja idealnega zapisa.

1.3.2 Agilni pristop

Agilni pristop je oznaka za množico sodobnih pristopov k razvoju programske opreme, kot so ekstremno programiranje, SCRUM, adaptivni razvoj in podobni [19]. Hkrati pa je tudi zbirka enostavnih splošnih vrednot in principov na temo razvoja programske opreme. Nastal je kot krovni pristop za navedene pristope, pa tudi kot odziv na obširne in relativno zapletene procese razvoja programske opreme, ki so nastali v zadnjih desetih, dvajsetih letih. Avtorji v *agilnem manifestu* [19] navajajo naslednje vrednote:

- *posamezniki in interakcije* pred procesi in orodji,

- *delujoča programska oprema* pred obsežno dokumentacijo,
- *sodelovanje s stranko* pred pogodbenim pogajanjem,
- *odzivanje na spremembe* pred sledenjem planu.

Priporočajo tudi sledenje dvanajstim principom:

- zadovoljstvo strank z zgodnjo in pogosto dostavo programske opreme,
- sprejemanje sprememb tudi pozno v razvoju,
- pogosta dostava delujočih delnih programskih rešitev,
- spodbujanje pogostega sodelovanja in komunikacije vseh vpletenih v projekt,
- osredotočanje na ljudi kot individuume in grajenje projektov na njih,
- poudarjanje pomena komunikacije v živo,
- glavna mera napredka je delujoče programska oprema,
- zagotavljanje vzdržnega tempa razvoja,
- stalno stremenje k dobremu načrtovanju in tehnični odličnosti,
- težnja k enostavnosti,
- najboljše arhitekture, zahteve in načrti se porajajo v samoorganiziranih skupinah,
- periodično preverjanje in izboljševanje procesa razvoja.

Agilni manifest je res le preprosta zbirka napotkov, avtorji namenoma niso hoteli biti obširni, s čimer hočejo poudariti pomembnost teh osnovnih načel pred zapletenimi procesi.

Na temo agilnega pristopa je bilo napisane veliko literature, ki se ukvarja z uporabo agilnega pristopa v kombinaciji z raznimi drugimi idejami, bistvo pa še vedno ostaja v enostavnem in prilagodljivem procesu, ki sledi napotkom agilnega manifesta. Tako se agilnost dobro sklada tudi z modelno vodenim razvojem [12].

1.3.3 Domensko-specifični jeziki

Domensko-specifični jezik (angl. domain-specific language) je jezik, specifičen za določeno, ponavadi ožje definirano domeno [16]. Domensko-specifični jeziki se pojavljajo kot nasprotje oziroma dopolnitev *splošno-namenskih jezikov* [8]. Večinoma se uporabljajo kot dopolnitev pri opisovanju delov sistema, ki jih je s splošno-namenskimi jeziki težko elegantno opisati, lahko pa se uporabljajo tudi v večjem obsegu; v skrajnem primeru za opis celotnega problema. Primer domensko-specifičnega jezika so *regularni izrazi*.

Za modelno vodeni razvoj so še posebej zanimivi *domensko-specifični modelni jeziki*, ki se pri modelno vodenemu razvoju uporabljajo na podoben način kot splošno-namenski modelni jeziki. Možno je uporabiti standardne domensko-specifične jezike in morebitne že obstoječe preslikovalne funkcije zanje, lahko pa se razvije lasten jezik za specifično domeno in za ta jezik ustrezno preslikovalno funkcijo. Več o domensko-specifičnih jezikih bo povedanega pri ovrednotenju modelno vodenega razvoja.

1.3.4 Tovarne programske opreme

Tovarne programske opreme (angl. software factories) so eden najsodobnejših praktično uporabnih pristopov k razvoju programske opreme. Združujejo vse do sedaj znane dobre prakse, kot so modeliranje, vzorci, ogrodja, agilnost, domensko-specifični jeziki in podobno, v nov pristop, ki poudarja oblikovanje nekakšnih tovarn oziroma proizvodnih linij za izdelavo programske opreme določenega tipa.

Avtorji tovarn programske opreme so svojo vizijo zelo dobro predstavili v [13]. Knjiga je odlična predstavitev tovarn programske opreme, pa tudi vseh ostalih (uporabljenih) sodobnih pristopov k razvoju programske opreme. Razlago začnejo standardno z novimi okoliščinami in problemi razvoja programske opreme (povečanje kompleksnosti programske opreme, skrajševanje razvojnega časa, zmanjševanje stroškov). Sledi predstavitev grobe ideje tovarn programske opreme: sestavljanje aplikacij v tovarni programske opreme, kjer je tovarna programske opreme vnaprej pripravljen sistem vzorcev, modelov, ogrodij in orodij. Pristop ugotavlja, da obstajajo glede na razvoj različni tipi programske opreme in predlaga izdelavo oziroma sestavljanje posebnih tovarn programske opreme za posamezne tipe programske opreme. Pomembna je tudi ugotovitev, da se področje programske opreme, za razliko od drugih (tehničnih) področij, vztrajno upira industrializaciji, kar je tudi eden izmed glavnih ciljev tovarn programske opreme.

Pri razvoju programske opreme se uporablja proces *abstrakcije*. To je proces, v katerem se ohrani za določen kontekst pomembne informacije, nepomembne pa zavrže. Problemsko domeno se najprej opiše na idealnem visokem abstraktnem nivoju. Nato se iz tega opisa razvije končno programsko rešitev tako, da se ta visokonivojski zapis prepiše v računalniku razumljiv zapis na nižjem abstraktnem nivoju. Pristop opozarja na ta *abstraktni prepad* (angl. abstraction gap), na prepad oziroma vrzel med abstraktnim idealnim nivojem opisa in abstraktnim nivojem računalniku razumljivega opisa. Pristop ugotavlja, da se veliko časa in energije zgublja ravno s pretvorbo zapisa med tema dvema nivojema. Tovarne programske opreme, kot tudi drugi sodobni pristopi, vidijo rešitev problemov, povezanih z razvojem programske opreme, v dvigovanju nivoja abstrakcije. Seveda je napredek na področju razvoja programske opreme s postopnimi prehodi iz strojnega jezika na sedanje jezike abstraktni prepad že zelo zmanjšal.

Veliko problemov pri razvoju programske opreme izhaja iz kompleksnosti problemske domene in ob prehodu na nižji nivo abstrakcije posledično še večje kompleksnosti zapisa te domene na nižjem abstraktnem nivoju. Tovarne programske

opreme izpostavljajo problem kompleksnosti in poudarjajo pomembnost njenega obvladovanja. Poleg obvladovanja kompleksnosti so pomembne tudi spremembe in njihovo obvladovanje. Programska oprema se spreminja in s tem dejstvom se je v procesu razvoja treba sprijazniti in ga obvladovati.

Tovarne programske opreme kot največje probleme objektno usmerjenega pristopa navajajo pomanjkanje dobrih mehanizmov za oblikovanje ponovno uporabnih komponent in s tem preprečevanje monolitnih rešitev, ter da je za reševanje večine problemov objektno usmerjeni pristop (z glavnim konceptom razreda) preveč splošen. Večinoma se vsak projekt začne čisto na novo in zaradi slabih procesov večina projektov preseže časovne in denarne okvirje.

Pristop vpeljuje, kot jih sami imenujejo, *kritične inovacije*: sistematično ponovno uporabo, doseženo z *linijami programske opreme* za izdelavo med seboj podobnih programskih rešitev, razvoj s sestavljanjem različnih standardiziranih prilagodljivih komponent in z uporabo raznih standardov, modelno vodeni razvoj z visoko formaliziranimi modelnimi jeziki in procesna ogrodja za vodenje in nadzor projektov razvoja. Iz teh idej je potrebno razviti dobra ogrodja in orodja, vključiti stranke v proces razvoja in težiti k zajemu dejstev v obliki, ki je čimbolj primerna za generiranje končnih programskih izdelkov.

Tovarna programske opreme je definirana kot 'proizvodna' linija za izdelavo programske opreme, ki konfigurira razširljiva orodja, procese in vsebino z uporabo *predloge tovarne programske opreme* (angl. software factory template) osnovane na *shemi tovarne programske opreme* (angl. software factory schema) z namenom avtomatizacije razvoja in vzdrževanja različic arhetipskega izdelka s prilagajanjem, sestavljanjem in nastavitvami komponent, baziranih na ogrodju. Poenostavljeno povedano, ob identifikaciji določene vrste programskih izdelkov, kot je na primer programska oprema za upravljanje s strankami, se za njihov razvoj vzpostavi neke vrste tovarno, prilagojeno za izdelavo programskih rešitev te vrste. To tovarno se vzpostavi z izbiro in prilagoditvijo za to najbolj primernih orodij, ogrodij, procesov in drugih elementov.

Shema tovarne programske opreme definira izdelke (modeli, dokumenti, nastavitvene datoteke, izvorne datoteke, namestitveni opisi ...) in sredstva, s katerimi pridemo do njih. Izdelki in sredstva so ponavadi organizirani v tabelo, kjer ena dimenzija predstavlja vidike (angl. concern), druga pa nivoje abstrakcije. Posamezne celice te tabele tako predstavljajo različne poglede na problem in so hkrati osnova za razvoj določenega vidika programske rešitve. Shema tovarne programske opreme je le definicija na papirju.

Predloga tovarne programske opreme je konkretizacija oziroma implementacija sheme tovarne programske opreme. Dejansko so to koda v širšem pomenu besede in meta podatki, s katerimi ožje definiramo delovanje splošnih razširljivih orodij za razvoj dejanskih programskih rešitev. Definicija je precej ohlapna in ne omejuje natančno, kaj mora obsegati takšna predloga; širina je prepuščena oblikovalcem predloge. Tako se oblikuje tovarno oziroma produktno linijo za razvoj (ali bolje rečeno proizvodnjo) programskih rešitev določenega tipa. Posamezne programske rešitve se razvije po procesu z uporabo orodij, ogrodij in drugih elementov tovarne

programske opreme.

Tovarne programske opreme so, na kratko povzeto, v smiselno celoto povezani glavni sodobni pristopi in koncepti z dodatno idejo oblikovanja teh konceptov v okolja, primerna za razvoj določenih tipov programske opreme. Poučno jih je preučiti že zato, ker vključujejo večino pomembnih sodobnih pristopov.

1.4 Magistrska naloga

V tem razdelku so opisani cilji in pričakovani rezultati pričujoče magistrske naloge.

1.4.1 Cilji

Cilji, ki smo si jih zadali v okviru magistrskega dela, so naslednji:

- podrobna proučitev in ovrednotenje modelno vodenega pristopa in sorodnih konceptov, v katere je le-ta vpet,
- osvetlitev najboljših praks pristopa,
- predstavitev možnosti za izdelavo podpornih orodij,
- izdelava prototipov podpornih orodij za različne vidike obravnavanega pristopa, ter
- nadgradnja ideje modelno vodenega pristopa z definicijo prototipa visokonivojskega zapisa za opis problemskih domen in praktično izvedbo prevajanja tega zapisa v izvedljivo obliko.

1.4.2 Pričakovani rezultati

Kot je že veliko ljudi ugotovilo, znanost praviloma napreduje po zelo majhnih korakih, ki so ponavadi plod trdega dela in sodelovanja organiziranih skupin ljudi. Kot rezultate pričujočega dela tako, v kontekstu ravno navedene misli, navajamo:

- kritično ovrednotenje modelno vodenega pristopa, ter
- definicija prototipa visokonivojskega zapisa za opis problemskih domen in praktična izvedba prevajanja tega zapisa v izvedljivo obliko.

1.5 Zgradba nadaljnjega besedila

Nadaljevanje naloge poleg uvoda in zaključka obsega tri velike sklope oziroma poglavja, v katerih bomo poskušali doseči zastavljene cilje naloge.

Prvi sklop je splošna obravnava in ovrednotenje modelno vodenega in sorodnih pristopov. Začne se s prikazom tipičnega začetka uporabe modelno vodenega pristopa v praksi, z namenom orisa osnovnih značilnosti, prednosti in problemov

pristopa. Naprej sledi obravnava najbolj uporabljanega modelnega jezika UML z namenom ovrednotenja njegovih izraznih možnosti in s tem hkrati orisa izraznih možnosti grafičnih modelnih jezikov. Naprej so predstavljeni domensko-specifični jeziki kot alternativa oziroma dopolnitev UML-ju in drugim splošno-namenskim jezikom. Po teh začetnih obravnavah z namenom osnovnega orisa dela po modelno vodenem pristopu sledi obravnava jezikov na splošno, primerjava modelnih jezikov s klasičnimi programskimi jeziki ter opis novosti v klasičnih programskih jezikih, ki se v podobni obliki pojavljajo tudi v modelnih jezikih oziroma so pomembne z vidika preslikav pri modelno vodenem pristopu. Obravnava modelno vodenega pristopa se nadaljuje s podrobnejšo obravnavo modelov in preslikav. Sledi še nekaj misli o pomanjkanju teoretičnih rešitev, zaradi katerih modelni jeziki niso idealni. Sklop se zaključi s podajanjem bistva modelno vodenega pristopa ter kratkim pregledom glavnih ugotovitev.

Drugi sklop predstavlja dejanske pristope, koncepte in orodja, večinoma podkrepljene s primeri. Predstavitev se začne z opisom izdelave lastnih orodij za podporo modelno vodenemu pristopu od začetka ter se preko predstavitve vmesnih stopenj in možnosti konča z opisom celovitih razvijalskih okolij za podporo obravnavanemu pristopu. Namen tega sklopa je podroben pogled v delovanje naprednih orodij za podporo modelno vodenemu pristopu ter s tem razblinjanje mita magičnosti orodij, ki iz narisanih modelov z aktivacijo magičnega gumba naredijo izvedljivo programsko rešitev. Po pregledu pristopov, konceptov in orodij je podanih nekaj krajših opisov praktično izvedenih prototipov orodij za razne vidike modelno vodenega razvoja ter nekaj programskih rešitev, ki prikazujejo možnosti uporabe idej obravnavanega pristopa v okviru klasičnega razvoja. Sklop se zaključi s povzetkom ugotovitev in spoznanj pridobljenih skozi obravnavo.

Tretji sklop se nanaša na zadnji cilj, izdelavo visokonivojskega zapisa za opis problemskih domen ter orodja za njegovo izvajanje. Najprej so podana izhodišča, na katerih je zastavljena konceptualna ideja rešitve, imenovana *izvedljiva specifikacija*. Nato je podana praktična rešitev s predstavitvijo procesa načrtovanja izraznih možnosti zapisa ter izdelave orodja za izvedbo definiranega zapisa. Na koncu sklopa so podani predlogi za izboljšavo predstavljene rešitve.

Poglavje 2

Ovrednotenje modelno vodenega in sorodnih pristopov

V tem poglavju bo podrobneje predstavljen in ovrednoten modelno vodeni pristop k razvoju programske opreme ter sorodni sodobni pristopi. Poglavje se začne s predstavitvijo tipične začetne izkušnje dela z modelno vodenim pristopom, nadaljuje pa z obravnavo jezika UML, domensko-specifičnih modelnih jezikov, jezikov na splošno in klasičnih programskih jezikov ter njihovih novosti. Sledita podpoglavji, ki podrobneje obravnavata modele in preslikave. Poglavje se zaključi z množico ugotovitev o problemih modelno vodenega razvoja in razvoja na splošno, podajanjem bistva modelno vodenega pristopa in pregledom glavnih ugotovitev.

2.1 Začetni praktični poskus modelno vodenega razvoja

Za začetek obravnave je podan tipičen prvi poskus proučevanja in praktičnega preizkušanja obravnavanih tem. Ta razdelek podaja določene začetne ugotovitve in služi kot kontekst k nadaljnji vsebini. Namen te predstavitve je prikazati praktičen razvoj resnih programskih rešitev po modelno vodenem pristopu ter s tem njegove realne zmožnosti, ki precej odstopajo od začetnih predstav.

2.1.1 Pričakovanja in realnost novosti

Ob predpostavki poznavanja objektno usmerjenega pristopa in kompleksnih procesov razvoja izpred 10-ih let je že ob začetnem branju literature na temo modelno vodenega razvoja, agilnega pristopa, domensko specifičnih jezikov, tovarn programske opreme in podobnega, jasno, da gre za ideje, ki so že in še bodo precej izboljšale razvoj programske opreme v prihodnosti.

Na tem mestu bi radi opozorili na zelo zavajajoče trditve z nič ali malo resnice, ki se pogosto pojavljajo ob napovedih na temo modelno vodenega razvoja.

Povedano ne velja le za novosti na obravnavanem področju razvoja programske opreme, ampak precej splošno.

Vedno se govori, da je novost nekaj res revolucionarnega, kar bo popolnoma spremenilo trenutno stanje stvari. Na začetku ljudje ponavadi vidijo le prednosti novosti, šele ko jo začnejo uporabljati, spoznajo tudi slabe lastnosti le-te. Slabe lastnosti tako zavržejo, novih pa ne uporabljajo izključno, ampak jih dodajo starim, še vedno aktualnim in dobrim konceptom. V nadaljevanju tako ne bomo fanatično trdili, da bodo obravnavani pristopi zamenjali vse pred njimi.

Nekateri primitivni koncepti, na primer osnovne računske operacije, so še vedno zelo aktualni. Podpiral jih je že zbirni jezik, pa se še vedno najdejo v opisih poslovne logike na najvišjih nivojih.

Tako so predvsem oglaševalske trditve, da bodo sodobni pristopi v celoti zamenjali objektno usmerjen pristop, le malo verjetne. Objektni pristop ima veliko dobrih lastnosti in je eden izmed revolucionarnejših korakov v razvoju, ima pa tudi nekaj slabih lastnosti oziroma slabo pokritih področij, ki jih novejši pristopi bolje rešujejo. Tudi objektni pristop ni čisto na novo razvit pristop, ampak gradi na prejšnjih. Tako tudi novi pristopi ne bojo idealen odgovor na vse probleme prejšnjih pristopov.

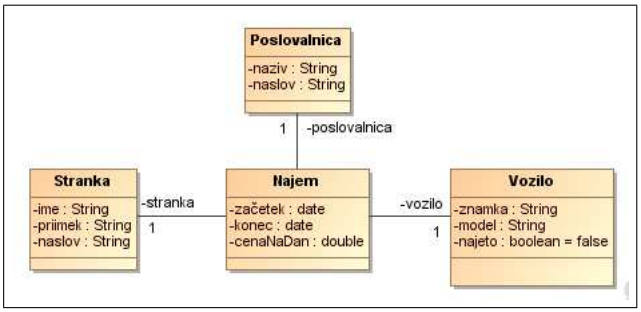
2.1.2 Prvi vtisi praktične uporabe

V tem razdelku bo predstavljen iz izkušenj povzet scenarij začetnih praktičnih poskusov modelno vodenega razvoja z namenom prikaza njegovih takoj vidnih prednosti in slabosti.

Izbranega načina seznanjanja s praktičnim delom po modelno vodenem pristopu ne bi označili kot pesimističnega, ampak kot realističnega. Ta skozi celotno delo prisotna kritičnost izhaja iz ocenjevanja realne uporabnosti modelno vodenega pristopa na realnih projektih, skozi razmišljanje o izvedbi v karieri že izvedenih dejanskih projektov z uporabo obravnavanega pristopa. Pri tem ocenjevanju pa ni smiselno imeti v mislih enostavnih projektov oziroma problemov, ampak obsežne in relativno zahtevne realne primere, s pravo mero posebnih nestandardnih zahtev, do katerih v realnosti prihaja[1]. Določena mera kritičnosti pa je tudi nujno potrebna, saj je le tako mogoče ob pozitivnih lastnostih pristopa odkriti tudi negativne in jih poskušati izboljšati.

Za praktični prikaz je bila izbrana poenostavljena problemska domena poslovanja podjetja, ki izposoja vozila. Uporaba modelno vodenega pristopa v praksi se ponavadi začne z risanjem razrednega UML ali podobnega diagrama, ki prikazuje osnovni statični vidik obravnavanega področja. Razredni diagram za izbrani primer izposoje vozil vsebuje razrede: Stranka, Vozilo, Najem, Poslovalnica in podobne, razredi imajo potrebne atribute ter so povezani z ustreznimi povezavami (glej sliko 2.1). Če je diagram narisan v enem izmed orodij za modelni razvoj, je

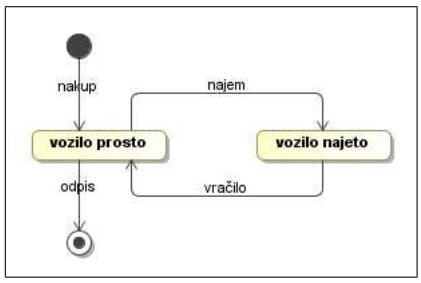
[1]Kot primer nestandardne zahteve lahko služi dodatna zahteva iz prakse v originalni neformalni obliki: 'Ko v sistem prispe nova zahteva, naj v kotu okna programa začne utripati majhen sonček.'



Slika 2.1: *Enostaven razredni diagram za domeno najema vozil.*

ponavadi možno že s pritiskom na gumb generirati kodo v izbranem programskem jeziku. Tako je možno veliko hitreje generirati kodo v primerjavi s klasičnim ročnim pisanjem. V naprednih orodjih ponavadi aktivacija 'magičnega' gumba generira še entitetni model in iz njega fizični podatkovni model, iz katerega se že naredijo ustrezni elementi v podatkovni bazi. Tako se da presenetljivo hitro in uspešno modelirati osnovni statični vidik programske rešitve.

Naprej je na vrsti razvoj zaslonske maske za vnos najema avtomobila. Postavlja se vprašanje, katere diagrame naj uporabimo. Morda je za obravnavani problem primeren diagram prehoda stanj, v katerem ima razred Vozilo stanji *vozilo prosto* in *vozilo najeto*, med katerima prehaja s prehodoma ob dogodkih *najem* in *vračilo* (glej sliko 2.2). Posledica dodajanja teh akcij so ponavadi metode v generiranih razredih iz razrednega diagrama, v katere pa je treba ročno dodajati programsko kodo, kar že ni več čisti modelno vodeni razvoj. Postavlja se vprašanje izdelave vnosne zaslonske maske najema vozila. Ali je treba dopolniti razredni diagram z razredi za predstavitev grafičnega vmesnika, z razredi *NalogZaslonskaMaska* in podobnimi? Ali ti razredi grafičnega vmesnika spadajo v poseben diagram? Klasično bi že omenjeni razred *NalogZaslonskaMaska* razširjal nek splošni razred *ZaslonskaMaska*. Nekatera orodja omogočajo uvoz razredov nastavljenega programskega jezika in je na tak način možno izdelati zaslonsko masko. Naprej je morda na vrsti razmišljanje o načinu obdelave vnešenih podatkov. Klasičen vzorec obdelovanja vnešenih podatkov je *model-pogled-krmilnik*



Slika 2.2: *Enostaven diagram prehajanja stanj za domeno najema vozil.*

(angl. model-view-controller). Spet se postavlja vprašanje ali je prava smer načrtovanje razrednih diagramov za vse razrede omenjenega vzorca, pa še kako narediti kreiranje primerka razreda najema in temu primerku nastaviti vnešene vrednosti. Ali bi bilo za to smiselno kar napisati kodo ali začeti morda risati sekvenčni diagram opisanega dogajanja, ki se mora zgoditi, ko uporabnik vnese vsa polja in pritisne na gumb za potrditev vnosa?

Morda ima uporabljeno orodje celo možnost podajanja zahteve za avtomatično oblikovanje standardnih operacij, kot je vnos in popravljanje ter prikazovanje podatkov. Take funkcionalnosti ponavadi generirajo določeno kodo, vendar je ta koda in njen rezultat precej standarden nabor operacij dodajanja, popravljanja in brisanja zapisov oziroma njihovih atributov na generičen način, kar je hitro premalo že pri enostavnih problemih.

Postavlja se vprašanje smiselnosti in izvedljivosti reševanja raznih kompleksnejših problemov z veliko bolj obsežno in zapleteno poslovno logiko. Še bolj zanimivo vprašanje je, kako bi z osnovnimi modeli in dodatnimi ozko usmerjenimi funkcionalnostmi razvijali rešitve izven klasične domene podpore poslovanju podjetij, kot na primer obračunske sisteme ali računalniške igre.

2.1.3 Analiza praktične uporabe

V tem razdelku bodo podane ugotovitve analize v prejšnjem razdelku opisanega začetnega poskusa dela po modelno vodenem razvoju.

Pri statičnem vidiku ponavadi ni težav, delo je hitrejše, razredni UML in podobni diagrami delujejo kot zelo primerni za opis tega vidika. Prehod od pisanja razredov v programskem jeziku in doseganja enakega rezultata z risanjem diagramov je definitivno napredek, ampak le s statičnega vidika, brez določanja vsebine metod.

Naprej pa se stvari ponavadi začnejo zapletati. Težko je vedeti prav kako naj pravilno nadaljujemo. Delno je možno težave pripisati pomanjkanju vedenja o novem pristopu, tako kot pri prehodu na objektno usmerjeni način na začetku ni bilo prav jasno kaj naj počnemo z objekti oziroma razredi. Dober primer razvoja po modelno vodenem razvoju je rešitev za podporo poslovanju dostave zajtrkov[2] v [17].

Večina težav pa izhaja iz dejstva, da ljudje ob srečanju z modelno vodenim razvojem pričakujejo, da bojo risali modele, ki se jih bo potem dalo preslikati v izvedljiv rezultat. Dejansko pričakujejo, da bojo vso poslovno logiko ali vsaj obljubljenih 80 odstotkov logike pokrili z modeliranjem, preostanek pa napisali v klasičnem programskem jeziku. Vendar ne najdejo diagramov, s katerimi bi elegantno opisali predvsem dinamičen vidik problemske domene. Izkaže se, da ne znajo niti dobro odgovoriti, kakšni bi ti diagrami morali biti.

Dejstvo je, da diagrami, s katerimi bi na prikladen način lahko modelirali različne vidike problemske domene, ne obstajajo, in jih tudi še ne bo tako kmalu. S

[2] Orignalno Rosa's Breakfast Service.

tem se razbije glavna iluzija modeliranja, začetni občutek, da se namesto dolgotrajnega programiranja probleme sedaj reši hitro z nekaj modeli.

Na vprašanja, ki so se pojavila v podanem primeru, bomo poskušali odgovoriti v nadaljevanju, hkrati z obravnavo omenjenih teoretičnih problemov.

2.2 Vidiki obravnave

Pred začetkom analize modelno vodenega razvoja je treba poudariti dva vidika gledanja na modelno vodeni razvoj, ki se ju je pri proučevanju modelno vodenega pristopa treba zavedati in striktno ločevati.

Prvi vidik je vidik razvijalca programske opreme z zrelo, do sedaj še ne doseženo, stopnjo modelno vodenega pristopa. S tega vidika so pomembni le koncepti modelno vodenega pristopa, ki jih razvijalec uporablja v procesu izdelave programske rešitve. Za razvijalca so pomembni modelni jeziki s katerimi modelira rešitev, smernice njihove uporabe, orodja za modeliranje in podobno. Tu pa se zanimanje razvijalca programskih rešitev konča. Ne zanima ga, kako se izdelani modeli prevedejo na nižji nivo, in razne druge podrobnosti delovanja orodij in konceptov, skritih v njih. Takšno razmišljanje je popolnoma logično in pravilno, saj je osnovna prednost modelno vodenega razvoja ravno dvig na višjo abstraktno raven. Pravilnost takšnega vidika potrjuje tudi pogled nazaj, na prehod iz zbirnega jezika na programske jezike tretje generacije. V slednjih razvijalci danes uspešno razvijajo programske jezike tudi popolnoma brez znanja zbirnega jezika; potrebujejo le prevajalnik.

Drugi vidik pa obravnava modelno vodeni razvoj kot celoto in zagotavlja modelne jezike, preslikave, smernice, orodja in vse ostalo, potrebno za omogočanje prvega vidika, torej razvoja po modelno vodenem pristopu. V ta drugi vidik sodijo raziskave in inovacije na tem področju, še posebej v sedanjem času, ko se modelno vodeni pristop šele razvija. Tu se je treba zavedati, da je cilj tega celovitega vidika oblikovanje prvega vidika, katerega glavni cilj je čim bolj olajšati delo razvijalcu kot uporabniku rezultata dela tega drugega vidika.

V tem vmesnem obdobju, ko se modelno vodeni pristop še razvija in tako še ni na voljo popolnih rešitev za razvoj z modeliranjem brez pisanja klasične programske kode, delo izključno v smislu prvega vidika seveda ni možno. Tako je za dele, za katere ne obstaja standardna podpora za modelno vodeni razvoj, mogoče ubrati dve poti:

- razvoj teh delov s klasičnimi jeziki oziroma orodji, ali
- izdelava modelno vodene podpore za manjkajoče domene ter uporaba te podpore za izdelavo teh delov.

V okviru celovitega vidika, torej oblikovanja rešitev za osnovni razvojni vidik, opozarjamo na dva načina razmišljanja. Razmišljamo lahko, kako koncepte na nižjem nivoju sestavljati v kompleksnejše koncepte na višjem abstraktnem nivoju.

Veliko bolj produktivno in hkrati težje pa je razmišljati ravno obratno, torej oblikovati idealne koncepte na višjem abstraktnem nivoju in potem poskušati najti dobro prevedbo teh konceptov na nižje nivoje. Bolje je razmišljati o idealnem diagramu za opis nečesa, kot pa za opis uporabiti že obstoječ, precej neprimeren diagram. To pravilo velja tudi na splošno, za vse kreativne procese; ugotavljanje kaj želimo doseči, in nato iskanje sredstev za izvedbo te ideje, večinoma daje veliko boljše rezultate kot pa izdelava rešitve na podlagi osnovnih trenutno dosegljivih sredstev.

2.3 Uporabnost jezika UML kot modelnega jezika

Že v podanem primeru v razdelku 2.1 je bilo pokazano, da današnji jeziki za modeliranje še niso idealni za modeliranje vseh vidikov problemskih domen. Ugotovljeno je bilo, da za nekatere, predvsem statične, vidike obstajajo dobri modelni jeziki, do težav pa pride pri modeliranju obnašanja oziroma dinamičnih vidikov sistema. V nadaljevanju bo predstavljen jezik UML kot najširše uporabljen modelni jezik in osnova modelno vodenega pristopa po MDA različici.

UML je splošno-namenski modelni jezik, ki je nastal nekako skupaj z objektno usmerjenim pristopom. V uvodnem razdelku 1.2.3 so bile, v kontekstu uporabe pri modelno vodenem razvoju, že navedene njegove prednosti in slabosti ter standardne možnosti razširjanja. Na praktičnem primeru[3] pa je bilo ugotovljeno, da so njegovi razredni diagrami precej primerni za modeliranje osnovnih statičnih vidikov problemske domene, za druge vidike pa ga je malo manj trivialno uporabiti.

V sledečih razdelkih bodo obravnavani posamezni UML diagrami in podane ocene možnosti oziroma prikladnosti njihove uporabe kot modelov pri modelno vodenem razvoju. Za razumevanje sledečih obravnav UML diagramov je predpostavljeno njihovo dobro poznavanje. Obsežna razlaga UML diagramov na tem mestu ne bi bila primerna; na to temo obstaja obilo literature [9].

2.3.1 Diagrami primerov uporabe

Primer uporabe je množica *scenarijev*, ki skupaj opisujejo določen cilj uporabnika, kjer je scenarij zaporedje korakov, ki opisujejo interakcijo med uporabnikom in sistemom.

Diagrame primerov uporabe se da za modelno vodeni razvoj tako uporabiti za modeliranje ravno opisane interakcije med uporabnikom in sistemom na visokem nivoju. Vendar ti diagrami sami po sebi ne vsebujejo niti približno dovolj informacij za generiranje česa konkretnega. Natančneje rečeno, diagrami vsebujejo podrobne informacije, vendar so podrobne specifikacije posameznih primerov uporabe oziroma scenarijev zapisane v naravnem jeziku, kar je premalo formalizirana oblika za vhod v preslikavo. Zato so potrebni podrobnejši opisi sistema.

[3] Glej razdelek 2.1.

Praktična uporabnost diagramov primerov uporabe je tako smiselna le v navezi z drugimi diagrami, s čimer pa se postavlja vprašanje smiselnosti izdelave diagramov primerov uporabe v ožjem smislu uporabe v modelno vodenem razvoju.

To vprašanje se da posplošiti na vprašanje idealnega opisa sistema, torej, ali je smiselno opis podati na več nivojih od najvišjega zelo skrčenega opisa do najnižjega, katerega je možno uporabiti v modelno vodenem pristopu, ali je opis smiselno podati le na tem najnižjem nivoju, direktno uporabnem za generiranje rešitve. Za odgovor na omenjeno vprašanje pa je treba najprej odgovoriti na vprašanje smiselnosti podajanja dejstev v modelih, ki niso direktno uporabna pri modeliranju. Ta vprašanja so precej ekvivalentna smiselnosti komentiranja klasične programske kode, s tem da je tu treba upoštevati ugotovitve principa izgube namena.

Obravnava diagramov primerov uporabe je tako pokazala pomembno dejstvo, da niso kar vsi diagrami uporabni v modelno vodenem razvoju, vsaj v celoti ne. Tako mora biti modeliranje sistema za namene modelno vodenega razvoja premišljeno, po določenih smernicah, ki pripeljejo do modelov, ki nosijo prave informacije za preslikavo v popolno izvedljivo rešitev. Podobno kot programski jeziki narekujejo ukaze, s sestavljanjem katerih pridemo do smiselno delujoče programske rešitve, je tudi tu treba poznati, kaj posamezen element modela pomeni za končno rešitev oziroma, kako podati določeno dejstvo domene na predpisan pravilen način.

Diagrame primerov uporabe se v modelno vodenem pristopu lahko uporabi kot osnovo za generiranje testov. Diagram primera uporabe opisuje vhodne zahteve uporabnika do sistema in pričakovane izhode. Problem je le, da ti vhodi in izhodi niso podani z dejanskimi vrednostmi. Dejanske vrednosti se da dodatno specificirati, vsaj delno pa se jih da avtomatsko generirati glede na njihov tip in druge lastnosti, ki jih je morda možno pridobiti iz diagramov primerov uporabe.

2.3.2 Razredni diagrami

Razredni diagrami so pri modelno vodenem razvoju najbolj neposredno in intuitivno uporabni. Z njimi na zelo idealen način predstavimo statični vidik sistema z osnovnimi koncepti in povezavami med njimi. Razredni diagram prikazuje *razrede* kot osnovne statične elemente abstrakcije problemske domene ter *asociacije* med njimi. Razredi imajo določene *atribute* in *operacije*.

V osnovi se razredni diagrami pri modelnem pristopu uporabljajo enako kot pri objektnem pristopu. Z abstrakcijo problemske domene identificiramo razrede z atributi in operacijami, ter asociacije med njimi. Pri tem upoštevamo vse dobre prakse, ki jih pozna že objektni pristop: pravilna identifikacija za reševani problem relevantnih konceptov, njihovo pravilno poimenovanje, uporaba raznih preverjanj za potrditev pravilnosti diagrama in podobno.

Za namene procesne popolnosti modelov je treba osnovne razredne diagrame ponavadi dopolniti še z raznimi informacijami. Posamezne različice modelnega pristopa predlagajo različne razširitve. Večina različic predlaga *jezik za določanje*

omejitev OCL (Object Constraint Language), s katerim je možno podati razne omejitve, ki izhajajo iz obravnavane domene, kot na primer enoličnost ISBN številke pri knjigah. Izvedljivi UML [20], predstavljen v uvodnem razdelku 1.2.5, poleg OCL predlaga še *akcijski jezik* (angl. action language) za izražanje obnašanja.

2.3.3 Diagrami prehajanja stanj

Ena prvih izbir za modeliranje dinamičnih vidikov pri modelno vodenem razvoju so ponavadi *diagrami prehajanja stanj* ali krajše kar diagrami stanj. Diagrami stanj prikazujejo *stanja* določenega razreda, možne *dogodke*, ki povzročijo *prehode* med stanji, ter *aktivnosti* oziroma *operacije*, ki se zgodijo ob prehodih.

Diagrami stanj so, vsaj izven konteksta modelno vodenega razvoja, zelo uporabni za prikaz možnih stanj razreda in prehodov med njimi. Za namene modelno vodenega pristopa se jih sicer da uporabiti, hitro pa je mogoče ugotoviti, da so za nekatere probleme oziroma razrede stanja in prehodi zelo pomembni, za druge manj, nekateri pa stanj v okviru izbrane abstrakcije sploh nimajo.

Podan primer izposoje vozil z razredom Vozilo, ki je izposojeno ali pa ne, in ima tako stanji *vozilo izposojeno* in *vozilo prosto*, med katerima prehaja s prehodoma *najem* in *vračilo* (glej sliko 2.2 na strani 21). Diagram stanj za tak razred definira izposojo in vračanje vozil, kar je ključno za obravnavano domeno. Iz primera sledi občutek, da diagrami stanj že kar določajo glavne enote dogajanja problemske domene in hkrati tudi strukturo rešitve. V podanem primeru izvedba oziroma modeliranje aktivnosti ob izposoji in vračanju že pripelje bližje h končni programski rešitvi. Vprašanje je, ali to velja za vse probleme in njihove diagrame stanj.

Sledi obravnava primera informacijske podpore trgovini, ki nabavlja in prodaja izdelke. Razredni diagram definitivno vsebuje razred Izdelek. Zanj si je sicer možno zamisliti in narisati diagram prehajanja stanj s stanji *nabavljen* in *prodan* ali kaj podobnega. Ampak hitro je možno ugotoviti, da razmišljanje o stanjih izdelka ne spada v okvir primerne abstrakcije obravnavanega problema. Trgovca glede izdelka zanima količina na zalogi, prodana količina in razne druge ekonomske kategorije poslovanja. Ne sprašuje se, kakšno je stanje posameznega izdelka. Morda podpora poslovanju trgovine zahteva vpeljavo razreda Pošiljka s stanji *pripravljena*, *pregledana*, *oddana* ali podobno, ampak to še vedno ne spremeni dejstva, da v danem kontekstu za razred Izdelek ni smiselno razmišljati o stanjih. Risanje diagramov za stvari izven konteksta ni samo nesmiselno, ampak tudi precej zavajajoče, ker lahko vodi do načrtovalsko slabih rešitev, če sploh pripelje do uporabne programske rešitve.

France in sod. [10, str. 63] natančneje ugotavljajo uporabnost diagramov prehajanja stanj predvsem v vzročnih sistemih in hkrati navajajo nevarnost povečanja *posledične kompleksnosti* ob uporabi teh diagramov pri drugih tipih sistemov. Posledična kompleksnost pri programski opremi [13, str. 36] je kompleksnost, ki se ob osnovni kompleksnosti sistema pojavi kot posledica opisa rešitve na nižjem

abstraktnem nivoju.

Tako je možno ugotoviti, da so diagrami prehajanja stanj uporabni za modeliranje konceptov, ki imajo že po svoji naravi določena stanja, med katerimi prehajajo in se ob teh prehodih nekaj zgodi, ob predpostavki, da je modelirano dogajanje relevantno za obravnavani problem. Trditev s stališča teorije deluje trivialno, s primerom razreda Izdelek v kontekstu problemske domene trgovine pa je bilo videti, da se je je v praksi težko zavedati in izogibati.

V literaturi je možno najti precej dobrih primerov analize problemskega področja in iz nje izpeljanih diagramov prehajanja stanj, ki zelo dobro opisujejo oziroma modelirajo dano problemsko področje [20, pogl. 9.2.1]. Večinoma pa so opisovani primeri med najprimernejšimi za opis z diagrami prehajanja stanj, manj pa literatura govori o manj primernih primerih.

Po izdelavi upravičenega in smiselnega diagrama prehajanja stanj, je naslednji korak proti cilju procesne celovitosti modeliranje akcij, ki so bile določene posameznim stanjem ob prehodih.

2.3.4 Diagrami interakcije

Naslednji na prvi pogled zelo prikladni diagrami so *diagrami interakcije* (angl. interaction diagrams), kjer poznamo *sekvenčne diagrame* (angl. sequence diagrams) in *diagrame sodelovanja* (angl. collaboration diagrams). Obe vrsti diagramov interakcije prikazujeta interakcijo med objekti preko sporočil skozi čas, med seboj pa se razlikujeta le v vizualni obliki podajanja te interakcije.

Kljub navidezni prikladnosti za predstavitev obnašanja diagrami interakcije niso posebno primerni za precizno definicijo obnašanja, kar ugotavljajo tako avtorji UML-ja [9, str. 77], kot izvedljivega UML-ja [20, pogl. 10.3.5]. Izvedljivi UML tako za predstavitev obnašanja vztraja pri diagramih stanj z definicijo aktivnosti ob prehodih v stanja z *akcijskim jezikom*.

Neprimernost interakcijskih diagramov se je pokazala že pred modelno vodenim razvojem pri orodjih, ki so poskušala uvesti razvoj, v katerem bi lahko programske rešitve razvijali v programski kodi ali z modeliranjem, z možnostjo poljubnega prehajanja med obema načinoma. Iz izkušenj z orodjema Together [34] in Rational Rose [35] lahko povemo, da so iz programske kode dobljeni sekvenčni diagrami precej nepregledni že na manjših testnih razredih in njihovih metodah. O problemu podobnih prehodov med modeli na različnih abstraktnih nivojih bo govora pozneje.

Drugi dober pokazatelj neprimernosti diagramov interakcije za podrobno modeliranje je poskus modeliranja določenega obnašanja, 'programiranja' z njimi. Risanje takšnih diagramov je zelo nerodno. Diagrami hitro postanejo nepregledni in preveliki za zaslon.

2.3.5 Ostali UML diagrami

Diagrami aktivnosti (angl. activity diagrams) prikazujejo zaporedje aktivnosti v določenem dogajanju. Izven konteksta modelno vodenega razvoja so kar uporabni,

še posebej v smislu spodbujanja modeliranja paralelnega dogajanja. Za modelno vodeni razvoj pa niso tako primerni, ker nimajo jasne povezave med akcijami in objekti. Drugače rečeno, težko je za dejanski primer narisati diagram aktivnosti in povedati, kaj naj bi se iz njega generiralo. Za uporabo v modelno vodenem razvoju tako ti diagrami niso dovolj eksaktni.

Komponentni diagrami (angl. component diagrams) prikazujejo komponente sistema in odvisnosti med njimi. Pri modelnem razvoju bi se lahko uporabljali za oblikovanje komponent. V resnici pa se vsaj zaenkrat za grupiranje večinoma uporablja koncept *paketov* (angl. package). Verjetno zato, ker je koncept paketov uporabljen tudi v programskih jezikih, v katere se modeli ponavadi preslikajo.

Namestitveni diagrami (angl. deployment diagrams) prikazujejo strojne in programske komponente sistema ter povezave med njimi. V kontekstu modelno vodenega pristopa se jih da uporabiti za prikaz zamišljenega delovanja celotnega sistema s strojno in programsko opremo. Z ustreznimi orodji in profili posameznih namestitev pa se da postopek od izdelanih modelov do nameščene rešitve popolnoma avtomatizirati.

2.3.6 Novejše različice jezika UML

Leta 2003 je OMG izdal različico UML-ja 2.0. Od takrat naprej pa izdaja razne dopolnitve in popravke [10, 25]. UML 2.0 je nastal kot odziv na probleme 1.x različic in podrazličic. Različice 2.x sicer prinašajo določene izboljšave, niso pa končna rešitev za modelno vodeni razvoj.

Tako je UML 2.x predstavljen kot družina splošno-namenskih jezikov z izboljšanimi izraznimi zmožnostmi, kot pomembno orodje za modeliranje obnašanja sistema pa poudarjajo *akcijsko semantiko*. UML 2.0 definira tudi štiri poglede, ki opisujejo štiri vidike sistema: statična struktura, interakcija, aktivnosti in stanja. Ti štirje pogledi uporabljajo ustrezne diagrame, med seboj pa se prepletajo, v smislu nastopanja nekaterih elementov diagramov v dejanskih diagramih različnih pogledov.

UML 2.x uvaja *strukturiran klasifikator* (angl. structured classifier) kot nadkoncept, ki združuje več konceptov v enega samega, na primer razred Avto, ki združuje več instanc razreda Kolo, povezanih z razredom Motor. Uvaja tudi *vmesnik* (angl. port), s katerim je mogoče strukturiranemu klasifikatorju določiti točke interakcije struktruriranega klasifikatorja. Opisana koncepta omogočata predvsem lažjo uporabo elementov v različnih diagramih.

Različica 2.x UML-ja je za modeliranje obnašanja (angl. behavior) posvojila *akcijsko semantiko* (angl. action semantics). Akcijska semantika je precej vzporedna konceptom iz programskih jezikov, definira na primer akcije za kreiranje objektov, nastavljanje vrednosti njihovih atributov in klicanje metod teh objektov, in tako ne pripomore mnogo k dvigu nivoja abstrakcije. Ta ugotovitev ni mišljena kot kritika, saj je vprašanje koliko obnašanja je sploh možno podati na višjem abstraktnem nivoju. Modeliranje obnašanja sistema je eno izmed manj raziskanih področij, ali drugače rečeno na področju dviga nivoja abstrakcije obnašanja sis-

tema do sedaj ni bilo bistvenega napredka. Akcijska semantika je definirana le na abstraktnem nivoju, brez konkretnega zapisa (angl. concrete syntax). Na podlagi te abstraktne definicije se je kasneje neodvisno razvilo nekaj konkretnih zapisov. Tako je uporabnost akcijske semantike pri modelnem pristopu nekoliko boljša od diagramov interakcije le zaradi dejstva, da si lahko oziroma moramo sami oblikovati bolj primeren konkreten zapis.

UML 2.x poskuša tudi izboljšati uporabnost interakcijskih diagramov z razbijanjem interakcije na fragmente in uvajanjem *preglednega diagrama interakcij* (angl. interaction overview diagram). Kljub napredku pa so ti diagrami še vedno približno enako uporabni kot pri prejšnjih različicah. Velike izboljšave so bile storjene tudi pri diagramih aktivnosti, ki so sedaj veliko bolje definirani.

Novejše različice UML-ja sicer odpravljajo določene drobne pomanjkljivosti in nedefiniranosti s stališča pomena posameznih podrobnosti modelov, ki so v prejšnjih različicah delale probleme. Na primer z uvajanjem strukturiranih klasifikatorjev oziroma vmesnikov. Še vedno pa v določenih primerih ni nedvoumnih smernic, kako nekaj modelirati.

Največ kritik UML požanje na račun svoje kompleksnosti. Specifikacija UML-ja v zadnji različici 2.1.2, razdeljena na specifikacijo superstrukture, infrastrukture, izmenjave modelov in jezika OCL za podajanje omejitev, obsega čez 1200 strani, specifikacija akcijske semantike pa še preko 700 strani. Avtorji UML-ja sicer priporočajo uporabo podmnožice celotne specifikacije, oblikovanje podmnožice pa je zelo zahtevno, saj zahteva precej natančno poznavanje večine UML-ja, zaradi odvisnosti pa morajo takšne podmnožice vsebovati kar veliko različnih paketov. Obsežnost in kompleksnost UML-ja je hitro opazna tudi pri delu z UML meta modelom.

Tako lahko zaključimo, da tudi novejše različice UML-ja ne prinašajo revolucionarnih izboljšav za modelno vodeni razvoj. Kot največji izboljšavi različic 2.x bi lahko navedli abstraktno semantiko, ki je na žalost brez konkretnega zapisa, ter razne manjše izboljšave v smislu boljše definiranosti.

2.3.7 Jezik OCL za podajanje omejitev, poizvedb in postopkov

Jezik OCL (Object Constraint Language) je bil v začetni različici pravilno poimenovan kot *jezik za podajanje omejitev* v UML diagramih [31]. V novejših različicah pa je bil OCL precej razširjen z možnostmi, ki presegajo podajanje omejitev začetne različice. Po novem je z njim možno dobro izraziti tudi poizvedbe ter omejeno še postopke oziroma obnašanje (angl. behavior). Tako njegovo originalno ime ne opisuje več dobro vseh njegovih zmožnosti, po novem bi se moral imenovati *jezik za podajanje omejitev in poizvedb*, v ne najbolj idealnem smislu pa še *jezik za podajanje splošnih postopkov*.

Njegove dobre lastnosti so, poleg ravno omenjene možnosti specifikacije omejitev, poizvedb in drugih splošnejših postopkov tudi njegova matematična definicija skupaj s prikladnejšim nematematičnim zapisom, močna tipiziranost (angl. strongly typed) in deklarativnost.

OCL izrazi se vedno nanašajo na določen gradnik UML diagramov. Ta gradnik se v OCL terminologiji imenuje *kontekst* (angl. context). OCL izrazi so lahko dodani posameznim gradnikom UML-ja, lahko pa so podani v posebni tekstovni datoteki. Pri podajanju OCL izrazov na UML gradnikih samih, kontekst predstavlja že gradnik sam, pri tekstovnem zapisu pa je kontekst naveden z imenom pred izrazom.

OCL doprinese k dvigovanju nivoja abstrakcije v mnogih pogledih. Še najbolj s svojo deklarativnostjo, kjer kot deklarativen jezik specificira, kaj se mora zgoditi, ne pa natančno kako. Konkretneje pa, na primer, s pred/po-pogoji in invariantami[4]. Njegova močna točka je tudi možnost poizvedovanja, kjer ob deklarativnosti še posebej izstopa kompaktnost zapisa poizvedb. Z OCL omejitvami se da razrešiti veliko dvoumnosti, ki nastanejo zaradi omejenih izraznih zmožnosti UML diagramov, na primer pri razredih in asociacijah med njimi ter s tem povezanimi problemi, kot so dinamična številnost, cikli in podobno.

Nekateri OCL izrazi in sam njihov način podajanja pa so precej podobni programski kodi in že na prvi pogled delujejo kot programska koda, napisana v malo drugačni obliki. V takem primeru je kontekst večinoma kar UML razred ali njegova metoda, sam OCL izraz pa sploh ni deklarativen, ampak se bere podobno kot programska koda.

Med tema dvema tipoma izrazov obstaja še nekakšna vmesna skupina OCL izrazov, ki jih je že mogoče zaslediti v programskih jezikih oziroma se jih da uporabiti preko raznih dodatnih programskih knjižnic. Sem sodi na primer podpora delu z zbirkami (angl. collections) in razne napredne oblike zank, kot je *za-vsak-element* (angl. for-each).

Z OCL se dvig nivoja abstrakcije še najbolj pozna pri razrednih UML diagramih, kar je vidno že skozi dosedanjo obravnavo OCL-ja.

Pri diagramih za modeliranje postopkov, kot so interakcijski diagrami, diagrami stanj in aktivnostni diagrami se OCL uporablja predvsem za opisovanje postopkov določenih elementov teh diagramov, na primer za določanje postopka operacij diagramov interakcij. V tem smislu je uporaba OCL za enostavne operacije, ki jih je moč opisati s pred/po-pogoji in invariantami dobra rešitev, pri kompleksnejših postopkih pa se izrodi v zgoraj opisano zamenjavo za programsko kodo, ki ne doprinese kaj dosti k dvigu abstraktnega nivoja. Je pa ob tej kritiki treba povedati, da za opis kompleksnih postopkov vsaj konceptualno ni boljših alternativ. Opremljanje diagramov z OCL definicijami naredi diagrame še obsežnejše in manj pregledne.

Uporaba OCL-ja je zanimiva tudi pri diagramih primerov uporabe. Tu lahko z OCL pred-pogoji in po-pogoji na standarden način zelo formalno definiramo začetna in končna stanja primerov uporabe. To je zelo dobro, saj s tem formalno oziroma nedvoumno določimo vhode in izhode primerov uporabe na dovolj enostaven način, da jih lahko razumejo tudi strokovnjaki za domeno, ki ponavadi nimajo informacijskih znanj. Seveda tu uporabimo okrnjeno različico OCL-ja,

[4]Ideja povzeta po metodi načrtovanja po pogodbi (angl. design by contract) [11, str. 79–85].

ponavadi brez formalno definiranega konteksta.

OCL definicije se da precej enostavno preslikati v programsko kodo [31, pogl. 4]. Pred-pogoji in po-pogoji se tako prevedejo v zelo podobno programsko kodo na začetku oziroma koncu metod, ki ob neizpolnjenem pogoju vržejo *izjemo* (angl. exception). Invariante je smiselno testirati ob vsaki spremembi stanja, na katero se invarianta nanaša, čeprav je to morda v določenih primerih vprašljivo s stališča časovne zahtevnosti. Navigacija po razredih preko asociacij skorajda ne potrebuje preslikave, podobno so precej enostavne preslikave operacij nad zbirkami.

S stališča modelno vodenega razvoja je uporaba jezika OCL zanimiva tudi za definicijo preslikav iz modelov v druge modele oziroma programsko kodo. Pri tej uporabi gre za delovanje na enem meta nivoju višje. Ne ukvarja se več s primerki UML gradnikov, ampak z meta elementi oziroma UML meta modelom, kot so UML razredi, povezave med njimi ali pa z gradniki diagramov stanj ali interakcije.

OCL je tako zelo dobra dopolnitev k UML diagramom v smislu zmanjševanja dvoumnosti ter povečevanja natančnosti in informativnosti diagramov. Predvsem je dober zaradi svoje deklarativnosti in kompaktnosti, v nekaterih primerih pa postane malo preveč podoben programski kodi in tako ne doprinese kaj dosti k dvigu nivoja abstrakcije.

2.3.8 Dvig abstraktnega nivoja z jezikom UML

Po ogledu posameznih vrst diagramov jezika UML sledi obravnava vpliva najbolj uporabnih vrst UML diagramov na dvig abstraktnega nivoja.

Razredni diagrami glede na objektne jezike s stališča dviga nivoja interakcije ne pomenijo praktično nobenega napredka. Prevajanje razrednega diagrama v nižji objektno usmerjen abstraktni nivo je precej enostavna operacija, ki razrede UML diagrama ter njihove atribute in operacije preslika v objektne razrede z atributi in operacijami, asociacije pa preslika v dodatne atribute razredov.

Diagrami stanj, kot drugi najbolj uporabni UML diagrami, imajo že večji vpliv na dvigovanje nivoja abstrakcije, saj je z njimi možno izraziti stanja, prehode med njimi in akcije, ki se izvedejo ob vhodih v stanja. Diagrami prehajanja stanj zelo dobro rešujejo specifičen tip problemov, kjer so stanja zelo pomembna.

K dvigu nivoja abstrakcije, hkrati pa tudi povečevanju informativnosti ter reševanju dvoumnosti UML diagramov, precej pripomore jezik OCL za določanje omejitev s svojo deklarativnostjo.

Različica 2.x UML jezika je v primerjavi s prvimi različicami veliko bolj primerna za uporabo pri modelno vodenem razvoju, v bistvu je bilo nastajanje druge različice predvsem pod vplivom uporabe v modelno vodenem razvoju. S tega stališča so tudi ostali diagrami bolj primerni za uporabo pri modelno vodenem razvoju in s to povečano uporabnostjo tudi nekaj doprinesejo k dvigu abstraktnega nivoja.

2.3.9 Povzetek uporabnosti jezika UML

Od vseh diagramov UML-ja so se za namene modelno vodenega razvoja izkazali najbolj uporabni razredni diagrami ter diagrami prehajanja stanj, oboji dopolnjeni z jezikom OCL za podajanje omejitev ali določeno konkretno akcijsko semantiko. Uporabni so tudi komponentni in namestitveni diagrami, vendar ti dve vrsti nimata takega pomena kot razredni diagrami in diagrami prehajanja stanj, s katerimi podrobno opišemo sistem. Od teh dveh tipov diagramov k dvigu nivoja abstrakcije doprinesejo le diagrami prehajanja stanj. Drugi diagrami so za direktno uporabo v modelno vodenem pristopu manj primerni. Še vedno pa jih lahko uporabimo v njihove originalne namene, torej za opisovanje sistema. Uporabnost UML diagramov precej poveča uporaba jezika OCL za določanje omejitev, poizvedb in v omejeni obliki tudi splošnejših postopkov, z dopolnjevanjem omejenih izraznih možnosti UML diagramov.

Nakazali smo tudi, da UML diagrami za modeliranje določenih vidikov sistema ne delujejo najbolj primerno. O tem problemu in alternativah bo govora pozneje. Ugotovili smo še, da tudi različica UML-ja 2.x, razen akcijske semantike brez konkretnega zapisa in manjših izboljšav, ne prinaša nič posebno novega za modelno vodeni pristop.

2.4 Domensko-specifični (modelni) jeziki

Obravnava jezika UML je pokazala, da je ta le delno uporaben za namene modelno vodenega razvoja. Kljub navidezni uporabnosti večine njegovih diagramov, se v praksi izkažejo zares uporabni le redki. Tako se samo od sebe postavlja vprašanje alternativnih modelnih jezikov za modeliranje stvari, za katere splošno-namenski modelni jeziki niso najbolj primerni.

Kot nasprotje oziroma dopolnitev splošno-namenskih jezikov so se pojavili *domensko-specifični jeziki*. Domensko-specifični jezik je jezik za opisovanje določene domene[5]. Tak jezik za primernejši opis domene vsebuje nestandardne na novo definirane konstrukte, ki ustrezajo konceptom domene jezika. Domensko specifični jezik večinoma opisuje ožje definirano domeno. Modelni jeziki za opis specifičnih domen se imenujejo *domensko-specifični modelni jeziki* [2], ki so še posebej zanimivi za modelno vodeni razvoj.

Domensko-specifični modelni jeziki za dejanske domene lahko že obstajajo, sicer pa se jih lahko razvija kar sproti, ko se oceni, da so boljša rešitev kot uporaba obstoječih možnosti. Večinoma se jih uporablja kot dopolnitev splošno-namenskih modelnih jezikov, lahko pa tudi samostojno.

Z domensko-specifični jeziki se da idealno opisati poljubno domeno, saj so ti jeziki načrtovani specifično za opis dejanske domene. Hkrati pa oblikovanje svojega domensko-specifičnega modelnega jezika zahteva tudi izdelavo preslikovalne funkcije iz zamišljenega modelnega jezika na nižji abstraktni nivo.

[5] Glej osnovno predstavitev v uvodnem razdelku 1.3.3.

Za namene modelno vodenega razvoja so večinoma primerni domensko-specifični *modelni* jeziki, kar ohlapno rečeno pomeni klasične domensko-specifične jezike z grafičnim zapisom.

2.4.1 Domensko-specifični in splošno-namenski modelni jeziki

Tako se pri modeliranju za namene modelno vodenega razvoja ponujata dve osnovni možnosti s svojimi prednostmi in slabostmi. Uporabljati je možno splošno-namenske jezike, ki lahko že imajo preslikave, vendar z njimi ni mogoče izraziti vsega, vsaj na idealen način ne. Lahko pa se izdela lastne domensko-specifične jezike, s katerimi je mogoče na precej idealen način opisati določeno domeno. Vendar je treba za te jezike izdelati še prevedbo na nižji abstraktni nivo.

Obe možnosti je možno tudi kombinirati. Pri večini problemov je smiselno uporabiti en splošno-namenski modelni jezik ter več domensko-specifičnih, po enega za vsako od osnovnih domen, ki sestavljajo celotno domeno reševanega problema.

OMG z MDA se v osnovi še vedno drži le UML-ja. Bolje rečeno brez dodatnih dejanskih rešitev dopušča vse, kar se da zapisati v standardu MOF. Podobno tudi pristaši izvedljivega UML-ja vztrajajo pri svojih osnovnih modelih in ne eni ne drugi ne posvečajo preveč pozornosti domensko-specifičnim modelnim jezikom. Ob poznavanju pomanjkljivosti splošno-namenskih in prednosti domensko-specifičnih modelnih jezikov pa je težko oporekati zagovornikom vsaj delne uporabe domensko-specifičnih jezikov. Obstoječa orodja in rešitve to potrjujejo, saj vsaj delno uporabljajo tudi domensko-specifične modelne jezike, več o tem pri obravnavi orodij.

Ideja domensko-specifičnih jezikov, ki jih je treba domisliti in zanje izdelati tudi prevedbo na nižji abstraktni nivo, pomeni odmik od idealnega modelno vodenega razvoja, opisanega v prvem vidiku v razdelku 2.2, kjer naj bi se razvijalec ukvarjal le z risanjem modelov, ki bi se jih dalo neposredno ali posredno izvajati. Vendar zaradi prednosti in slabosti posameznih tipov jezikov deluje kombiniranje različnih modelnih jezikov kot prava smer pri modelno vodenem razvoju, pri čemer je še vedno smiselno ločevati samo modeliranje problema od načrtovanja modelnega jezika in izdelave preslikave na nižji nivo. To ločevanje pripomore k oblikovanju dobrih domensko-specifičnih jezikov, neodvisnih od izvedbe njihovih preslikav na nižji abstraktni nivo.

2.4.2 Problemi kombiniranja modelnih jezikov

Kombiniranje modelnih jezikov uvaja dodatne probleme. Težave se pojavijo predvsem pri povezovanju več konceptov skupaj, kjer na mejah morda nimamo istih konceptov ali pa ti niso direktno tehnično združljivi.

Podobne probleme imamo tudi v programskih jezikih pri sestavljanju domensko-specifičnih jezikov in/ali (ponovno uporabnih) komponent v končno rešitev. Tu se problemi ponavadi rešujejo s *povezovalno kodo* (angl. glue code), ki služi

za pretvorbo robnih konceptov ene domene v robne koncepte druge domene. Podobno izvedljivi UML, ki tudi priporoča oblikovanje domen[6], za njihovo povezovanje uporablja tako imenovane *mostove* (angl. bridges) [20, pogl. 3]. Most je pri izvedljivem UML-ju definiran kot povezovalni element dveh domen, pri kateri ena domena postavi določene predpostavke, ki so enake zahtevam druge domene, kar poenostavljeno povedano pomeni dogovorjen način prehoda podatkov oziroma komunikacijo med domenama.

Enega izmed možnih načinov povezovanja različnih modelnih jezikov podaja skupina ATLAS [1]. Na konceptualnem nivoju predlagajo oblikovanje preslikav med modelnimi jeziki, kar na konceptualnem nivoju deluje trivialno, vendar predlagatelji ideje podajajo tudi dejansko rešitev. Ugotavljajo, da so po eni strani meta modeli domensko-specifičnih modelnih jezikov izrazljivi z MOF-om, v uvodu omenjenem standardu za formalno definicijo modelov in meta modelov, približevanje z druge strani pa vidijo v uporabi UML profilov za omejevanje splošnosti UML-ja, seveda le v kontekstu reševanja dejanskega problema. Tako dobijo približana modela in, vzporedno na enem meta nivoju višje, meta modela. Nato določijo, kateri elementi enega modela oziroma meta modela sovpadajo z elementi modela oziroma meta modela druge strani. Z določanjem teh sovpadanj je tako že definirana preslikava med temi modelnimi jeziki. Dejansko pretvorbo naredijo s procesom, ki ga poimenujejo *tkanje* (angl. weaving). Podrobneje tudi podajajo izvedbo z lastnimi rešitvami tkanja in preslikav. Na kratko povzeto je glavna ideja predstavljenega predloga povezovanja različnih modelnih jezikov definicija pretvorbe istoležnih robnih konceptov teh modelnih jezikov ter oblikovanje in uporaba preslikave po tej definiciji. Ideja skupine ATLAS deluje zelo uporabno.

Splošno velja, da je dve domeni možno povezati med seboj, če se da robne elemente ene domene nekako pretvoriti v robne elemente druge, kjer je ta pretvorba lahko bolj ali manj zapletena. Če je pretvorba zelo zapletena ali pa ne izgleda možna, to morda pomeni, da med obravnavanima domenama obstaja še ena ali več drugih domen, preko katerih je treba povezati obravnavani domeni. Ali pa, da je treba eno izmed povezovanih domen razširiti, da bo ponujala vse za povezovanje potrebne podatke oziroma koncepte.

Seveda je treba vedno težiti k temu, da pri kombinirani uporabi različnih modelnih jezikov ne prihaja do velikih problemov. Na to je možno vplivati z izbiro morebitnih že izdelanih modelnih jezikov ali s prilagojeno izdelavo lastnih modelnih jezikov. Pri izdelavi lastnih modelnih jezikov pa je, v smislu skladnosti z drugimi modelnimi jeziki, treba ravnati zelo premišljeno. Po eni strani je dobro domensko-specifični jezik narediti čim bolj splošno oziroma ponovno uporaben, po drugi strani pa lahko zaradi tega nastopijo težave pri njegovi uporabi pri dejanskem primeru. To je zapleten problem, o katerem bo še govora v nadaljevanju.

[6]Izvedljivi UML sicer priporoča oblikovanje domen, vendar le z uporabo izbranih UML modelov in akcijskega jezika.

2.4.3 Ponovna uporaba in domenski standardi

Za ponovno uporabo domensko-specifičnih modelnih jezikov velja podobno kot za druge ponovno uporabne koncepte. Uporaba obstoječega domensko-specifičnega jezika pomeni prihranek določene količine dela in posledično povečanje produktivnosti. Namesto lastnih rešitev so uporabljene preverjene rešitve. Pri poskusu ponovne uporabe določenega domensko-specifičnega jezika gre v bistvu za ravno obravnavano kombiniranje domensko-specifičnih jezikov. Tako je na tem mestu, brez ponovnega naštevanja ugotovitev problemov kombiniranja jezikov, smiselno zapisati, da je ponovna uporabnost domensko-specifičnih jezikov možna in da zanjo veljajo vse ugotovitve na temo kombiniranja modelnih jezikov prejšnjega razdelka.

Ponovno uporabo precej olajšajo že uveljavljeni standardi za določene domene. Kot primer takšnih domen in standardov se mnogokrat pojavljata AutSAR iz avtomobilske domene in IMS (IP Multimedia Subsystem) iz domene telekomunikacij in mrež [14]. Domenski standardi so veliko boljši vhod v načrtovanje kot lastna analiza, saj so ponavadi rezultat premišljenega dela večih strokovnjakov iz obravnavane domene. Izdelanih obstoječih splošno uporabnih modelnih jezikov, striktno v smislu uporabe za modelno vodeni razvoj, danes še ni zares na voljo, bi se pa lahko pojavili v bližnji prihodnosti. Tu je morda smiselno opozoriti na vse druge oblike enostavnejše podpore razvoju programskih rešitev za določene domene. Morda je dober primer SAP s podporo za razne finančno in tudi drugače usmerjene domene, kar dodatno dokazuje smiselnost uporabe domensko-specifičnih jezikov.

Obstoj standardov seveda ne pomeni, da je treba vedno uporabiti ali izdelati modelni jezik, ki popolnoma sledi standardu. Treba je razmišljati pragmatično in agilno ter izdelati za dejanski problem najbolj primerno rešitev, hkrati pa tudi razmišljati o ponovni uporabnosti. Dobro priporočilo je uporaba le tistega dela standarda, ki je zanimiv za obravnavani problem – spet z mislijo na kompromis med dejanskim problemom in ponovno uporabnostjo. Smiselno je poskusiti razbiti domeno in standard na pod-domene in pod-standarde ter oblikovati domensko-specifični modelni jezik le za eno ali več zanimivih pod-domen.

2.4.4 MDA in domensko-specifični modelni jeziki

Treba je povedati, da MDA ni omejen le na UML modele, ampak dovoljuje tudi druge modelne jezike. Natančneje dovoljuje vse modelne jezike, ki se jih da definirati po standardu MOF. OMG se je z MDA kljub temu od UML modelov oddaljil le z definicijo UML profilov za razne standarde, kot so EJB (Enterpise Java Beans) ali CORBA. Pod okriljem OMG je sicer nastalo veliko definicij raznih standardnih domen s področja računalništva, kemije, ekonomije, vojske in podobnih. Te definicije so podane zelo jasno in natančno, vendar večinoma le na nivoju vmesnikov, in tako uporabljajo skoraj izključno razredne UML diagrame, mnogokrat pa so podane še celo v obliki CORBA vmesnikov.

Tako MDA ne prepoveduje uporabe domensko-specifičnih jezikov, hkrati pa

jih nikjer posebej ne priporoča. Druga literatura [11, str. 155] pa včasih podaja tudi omenjeno možnost uporabe modelnih jezikov z obvezno definicijo MOF meta modela za razviti modelni jezik. Definicija MOF meta modela za posebej razvite modelne jezike je predpogoj za uporabo takšnih jezikov v standardnih MDA orodjih.

Omenjena definicija UML profilov za dejanske tehnične rešitve se nam ne zdi prava končna smer, saj menimo, da bi se razvijalci morali osredotočati na modeliranje poslovne logike in oddaljevati od tehničnih podrobnosti.

2.4.5 Realnost domensko-specifičnih jezikov

Domensko specifični jeziki pa sami po sebi niso univerzalno zdravilo za vse probleme MDA različice in modelno vodenega pristopa v splošnem. Razviti je sicer možno poljuben jezik, vendar je treba tak jezik najprej domisliti.

Tako je relativno enostavno razviti domensko-specifične jezike za razne domene deklarativne narave, ki jih je v osnovi možno malo bolj okorno opisati tudi s standardnimi UML razrednimi diagrami. Primer takega domensko-specifičnega jezika je jezik za opisovanje grafičnega vmesnika[7]. Grafični vmesnik je že sam po sebi vizualen, njegov modelni jezik si je tako enostavno zamisliti. V osnovi ga sestavljajo koncepti, kot so okno, gumb, vnosno polje in podobno. S tem, da je grafični vmesnik podan le konceptualno, način izrisa in podobne obstranske zadeve pa je smiselno ločiti v poseben vidik.

Domensko-specifični jeziki sami po sebi prav nič ne pomagajo pri, na primer, iskanju idealnega zapisa obnašanja. Domensko-specifični jeziki so konec koncev le pristop za konkretizacijo teoretičnih idej, ki jih je na določenih področjih še potrebno izumiti.

2.5 Splošno o jezikih

Modeli so izraženi z modelnimi jeziki, ki so le posebna vrsta splošnih jezikov. Jezike v osnovi delimo na *naravne* in *formalne*. Za idealen opis problema[8] so primerni le formalni jeziki. Pri modelno vodenem razvoju uporabljamo formalne modelne jezike, torej jezike s katerimi lahko z modeliranjem izdelamo idealen opis problema.

Formalni jezik je določen s *pomenom* (angl. semantics), *abstraktnim zapisom* (angl. abstract syntax) ter enim ali več *konkretnimi zapisi* (angl. concrete syntaxes) [13, str. 280]. Abstraktni zapis določa gradnike jezika ter njihove dovoljene kombinacije na konceptualnem nivoju, na primer pri razrednih UML diagramih so to razredi in povezave med njimi. Pomen jezika definira pomen njegovih konstruktov, pri razrednih UML diagramih na primer: razredi opisujejo neko stvar, dejstvo ali pojem, povezave pa odvisnosti med njimi. Konkreten zapis pa definira način

[7] Domensko-specifični jezik za preprost grafični vmesnik bo podan v nadaljevanju.
[8] Za podrobno obrazložitev pojma idealnega opisa problema glej razdelek 1.3.1.

in simbole za podajanje jezika, ponavadi v vizualni obliki, pri razrednih UML diagramih je to definicija predstavitve razreda s pravokotnikom, razdeljenim na tri dele za ime, atribute in operacije in tako naprej. Tako formalen jezik definirajo koncepti, njihov pomen ter en ali več konkretnih načinov za zapis teh konceptov.

Poznavanje predstavljene definicije je zelo pomembno za ukvarjanje z modeli. Razlika med koncepti jezika in njihovim pomenom je precej intuitivna, manj ljudi pa se zaveda abstraktnega in konkretnih zapisov. Abstraktni zapis, torej ideja konceptov jezika, je poleg pomena najpomembnejši vidik jezika. Ljudje miselno operirajo z abstraktnim zapisom, konkreten zapis sam je drugotnega pomena in služi predvsem kot pripomoček za pomoč pri delu z množicami konceptov, ki so za človekove sposobnosti pomnjenja preobširne, za komunikacijo idej med ljudmi in seveda za modelno vodeni razvoj.

Zelo pomembna je tudi ugotovitev, da lahko obstaja več konkretnih zapisov jezika. Tako je predstavitev razreda iz razrednih UML diagramov kot zgoraj opisanega pravokotnika le ena izmed možnih zapisov oziroma vizualnih predstavitev koncepta razreda iz razrednih UML diagramov. Izdelava novega konkretnega grafičnega zapisa za koncepte razrednih UML diagramov namesto standardnih simbolov sicer ni preveč smiselna, je pa izvedljiva. Hitro je možno ugotoviti, da grafična oblika ni edina možna oblika zapisa. Koncepte razrednih UML diagramov bi bilo možno zapisati tudi v tekstovno datoteko. Na primer najprej razrede z vsemi njihovimi podrobnostmi, nato pa še asociacije med njimi z vsemi podrobnostmi, vsebujoč tudi informacijo, katere razrede povezujejo. Tako diagrame v resnici shranjujejo programi za risanje modelov. Vse te ugotovitve veljajo tudi za klasične programske jezike, ki so ravno tako definirani z abstraktnim zapisom, njegovim pomenom in ponavadi le enim konkretnim zapisom.

Pri UML-ju in drugih splošno-namenskih (modelnih) jezikih dobijo alternativne oblike zapisa pomembnejšo vlogo, če zgornji opis abstraktnega in konkretnega zapisa obrnemo. Preoblikovana trditev pravi, da je mogoče abstraktni zapis in pomen za določen konkreten jezik pridobiti iz različnih zapisov oziroma virov. Tako vhod v funkcijo preslikave pri modelno vodenem razvoju ni nujno grafični model, ampak je lahko tudi določen lasten zapis, ki vsebuje dovolj vhodnih informacij za namen preslikave. Ti zapisi lahko vsebujejo samo tiste elemente abstraktnega zapisa, ki so zanimivi v kontekstu obravnavanega problema. Tak način razmišljanja sodi že bolj k domensko-specifičnim jezikom oziroma nakazuje nestandardne bolj pragmatične možnosti praktične uporabe idej modelno vodenega pristopa.

Zanimivo je primerjati tudi formalne jezike objektnih programskih jezikov in UML razrednih diagramov. Ugotoviti je možno, da gre približno za en sam jezik z dvema konkretnima zapisoma. Manj idealno je v praksi videti ta skoraj enak abstraktni zapis pri vsebini operacij oziroma metod, kjer si je kot alternativen konkretni zapis za vsebino metod pri programski kodi možno predstavljati UML diagrame interakcije. Opažanja potrjuje tudi dejstvo, da pretvorbe med programsko kodo in tema dvema vrstama diagramov podpira veliko orodij za klasičen razvoj v objektnih programskih jezikih. Obširna podpora tej ‘pretvorbi’ je tako

razširjena, ker sploh ni pretvorba oziroma preslikava modelov, ampak le vizualna predstavitev istih konceptov v drugem konkretnem zapisu.

Pri domensko-specifičnih jezikih je oblikovanje preprostih in pragmatičnih konkretnih zapisov še posebej smiselno. Ker gre za oblikovanje lastnega jezika za lastno uporabo, ni smiselno razvijati zahtevnih orodij za podporo modeliranju z grafičnimi simboli, ampak ideje podati čim enostavneje. Ta napotek sledi osnovni težnji pragmatičnosti oziroma enostavnosti (angl. simplicity) domensko-specifičnih jezikov.

Smiselna je tudi možnost, da se namesto modeliranja posebnega modela za vhod v preslikovalno funkcijo raje uporabi v okviru reševanega problema že obstoječe modele oziroma zapise. Ta možnost bo praktično predstavljena v nadaljevanju pri obravnavi praktičnih primerov ukvarjanja z modelno vodenim pristopom.

2.6 Klasični programski jeziki

Zapis programske rešitve je pri sodobnih in danes največ uporabljanih objektnih programskih jezikih, kot sta na primer Java in C#, sestavljen iz množice tekstovnih datotek z izvorno programsko kodo[9], hierarhično urejenih v mape oziroma *pakete* (angl. packages). Paketi oziroma datoteke z izvorno programsko kodo so ponavadi grupirane v določene smiselne komponente in module, ki hkrati predstavljajo minimalne enote, iz katerih se s prevajanjem dobi prevedene komponente, katerih smiselne skupine tvorijo izvedljive programske rešitve.

Sama vsebina tekstovnih datotek mora ustrezati enemu od natančno predpisanih formatov. V Javi na primer lahko tako ena datoteka s programsko izvorno kodo definira razred ali *vmesnik* (angl. interface), v novejših različicah pa še *zaznamek* (angl. annotation) in *naštevni tip* (angl. enumeration), od katerih je razred daleč najbolj pomemben in uporabljan. Podobno velja tudi za C# in podobne jezike. Tekstovni opis razreda vsebuje navedbo paketa razreda, uporabljenih drugih razredov, imena razreda, morebitnega prednika, vmesnikov, atributov in metod z njihovo definicijo.

Pisanje in urejanje programske kode tako pomeni urejanje visoko formalizirane vsebine tekstovnih datotek. Posledica tega je narava orodij, kjer razvijalec, poleg uporabe funkcionalnosti prevajalnika in podobnega, v osnovi ureja tekst programske kode.

Slabosti in pomanjkljivosti

Problem zapisa programskih jezikov je že omenjeni prenizek abstraktni nivo, na katerega je treba predelati opis iz idealnega abstraktnega nivoja, s čimer pride do izgube namena[10], prevedba oziroma preslikava pa hkrati zahteva še veliko dela. Na voljo so le predstavljeni koncepti, torej le razredi z atributi in operacijami, za

[9] Poleg datotek z izvorno programsko kodo celotno rešitev ponavadi sestavljajo tudi razne druge nastavitvene in podobne datoteke.

[10] Pojma *abstraktni nivo* in *izguba namena* sta natančneje opisana v razdelku 1.3.1.

izražanje bolj kompleksnih konceptov je treba iz danih konceptov sestavljati nove. Lep primer tega je grafični vmesnik, kjer v osnovi jezika ni konstruktov za elemente grafičnega vmesnika, ampak so ti izraženi kot razredi ali komponente, sestavljene iz večih razredov; v osnovi je podprta le funkcionalnost risanja na zaslon. Taka podpora za gradnjo grafičnega vmesnika je v sodobnih jezikih sicer ponavadi že vključena v osnovni razvijalski paket oziroma je na voljo v več dodatnih različicah, nikakor pa ta podpora ni del osnovnih elementov jezika. Gumb je na primer predstavljen z razredom Gumb in ne obstaja kot samostojen koncept enakovreden razredu.

Doseganje višjega abstraktnega nivoja

Problem preveč nizkega abstraktnega nivoja pri programskih jezikih je tako možno reševati s sestavljanjem osnovnih konceptov programskih jezikov v bolj kompleksne komponente. Take komponente se lahko gradi po trenutnem občutku, lahko pa se jih gradi bolj premišljeno, za določeno domeno, kar pomeni izdelavo klasičnih nemodelnih domensko-specifičnih jezikov.

Sestavljanje bolj kompleksnih konstruktov v programskem jeziku v bistvu pomeni ustvarjanje določenega višjega abstraktnega nivoja. Gre za enako idejo kot pri modelno vodenem razvoju, s pomembno razliko, da se tu namesto standardne preslikovalne funkcije v programski kodi aktivira ponovno uporabne komponente.

Zanimiv je tudi pogled na programsko kodo z vrha navzdol oziroma pregledovanje programske kode z začetkom v točki, v kateri se program začne izvajati. Ob predpostavki dobre arhitekture in poimenovanja je iz začetka kmalu videti glavne funkcionalnosti pregledovanega programa. Dobro napisana programska koda se ob omenjenem načinu pregledovanja bere na precej visokem abstraktnem nivoju, s pregledovanjem podrobnosti izvedbe pa se počasi spuščamo na nižje abstraktne nivoje. Seveda še tako dobro napisana programska koda ne more preprečiti vsaj delne izgube namena oziroma zelo nevarnega napačnega ali pomanjkljivega razumevanja pri popravkih te kode.

2.7 Novosti na področju razvoja programske opreme

Vzporedno z modelno vodenim pristopom se seveda razvijajo tudi drugi pristopi in koncepti, od katerih je modelno vodeni razvoj odvisen oziroma se pojavljajo novi koncepti, katere je v modelno vodenem pristopu možno s pridom uporabiti. Z izboljšavami obstoječih in pojavom novih konceptov se v bistvu dviga nivo abstrakcije, na katerega morajo preslikave modelno vodenega razvoja preslikovati modele. S tega vidika je pomemben tudi napredek na področju programskih jezikov, ki so večinoma ciljni modelni jezik preslikav. V nadaljevanju je tako podana obravnava pomembnejših novosti na področju razvoja programske opreme in možnosti uporabe teh novosti v modelno vodenemu pristopu.

2.7.1 Aspektno-orientirano programiranje

Aspektno-orientirano programiranje [6, pogl. 7] sloni na ideji *ločevanja vidikov* (angl. separation of concerns). Ločevanje vidikov je dobra praksa, ki obravnavano domeno razdeli na smiselne funkcionalne enote. Poleg tako dobljenih funkcionalnih enot pa obstajajo še drugi vidiki, ki jih ni možno uvrstiti v nobeno od identificiranih funkcionalnih enot, hkrati pa se ti vidiki raztezajo čez vse funkcionalne enote. V klasični programski kodi so ti dodatni vidiki vpleteni v osnovne funkcionalne enote, kar po principu ločevanja vidikov ni dobro. Primeri takih posebnih vidikov so varnost (angl. security), zapisovanje dnevnika (angl. logging), transakcije, sinhronizacija, merjenje različnih parametrov programov in podobno.

Aspektno-orientirano programiranje poskuša te posebne vidike, ki se raztezajo čez več funkcionalnih enot, izolirati iz programske kode funkcionalnih komponent. Izolacijo je mogoče doseči na več precej različnih načinov, v nadaljevanju bo podan v praksi najbolj uporabljan pristop. Ta pristop omenjene posebne vidike izloča iz programske kode funkcionalnih komponent ter jih podaja ločeno v obliki *aspektov* (angl. aspect). Natančneje je to doseženo z *nasvetom* (angl. advice), ki določa učinek aspekta in *mesti uporabe* (angl. pointcuts) nasveta v programu, kjer se ta nasvet efektivno izvede. Ta mesta so ena izmed *možnih mest uporabe* (angl. join points). Ta možna mesta uporabe so ponavadi klici metod ali dostop do polj razreda.

Primer aspekta in njegove uporabe je vidik izvajanja določene kode v transakciji. Definirati je treba nasvet, ki mora pred izvedbo kode, na katero bo apliciran, začeti transakcijo, po koncu pa to transakcijo zaključiti. Nato je treba le še povedati, kateri deli programske kode se morajo izvajati v transakciji, kar se doseže z določanjem metod, ki se morajo izvesti v transakciji. Celotna rešitev posebnega vidika izvajanja v transakcijah je tako definicija nasveta za meje transakcije ter definicija metod, okrog katerih se mora ta nasvet oviti.

Ideja aspektno-orientiranega programiranja je direktno uporabna v večini pristopov in tako tudi pri modelno vodenem razvoju. Skupaj z idejo ločevanja vidikov je uporabna kot dobra praksa, obstoječe rešitve aspektno-orientiranega programiranja v klasičnih programskih jezikih pa lahko služijo kot ciljni model, na nekoliko višjem nivoju od klasičnih programskih jezikov, na katerega preslikovalna funkcija preslikuje modele. V modelih pa se v osnovi modelira osnovne funkcionalne enote, hkrati pa se modelom, njihovim delom ali elementom na dogovorjen način določi aspekte oziroma namene, ki se nanje nanašajo. Možen je tudi drugačen način podajanja mest uporabe namenov, kot podajanje mest namenov izven modelov, kar je morda slabše. Namene same je večinoma možno izraziti v kompaktni in deklarativni obliki z enim samim pojmom, včasih z morebitnimi dodatnimi parametri. Tako bi bilo za splošno definicijo aspektov v jeziku UML možno definirati nov UML profil, kjer bi bila mesta uporabe aspektov določena z izbranimi stereotipi, njihovi morebitni parametri pa s predpisanimi zaznamki.

2.7.2 Inverzija kontrole in vrivanje odvisnosti z ogrodji

Inverzija kontrole (angl. inversion of control) pomeni zasuk v mišljenju glede kontrole poteka izvajanja programa [15]. Pri klasičnem programiranju je potek programa viden direktno iz programske kode. Program se začne v določeni glavni metodi, od tam se po vrsti kličejo naštete metode, ki spet kličejo druge metode in tako naprej. Pri inverziji kontrole pa za kontrolo skrbi izbrano ogrodje, ki kliče s strani razvijalca razvite in nastavljene metode oziroma komponente. Kontrola se tako preseli v uporabljeno ogrodje.

Za ponazoritev ideje inverzije kontrole lahko služi enostaven program, ki najprej zahteva vnos imena in nato izpiše pozdrav za vpisano ime. Enostavna klasična različica je sestavljena iz ukaza za branje imena iz vhoda in ukaza za izpis pozdrava na izhod. Pri tej različici je kontrola izvajanja v celoti podana v opisanem programu. Primer za inverzijo kontrole pa je različica istega programa z grafičnim vmesnikom. V sodobnih programskih jezikih se grafični vmesnik le definira, v podanem primeru vnosno polje in gumb za potrditev konca vnosa in izpis sporočila, ter akcije, ki se zgodijo ob raznih dogodkih, kot je na primer aktivacija gumba. Ob začetku programa je treba tako le zagnati ogrodje za grafični vmesnik, ki mu je bila na predpisan način podana omenjena definicija grafičnega vmesnika, vključno z odzivi na možne sprožene dogodke. Kontrolo nad izvajanjem programa ima tako ogrodje oziroma knjižnica za grafični vmesnik. Tako ni treba pisati precej obširne programske kode za grafični vmesnik, ampak podati le za dejanski program relevantna dejstva.

Koncept inverzije kontrole je uporaben v problemih, kjer ni enega samega pravega zaporedja ukazov, ki privedejo do končnega rezultata, kot na primer pri problemu iskanja prvih n praštevil, ampak v problemih, kjer izvajanje traja več časa, in je pri tem treba ponavljajoče sprožiti določene funkcionalnosti kot odziv na določen vhod, na primer aplikacije z uporabniškim vmesnikom za urejanje določenih podatkov, streženje določenih zahtev in podobno. Glavna prednost uporabe inverzije kontrole je tako uporaba določene ponovno uporabne komponente za del rešitve v obliki ogrodja, kot je v podanem primeru uporabljeno ogrodje za grafični vmesnik. Tako so podana le osnovna dejstva poslovne logike, kar naredi zapis rešitve bolj kompakten in razumljiv.

Vrivanje odvisnosti (angl. dependency injection) je posebna oblika inverzije kontrole, kjer se določeni komponenti poda oziroma injektira zahtevano storitev (angl. service) v obliki druge komponente, ki omogoča zahtevano storitev.

Primer komponente, ki uporablja zahtevano storitev v obliki druge komponente, je komponenta, ki iz vrstic računa sestavi končni račun, kjer je treba sešteti zneske posameznih vrstic v končni znesek računa in le-tega pretvoriti v podano tujo valuto. Omenjena pretvorba med valutami je storitev, ki jo lahko ponuja splošna komponenta s funkcionalnostjo pretvorbe zneska v vhodni valuti v izhodno valuto na določen dan. Tako ima komponenta za sestavljanje računa vrinjeno komponento za storitev pretvorbe zneskov med valutami. Tako zastavljena komponenta za pretvorbo zneskov med valutami je čisto splošna komponenta in je kot ponovno

uporabna komponenta lahko vrinjena tudi v druge komponente, ki zahtevajo to storitev.

Vrivanje odvisnosti spodbuja oblikovanje smiselnih komponent ter njihovo ponovno uporabnost. To je uporabno pri razbijanju obsežnejših postopkov v smiselno povezane komponente. Ravno obsežni postopki oziroma obnašanje pa so šibka točka modelno vodenega razvoja in se tako iz ideje vrivanja odvisnosti da naučiti mnogo koristnega in te ideje uporabiti pri modelno vodenem razvoju.

Vrivanje odvisnosti še posebej dobro deluje z vzorcem *vmesnik* (angl. interface) oziroma konceptom *programiranja k vmesnikom* (angl. programming to interfaces). Za komponento, ki potrebuje določeno storitev, se to storitev opiše z vmesnikom, predvidi se tudi dostop do te storitve preko nastavljive komponente, ki ustreza (angl. implements) temu vmesniku. Pri sestavljanju komponent v program se osnovni komponenti nastavi komponento za dano storitev. V prihodnosti se lahko za storitev napiše novo implementacijo, s katero je možno enostavno zamenjati obstoječo.

Inverzija kontrole, še posebej pa vrivanje odvisnosti, sta se v zadnjih letih začela precej široko uporabljati pri razvoju programske opreme z objektno usmerjenimi jeziki. Ideji na prvi pogled delujeta nerodno, s časom pa si je težko predstavljati razvoj programov brez niju. Ob osvojitvi teh idej se pogled na strukturo programov spremeni, na programe se gleda kot na komponente, sestavljene iz različnih drugih komponent. Osnovne komponente se sestavlja v kompleksnejše komponente in te na koncu v program kot lego kocke. Pri priučevanju na omenjena koncepta gre za podoben problem kot pri prehodu na objektno usmerjene jezike. Na začetku ni bilo nikomur jasno kaj naj počne z objekti, danes bi pa težko shajali brez njih.

Za dvig priljubljenosti, predvsem vrivanja odvisnosti, so poskrbela razna ogrodja, ki omogočajo deklarativno določanje komponent in nastavljanje njihovih odvisnosti. Eno najbolj uporabljanih takih ogrodij v Java svetu je Spring [15], ki omogoča deklaracijo komponent in vrivanje odvisnosti, večinoma podano v posebnih XML datotekah. Na začetku programa se iz teh datotek na precej enostaven način pridobi ponavadi en sam primerek osnovne komponente, sestavljene iz drugih komponent, na katerem je treba poklicati metodo, ki začne izvajanje funkcionalnosti te osnovne komponente. Seveda so vse komponente napisane v programskem jeziku; prednost je predvsem v sestavljanju teh komponent zunaj programske kode. Tako se razvijalci lažje posvetijo dobrim komponentam s smiselno in po možnosti ponovno uporabno funkcionalnostjo.

Inverzija kontrole in vrivanje odvisnosti sta v bistvu dva vzorca. Posebno pozornost si zaslužita, ker sta za razliko od večine drugih relativno ozko uporabnih vzorcev res zelo splošno uporabna. Za modelno vodeni razvoj pa sta še posebej zanimiva zaradi že omenjenega razbijanja obsežnih postopkov oziroma obnašanja v smiselne komponente in sestavljanja teh komponent v programske rešitve.

2.7.3 Zaznamki

Zaznamki (angl. annotations) so s stališča modelno vodenega razvoja še posebej zanimivi, saj so se pojavili iz podobnih razlogov kot modelno vodeni razvoj. Klasični programski jeziki so izrazno precej omejeni, eno takih omejenih področij je dodajanje meta podatkov raznim konstruktom jezika. Pred zaznamki so se meta podatki dodajali v obliki komentarjev, kar ni bilo najbolj idealno. Boljša rešitev so zaznamki, ki meta podatke dodajajo na formalen način. Definicija zaznamka obsega njegovo ime, konstrukte programskega jezika, ki jih lahko zaznamujemo, ter morebitne dodatne parametre. Tako definiran zaznamek je možno dodajati dovoljenim konstruktom jezika, pri tem dodajanju pa je možno določiti vrednosti morebitnim parametrom zaznamka. Od zaznamkov pri programskih jezikih je možno potegniti vzporednice do vseh treh načinov razširjanja UML diagramov, še najbolj k UML zaznamkom in stereotipom.

Zaznamki so uporabni za res velik spekter namenov od čisto specifičnih do splošnih. Definirati je možno zelo splošne zaznamke, kot na primer zaznamek *NotNull* za označevanje parametrov, ki ne smejo imeti prazne vrednosti, ali pa zaznamek, ki pove format podatka pri preverjanju pravilnosti podatkov ali v procesu razčlenjevanja (angl. parsing). Seveda je treba napisati programsko kodo, ki te zaznamke upošteva.

Lahko pa se uporabi že obstoječe zaznamke in programske komponente, ki te zaznamke uporabljajo. Primer za to so zaznamki za označevanje razredov, katerih stanje se mora ohraniti (angl. persistent) in programska knjižnica, ki uporablja te zaznamke in s tem zelo poenostavi problem shranjevanja. Obstaja mnogo podobnih uporab, kjer se na glavne podatkovne razrede doda zaznamke za razne vrste operacij nad temi razredi, kot je že omenjeno shranjevanje stanja, preoblikovanje v razne druge oblike za namene komunikacije z drugimi sistemi in podobno.

Pri nazadnje omenjenih primerih uporabe zaznamkov gre za pomemben napredek v smislu modelno vodenega razvoja, v smislu dviga nivoja abstrakcije. Namesto pisanja programske kode za shranjevanje, preoblikovanje oziroma druge podobne operacije za vsak razred posebej se tem razredom le doda za te operacije potrebne meta podatke v obliki zaznamkov. Operacije pa se razvije za vse razrede skupaj v eni splošni funkcionalnosti, ki pri obravnavi določenega vhodnega podatka uporablja meta podatke tega parametra, natančneje njegovega tipa. Primer takega pristopa bo podan v nadaljevanju pri predstavitvi praktičnega ukvarjanja z modelno vodenim razvojem. Omenjeni primeri niso edini, možnosti so res velike.

Pri uporabi zaznamkov na pravkar opisan način se precej spremeni pogled na reševanje problemov. Ne poskuša se več rešiti dejanskih problemov, namesto tega se poskuša najti množico podobnih problemov in zanje razviti skupno rešitev na enem meta nivoju višje. To pa je samo z drugimi besedami predstavljena ideja dvigovanja nivoja abstrakcije oziroma ideja ločevanja reševanja splošnega tehničnega problema od dejanskega vsebinskega problema.

2.7.4 Kompaktnost zapisa in privzete vrednosti

Poleg že omenjenih prednosti višjih abstraktnih nivojev[11] je s stališča zapisa navodil računalniku velika prednost tudi *kompaktnost zapisa* na višjih abstraktnih nivojih. Kot je en konstrukt klasičnega programskega jezika lahko zamenjava za več sto vrstic kode zbirnega jezika, je pri modelno vodenem razvoju označevanje razreda razrednega diagrama z določenim stereotipom lahko zamenjava za veliko programske kode v klasičnem programskem jeziku. Prednost v smislu kompaktnosti zapisa imajo tudi razne grafične in druge tehnike, ki na omejenem fizičnem prostoru (zaslona) omogočajo prikaz več kompaktneje prikazanih konceptov.

Privzete vrednosti

H kompaktnosti zapisa zelo pripomorejo tudi *privzete vrednosti* (angl. default values). Privzete vrednosti so vnaprej določene vrednosti za izbrane parametre koncepta, ki so v večini primerov že dovolj dobre, v dejanskih primerih pa se lahko za te parametre s privzetimi vrednostmi poda druge vrednosti, ki se uporabijo namesto privzetih.

Dober primer za privzete vrednosti je dolžina polj v modelu podatkovne baze, ki se generira iz podatkovnega modela[12]. Tako je lahko privzeta vrednost za nize v podatkovni bazi 100 znakov, v primerih, kjer je potrebno shranjevati daljše nize oziroma bi bili nizi privzete dolžine preveč potratni, pa se posebej določi daljše oziroma krajše dolžine.

Privzeta vrednost v najširšem smislu obsega vse, kar razvijalec ne določi sam. Tako je na primer privzeta vrednost pri grafičnih vmesnikih tudi vizualna oblika gumba. Gumb se izriše v določeni obliki, če pa to razvijalcu ne ustreza, ima ponavadi možnost določanja lastnega izrisa gumba. Iz podanega primera je videti, da se pojem privzete vrednosti v širšem smislu počasi že meša z razširljivimi in nastavljivimi (angl. configurable) ponovno uporabnim komponentami.

Največja prednost privzetih vrednosti je kompaktnost zapisa. Treba pa je poudariti, da je za razumevanje kompaktnih zapisov s privzetimi vrednostmi potrebno poznavanje vseh možnih privzetih vrednosti, kar ima za slabost zahtevnejše razumevanje takšnih zapisov. Ta problem berljivosti oziroma poznavanja vrednosti atributov je mogoče v orodjih rešiti z omogočanjem vklopa prikaza atributov s privzetimi vrednostmi, katerih vrednosti niso bile posebej določene.

Tu bi opozorili na splošnejši problem razumevanja delovanja programske opreme s stališča njenih razvijalcev. Pri izključni uporabi programskega jezika razvijalec precej natančno razume, kaj napisani program počne. Uporaba CASE in podobnih orodij, ponovno uporabnih komponent, vmesne programske opreme in podobnega, nenazadnje tudi uporaba prednastavljenih vrednosti, pa v programske

[11] Glej razdelek 1.3.1.

[12] Uporaba objektno-relacijskih preslikav (angl. object-relation mapping) in generiranje modela podatkovne baze je zelo razširjen koncept pri klasičnih programskih jezikih, v podobni obliki pa se uporablja tudi pri modelno vodenem pristopu.

rešitve vnaša cele sklope funkcionalnosti, ki so za razvijalca črna škatla[13]. S tem v osnovi ni nič narobe, ponovna uporaba je danes nuja. Probleme včasih povzroča zahtevnost uporabe omenjenih funkcionalnosti zaradi kompleksnosti, slabe dokumentacije in morebitnega posledičnega napačnega razumevanja. Za podkrepitev kritičnosti opazke podajamo primer konfiguracije porazdeljenih transakcij v raznih aplikacijskih strežnikih, kjer za vsako osnovno transakcijo in za porazdeljeno transakcijo obstaja več deset (prednastavljenih) nastavitev, za razumevanje katerih je potrebno natančno poznati teorijo in razumeti skopo dokumentacijo, kar se večini primerov ne izvede najbolj idealno.

2.7.5 Deklarativnost

Deklarativno programiranje pomeni podajanje navodil računalniku v smislu, *kaj* naj program naredi, za razliko od *imperativnega programiranja*, kjer razvijalec poda algoritem, s katerim pove, *kako* naj program deluje [31, str. 18]. Z deklarativno definicijo, torej s *kaj* že povemo dovolj oziroma že imamo *procesno popolno* definicijo, *kako* je potreben le zaradi računalnika oziroma zaradi *abstraktnega prepada*.

Deklarativno programiranje oziroma podajanje navodil računalniku je veliko bolj enostavno in kompaktno ter tudi bližje človeku. Pri imperativnih programih je iz programskega zapisa ponavadi težje izluščiti, kaj program počne, medtem ko deklarativni program že po definiciji pove, kaj počne. Tako imajo deklarativni zapisi manjšo *izgubo namena*.

Pojma deklarativno in imperativno programiranje se da posplošiti na deklarativno in imperativno podajanje navodil računalniku, iz česar je videti, da sta koncepta deklarativnosti in imperativnosti smiselna tudi pri modelno vodenem razvoju. Pri modelno vodenemu razvoju z modeli kot osnovnimi izdelki razvijalca je deklarativnost še posebej dobrodošla. Deklarativnost je za določene koncepte že dobro domišljena in podprta (na primer UML razredni diagrami za opisovanje osnovnega statičnega vidika problemske domene), za druge koncepte pa smiselni deklarativni načini opisovanja še ne obstajajo.

2.7.6 Pravila

Pravila (angl. rules) so tudi zelo zanimiva s stališča modelno vodenega razvoja. Osnovna ideja pravil je podajanje določenih delov navodil računalniku v obliki pravil. Tako se izbranih delov programa ne napiše klasično v programskem jeziku, ampak se namesto tega na teh izbranih mestih v programu zažene *stroj za izvajanje pravil* (angl. rule execution engine), ki izvede določeno logiko z upoštevanjem podanih pravil. Pravila so oblike *če pogoj, potem akcija*, torej omogočajo podajanje logike v obliki pogojnega izvajanja akcij. Pravila so ponavadi napisana na zelo visokem abstraktnem nivoju, včasih kar v visoko formaliziranih jezikih, ki

[13] Včasih ima razvijalec sicer možnost spoznati vsebino te črne škatle, vendar zaradi obsežnosti vsebine črnih škatel tega ne naredi.

že spominjajo na naravne jezike. Uporaba pravil je smiselna na mestih, kjer se poslovna logika lahko hitro in zelo pogosto spreminja. Ob spremembi je treba le zamenjati ali dopolniti pravila, osnovna logika ostane nespremenjena. Zaradi visokega abstraktnega nivoja zapisa pravil lahko poleg razvijalcev načeloma ta pravila pišejo tudi poznavalci problemske domene.

Klasičen primer uporabe pravil so popusti v obračunskih sistemih. Oblikovalci storitev si lahko iz dneva v dan izmišljujejo razne nove popuste, kot na primer deset odstotni popust na osebe, ki so mlajše od določenega števila let, porabijo več kot določen fiksni znesek ali so določenega spola. Takšne spremembe poslovne logike je s pravili v primerjavi s klasičnim programiranjem možno implementirati veliko hitreje, enostavneje, ceneje in varneje. Treba je le dodati nova ali popraviti obstoječa pravila.

Pravila še najbolj spominjajo na domensko-specifične jezike, pri obeh pristopih se za opisovanje določenega dela sistema uporablja neke vrste poseben jezik. Razlika je le, da se pri domensko specifičnih jezikih oblikuje poseben jezik, ki se uporabi v klasičnem programskem jeziku oziroma modelno vodenem razvoju, pri pravilih pa se iz klasičnega programskega jezika preko stroja za izvajanje pravil da pravilom na voljo določene podatke, ki jih pravila lahko spreminjajo. Pravila so namenjena naknadnemu spreminjanju delovanja programa in so tako veliko bolj fleksibilna od klasičnega načina uporabe domensko-specifičnih jezikov.

2.7.7 Vzorci

Vzorci (angl. patterns) so splošen način podajanja dobrih praks. Ko se za določen problem ugotovi splošno dobro rešitev, se to rešitev opiše v obliki vzorca z namenom podajanja ugotovljene dobre prakse v bolj ali manj formalni obliki.

Na področju razvoja programske opreme so najbolj znani *načrtovalski vzorci* (angl. design patterns), ki podajajo dobre prakse pri oblikovanju arhitekture programskih rešitev, pa tudi drugih vidikov razvoja. Razviti so bili z namenom uporabe pri razvoju programov v klasičnih programskih jezikih, so pa direktno uporabni tudi pri modelno vodenemu pristopu. V MDA pa tudi drugih različicah se ti vzorci ponavadi uporabljajo kot vmesni nivo modelnih jezikov med opisom na visokem nivoju in ciljnim nivojem.

Pojem vzorca je seveda čisto splošen, tako je mogoče govoriti o vzorcih v smislu dobrih praks seveda tudi v okviru modelno vodenega pristopa. Nekatera orodja tako v opisu svojega delovanja govorijo na primer o preslikovalnih vzorcih.

Poleg formaliziranega podajanja znanja je ena bistvenih dobrih lastnosti vzorcev tudi izboljšanje komunikacije med strokovnjaki, saj je ob predpostavki poznavanja splošnih vzorcev možno dobre prakse določenega standardnega vzorca podati že z imenom vzorca.

2.7.8 Generiranje programske kode

Veliko razvijalcev že dolgo časa uporablja (ne da bi se tega zavedali) modelno vodeni razvoj v nekakšni osnovni okrnjeni različici. Gre za koncept *generiranja programske kode*, kjer se s preslikavo, podobno tisti pri modelno vodenem razvoju, iz določenega vhoda, neke vrste modela, generira programska koda. Bistvena novost modelno vodenega razvoja glede na generiranje programske kode je predvsem premik od generirane kode k preslikavam in predvsem modelom kot ključnim izdelkom.

Glavna ideja generiranja programske kode je le generirana koda, bolj se osredotoča na preslikave, modele pa pozna bolj kot neko nujno potrebno sestavino za vhod v preslikavo. Pri modelno vodenem razvoju so, s stališča razvoja programske opreme, modeli bolj pomembni kot vsi drugi (vmesni) rezultati razvoja, celo bolj kot končna rešitev. Najbolj revolucionarna ideja modelno vodenega razvoja je ravno ta premik od kode programskih jezikov k modelom. Razvijalec ne piše več programske kode, ampak modelira. Modelno vodeni razvoj je vsekakor zanimiv za uporabnike generiranja kode, saj jim generiranje kode prikaže v novi luči, začnejo se zavedati pomena modelov, natančneje dejstva, da je vhod v generiranje bolj pomemben kot njegov rezultat ali preslikava.

2.8 Podrobneje o modelih

Po predstavitvi domensko-specifičnih modelnih jezikov in obravnavi klasičnih programskih jezikov ter v kontekstu ugotovitve, da se modelno vodeni razvoj ne vrti le okrog UML in MDA, se modeli kažejo v nekoliko drugačni luči. Kot centralni koncept modelno vodenega razvoja, natančneje idealno edini izdelek razvijalcev, si zaslužijo podrobnejšo obravnavo.

Sledeče ugotovitve se večinoma nanašajo na grafične konkretne zapise modelnih jezikov, pri čemer ne gre pozabiti, da je določen konkreten zapis le eden izmed možnih načinov podajanja abstraktnega zapisa, kot je bilo predstavljeno pri obravnavi jezikov na splošno.

2.8.1 Osnovna anatomija modelov

Modeli so večinoma sestavljeni iz elementov in povezav med njimi. Tako na primer UML razredne diagrame sestavljajo razredi in asociacije med njimi. Podobno velja tudi za druge UML diagrame, pa tudi za domensko-specifične modele. Vedno gre za določene koncepte in relacije oziroma odvisnosti med njimi.

Koncepti imajo določeno notranjo strukturo, ki je bolj ali manj zapletena, in ponavadi predstavljajo smiselne celote iz obravnavane domene. Relacije med koncepti pa te smiselne celote domene povezujejo med seboj oziroma določajo odnose med njimi. Notranja struktura konceptov je lahko celo sestavljena iz drugih med seboj povezanih konceptov na istem ali drugem nivoju.

Ugotovljeni vzorec anatomije modelov kot konceptov in odvisnosti med njimi je zelo zanimiv in pomemben. Ta vzorec nakazuje dejstvo, da realnost ni homogena, ampak je strnjena v sisteme, ki so med seboj v določeni odvisnosti. Pomembno je tudi dejstvo, da so kompleksni sistemi ponavadi sestavljeni iz primitivnejših sistemov. Iz tega sledi možnost obravnavanja sistemov in podsistemov kot omejenih celot z določenimi vhodi, znano ali neznano notranjo sestavo in določenimi izhodi.

Ugotovljena anatomija modelov v smislu konceptov in odvisnosti med njimi velja tudi za programsko kodo. Vendar je vsa ugotovljena dejstva veliko lažje videti z vidika obravnave modelov kot z vidika obravnave programske kode. To nekako nakazuje večjo primernost modelov za *idealen opis domen*[14]. V nadaljevanju pa bo podanih tudi nekaj argumentov proti tej trditvi.

2.8.2 Grafična predstavitev in njene prednosti

Predstavitev modelov v grafičnem konkretnem zapisu je v primerjavi z zapisom programske kode veliko bolj fleksibilna. K temu pripomore že sama v prejšnjem razdelku ugotovljena anatomija modelov, sestavljena iz pomembnih konceptov in odvisnosti med njimi.

Zelo prikladna je tudi možnost pozicioniranja elementov in določanja poti povezav med njimi. Tako se sorodne koncepte ponavadi vizualno postavi skupaj in s tem poveča razumljivost diagramov. Programska koda sicer tudi omogoča grupiranje sorodnih konceptov skupaj, vendar v veliko bolj togi obliki.

Pomembna lastnost nekaterih modelov in orodij za delo z njimi je tudi določanje nivoja prikazovanja podrobnosti elementov in povezav modela. Tako je z neprikazovanjem določenih podrobnosti mogoče na manjšem fizičnem prostoru (zaslona) prikazati višjenivojski pogled na določeno domeno ali poddomeno. Omenjene prednosti grafične predstavitve so tesno povezane s pogledi. *Pogled* (angl. view) pomeni predstavitev oziroma prikazovanje le določene podmnožice elementov celotnega modela domene, kjer so lahko elementi prikazani v celoti ali pa le njihove izbrane podrobnosti. Poglede se oblikuje iz različnih razlogov. Osnovni razlog oblikovanja pogledov je že sama obsežnost celotnega modela domene. Tako se celotno domeno ob upoštevanju principa *ločevanja vidikov*[15] razdeli na smiselne funkcionalne poglede. Podobno je sicer možno deliti tudi programsko kodo. Oblikovati pa je možno tudi poglede za natančnejšo predstavitev določenega vidika domene, kar v programski kodi ni možno. Oblikovanje takšnih posebnih pogledov izboljša razumevanje sistema in delno tudi zmanjša *izgubo namena*[16].

2.8.3 Ponovna uporaba, odvisnosti in različice

Za dosego ponovne uporabnosti modelov, je treba pri modeliranju sorodne koncepte modelirati v smiselnih ponovno uporabnih domenah oziroma poddomenah.

[14] Pojem je podrobneje predstavljen v razdelku 1.3.1.
[15] Za razlago pojma glej razdelek 2.7.1.
[16] Za razlago pojma glej razdelek 1.3.1.

Tako je model celotne programske rešitve sestavljen iz večih podmodelov. Oblikovanje delnih modelov in sestavljanje teh modelov v končno rešitev zahteva ustrezne mehanizme. Tako se določeni koncepti določenega delnega modela sklicujejo na koncepte drugih delnih modelov, kar zahteva mehanizem odvisnosti oziroma sklicevanja med delnimi modeli. Potrebna je tudi podpora različicam modelov, podobno kot pri programski kodi. Sem spada tudi upravljanje z različicami in zagotavljanje uporabe med seboj skladnih različic.

2.8.4 Razumljivost modelov

Opisovanje obravnavane domene z modeli namesto s programsko kodo ni nujno avtomatično boljše. To še posebej velja v vmesnem obdobju, ko se modelno vodeni pristop še razvija in še ni na voljo idealnih modelnih jezikov. Primerjava opisa programske rešitve v obliki programske kode in v obliki modelov z morebitno dodatno programsko kodo za dele, ki jih ne moremo opisati z obstoječimi modelnimi jeziki, lahko pokaže slabšo razumljivost opisa z modeli.

Prej omenjena večja fleksibilnost modelov hkrati pomeni tudi manjšo standardiziranost zapisa. Podobno velja tudi za poglede in privzete vrednosti, ki lahko namesto povečevanja razumljivosti povzročijo celo napačno razumevanje. Vprašljiva je tudi razumljivost nestandardnih modelnih jezikov. Tu so še posebej problematični domensko-specifični modelni jeziki. Do zmanjšanja razumljivosti lahko privede tudi kombinirana uporaba večih modelnih jezikov. Dodatno zmedo vnaša uporaba UML-ja za modelno vodeni razvoj in za druge manj formalizirane namene.

Hitro se da ugotoviti, da mora biti tudi opis programske rešitve v modelnem jeziku *procesno popoln*[17]. Večina omenjenih problemov izhaja iz nepoznavanja celovitega pomena elementov modelov in njihovih podrobnosti.

Iz ugotovljenega se ponovno kaže problem izgube namena in abstraktnega prepada. Rešitev teh težav pa je v dvigu abstraktnega nivoja opisovanja problemskih domen[18].

2.8.5 Kombiniranje modelov in kode

V trenutnem vmesnem razvojnem stanju modelno vodenega pristopa še ni možno opisati celotne problemske domene izključno v modelih. Tako se ponavadi del problema reši z modeli in ustreznimi preslikavami v klasično programsko kodo, preostali del pa se direktno razvije v klasični programski kodi. Pri kombiniranju teh dveh pristopov obstaja več možnosti, od enostavnih do morda že preveč kompliciranih, od katerih ima vsaka svoje omejitve in probleme [11, str. 230–244].

Problemi so večinoma posledica prepletanja modelov in programske kode ter izoliranosti sprememb na le eno od teh oblik. Pri klasičnem MDA pristopu se,

[17] Za razlago pojma glej razdelek 1.2.5.

[18] Vsi omenjeni pojmi so predstavljeni v razdelku 1.3.1.

na primer, mnogokrat definira razrede in njihove metode, iz tega generira programsko kodo s praznimi metodami, vsebino metod pa se nato napiše v klasičnem programskem jeziku.

Enostavna možnost kombiniranja modelov in programske kode je ločevanje uporabe teh dveh prijemov čim bolj vertikalno. Tako se določene funkcionalne enote modelira, druge pa napiše v programski kodi. S tem se omenjeni problemi kombiniranja pojavljajo le na mejah teh enot. Druga enostavna možnost predlaga najprej fazo modeliranja, nato preslikavo modelov v programsko kodo in nato ročno dopolnjevanje te generirane programske kode. Tu se problemi pojavijo, ko po modeliranju, generiranju in dopolnjevanju generirane kode pride do potrebe popravljanja modelov in ponovnega generiranja, pri čemer se prvotne ročne dopolnitve ne smejo izgubiti, pa še nekako jih je treba uskladiti s spremembami modela. Najbolj zapletene različice kombiniranja modelov in kode pa celo spodbujajo mešanje teh dveh oblik oziroma prehajanje med njima, kar povzroči še večja neskladja, ki jih je še težje reševati.

Omenjene probleme še dodatno zaplete sodelovanje večih ljudi pri takem načinu razvoja. Vsi ti problemi so se v podobni obliki pojavljali tudi pred modelno vodenim razvojem. Do le-teh pride pri vsaki uporabi več različnih razvojnih izdelkov, katerih spremembe niso med seboj direktno povezane. Klasičen primer je uporaba orodij za načrtovanje modelov podatkovne baze skupaj s programskimi jeziki.

Vsi ti problemi izhajajo iz ločenega obravnavanja obeh osnovnih izdelkov: modelov in programske kode. Rešitev problema je obravnavanje teh dveh razvojnih izdelkov skupaj oziroma upoštevanje njune povezanosti. Z ustreznimi orodji in principi, kot je na primer *preoblikovanje kode* (angl. refactoring), se da te probleme sprememb omiliti.

Idealna rešitev omenjenih problemov pa je seveda uporaba danes še ne izumljenega modelnega jezika, ki bo omogočal relativno homogen idealen opis problemskih domen.

2.8.6 Vzporednice med klasičnimi in modelnimi jeziki

Pri primerjavi jezikov na splošno ter klasičnih in modelnih jezikov se kažejo velike vzporednice. Obstajajo precejšnje enakosti med splošno-namenskimi navadnimi in modelnimi jeziki, prav tako pa tudi med domensko-specifičnimi navadnimi in modelnimi jeziki. Primerjava UML razrednih diagramov z objektnimi programskimi jeziki kaže precejšnjo enakost, podobno se da razviti pomensko čisto enake navadne in modelne domenske jezike. Tako ni težko ugotoviti, da gre pri navadnih programskih oziroma domensko-specifičnih jezikih za skoraj enake stvari kot pri ustreznih modelnih jezikih.

Formalno rečeno, trenutno obstaja več navadnih in modelnih jezikov s skoraj enakimi abstraktnimi zapisi in pomenom in s precej različnimi konkretnimi zapisi, ki so pri modelnih jezikih grafični simboli, pri navadnih pa tekst oziroma programska koda.

2.9 Podrobneje o preslikavah

Že v uvodu je bilo povedano, da je pri modelno vodenem razvoju preslikava definirana kot funkcija, ki ima za vhod poljuben model, ki ustreza predpisanemu vhodnemu meta modelu, in za izhod model, ki ustreza predpisanemu izhodnemu meta modelu. Vhodni in izhodni model ter definicija funkcije v splošnem niso podrobneje definirani. Splošnost te definicije je zelo primerna in posledično pomeni možnost poljubnega rezultata.

V praksi pa je smiselno oblikovati relativno enostavne preslikave, ki delujejo nad relativno enostavnimi modeli in za dosego kompleksnejših rezultatov te enostavne preslikave kombinirati med seboj. Reševanje obsežnih problemov z eno kompleksno preslikavo se ponavadi izkaže za preveč zapleteno in težko obvladljivo. Eden izmed največjih problemov oblikovanja in še posebej implementacije preslikav in modelnih jezikov je namreč prav obvladovanje njihove kompleksnosti. V začetku sta jezik in ustrezna preslikovalna funkcija enostavna, s časom pa se pojavljajo dodatne zahteve in spremembe, ki razvijani modelni jezik, še posebej pa preslikovalno funkcijo, naredijo zelo obširno in nerazumljivo. Vse to velja že za relativno enostavne preslikave po priporočilu prejšnjega odstavka. Zahtevnost oblikovanja preslikav in meta modelov jezikov izhajata predvsem iz dejstva, da se jih oblikuje na enem meta nivoju višje od standardnega meta nivoja klasičnega razvoja programske opreme.

Reševanje oziroma preprečevanje nastanka omenjenih problemov je odvisno predvsem od načina oziroma tehnologije, v kateri se preslikavo implementira. Ravno tako kot za modelne jezike tudi za preslikave velja *izguba namena*, zato je za podajanje preslikav najbolje uporabiti namenski zapis na čim višjem nivoju.

V osnovi so preslikave definirane kot funkcije iz vhodnega v izhodni model. V praksi, predvsem pri ad-hoc rešitvah, pa se včasih izdelujejo poenostavljene različice preslikav, katerih izhod so direktno datoteke programske kode. To je pragmatična, včasih čisto upravičena poenostavitev.

2.9.1 Standard QVT za poizvedbe, poglede in preslikave

V tem razdelku bo predstavljen najbolj znan standard za podajanje preslikav, to je standard za *poizvedbe, poglede in preslikave* (angl. query view transformation), v nadaljevanju QVT [24]. QVT je del standardne MDA različice modelno vodenega pristopa. Kot samo ime pove, poleg preslikav vsebuje tudi zmožnosti za oblikovanje poizvedb in pogledov nad modeli. Gre za v smislu organizacije OMG precej podroben, obsežen in zelo formalno definiran standard. QVT je precej zahteven za razumevanje, hkrati pa je zares dobra rešitev z nekaj naravnost odličnimi idejami.

Vhodni in izhodni modeli morajo pri QVT preslikavah ustrezati standardu MOF. To pomeni, da je QVT mogoče uporabljati za poljubne splošno-namenske in domensko-specifične modelne jezike. Treba je le oblikovati MOF meta modele za te jezike. Zahteva po ustreznosti meta modela MOF-u pa hkrati pomeni, da z osnovnim QVT-jem ni možno direktno generirati programske kode v končni

tekstovni obliki. To ni tako resen problem, saj je možno izdelati MOF meta model ciljnega programskega jezika, če ta še ne obstaja, ter izdelati QVT preslikave v ta ciljni model, katerega zapis v tekstovno obliko je precej trivialen. Kljub trivialnosti je izdelava takšnega zapisovanja relativno obsežna naloga, oblikovanje MOF meta modela ciljnega programskega jezika pa je še bolj zahtevno. Problem je mogoče rešiti tudi z v nadaljevanju opisanim QVT principom *črne škatle*, vendar v tem primeru to ni preslikava, definirana v osnovnem QVT-ju.

Same QVT preslikave so tudi modeli, ki ustrezajo MOF standardu. To je dobrodošla lastnost, ki omogoča izdelavo standardnih splošnih orodij in pol-orodij za delo s takšnimi preslikavami, hranjenje modelov skupaj s preslikavami v standardnem MOF repozitoriju in podobno.

QVT je sestavljen iz treh domensko-specifičnih jezikov: *Core*, *Relations* ter *Operational Mappings* in posebnega principa *Black Box*. QVT za osnovo uporablja jezik OCL.

QVT Core

QVT *Core* je osnovni jezik za podajanje preslikav. Ta jezik predstavlja zelo nizek nivo za specifikacijo preslikav in ni mišljen za uporabo v praksi. QVT Core je v celoti deklarativen jezik, v katerem je preslikava med dvema meta modeloma podana z množico pravil, imenovanih *mapiranja*[19] (angl. mappings), v obliki *'če v prvem meta modelu obstaja določen element, potem iz tega sledi, da mora v drugem modelu obstajati določen drugi element in obratno'*. Takim mapiranjem lahko dodamo dodatne pogoje za proženje in poljubne dodatne akcije ob izpolnjenih pogojih. Primer preslikave med UML razrednim diagramom in modelom podatkovne baze bi tako vseboval trditev *'če v razrednem diagramu obstaja razred, v modelu baze obstaja tabela in obratno'*. Mapiranja omogočajo tudi trditve v smislu *'če v razrednem diagramu obstaja atribut uA, ki pripada razredu uC, ki mu ustreza tabela bT, potem je v modelu podatkovne baze stolpec bS, ki pripada tabeli bT in obratno'*. S tem je možno povezovanje generiranih konceptov znotraj modelov.

Zelo zanimiva lastnost QVT-ja, ki se uporablja tudi v QVT Core, je *sledenje* (angl. tracing). Sledenje se doseže skozi definicijo mapiranja tako, da se po vzoru klasičnih objektnih razredov definira sledilni razred, ki kot atribute vsebuje elemente obeh modelov, ki sovpadajo po tem mapiranju. Tako bi bilo treba za sledenje pri mapiranju med razredi in tabelami v zgornjem primeru definirati razred za mapiranje med UML razredom in tabelo z atributoma *razred* in *tabela*, v samem mapiranju pa tema dvema atributoma prirediti obstoječe ali na novo generirane vrednosti. Opisano sledenje omogoča pregled poteka preslikave. Hranjenje sledilnih podatkov skupaj z atomičnostjo definicij mapiranja pa ob spremembah omogoča selektivno ponovno preslikovanje le spremenjenih elementov modela, kar

[19] Na tem mestu je namenoma uporabljen izraz *mapiranje*, ki morda ne deluje najbolj slovensko. Razlog za uporabo tega izraza je dejstvo, da podrobna definicija standarda QVT vsebuje res veliko temu pojmu sorodnih pojmov in bi kakršenkoli približen prevod lahko povzročil napačno razumevanje.

```
class RazredTabela {
  ime : String;
  umlRazred : Class;
  tabela : Table;
}

map razredTabela {
  uml () { r:Class }
  rdbms () { t:Table }
  where () { rt:RazredTabela | rt.umlRazred = r; rt.tabela = t; }
  map {
    where () {
      rt.ime := r.name;
      rt.ime := t.name;
      r.name := rt.ime;
      t.name := rt.ime;
    }
  }
}
```

Slika 2.3: *Definicija QVT Core preslikave med razredom UML razrednega diagrama in tabelo podatkovne baze.*

posnema idejo *inkrementalnega prevajanja*. Preslikava le spremenjenih elementov je v praksi ogromna prednost, saj je preslikava obsežnih modelov med razvojem za računalniške zmogljivosti še bolj zahtevna naloga od prevajanja klasičnih programskih jezikov. Same strukture sledenja bi bilo možno uporabiti tudi za razlago poteka preslikave, podobno kot SQL *izvajalni plan* (angl. explain plan).

Ker si je QVT Core praktično zelo težko predstavljati, je na sliki 2.3 podan enostaven primer mapiranja med UML razredom in tabelo podatkovne baze. Iz primera je razvidno, da gre za mapiranje med UML razredom in tabelo podatkovne baze, zgradi se sledilni razred RazredTabela, s štirimi končnimi prireditvami pa se pravilno nastavijo vsa tri imena, neodvisno od smeri preslikovanja.

Z zadnjo trditvijo je bila nakazana ena najboljših lastnosti QVT-ja, to je zmožnost preslikave v obe smeri. Tako je z zgornjo definicijo možna preslikava iz razredov v tabele in iz tabel nazaj v razrede. S tem se odpre mnogo možnosti, kot so (delno avtomatično) propagiranje ročnih popravkov nazaj v modele, začetna izgradnja modelov iz obstoječih meta podatkov, na primer izdelava modelov iz obstoječe podatkovne baze, in podobno. Seveda je te prednosti mogoče izkoristiti le, če oba meta modela omogočata hranjenje določene informacije oziroma je manjkajočo informacijo mogoče določiti implicitno. Tako imajo vse definicije mapiranj lastnost *ekvivalentnosti*, vendar ekvivalenca ni zagotovljena v podrobnosti s samim zapisom; zagotovljena je ekvivalenca osnovnih elementov, med katerimi je defini-

rano mapiranje, ne pa tudi vseh njihovih atributov. To je razvidno iz zgornjega primera, kjer lahko ne bi bile navedene katere izmed zadnjih štirih prireditev in se ime tako ne bi vedno preslikalo.

QVT Relations

QVT *Relations* je višjenivojski deklarativni jezik za specifikacijo preslikav, ki se v ozadju prevede na QVT Core. QVT Relations je tako v okviru QVT priporočen jezik za specifikacijo preslikav. Omenjeni višji nivo pomeni sicer enake efektivne izrazne možnosti, vendar vsebuje naprednejše izraze, kar ima za ugodno posledico bolj kompakten zapis preslikav. QVT Relations ravno tako podpira sledenje, ki ga ni treba posebej definirati; sledenje se določi avtomatično ob prevedbi QVT Relations na QVT Core.

Preslikava s QVT Relations je definirana deklarativno z množico *relacij*, od katerih vsaka relacija v osnovi predstavlja preslikavo med dvema meta elementoma dveh domen. Relacija je konceptualno podobna mapiranju pri QVT Core, le da je tu na voljo malo drugačen enostavnejši in intuitivnejši višjenivojski zapis. Tako je primer preslikave UML razreda v tabelo in obratno skupaj z dejstvom, da se paket razreda preslika v shemo tabele in obratno in da je treba vsak atribut razreda preslikati v stolpec tabele, podan z relacijo *'UML razred ustreza tabeli podatkovne baze, kjer se ime prenese in se tabeli doda stolpec za glavni ključ, ko se aplicira relacija ekvivalence paketa in sheme, kar povzroči apliciranje relacije ekvivalence atributa in stolpca'*. V osnovi je tako definicija relacije podobna mapiranju iz QVT Core, hkrati pa definira še nad-koncept, iz katerega se relacija aktivira, in pod-koncepte, ki se aktivirajo kot posledica trenutne aktivacije.

QVT Operational Mappings

QVT *Operational Mappings* je imperativen jezik za določanje preslikav, ki ga je možno uporabiti samostojno ali ga kombinirati z QVT Relations. Ta jezik tako zaradi svoje imperativnosti ponuja standardne konstrukte, kot so zaporedno izvajanje ukazov, pogojni stavki, zanke, pa tudi standardne operacije nad zbirkami elementov in podobno. Obstaja celo približen ekvivalent metodam in lokalnim spremenljivkam.

S tem jezikom je zaradi imperativnih elementov omogočena večja fleksibilnost in izraznost, posledično pa so takšne preslikave le enosmerne. Tako njegova uporaba ni priporočena, če to ni res nujno.

QVT Black Box

QVT princip *Black Box* je namenjen podpori nestandardnih preslikav, kot je na primer XSLT. S to podporo je tako možno v standardno orodje s podporo za QVT vključiti tudi nestandardne preslikave.

Izvedbe QVT standarda

QVT predstavlja trenutno najboljši standard za preslikave. Njegov glavni problem je, da v osnovi obstaja le kot specifikacija. Praktična izvedba te specifikacije je ogromen projekt, katerega izvedbo si verjetno lahko privoščijo le razvijalci resnih orodij za podporo modelno vodenemu pristopu.

Za uporabo standarda QVT v raznih ad-hoc rešitvah bi bila tako potrebna že obstoječa implementacija. Trenutno obstaja ATL odprtokodna implementacija [4], ki pa standardu QVT ne sledi popolnoma, ampak bolj pragmatično.

Zaključek

Podani opis standarda QVT dokazuje res dober standard za definicijo preslikav, pogledov in poizvedb, izstopata predvsem ekvivalenčnost in sledljivost ter njune pozitivne posledice. Danes skoraj vsa orodja oglašujejo popolno skladnost s tem standardom, kar postaja vedno bolj resnično. Primer takega orodja je v poglavju o orodjih opisan OptimalJ. QVT in MOF skupaj tvorita splošno podporo za oblikovanje poljubnih modelov in preslikav med njimi.

2.9.2 Preslikave s programsko kodo

Ena izmed možnosti podajanja preslikav je tudi implementacija preslikav v izbranem programskem jeziku. Ta možnost je zelo splošna in izrazna, vendar morda ne najbolj priporočljiva, saj je preslikava podana v relativno zapleteni obliki in jo je težko razumeti in vzdrževati.

Izvedba preslikovalne funkcije s programsko kodo je problematična tudi s stališča neobstoja standardov oziroma smernic za njihovo obliko, način zaganjanja in druge vidike, ki bi lahko bili standardizirani. Truditi se je treba vzpostaviti določene minimalne smernice vsaj na projektu, sicer se preslikave lahko hitro izrodijo v skupek zelo različne in neobvladljive programske kode. Več bo o praktični izdelavi takšnih preslikav napisanega pri obravnavi možnosti razvoja orodij.

2.9.3 Tekstovne predloge

Priljubljena relativno enostavna rešitev preslikovanja so tekstovne predloge. Ideja tega pristopa je izdelava predlog v obliki tekstovnih datotek, ki vsebujejo spremenljivke in omejene imperativne zmožnosti. Za uporabo takšne predloge je ponavadi treba napisati preprost program, ki iz vhodnega modela pridobi njegove elemente in za vsak ustrezen element konkretizira predlogo, kjer vrednosti spremenljivk predloge pridobi iz podatkov elementa modela. Rezultat takšnih preslikav je tako vedno tekst oziroma tekstovna datoteka.

Predloge so uporabne pri pogoju, da so končni rezultat preslikave tekstovne datoteke, ki imajo določeno precej standardno obliko. To velja za večino problemov preslikovanja, saj je rezultat ponavadi programska koda ali razne nastavitvene in

druge pomožne datoteke. Priporočene enostavne preslikovalne funkcije pa imajo ponavadi za rezultat strukturno precej podobne formate. Tako so tekstovne predloge zelo široko uporabne, zaradi enostavnosti predvsem pri raznih ad-hoc rešitvah.

2.9.4 Drugi pristopi k preslikovanju

Poleg opisanih najbolj zanimivih in v praksi uporabljanih pristopov preslikovanja obstaja še veliko drugih možnosti. Dober pregled teh možnosti podajajo Czarnecki in sod. [7], ki načine preslikovanja obravnavajo po različnih zmožnostih in lastnostih: elementi preslikovalnih pravil (spremenljivke, vzorci, logika), obseg vhodnega in izhodnega modela, nad katerima ta pravila delujejo, odvisnost oziroma povezanost med vhodnim in izhodnim modelom v smislu možnosti preslikovanja v obstoječ izhodni model in podobno, determinizem izvajanja pravil, vrstni red izvajanja pravil, organizacija samih preslikovalnih pravil, zmožnost sledenja preslikavi in smernost preslikave.

Na podlagi teh lastnosti načine preslikovanja kategorizirajo v preslikave iz modelov v tekst in na preslikave iz modelov v modele. Med preslikave iz modelov v tekst sodi že predstavljen pristop na osnovi predlog in pristop na osnovi *vzorca obiskovalca* (angl. visitor pattern), kjer je treba podati logiko, ki kot odziv na 'obisk' določenega elementa vhodnega modela dopolni izhodni tekst, za samo 'obiskovanje' pa skrbi ogrodje, kateremu je treba podati omenjeno logiko. Med preslikave iz modelov v modele pa sodi že omenjeni pristop pisanja preslikovalne funkcije v programskem jeziku z možnostjo direktnega dostopa do vhodnega in izhodnega modela, relacijski pristop s podajanjem relacij med elementi vhodnega in izhodnega modela, katerega primer je QVT, pristopi na podlagi teorije preslikav grafov, strukturno vodeni pristopi, kjer so preslikave podane kot funkcije elementov izhodnih modelov v odvisnosti od vhodnih, ter hibridni pristopi kot kombinacija naštetih osnovnih pristopov.

Zaključujejo z ugotovitvijo, da so preslikave med modeli še precej nerazvito področje, najboljša rešitev pa je verjetno hibridni pristop z relacijskim pristopom kot osnovo.

2.10 Problem modeliranja kot posledica neobstoja teoretičnih rešitev

Osnovni problem na poti do idealnega modelno vodenega razvoja so modelni jeziki, torej način, kako modelirati določene koncepte. Glavni problem je tako dejstvo, da za predstavitev določenih konceptov še ne obstaja ustrezna teoretična rešitev, oziroma s stališča jezikov, da za te koncepte še ne obstaja niti dober abstrakten zapis.

Preslikave kljub svoji zapletenosti niso problematične. Preslikave med določenim vhodnim in izhodnim modelom so samo izvedljive ali neizvedljive, izvedljive pa so, če na vhodu obstajajo ustrezne informacije. Na primer, obveznosti polj tabel

v bazi, kot jo narekuje poslovna logika, ni možno generirati brez opremljanja vhodnega UML razrednega modela s to informacijo.

2.10.1 Problem modeliranja obnašanja

V dosedanjem pisanju je bil že večkrat omenjen problem modeliranja *obnašanja*[20] (angl. behavior). Za modeliranje obnašanja sicer obstaja veliko rešitev, kot so razni UML diagrami[21] in podobno, vendar nobena ne deluje veliko boljše od zapisa v programskih jezikih. Zaradi grafične konkretne notacije je večina teh zapisov še slabše berljivih in precej okornih za uporabo.

Še največ koristi do sedaj je od diagramov stanj, vendar, kot je bilo ugotovljeno pri njihovi obravnavi, le pri določeni vrsti problemov, ki imajo že po naravi zasnovo stanj. Preostali napredek je mogoče zajeti kot izboljševanje standardnega imperativnega zapisa programskih jezikov, kar potrjujeta tudi obravnavani jezik OCL in akcijska semantika. Pri teh imperativnih različicah se večinoma kaže trend vračanja k tekstovnemu zapisu, kar morda nakazuje manjšo primernost grafičnih konkretnih zapisov za obnašanje.

Iz dobrih izkušenj z deklarativnostjo pri opisovanju statičnega vidika se tudi pri opisovanju obnašanja teži k deklarativnosti. Vendar do sedaj še ni bil odkrit dober način za deklarativno opisovanje splošnega obnašanja. Uspešno pa se deklarativno podajanje obnašanja uporablja v raznih posebnih primerih. Primer takšne uporabe so pred-pogoji in po-pogoji pristopa *načrtovanja po pogodbi* (angl. design by contract), kjer UML v ta namen predlaga že obravnavani jezik OCL za deklarativno podajanje pogoja. Podajanje izključno pogoja je veliko boljše od imperativnega zapisa teh pogojev v obliki pogojnih stavkov, ki v primeru neizpolnjenosti pogoja prekinejo izvajanje, na primer vržejo izjemo. Ključna informacija pri teh pogojih je le pogoj sam, vse drugo je redundantno. Primer za deklarativno opisovanje obnašanja so tudi grafični vmesniki, ki jih je možno že v nemodelnih orodjih kar narisati oziroma podati deklarativno. Obstaja še veliko domen, ki v osnovi delujejo dinamično, vendar se jih da primerneje izraziti na deklarativen način.

Iz podanih primerov je videti napačnost običajnega prvega vtisa o sestavljenosti opisa sistema iz statičnega in dinamičnega vidika, kjer se osnovni statični vidik modelira z UML razrednimi diagrami, vse ostalo pa spada pod obnašanje, za katero obstaja konceptualno en sam splošen imperativen način opisa. Narobe je v problem obnašanja 'tlačiti' vse, kar se ne da opisati z danes znanimi modelnimi jeziki. Sama ideja uporabe obnašanja za opis večjega dela problema poleg pomanjkanja teoretičnih rešitev delno izhaja tudi iz izkušenj ljudi z delom s klasičnimi programskimi jeziki.

[20] Za pojem obnašanja obstaja več podobnih izrazov, kot na primer dinamični vidik sistema, algoritmi, postopki.

[21] UML diagrami so bil v svoji osnovi mišljeni predvsem kot standarden mehanizem za skiciranje idej oziroma za načrt programske rešitve, ne pa kot modelni jeziki.

2.10.2 Idealni opis sistema (konceptualno)

V duhu zgornjih ugotovitev je smiselno nadaljevati, da tudi dobri splošni pristopi k opisovanju določenih vidikov niso vedno najboljši. Opis določenih statičnih vidikov sistema je izvedljiv z UML razrednimi diagrami, vendar taki opisi ne delujejo vedno najbolj idealno, predvsem zaradi prevelike splošnosti teh diagramov. Primer takega statičnega opisa je opis sestavljenosti komponent programske rešitve, kjer bi bila morda bolj primerna posebna oblika zapisa, ki tesneje sledi idejam *inverzije kontrole*. Po drugi strani pa velja, da ni dobro imeti preveč izraznih možnosti, saj te obremenjujejo in morda celo zavajajo njihovega uporabnika.

Iz vseh dosedanjih ugotovitev si je možno na konceptualnem nivoju ustvariti približno predstavo idealnega nabora modelnih jezikov. Za opisovanje osnovnih statičnih dejstev so tako primerni UML razredni diagrami ali podobni modelni jeziki. Za specifične, statične ali dinamične, vidike je smiselno razviti teoretične osnove in na njihovi podlagi posebne domensko-specifične modelne jezike. Primeri takih specifičnih vidikov so že omenjeni regularni izrazi, sestavljanje komponent in grafični vmesniki. Tako ostane še potreba po dobrem splošnem jeziku za opisovanje obnašanja.

S povečevanjem množice specifičnih rešitev, ki ima še ogromno potenciala, se avtomatično zmanjšuje obseg potrebe po opisovanju splošnega obnašanja. V ekstremni različici z ogromno modelnimi jeziki za vse možne domene postanejo UML razredni diagrami, diagrami stanj, regularni izrazi, domensko-specifični jeziki za grafični vmesnik in podobno le eden izmed enakovrednih jezikov. Iz izkušenj pri uporabi ponovno uporabnih komponent pa je slutiti, da bo za povezovanje in sestavljanje teh komponent še vedno potrebno nekaj, kar bo omogočalo prilagoditve, nekakšen ekvivalent *povezovalni kodi* (angl. glue code) v klasičnih programskih jezikih. Ugotavljanje, ali bo to določena vrsta splošne podpore opisovanja obnašanja oziroma algoritmov, ali kaj naprednejšega, je verjetno prevelika špekulacija.

Za konec je smiselno opozoriti na dve relativno hitro razvidni težavi, ki ju prinaša zgoraj opisano oblikovanje domensko specifičnih jezikov za vse možne domene. Prva težava je v omejeni človeški sposobnosti poznavanja vseh teh jezikov. Gre za fenomen izumljanja že izumljenega zaradi zahtevnosti poznavanja že obstoječih rešitev. Druga težava pa je nerealnost izdelave domenskih jezikov za vse, kar si je možno izmisliti, predvsem za vse možne različice potrebe po podpori za posamezne domene iz različnih vidikov in zaradi različnih zahtev.

Tako je najbolj verjetno, da bo v prihodnosti še vedno obstajala potreba po boljših možnostih za opisovanje splošnih dejstev, s poudarkom na problemu opisovanja obnašanja, za katerega ni dobrih rešitev glede na pričakovanja.

2.10.3 Možnosti razvoja modelnih jezikov

Menimo, da je napredek na področju boljšega splošnega načina opisovanja obnašanja in drugih problematičnih vidikov v smeri, ki si jo večina ljudi pred-

stavlja, vprašljiv. Ta smer ni nič konkretnega. Obstaja predvsem pričakovanje, da bo podajanje obnašanja in ostalih dejstev postalo tako enostavno kot modeliranje osnovnih statičnih konceptov realnosti z UML razrednimi diagrami. Do tega zaključka vprašljivosti se da priti s študijo natančnih specifikacij programske opreme.

Specifikacije programske opreme v trenutnem smislu besede pomenijo popoln opis delovanja programske opreme s kombinacijo čim bolj formalne uporabe naravnega jezika in raznih drugih bolj formalnih načinov opisa, kot so UML in drugi diagrami. Modelno vodeni razvoj v bistvu obljublja, da bodo te specifikacije v bližnji prihodnosti zapisane v popolnoma formalni obliki in da bo ta oblika zapisa na dovolj visokem nivoju, da bo odpravila oziroma bistveno zmanjšala problem *izgube namena*. Glede izvedljivosti formalne oblike ni dvomov (dokaz za to so klasični programski jeziki) bolj je vprašljiv dovolj visok nivo, ki bo preprečil izgubo namena.

Sumarne specifikacije obsegajo oris programske rešitve v stilu *'programska rešitev mora omogočati to in drugo in tretje'*. Bolj podrobne specifikacije že opisujejo osnovne zahteve v smislu akcij, ki jih programska rešitev omogoča, na primer z opisovanjem grafičnega vmesnika in dogajanja v ozadju. Takšne vrste specifikacij v naravnih jezikih vsebujejo relativno nezapletene stavčne konstrukte in ne vsebujejo vseh podrobnih informacij za izdelavo programske rešitve.

Pri specifikacijah, ki zares podrobno definirajo programsko opremo, pa se ti stavčni konstrukti lahko že pri na videz enostavnih domenah zelo zapletejo, sama specifikacija pa posledično postane zelo obširna. Kot primer zapletene domene in specifikacije zanjo lahko služi originalna specifikacija predstavljenega QVT jezika. Podobno zapleteni so odgovori razvijalcem na vprašanja o podrobnostih problemske domene, ki so v specifikaciji slabo definirani.

Iz zapletenosti, podrobnosti in obširnosti popolnih specifikacij v primerjavi z enostavnostjo in kompaktnostjo njihovega osnovnega statičnega dela je videti, da je osnovni statični del specifikacije relativno majhen. Drugače povedano – za popolno specifikacijo programske rešitve je poleg osnovnega statičnega dela potrebno specificirati še mnogo drugih vidikov, ki so vsebinsko veliko obsežnejši kot ta osnovni vidik in ki se jih ponavadi manj idealno kategorizira pod obnašanje. Klasičen primer podajanja dejstev v manj idealni obliki obnašanja je izbira elementov iz zbirke (angl. collection) elementov, ki ustrezajo določenemu pogoju, kar se namesto definiranja sprehajanja čez zbirko in dodajanja pogoju ustreznih elementov v rezultat da rešiti le s podajanjem pogoja oziroma predikata standardni funkcionalnosti.

Končna ugotovitev je tako dejstvo, da so pričakovanja o izdelavi relativno zapletenih programskih rešitev z izdelavo nekaj vrst modelov kompleksnosti reda UML razrednih diagramov zgrešene in preprostega razloga, da je vsako lastnost in funkcionalnost programske rešitve treba na določen način specificirati in da določenih dejstev ni mogoče specificirati na enostaven način.

Kot končni komentar je morda smiselno navesti razliko med različnimi nivoji zahtev po dodelanosti programskih rešitev. V tem smislu najenostavnejšo skraj-

nost predstavlja na hitro in zelo generično izdelana aplikacija, ki na manj idealen način ustreza ohlapno definiranim osnovnim zahtevam, najzahtevnejšo skrajnost pa premišljena idealna aplikacija z raznimi specifičnimi in podrobnimi zahtevami. Enostavno si je predstavljati, da je specifikacija najzahtevnejše rešitve v primerjavi s specifikacijo enostavne veliko bolj zapletena in obsežna. Enostavno različico je možno zelo hitro izdelati z dobrim orodjem za modelno vodeni razvoj ali z ustreznim CASE orodjem, medtem ko bi pri izdelavi najzahtevnejše različice pri uporabi CASE orodij lahko prišlo do nepremostljivih težav, z uporabo modelno vodenega pristopa pa bi bilo veliko dela.

2.11 Navidezna dvojnost pomena pojma 'dvig nivoja abstrakcije'

Pri ukvarjanju z modelno vodenim pristopom se pojavi občutek, da so nekatera na višjem abstraktnem nivoju bolj kompaktno izražena dejstva v osnovnem nižjenivojskem zapisu podana preveč redundantno. Tako zgleda, da se poleg ekvivalentnih zapisov pojavljajo še približno enaki zapisi, torej, da poleg eksaktnega dviga nivoja abstrakcije zapisa obstaja še *približen*. V nekaterih primerih je redundanca zares prisotna, v drugih pa ti približni opisi niso vse, kar sestavlja višjenivojski zapis, ampak je del opisa podan na drugačen način, najpogosteje v obliki privzetih vrednosti.

Za ponazoritev predstavljenega dejstva je zelo dober primer grafičnega vmesnika. V osnovi je možno grafični vmesnik implementirati z osnovnimi funkcijami grafične knjižnice, kjer je treba določiti vsebinske in oblikovne vidike rešitve. Že s pojavom programskih knjižnic za grafične vmesnike je bilo dognano, da je treba vsebino in obliko ločevati. Ta koncept se je prenesel tudi v modelno vodeni pristop. Tako je pri uporabi programske knjižnice kot tudi pri modeliranju grafičnega vmesnika potrebno podati le njegovo vsebino, torej komponente z njihovimi lastnostmi. Oblika grafičnega vmesnika je v bistvu prednastavljena vrednost, ki se jo da v nekaterih rešitvah tudi spremeniti. Domišljenost problema grafičnih vmesnikov dokazujejo tudi *razporejevalniki* (angl. layouts) komponent znotraj starševske komponente, s katerimi se deklarativno določi od velikosti komponente odvisno strategijo določanja lokacije in velikosti vsebovanih komponent.

Tako navidezno pomanjkljiv zapis v veliko primerih v resnici predstavlja le del enako dobro definirane celote, ki je le bolj domišljeno razbita na različne vidike, ki so lahko različno podani, tudi v obliki privzetih vrednosti. Ni mogoče izdelati delujočega grafičnega vmesnika, ne da bi bila nekje določena tudi njegova oblika. Tako je na splošno mogoče reči, da so oziroma morajo biti vsi vidiki in podrobnosti končne rešitve na določen način specificirani. Ta trditev ni v nasprotju s končnimi ugotovitvami prejšnjega razdelka, ki govori o različnih nivojih zahtev po dodelanosti programskih rešitev, kar je čisto nekaj drugega.

2.12 Bistvena dodana vrednost modelno vodenega pristopa

Iz ugotovitev o skorajšnji enakosti med obstoječimi modelnimi in klasičnimi jeziki na nivoju abstraktnega zapisa in možnosti uporabe drugih naprednih novosti v klasičnih programskih jezikih, vključno z domensko-specifičnimi jeziki, se samo od sebe postavlja vprašanje, kaj je sploh bistvena dodana vrednost modelno vodenega pristopa glede na prejšnje pristope.

Samo modeliranje z uporabo istih konceptov v drugi vizualni obliki za dosego istega cilja ne predstavlja bistvene prednosti. Tako je na primer modeliranje UML razrednih diagramov ekvivalentno pisanju razredov v programskem jeziku, z izjemo podajanja vsebine metod, ki je pri UML-ju problematična.

Bistvena dodana vrednost modelnega pristopa je tako v podajanju navodil računalniku v kompaktnejšem visokonivojskem zapisu, ki odpravlja ali vsaj zmanjšuje problem *izgube namena*, zmanjšuje količino nepotrebnih (tehničnih) podrobnosti, zaradi učinkovitosti pa posledično zvišuje produktivnost razvoja programskih rešitev.

Kot je bilo razvidno skozi dosedanjo obravnavo, pa doseganje višjega nivoja zapisa sloni na rešitvah splošnih teoretičnih problemov specifikacije oziroma definiranja posameznih vidikov programskih rešitev. Te teoretične rešitve je možno uporabiti tudi v programskih jezikih v obliki ponovno uporabnih komponent, kar še dodatno potrjuje bistvo modelnega pristopa kot podajanje navodil računalniku v kompaktnejšem višjenivojskem zapisu. Tako je edino, kar programski jeziki v konceptualnem smislu neposredno ne omogočajo, fleksibilnost za dosego poljubnega teoretično podprtega nivoja zapisa navodil računalniku. Vse druge oglaševane prednosti so, striktno gledano, le rezultat marketinga.

Za podkrepitev ugotovitve podajamo še navidezen protiprimer in razlago njegove neupravičenosti. Protiprimer se nanaša na precej razširjeno prakso pisanja entitetnih razredov in praktično skoraj enakih razredov za prenos podatkov (na primer med strežnikom in odjemalcem), kjer je treba te skoraj enake razrede ročno pisati dvakrat, čeprav bi se jih z modelno vodenim pristopom dalo definirati le enkrat in generirati v obe obliki. Protiprimer je moč ovreči z upravičenim dvomom v nujnost obstoja razredov za prenos podatkov, namesto katerih bi bilo možno za prenos podatkov uporabiti kar entitetne razrede ali generičen zapis v drugi tehnologiji, na primer z XML-om. Dvojnost razredov je na nivoju programskega jezika morda upravičena s stališča obvladovanja kompleksnosti, striktno gledano pa predstavlja podvajanje specifikacije in je tako nepotrebna tehnična podrobnost. Pričakovan protiargument ravno predstavljenemu ovrženju bi morda bil, da so razredi za prenos podatkov nujno potrebni, ko je treba podatke prenašati v posebnih strukturah, drugačnih od entitet. To ni protiargument, saj je kakršnokoli drugačno strukturo treba posebej definirati, tudi v modelih.

Poudariti pa je treba, da kljub izvedljivosti vseh idej modelno vodenega pristopa v okviru klasičnih programskih jezikov z izjemo fleksibilnosti zapisa, mo-

delno vodeni pristop v praksi sam po sebi spodbuja uporabo lastnih idej, medtem ko se je teh idej v okviru klasičnih jezikov težje zavedati.

2.13 Povzetek ugotovitev

Osnovna ideja modelno vodenega pristopa je modeliranje oziroma opisovanje sistema z modeli, ki vsebujejo dovolj informacij za avtomatično generiranje končne izvedljive programske rešitve.

Bistveni prvini modelno vodenega pristopa sta tako model in preslikava modela. Osnovna različica pristopa je MDA različica s svojimi splošno-namenskimi standardi MOF, QVT in UML, katere največja pomanjkljivost je pomanjkanje praktičnih rešitev za modeliranje specifičnih problemskih domen. Kot alternativa oziroma dopolnitev se ponujajo domensko-specifični modelni jeziki, ki pa jih je treba šele razviti, skupaj s preslikovalnimi funkcijami.

Na modelno vodeni pristop je danes smiselno gledati z dveh vidikov. Prvi vidik je vidik razvijalca, uporabnika zrelega modelno vodenega pristopa, ki razvija dejanske programske rešitve izključno z uporabo že domišljenih modelnih jezikov in se ne ukvarja s preslikovalnimi funkcijami in drugimi podrobnostmi ozadja pristopa. Danes ta vidik še ni dosežen, zato je pomemben drugi vidik zagotavljanja podpore za prvi vidik, torej manjkajočih modelnih jezikov in preslikav zanje ter gradnje orodij iz teh dveh prvin, kar je ponavadi zelo zahtevna naloga.

Pri modelno vodenem pristopu so tako ključni modeli: z razvojnega vidika v smislu podajanja navodil računalniku izključno preko modelov, z vidika zagotavljanja podpore modelno vodenemu pristopu pa v smislu problematike neobstoja idealnih modelnih jezikov za opisovanje večine dejanskih domen, kar je v bistvu posledica pomanjkanja teoretičnih rešitev za specifikacije teh domen. Preslikave so sekundarnega pomena; z razvojnega vidika je pomembno le zagotovilo, da bo iz narisanih modelov mogoče generirati izvedljivo programsko rešitev, z vidika zagotavljanja podpore modelno vodenemu pristopu pa so preslikave, kjub svoji morebitni kompleksnosti, le izvedljive ali neizvedljive v smislu, da je na izhodu preslikave mogoče pričakovati nekaj vsebinsko originalnega le, če za izračun tega obstajajo vse potrebne informacije v vhodnem modelu.

Striktno gledano je glede na programske jezike bistvena dodana vrednost modelno vodenega pristopa v fleksibilnem visokonivojskem zapisu modelov, vse druge ideje modelno vodenega pristopa je mogoče doseči tudi v programskih jezikih. Res pa je, da v praksi za razliko od programskih jezikov modelno vodeni pristop sam po sebi spodbuja uporabo teh idej.

Modelno vodeni pristop je še v razvojni fazi. Največji problem je že omenjeni neobstoj modelnih jezikov, ki izhaja iz neobstoja teoretičnih rešitev za opis poljubnih domen. Stanje bodo tako izboljšali dosežki na tem problematičnem področju. Splošno prepričanje, da bo razvoj po modelno vodenem pristopu veliko enostavnejši od trenutno znanih oblik, tako sloni na pričakovani enostavnosti prihodnjih rešitev opisovanja poljubnih domen in vidikov, kar je vprašljivo, saj

je iz proučevanja podrobnih procesno popolnih specifikacij videti, da za podrobne opise ni bližnjic in da je treba vse lastnosti in podrobnosti končnih rešitev dejansko podati.

Poglavje 3

Zgradba in izdelava orodij ter prototipi

Po teoretični obravnavi modelno vodenega pristopa k razvoju programske opreme ter njegovih prednosti in pomanjkljivosti sledi obravnava uporabe pristopa v praksi skozi uporabo in izdelavo orodij ter pregled ostalih oblik podpore na temo obravnavanega pristopa.

Skozi predstavitev in ovrednotenje pomembnejših orodij in ostale podpore bo prikazana trenutna stopnja razvitosti podpore modelno vodenemu pristopu. Ta obravnava je še najbolj pomembna zato, ker prikazuje ideje za odpravo ugotovljenih pomanjkljivosti modelno vodenega pristopa ter nakazuje dobre prakse pri tem pristopu. Tako se sledeči pregled orodij bolj posveča orodjem z zanimivimi rešitvami, kot pa trudi biti popoln pregled vseh obstoječih orodij.

Treba je poudariti, da integrirana razvojna okolja (angl. Integrated Development Environments), ki v enem paketu ponujajo (na videz) popolno zbirko mehanizmov za modeliranje in generiranje kode, niso edina možnost modelno vodenega razvoja. To še posebej velja v današnjem času, ko se modelno vodeni razvoj še uveljavlja in so orodja predvsem pa principi še daleč od ideala. Alternativa tem orodjem so razne ad-hoc rešitve, ki bodo tudi obravnavane.

V naslednjih razdelkih bo podanih več osnovnih in naprednih orodij z ustreznimi filozofijami modelno vodenega razvoja. Obravnavo je smiselno začeti z razlago modelno vodenega razvoja z lastnimi sproti napisanimi orodji, saj ta način ponuja vpogled v ozadje modelno vodenega razvoja z obstoječimi orodji. S tem bodo predstavljeni vsi vidiki modelno vodenega razvoja in podrobno delovanje orodij, opis obstoječih orodij v nadaljnjih razdelkih pa bo prikaz možnosti za avtomatizacijo opisanih osnovnih konceptov in korakov.

3.1 Modelno vodeni razvoj z lastnimi orodji

Na začetku je treba še enkrat poudariti, da je razvoj lastnih orodij v celoti zelo zahtevna naloga – tako časovno kot po potrebnih sposobnostih in znanju. Tako

je izdelava lastnih orodij vsaj na začetku smiselna za posamezne manjše dele celotnega problema, saj se tudi na videz enostavni problemi s časom izkažejo za zahtevne.

Večini se verjetno zdi preslikava UML razrednih diagramov v sodoben objektni jezik enostaven problem, vendar je že brez vsebine operacij popolna preslikava sicer obvladljiv, ampak zelo obširen problem, katerega obširnost določata podrobna meta modela UML-ja in ciljnega programskega jezika. Večina bi verjetno tudi najprej ugotovila, da ne potrebuje vseh lastnosti jezikov, in tako izdelala pragmatično rešitev, ki se ponavadi hitro pokaže za nezadostno.

Ideja izdelave lastnih orodij je na prvi pogled podobna generiranju programske kode. Glavna razlika modelno vodenega pristopa glede na generiranje programske kode je predvsem v osredotočanju na modele in ne več na preslikave[1].

Lastne rešitve je priporočeno oblikovati za ožje usmerjene domene in pri tem stremeti k čim boljšemu prileganju takšnih rešitev drugim. V nadaljevanju so predstavljeni splošni koraki za eno tako ozko usmerjeno domeno. Sledeči opis je precej pragmatično naravnan na preproste in hitre rešitve, seveda pa je enako uporaben tudi za zahtevnejše in obsežnejše rešitve ob zavedanju opozoril o zahtevnosti.

Definicija domene in rešitve

Kot pri vsakem resnem razvoju je tudi tu treba najprej izdelati natančno definicijo ozko usmerjene domene. Pravilno je treba identificirati vse za problematiko pomembne elemente z vsemi pomembnimi lastnostmi in odvisnostmi.

Modelni jeziki

Nato je treba določiti lastne *modelne jezike* za opis problemske domene. Uporabi se lahko že obstoječe modelne jezike, na primer UML (z morebitnimi prilagoditvami in omejitvami), lahko pa se razvije lastne modelne jezike. Izdelava lastnih orodij je še posebej smiselna pri uporabi nestandardnih modelnih jezikov, kot so na primer že predstavljeni domensko-specifični modelni jeziki.

Grafični konkretni zapis za uporabljene jezike ni nujno potreben. Pri uporabi lastnih orodij je modele ponavadi še najenostavneje opisati v preprostem tekstovnem zapisu. Alternativa tekstovnemu zapisu je uporaba standardnega zapisa, kot na primer XMI[2], ali enega izmed orodij za razvoj oziroma specifikacijo modelnih jezikov[3].

Smiselno je oblikovati enostavnejše in manjše modelne jezike, saj kompleksni in obsežni modelni jeziki v smislu izdelave lastnih orodij kmalu postanejo neobvladljivi. Nenazadnje je ideja modelno vodenega razvoja poenostaviti in skrajšati čas razvoja programske opreme. Kompleksnejši modelni jeziki so upravičeni le, če so ponovno uporabni.

[1] Za podrobnejšo obravnavo generiranja programske kode glej razdelek 2.7.8.
[2] Za podrobnejšo razlago glej razdelek 1.2.4.
[3] Predstavitev takega orodja sledi v enem izmed naslednjih razdelkov.

Modelno vodeni pristop je smiselno uporabiti za določen vidik sistema, ki se pogosto ponavlja. Za enkratne primerke konceptov je uporaba modelno vodenega razvoja, vsaj v trenutni prehodni fazi, ko je izdelava podpore modelno vodenemu razvoju zahtevno in potratno opravilo, manj smiselna. Primer konceptov z veliko primerki so klasični razredni diagrami, iz katerih se da s preslikavami pridobiti entitete in razrede za prenos podatkov (angl. Data Transfer Objects) ali pa modelni jezik za grafični vmesnik, iz katerega je z različnimi preslikavami mogoče dobiti izvedljivo kodo za različne knjižnice za grafične vmesnike. Modelno vodeni razvoj tako ponavljajočih se vidikov se obrestuje na dolgi rok, saj so tudi razne dopolnitve programske rešitve, ob predpostavki dovolj splošnih preslikovalnih funkcij, enostavne in hitre.

Za tako razvite lastne modelne jezike je potrebno izdelati tudi podrobno opisane meta modele, ki natančno in nedvoumno določajo posamezne elemente modelnega jezika z njihovimi lastnostmi in možnosti kombiniranja teh elementov.

Treba je izbrati tudi ciljni modelni jezik. Dandanes to večinoma pomeni enega izmed programskih jezikov, v bližnji prihodnosti pa bi to izbiro lahko že zamenjal morebitni višjenivojski modelni jezik.

Preslikave

Po izbiri enega ali večih modelnih jezikov, skupaj z ciljnim, za katerega že obstaja preslikava oziroma prevajalnik, je treba definirati še *preslikovalne funkcije*. V enostavnejših primerih se izdela le en, ponavadi enostaven, modelni jezik, ki se direktno preslika v izbrani ciljni modelni, večinoma kar programski jezik, kar pomeni le eno preslikovalno funkcijo. Kot je že bilo omenjeno, vzamejo kompleksnejše sheme načina dela z več modelnimi jeziki in posledično večimi preslikovalnimi funkcijami preveč časa in tako niso smiselne, če ne dajejo ponovno uporabnih rešitev.

Pri zapletenih preslikavah je včasih smiselno preslikavo razbiti na dve zaporedni, kar ima za posledico vmesni modelni jezik. Tovrstno večnivojskost v svoji osnovi predlaga tudi MDA.

Preslikavo je treba natančno definirati, kar kot prvo vključuje izbiro vhodnega in izhodnega meta modela. Natančna definicija preslikave nedvoumno definira rezultat v izhodnem modelnem jeziku glede na vhodni modelni jezik. Zapis preslikave je zaradi nedvoumnosti dobro podati v visoko formalizirani obliki.

Izdelava lastnega orodja

Lastno orodje za modelno vodeni razvoj je možno izdelati v poljubnem programskem jeziku z morebitno uporabo za to namenjenih programskih knjižnic ali pa v enem izmed namenskih orodij, skriptnih jezikov ali drugih poljubnih pripomočkov.

V izbranem programskem jeziku ali orodju je treba najprej razviti meta model za vhodni in izhodni modelni jezik. Če se za izdelavo lastnega orodja uporabi objektni programski jezik, se za vhodni in izhodni meta model posebej razvije

razrede za posamezne elemente meta modela. Če se za opis vhodnega meta modela uporablja eden izmed standardnih načinov, je smiselno preveriti, ali za ta meta model že obstaja programska knjižnica za programski jezik, v katerem bo izdelano lastno orodje. Ker je izhodni model ponavadi programski jezik, je smiselno uporabiti morebitno že obstoječo podporo za meta model programskega jezika oziroma programsko knjižnico, ki to omogoča.

Treba je razviti tudi branje in razčlenjevanje (angl. parsing) vhodnih modelov ter funkcionalnost zapisa ciljnega modela. Spet je smiselno uporabiti morebitno že obstoječo podporo.

Najtežji del razvoja lastnega orodja ponavadi predstavlja implementacija preslikovalne funkcije. Prvi predpogoj za uspeh je že omenjena dobra definicija vhodnega in izhodnega meta modela ter preslikovalne funkcije same. Sama izvedba preslikovalne funkcije ponavadi obsega pregledovanje elementov vhodnega modela ter dodajanje in dopolnjevanje elementov izhodnega modela.

Končno praktično uporabno orodje dobimo tako, da se vse opisane razvite dele sestavi v celoto. Ta sestavljena rešitev najprej prebere vhodni model v primerek vhodnega meta modela, tega s preslikovalno funkcijo preslika v izhodni model in slednjega zapiše kot rezultat v ustrezni obliki, večinoma v datoteke s programsko kodo.

Obstajajo tudi alternativni načini preslikave, ki večinoma poenostavljajo ravno predstavljen najbolj splošen način. Uporaba teh načinov je smiselna, če so dovolj fleksibilni za reševani problem, treba pa se je tudi zavedati njihovih omejitev ob morebitnih razširitvah orodja. Primer takega alternativnega načina je uporaba že predstavljenih tekstovnih predlog (angl. template).

Uporaba lastnega orodja

Uporaba na opisani način izdelanega lastnega orodja se v osnovi nič ne razlikuje od uporabe obstoječih in ponavadi naprednejših orodij. Programsko opremo ali njen del oziroma vidik se razvije z modeliranjem v izdelanem vhodnem modelnem jeziku, izdelani model pa se nato z lastnim orodjem prevede v ciljni model, ki je ponavadi programski jezik, ta rezultat pa se v končni fazi prevede v izvedljivo programsko rešitev.

Kot je bilo že povedano, je ideal modelno vodenega razvoja delo izključno z visokonivojskimi modeli. Temu idealu se da približati s sestavljanjem večih manjših lastnih in obstoječih orodij v celovito orodje. Zagon tako sestavljenega orodja omogoča preslikavo izdelanih modelov v več morebitnih korakih v izvedljivo programsko kodo in testni zagon programske rešitve.

Prednosti, slabosti in smiselnost

Glavna *prednost* razvoja in uporabe lastnih orodij in modelnih jezikov je predvsem popolna svoboda oziroma popolna neomejenost. Kot bo videti v nadaljevanju,

naprednejša orodja ponavadi postavljajo določene omejitve, ki znajo povzročati probleme.

Glavna *slabost* pa sta čas in trud, ki ju je treba vložiti v opisani razvoj orodja. Po zahtevnosti izstopa izvedba preslikave, predvsem zaradi dela na enem meta nivoju višje. Programska koda preslikovalne funkcije tako deluje zelo splošno in hitro postane zelo zapletena.

Izdelava lastnega orodja (skupaj z modelnimi jeziki) je *smiselna*, če prinese prihranek pri razvoju dejanske programske rešitve ali če je tako dobro zastavljeno, da se ga da ponovno uporabiti. Ponavadi se z enim modelnim jezikom reši določen vidik problemske domene, ki je bil pred tem identificiran kot določeno pravilo oziroma vzorec v obravnavani problemski domeni. Zgoraj sta že bila podana dva klasična primera takega vzorca: razredni diagrami za entitetne in druge podatkovne razrede ter modelni jezik za grafični vmesnik.

Za konec je treba še enkrat poudariti najpomembnejše dejstvo modelno vodenega pristopa. Ideja modelno vodenega pristopa je opis problemske domene izključno v visokonivojskih modelnih jezikih. Največji pomen imajo tako razviti lastni modelni jeziki, pomembna so tudi orodja, preslikave pa so kljub zahtevnosti z idealnega vidika uporabe pristopa manj pomembne.

3.1.1 Primer – grafični vmesnik

V tem razdelku bo podan res enostaven primer razvoja lastnega modelnega jezika in orodja zanj za problemsko domeno grafičnega vmesnika. Primer je zaradi omejenosti prostora res stilizirano enostaven, ampak še vedno dovolj nazoren.

V opis primera je pri definiciji preslikave najprej vključeno manj primerno razmišljanje, ki mu sledi ponoven premislek z boljšo rešitvijo. Namen vključitve napačnega razmišljanja je prikazati, kako zahtevno je najti pravo pot pri oblikovanju modelnih jezikov in preslikav.

Definicija domene

Najprej je treba definirati, kaj je sploh mišljeno pod pojmom ‘preprosti grafični vmesnik’. Obravnavani enostavni grafični vmesnik podpira *okno*, določeno s *širino*, *višino*, *naslovom* in množico *komponent*, ki jih vsebuje. Vsaka komponenta ima določeno tudi lego znotraj okna s koordinatama *x koordinata* in *y koordinata*, ter druge, od tipa komponente odvisne podatke. Obravnavani preprosti grafični vmesnik definira le komponenti *gumb* z atributom *napis* in komponento *vnosno polje* z atributom *vnos*, preko katere je mogoče dobiti vnešeno vsebino vnosnega polja. Za dejansko uporaben grafični vmesnik bi bilo treba definirati še vse druge standardne komponente in možnost odziva na dogodke grafičnega vmesnika, kot je pritisk gumba in podobno.

Definicija (meta modela) modelnega jezika

Iz te osnovne definicije je treba definirati modelni jezik oziroma njegov meta model. Dejanski modeli predstavljenega preprostega grafičnega vmesnika vsebujejo več primerkov koncepta *Okno*. Koncept *Okno* opisuje zgoraj predstavljeni pojem *okno* z omenjenimi štirimi atributi. Meta model vsebuje abstraktni koncept *Komponenta* z atributoma *xKoordinata* in *yKoordinata*, ki ga razširjata (angl. extend) koncepta *Gumb* in *VnosnoPolje* z zgoraj definiranimi enako poimenovanimi atributi. Element *Okno* ima še funkcionalnost *izvedi*, ki zgradi in prikaže okno ter se izvaja, dokler ga uporabnik ne zapre. Res osnovnih funkcij, kot je omenjeno zapiranje okna, v modelnem jeziku ni nujno definirati, saj so le-te danes že popolnoma samoumevne in podprte v vseh knjižnicah za grafične vmesnike. Te osnovne funkcije so nekako že sestavni del abstraktnega nivoja ciljnega modela (knjižnic za grafične vmesnike programskih jezikov). Treba je še definirati *obveznost* atributov konceptov: vsi atributi pri vseh konceptih naj bodo obvezni; edina izjema je atribut *vnos* pri komponenti *VnosnoPolje*. Za nedvoumno definicijo bi bilo treba določiti še mnogo podrobnosti, kot na primer zahtevo, da se velikost komponent avtomatsko prilagaja njihovi vsebini, največje število znakov, ki jih je mogoče vnesti v vnosno polje in podobno, vendar bodo te podrobnosti zaradi omejenega prostora izpuščene. Pričujoči modelni jezik je v bistvu domensko specifičen modelni jezik za domeno grafičnega vmesnika. Modeli po tako definiranem meta modelu bodo vhod za preslikovalno funkcijo.

Konkretni zapis

Določiti je treba konkretni zapis za tako definiran domensko specifični modelni jezik. Kot je bilo pri obravnavi teorije že povedano, je možnih konkretnih notacij mnogo, za ta primer pa bo uporabljen kar tekstovni zapis. Tekstovni zapis je definiran z nekaj preprostimi pravili. Vsak koncept je opisan v eni vrstici, najprej mora biti navedeno ime koncepta, nato pa so v navadnih oklepajih, ločeni z vejico, v vrstnem redu iz definicije, našteti atributi in njihove vrednosti. Izjema temu pravilu je atribut *komponente* pri konceptu okna, ki se ga ne navaja v oklepajih, ampak po enega na vrstico za vrstico, ki definira primerek okna. Atribut in njegova vrednost morata biti zapisana v naslednjem formatu: najprej ime atributa (kot je definirano v definiciji), nato znak za enakost in za tem vrednost atributa. Če je vrednost atributa niz, je le-ta zapisan v enojnih navednicah. Opozoriti je treba še na problem nizov, ki vsebujejo enojno navednico ali druge ločevalne znake.

Za boljšo predstavo je na sliki 3.1 podan primer za pravkar opisano konkretno notacijo modela z enim oknom, ki vsebuje eno vnosno polje in en gumb.

Izbira ciljnega jezika in tehnologije

Naslednji korak je določanje ciljnega modelnega oziroma programskega jezika, ki bo v pričujočem primeru programski jezik Java. Z izbiro katerega drugega objektnega jezika bi bil nadaljnji potek precej podoben.

```
Okno(širina=250, višina=200, naslov='TestnoOkno')
VnosnoPolje(xKoordinata=50, yKoordinata=50)
Gumb(xKoordinata=50, yKoordinata=100, napis='Pritisni')
```

Slika 3.1: *Primer zapisa v razvitem domensko-specifičnem jeziku.*

V okviru izbranega programskega jezika je treba določiti tudi tehnologijo oziroma knjižnico, s katero bo izveden grafični vmesnik oziroma v katero bo preslikovalna funkcija preslikala modele. Za primer bo ob prejšnji odločitvi za Javo za grafični vmesnik uporabljena standardna Java knjižnica SWING. Lahko se seveda izdela več preslikovalnih funkcij v več tehnologij oziroma knjižnic.

Meta model ciljnega jezika

Treba je določiti tudi meta model za izbrani programski jezik Java. V duhu opisovanja modelno vodenega razvoja na najnižjem nivoju bo uporabljen preprost lasten meta model. Meta model ima koncept *Razred* z atributoma *atributi* in *metode*. Atribut *atributi* je seznam z elementi tipa *Atribut*, atribut *metode* pa seznam z elementi tipa *Metoda*. *Atribut* ima atributa *ime* tipa niz ter *tip* tipa *Razred*. *Metoda* ima atribute *ime* tipa niz, *tipRezultata* tipa *Razred*, ter zaradi enostavnosti še kar atribut *vsebina* tipa niz, ki predstavlja telo metode. Meta model je res zelo poenostavljen. Manjka mu veliko elementov uporabljenega programskega jezika, kot so paketi, vidljivost in podobno. Še največja poenostavitev je uporaba niza za vsebino metod, kar se hitro pokaže v praksi. Boljša rešitev bi bila definicija vseh možnih konstruktov vsebine metode kot elementov meta modela in definicija vsebine metode kot seznama teh konstruktov. Tako bi preslikovalna funkcija namesto lepljenja nizov gradila model iz omenjenih konstruktov. V pričujočem primeru je uporabljen preprostejši način, spet zaradi obsežnosti.

Definicija preslikovalne funkcije – prva različica

Tako je nastopil čas za najbolj zahteven del procesa: podrobno, natančno in nedvoumno definicijo preslikovalne funkcije iz primerka meta modela zgoraj definiranega modelnega jezika za preproste grafične vmesnike v primerek enostavnega Java meta modela.

Pri razvoju definicije preslikovalne funkcije je možnih več pristopov. Nekako deluje najbolj primerno pristop *z vrha navzdol* (angl. top down), saj je smiselno najprej razmisliti o vseh nastopajočih konceptih vhodnega in izhodnega meta modela ter ugotoviti korelacije med njimi in se šele nato spustiti v podrobnosti.

Pri obravnavanem primeru je smiselno vsakega od osnovnih treh konceptov vhodnega meta modela, *Okno*, *VnosnoPolje* in *Gumb*, preslikati v koncept *Razred* izhodnega meta modela z ustreznim imenom. Zadnja trditev je resnična, vendar ne najbolj relevantna. Pomembneje je, da je potrebno v ciljnem modelu narediti po en primerek koncepta *Razred* z imenom koncepta v vhodnem modelu. Drugače

rečeno, v vhodnem modelu obstaja več oken, gumbov in vnosnih polj; tako je za vhodni model z dvema oknoma z imeni *OknoPrijava* in *OknoPodatki* treba v izhodnem modelu narediti dva *Razreda*, kjer ima prvi ime *OknoPrijava*, drugi pa *OknoPodatki*. Tako bi prej omenjeni generirani osnovni trije *Razredi* za okno, gumb in vnosno polje lahko služili kot predniki tako generiranim primerkom.

Omenjeni razredi za osnovne tri koncepte pa morajo nujno razširjati razrede uporabljene grafične knjižnice, sicer generiranih razredov ne bo mogoče uporabiti. Celotna razredna hierarhija ne deluje najbolj idealno. Rešitev v tej smeri bi zahtevala dodajanje komponent okna v konstruktorju programske kode posameznega okna. Iz teh dveh razlogov trenutno razmišljanje ni najboljše. V takih trenutkih je smiselno začeti iskati alternativne rešitve.

Tako je s celotno definicijo do preslikave vse v redu, le v definiciji preslikave je bila ubrana manj primerna pot. Ta manj primerna pot in sam njen začetek s poskusom slepega preslikovanja elementov vhodnega meta modela *Okno*, *VnosnoPolje* in *Gumb* v razrede programskega jezika najverjetneje izhaja iz klasičnega MDA razumevanja modelno vodenega razvoja.

Definicija preslikovalne funkcije – boljša alternativa

Tako je treba ponovno razmisliti, kaj sploh je preslikovalna funkcija pri obravnavanem problemu grafičnega vmesnika in kaj je dejanski rezultat preslikave, kakšne vrste razredi programskega jezika. Za odgovore na ta vprašanja je treba pogledati, kakšen je namen razvijanega modelnega jezika. Na to vprašanje bi bilo treba odgovoriti že čisto na začetku, s čimer bi se verjetno bilo mogoče izogniti pomanjkljivostim definicije, vendar je takšne zaplete skoraj nemogoče predvideti.

Zastavljeni model grafičnega vmesnika v resnici služi za opis grafičnega vmesnika dejanske programske rešitve. Z njim je možno definirati grafična okna in komponente na njih, v zgornjem primeru definicije okna z enim vnosnim poljem in enim gumbom za vnos enega podatka.

Za dognanje, kakšne razrede v programskem jeziku bi morala preslikovalna funkcija zgraditi, je morda dobro razmišljati kar o tem, kako bi problem izgradnje posameznih grafičnih vmesnikov rešili v programskem jeziku, brez obstoja razredov za posamezne elemente grafičnega vmesnika v vhodnem modelu. Ena izmed bolj smiselnih rešitev bi bili razredi za *izgradnjo* (angl. builder) posameznih oken. Ker je v zastavljenih ciljih možno zgraditi več oken, bi bilo morda smiselno narediti še en krovni razred, *tovarno* (angl. factory) za izdelavo oken, ki bi imela metodo *zgradiOkno* z določeno identifikacijo okna, ki bi jo bilo treba dodati v definicijo. Zaradi preprostosti bo za to uporabljen kar atribut *naslov*, ki mora zaradi tega biti *edinstven* (angl. unique).

Ob odločitvi za rešitev generiranja razredov za izgradnjo oken ostane še vprašanje smiselnosti osnovnih treh razredov *Okno*, *VnosnoPolje* in *Gumb*. Obstajata dve možnosti: da jih ni (in razredi za izgradnjo oken vračajo kar dejanske razrede grafične knjižnice) ali pa, da so (in razširjajo razrede grafične knjižnice). Boljša rešitev je obstoj teh razredov; s tem je mogoče v njih hraniti določene

vrednosti, katerih ne podpirajo razredi grafične knjižnice, hkrati pa je rešitev bolj neodvisna od morebitnih sprememb grafične knjižnice.

Tako se postavljata dve vprašanji: kako naj te tri razrede preslikovalna funkcija naredi v izhodnem modelu in kateri elementi vhodnega (meta) modela so vhod v ta del preslikovalne funkcije. Ti trije razredi nimajo nobenega vhodnega parametra iz vhodnega meta modela. To z drugimi besedami pomeni, da so statični. Preslikovalna funkcija lahko te tri razrede naredi tako, da v izhodnem modelu naredi tri primerke koncepta *Razred* z ustreznimi vrednostmi atributov.

Večina naprednejših orodij, kot na primer v nadaljevanju predstavljeni OptimalJ, ponavadi uporabniku ne prikazujejo modela ciljnega programskega jezika, ampak kar direktno generirano programsko kodo. V tem smislu koraka preslikave in zapisovanja ciljnega modela navidezno združujejo v en sam korak. Iz tega izhaja ideja, da je možno te tri razrede napisati kar v programskem jeziku.

Pri direktnemu pisanju ciljnih razredov pa je treba paziti na dejstvo, da morajo ti razredi načeloma obstajati v ciljnem modelu, saj se s preslikovalno funkcijo oblikovani primerki koncepta *Razred* sklicujejo na te statične razrede. Slednji problem je mogoče rešiti na več načinov, na primer z oblikovanjem statičnih razredov v ciljnem modelu in označevanjem, da se jih ne sme zapisati v ciljno programsko kodo, ali pa z vodenjem posebnih meta podatkov o teh razredih.

Sprejeta je bila odločitev, da se te tri osnovne razrede napiše v programskem jeziku. Kljub temu bodo dodani v ciljni model v minimalni obliki, potrebni za delovanje zapisovanja tega modela v programsko kodo, saj se bodo na te razrede sklicevali generirani razredi. Tako je treba na nek način dodati možnost, da se nekateri primerki *Razred*-a ne zapišejo v programsko kodo, kar je, po principu dobrega načrtovanja zaključenih programskih celot, boljše izvesti zunaj meta modela, kot pa dodajati v meta model. Preprosta rešitev tega problema je vodenje seznama razredov, ki se ne smejo zapisati v programsko kodo.

Definicija preslikovalne funkcije skupaj z odločitvami o implementaciji tako obsega:

- pisanje razredov *Okno*, *VnosnoPolje* in *Gumb* direktno v programski kodi,
- za vsak element *Okno* iz vhodnega modela je treba narediti element *Razred* z imenom *KonstruktorOkna + naziv okna* in metodo *zgradiOkno* brez parametrov,
- v ciljnem modelu je treba oblikovati še razred TovarnaOken z metodo *zgradiOkno* s parametrom *nazivOkna*, ki seveda delegira izgradnjo okna v prejšnji alineji definiranim razredom za generiranje posameznega okna,
- ker imamo več razredov za konstruiranje oken, je smiselno v končno rešitev dodati še vmesnik (angl. interface) *KonstruktorOkna* z metodo *zgradiOkno*, katerega posamezni konstruktorji oken implementirajo.

Za zahtevo iz zadnje alineje je potrebno dopolniti meta model z vmesnikom in razširiti koncept *Razred* s seznamom vmesnikov katere implementira.

Implementacija

Po vseh tako sprejetih odločitvah je čas za implementacijo orodja. Pred tem je smiselno še enkrat dobro razmisliti in preveriti celotno definicijo.

Implementacija razredov *Okno*, *VnosnoPolje* in *Gumb* je načeloma enostavna. Treba je določiti ustrezne koncepte izbrane knjižnice SWING, s katerimi bodo implementirani ti trije razredi. Smiselna izbira so razredi *JDialog*, *JTextField* in *JButton*, vsi iz paketa *javax.swing*. Treba se je tudi odločiti na kakšen način bodo ti razredi uporabljeni. Morda je najbolj naravno razširjanje (angl. extend) izbranih SWING razredov v zahtevane, kar pa ima s stališča dobrega programskega aplikacijskega vmesnika tudi slabosti, ki jih rešuje kompozicija [3]. Dober aplikacijski vmesnik je kljub navidezni skritosti generirane kode zelo pomemben predvsem z vidika kombiniranja razvite rešitve z drugimi modelnimi jeziki. V razredu *Okno* je treba definirati splošno metodo *dodajKomponento*, ki oknu doda podano komponento. Ta metoda bo uporabljena v generiranih razredih za konstruiranje oken za samo gradnjo grafičnega vmesnika. S to funkcijo bodo dodane posamezne vizualne komponente na okno.

Implementacija vhodnega in izhodnega modela je precej trivialna. Treba je napisati le zgoraj definirane razrede z njihovimi atributi. Za imena razredov vhodnega modela je dobro izbrati imena, ki so drugačna od imen na roko napisanih razredov, sicer pride do problema po dveh razredov z istim imenom. Smiselno bi bilo pustiti imena na roko napisanih razredov *Okno*, *VnosnoPolje* in *Gumb*, in razrede, ki opisujejo elemente meta modela poimenovati z zapono *MetaElement*, kar da imena *OknoMetaElement*, *VnosnoPoljeMetaElement* in *GumbMetaElement*.

Komponenta za branje in razčlenjevanje definirane konkretne notacije vhodnega modela tudi ni posebej zapletena. Zapisovanje ciljnega modela programskega jezika Java v datoteke s programsko kodo tudi ni preveč zahtevno. Tako bo zaradi omejenega prostora podroben opis teh dveh delov izpuščen.

Tako ostane še najbolj zapleten del, implementacija preslikovalne funkcije. Le to je smiselno implementirati v razredu z imenom *GrafičniVmesnikPreslikava* z metodo *preslikaj*, ki ima za vhod vhodni model v obliki seznama primerkov razreda *OknoMetaElement*, za izhod pa izhodni model v obliki seznama primerkov razreda *Razred*, ki predstavljajo generirane *KonstruktorjeOken*.

Preslikovalna funkcija mora za vsak vhodni *OknoMetaElement* v rezultat dodati ustrezno generiran *Razred*. Takšnemu *Razred*u je treba nastaviti ime ter dodati metodo *zgradiOkno* z ustrezno vsebino. Vsebina metode se začne s konstruiranjem primerka na roko napisanega razreda *Okno*. Nato se je treba sprehoditi čez vse komponente vhodnega modela okna in za vsako v odvisnosti od tipa komponente dodati vrstico programske kode, ki k narejenemu primerku okna doda ustrezen primerek na roko napisane komponente z ustreznimi parametri. Če na primer v vhodnem modelu pri obravnavanemu *OknoMetaElement* generator naleti na komponento tipa *GumbMetaElement* z *napisom* 'Pritisni' in koordinatama (10, 20), mora v generirano kodo dodati vrstico, ki bo konstruiranemu primerku okna v prvi vrstici z metodo *dodajKomponento* dodala primerek komponente *Gumb* z

napisom 'Pritisni' in koordinatama (10, 20). Na koncu je treba le še dodati vrstico kode, ki vrne zgrajeni primerek *Okna*.

Rezultat generiranja za zgoraj navedeni primer testnega okna za konkretno notacijo je na sliki 3.2, rezultat uporabe tako generirane programske kode pa na sliki 3.3.

```
public class KonstruktorOknaTestnoOkno {
  public Okno zgradiOkno() {
    Okno okno = new Okno("TestnoOkno", 250, 200);
    okno.dodajKomponento(50, 50, new VnosnoPolje());
    okno.dodajKomponento(50, 100, new Gumb("Pritisni"));
    return okno;
  }
}
```

Slika 3.2: *Generirana programska koda iz vhodnega modela.*

Slika 3.3: *Izgled grafičnega okna izrisanega z zaganjanjem generirane programske kode.*

Končni komentarji

Veliko stvari bi se pri danem primeru dalo narediti drugače. To velja za vse faze razvoja predstavljenega modelnega jezika in orodja, od začetnih definicij do implementacije končnega orodja. Predvsem je treba posvečati pozornost oblikovanju zaokroženih smiselnih modelnih jezikov za primerno obsežne domene ter paziti na upoštevanje načel načrtovanja dobrih aplikacijskih vmesnikov. Priporočila so še posebej pomembna pri izdelavi ponovno uporabnih modelnih jezikov in orodij zanje.

Zelo priporočljivo je ločevati omenjene sklope orodja, od razčlenjevanja modela v vhodni notaciji do zapisovanja izhodnega meta modela. Ločevanje je dobro zaradi razbijanja kompleksnega problema na manjše in bolj obvladljive sklope in možnosti ponovne uporabe posameznih sklopov. Seveda je smiselno uporabiti že obstoječe rešitve, predvsem to velja za ciljne meta modele in njihovo zapisovanje.

Programska koda večine sklopov orodja je relativno enostavna. Po kompleksnosti zelo izstopa le preslikovalna funkcija. Treba pa se je zavedati obširnosti zastavljenih sklopov in izbrati za dani problem ravno dovolj široko različico. V pričujočem primeru je bila na primer sprejeta odločitev, da se za vsebino metod uporabi kar niz. Tako je bilo v preslikovalni funkciji za izgradnjo okna treba niz z vsebino metode sestaviti s seštevanjem nizov posameznih vrstic programske kode. Če bi bil za vsebino metode namesto niza uporabljen meta model, ki bi podpiral vse možne konstrukte vsebine metod, bi bilo sestavljanje vsebine veliko lažje in preglednejše, je pa z razvojem takega meta modela in njegovim zapisovanjem ogromno dela.

Preslikovalne funkcije zelo hitro postanejo preveč zapletene. V praksi se modelne jezike in orodja razvija iterativno, kar pomeni razširjanje modelnega jezika in orodja ob ugotovitvi, da trenutna različica nečesa ne podpira. Tako se skozi daljši čas postopoma dodaja podpora za razne nove zmožnosti in posebnosti. S tem modelni jezik postane zelo obsežen in manj razumljiv, preslikovalne funkcije pa lahko hitro neobvladljive.

Moč oziroma uporabnost modelnega jezika je po enem kriteriju toliko večja, kolikor enostavnejši in kompaktnejši je vhodni zapis v primerjavi z izhodnim. Tako je smiselno oblikovati srednje ali visoko specializirani modelni jezik za grafični vmesnik s podporo bolj specializiranim komponentam, ki v rešitvah pogosto nastopajo. Včasih je izdelani modelni jezik bolj kompakten, ker za vrednosti nekaterih nastavitvenih parametrov uporablja vnaprej definirane vrednosti, ki bi jih sicer bilo treba navesti v programski kodi.

Podan primer je res enostaven in tako ni posebno impresiven. Za praktično uporabnost predstavljenega modelnega jezika bi bilo treba podpreti vse standardne komponente grafičnega vmesnika ter morda uporabiti še katero izmed v tem razdelku predstavljenih idej.

Predstavljeni osnovi modelnega jezika za grafični vmesnik bi bilo nujno treba dodati še možnost povezovanja grafičnega vmesnika s poslovno logiko. Grafični vmesnik in poslovna logika morata biti kar se da ločena, saj gre za dva vidika rešitve, kjer je grafični vmesnik lahko le eden izmed možnih načinov proženja poslovne logike. Tako mora grafični vmesnik le omogočati specifikacijo poslovne logike, ki se proži ob aktiviranju njegovih posameznih komponent, poslovna logika pa mora nastavljanje vrednosti komponent grafičnega vmesnika izvesti na način, ki ni preveč vezan na tehnologijo grafičnega vmesnika, na primer z vzorcem *model-pogled-nadzornik* (angl. model-view-controller).

S tem je zaključena obravnava modelno vodenega razvoja in razvoja modelnih jezikov ter orodij čisto od začetka. V nadaljnjih razdelkih sledi obravnava različnih orodij, ki v bistvu le olajšajo določene izmed vseh tu obravnavanih konceptov in postopkov na poti od ideje, preko orodij, do končne programske rešitve.

3.2 Nizkonivojska podpora

Pri večini projektov razvoja programske opreme po modelno vodenem pristopu ni smiselno razvijati vseh podpornih orodij, vsaj v celoti ne. Kot pri drugih pristopih je treba glede na naravo in posebnosti posameznega projekta izbrati najprimernejša pol-orodja, orodja in pristope in z njimi oblikovati oziroma razviti okolje, v katerem nato poteka modelno vodeni razvoj oziroma modeliranje[4].

Pojem *nizkonivojska podpora* v kontekstu tega dela obsega široko množico pol-orodij, programskih knjižnic in konceptov z izjemo celovitih integriranih razvojnih okolji, ki bodo predstavljene posebej. Sem spadajo:

- razne programske knjižnice in druge elementarne rešitve, ki olajšajo ravno predstavljen razvoj lastnih orodij,
- razne celovite rešitve, ki se osredotočajo na reševanje posameznih vidikov modelnega razvoja,
- razna ogrodja (angl. framework), ki jih je možno prilagoditi z nastavitvami in lastnimi dodatnimi moduli.

Te tri skupine bodo v nadaljevanju podrobneje predstavljene s primeri orodij. Predstavljena delitev služi le za organizacijo zelo obširne množice orodij in idej v smiselne predstavitvene sklope in tako ne predstavlja nobene standardne kategorizacije.

3.2.1 Knjižnice ter druge elementarne ideje in rešitve

Pri razvoju določene komponente programske opreme se je smiselno vedno vprašati, ali je določeno komponento res treba razviti, ali morebiti že obstaja ustrezna splošna komponenta, ki se jo da uporabiti direktno ali s prilagoditvijo.

Knjižnice za ciljne meta modele

Največja verjetnost za obstoj standardne in splošno uporabne komponente pri razvoju lastnih orodij za podporo modelnemu pristopu je na področju ciljnega meta modela, torej meta modela rezultata preslikovalne funkcije; še posebej če je to eden izmed standardnih programskih jezikov. Za veliko uporabljane programske jezike ponavadi obstajajo knjižnice za meta modele teh jezikov ter za zapisovanje primerkov meta modela v programsko kodo in obratno. Kot se je pri predstavitvi izdelave lastnega orodja izkazalo, na začetku izgleda, da je potreben le manjši del meta modela ciljnega programskega jezika, s časom pa se začetna ocena izkaže kot pomanjkljiva. V podanem primeru se je izkazalo, da je bila predstavitev vsebine metod z nizom prevelika poenostavitev, ki je dodatno zapletla preslikovalno funkcijo, v kateri se je bilo treba ukvarjati s podrobnim zapisom programskega jezika.

[4]Povedano precej spominja na ideje tovarn programske opreme, opisane v razdelku 1.3.4.

Pri praktičnem spoznavanju modelno vodenega razvoja oziroma pisanja lastnih orodij za modelno vodeni razvoj, kjer je bila za ciljni jezik izbrana Java, je bilo na voljo kar nekaj programskih knjižnic za ta programski jezik. Mnogo izmed teh knjižnic je bilo razvitih v akademskih krogih za specifične raziskovalne namene, nekatere druge niso bile dovolj popolne ali pa so se nanašale na starejše različice Jave. Edina dobra knjižnica je bila osnova rešitve Spoon[5] [28, 27]. Osnova rešitve Spoon (imenovana spoon-core) obsega celoten meta model Jave za različico 1.5. Omogoča izgradnjo modelov sestavljenih iz paketov in razredov z vsemi pripadajočimi elementi ter pretvorbo teh modelov v nize, ki jih je treba le še zapisati v datoteke. Glej primer na sliki 3.4.

Podpora za tekstovne predloge

Drugo zelo dobro podprto področje je podpora generiranju programske kode po že predstavljenih *tekstovnih predlogah*. Predloga za generiranje programske kode je navadna tekstovna datoteka, katere vsebina je sestavljena iz statičnega teksta, ki je enak vsem generiranim datotekam, in iz dinamičnih odsekov, ki v splošnem določajo splošna navodila za izgradnjo končnih generiranih primerkov. Navodila za gradnjo dinamičnih delov se nanašajo na vhodni meta model generiranja in imajo v različnih rešitvah generiranja tekstovnih datotek iz predlog različno izrazno moč. Področje generiranja na podlagi predlog je zelo podprto zaradi širše uporabnosti principa, na primer pri gradnji dinamičnih spletnih strani.

V praksi smo preizkusili programsko knjižnico StringTemplate [26]. Za uporabo te knjižnice je treba najprej napisati predlogo v obliki tekstovne datoteke, ki vsebuje dinamične dele v predpisanemu formatu. V lastnem orodju je možno s pomočjo omenjene knjižnice to predlogo uporabiti z aktivacijo preprostega razreda, kjer je za parametre treba navesti datoteko predloge in vrednosti, iz katerih se izračunajo dinamični deli predloge. Dinamični deli so relativno preprosti, da se navesti podane vrednosti, poleg tega pa obstaja še nekaj enostavnih konstruktov, kot so pogoji, sprehajanje čez sezname, navigacija do vrednosti posameznih atributov in podobno. Te elemente dinamičnih delov se da med seboj kombinirati, s čimer se da, na primer, v rezultat pogojno izpisati določeno podano vrednost, v odvisnosti od neke druge podane vrednosti. Rešitev je kljub svoji enostavnosti precej uporabna.

Uporaba predlog pri razvoju orodij za modelno vodeni razvoj je smiselna pri reševanju problemov, katerih rezultat že po naravi ustreza predlogam. Predloge

[5] Celotna rešitev Spoon je namenjena manipulaciji programske kode tik pred prevajanjem. Prevajalnik najprej iz izvornih datotek zgradi abstraktno sintaktično drevo (angl. AST – Abstract Syntax Tree), ki ga je z uporabo Spoon-a možno manipulirati. Ta manipulacija omogoča odkrivanje ter dodajanje in odvzemanje razredov, vmesnikov, metod, atributov in drugih elementov. Tako spremenjen program se nato še prevede. Obstajajo tudi podobne rešitve za manipuliranje že prevedene programske kode. Ponavadi se te rešitve uporabljajo za dodajanje posebnih vidikov oziroma aspektov programom. To so vsekakor zelo zanimive ideje z vidika modelno vodenega razvoja.

```java
public class SpoonPrimer {
    public static void main(String... args) {
        // skupno
        Factory tovarna = new Factory(new DefaultCoreFactory(),
            new StandardEnvironment());
        Set<ModifierKind> javen = new HashSet<ModifierKind>(){{add(ModifierKind.PUBLIC);}};
        CtTypeReference<Object> nizTip = tovarna.Type().createReference("java.lang.String");

        // razred
        CtClass razred = tovarna.Class().create("Oseba");
        razred.setModifiers(javen);

        // dva atributa
        tovarna.Field().create(razred, javen, nizTip, "ime");
        tovarna.Field().create(razred, javen, nizTip, "priimek");

        // metoda
        final CtCodeSnippetStatement vsebina =
            tovarna.Code().createCodeSnippetStatement("return this.ime + \" \" +  this.priimek");
        CtBlock<Object> vsebinaMetode = tovarna.Core().createBlock();
        vsebinaMetode.setStatements(new ArrayList<CtStatement>(){{add(vsebina);}});
        tovarna.Method().create(razred, javen, nizTip, "vrniPolnoIme", null, null,
            vsebinaMetode);

        // izpis
        System.out.println(razred.toString());
    }
}
```

```java
public class Oseba {
    public java.lang.String ime;

    public java.lang.String priimek;

    public java.lang.String vrniPolnoIme() {
        return this.ime + " " +  this.priimek;
    }

}
```

Slika 3.4: *Primer grajenja razreda z uporabo knjižnice Spoon. Najprej se izdela nov primerek razreda z imenom* Oseba, *temu pa se doda dva atributa* ime *in* priimek *tipa* niz *in metodo, ki vrne sestavljeno polno ime. Tako zgrajeni razred se na koncu primera še izpiše, rezultat je viden v spodnjem delu slike.*

so tako primerne za generiranje programske kode, kjer so si generirani razredi programske kode med seboj vsaj strukturno zelo podobni.

Primer standardnega problema, primernega za reševanje z uporabo predlog, je generiranje že omenjenih razredov za prenos podatkov, ki se uporabljajo za prenos podatkov (angl. Data Transfer Objects) med programskimi moduli in njihovo okolico[6]. Ti razredi za prenos podatkov so po imenu razreda in njegovih atributih zelo podobni osnovnim internim razredom modula. Posebne razrede za prenos podatkov se oblikuje zato, da se zunanjemu svetu razkrije le minimalno potrebni aplikacijski programski vmesnik (angl. Application Programming Interface) brez napačnega in nepotrebnega razkrivanja internih podrobnosti sistema.

Take razrede je možno zgraditi z lastnim orodjem, kateremu se poda le seznam razredov in predlogo. Za izdelavo takšnega orodja je treba najprej definirati poseben relativno preprost meta model, ki omogoča opis množice razredov z njihovimi atributi. Model lahko obsega le razrede z njihovimi imeni in seznamom atributov; pri atributih pa so pomembna le njihova imena in tipi. Samo orodje iz seznama razredov z uporabo v jezik vgrajenih zmožnosti ugotavljanja meta podatkov razredov[7] zgradi ravno predstavljeni model, čez katerega se orodje sprehodi in za vsak razred modela sproži generiranje ciljnega razreda po predlogi, katere dinamični deli se izračunajo iz podatkov posameznega razreda. V predlogo se da vključiti dinamični del, ki za vsak atribut razreda iz modela generira programsko kodo za deklaracijo atributa z enakim ali podobnim imenom in privatnim dostopom ter standardni metodi za dostop do tega atributa. Enostavnost izvedbe dokazuje tudi enostavna izvedba opisane ideje na sliki 3.5.

Tako orodje je precej standardno in ponovno uporabno. Z njim je možno z zamenjavo predloge generirati različne množice razredov. Bolj splošna abstraktna različica ravno opisanega orodja se od opisane razlikuje po načinu pridobitve modela. Še bolj splošna različica ima čisto splošen model in iz njega ne generira nujno programske kode, ampak druge tekstovne vsebine, ki so lahko sestavni del programske rešitve.

Splošno o uporabi opisanih pripomočkov

Pri uporabi raznih knjižnic v lastnih orodjih je predvsem pomembno, da se jih vključi na način, ki čim manj meša programsko kodo orodja in knjižnice. Eden izmed poglavitnejših razlogov za takšno vključitev je možnost zamenjave uporabljene knjižnice v prihodnosti, minimalna prepletenost pa je smiselna že zaradi ločevanja logike orodja od ponavadi nižjenivojskih zmožnosti, ki jih ponuja knjižnica.

Za čim manjše prepletanje osnovne kode s tisto iz zunanjih knjižnic se lahko uporabi razne standardne pristope in vzorce, kot na primer tovarno primerkov razredov, ki jo uporablja osnovna koda za kreiranje primerkov razredov upora-

[6] Na primer med strežnikom in odjemalcem ali za sinhronizacijo z zunanjimi sistemi.

[7] Omenjena zmožnost se v angleščini imenuje *reflection*, njen direkten prevod *refleksija* pa je v družboslovnem smislu precej idealen.

```
public class Razred {
    public String ime;
    public List<Atribut> atributi =
        new ArrayList<Atribut>();
}
public class Atribut {
    public String ime;
    public Class tip;
}
```

```
group preslikava;
razred(ime, atributi) ::= <<
// Generirana koda
public class $ime$Prenos {
   $atributi:atr();separator="\n"$
}>>
atr(a) ::= <<$a.tip.name$ $a.ime$Vrednost;>>
```

```
public class Orodje {
    private Razred razcleniMetapodatke(Class clazz) {
        Razred razred = new Razred();
        razred.ime = clazz.getSimpleName();
        for (Field field : clazz.getFields()) {
            Atribut atribut = new Atribut();
            atribut.ime = field.getName();
            atribut.tip = field.getType();
            razred.atributi.add(atribut);
        }
        return razred;
    }

    private String preslikaj(Razred razred, String predloga) {
        StringTemplateGroup group = new StringTemplateGroup(
            new StringReader(predloga), DefaultTemplateLexer.class);
        StringTemplate template = group.getInstanceOf("razred");
        template.setAttribute("ime", razred.ime);
        template.setAttribute("atributi", razred.atributi);
        return template.toString();
    }

    public static void main(String... args) {
        Orodje orodje = new Orodje();
        System.out.println(orodje.preslikaj(orodje.razcleniMetapodatke(Oseba.class),
            orodje.preberiDatoteko("preslikava.txt")));
    }

    private String preberiDatoteko(String imeDatoteke)
    {...}
}
```

```
public class Oseba {
    public String ime;
    public int starost;
}
```

```
// Generirana koda
public class OsebaPrenos {
   java.lang.String imeVrednost;
   int starostVrednost;
}
```

Slika 3.5: *Primer orodja z uporabo tekstovnih predlog. V zgornji levi tretjini sta podana razreda Razred in Atribut, ki predstavljata meta model rešitve, na desni pa preslikava za knjižnico StringTemplate za delo s predlogami. V sredini je podano splošno orodje z metodama za razčlenjevanje Java razredov v Razred-e in za preslikavo tako razčlenjenih razredov po podani predlogi. V glavni metodi razreda je definiran zagon orodja za testni razred. V spodnjem delu slike je podan testni razred Oseba ter rezultat preslikave tega razreda v razred OsebaPrenos. S spremembo predloge je možno orodje uporabiti za poljubne preslikave, seveda v okviru podanega precej skromnega meta modela.*

bljene knjižnice. Ena možnost je tudi razvoj lastnih razredov, ki sovpadajo s 'podatkovnimi' razredi knjižnice in le delegirajo ukaze tem ustreznim sovpadajočim razredom knjižnice.

Doseganje te minimalne prepletenosti s podanimi in drugimi standardnimi vzorci je enostavno in hitro izvedljivo pri predlogah, kjer je programski vmesnik takih knjižnic zelo majhen. Pri knjižnicah za meta modele pa doseganje popolnega ločevanja kode ponavadi ni smiselno, saj je zaradi velikega števila elementov meta modela takšna ločevalna koda hitro preobsežna. Ne glede na stopnjo ločevanja programske kode orodja in knjižnice pa se je dobro zavedati prednosti ločevanja in morebitnih težav zaradi neločevanja v prihodnosti.

3.2.2 Celovitejše rešitve za posamezne vidike

Za gradnjo orodij za modelno vodeni razvoj obstaja mnogo dobrih rešitev, ki celovito podpirajo določen vidik orodja. Ta orodja se nanašajo na že opisane splošne korake orodja od branja konkretnega zapisa vhodnega meta modela preko preslikovalne funkcije do zapisovanja konkretnega zapisa izhodnega meta modela. V nadaljevanju bo predstavljena ena izmed takšnih zanimivih rešitev.

Generic Modeling Environment

Generic Modeling Environment [2, 18] (v nadaljevanju GME) je prilagodljivo orodje za domensko-specifično modeliranje. V osnovi je GME nastal kot odziv na pomanjkanje orodij za grafično modeliranje specifičnih, manj pogosto uporabljanih domen.

Prilagoditev orodja se izvede z definicijo meta modela domensko-specifičnega jezika. Tako prilagojeno orodje je nato možno uporabljati za izdelavo modelov, ki ustrezajo prej definiranemu meta modelu. Pri gradnji GME meta modelov ima uporabnik preko bogatega nabora konceptov možnost zelo natančne definicije domenskega jezika z vidika zapisa, pomena in tudi predstavitve.

Osnovna enota v GME je *projekt* (angl. project) z več *načrti* (angl. maps), ki so mehanizem za smiselno organizacijo *modelov*. Modeli so sestavljeni iz tako imenovanih *objektov prvega reda* (angl. first class objects), to so *atomi*, *reference*, *povezave* in *množice*. *Atomi* so elementarni objekti, ki ne morejo vsebovati drugih elementov. *Povezave* (angl. connections) so mehanizem za povezovanje objektov v modelu. *Reference*, podobno kot kazalci v objektno usmerjenih jezikih, predstavljajo kazalec na druge objekte prvega reda, ki lahko nastopajo v modelu. *Množice* (angl. sets) pa so splošne množice poljubnih elementov modela.

Na voljo je še nekaj dodatnih konceptov za natančnejše določanje meta modelov. Nad opisanimi osnovnimi gradniki lahko definiramo *omejitve* (angl. constraints). Za določanje vidljivosti elementov se uporablja *aspekte*. Gradniki imajo v modelih definirane tudi *vloge* (angl. role), elementom modela pa se da določiti poljubne *atribute*.

Poleg osnovnih gradnikov lahko modeli kot elemente vsebujejo tudi druge

modele istega ali drugega meta modela. Izdelani modeli postanejo *tipi* (angl. type), podobno kot razredi v objektno usmerjenih jezikih. Te tipe je mogoče tudi razširjati (angl. extend) in zanje oblikovati primerke (angl. instantiate). Predstavljeni koncepti veliko pripomorejo k izrazni moči rezultatov GME-ja, z njimi pa je mogoče oblikovati ponovno uporabne zbirke modelov.

Meta modele za specifične domene se definira v klasičnem UML-ju z razrednimi diagrami in z jezikom OCL za določanje omejitev. Za določanje predstavitvenega vidika in še nekaterih drugih podrobnosti se uporabljajo standardne razširitvene možnosti UML-ja. Tak meta model se nato prevede v konfiguracijo GME okolja, ki omogoča modeliranje v tako definirani domeni.

GME je trenutno eno najbolj splošnih orodij za oblikovanje okolij za domensko specifično modeliranje. V praksi je bil uporabljen za zelo različne namene, kot na primer za modeliranje proizvodnje od najvišjega nivoja do posameznih strojev, v grafičnih razvojnih okoljih, za modeliranje električnih vezij in podobno.

GME je ravno zaradi opisane splošnosti zelo idealen za podporo domensko-specifičnim jezikom pri modelno vodenem razvoju. Za zamišljeni domensko-specifični jezik je treba v orodju najprej izdelati njegov meta model, ki orodje iz splošnega modelnega orodja prilagodi v orodje za modeliranje domene določene z meta modelom. Z uporabo GME-ja je tako z relativno malo truda možno priti do precej dobrega orodja za modeliranje v lastnih domensko-specifičnih jezikih.

S stališča uporabe je pomembno tudi vprašanje branja tako izdelanih modelov v lastno orodje, ki te modele naprej preslika po preslikovalni funkciji. GME je zasnovan modularno in omogoča dodajanje raznih lastnih razširitev. Tako poleg splošnih možnostih razširjanja omogoča celo več načinov in formatov dostopa to teh modelov.

GME se v praksi, po začetnem privajanju na vse opisane koncepte, izkaže zelo dobro. Zelo dobrodošla je možnost prilagoditve predstavitve z določanjem manjših slik za grafično predstavitev posameznih konceptov meta modela.

Kljub temu, da GME omogoča izdelavo poljubnih meta modelov, pa seveda ne rešuje vsebinskih problemov, in tako na primer ne ponuja rešitev za probleme, kot je modeliranje obnašanja (angl. behavior) oziroma algoritmov. Ravno pri razmišljanju o modeliranju obnašanja in podobnih stvari pa se postavlja tudi vprašanje prikladnosti grafičnih modelov za modeliranje poljubnih domen oziroma vidikov.

3.2.3 Ogrodja

Iz pisanja lastnih rešitev v celoti oziroma že iz teorije modelno vodenega razvoja je hitro razbrati, da orodja za podporo modelno vodenemu razvoju sledijo določenim pravilom oziroma korakom. Vhod je model v določenem jeziku, ta model je treba prebrati in razčleniti, s preslikovalno funkcijo preslikati v ciljni model in tega zapisati. Iz tega spoznanja se je razvilo več *ogrodij* (angl. framework), ki nekatere izmed opisanih korakov poenostavljajo ali rešujejo v celoti ter omogočajo povezovanje posameznih korakov v celovito orodje.

Nekatera izmed teh ogrodij so precej splošna, nekatera pa postavljajo določene omejitve, ki ob zmanjšanju izraznih možnosti poenostavljajo razvoj orodij. Pogosto ogrodja omejujejo vhodni modelni jezik na UML ter izhodni jezik na izbrani programski jezik. Tako se izrazne zmožnosti zmanjšajo, hkrati pa je za konkretizacijo ogrodja v orodje treba definirati le preslikovalno funkcijo poleg definicije vhodnega modelnega jezika v smislu omejitve UML-ja.

Ogrodje AndroMDA

Eno izmed prvih in hkrati tudi bolj znanih ogrodij za izgradnjo orodij za podporo modelno vodenemu razvoju je AndroMDA [36]. Že iz imena je videti, da gre v osnovi za podporo MDA pristopu. Ciljni programski jezik je poljuben, vendar poleg osnovne splošne rešitve AndroMDA vsebuje že znaten nabor razširitev za programski jezik Java.

AndroMDA je v osnovi ogrodje za generiranje programske kode. Ta osnova za generiranje določenih sklopov programske kode uporablja posebne *module* (angl. cartidges). V osnovno različico AndroMDA je vključenih nekaj standardnih modulov za najpogosteje uporabljane tehnologije za programski jezik Java, kot so EJB, Spring, Hibernate [37], Struts [38] in podobne. Poleg možnosti minimalnega prilagajanja generiranja preko spreminjanja nastavitev obstoječih modulov je zelo zanimiva možnost razvoja lastnih modulov.

AndroMDA za generiranje programske kode uporablja že obravnavane *predloge*. Za *stroj za obdelavo predlog* (angl. template engine) uporablja zelo znano rešitev Velocity [39]. Predloge vsebujejo dinamične dele, ki se nanašajo na elemente določenega diagrama vhodnega UML modela. Zaradi že opisanih prednosti ločevanja programske kode, v tem primeru indirektnega dostopa do vhodnega modela, v predlogah v resnici dostopamo do tako imenovanih *meta-fasad* (angl. meta-facade), ki na kontroliran način razkrivajo elemente meta modela in njihove lastnosti. Predloga se aplicira na določene elemente vhodnega modela, označene z ustreznim UML stereotipom. Definicija lastnega modula tako obsega definicijo večih predlog, kjer je treba vsaki določiti stereotip, na katerega se proži. Možno je napisati tudi lastne posebne meta-fasade. Dodatno napisane module je treba na predpisan način vključiti med obstoječe module.

Celoten postopek generiranja se pri AndroMDA začne z branjem vhodnih datotek z UML modeli v XMI obliki in razčlenjevanjem teh datotek. Nato se ogrodje, ki že ima naložene vse nastavljene module, sprehodi po elementih vhodnih modelov in glede na stereotipe elementov izvede generiranje datotek s programsko kodo po predpisani predlogi.

AndroMDA je zelo cenjeno in razširjeno ogrodje. Uporablja se tako direktno na realnih in velikih projektih, kot tudi kot osnova za višjenivojska splošno uporabna ali razširljiva ogrodja in orodja. Uporaba v praksi vsaj v začetku ni preveč enostavna, predvsem zaradi relativno zapletenih nastavitev, razpršenih preko množice nastavitvenih datotek in zaradi zaganjanja iz ukazne vrstice. Orodje je brezplačno, zato mu je to pomanjkljivost lažje odpustiti. Pisanje lastnih razširitev je tudi re-

lativno zapleteno, kar pa je za razliko od zapletenosti uporabe bolj razumljivo.

Samo od sebe se seveda postavlja vprašanje uporabnosti ogrodja AndroMDA. Obstoječi moduli generirajo precej trivialne razrede. Na primer, modul za preslikavo med objekti in relacijsko podatkovno bazo (angl. object-relation mapping) za znano knjižnico Hibernate iz razrednih diagramov za vsak razred z ustreznim stereotipom generira Java razred, primeren za uporabo v omenjeni knjižnici. Tako generiran razred je precej preprost, vsebuje le privatne spremenljivke, ki ustrezajo atributom UML razreda ter standardne metode za dostop do teh atributov. Omenjena knjižnica Hibernate zelo dobro izkorišča že predstavljeni princip *privzetih vrednosti*. Tako so na primer imena objektov v podatkovni bazi določena iz imen razredov in določenih spremenljivk. Večino privzetih vrednosti, kot na primer ime atributa v podatkovni bazi, pa se da po želji posebej določiti. V programski kodi se take vrednosti določi z *zaznamkom*. Ker je bilo treba za popolno podporo podpreti tudi te posebej določene vrednosti, so snovalci obravnavanega AndroMDA modula uporabnikom dali na voljo določanje teh vrednosti preko standardnih načinov razširjanja UML-ja. Tako je celotna specifikacija AndroMDA modula za Hibernate v bistvu po izraznih možnostih enaka specifikaciji samih zmožnosti knjižnice Hibernate.

Kaj to pove o uporabnosti obravnavanega AndroMDA modula za Hibernate? Hitro lahko vidimo, da gre le za drugi konkreten zapis istega modelnega jezika. UML zapis v obliki enega UML modela, kjer hkrati lahko vidimo več razredov z njihovimi atributi, pa je v primerjavi z več datotekami s programsko kodo neprimerno boljši – predvsem zaradi svoje kompaktnosti in grafičnega konkretnega zapisa. Ta argument malo zbledi, ko je potrebno prilagoditi veliko privzetih vrednosti, saj je prikazovanje razširitvenih elementov UML diagramov včasih problematično. O dvigu nivoja abstrakcije tu ne more biti govora, saj gre izključno za drugo konkretno notacijo. Te ugotovitve v bistvu le dokazujejo, da je področje ki ga pokriva obravnavani modul, teoretično že zelo zrelo.

Zanimivejši je na primer modul za znano programsko knjižnico Spring, katere osnova je podpora že opisani inverziji kontrole. Tu je možno definirati *storitve* (angl. services) nad entitetnimi razredi, ki predstavljajo smiselne zaključene operacije, ki jih ponuja sistem, ponavadi strežnik v navezi strežnik-odjemalec. Storitve so UML operacije na UML razredih, ki so označeni s stereotipom ≪*Service*≫. Vsebina storitev se določi s standardnim jezikom OCL za podajanje omejitev v UML diagramih. OCL izrazi se v procesu generiranja kode preslikajo v poizvedbe na podatkovno bazo. Modul omogoča tudi definicijo že omenjanih razredov za prenos podatkov med moduli in določanje teh razredov za prenos podatkov za tip rezultatov storitvenih operacij.

Z opisano možnostjo deklarativnega oblikovanja storitve v UML diagramu pa je definitivno dosežen dvig abstraktnega nivoja. Po drugi strani pa samo s pretvorbo OCL izrazov v poizvedbe za podatkovno bazo ni možno definirati vse poslovne logike. Spring modul se ne ukvarja z obstojnostjo entitetnih razredov, ta problem je že rešen v modulu Hibernate ali drugem podobnem modulu za isti namen.

Iz proučevanja AndroMDA se da veliko naučiti. Zanimiva je rešitev pove-

zovanja rezultatov posameznih modulov. Gre za najbolj naravno različico, kjer so posamezni moduli zasnovani tako, da njihova generirana programska koda uporablja programsko kodo, ki jo generirajo drugi moduli. Tako na primer predstavljeni modul Spring uporablja rezultate modula Hibernate. Za določeno področje, kot je dostop do podatkovne baze, obstaja celo več različnih med seboj zamenljivih modulov, ki v ozadju uporabljajo različne programske knjižnice. Z drugimi besedami to pomeni, da so med moduli definirane določene odvisnosti, nekakšni nenapisani vmesniki, katerih uporabnik, ki le modelira, sploh ne zaznava, saj se v procesu generiranja vse lepo izide.

Sumarno gledano je AndroMDA skupaj s standardnimi moduli kar dobra osnova za razvoj programskih rešitev. Res veliko stvari se da rešiti s podprtim modeliranjem. Kot pri vseh današnjih orodjih za podporo modelno usmerjenemu razvoju je najbolj šibko področje AndroMDA specifikacija obnašanja oziroma postopkov in raznih specifičnih domen. Za specifikacijo postopkov sicer obstaja poseben modul, ki podpira diagrame aktivnosti, vendar je bilo že pri obravnavi UML-ja ugotovljeno, da noben UML diagram ni najbolj idealen za specifikacijo algoritmov.

3.3 Visokonivojska podpora

Najbolj celovito podporo modelno vodenemu razvoju predstavljajo celovita razvojna okolja, ki se v veliko vidikih že približujejo idealu modelno vodenega razvoja. Omogočajo torej modeliranje in vsebujejo 'magični' gumb, ki iz modelov izdela izvedljiv program in ga požene. V realnosti se orodja temu idealu zelo približajo, predvsem v najbolj podprtem segmentu relativno nezapletenih klasičnih informacijskih sistemov za podporo poslovanju, vendar je treba določene vidike še vedno razviti v klasičnih programskih jezikih ali orodje prilagoditi za podporo tem vidikom.

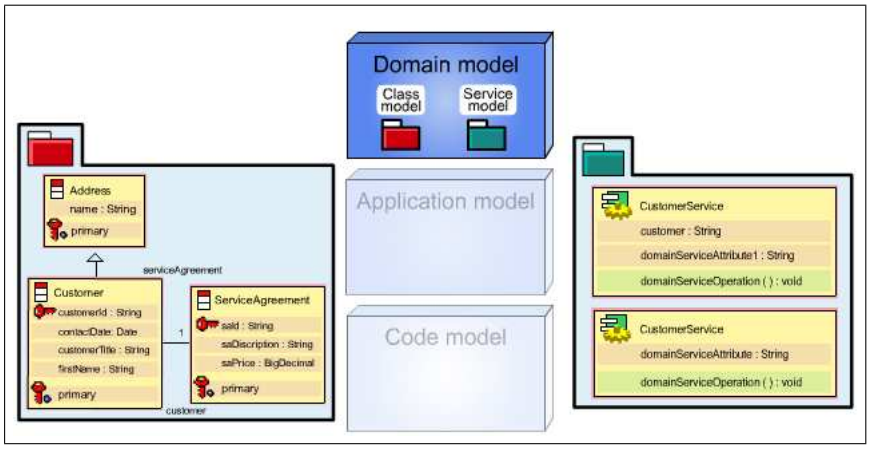
3.3.1 Razvojno okolje OptimalJ

Eno najbolj cenjenih celovitih razvojnih okolij za podporo razvoju programske opreme po modelno vodenem pristopu je OptimalJ [40]. OptimalJ precej striktno sledi MDA definiciji modelno vodenega razvoja. Poleg modelno vodenega pristopa OptimalJ uporablja še *poslovna pravila* z namenom hitrega uvajanja sprememb poslovne logike in *vzorce* (angl. patterns) za preslikave modelov iz višjih na nižje nivoje.

Osnovna predstavitev

OptimalJ definira tri nivoje in z njimi modele, ki ustrezajo trem MDA nivojem, to so *domenski* nivo, *aplikacijski* nivo in nivo *programske kode*.

Domenski nivo je opisan v domenskem modelu v klasičnem UML-ju z možnostjo definicije raznih omejitev in ustreza modelu PIM standarda MDA. Domenski



Slika 3.6: *Domenski model orodja OptimalJ z razrednim in storitvenim diagramom.*

model sestavljajo (slika 3.6):

- *razredni* domenski model – opisuje osnovno strukturo obravnavane domene z razredi, njihovimi atributi in poslovnimi operacijami ter razmerja med njimi,
- *storitveni* (angl. service) domenski model – določa poslovna opravila oziroma osnovne funkcionalnosti, ki jih mora aplikacija podpirati,
- *procesni* domenski model – določa poslovni proces kot zaporedje osnovnih aktivnosti z UML diagrami aktivnosti.

Po definiciji domenskega modela OptimalJ le-tega avtomatično preslika v *aplikacijski* model, ki ustreza nivoju PSM standarda MDA. Tako dobljeni aplikacijski model je možno, in v večini primerov tudi treba, popravljati oziroma dopolnjevati. Ciljni model pri OptimalJ je Java, natančneje Java Enterprise Edition (v nadaljevanju JEE). Aplikacijski model je sestavljen iz treh podmodelov:

- model *poslovne logike* – ustreza strežniku v navezi strežnik-odjemalec, ki odjemalcu daje na voljo osnovne storitve poslovne logike oziroma elementarne funkcionalnosti, ki jih odjemalec uporablja posamično ali jih povezuje v smiselne procese,
- *predstavitveni* aplikacijski model – ustreza odjemalcu, odjemalec se v osnovi zgradi po domenskem modelu tako, da omogoča osnovne operacije nad osnovnimi razredi domenskega modela. To začetno osnovo je mogoče prilagajati na različne načine v smislu formatiranja in preverjanja posameznih podatkov, oblikovanja izgleda zaslonov, povezovanja zaslonov v zaporedja in podobno,
- model *podatkovne baze* – obsega objekte podatkovne baze, določene na podlagi domenskega razrednega modela.

Iz aplikacijskega modela OptimalJ generira *programsko kodo* ter razne druge nastavitvene datoteke in skripte za generiranje podatkovne baze. Tako generirano programsko kodo je mogoče še ročno dopolnjevati na določenih vnaprej predpisanih mestih. Tako je mogoče določiti vsebino raznih metod, ogrodja generirane programske kode pa ni mogoče popravljati.

Podani opis je res osnovni oris delovanja OptimalJ. Celotna rešitev je zares obsežna in razdelana in tako na različnih nivojih ponuja različne možnosti. Gre za izjemno obsežen in zelo dober projekt, ki se je kalil skozi več let in različic. Za občutek podrobnosti bo podanih še nekaj standardnih lastnosti in rešitev:

- možna izbira dvo-nivojske ali tri-nivojske arhitekture,
- podpora različnim končnim tehnologijam; možno je izbrati tehnologijo odjemalca, narečje (angl. dialect) podatkovne baze in podobno,
- avtomatsko postopno branje in prikazovanje velikih seznamov podatkov,
- možnost oblikovanja ponovno uporabnih vzorcev modelov,
- uvoz in izvoz UML modelov,
- možnost uvoza modela podatkovne baze in generiranja ostalih nivojev v obratni smeri,
- povezovanje elementov modelov OptimalJ z zahtevami v posebnem programu OptimalTrace ter sledenje spremembam,
- poročila o neskladjih, ki jih povzročijo spremembe in brisanje elementov modelov,
- avtomatična namestitev končnih rešitev na najbolj uporabljane JEE strežnike,
- modelno podprto povezovanje (angl. integration) z drugimi sistemi preko spletnih storitev (angl. web service) in podobnih standardnih tehnologij.

Arhiterkturna različica

Posebna arhitekturna različica omogoča spreminjanje standardnega razvojnega okolja, predvsem v smislu prilagajanja obstoječih ali izdelave novih modelov in preslikav med njimi. Tako je možno oblikovati prilagojeno tovarno programske opreme, ki omogoča udobnejši razvoj programskih rešitev za podprto domeno.

Prilagoditve oziroma izdelava novih modelov in preslikav je mogoča zaradi zelo modularne zasnove orodja. To predvsem velja za preslikave, ki so organizirane v skupine vzorcev glede na uporabo. Preslikave je najbolj splošno mogoče podati s pisanjem preslikave v programskem jeziku, priporočen način pa je uporaba posebnega jezika za predloge, v katerem se je možno sklicevati na elemente meta modelov.

Arhitekturna različica odpira čisto nove dimenzije uporabnosti, omogoča oblikovanje domensko specifičnih jezikov in kombiniranje obstoječih in lastnih zmožnosti v tovarne programske opreme. Seveda razvoj lastne visoko specializirane tovarne programske opreme zahteva veliko dela, visoka specializiranost pa hkrati s poenostavitvijo modeliranja omejuje izrazne možnosti.

Ovrednotenje

Že predstavljeno ogrodje AndroMDA je tudi imelo določene standardne zmožnosti modeliranja osnovnih razredov domene, logike za dostop do podatkovne baze, storitev (angl. service) kot osnovnih funkcionalnosti modelirane poslovne logike, ter osnovnega grafičnega vmesnika. Pri do sedaj predstavljenih osnovah okolja OptimalJ gre za v osnovi precej podobne zmožnosti, le da so nekateri vidiki precej bolj podrobno podprti. Z drugimi besedami to pomeni, da imajo modeli v OptimalJ večje izrazne zmožnosti. Druga velika prednost pa je integrirano razvojno ogrodje, v katerem delo poteka bolj enostavno in hitreje.

Dodana vrednost okolja OptimalJ tako izhaja iz obsežnosti in podrobnosti njegovih zmožnosti. Ob spoznavanju podrobnosti delovanja je možno hitro uvideti, da v začetku omenjeni 'magični' gumb, ki iz modelov naredi izvedljiv program, ni nič magičnega, ampak skupek manjših enostavnih enot, od katerih vsaka prispeva delček v končno celovito smiselno rešitev v obliki programske kode.

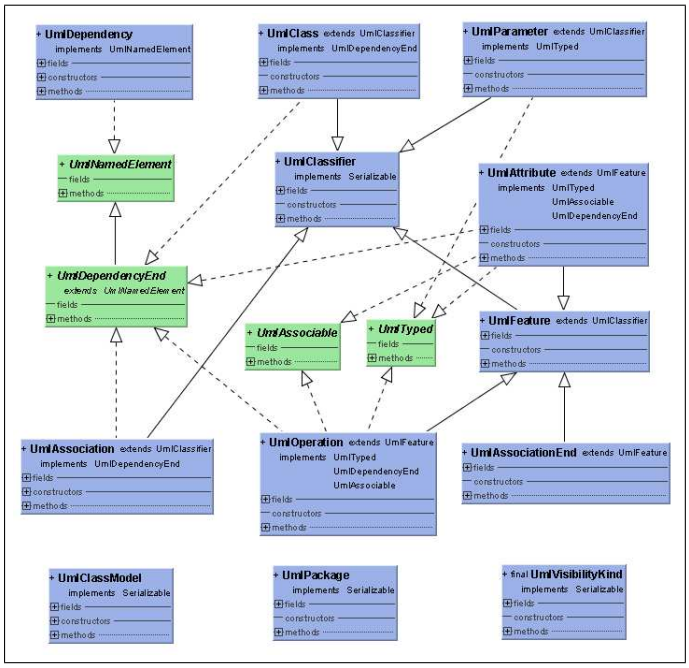
Arhitekturna različica z možnostjo razvoja prilagojenega orodja oziroma prave tovarne programske opreme odpira čisto nove možnosti. Iz osnovne različice naredi univerzalno prilagodljivo orodje za modelno vodeni razvoj. Te prilagoditvene zmožnosti so relativno univerzalne, sam proizvajalec pa ne priporoča prevelikih prilagoditev, predvsem zaradi problemov pri prehodu na novejše različice orodja, pri čemer bo morda lastne prilagoditve treba popravljati.

O idealnosti okolja OptimalJ je težko izreči splošno mnenje. Končna sodba je odvisna od gledišča ocenjevanja. Vnaprej predvidene in z modeli podprte stvari naredimo hitreje kot s klasičnim programiranjem. Prilagajanje standardnih rešitev zahteva podrobno poznavanje parametrov prilagoditev in kar nekaj dela. Včasih pa vnaprej predvidene zmožnosti prilagajanja nimajo možnosti želene prilagoditve in je treba take prilagoditve izvesti z omenjenim pisanjem programske kode. Pisanje programske kode pa je omejeno le na določene dele programske kode in tako z njim ni možno narediti česarkoli. Splošno rečeno, ta želeni višji abstraktni nivo skriva določene podrobnosti, na katere razvijalec nima več vpliva.

Tako je okolje OptimalJ v smislu dela znotraj njegovih sicer kar obširnih zmožnosti smiselno oceniti kot zelo dobro, preseganje teh osnovnih zmožnosti pa lahko povzroča velike težave.

3.4 Prototipi podpornih orodij

V tem razdelku bo predstavljenih nekaj prototipov, ki so bili razviti v okviru spoznavanja modelno vodenega pristopa, natančneje v smislu ugotavljanja možnosti



Slika 3.7: *Najpomembnejši podprti koncepti vhodnega UML meta modela izdelanega osnovnega prototipa orodja za podporo modelno vodenemu razvoju.*

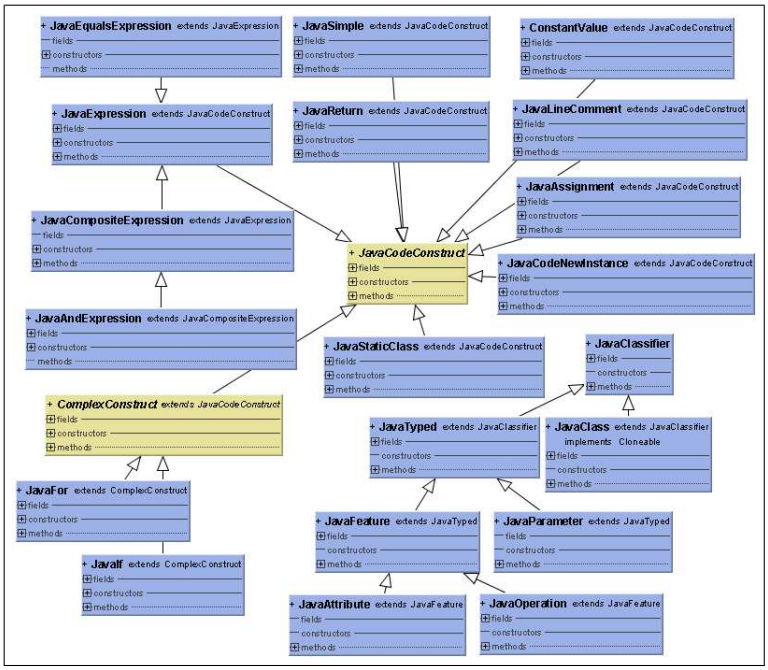
in zahtevnosti izdelave orodij za podporo modelno vodenemu pristopu oziroma razvoju programske opreme po idejah modelno vodenega pristopa v širšem smislu.

3.4.1 Osnovni elementi modelno vodenega orodja

Najprej smo se ukvarjali z izdelavo osnovnega orodja s standardnimi elementi branja in razčlenjevanja vhodnega modela, preslikavo in zapisovanjem ciljnega modela. Orodje je razvito v programskem jeziku Java. Za vhodni meta model služi poenostavljena različica UML jezika, predvsem razredni diagrami. Vhodni model je tako podan z množico primerkov javanskega razreda *UmlRazred*. Za izhodni model služi programski jezik Java. Izhodni model je tako podan z množico primerkov javanskega razreda *JavaRazred*. Podprti koncepti UML vhodnega in Java izhodnega meta modela so prikazani na slikah 3.7 in 3.8.

Preslikave delujejo na principu izdelave in dopolnjevanja ciljnega modela glede na elemente vhodnega modela. Nad to osnovo je bila razvita osnovna preslikava UML razrednih diagramov v razrede ciljnega jezika (glej sliko 3.9 za osnovno preslikavo razreda).

Ta osnova vsebuje možnost dodajanja dodatnih funkcionalnosti v smislu *vzorca dopolnjevanja* (angl. decorator pattern) ciljnega elementa glede na določene dodatne informacije v vhodnem elementu. Izdelana je bila podpora dopolnjevanja na nivoju razreda, kar posredno seveda dovoljuje tudi spreminjanje vseh drugih lastnosti razreda. Posamezna dopolnjevanja se poda s pogojem in z logiko za



Slika 3.8: *Najpomembnejši podprti koncepti Java izhodnega meta modela izdelanega osnovnega prototipa orodja za podporo modelno vodenemu razvoju.*

dopolnjevanje.

Pogoj se poda z implementacijo programskega vmesnika *ClassTransformationCondition* z eno samo metodo *runTransformationPartOn()*, ki za vhod sprejme primerek razreda vhodnega modela in vrne logični *da* (angl. true), če podani razred ustreza pogoju. Izdelanih je bilo tudi nekaj splošno uporabnih pogojev, kot je na primer *StereotypeTransformationCondition*, ki mu je treba podati stereotip in vrne pozitiven odgovor za vse UML razrede, ki so označeni s podanim stereotipom. Logiko dopolnjevanja pa podamo z implementacijo programskega vmesnika *ClassTransformationDecorator* z metodo *decorate()*, ki ima za parametre primerek UML razreda, dopolnjevani primerek Java razreda ter dostop do konteksta preslikave in celotnega ciljnega modela.

Primer enostavnega dopolnjevanja je opremljanje ciljnih Java razredov z zaznamkom *javax.persistence.Entity*, če je ta Java razred rezultat preslikovanja UML razreda označenega s stereotipom ≪Obstojen≫ (angl. persistent). Za pogoj tega dopolnjevanja uporabimo zgoraj opisani pogojni razred *StereotypeTransformationCondition*, dopolnjevanje pa pomeni dodajanje zaznamka Java razredu z *'javaClass.addAnnotation("javax.persistence.Entity")'*.

Izdelana je bila tudi podpora branja UML razrednih modelov iz XMI datotek. Ker orodja različno shranjujejo modele v XMI datoteke, je bila izdelana in preizkušena podpora za orodje MagicDraw [41], eno izmed boljših delno brezplačnih orodij za risanje UML diagramov.

```
public class CoreClassTransformation implements ClassTransformation  {
    private VisibilityKindTransformation visibilityKindTransformation;
    private AttributeTransformation attributeTransformation;
    private OperationTransformation operationTransformation;

    // izpuščene tri metode za nastavljanje vrednosti zgornjim trem komponentam

    public JavaClass transform(UmlClass umlClass,
            JavaClassModel targetJavaClassModel,
            TransformationContext transformationContext) {
        JavaClass javaClass = transformationContext.classRequired(umlClass);

        javaClass.setPackageName(umlClass.getPackage().getFullName());
        javaClass.setName(umlClass.getName());
        javaClass.setVisibility(visibilityKindTransformation.transform(umlClass
                .getVisibility()));

        if (umlClass.getGeneralizationOf() != null) {
            JavaClass extendsClass = transformationContext
                .classRequired(umlClass.getGeneralizationOf());
            javaClass.setExtend(extendsClass);
        }

        for (UmlFeature umlFeature : umlClass.getFeatures())
            if (umlFeature instanceof UmlAttribute)
                javaClass.addFeature(
                        attributeTransformation.transform(
                                (UmlAttribute) umlFeature, javaClass,
                                targetJavaClassModel, transformationContext));
            else if (umlFeature instanceof UmlOperation)
                javaClass.addFeature(
                        operationTransformation.transform(
                                (UmlOperation) umlFeature,
                                targetJavaClassModel, transformationContext));

        targetJavaClassModel.getClasses().add(javaClass);
        transformationContext.classParsed(umlClass, javaClass);
        return javaClass;
    }
}
```

Slika 3.9: *Slika prikazuje programsko kodo, ki preslika primerek UML razreda v primerek Java razreda. Preslikava naredi nov Java razred, mu nastavi ime paketa, ime razreda, vidljivost, morebitnega prednika ter delegira preslikovanje atributov in operacij ustreznima komponentama. Pomembna je tudi uporaba konteksta preslikave, v katerega se registrirajo nastali elementi izhodnega modela in povezave teh elementov z elementi vhodnega modela.*

Opisana preprosta rešitev tako omogoča generiranje programske kode iz UML modelov, narisanih v omenjenem orodju. Primer bolj smiselne uporabe tako razvitega ogrodja bo podan v nadaljevanju.

3.4.2 Poizvedbe

Pri programskih rešitvah se pogosto pojavi problem iskanja podatkov. Včasih je treba poiskati zapise iz določene tabele podatkovne baze, ki imajo določene zahtevane vrednosti atributov teh oziroma povezanih zapisov, podobno pa je treba včasih najti objekte iz množice objektov v pomnilniku. Iskanje podatkov iz podatkovne baze se doseže z izvajanjem poizvedovalnih stavkov, na primer v jeziku SQL, iskanje podatkov v zbirkah objektov pa s sprehajanjem po zbirki objektov in ugotavljanjem, ali posamezni element ustreza iskalnim pogojem.

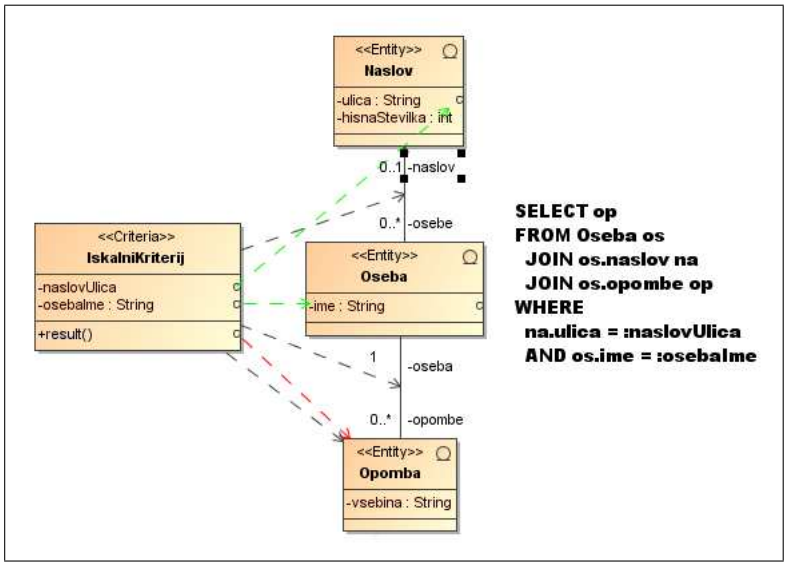
Ta dva načina iskanja sta konceptualno enaka; v obeh primerih gre za iskanje podatkov z istimi parametri, le tehnično so v drugačni obliki. Tako smo prišli na idejo splošne deklarativne rešitve obravnavanega problema iskanja podatkov. Ta rešitev omogoča deklarativno definicijo iskanja z definiranjem UML razreda, označenega s stereotipom ≪*Criteria*≫ (kriterij), ki ima za atribute poljubno poimenovane iskalne parametre, z asociacijami povezane na atribute razredov, po katerih naj iskanje poteka, poleg tega pa še standardno metodo *results()* (rezultati), ki mora imeti asociacijo na razred, katerega primerki so rezultat iskanja. V izdelani različici je treba narisati še asociacije do uporabljenih povezav med preiskovanimi razredi ter do razredov, ki niso implicitno vključeni v poizvedbo preko asociacij do iskalnih atributov.

Za opisano teoretično idejo sta bili izdelani dve preslikavi, ena ima za rezultat programsko kodo za iskanje objektov v pomnilniku, druga pa poizvedovalni stavek Hibernate. Ta stavek je v poizvedovalnem jeziku znane knjižnice Hibernate, ki je zelo podoben jeziku SQL, le da je na malo višjem abstraktnem nivoju in tako bolj primeren za ciljni model. Za boljše razumevanje je na sliki 3.10 podan primer z razlago.

Rešitev je tehnično izdelana nad predstavljeno osnovno rešitvijo iz prejšnjega razdelka. Generiranje omenjenih dveh oblik poizvedovanja je doseženo z uporabo opisanih možnosti dodajanja logike za dopolnjevanje ciljnih elementov na podlagi dodatnih informacij iz modela. Pogoj za dopolnjevanje je obstoj stereotipa ≪*Criteria*≫ na vhodnem UML razredu, dopolnjevanje pa je narejeno posebej za iskanje objektov v pomnilniku in za gradnjo poizvedbe. Zaradi uporabljenega pogoja dopolnjevanja omenjeni dopolnjevanji tako dobita za vhod ≪*Criteria*≫ UML razred. Dopolnjevanji pregledujeta asociacije vhodnega UML razreda in ustrezno gradita metode za iskanje oziroma poizvedovalni stavek.

3.4.3 Grafično orodje

Pomemben vidik modelno vodenega razvoja so tudi orodja za modeliranje, zato je bilo smiselno preizkusiti zahtevnost izdelave grafičnega urejevalnika modelov.



Slika 3.10: *Primer poizvedovalnega kriterija za iskanje Opomb Oseb po podanem imenu osebe in ulici. Na desni je generirani poizvedovalni stavek v Hibernate poizvedovalnem jeziku.*

Izdelan je bil prototip urejevalnika UML razrednih diagramov.

Pisanje urejevalnika od začetka je kar zahtevna naloga, zato je smiselno uporabiti katero izmed delnih rešitev. Uporabljeno je bilo ogrodje GEF (Graphical Editing Framework) [42], ki je namenjeno izdelavi grafičnih urejevalnikov. Ogrodje ponuja podporo za delo s sestavljenimi grafičnimi elementi in povezavami med njimi. Kljub tej podpori pa je izdelava urejevalnika razrednih diagramov še vedno precej zahtevna naloga. Ogrodje je sicer zelo široko zastavljeno, ima pa precej strmo učno krivuljo.

Izdelan je bil prototip, ki omogoča urejanje razredov in njihovih atributov ter osnovnih povezav med razredi, izdelana je bila tudi možnost shranjevanja in branja narisanih modelov. Nauk tega poskusa je ugotovitev, da je izdelava popolne podpore za urejanje UML (razrednih) diagramov veliko zahtevnejša naloga kot izgleda na prvi pogled. Za razne ad-hoc rešitve je tako morda bolje uporabiti enega izmed obstoječih urejevalnikov modelov. Za UML obstaja veliko dobrih urejevalnikov, na primer že omenjeni MagicDraw, širše uporabna možnost pa so splošnejša orodja, kot je že predstavljeno orodje GME.

3.4.4 Domensko-specifični jezik za vhodno-izhodne probleme

Do izbire modelno vodenega pristopa za temo naloge je privedlo večletno ukvarjanje z raznimi vhodno-izhodnimi rešitvami. Sem sodijo razne migracijske in integracijske rešitve, ki iz zunanjega sveta prejemajo podatke v sistem ali jih iz sistema pošiljajo drugim sistemom. Večinoma je tu vključena še razna dodatna

logika razčlenjevanja in preoblikovanja podatkov.

Obvladovanje mnogo takšnih rešitev, napisanih v programskih jezikih, postane s časom zelo zahtevna naloga. Analiza teh rešitev pokaže veliko skupnih lastnosti; izvor in ponor podatkov sta večinoma omejena na datoteke, podatkovne baze in vrste (angl. queue); formati nerazčlenjenih podatkov so ali podatki fiksne širine ali podatki ločeni z ločevalnim znakom in podobno.

Iz tega se je razvila ideja o jeziku za reševanje vhodno-izhodnih problemov. Ta jezik je v prvih različicah omogočal deklarativno definicijo vhoda, izhoda in preoblikovanja podatkov. S časom pa se je jezik razširil z imperativnimi zmožnostmi in raznimi drugimi deklarativnimi poenostavitvami standardnih zahtev, kot je na primer večnitnost, zagotavljanje neprekinjenega delovanja s ponovno vzpostavitvijo dostopa do virov ob napakah in podobno.

Za konkreten zapis jezika je bil izbran XML. Napisan je bil tudi interpreter za interpretacijo programskih rešitev v izdelanem jeziku. Rešitev je bila kar uspešna. Interpretacija jezika je precej poenostavila razvoj in vzdrževanje, saj je celotna programska rešitev podana v obliki ene ali nekaj XML datotek. Za razne ad-hoc rešitve in spremembe je možno enostavno popraviti ukazno XML datoteko in jo brez prevajanja in sestavljanja programske rešitve takoj uporabiti. Te ukazne datoteke so s časom začeli spreminjati in dopolnjevati tudi ljudje, ki ne znajo programirati, znali pa so popraviti na primer stavek SQL ali kakšno drugo intuitivno malenkost v ukazni XML datoteki. Za primer glej sliko 3.11.

3.4.5 Domensko-specifični jezik za opisovanje zvoka

Predstavljeni projekt je nastal iz fascinacije nad orodji za elektronsko izdelavo in obdelavo zvoka oziroma glasbe. Zvok je nihanje zraka, ki ga je v digitalni obliki možno opisati z zaporedjem števil kot vzorcev idealne funkcije nihanja. Sintetizatorji in drugi elektronski oziroma računalniški pripomočki omogočajo izdelavo osnovnih zvokov, s sestavljanjem osnovnih (preoblikovanih) zvokov pa tudi kompleksnejše zvoke, kar se v določenih primerih imenuje glasba. Osnovne zvoke je možno posneti ali pa jih generirati kot sinusne in podobne signale, iz katerih je s spreminjanjem frekvence in amplitude mogoče dobiti zanimive zvoke.

Za predstavljeni problem je bil razvit domensko-specifični jezik v obliki funkcij. Osnovne funkcije služijo kot generatorji osnovnih zvokov, na primer funkcija *vzorec(≪ime datoteke z vzorcem≫)* ali *sin(≪frekvenca v Hz≫, ≪trajanje v milisekundah≫)*. Nad temi osnovnimi funkcijami je možno uporabiti funkcije za preoblikovanje teh osnovnih vzorcev, na primer *faktor(≪vzorec≫, ≪faktor≫)*, kjer je na primer *faktor(sin(400, 10000), 0.5)* pol tišji 10 sekundni 400Hz sinusni zvok od *sin(400, 10000)*. Funkcij za preoblikovanje je veliko. Med najzanimivejše sodi ovojnica, s katero se vzorcu časovno spreminja amplitudo, klasično iz čiste tišine do maksimuma, zadrževanje na maksimumu ter počasno zmanjševanje. Zanimiv je tudi odmev, ki ga je možno definirati s številom zakasnjenih vzorcev, zakasnitvijo ponovitev ter zmanjšanjem amplitude zakasnjenih ponovitev. Zelo osnovna je tudi možnost seštevanja za izdelavo kompleksnih zvokov, skupaj s funkcijo zakasnitve

```
<Transformer_V2>
    <environment>
        <resources>
            <resource pbPovezavaVir="DBResource">
                <db_driver>oracle.jdbc.driver.OracleDriver</db_driver>
                <db_url>*****</db_url>
                <db_username>*****</db_username>
                <db_password>*****</db_password>
            </resource>
        </resources>
    </environment>
    <class name="Primer">
        <method method_name="main">
            <Daemon sleepOnIdle="L5000" sleepOnException="L100000"
                    con="pbPovezava" conResource="pbPovezavaVir">
                <var sporociloId="Integer"/>
                <var sporociloVsebina="String"/>
                <SQL connection="pbPovezava">
                    SELECT
                      [vs.Id, sporociloId, DBInt],
                      [vs.Vsebina, sporociloVsebina, DBString]
                    FROM Vhodno_Sporocilo vs
                    WHERE vs.Obdelano = 0 AND ROWNUM = 1
                    ORDER BY vs.Id
                </SQL>
                <If COND="sporociloId != NULL">
                    <then>
                        <Log info="'Vhodno sporočilo:' + sporociloVsebina"/>
                        <SQL connection="pbPovezava">
                            UPDATE Vhodno_Sporocilo
                            SET Obdelano = 1
                            WHERE Id = [sporociloId, DBInt]
                        </SQL>
                    </then>
                </If>
            </Daemon>
        </method>
    </class>
</Transformer_V2>
```

Slika 3.11: *Primer zapisa za vhodno-izhodni domensko-specifični jezik. Interpretacija prikazanega zapisa z izdelanim orodjem izvede glavno zanko (Daemon del) vsakih 5 sekund. V tej zanki je zagotovljena povezava na podatkovno bazo, ki je podana kot vir (angl. resource) v začetku primera. Zagotovljeno je tudi neprekinjeno delovanje, saj se vse izjeme (angl. exceptions) izpišejo in ignorirajo. Vsebina zanke ob vsakem izvajanju iz podatkovne baze izbere prvo izmed morebitnih neobdelanih sporočil, ga izpiše in označi za obdelanega.*

pa tudi za sestavljanje zvokov v zaporedje. Tako je dva zaporedna tona klavirja možno opisati kot *vsota(vzorec(klavir), zakasnitev(vzorec(klavir), 1000))*.

Jeziku so bile dodane še tri zmožnosti. Prva je oblikovanje sinonimov za funkcije, ki v jeziku delujejo kot nekakšne lokalne spremenljivke za vzorce, ki jih je možno v nadaljevanju uporabljati kot parametre drugih funkcij. Druga zmožnost so značilne funkcije, ki jih je možno uporabiti za parametre funkcije, frekvenca sinusne funkcije ni nujno konstanta, ampak se lahko linearno, sinusno ali kako drugače spreminja, na primer *sin(400 + sin(10, 10000), 10000)*. Tretja zmožnost pa je oblikovanje dveh kanalov ter določanje dodatnih funkcij za preoblikovanje posameznega kanala, kar omogoča stereo zvok.

Opisani jezik je bil tudi razvit, zanj je bilo izdelano tudi orodje, ki je iz podanih funkcij izračunalo končni vzorec, ga prikazalo na zaslonu ter predvajalo.

3.4.6 Prenos podatkov med sistemi

Za konec bo predstavljen še projekt, katerega izdelava je bila precej pod vplivom raziskovanja modelno vodenega pristopa, in predstavlja večkrat omenjeno ad-hoc uporabo idej tega pristopa.

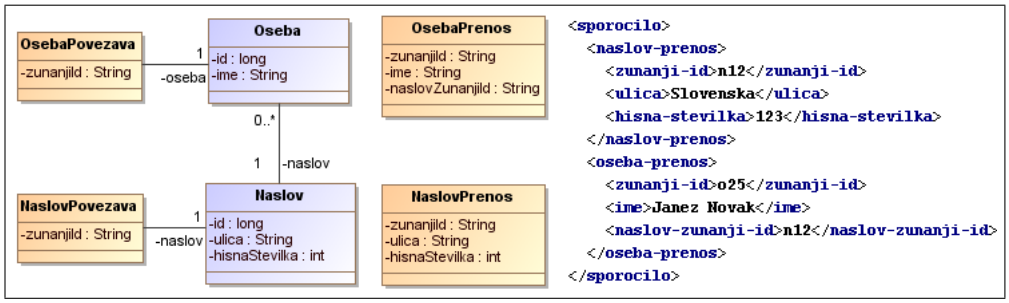
Izdelati je bilo treba prenos oziroma integracijo podatkov med večimi izvornimi sistemi in ponornim sistemom[8]. Zaradi narave problema in zahtev je bila sprejeta odločitev, da je potrebna vmesna oblika podatkov v XML tehnologiji, ki predstavlja format, ki je skoraj enak formatu podatkov ponornega sistema. Edina neenakost so glavni ključi zapisov ter reference na druge zapise, ki so v zunanjem sistemu drugačni in je tako zanje potrebna preslikava. Tako ponorni XML format določa obliko sporočil, ki jih morajo pošiljati izvorni sistemi ob spremembah ustreznih podatkov.

Primer takšnih sporočil bi bili sporočili za zapis o stranki in njenem naslovu. Izvorni sistem bi najprej poslal sporočilo o naslovu z določeno zunanjo identifikacijo tega naslova. Obdelava tega sporočila bi imela za posledico shranjevanje tega naslova in povezave med tem naslovom in njegovo zunanjo identifikacijo v ponorno podatkovno bazo. Ob prispetju sporočila o stranki, ki vsebuje tudi zunanjo identifikacijo njenega naslova, mora obdelava sporočil preko prej zabeležene povezave med naslovi in njihovimi zunanjimi identifikacijami najti naslov v ponorni bazi ter ga nastaviti ravno shranjevanemu zapisu o stranki.

Ponorna stran

Tako je bilo na ponorni strani za nekaj deset že obstoječih entitetnih razredov treba izdelati ustrezne razrede za prenos podatkov, iz njih XML shemo (angl. XML Schema) ter še entitetne razrede za hranjenje povezav med zunanjo identifikacijo in shranjenimi zapisi v ponorni bazi. Problem bi bilo mogoče rešiti z ročnim pisanjem vseh teh razredov, vendar narava problema kar kliče po uporabi modelno vodenih idej.

[8] Dejansko med sistemi za splošno podporo poslovanju ter obračunskim sistemom.



Slika 3.12: *Slika prikazuje obstoječa razreda* Oseba *in* Naslov, *iz njiju generirana razreda* OsebaPrenos *in* NaslovPrenos *za prenos podatkov, generirana razreda* OsebaPovezava *in* NaslovPovezava *za hranjenje povezave med podatkovnim razredom in njegovim zunanjim identifikatorjem ter primer XML sporočila za prenos podatkov za osebo in njen naslov.*

Vhodni model predstavljajo že obstoječi entitetni razredi. Izdelana je bila preslikava, ki na podlagi teh razredov izdela razrede za prenos podatkov, ki imajo vse preproste atribute enake atributom entitetnih razredov, glavni identifikator *id* nadomešča atribut *zunanjiId* tipa niz, atribute N-proti-1[9] kot atribute tipa niz z imenom, sestavljenim iz imena osnovnega atributa in pripone *ZunanjiId*[10], atributi 1-proti-N[11] pa se ignorirajo. Tako bi iz zgoraj omenjenega entitetnega razreda za stranko z glavnim identifikatorjem *id*, enostavnim atributom *ime* tipa niz in N-proti-1 atributom *naslov*, preslikava izdelala razred za prenos podatkov za stranko z atributi *zunanjiId*, *ime* in *naslovZunanjiId* tipa niz. Iz teh razredov se da s standardnimi orodji avtomatično izdelati XML shemo, ki predstavlja specifikacijo sporočil za izvorne sisteme (glej sliko 3.12).

Izdelana je bila tudi preslikava, ki iz entitetnih razredov generira dodatne entitetne razrede za hranjenje povezav med zunanjimi identifikatorji in ustreznimi osnovnimi entitetami. Za zgoraj opisani entitetni razred *Naslov* se v preslikavi izdela razred *NaslovPovezava* z atributom *zunanjiId* tipa niz in atributom *naslov* tipa *Naslov*.

Naslednji problem je obdelava tako definiranih prihajajočih sporočil, ki se jo da tudi avtomatizirati. Možno bi bilo generirati razrede za obdelavo teh sporočil, izbran pa je bil generičen način obdelave, ki uporablja *refleksijo*[12] (angl. reflection) uporabljenega programskega jezika Java. Tako se prejeto XML sporočilo s standardnimi funkcionalnostmi najprej pretvori v prej generirani prenosni razred. Nato se iz njega ugotovi entitetni razred, izlušči identifikator zapisa in preko podatkov o povezavah ugotovi, ali gre za novi ali obstoječi zapis. Če gre za obstoječi

[9] Desna *N* stran predstavlja razred, ki ima *1* primerek atributa, tuj ključ v terminologiji podatkovnih baz, na primer razred Naslov ima *N-proti-1* atribut *pošta* tipa *Pošta*.

[10] *Id* je krajša oblika za *Identifikator*.

[11] Obratno od *N-proti-1*; *1* razred ima atribut, ki je zbirka *N* primerkov drugega.

[12] Funkcionalnost odkrivanja meta podatkov o razredih, atributih, metodah in drugih elementih programskega jezika.

zapis, se v nadaljevanju uporabi ta zapis, sicer se izdela nov primerek entitenega razreda in shrani ustrezno povezavo. Nato se splošna logika sprehodi po atributih entitetnega razreda in vzporedno razreda za prenos podatkov ter pri enostavnih atributih le prenese njihovo vrednost, pri N-proti-1 atributih pa za pridobitev vrednosti naredi poizvedbo v ustrezno povezovalno tabelo z vrednostjo zunanjega identifikatorja atributa.

Tako je celotna rešitev na ponorni strani izdelana na enem meta nivoju višje od klasičnega programiranja, v obliki preslikovalnih funkcij in splošne logike za obdelavo sporočil. Ob spremembi entitetnega modela ponornega sistema je treba le ponovno izvesti preslikovalni funkciji in avtomatično izdelati novo XML shemo sporočil za izvore, kar je čisto mehanično opravilo.

Izvorna stran

Izvorna stran je tudi rešena na podoben splošen način. Dodaten problem povzroča le neskladnost oblike podatkov izvorne strani in XML formata ponorne strani. Namesto pisanja prevedbe za vsako entiteto posebej je bila izdelana splošna rešitev, ki ima omenjene razlike definirane deklarativno v posebnem formatu. V tem formatu je možno definirati, kateri izvorni entitetni tip se preslika v kateri tip sporočila, po potrebi pa še imena pomensko in vsebinsko enakih, ampak različno poimenovanih atributov izvorne entitete in ponornega sporočila. Za nedefinirane atribute se predpostavlja, da sta imeni atributov na obeh straneh enaki, kar pomeni uporabo principa prednastavljenih vrednosti. V tej definiciji razlik je možno definirati tudi, kako priti do atributa, ki ne obstaja v osnovni izvorni entiteti, ampak v nanjo povezani entiteti.

3.4.7 Spoznanja ob izdelavi prototipov

Ob praktičnem ukvarjanju z modelno vodenim razvojem je možno hitro ugotoviti, da so orodja za celovito podporo pristopu mnogo preveč zahtevna, da bi jih razvijali sami. Velikokrat pa si je možno zelo olajšati delo z raznimi delnimi orodji ali celo ad-hoc uporabo idej modelno vodenega razvoja v okviru klasičnega razvoja.

Iz predstavljenih rešitev je videti bistveno praktično prednost modelno vodenih idej. Te ideje omogočajo specifikacijo le nujno potrebnih dejstev problemske domene na relativno idealnem abstraktnem nivoju, ki postanejo računalniku razumljiva s preslikavami ali splošno obdelavo na enem meta nivoju višje.

3.5 Povzetek ugotovitev

Za konec pa še povzetek ugotovitev o modelno vodenem razvoju s stališča zgradbe, izdelave in uporabe orodij z navezavo na ugotovitve iz prejšnjih poglavij.

Navidezna magičnost modelnih orodij

S prikazom orodij, pristopov in primerov od lastnih rešitev do visokonivojskih prilagodljivih razvojnih okolij je bilo predstavljeno današnje stanje in možnosti modelno vodenega razvoja. Orodja in principi so bili predstavljeni po principu bele škatle; predstavljena je bila torej tudi notranja mehanika delovanja. Namen takšnega prikaza je, poleg razumevanja podrobnosti modelno vodenega pristopa, predvsem v postopnem približevanju spoznanju, da obsežnejše in visokonivojske rešitve za podporo modelno vodenemu pristopu niso nič magičnega.

Začetni stik z naprednimi orodji, še posebej pa reklamno gradivo za razna orodja za podporo modelno vodenemu razvoju, dajejo vtis, da se da zapletene in obsežne aplikacije narediti precej hitro le z risanjem modelov. Ustvarja se občutek, da se da iz le nekaj modelov z malo dodatnega kodiranja izdelati poljubne programske rešitve.

Realno gledano pa je, neodvisno od pristopa, treba določiti vse lastnosti in podrobnosti končne programske rešitve. Pri tem so sicer v veliko pomoč prednastavljene vrednosti, komponente ali drugi koncepti in pripomočki, vendar to še vedno ne izpodbija osnovnega dejstva.

Če mora biti na primer eno izmed oken v grafičnem vmesniku programske rešitve rdeče barve, morajo izrazne zmožnosti omogočati podajanje barve oken in je treba to zahtevo na predpisan način tudi podati. Tako tudi obstoječa orodja za modelno vodeni razvoj, ki znajo le iz razrednega UML diagrama zgraditi aplikacijo, ki preko grafičnega vmesnika omogoča osnovne operacije branja in zapisovanja za posamezne razrede diagrama, večinoma niso dovolj prilagodljiva rešitev, saj takšne zahteve večinoma obsegajo le majhen del zahtevane poslovne logike. Preostala poslovna logika pa je večinoma precej raznolika in je v nekaterih primerih lahko zelo zapletena.

Smiselna kombinacija večih delnih rešitev

Zanimiva je predvsem ugotovitev, da so na prvi pogled zapletene rešitve podpore standardni poslovni logiki v resnici sestavljene iz večih relativno preprostih in med seboj dopolnjujočih se rešitev.

Klasičen primer je modeliranje in generiranje strežnika, kjer ponavadi iz osnovnega razrednega diagrama preko posameznih enostavnih preslikav nastanejo entitetni razredi in razredi za prenos podatkov (angl. Data Transfer Objects) ter komponente za pretvorbo med tema dvema predstavitvama podatkov. Storitve oziroma osnovne funkcionalnosti strežnika pa običajno nastanejo iz razrednega ali podobnega diagrama, v katerem vsak razred ali podoben koncept predstavlja določeno komponento, operacije razreda pa storitve, ki imajo za vhode in izhode prej definirane razrede za prenos podatkov. Tako ena preslikovalna funkcija generira razrede za prenos podatkov, druga pa generira storitvene komponente, v katerih se sklicuje na razrede za prenos podatkov, ki so rezultat prve preslikovalne funkcije.

Netrivialna poslovna logika

Za reševanje bolj ali manj zapletenih delov sistema, za katere ni standardnih rešitev, je na voljo prilagajanje orodij z izdelavo lastnih modelnih jezikov in preslikav zanje. Ta vidik modelno vodenega razvoja sicer še ni ravno idealno podprt, pokazana pa je bila relativna enostavnost razvoja takšnih lastnih rešitev čisto od začetka.

Večji problem od podpore orodij pa je dejstvo, da je razvoj lastnih rešitev v obliki modelnih jezikov in preslikovalnih funkcij vsaj pri obsežnejših in zapletenejših primerih lahko zelo časovno in izvedbeno zahteven. Zato je smiselno že pred razvojem takšnih rešitev temeljito premisliti o smiselnosti takih podvigov, pri čemer je treba imeti v mislih možnosti ponovne uporabnosti in morebitnih prihodnjih razširitev.

Najboljše rezultate dajejo skupki manjših univerzalnih rešitev, ki imajo tako avtomatično več možnosti kombiniranja oziroma ponovne uporabnosti.

Primerjava s klasičnim razvojem

Mnogokrat se klasični razvoj s programskimi jeziki v primerjavi z modelno vodenim pristopom neupravičeno preveč zapostavlja oziroma označuje kot veliko bolj okornega.

Osnova ideje modelno vodenega razvoja je predvsem v odkrivanju vzorcev in oblikovanju standardnih rešitev za tako odkrite vzorce v obliki modelov za čim bolj kompaktno predstavitev dejstev na idealnem abstraktnem nivoju ter oblikovanju preslikovalne funkcije, ki ta dejstva naredijo razumljiva računalniku.

Vendar je ideja odkrivanja vzorcev v delovanju stvari človeku prirojena sposobnost poenostavljanja in se tako nanaša na vsa področja njegovega delovanja. Tako je tudi pri razvoju programskih rešitev v klasičnih programskih jezikih na voljo možnost oblikovanja komponent, ki se jih da nato na kompakten način večkrat uporabiti.

Je pa res, da je ta človeku prirojeni čut iskanja izboljšav pri modelno vodenem pristopu precej bolj v ospredju.

Dobro in slabše razvita področja modelne podpore

Osnova modelno vodenega razvoja v smislu modelov in preslikav je že precej dobro zastavljena. UML kot splošen modelni jezik kljub svojim pomanjkljivostim kar dobro opravlja svojo nalogo. Podobno je relativno enostavna tudi izdelava priporočenih enostavnih preslikav.

Na trgu je na voljo zares ogromno orodij, ki vsaj delno podpirajo modelno vodeni razvoj. Ta orodja se lotevajo določenih slabo standardno podprtih vidikov modelno vodenega razvoja na različne bolj ali manj uspešne načine. Tako se oblikuje ogromna množica idej, ki se stalno izboljšuje, in iz katerih se bodo s časom verjetno razvile dobre prakse in standardi. Ti standardi morda še najbolj

manjkajo na področju modelnih jezikov v smislu, da tudi v računalniški teoriji še ni domišljenih dobrih načinov za opisovanje določenih konceptov.

Tako so na primer razredni diagrami precej idealni za opisovanje osnovnih konceptov obravnavanih domen z njihovimi atributi in definicijami operacij[13], podobno bi bilo treba izumiti še diagrame za opis obnašanja oziroma algoritmov, grafičnih vmesnikov, povezovanja osnovnih funkcionalnosti v smiselna zaporeda in podobnih splošnih pa tudi bolj specifičnih konceptov.

Teorija za izdelavo preslikav je s standardom QVT precej dobro definirana, ni pa še preveč udejanjena v praksi.

Prihodnost

Vsaj nekatere ideje modelno vodenega in sorodnih pristopov so več kot dovolj dobre, da jih bo v taki ali drugačni obliki videti tudi v prihodnosti.

Tu predvsem izstopata ideja dviga nivoja abstrakcije oziroma podajanja dejstev na najprimernejšem abstraktnem nivoju ter ideja modela in preslikovalne funkcije. Čez desetletje ali dve bo od današnjih klasičnih objektnih programskih jezikov verjetno ostal le še spomin na preteklost, podobno kot je danes govora o zbirnem jeziku kot o stvari preteklosti oziroma peščice ljudi, ki piše prevajalnike.

Na modelno vodena orodja je mogoče gledati tudi kot na reinkarnacijo CASE orodij izpred desetletij v novejši in bolj fleksibilni obliki.

[13]Natančneje s specifikacijami operacij; z definicijo algoritma so že problemi.

Poglavje 4

Predlog: izvedljiva specifikacija

V tem poglavju predlagamo alternativen način podajanja navodil računalniku. Predlog ima korenine v ideji mentalnega preskoka od programiranja računalnikov, preko morda danes dosegljivega podajanja navodil računalniku v naprednih zapisih, k futurističnim oblikam razumevanja pojma podajanja navodil računalniku. Podani predlog se v veliki meri opira na v prejšnjih poglavjih predstavljene ideje, predvsem pa na modelno vodeni pristop.

4.1 Izhodišče

Za izhodišče nadaljnjih ugotovitev so v tem razdelku povzete zares bistvene ugotovitve iz prejšnjih poglavij. Podana je tudi kratka predstavitev ideje programiranja z naravnimi jeziki, po kateri se predlog delno zgleduje. Na koncu je nakazana še smer predloga, torej, katere problematične vidike razvoja programske opreme ta predlog rešuje.

Bistvene ugotovitve

Cilj razvoja in raziskav na področju razvoja programske opreme je v bistvu iskanje idealnega načina podajanja navodil računalniku. Trud za doseganje tega cilja je v klasičnem smislu mogoče razdeliti na dve osnovni smeri delovanja:

- iskanje idealnega načina specifikacije programske opreme in
- premagovanje abstraktnega prepada med tako oblikovanim idealnim nivojem in računalniku razumljivim nivojem.

Zgornji alineji veljata tudi za modelno vodeni pristop, ki v bistvu obljublja idealen način specifikacije programske opreme ter direktno uporabo takšnega opisa za avtomatično izdelavo programske opreme. Drugače povedano – išče se idealen zapis oziroma modelni jezik, za katerega je seveda treba izdelati prevedbo oziroma preslikavo.

Glavni problem predstavlja prva alineja – razvoj idealnega načina specifikacije programske opreme. Druga alineja, problem prevedbe tega idealnega zapisa na računalniku razumljiv nivo v smiselno programsko rešitev, je lahko izvedljiva ali pa ne, kjer je pogoj za izvedljivost popoln in nedvoumen zapis.

V tej bistveno problematični smeri izumljanja idealnega načina specifikacije modelno vodeni pristop v širšem smislu ponuja ideje in konceptualne prijeme za oblikovanje idealnega načina specifikacije ter njegove prevedbe, šibkejši pa je pri dejanskih rešitvah. Glavni prispevek modelno vodenega pristopa so v njegovi idealni različici razvijalcu vidni splošno-namenski in domensko-specifični modelni jeziki s poudarkom na grafičnem konkretnem zapisu ter, spet idealno, razvijalcu skriti del preslikovanja z vsemi standardi. S tem modelno vodeni razvoj postavlja osnovo za nadaljnji napredek.

Programiranje z naravnimi jeziki

Programiranje z naravnimi jeziki (angl. natural language programming), krajše tudi *naravno programiranje*, je pristop podajanja navodil računalniku v naravnih ali njim podobnih jezikih [32]. Področje naravnega programiranja je zelo široko in ima tako tudi širok spekter rešitev in njihovih zagovornikov. Eno skrajnost pri tem predstavljajo poskusi razumevanja in preslikovanja običajnih tekstov v naravnem jeziku v računalniku razumljive zapise, kjer računalnik ponavadi 'razume' le del besedila. Druga skrajnost pa je oblikovanje popolnoma formalnih jezikov, ki so zelo podobni naravnim jezikom. Ti formalni jeziki so tako praktično uporabni za specifikacije, do določene mere pa se jih uporablja tudi že kot izrazne zmožnosti za podajanje navodil računalniku, vendar je večina teh rešitev omejena na enostavne prototipe za ozko definirane domene [44].

Smer predloga

Bistven problem napredka na področju razvoja programske opreme je tako pomanjkanje teoretičnih osnov za idealen način specifikacije navodil računalniku. Že sama splošnost tega problema pove, da njegova rešitev ni enostavna.

Predlog tako v osnovi podaja določene rešitve teh problemov v obliki relativno idealne konceptualne ideje idealnega načina specifikacije navodil računalniku ter postopno konkretizacijo te ideje v končno obliko prototipa, ki zaradi omenjene širine in zapletenosti bistvenega problema seveda ni idealna rešitev.

Bistvena smer predloga je poskus oblikovanja idealnega načina specifikacije navodil računalniku, v smislu osredotočenja razmišljanja na idealno specifikacijo in manj na prevedbo, pri čemer je morda najbolj pomemben miselni prehod od izboljšav že obstoječih računalniku razumljivih zapisov k osredotočanju na popolnoma nov idealen način specifikacije navodil računalniku.

Predlog se v osnovi opira na ideje modelno vodenega pristopa in programiranja z naravnimi jeziki.

4.2 Konceptualna rešitev

Predlog bo najprej podan na konceptualnem nivoju v obliki razmišljanja, ki postopoma pripelje do osnovnih idej predloga.

4.2.1 Ideja enotnega zapisa

Poenostavljeno povedano se programske rešitve klasično izdeluje v treh osnovnih fazah. Najprej je treba opraviti analizo obravnavane problemske domene, nato sledi faza načrtovanja z izdelavo načrta oziroma specifikacije rešitve, po kateri se v fazi izvedbe izdela končno programsko rešitev. Idealna specifikacija programske rešitve mora natančno in nedvoumno podajati vse zmožnosti in omejitve končne programske rešitve. Drugače rečeno, v fazi izvedbe se ne sme pojaviti nobeno vprašanje, na katerega se ne da odgovoriti iz informacij specifikacije.

Taka idealna specifikacija predstavlja dovolj dober zapis, ki že po svoji definiciji zagotavlja izvedljivost avtomatične prevedbe te specifikacije v končno delujočo rešitev. Ob zadostitvi pogoju avtomatične prevedbe tako ostane odprto še vprašanje idealnosti tega zapisa v smislu kompaktnosti, visokega nivoja, organizacije in drugih znanih in še ne poznanih pozitivnih vidikov.

Modelno vodeni razvoj in drugi podobni pristopi se trudijo zvišati abstraktni nivo tega zapisa, vendar še vedno govorijo o teh zapisih (pri modelno vodenem razvoju o modelih) le kot o zamenjavi programske kode s temi zapisi v smislu izdelkov, ki jih izdelujejo razvijalci – tako imenovanih *izdelkov prvega (raz)reda* (angl. first class objects).

Prva pomembna in gotovo ne najbolj originalna ideja predlaganega pristopa je *ideja zamenjave specifikacijskega zapisa in zapisa izdelkov prvega reda z enim samim univerzalnim zapisom.* Namesto koraka izdelave specifikacije, ki mu sledi korak izvedbe z izdelavo izdelkov prvega reda, predlagamo oblikovanje zapisa, ki je primeren za specifikacijo in hkrati tudi primeren kot množica izdelkov prvega reda, iz katerih obstaja avtomatična prevedba v izvedljivo programsko rešitev.

Ideja je že danes izvedljiva v praksi, saj ne izpostavlja kompaktnosti, visokega nivoja in podobnih naprednih lastnosti zapisa. V najosnovnejši različici je to lahko zapis na abstraktnem nivoju sodobnih programskih jezikov s konkretnim zapisom, bolj primernim za specifikacije. Tej osnovni splošno namenski različici je mogoče dvigovati nivo abstrakcije in s tem tudi oblikovati za namene specifikacije bolj primeren konkretni zapis na več načinov, na primer z uporabo domensko-specifičnih konstruktov.

Ideja izhaja iz ugotovitve o nesmiselnosti dvojnega dela. Najprej je treba izdelati specifikacijo, ki mora biti dovolj popolna, da natančno definira izvedbo. Nato pride na vrsto izvedba, ki je ob zahtevani popolnosti specifikacije le prepis specifikacije v obliko, za katero že obstaja prevedba na osnovni računalniku razumljiv nivo strojnega jezika, pa naj si bodo to sodobni programski jeziki ali modeli. Bistvo ideje tako ni v abstraktnem nivoju, ampak v poenotenju zapisa in s tem preprečevanju dvojnega dela.

Omenjena, danes klasična, dvojnost dela zelo vpliva na kvaliteto specifikacij. Ker je cilj izvedljiva programska rešitev, je zares pomembna le izvedba, zato se mnogokrat ne izdela popolnoma podrobne specifikacije. Zato pri izvedbi lahko nastopijo težave pri razvoju v specifikaciji nedefiniranih podrobnosti. V najslabšem primeru lahko analiza takih problemov v nedefiniranih podrobnostih pokaže neprimernost ključnih konceptov specifikacije, kar pomeni preoblikovanje teh ključnih konceptov, to pa ima za posledico popravke vseh izdelkov prvega reda, ki so odvisni od teh konceptov.

4.2.2 Ideja izvedljive specifikacije

Ob ugotovitvi pomanjkanja teoretičnih rešitev tako ostaja bistveno vprašanje idealnega zapisa za idealno podajanje poljubnih dejstev najprej na konceptualnem nivoju. Najprej pa je seveda treba ugotoviti, kakšen naj bi ta idealni zapis sploh moral biti.

Preskok na idealno stran

Večina napredka na področju razvoja programske opreme je do sedaj delovala v smeri zviševanja abstraktnega nivoja zapisa, za katerega obstaja prevedba v računalniku neposredno razumljivo obliko. Konkretneje gre za prehod od strojnega jezika preko zbirnega jezika k sodobnim programskim jezikom in seveda tudi k modelno vodenim in drugim sorodnim pristopom.

Ideja enotnega zapisa pa zahteva mentalni preskok k razmišljanju o idealnem zapisu, torej ne gre več za izboljševanje obstoječih direktno prevedljivih zapisov, ampak za direktno oblikovanje povsem novega kompaktnega visokonivojskega dobro organiziranega zapisa. Gre za mentalni preskok od razmišljanja, kaj je mogoče storiti z obstoječimi rešitvami, k razmišljanju, kaj je idealna rešitev.

Pri iskanju idealne rešitve še vedno obstaja problem pomanjkanja teoretičnih rešitev za podajanje splošnih dejstev, vendar gre zaradi omenjenega preskoka za iskanje rešitev v širšem smislu, zunaj klasičnih okvirov[1].

Primernost različnih konkretnih zapisov

Pri oblikovanju jezika je v osnovi pomemben le abstraktni zapis (angl. abstract syntax) jezika, v samem procesu razvoja pa ljudje razmišljajo v naravnih jezikih. Miselni procesi dejansko potekajo v abstraktnem zapisu naravnih jezikov. Pri specifikaciji prijavnega okna si človek v mislih govori: "Ob zagonu aplikacije se mora prikazati prijavno okno z vnosnima poljema za uporabniško ime in geslo z ustreznima napisoma ter gumbom za potrditev vnosa". Pri tem so uporabljene besede in njihove zveze tudi računalniškim nestrokovnjakom nedvoumno razumljive. Posamezne besede tako pomenijo določene standardne koncepte. Te

[1] Angl. thinking out-of-the-box.

koncepte se v specifikaciji predstavi v konkretnem zapisu, v večini primerov v slikovni oziroma vizualno zaznavni obliki.

Za vizualno obliko podajanja specifikacij se je v začetku uporabljal naravni jezik v bolj ali manj formalni obliki. S časom so se za določene dele specifikacije pokazali primernejši razni grafični diagrami, na primer diagrami poteka ali UML diagrami. Modelno vodeni razvoj in drugi sodobni pristopi, gledano striktno v smislu visokega abstraktnega nivoja zapisa z obstoječo prevedbo, se nagibajo k uporabi diagramov oziroma modelov kot ključnega koncepta vizualnega podajanja specifikacij. Vendar imajo diagrami poleg dobrih lastnosti tudi slabe lastnosti, kot so znanje, potrebno za njihovo razumevanje, omejene izrazne možnosti in podobno.

Predvsem zaradi potrebnega znanja za razumevanje diagramov se tako pojavlja dvom v njihovo absolutno primernost za idealne zapise. Ljudje razmišljamo v abstraktnem zapisu naravnih jezikov, v mislih si govorimo ideje, ki jih je brez prevedbe možno izgovoriti oziroma zapisati v vizualno obliko, ki jo ostali ljudje spet razumejo brez posebnih znanj oziroma prevedb.

Obvladovanje večih jezikov

Zgornje razmišljanje o človeku direktno nerazumljivih oblikah zapisa je smiselno navezati na problem obvladovanja velike množice umetno ustvarjenih jezikov za različne domene. To vprašanje je splošna različica smiselnosti ideje visoko formaliziranih domensko-specifičnih jezikov, ki ob slabši primernosti visoko formaliziranih splošno-namenskih jezikov za določeno domeno ali vidik problema predlaga razvoj lastnih jezikov.

Smiselnost poznavanja velikega števila jezikov z njihovimi abstraktnimi in konkretnimi zapisi ter pomenom je vprašljiva iz večih vidikov. Eden glavnih problemov je že sama številčnost teh potencialnih novih specifičnih jezikov, ki se nakazuje s številčnostjo idej nadgradnje klasičnih programskih jezikov, od katerih so nekatere na voljo še v večih različicah.

V splošnem se tako da izdelati programske rešitve s čisto osnovnim znanjem osnovnih izraznih možnosti, z dodatnim vložkom napora v poznavanje naprednejših izraznih zmožnosti pa je določene dele programskih rešitev mogoče izraziti hitreje in bolj elegantno.

Problem je lahko tudi premajhna podrobnost izraznih možnosti določenih jezikov, ko uporabljeni jezik ne omogoča izvedbe določene podrobnosti. Primer bi bila izvedba zahteve za nastavitev barve grafične komponente pri uporabi visokonivojskega jezika za grafični vmesnik, ki uporablja izris, določen z operacijskim sistemom, in nastavitve barve preprosto ne omogoča.

Uporaba velikega števila jezikov je tako omejena s človeškimi zmožnostmi njihovega obvladovanja in s specifičnostjo teh jezikov. Najbolj smiselna generalna usmeritev je tako morda uporaba dobrega splošno-namenskega načina podajanja dejstev ter obstojem smiselnih, bolj specifičnih rešitev na višjem nivoju za pogosto nastopajoče vidike, ki jih je s splošno-namenskimi zmožnostmi mogoče podati na manj idealen način in katerih uporaba je sicer priporočena, ne pa obvezna. Ta

usmeritev seveda ne omejuje ali izključuje nobene dobre prakse, kot je na primer ponovna uporaba.

Analiza specifikacij

Pred oblikovanjem konceptualnih smernic za idealen zapis je smiselno analizirati zapise obstoječih dobrih in popolnih specifikacij. Dobre in popolne specifikacije, ki ne puščajo nobenega dvoma v podrobnosti končne programske rešitve, imajo določene skupne lastnosti. Kljub temu, da so večinoma napisane v naravnem jeziku, morajo biti za zadostitev popolnosti zelo nedvoumne, formalne in matematične. Za eksaktno izražanje res zapletenih dejstev je mnogokrat najbolj primerna uporaba matematičnih konstruktov, kot so množice, funkcije, kvantifikatorji (na primer *za vsak* ali *obstaja*), predikati in podobno.

Za specifikacijo nekaterih dejstev so že zelo primerni konstrukti obstoječih programskih jezikov oziroma njihovih standardnih razširitev. Dober primer takih izraznih možnosti so funkcionalnosti knjižnic za delo z zbirkami elementov (angl. collections), kjer se na primer iskanje elementov z določenimi lastnostmi ne poda več klasično *'z zanko se sprehodi čez vse elemente, za vsakega preveri, ali ustreza pogoju, in če ustreza, ga dodaj v rezultat'*, ampak se le definira predikat kot logično funkcijo danega pogoja, ki se ga uporabi v ukazu knjižnice, ki izbere vse elemente seznama, ki ustrezajo predikatu, kar je veliko bolj podobno definiciji v naravnem opisu: *'iz [oznaka zbirke elementov] izberi elemente, ki [predikat]'*.

Organizacija in nivoji opisov

Pri specifikacijah za zapletene domene je zelo pomembna njihova organizacija, torej način, kako linearno predstaviti zapletena in med seboj povezana dejstva. Drugače povedano, kompleksne domene so graf med seboj povezanih dejstev, ki ga je vsaj klasično za namene specifikacije in tudi za podajanje navodil računalniku potrebno zapisati linearno kot poved za povedjo v specifikaciji oziroma ukaz za ukazom pri klasičnih načinih podajanja navodil računalniku.

V duhu ideje enotnega zapisa je smiselno uporabiti idejo, opisano pri obravnavi orodij modelno vodenega razvoja, kjer se celovite končne rešitve dobi z opisovanjem posameznih vidikov in preslikovanjem teh rešitev v delne rešitve, ki se sestavijo v končno celovito rešitev. Tako se na primer na enem mestu definira osnovne koncepte domene, iz katerih se generirajo razredi za te koncepte, na drugem mestu pa se z drugimi izraznimi zmožnostmi zapisa opiše določen drugi vidik, pri čemer se uporabi v opisu prvega vidika uporabljene koncepte in se iz tega drugega opisa generira programska koda, ki uporablja generirane razrede iz prvega vidika. Tako je tudi pri oblikovanju idealnega zapisa smiselno težiti k opisovanju posameznih vidikov posamično v zaključenih manjših relativno univerzalnih enotah tako, da se te enote lahko sklicujejo na druge tako definirane enote. S pametnim oblikovanjem takih osnovnih enot zapisa je možno izdelati precej idealen in uporaben jezik, ki posamezne vidike opisuje na najbolj primeren način in tako omogoča najbolj

kompaktne in intuitivne visokonivojske zapise brez problema izgube namena.

Postavlja se tudi vprašanje različnih nivojev opisov oziroma, ali je pri podrobni specifikaciji zares potrebno podajati višjenivojska dejstva, ki povzemajo skupek dejstev na idealnem nivoju podajanja dejstev. Ta višjenivojska dejstva je možno uporabiti za organizacijo dejstev na idealnem nivoju v določene smiselne enote in za sestavljanje teh enot v večje enote in na koncu v končno programsko rešitev. To je podobno, kot je celotna programska rešitev v klasičnih jezikih organizirana v metode ali funkcije, ki kličejo druge funkcije nekako po principu *deli in vladaj*.

Nekatera višjenivojska dejstva specifikacije, kot na primer v eni povedi podan kratek opis namena specifikacije, v smislu končne programske rešitve na prvi pogled nič ne pomenijo, vendar se da veliko teh dejstev v programski rešitvi smiselno uporabiti, na primer za naslove oken, namige (angl. hint) in podobno. V eni povedi podan kratek opis namena specifikacije bi bilo smiselno uporabiti v standardnem *O programu* oknu.

Glavni problem ugotavljanja smiselnosti in pravilnosti posameznih univerzalnih dejstev za namene specifikacije in programske rešitve je verjetno nenavajenost na idejo enotnega zapisa oziroma iskanja idealnih načinov podajanja dejstev, ustreznih tako za specifikacijo kot za generiranje programske rešitve.

Podajanje znanja

Pri iskanju idealnega zapisa je smiselno pogledati tudi idealne načine podajanja znanja za namene uporabe ljudi.

Največjo zbirko znanja danes predstavlja Internet, večinoma v obliki spletnih strani. Spletne strani so v osnovi napisane v HTML jeziku, ki ima za razliko od navadnih tekstov tudi možnost povezav (angl. hyperlink) na druge spletne strani.

Spletne strani v splošnem možnosti povezav ne izkoriščajo dobro in seveda tudi ne predstavljajo zapisov v smislu dobrih specifikacij znanja. Bolje pa jih izkoriščajo poskusi bolj formaliziranega podajanja znanja, primer takega spletnega mesta je zelo znana spletna enciklopedija Wikipedia[2]. Podajanje znanja na tovrstnih spletnih mestih je precej podobno zgoraj opisanim smernicam idealnega podajanja znanja. Gre za oblikovanje majhnih enot znanja, kjer se podaja le za opisovani pojem pomembne informacije, za razlago v opisu uporabljenih manj razumljivih ali na novo definiranih pojmov pa se poda povezave na opise teh pojmov.

Zanimiva oblika podajanja znanja je tudi *bliki*, kombinacija Wikipedia *wiki* zapisa in *blog* koncepta, ki ga za podajanje znanja zelo dobro uporablja znani računalniški strokovnjak Martin Fowler na svojem spletnem portalu[3], ki svoje ideje podaja v obliki manjših enot, v katerih se preko povezav sklicuje na druge enote.

Te ideje so se razvile iz splošnega znanstvenega pristopa oblikovanja elementarnih enot znanja in njihovega poimenovanja kot načina za učinkovitejše komuni-

[2] http://www.wikipedia.org/
[3] http://martinfowler.com/bliki/

ciranje. Izvedba podajanja znanja s povezavami na razlago manj znanih pojmov pa je naravnost idealna za natančne nedvoumne in popolne specifikacije.

Ideja idealnega zapisa – izvedljiva specifikacija

Idealen zapis na konceptualnem nivoju, v smislu enotnega zapisa za podajanje specifikacij, ki so primerne za prevedbo na računalnikov osnovni nivo, je tako kombinacija vseh do sedaj podanih idej. Idealen specifikacijski zapis bi tako bilo smiselno poimenovati *izvedljiva specifikacija* in bi moral biti:

- *sestavljen iz množice enot*, od katerih vsaka opisuje določen ožje definiran vidik oziroma domeno in se pri tem sklicuje na pojme, ki so opisani v drugih takšnih enotah,
- *v visoko formaliziranem zapisu*, ki bi moral biti kombinacija predvsem visoko formaliziranega splošno-namenskega jezika, zelo podobnega naravnim jezikom z dobrimi lastnostmi oblikovanja in uporabe novih konstruktov in manjše množice specifičnih zapisov, s prednostjo boljše izraznosti in slabostjo zahtevnosti njihovega poznavanja,
- *s poudarkom na zapisu za izvedljivo specifikacijo*, torej zapisom, ki je primeren za popolne specifikacije in hkrati direktno primeren za avtomatično preslikavo v celovite končne programske rešitve brez dodatnega ročnega dela.

4.2.3 Dokumentacija

Podobno kot specifikacija in zapis navodil v računalniku razumljivem nivoju predstavljata podvajanje, je tudi dokumentacija neke vrste podvajanje že zapisanih dejstev v specifikaciji oziroma implementaciji.

Dokumentacij je več vrst, tako uporabniška dokumentacija ne obsega vseh informacij specifikacije, ampak le opis funkcionalnosti programske rešitve, kar bi se dalo definirati kot specifikacija do določenega nivoja podrobnosti. Podrobna razvijalska dokumentacija programske kode oziroma izdelkov prvega reda pa se uporablja za podajanje informacij, ki jih je iz izdelkov prvega reda zaradi problema izgube namena težko razbrati. Tako različne vrste dokumentacije večinoma zajemajo le podmnožice informacij specifikacije oziroma implementacije.

Namen tega razdelka je opozoriti na dokumentacijo kot tretjo obliko istih dejstev že podanih v specifikaciji in implementaciji; s tem problemom se naloga ne ukvarja podrobneje. Idealna ideja rešitve pa je verjetno en sam zapis za specifikacijo, navodila računalniku in dokumentacijo, kar predlagani zapis do določene mere že izpolnjuje.

4.3 Dejanska rešitev

V nadaljevanju bo predstavljena rešitev zgoraj konceptualno podane ideje *izvedljive specifikacije*.

Treba se je zavedati, da je idealna in celovita izvedba izvedljive specifikacije zelo obsežen projekt, zato je bila v okviru naloge domišljena in tudi razvita osnova celotne rešitve. Natančneje povedano, domišljeno in razvito je bilo splošno ogrodje, ki omogoča izdelavo celovite, praktično uporabne rešitve. Zaradi prevelike obsežnosti in že večkrat omenjenega pomanjkanja teoretičnih rešitev pa je bila na tej splošni osnovi razvita podpora za osnovne splošno namenske koncepte ter specifična rešitev za domeno grafičnega vmesnika. Za praktično in splošno uporabno rešitev bi bilo to izvedbo treba še precej dopolniti.

Predstavljena rešitev je le ena izmed možnih izvedb *izvedljive specifikacije*. Tako je v nadaljevanju treba na vse sprejete odločitve gledati kot na izbiro ene izmed večih, včasih ne najbolj idealnih, alternativ na poti k izdelani praktični rešitvi *izvedljive specifikacije*.

Predstavitev rešitve v sledečih razdelkih vsebuje opis procesa načrtovanja zapisa in ogrodja rešitve, podprte izrazne zmožnosti, primer zapisa, podrobnosti implementacije rešitve ter nekaj končnih komentarjev.

4.3.1 Načrtovanje zapisa in ogrodja

Najprej je treba domisliti obliko in pomen zapisa dejanske izvedljive specifikacije. Po zgornjih ugotovitvah bi ta zapis moral omogočati oblikovanje posameznih ožje definiranih enot, na katere se je možno sklicevati v drugih takšnih enotah. Same enote bi morale biti podane v visoko formaliziranemu in razširljivemu jeziku precej podobnemu naravnim jezikom, ki mora biti seveda avtomatično prevedljiv na računalniku razumljiv nivo.

Enote specifikacije

V klasičnih specifikacijah so vsebinsko zaključene enote podane kot podpoglavja, pomen oziroma tema enote pa je podana v naslovu podpoglavja. Naslov in vsebina sta tako dva osnovna, precej ločena elementa enote specifikacije. Naslovi so v bistvu zelo kratke oznake enote specifikacije, njihova hierarhija tvori kazalo in tako skupaj prikazujejo grobo zgradbo celotne specifikacije. Vsebina enote specifikacije je sestavljena iz zaporedja nedvoumnih dejstev, ki podajajo za reševani problem pomembno abstrakcijo ozko usmerjenega vidika oziroma domene. Ta dva elementa enote sta po eni strani vsebinsko povezana, hkrati pa precej ločena, saj se v naslovu ne sklicuje na vsebino, v vsebini pa se sklicuje na koncepte definirane v drugih enotah, kar zahteva navajanje oznak oziroma naslovov teh drugih enot. Navajanje uporabe teh drugih enot tako striktno gledano ne sodi direktno v vsebino, ampak ga je v bolj formalnih oblikah smiselno podajati na določen standarden način ločeno od vsebine.

Sama specifikacija je podana v datotekah. Zaradi ponovne uporabe in omogočanja fleksibilnosti pri organizaciji dejstev v enote je smiselno omogočiti zapisovanje enot specifikacije v več datotek in omogočiti sklicevanje na enote v drugih datotekah. V eni datoteki je smiselno omogočiti zapisovanje večih enot,

zato je treba definirati ločevanje teh enot med seboj.

Predstavljena različica tako podpira navadne tekstovne datoteke, ki lahko obsegajo več zaporednih enot, od katerih je vsaka enota sestavljena iz treh zaporednih delov:

- *ime enote* oziroma njen formaliziran naslov; zaradi razmejevanja enot je to ime podano v vrstici, ki se začne z *'DEFINTION:'*, ki mu sledi ime enote,
- nič ali več *sklicev na druge enote* v trenutni ali drugih datotekah v vrstici, ki se začne z *'USES:'*, ki mu sledita z dvopičjem ločena neobvezno (v primeru sklicevanja na enoto v isti datoteki) ime datoteke in obvezno ime sklicevane enote,
- *vsebina enote*, ki bo obravnavana v nadaljevanju.

Vsebina enot v naravnem jeziku

Sama vsebina enot bi po zgornjem konceptualnem razmišljanju morala biti strogo formalizirana, hkrati pa s konkretnim zapisom precej podobna naravnim jezikom.

Za dosego omenjenih zahtev je smiselno začeti razmišljati o realnih specifikacijah, za začetek o specifikaciji splošnih domen. Tako bi se na primer osnovna specifikacija domene *avtomobilov*, kjer je osnovni koncept *avto*, ki ima v izbrani preprosti abstrakciji volan, kolesa in zavore, začela nekako takole:

```
Avtomobili

Avto ima volan, kolesa in zavore.
Volan obrača kolesa, zavore zmanjšujejo hitrost.
```

Iz podanega primera oziroma predstav o obsežnejših primerih je moč hitro videti, da gre za opisovanje določenih osnovnih konceptov domene, njihovih lastnosti v smislu razmerij med temi koncepti ter dogajanja v zvezi s temi koncepti.

Različne možnosti podajanja istih dejstev

Zgoraj naveden primer specifikacije avtomobilske domene je v svoji majhnosti precej nedvoumen, za identificirane izrazne zmožnosti primera bi bilo možno tudi izdelati prevedbo na računalniku razumljiv nivo, vendar je moč hitro videti, da v naravnih jezikih obstaja veliko ekvivalentnih načinov nedvoumnega podajanja istih dejstev. Vse te izrazne oblike so zelo dobrodošle v poeziji in podobnih standardnih uporabah, pri oblikovanju visoko formaliziranega jezika pa je smiselno razmišljati o omejevanju vseh teh izraznih zmožnosti.

Možnost podajanja določenih dejstev na več načinov (povedano obrnjeno) pravi, da za en sam pomen obstaja več njegovih predstavitev v naravnem jeziku. To pa v teoriji jezikov pomeni en abstrakten zapis s pomenom in več konkretnimi zapisi. Tako se je treba primarno osredotočiti na abstrakten zapis in njegov

pomen ter za konkreten zapis izbrati enega ali morda celo nekaj najbolj primernih različic.

S tega vidika je za splošno-namenske zmožnosti izvedljive specifikacije smiselno oblikovati pojem *koncepta domene* z njegovimi *lastnostmi* oziroma *relacijami* in *dinamičnim vidikom* oziroma *obnašanjem*. Ravno podana ideja je v bistvu ideja objektnega pristopa, ki predstavlja danes najboljšo teoretično osnovo splošno-namenskega izražanja.

Določanje konkretnega zapisa vsebine

Tako ostaja odprto vprašanje izbire konkretnega zapisa za obravnavane splošno-namenske abstraktne gradnike. Ugotovljeno je že bilo, da je konkretnih zapisov zelo veliko, zato bi bilo smiselno zmožnosti oblikovanega jezika za specifikacije omejiti. Na predpostavkah oblikovanja idealnega zapisa in prenašanja bremena prevedbe oziroma dela iz človekovih ramen na računalnik je smiselno podpreti več načinov podajanja istih dejstev. To pomeni na nek način več konkretnih zapisov ali celo različic konkretnih zapisov za konkretizacijo enega samega abstraktnega zapisa z enim samim pomenom. Vse te ugotovitve seveda veljajo ob predpostavki ohranjanja nedvoumnosti, pa tudi v zdravi meri različic konkretnih zapisov, saj naj bi bila ena izmed osnovnih pozitivnih posledic uporabe jezika podobnega naravnim jezikom tudi v majhnem zahtevanem obsegu znanja za razumevanje takšnih zapisov.

Koncept kot osnovni element jezika

Tako je prvo splošno-namensko dejstvo, za katerega je smiselno oblikovati zapise in pomen, pojem *koncepta* (iz opisovane domene). V zgornjem primeru za domeno avtomobilizma je treba povedati, da je *Avto* eden izmed konceptov opisovane domene.

Ker je beseda *koncept* uporabljena kot osnovni splošni element razvijanega jezika, bodo zapisi in pomen predstavljeni s pojmi iz objektnega sveta. Tako je *koncept* v osnovi definiran kot razred z imenom *Koncept* in nizovnima atributoma *ime* in *domena* koncepta. Gre za en meta nivo višje dejstvo, s katerim je na enem meta nivoju nižje možno izraziti dejansko specifikacijsko dejstvo, na primer, da je *Avto* koncept in da sodi v domeno *avtomobilizem*.

Za zamišljen abstraktni zapis in njegov pomen pa je po ideji več konkretnih zapisov dejstvo obstoja *koncepta* možno oziroma smiselno zapisati na različne načine, na primer:

```
<ime_koncepta> je koncept v domeni <ime_domene>.
V domeni <ime_domene> obstaja <ime_koncepta>.
<ime_domene> vsebuje <ime_koncepta>.
```

Vprašanje izbire ene ali manjšega števila izmed podanih oziroma drugih konkretnih zapisov je precej zahtevno in bo obravnavano kasneje.

Odločitev o potrebi specifikacije določenega pojma ali stvari kot koncepta domene z vsemi pomembnimi podrobnostmi je ena izmed na začetku omenjenih izbir med več alternativami. Dejanska odločitev nakazuje naravo razvijanega jezika, katerega namen ni biti naravni jezik, iz katerega prevedba pretežno ugiba pomen vsebine, ampak visoko formaliziran nedvoumen jezik.

Osnovna ideja izvedbe preslikovanja

Ker je pri načrtovanju jezika treba definirati tudi postopek preslikovanja, naravno razmišljanje o konceptu kot o razredu *Koncept* z atributoma *ime* in *domena*, nakazuje primernost uporabe objektnih programskih jezikov ali vsaj pristopov za izvedbo preslikave. Ta smer razmišljanja tudi nakazuje oblikovanje vmesnega jezika, v katerega je smiselno najprej preslikati več morebitnih konkretnih zapisov, in iz tega vmesnega jezika nato generirati direktno prevedljivo programsko kodo.

Omenjena prva prevedba je bolj prepoznavanje oziroma razčlenjevanje razvijanega specifikacijskega jezika, zapisanega v različnih konkretnih zapisih, v vmesni nivo enega samega, za nadaljnjo preslikavo primernega abstraktnega zapisa. Razbitje problema celotne preslikave na razčlenjevanje v vmesno obliko in preslikavo te oblike v programsko kodo je v tem primeru skoraj nujno, saj bi bila preslikava iz vhodnega razvijanega jezika z več konkretnimi zapisi direktno v programsko kodo zelo zapletena.

Namen omenjanja različnih konkretnih zapisov za izražanje dejstev z istim pomenom ni v spodbujanju oblikovanja jezika v tej smeri, ampak kot možnost, ki jo je treba trezno uporabljati.

Ideja dvostopenjske definicije elementov zapisa

Tako je smiselno določene elemente zapisa, v smislu oblikovanja razredov po objektnem principu in razčlenjevanja različnih konkretnih zapisov v primerke teh razredov, definirati dvostopenjsko:

- identifikacija določene abstrakcije in njena formalizacija v objektni razred z določenimi atributi,

- določanje možnih zapisov podajanja te abstrakcije v smislu podajanja njenih neobveznih atributov; po želji v večih konkretnih zapisih.

S tema dvema nivojema je dosežena velika fleksibilnost oblikovanja jezika, ki pa jo je treba pazljivo uporabljati.

Prva različica ideje je imela drugo stopnjo razbito na dve, kjer je bilo mogoče za isti nabor neobveznih parametrov podati več konkretnih zapisov, vendar se je to izkazalo za preveliko fleksibilnost, ki je povzročala napačno razmišljanje in odvračala od oblikovanja minimalnega nabora konkretnih zapisov.

Primer: gumb grafičnega vmesnika

Za ponazoritev povedanega je v nadaljevanju podan primer analize in načrtovanja za gumb grafičnega vmesnika v zgornjih dveh stopnjah:

- v prvi stopnji se ugotovi obstoj gumba in določi njegova abstrakcija, ki vsebuje atribute: obvezno *ime*, obvezen *tekst* in neobvezna *barva*,
- v drugi stopnji se na primer določi dve možnosti podajanja gumba; ena samo z obveznima atributoma v obliki '*[ime] je gumb s tekstom "[tekst]"*', druga pa z vsemi atributi v obliki '*[ime] je [barva] gumb s tekstom "[tekst]"*'.

Omenjeni dve možnosti podajanja gumba sta zgoraj podani kot primer dveh načinov podajanja gumba, bolj idealno pa jih je podati kot eno samo možnost z označevanjem dela *'[barva] '* kot neobveznega, kar razširja idejo statičnega konkretnega zapisa v idejo konkretnega zapisa z določenimi neobveznimi deli oziroma širše z nekaterimi deli, ki se lahko pojavijo večkrat.

Tako definirana dvostopenjska definicija, skupaj z možnostjo neobveznih delov, se je izkazala kot najbolj primeren način definiranja abstraktnega zapisa jezika, njegovega pomena in večih konkretnih zapisov.

Podobnost ideji poslovnih pravil

Nakazana oblika podajanja dejstev kot statičnega konkretnega zapisa z možnostjo neobstoja oziroma ponavljanja nekaterih delov in z določenimi spremenljivimi, ob konkretni uporabi določljivimi deli, je precej podobna tudi ideji *poslovnih pravil* (angl. business rules). Pri naprednih izvedbah poslovnih pravil se pravila namreč definira v osnovnem programskem jeziku oziroma danih izraznih možnostih, nato pa se za takšno definicijo določi še *verbalizacijo* (angl. verbalization), s katero se ta pravila podaja v obliki formaliziranega naravnega jezika in s tem podobni tudi predlagani obliki *izvedljive specifikacije*.

Predstavljen predlog je tako mogoče razumeti tudi kot podporo izdelavi programskih rešitev z uporabo širokega splošno-namenskega nabora in drugih ožjih domenskih naborov verbaliziranih pravil, kjer je zapis programske rešitve s temi pravili hkrati tudi specifikacija programske rešitve.

Izbira naravnega jezika

Zaradi sklanjatev in drugih lastnosti je slovenski jezik manj primeren vir besed za razvijan visoko formaliziran jezik. To se pokaže že na zgoraj podanem pravilu '*[ime] je [barva] gumb s tekstom "[tekst]"*', katerega dejanska specifikacijska trditev bi vsebovala '*... je rdeč gumb ...*'. Pri grafičnih komponentah, katerih oznake so ženskega spola, pa bi bilo smiselno namesto '*rdeč*' seveda uporabiti *'rdeča'*, na primer '*... je rdeča površina ...*'. Problem je rešljiv, vendar le po nepotrebnem zaplete rešitev in na nek način naredi jezik manj formalen.

Tako je bil kot osnova za oblikovani jezik izbran angleški jezik, ki nima toliko teh 'problemov'. Tako bodo v nadaljevanju primeri in definicije zmožnosti jezika podani v originalnem zapisu, ki se bere kot angleščina, kar je ekvivalentno podajanju definicije in primerov klasičnih programskih jezikov v njihovem konkretnem zapisu.

Dodatne izrazne zmožnosti

Za sestavljanje smiselnih pomenski enot s tako definiranimi osnovnimi gradniki sta bila dodani še dve izrazni zmožnosti. Prva zmožnost je sestavljanje daljših povedi iz z vejico ločenih osnovnih gradnikov jezika. Druga zmožnost pa je neobveznost eksplicitnega navajanja opisovanega pojma, kjer se opisovani pojem razbere iz konteksta. Za to je uporabljena angleška beseda *it*. S tema dvema izboljšavama se ohranja visok nivo formalizma, hkrati pa so takšni zapisi bližje človeku.

V zgoraj podanem primeru, prevedenemu in z bolj razdelanim konkretnim zapisom, *'Potrdi is a button. Potrdi is red. Potrdi has text "Test".'* je na primer mogoče *potrdi* zamenjati z *it* in tako dejstva krajše in človeku razumljiveje podati kot *'Potrdi si a button, it is red, it has text "Test".'*. Takšna specifikacija deluje bolj naravno, z dodatnimi zmožnostmi pa bi se dalo zapis še izboljšati, seveda ob ohranjanju visoke formaliziranosti.

V praksi se izkaže, da se oblikovanje obsežnejših konkretnih zapisov ne izkaže vedno najbolj primerno in je včasih kompleksne zmožnosti zapisov bolje predelati v elementarnejše in jih v človeku smiselni obliki podati v povedi v obliki z vejico ločenih specifikacijskih stavkov.

Primer specifikacije

Na tem mestu je za boljše razumevanje podan obsežnejši primer. Primer specificira grafični vmesnik prijavnega okna. Primer je podan v dejansko podprtem zapisu, za boljše razumevanje primera pa bo potrebno prebrati pomen izraznih zmožnosti v nadaljevanju. Statični deli zapisa so izpisani normalno (nepoudarjeno), spremenljivi pa poudarjeno. Statični deli se berejo kot angleščina in morajo biti taki zaradi zapisa jezika, poudarjeno izpisani spremenljivi deli pa so poljubni in so v primeru tako v slovenskem jeziku.

Primer na sliki 4.1 je napisan v razviti *izvedljivi specifikaciji* in je tako tudi direktno izvedljiv. Rezultat izvedbe je prijavno okno, prikazano na sliki 4.2. Uporabljen izraz *površina* je prevod angleškega izraza *'panel'*, osnovne površine za dodajanje komponente.

Povzetek

Podana rešitev ogrodja tako definira specifikacijo v obliki večih osnovnih specifikacijskih enot, ki jih je možno podati v večih datotekah. Vsaka datoteka tako vsebuje eno ali več specifikacijskih enot. Vsaka enota je sestavljena iz naslova oziroma oznake enote, definicije odvisnih enot ter vsebine enot.

DEFINITION: *PrijavnoOkno*

Prijavna površina is a panel, it is *175* pixels wide and *100* pixels high, it's color is *light gray.*

It contains a *blue "Ime :"* label, *it* contains *10* chars wide *ime vnos* text field. *It* contains a *red "Geslo :"* label, *it* contains *10* chars wide *geslo vnos* text field. *Prijavna površina* contains *prijavni gumb* button with text *"Prijava".*

Prijavna testna aplikacija is a GUI application titled *"Test prijave"* using *prijavna površina.*

Slika 4.1: *Primer zapisa v* izvedljivi specifikaciji, *ki specificira prijavno okno in je direktno izvedljiv.*

Vsebina enot je podana v visoko formaliziranemu jeziku, ki je bil do sedaj predstavljen predvsem konceptualno, njegovi dejanski konstrukti oziroma podprte izrazne možnosti pa bodo natančneje predstavljene v nadaljevanju. Bistveno je predvsem to, da je ta jezik sestavljen iz abstrakcij z enim abstraktnim zapisom in njegovim pomenom, najbolj naravno podanim v obliki objektnega razreda z obveznimi in neobveznimi atributi, ogrodje pa omogoča podajanje teh dejstev v večih konkretnih zapisih v obliki nekakšnih vzorcev zapisov z določenimi neobveznimi ali večkrat ponovljenimi spremenljivimi deli, ki se jih določi pri sami uporabi takšnih vzorcev za določeno specifikacijo po *izvedljivi specifikaciji.*

4.3.2 Podprte izrazne zmožnosti

Podprta *izvedljiva specifikacija* se podaja v obliki z vejico ločenih stavkov, kjer posamezni stavki ustrezajo vzorcem posameznih v nadaljevanju predstavljenih izraznih zmožnosti z dejanskimi vrednostmi za nastavljive oziroma določljive dele vzorcev. Za posamezno določljivo vrednost vzorcev izraznih možnosti je možno uporabiti eno ali več besed. Zapis v določljivih delih ignorira presledke in ne loči med majhnimi in velikimi črkami.

Zamišljena in izdelana je bila podpora splošno-namenski osnovi, tako statičnemu

Slika 4.2: *Rezultat izvajanja* izvedljive specifikacije *s slike 4.1.*

kot dinamičnemu vidiku, ter specifični domeni grafičnega vmesnika. Predstavljene izrazne zmožnosti predstavljajo prototip predlagane ideje *izvedljive specifikacije* in tako niso dovolj popoln nabor izraznih zmožnosti za praktično uporabo.

V nadaljevanju bodo v formalni obliki predstavljene posamezne osnovne izrazne zmožnosti v smiselnih vsebinskih sklopih, vsaka v tabeli v naslednji obliki:

- *ime* oziroma opis namena,
- *izraz* oziroma vzorec izraza, ki ima poudarjeno izpisane določljive dele, neobvezne dele v obliki *'(neobvezni_del)?'*, ponovitve določenega določljivega dela oziroma parametra pa kot *ime_dela*:*ločevalni_niz_ponovitev*,
- *pomen* oziroma razlago pomena in učinka izraza,
- *primer* uporabe izraza.

Zaradi jedrnatosti in boljše razumljivosti se nekateri primeri sklicujejo na definicije iz predhodnih primerov.

Koncept

Koncept se uporablja za opisovanje stvari oziroma pojmov, podobno kot *Razred* v objektnem svetu. Konceptu je možno definirati *domeno*, možno je tudi *dedovanje* med koncepti, podana pa je tudi nujno potrebna možnost za definiranje osnovnih podatkovnih tipov.

Osnovna definicija koncepta	
izraz:	*conceptName* is a concept
pomen:	Specifikacija obstoja koncepta z imenom *conceptName*.
primer:	*Avto* is a concept.

Definicija domene koncepta z eksplicitnim navajanjem koncepta	
izraz:	*conceptName* is in the *domain* domain
pomen:	Specifikacija dejstva, da koncept *conceptName* sodi v domeno *domain*.
primer:	*Avto* is a concept, *avto* is in the *avtobilizem* domain.

Definicija domene koncepta z implicitnim navajanjem koncepta	
izraz:	in the *domain* domain
pomen:	Specifikacija dejstva, da nazadnje specificirani koncept sodi v domeno *domain*.
primer:	*Avto* is a concept, in the *avtobilizem* domain.

Definicija osnovnih konceptov	
izraz:	maps to underlying *underlyingIdentifier*
pomen:	Posebna oblika specifikacije koncepta kot ekvivalenta razreda implementacijskega jezika. Ta vzorec je splošen način za uporabo dobrih elementarnih rešitev implementacijskega jezika. S tem vzorcem je predvsem možno definirati nujno potrebne osnovne tipe, kot so niz, cela in decimalna števila ter logični tip.
primer:	*Niz* is a concept, maps to underlying *java/lang/String*.

Nadgradnja koncepta	
izraz:	upgrading *conceptName*
pomen:	Podpora dedovanju z nadgradnjo oziroma razširjanjem (angl. extend) koncepta.
primer:	*Vozilo* is a concept. *Avto* is a concept, upgrading *vozilo*.

Lastnost koncepta

Zgoraj opisani koncept ima lahko poljubno število *lastnosti*, od katerih ima vsaka *ime* in *tip koncepta*.

Osnovna definicija lastnosti	
izraz:	*conceptName* has *propertyConceptName* named *propertyName*
pomen:	Specifikacija lastnosti koncepta *conceptName* z imenom *propertyName* in tipom *propertyConceptName*.
primer:	*Število* is a concept, maps to java *java/lang/Integer*. *Avto* is a concept, *it* has *število* named *število potnikov*, it has *niz* named *znamka*.

Definicija lastnosti z implicitnim imenom	
izraz:	*conceptName* has *propertyConceptName*
pomen:	Specifikacija lastnosti koncepta *conceptName* s tipom *propertyConceptName* z imenom enakim tipu. Krajša oblika osnovne definicije lastnosti za primere, ko je ime lastnosti enako imenu tipa te lastnosti. V spodnjem primeru koncept *Najem avta* dobi lastnost koncepta oziroma tipa *avto* z imenom *avto*.
primer:	*Najem avta* is a concept, *it* has *avto*.

Obnašanje

Specifikacija *obnašanja* (angl. behavior) oziroma algoritmov se, ob že ugotovljenem pomanjkanju idealnih teoretičnih rešitev podajanja obnašanja, zgleduje po splošno-namenskem objektnem pristopu. Tako je specifikacija obnašanja vezana na *koncept*, obnašanje ima *ime*, neobvezen *koncept* oziroma *tip* rezultata, *parametre* z *imenom* in *konceptom* oziroma *tipom* ter *vsebino* sestavljeno iz *dejstev*, ki bodo definirana v nadaljevanju.

Definicija obnašanja	
izraz:	*conceptName* knows how to *behaviorName*(producing *resultConcept*)?(using *params:and*)?:*statements*
pomen:	Specifikacija obnašanja oziroma algoritma za koncept *conceptName* poimenovanega *behaviorName*, ki je podan z dejstvi *statements*. Obnašanje lahko izračuna rezultat, v tem primeru je treba podati njegov tip z *resultConcept*, lahko pa ima tudi več vhodnih parametrov *params* ločenih z *and*. Za začetek je podan enostaven primer s še nepojasnjenim konstruktom *print*, več primerov sledi pri obravnavi parametrov in konstruktov za dejstva.
primer:	*Robot* is a concept. *Robot* knows how to *pozdravi*: print *"Pozdravljen!"*.

Definicija osnovnega obnašanja	
izraz:	*conceptName* behavior *behaviorName* of type *resultConcept* maps to underlying "*underlyingCodeTemplate*"
pomen:	Specifikacija obnašanja koncepta *conceptName*, ki je bil definiran kot ekvivalent razredu implementacijskega jezika, z imenom *behaviorName* in tipom rezultata *resultConcept* ter s predlogo programske kode *underlyingCodeTemplate*, ki lahko vsebuje niz *'{this}'* kot izhodišče za predlogo kode, kateri sledi ime metode implementacijskega jezika.
primer:	*Niz* is a concept, maps to underlying *java/lang/String*. *Niz* behavior *dolzina* of type *stevilo* maps to underlying *"{this}.lenght"*.

Programska rešitev

Za praktično uporabnost mora specifikacijski jezik imeti zmožnost definiranja vsebine programske rešitve, natančneje, kateri del naj se začne izvajati, ko se programska rešitev zažene.

Definicija aplikacije	
izraz:	*appName* is an application running *behaviorName* on *conceptName*
pomen:	Specifikacija aplikacije z imenom *appName*, ki ob zagonu začne izvajati obnašanje *behaviorName* definirano v konceptu *conceptName*.
primer:	*Robot* is a concept. *Robot* knows how to *pozdravi*: print *"Pozdravljen!"*. *Robot aplikacija* is application running *pozdravi* on *robot*.

S kasneje predstavljenim orodjem za direktno izvajanje *izvedljive specifikacije* zgornji primer specifikacije izpiše pozdravno sporočilo *'Pozdravljen!'*.

Dejstva obnašanja

Pri definiciji obnašanja je bilo povedano, da se obnašanje podaja z zaporedjem *dejstev*, ki bodo predstavljena v nadaljevanju. Izdelana rešitev podpira nabor osnovnih standardnih vzorcev podajanja dejstev, s katerimi je možno definirati kompleksna obnašanja. Podan nabor pa ni popoln, manjkajo na primer določene osnovne računske operacije, odločitveni stavki in podobno.

V dejstvih obnašanja nastopa nekaj osnovnih pojmov, katere je treba najprej pojasniti:

- *konstanta* – konstantna vrednost določenega koncepta,
- *primerek koncepta* – dejanski primerek koncepta, na primer primerek v primerih uporabljenega koncepta *avto* predstavlja en dejanski avto,
- *izdelava primerka* – proces oziroma zahteva po kreiranju novega primerka podanega koncepta,
- *operacija* – v smislu matematične oziroma podobne operacije z operandi in operatorji,
- *izraz* – sestavljen zapis iz zgoraj naštetih konceptov,
- *referenca* – možnost hrambe vrednosti določenega *izraza*.

Izrazi – Izdelava primerka koncepta	
izraz:	new *conceptName* instance
pomen:	Specifikacija izdelave novega primerka podanega koncepta *conceptName*.
primer:	new *avto* instance.

Izrazi – Začasna referenca	
izraz:	*conceptName* named *tempReferenceName*
pomen:	Specifikacija začasne reference imenovane *tempReferenceName* koncepta *conceptName*, nekakšen ekvivalent lokalne spremenljivke v programskih jezikih.
primer:	*avto* named *izposojen avto*

Izrazi – Prirejanje	
izraz:	*reference* is *expression*
pomen:	Specifikacija prirejanja izraza *expression* referenci *reference*, *referenca* in *izraz* sta pojmovno definirana zgoraj. Za vsakega od teh dveh pojmov je bila zgoraj že podana po ena specifikacijska definicija, več pa jih sledi v nadaljevanju.
primer:	*Avto* named *izposojen avto* is new *avto* instance.

Izrazi – Konstanta	
izraz:	*constantValue*
pomen:	Specifikacija konstante, konstanta mora biti podana v dvojnih narekovajih in je tako v osnovi niz, katerega pa lahko posamezne druge zmožnosti poljubno uporabljajo.
primer:	*"abc"*

Izrazi – Osnovne operacije	
izraz:	*expression1* ≪plus ali multiplied by≫ *expression2*
pomen:	Specifikacija osnovnih operacij.
primer:	*"abc"* plus *"def"*, *3* multiplied by *5*

Izrazi – Vračanje rezultata obnašanja	
izraz:	return *expression*
pomen:	Specifikacija rezultata obnašanja kot izraza *expression*.
primer:	return *"abc"* plus *"def"*

Izpis na standardni izhod	
izraz:	print *expression*
pomen:	Izpis izraza *expression* na standardni izhod kot osnovnega izhoda programov.
primer:	print *"Pozdravljen!"*

Izrazi – Parameter obnašanja	
izraz:	*paramName*
pomen:	Poimenska uporaba parametra *paramName* obnašanja v izrazu dejstva obnašanja. Parameter mora biti prej naveden v definiciji obnašanja.
primer:	*Robot* is a concept, *robot* knows how to *ponovi sporočilo* using *niz* named *sporočilo*: print *sporočilo*.

Izrazi – Lastnost trenutnega koncepta	
izraz:	*propertyName*
pomen:	Poimenska uporaba lastnosti *propertyName* obnašanja v izrazu. Gre za uporabo lastnosti koncepta v njegovih obnašanjih, kjer je možno lastnosti kar direktno navajati po imenu. Lastnost mora biti pred tako uporabo navedena.
primer:	*Avto* is a concept, *avto* has *niz* named *tip*. *Avto* knows how to *izpiši tip*: print *tip*.

Izrazi – Lastnost reference koncepta	
izraz:	*conceptInstanceReferenceName*'s *propertyName*
pomen:	Podajanje reference lastnosti *propertyName* reference primerka koncepta *conceptInstanceReferenceName*.
primer:	*Robot* is a concept, *it* knows how to *povej tip avta* using *avto* named *podan avto*: print *"Tip avta je "* plus *podan avto*'s *tip*.

Izrazi – Izvedba obnašanja	
izraz:	*behaviorName*(with *params:and*)?(on [*expression*])?
pomen:	Specifikacija zahteve izvajanja obnašanja *behaviorName* na izrazu *expression* z zahtevanimi parametri *params* uporabljenega obnašanja.
primer:	*Robot* knows how to *pozdravi*: *ponovi sporočilo* with *"Pozdravljen!"*

Vzorci za podajanje dejstva konstante, parametra obnašanja, lastnosti koncepta in obnašanja koncepta so v osnovi definirani brez statičnega teksta. Njihovo prepoznavanje se izvaja v podanem vrstnem redu.

Preprost grafični vmesnik

V nadaljevanju sledi še predstavitev izraznih zmožnosti za specifikacijo preprostega grafičnega vmesnika, ki omogoča dodajanje napisov, vnosnih polj in gumbov na *površino* (angl. panel) in specifikacijo grafične aplikacije s podano glavno površino.

Namen izdelave in predstavitve specifikacijskih zmožnosti za grafični vmesnik je predvsem prikaz možnosti oblikovanja naprednih specifikacijskih zmožnosti kot dopolnitev sicer nujno potrebnih in hkrati manj reprezentativnih splošnonamenskih zmožnosti, ki delujejo kot kopija objektnih konceptov. Podobno bi se dalo razviti tudi specifikacijske zmožnosti za druge znane uporabne rešitve, kot so na primer preverjanje veljavnosti podatkov, ideje aspektno orientiranega programiranja, deklarativne transakcije, ideje inverzije kontrole in podobno.

Površina (angl. panel)	
izraz:	*panelName* is a panel
pomen:	Specifikacija obstoja površine z imenom *panelName* kot osnovnega elementa za organizacijo drugih vsebinsko bolj pomembnih komponent grafičnega vmesnika.
primer:	*Prijavna površina* is a panel.

Velikost površine	
izraz:	*panelName* is *width* pixels wide and *height* pixels high
pomen:	Določanje širine *width* in višine *height* površine *panelName*.
primer:	*Prijavna površina* is a panel, *it* is *200* pixels wide and *100* pixels high.

Barva površine	
izraz:	*panelName*'s color is *color*
pomen:	Določanje barve površine *panelName*, podprte so osnovne barve, ki jih podajamo z angleškimi imeni.
primer:	*Prijavna površina* is a panel, *it*'s color is *red*.

Dodajanje komponent na površino	
izraz:	*panelName* contains *componentName*
pomen:	Dodajanje komponente *componentName* na površino *panelName*. Možno je dodati prej specificirano komponento ali pa kar na mestu specificirati dodajano komponento (glej zmožnosti specifikacije komponent v nadaljevanju).
primer:	*Prijavna površina* is a panel, *it* contains *gumb potrdi*, *it* contains *prijavni gumb* button with text *"Prijava"*.

Komponenta – Napis	
izraz:	(*labelName* is)?(a *labelColor*)?"*labelText*"label
pomen:	Specifikacija napisa; v najbolj okrnjeni različici le teksta napisa *labelText*, po želji pa še barve *labelColor* in imena *labelName* napisa. Različica brez imena je uporabna za specifikacijo dodajanega napisa (glej primer).
primer:	*Vnesi ime napis* is a *blue "Vnesi ime:"* label. *Površina* is a panel, *it* contains *Vnesi ime napis*, *it* contains a *green "Vnesi priimek:"* label, it contains *"Vnesi letnico:"* label.

Komponenta – Vnosno polje	
izraz:	(*charsWidth* chars wide)?(*fieldName*)?text field
pomen:	Specifikacija vnosnega polja, v najbolj okrnjeni različici le dejstva obstoja vnosnega polja, večinoma pa še imena *fieldName* in širine *charsWidth* vnosnega polja.
primer:	*Površina* is a panel, *it* contains *15* chars wide *vnosno polje za ime* text field.

Komponenta – Gumb	
izraz:	(*buttonName*)?button with text "*buttonText*"
pomen:	Specifikacija gumba, z neobveznim imenom *buttonName* gumba in obveznim tekstom gumba *buttonText*.
primer:	*Površina* is a panel, *it* contains *potrjevanje* button with text *"Potrdi"*.

Grafična aplikacija	
izraz:	*guiAppName* is a GUI application titled *"title"* using *panelName*
pomen:	Specifikacija grafične aplikacije *guiAppName* z naslovom *title* ki ima za glavno površino površino z imenom *panelName*.
primer:	*Testna grafična aplikacija* is a GUI application titled *"Test"* using *površina*.

4.3.3 Primer zapisa v izvedljivi specifikaciji

Za zmožnosti specifikacije grafičnega vmesnika je že bil podan primer na sliki 4.1 na strani 117.

Na sliki 4.3 je podan relativno kratek delujoč primer, ki specificira *avto* in njegov *najem*. V primeru je specificirana tudi manjša testna aplikacija, ki naredi najem za testni avto ter izpiše podatke o najemu in ceni najema na dan in za najemno obdobje. Na koncu primera je izpisan tudi rezultat zaganjanja.

4.3.4 Izdelava orodja za prevedbo in izvajanje

V okviru naloge je bilo razvito tudi orodje za direktno izvedbo specifikacij, napisanih po pravilih predstavljene *izvedljive specifikacije*. Orodje v izdelani različici podpira do sedaj predstavljene izrazne zmožnosti, zaradi modularne zasnove pa ga je mogoče enostavno razširiti z dodatnimi izraznimi zmožnostmi.

Orodje za izvedljivo specifikacijo je sestavljeno iz jedra in modulov. Jedro je gola osnova, ki sama po sebi ne podpira nobenih izraznih zmožnosti. Izrazne zmožnosti, tudi najbolj osnovne, so podane v modulih. Orodje je izdelano v programskem jeziku Java, uporablja pa tudi nekaj zanimivih knjižnic.

Moduli in vzorci

Modul je zbirka vzorcev, kjer *vzorec* na zunaj sestavljata njegov abstrakten zapis s pomenom, ter en ali več konkretnih zapisov. Ta zunanja podoba je bila podana pri specifikaciji podprtih izraznih zmožnosti, na primer vzorec *koncept* z opisom njegovega osnovnega abstraktnega zapisa z imenom, domeno in ostalim, ter s konkretnimi zapisi, kot je '*conceptName* is a concept'.

Poznavanje zunanjega vidika vzorca je dovolj za njegovo uporabo; izdelovalcu izvedljivih specifikacij ni treba poznati notranje mehanike uporabljenih vzorcev. Za razvoj lastnih modulov in razumevanje podrobnosti delovanja orodja pa bodo moduli in vzorci v nadaljevanju podrobneje pojasnjeni.

Jedro orodja

Jedro orodja predstavlja osnovno ogrodje orodja, kateremu se izrazne možnosti dodaja preko modulov. Jedro ob zagonu najprej avtomatično odkrije module, ki so na voljo, iz njih izlušči izrazne zmožnosti, nato pa obdela podano izvedljivo

DEFINITION: *IzposojaAvtomobilov*
USES: *Osnova*
Avto is a concept, *it* has *niz* named *znamka*, *it* has *niz* named *model*, *it* knows how to *izračunaj prikaz* producing *niz*:
return *znamka* plus *"/"* plus *model*.
Najem avta is a concept, *it* has *avto*, *it* has *stevilo* named *dni najema*, *it* knows how to *izračunaj prikaz* producing *niz*:
return *"Najem avta: "* plus *izračunaj prikaz* on *avto* plus
"za " plus *dni najema* plus *"dni."*.
Najem avta knows how to *izračunaj prikaz informativne cene* producing *niz* using *stevilo* named *cena*:
return *"Cena na dan: "* plus *cena* plus *"e, cena za "* plus
dni najema plus *"dni: "* plus *dni najema* multiplied by *cena* plus *"e."*.
Sklenjevanje najema is a concept, *it* knows how to *skleni najem* producing *najem avta* using *avto* named *najeti avto* and *stevilo* named *dni najema*:
najem avta named *najem* is new *najem avta* instance,
it's *avto* is *najeti avto*, *it*'s *dni najema* is *dni najema*, return *it*.

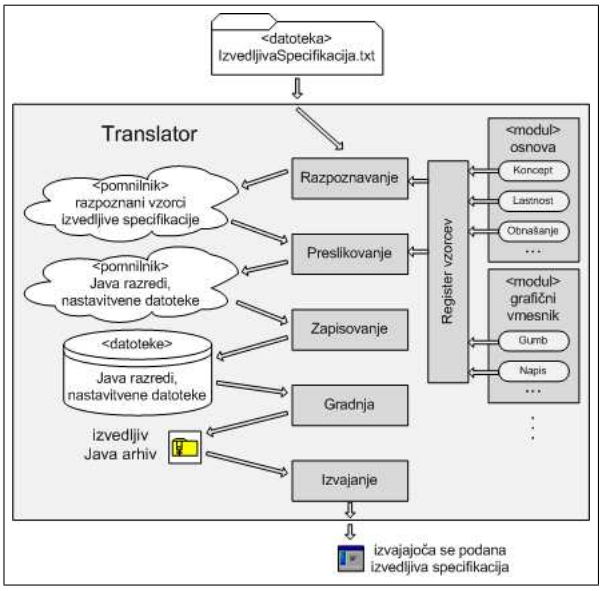
DEFINITION: *IzposojaAvtomobilovTest*
USES: *IzposojaAvtomobilov*
Test is a concept, *it* knows how to *izvedi test*:
avto named *najemani avto* is *pripravi testni avto* with *"Zastava"* and *"101"*,
najem avta named *sklenjeni najem* is *skleni najem* with *najemani avto* and
5 on new *sklenjevanje najema* instance,
print *izračunaj prikaz* on *sklenjeni najem*,
print *izračunaj prikaz informativne cene* with *10* on *sklenjeni najem*.
Test knows how to *pripravi testni avto* producing *avto* using *niz* named *znamka* and *niz* named *model*:
avto named *testni avto* is new *avto* instance, *it*'s *znamka* is *znamka*,
it's *model* is *model*, return *testni avto*.
Izposoja aplikacija is an application running *izvedi test* on *test*.

DEFINITION: *Osnova*
Stevilo is a concept, maps to underlying *java/lang/Integer*.
Niz is a concept, maps to underlying *java/lang/String*.

Izvedba primera izpiše:

```
Najem avta: Zastava/101 za 5 dni.
Cena na dan: 10e, cena za 5 dni: 50e.
```

Slika 4.3: *Primer zapisa v* izvedljivi specifikaciji, *ki specificira najem avta, skupaj s testno aplikacijo. Specifikaciji sledi izpis zagona testne aplikacije.*



Slika 4.4: *Prikaz delovanja orodja za izvedbo zapisov v* izvedljivi specifikaciji *z njegovimi stopnjami v sredini ter na levi vhodi in rezultati, ki so hkrati vhodi v naslednjo stopnjo. Na desni je predstavljen sistem vzorcev, organiziranih v module.*

specifikacijo do podane stopnje. Stopnje so logično zaključeni koraki preoblikovanja zapisa v izvedljivi specifikaciji v izvedljivo programsko rešitev in njen zagon (glej sliko 4.4). Stopnje orodja v izvajanem vrstnem redu so:

- *razpoznavanje* (angl. recognize) podane izvedljive specifikacije s pomočjo vzorcev v odkritih modulih, kar da za rezultat model s primerki vzorcev po podani izvedljivi specifikaciji; razpoznavanje bo natančneje pojasnjeno v nadaljevanju,
- *preslikovanje* (angl. generate) modela iz stopnje razpoznavanja na nivo, za katerega že obstaja prevedba, v izdelani rešitvi v razrede programskega jezika Java, poleg tega pa še druge nastavitvene datoteke ter datoteke za pomoč pri izgradnji končne programske rešitve; rezultat te stopnje je model, sestavljen iz Java programskega modela ter modela nekaterih drugih nastavitvenih in podobnih datotek,
- *pisanje* (angl. write) rezultatov preslikovanja v datoteke,
- *gradnja* (angl. build) končne programske rešitve, prevedba generiranih in zapisanih java razredov ter pakiranje teh razredov in drugih pomožnih datotek v izvedljivo programsko rešitev[4],

[4]Pri programskem jeziku Java so to *jar* arhivske datoteke, izgradnja rešitve pa obsega tudi izdelavo skripte za zagon programske rešitve.

- *izvajanje* (angl. start) v prejšnji stopnji zgrajene programske rešitve.

Zagon orodja vsaj do stopnje *gradnje* zgradi izvedljivo rešitev, ki jo je mogoče zaganjati samostojno brez uporabe orodja. Tako orodje omogoča direktno izvajanje zapisa izvedljive specifikacije, zgradi pa tudi samostojno izvedljivo programsko rešitev.

Uporaba orodja

Izdelano orodje za zagon izvedljivih specifikacij omogoča zagon primerkov zapisov v *izvedljivi specifikaciji*. Orodje se imenuje *TranslatorTool* (orodje za prevajanje) in ga je je mogoče pognati iz ukazne vrstice (angl. command line). Orodje za vhod sprejme dva obvezna parametra:

- prvi je *stopnja*, do katere se orodje izvede, in mora biti ena izmed zgoraj opisanih: *recognize*, *generate*, *write*, *build* ali *start*,
- drugi pa *oznaka* izvedljive specifikacije, ki je sestavljena iz treh, z dvopičjem ločenih delov v naslednjem vrstnem redu:
  - *ime datoteke* z zapisom v izvedljivi specifikaciji,
  - *oznaka vsebinske enote* znotraj te datoteke in
  - *oznaka izvedljivega vzorca* – trenutno sta to podana vzorca navadne in grafične aplikacije.

Primer na sliki 4.3 na strani 126, shranjen v datoteki *Najem.txt*, se tako izvede z: 'TranslatorTool start Najem.txt:IzposojaAvtomobilov:IzposojaAplikacija'.

Zaganjanje do vmesnih stopenj je zanimivo predvsem za razvijalce dodatnih izraznih zmožnosti, ki preko izpisa dnevnika (angl. log) lahko vidijo podrobno delovanje orodja. Pregledovanje dnevnika je do določene mere zanimivo tudi za uporabnike obstoječih izraznih zmožnosti, predvsem za pregledovanje postopka razčlenjevanja.

Razširjanje orodja in notranja zgradba modulov

Orodje oziroma definirane izrazne zmožnosti je mogoče razširiti z izdelavo dodatnih modulov. Obstoječi in na novo napisani moduli so sestavljeni iz vzorcev, ki določajo abstraktni zapis in pomen vzorcev, ter iz nastavitvene datoteke, ki določa konkretne zapise za posamezne vzorce ter razčlenjevanje teh konkretnih zapisov v primerke razredov vzorcev.

Razred vzorca mora ustrezati posebnemu vmesniku *Pattern*, ki zahteva definicijo:

- imena vzorca, na primer *concept* za vzorec koncepta,

- formalizacije imena primerka vzorca, na primer v '*conceptName* is a concept' je možno za *conceptName* podati več besed, formalizacija pa te besede formalizira v eno samo oznako z veliko prvo črko vseh besed, razen prve in brez presledkov,

- poljubne dodatne logike po stopnji razpoznavanja,

- preverjanja veljavnosti razpoznanega primerka vzorca,

- preslikave primerka vzorca v elemente izhodnega modela (podprti so Java razredi in zmožnosti za nastavitvene datoteke),

- imena oziroma identifikacije primerka vzorca, dobljenega z zgoraj definirano formalizacijo.

Tipičen razred vzorca, poleg implementacije zahtevanega vmesnika, seveda vsebuje tudi atribute za hranjenje lastnosti opisovanega vzorca. Na primer razred vzorca *koncept* vsebuje atribute za ime koncepta, njegovo domeno ter seznama razredov vzorcev za lastnosti in obnašanja koncepta.

Nastavitvena datoteka, ki določa izrazne zmožnosti modula, je sestavljena iz zaporedja definicij posameznih konkretnih zapisov ter razčlenjevanja teh zapisov v razrede vzorca. Pomemben je tudi vrstni red posameznih definicij v nastavitveni datoteki, ter med posameznimi moduli, če so ti med seboj odvisni. Odvisnosti med moduli je mogoče navesti v nastavitveni datoteki.

Konkreten zapis je podan kot niz posebnega formata. Ta format omogoča definicijo spremenljivih delov z oglatimi oklepaji, ki vsebujejo oznako spremenljivega dela, preko katere je mogoče v definiciji razčlenjevanja priti do dejanskih vrednosti. Mogoče je definirati tudi ponavljanje spremenljivih delov z dodajanjem dvopičja in niza, ki loči več teh spremenljivih delov. Na primer *[parameter:in]* pomeni več parametrov ločenih z besedo *in*. Možno je določiti tudi neobvezne dele konkretnega zapisa z uporabo navadnih oklepajev in vprašaja.

Razčlenjevanje zapisov se podaja v skriptnem jeziku Groovy [43], ki je zelo podoben jeziku Java. V teh skriptah je mogoče dostopati do spremenljivih delov zapisa, do modela razredov vzorcev in podobnih storitev. Možno je uporabljati tudi poljubne obstoječe ali lastne Java razrede. Skripta vrne primerek razreda vzorca ali pa prazno vrednost. Prazna vrednost pomeni, da izraz ne ustreza vzorcu.

Za boljše razumevanje je v nadaljevanju podan izsek iz nastavitvene datoteke osnovnega modula, ki definira zapis za določanje neobvezne domene koncepta:

```
<recognition>
  <expression>[conceptName] is in the [domain] domain</ex...>
  <effect>
    conceptName = utils.getRecognitionVariable("conceptName")
    conceptName =
      si.davidv.translator.pattern.concept.ConceptPattern
         .formalizePatternInstanceNameStatic(conceptName)
    concept = utils.getPattern("concept", conceptName)
    concept.domain = utils.getRecognitionVariable("domain")
    return concept
  </effect>
</recognition>
```

Klici *'utils'* metod so klici metod pripomočka, preko katerega je mogoče dostopati do razčlenjevanega modela. Vsi prikazani *'utils.get*'* klici zahtevajo obstoj iskane stvari, sicer vržejo izjemo z opisom napake, obstajajo pa tudi v različici, kjer iskane stvari niso obvezne.

Celoten postopek prepoznavanja izvedljive specifikacije

Stopnja razčlenjevanja tekst specifikacije najprej razbije na povedi, ločene s piko in te povedi naprej še na stavke, ločene z vejicami.

Dejanski zapisi se prevedejo v *regularne izraze* (angl. regular expressions) tako, da se presledki spremenijo v regularni izraz za poljubno število belih znakov (angl. whitespace), spremenljivi deli se uporabijo kot grupe regularnih izrazov, format neobveznosti delov pa že ustreza zapisu v regularnih izrazih.

Za vsak stavek se preveri ustreznost vsem možnim zapisom. V primeru ustreznosti se iz grup regularnih izrazov določi vrednosti spremenljivih delov, nato pa se izvede razčlenjevalno skripto, ki ima med drugim tudi dostop do teh spremenljivih vrednosti. Če skripta za rezultat vrne primerek razreda vzorca, se nad tem primerkom izvede preverjanje veljavnosti razpoznanega primerka vzorca, in če to uspe, je primerek vzorca umeščen v vhodni model in se tako uporabi v naslednji stopnji preslikovanja. Če pa skripta za rezultat vrne prazno vrednost, to pomeni, da razčlenjevani stavek ne ustreza vzorcu, kar je uporabljeno pri razčlenjevanju vsebine obnašanja konceptov.

Še nekaj o modulih

Modul lahko vsebuje le nastavitveno datoteko, brez razredov vzorcev. S tem je možno oblikovati dodatne konkretne zapise za že obstoječe razrede vzorcev, definirane v obstoječih modulih. Tako bi bilo možno na primer izdelati čisto drugačen konkreten zapis ali pa narediti 'prevedbo' konkretnega zapisa, ki bi namesto angleških izrazov uporabljal besede poljubnega drugega jezika.

Predstavljene izrazne zmožnosti so definirane v dveh modulih. En pokriva predstavljene osnovne splošno-namenske zmožnosti, drugi pa grafični vmesnik.

4.3.5 Dva vidika dela z izvedljivo specifikacijo

Podobno kot pri obravnavi modelno vodenega razvoja, se tudi pri *izvedljivi specifikaciji* pojavljata dva standardna vidika uporabe še ne dokončno izoblikovanih pristopov.

Prvi vidik je vidik uporabnika *izvedljive specifikacije*, ki piše izvedljive specifikacije z danimi izraznimi zmožnostmi. Tak razvijalec končnih rešitev mora tako poznati le izrazne zmožnosti izvedljive specifikacije in imeti na voljo napredno orodje za pisanje in izvajanje izvedljivih specifikacij, podrobnosti delovanja orodja in modulov ga ne zanimajo, tako kot danes ni več zanimiva prevedba klasičnih programskih jezikov na računalniku razumljive nivoje.

Drugi vidik je vidik razvijalca dodatnih izraznih zmožnosti v obliki dodatnih modulov. Podobno kot pri modelno vodenem razvoju tudi izvedljiva specifikacija še ni dovolj dodelana za delo izključno v smeri prvega vidika. Tako je za praktično uporabo treba včasih delovati tudi v smeri drugega vidika, pri čemer pa se je treba stalno osredotočati na idealnost zapisa izvedljive specifikacije. Za doseganje te idealnosti pa je najbolje pozabiti na trandicionalno reševanje obravnavanih problemov in tako neobremenjeno najprej poskušati razviti idealen zapis in šele nato razmišljati o prevedbi tega zapisa.

4.4 Predlogi za izboljšave izdelane rešitve

Predlagana konceptualna rešitev se nam zdi dobra, izdelana rešitev pa ni idealna in ni dovolj popolna za praktično uporabo. Zato bo za konec podanih nekaj kritik izdelane rešitve ter predlogov izboljšav predvsem na področju izraznih zmožnosti.

4.4.1 Osnovno ogrodje

Pri osnovnem ogrodju je najbolj vprašljiva uporaba regularnih izrazov in njihovega vrstnega reda za podajanje konkretnega zapisa ter omejenost le na tekstovne konkretne zapise.

Razpoznavanje z regularnimi izrazi

V prvi različici so bili za razpoznavanje izbrani regularni izrazi zaradi enostavnosti spreminjanja izraznih zmožnosti razvijanega jezika, kar je zelo priročno za eksperimentiranje. Pri vpeljavi dodatnih izraznih zmožnosti pa bi se uporaba regularnih izrazov skupaj z opisanim postopkom razčlenjevanja lahko pokazala kot manj primeren način razpoznavanja. Ker gre za čisto formalen jezik, bi bil morda primernejši klasičen pristop k definiciji in razčlenjevanju jezika.

Tekstovni konkretni zapis

Kljub kritiki primernosti grafičnih konkretnih zapisov bi bilo v izvedljivi specifikaciji mogoče smiselno omogočiti podporo grafičnim konkretnim zapisom. Tako

bi se konceptualna ideja specifikacijskih enot s tekstom razširila na idejo mešanja teksta in diagramov. Vendar še vedno zagovarjamo prednost tekstovnih opisov iz vseh razlogov, naštetih pri oblikovanju konceptualne ideje izvedljive specifikacije. Grafične zapise tako obravnavamo podobno kot slike v tekstih. Slik je v primerjavi s tekstom malo, uporabljajo se v redkih primerih, ko so bolj primerne za ponazoritev določenih dejstev, ali pa le kot dodaten način izražanja v besedilu že podanih dejstev.

4.4.2 Moduli

Moduli podajajo izrazne zmožnosti *izvedljive specifikacije*. Podane izrazne zmožnosti so precej osnovne in pomanjkljive. Iz začetne potrebe po splošno-namenskih izraznih zmožnostih se večina praktično podprtih izraznih zmožnosti zgleduje po idejah objektnega pristopa, s čimer so napredne izrazne zmožnosti nakazane le z izraznimi zmožnostmi preprostega grafičnega vmesnika. Tako bodo v tem razdelku v smiselnih vsebinskih sklopih navedene ideje za dodatne izrazne zmožnosti, ki bi jih bilo mogoče implementirati v obliki dodatnih modulov za predstavljeno orodje za izvedbo izvedljive specifikacije.

Manjkajoči splošno-namenski konstrukti

Predstavljenemu splošno-namenskemu modulu manjka nekaj osnovnih konstruktov. Predvsem mu manjkajo odločitveni stavki in osnovne zanke. Ta dva konstrukta zahtevata tudi definicijo blokov stavkov, nad katerimi delujeta. Manjka tudi podpora za zbirke elementov (angl. collections).

Pri *bloku stavkov* je treba nekako označiti njegov začetek ali vsaj konec. V smeri podprtih izraznih zmožnosti to dejstvo deluje kot zelo neugodno. V klasičnih specifikacijah se blokov večinoma ne označuje eksplicitno, kar pa včasih povzroča probleme pri razumevanju zahtevnejših delov specifikacij. Za označevanje blokov stavkov bi morda bili najbolj primerni kar oklepaji, na primer zaviti oklepaji. Alternativa je uporaba posebnih besed za začetek in konec bloka, vendar se to ne sklada z berljivostjo izvedljive specifikacije.

Pogojni stavek je sestavljen iz pogoja ter bloka stavkov pri izpolnjenem pogoju in bloka stavkov pri neizpolnjenem pogoju. Smiselno bi bilo uporabiti kar standardno obliko ‘If *pogoj* then *blok pri izpolnjenem pogoju* else *blok pri neizpolnjenem pogoju*’.

Zbirke elementov bi bilo v osnovi smiselno definirati kot označevanje obstoječih referenc kot zbirk elementov določenega koncepta. Tej definiciji pa bi bilo neobvezno možno dodati razne standardne lastnosti, kot so urejenost elementov, preprečevanje duplikatov in podobno, iz teh lastnosti pa bi preslikava izbrala ustrezno implementacijo v ciljnem jeziku. Konkretni zapis bi tako bil ’(ordered)?*conceptName*s collection(without duplicates)?’.

Manjkajoči konstrukti grafičnega vmesnika

Obstoječe izrazne zmožnosti grafičnega vmesnika bi bilo treba dopolniti še z vsemi ostalimi standardnimi gradniki grafičnega vmesnika, z možnostjo določanja postavitve komponent s pomočjo *razporejevalnikov* (angl. layouts) ter s povezavo grafičnega vmesnika z drugimi izraznimi zmožnostmi. Pri tem povezovanju se zdi najprimernejša preprosta rešitev specifikacija proženja določenega obnašanja kot posledice določenih dogodkov, ki jih sproži uporabnik preko grafičnega vmesnika.

Pri zgornjem razmišljanju se postavljajo razna vprašanja glede domensko-specifičnih rešitev in njihove navezave na splošno-namenske. Ali grafični vzorci *razširjajo koncept*, torej ali se, poleg tega, da predstavljajo grafične komponente, obnašajo tudi kot *koncepti* z lastnostmi in obnašanji ter ali se jih da razširjati?

Menimo, da bi morale biti grafične komponente samo grafične komponente z zmožnostjo enostavne definicije obnašanja, ki naj se izvede kot odziv na določene dogodke. Razvoj posebnih grafičnih komponent bi bilo verjetno smiselno izvesti v obliki dodatnega modula z novimi izraznimi zmožnostmi.

To smer razmišljanja bi priporočili tudi za druge specifične razširitve, saj ideja izvedljive specifikacije ni biti kopija klasičnih programskih jezikov, ampak idealen zapis, ki že sam po sebi s svojimi izraznimi zmožnostmi predstavlja idealen nivo, če pa za določen vidik idealen nivo ne obstaja, ga je smiselno dodati v obliki modula z dodatnimi izraznimi zmožnostmi. Seveda to mnenje ni mišljeno tako striktno, da iz izvedljive specifikacije naredi tog jezik, ki ga je treba stalno razširjati.

Obstojnost podatkov

Zanimivo področje izdelave programskih rešitev je zagotavljanje obstojnosti določenih podatkov. V klasičnih programskih rešitvah je kljub dobri podpori ta vidik kar zahteven. Zadnje čase se pojavlja veliko rešitev, ki problem obstojnosti podatkov rešujejo zelo deklarativno in z veliko mero uporabe prednastavljenih vrednosti. Uveljavitev objektnih podatkovnih baz bi ta problem še dodatno poenostavila.

Neobremenjen pogled na problem obstojnosti podatkov pokaže, da so določeni primerki določenih *konceptov* v določenih časovnih intervalih obstojni in v preostalih pa ne. Zelo preprosta, hkrati pa tudi precej zadostna, je rešitev uvedbe konkretnega zapisa za določanje določenega primerka koncepta kot obstojnega oziroma kot neobstojnega. Poleg te lastnosti bi bilo treba samo še definirati meje transakcije, kar je v objektnem svetu že mogoče podati deklarativno na nivoju določene metode. Podobno bi se dalo pri izvedljivi specifikaciji določiti transakcijo, na primer kar pri klicu obnašanja, kjer bi se dalo dodatno povedati, naj se obnašanje izvrši v transakciji.

Na iskanje podatkov po podatkovni bazi pa se v splošnem gleda preveč specifično. Na idealnem nivoju ni smiselno gledati na iskanje podatkov po podatkovni bazi drugače kot na iskanje podatkov v drugih oblikah. Zato, brez odvečnega ponavljanja, za specifikacijo iskanja podatkov predlagamo rešitev,

opisano v razdelku 3.4.2, seveda z za *izvedljivo specifikacijo* primernejšim zapisom. Morda bo kdo ugovarjal tem na videz prevelikim poenostavitvam, vendar so zadostne in realno izvedljive.

V nadaljevanju bo podanih še nekaj bolj pogumnih idej. Samo podajanje lokacije podatkovne baze, v katero želimo hraniti podatke, spada v širši problem identifikacije različnih virov, ki bi se ga dalo zelo poenostaviti. Podatkovne baze načeloma sploh ni treba podati, lahko bi bila podana kar implicitno, tudi v sistemih z več kot enim uporabnikom. Same izrazne zmožnosti pa bi lahko vsebovale tudi zmožnosti administracije podatkovne zbirke v visokonivojskem smislu prenašanja podatkov med več implicitno podanimi lokacijami z *'dodaj določene podatke iz Janezovega računalnika na mojega'* in podobno.

Zanimiva so tudi vprašanja optimizacije delovanja podatkovne baze, na primer z izdelavo indeksov na določenih stolpcih. Pri takih optimizacijah se večino časa izgubi za odkrivanje dela počasne programske kode in nato še morebitnega počasnega SQL stavka. Obe počasni točki bi se dalo najti avtomatično z merjenjem časov. Iz plana izvedbe (angl. execution plan) z ocenami zahtevnosti pa bi se dalo čisto avtomatično ugotoviti problem in izdelati ustrezen indeks. Zanimivo si je tako predstavljati orodje, v katerem bi bilo mogoče zahtevati poskus avtomatične optimizacije in bi sistem sam rešil problem z načini, podobnimi zgoraj opisanemu, in le vprašal, ali je nova različica sprejemljiva.

Preverjanje veljavnosti podatkov

Pogosta operacija je tudi preverjanje veljavnosti podatkov (angl. data validation). Pravila veljavnosti podatkov je mogoče večinoma podati kar deklarativno. Te zmožnosti že imajo jeziki, ki se uporabljajo pri modelno vodenem razvoju, na primer predstavljeni jezik OCL. Tako bi bilo preverjanje veljavnosti podatkov možno izvesti z možnostjo podajanja preprostega pogoja oziroma invariante za določen koncept.

Drugi vidik problema je vprašanje, kdaj preverjati te veljavnosti. Mnogo izmed zahtevanih mest preverjanja je mogoče odkriti kar implicitno, na primer ob določanju primerka koncepta za obstojnega oziroma natančneje ob koncu transakcije za vse primerke konceptov, ki bi morali biti interno vodeni kot obstojni in bi jih logika pred shranjevanjem preverila. Seveda pa bi bilo smiselno v izrazne zmožnosti dodati možnost eksplicitne zahteve po preverjanju veljavnosti. Pri implicitno podanih mestih preverjanja bi bilo smiselno omogočiti tudi specifikacijo preprečevanja preverjanja veljavnosti.

Ta zadnja misel napeljuje na zanimivo idejo podajanja novih dejstev z zanikanjem implicitnih ali celo eksplicitnih prej navedenih dejstev s trditvami podobnimi '*nekaj* je *obstoječe* brez *nečesa*', kar je zelo podobno človeškemu razmišljanju. Take zmožnosti bi bilo morda smiselno upoštevati v dodatni predstopnji, ki bi za stopnjo razčlenjevanja izdelala malo spremenjeno specifikacijo po pravilih, podobnih podanemu zgornjemu pravilu.

Napredni pristopi, dognani v okviru programskih jezikov

V okviru obravnave programskih jezikov je bilo podanih nekaj naprednih pristopov, kot so inverzija kontrole oziroma vrivanje odvisnosti, aspektno orientirano programiranje, zaznamki in drugi, ki so se razvili in uveljavili v zadnjih nekaj letih.

Ti pristopi so v programske jezike vključeni relativno nerodno, ker ti jeziki niso bil načrtovani za tako radikalne posege. Idealno bi morale biti te dodatne izrazne zmožnosti enakopravne vsem drugim standardnim izraznim zmožnostim. Tako ti pristopi kar kličejo po čisto na novo domišljenemu načinu podajanja navodil računalniku, česar poskus je tudi *izvedljiva specifikacija*.

Tako bi bilo smiselno izvedljivi specifikaciji po zgledu vrivanja odvisnosti dodati izrazne zmožnosti za sestavljanje kompleksnih komponent iz enostavnejših, izrazne zmožnosti za specifikacijo aspektov samih in mest njihovega apliciranja in podobno. Zaznamki pa v izvedljivi specifikaciji niso potrebni, saj v programskih jezikih rešujejo pomanjkanje omejenih, predvsem deklarativnih izraznih zmožnosti klasičnih programskih jezikov.

Smernice oblikovanja izraznih zmožnosti

Glavna smernica oblikovanja izraznih zmožnosti izvedljive specifikacije je iskanje idealne oblike podajanja dejstev brez obremenjevanja s preslikavo teh dejstev na nivoje, za katere že obstaja prevedba. To seveda ne pomeni zapisov, ki so zaradi previsokega nivoja že premalo splošni. Tako so na primer še vedno potrebne osnovne računske operacije, klasične oblike zagotavljanja obstojnosti podatkov pa bi lahko bile podane na višjem nivoju brez vseh danes potrebnih tehničnih podrobnosti.

4.4.3 Orodje

Trenutna različica orodja zna zagnati izvedljivo specifikacijo iz ukazne vrstice. Slabost vsakokratnega zagona orodja v celoti je predvsem v relativno dolgem času nalaganja modulov, natančneje prevajanja Groovy skript, ki pri podanih izraznih zmožnostih traja približno deset od manj kot dvajsetih sekund, potrebnih za celotno obdelavo in zagon krajše izvedljive specifikacije. Izvedljivo specifikacijo pa je treba urejati v enem izmed urejevalnikov tekstovnih datotek.

Specifikacijsko-izvajalno okolje

Razvojno okolje za pisanje in izvajanje izvedljivih specifikacij bi tako bilo kombinacija urejevalnika izvedljive specifikacije in možnosti njenega izvajanja. Razvojno oziroma specifikacijsko-izvajalno okolje ni namenjeno razvoju dodatnih modulov; za to so primerna že obstoječa orodja. Razvojno okolje bi bilo možno dodelati z večino naprednih možnosti, ki jih ponujajo sodobna razvojna okolja za klasični razvoj.

Ena najbolj zanimivih in uporabnih naprednih možnosti je definitivno *avtomatično dopolnjevanje* (angl. auto-complete) dejstev, tako v smislu ponujanja vzorcev izraznih možnosti, kot dopolnjevanja teh vzorcev in ponujanja možnosti za spremenljive dele. Zelo podobne možnosti že ponujajo urejevalniki poslovnih pravil (angl. business rule editor).

Zanimiva možnost je tudi *osvetljevanje zapisa* (angl. syntax highlighting), s katero bi bilo mogoče jasneje ločevati vzorce, spremenljive dele, ločili kot sta pika in vejica, rezervirani besedi *DEFINITION* in *USES* z njunimi vrednostmi in druge morebitne dodatne izrazne zmožnosti.

Dobrodošli bi bili tudi *razhroščevalnik* (angl. debugger), dobro dokumentirane izrazne zmožnosti in podobno.

4.5 Povzetek ugotovitev

Cilj razvoja in raziskav na področju razvoja programske opreme je iskanje idealnega načina podajanja navodil računalniku. Iz težav vgrajevanja novih teoretičnih dognanj v klasične programske jezike in še posebej iz nuje po višjem abstraktnem nivoju podajanja navodil računalniku se je razvil modelno vodeni pristop. Vendar tudi modelno vodeni pristop ne ponuja dejanskih rešitev za poljubne domene in ima tudi določene slabosti; predvsem je vprašljiva absolutna idealnost grafičnih modelnih jezikov in zahtevana količina znanja za njihovo uporabo in razumevanje.

Te pomanjkljivosti programskih jezikov in modelno vodenega pristopa pa so tudi eden izmed vzrokov za podvojevanje opisovanja domen, za katere je treba izdelati programske rešitve. Klasično se za programsko rešitev najprej izdela specifikacijo, po njej pa samo programsko rešitev, kar predstavlja omenjeno dvojnost opisa.

Konceptualna ideja predloga

Iz teh dveh problemov, slabosti zapisa programskih jezikov in modelno vodenega pristopa ter podvajanja opisovanja problemske domene, sledi ideja o oblikovanju novega enotnega zapisa, ki je direktno primeren za avtomatično prevedbo na računalniku razumljiv nivo, hkrati pa z dovolj visokim nivojem odpravlja problem izgube namena in je primeren tudi kot specifikacijski zapis.

Pri iskanju takega zapisa, ki se bere kot specifikacija in je direktno izvedljiv, smo preko vmesnih ugotovitev prišli do ideje *izvedljive specifikacije*. Konceptualna ideja izvedljive specifikacije je ideja zapisa, ki je zaradi organiziranosti, možnosti ponovne uporabe in razumljivosti razbit na smiselne vsebinske enote, ki se lahko sklicujejo ena na drugo, vsebina enot pa je popolnoma formalen jezik, ki se, zaradi težnje po intuitivnosti in minimalnosti znanja potrebnega za razumevanje, bere kot naravni jezik in je direktno uporaben za izvajanje brez dodatnega dela.

Izdelana rešitev

Na podlagi opisane konceptualne ideje *izvedljive specifikacije* je bila razvita ideja dejanskega formata smiselnih vsebinskih enot z oznako, odvisnimi enotami in vsebino. Vsebina mora ustrezati točno predpisanim izraznim zmožnostim, ki jih je možno modularno dodajati. Izrazne zmožnosti so v osnovi domišljene in podane v obliki abstraktnega zapisa s pomenom, za katerega se nato določi enega ali celo več konkretnih zapisov, ki predstavljajo dejanske izrazne zmožnosti za specifikacijo vsebine enot izvedljive specifikacije.

Domišljen in natančno definiran je bil tudi nabor izraznih zmožnosti za splošne namene, ki se zgleduje po objektnem pristopu kot najboljši teoretični podlagi za specifikacijo splošnih dejstev. Primer splošno namenskega konstrukta je *koncept*, ki ustreza razredu objektnega pristopa in ima v osnovi *ime*, *domeno* ter podporo za *lastnosti* in *obnašanja* (angl. behavior). Primer zapisa je '*Avto* is a concept, *it* is in the *avtomobilizem* domain, *it* has *stevilo* named *hitrost*, *it* knows how to *pospeši*: *hitrost* is *hitrost* plus *5*.'. Poleg tega je bila za prikaz primera oblikovanja od klasičnih jezikov naprednejših izraznih zmožnosti dodana še podpora za preprost grafični vmesnik. Grafični vmesnik podpira nekaj klasičnih gradnikov, kot so *površina* (angl. panel), *vnosno polje* in *gumb*, s čimer je možno specificirati grafične vmesnike s podajanjem dejstev v obliki '*Površina* is a panel, *it* contains a *15* chars wide *ime* text field, *it* contains button with text *"Potrdi"*. *Testna grafična aplikacija* is a GUI application titled *"Test"* using *površina*.'. Vse definirane izrazne zmožnosti so bile tudi podprte, zaradi obsežnosti izdelave podpore pa so bile druge izrazne zmožnosti podane le idejno.

Izdelano je bilo tudi orodje za izvajanje podprtih izraznih zmožnosti izvedljivih specifikacij. To orodje v osnovi predstavlja jedro orodja, ki se mu izrazne zmožnosti dodaja v obliki modulov. Moduli obsegajo enega ali več razredov za vzorce, ki določajo abstraktni zapis in pomen izrazne zmožnosti, poleg tega pa še nastavitveno datoteko z definicijami konkretnih zapisov in skript za razčlenjevanje teh konkretnih zapisov v primerke razredov vzorcev. Z orodjem je tako možno izvesti v tekstovni datoteki podano izvedljivo specifikacijo.

Povezanost z obstoječimi področji

Kot je že bilo povedano v izhodišču, se predlog opira na področji *modelno vodenega pristopa* k razvoju programske opreme in *programiranja z naravnimi jeziki*. Podani predlog *izvedljive specifikacije* vsebuje elemente obeh področij. Na *izvedljivo specifikacijo* je mogoče gledati kot na popolnoma formalen modelni jezik s konkretnim zapisom, ki se bere kot naravni jezik, ali obratno, kot na popolnoma formalno različico programiranja z naravnimi jeziki.

Predlogi za izboljšave

Zaradi že omenjene zahtevnosti in obsežnosti celovite praktične rešitve projekta izvedljive specifikacije je bila tako izdelana rešitev z omejenimi izraznimi

zmožnostmi. Tako so bile med predlogi za izboljšave podane določene rešitve, predvsem ideje za dodatne izrazne zmožnosti za določene nepodprte splošnonamenske konstrukte in konstrukte grafičnega vmesnika, za obstojnost in preverjanje veljavnosti podatkov in za druge sodobne teoretične rešitve, ki so vključene v programske jezike na precej neroden način in kar kličejo k snovanju povsem novega načine podajanja navodil računalniku.

Glavno sporočilo ideje

Glavno sporočilo podane ideje izvedljive specifikacije je potreba po sodobnem zapisu za podajanje navodil računalniku, ki na enakovreden način vključuje vse dobre lastnosti programskih jezikov, modelno vodenega razvoja in drugih domišljenih teoretičnih rešitev na področju razvoja programskih rešitev. Ta sodoben zapis pa bi lahko bil že na dovolj visokem nivoju, da bi bil hkrati primeren tudi za namene specifikacije. Gre torej za agregacijo obstoječe ogromne količine znanja in idej na temo razvoja programskih rešitev v kompakten visokonivojski specifikacijski jezik brez nepotrebnih tehničnih podrobnosti, ki ga je mogoče izvajati brez dodatnega dela.

Ob vsem povedanem velja za konec še enkrat poudariti, da mora biti opisani sodoben visokonivojski zapis dovolj izrazen, da omogoča specifikacijo vseh smiselnih podrobnosti programskih rešitev.

Poglavje 5

Zaključek

Zaključek podaja povzetek ugotovitev magistrske naloge ter predloge za nadaljnje možnosti raziskovanja obravnavanega področja modelno vodenega pristopa in predstavljenega predloga *izvedljive specifikacije*.

5.1 Povzetek ugotovitev naloge

V nalogi smo preučili modelno vodeni pristop k razvoju programske opreme, ki se kaže kot ena najmočnejših smernic prihodnosti na tem področju. Osnovna ideja pristopa je risanje modelov in njihovo avtomatično preslikovanje v končne programske rešitve[1]. Modelno vodeni pristop, poleg povečanja produktivnosti in zmanjšanja sredstev razvoja, obljublja povišanje *abstraktnega nivoja* podajanja navodil računalniku in s tem rešitev problema *izgube namena*, zaradi katerega iz klasične programske kode ni mogoče v celoti razbrati začetnih zahtev, po katerih je le-ta nastala.

Ovrednotenje pristopa

Modelno vodeni pristop je še v razvojni fazi, zato ga trenutno še ni mogoče uporabljati v idealni obliki izključnega risanja različnih standardnih modelov, za katere že obstajajo preslikave. Tako je treba za nepodprte domene najprej izdelati podporo modelno vodenemu pristopu v smislu modelnih jezikov in preslikav zanje, kjer je smiselno težiti k izdelavi splošnih ponovno uporabnih rešitev.

Bistvena komponenta modelnega pristopa in njegove podpore so *modeli* oziroma *modelni jeziki*, preslikave so z idealnega vidika drugotnega pomena[2]. Modelni jeziki morajo omogočati podajanje vseh potrebnih dejstev in podrobnosti obravnavane domene, kar že zagotavlja izvedljivost preslikave.

[1] Modeli se ponavadi preslikajo v programske jezike, za katere že obstajajo prevajalniki.
[2] Kot danes prevajalniki za obstoječe programske jezike.

Problem pomanjkanja podpore modelno vodenemu pristopu za poljubne domene v osnovi izvira iz pomanjkanja teoretičnih rešitev za opis teh domen, priporočeno z deklarativnim kompaktnim zapisom brez *izgube namena*. Zelo dobro je razvita podpora za podajanje splošnih statičnih vidikov sistema, slabše pa je podprto podajanje *obnašanja* (angl. behavior) in raznih specifičnih domen.

Doseganje višjega nivoja opisovanja domen je, ob predpostavki obstoja teoretičnih rešitev, v precejšnji meri mogoče doseči tudi z uporabo klasičnih programskih jezikov s pametnim oblikovanjem in sestavljanjem vsebinsko zaključenih komponent. *Bistvena dodana vrednost* modelno vodenega pristopa je tako možnost oblikovanja primernejšega kompaktnega zapisa dejstev na najbolj primernem abstraktnem nivoju, s čimer pristop omogoča in tudi spodbuja podajanje navodil računalniku na pravem abstraktnem nivoju.

Glavno različico modelno vodenega razvoja predstavlja *standard MDA*, ki ponuja jezik UML za splošno modeliranje, standard QVT za definicijo preslikav, ter standard MOF za podporo različnim vidikom dela z modeli in preslikavami na različnih meta nivojih. Standard MDA s tem ponuja splošno ogrodje, nič kaj dosti pa ne govori o dejanskih teoretičnih in praktičnih rešitvah. Zelo dobro idejo za oblikovanje dobrih izraznih možnosti za poljubne domene predstavljajo *domensko-specifični (modelni) jeziki*, ki predlagajo oblikovanje lastnih jezikov za poljubne specifične domene, za katere je seveda treba izdelati tudi preslikave. Tudi ideja domensko-specifičnih jezikov ne podaja dejanskih rešitev, ki jih zato sproti poskušajo izumljati izdelovalci podpornih orodij ter uporabniki teh orodij.

Zgradba in izdelava orodij ter prototipi

Obstoječe rešitve tako večinoma ponujajo osnovno možnost modeliranja statičnih vidikov sistema z razrednimi UML diagrami ter dodatne možnosti za podporo standardnim vzorcem dobrih teoretičnih rešitev, na primer za oblikovanje storitev (angl. service), grafičnega vmesnika in podobnih. Narisane modele znajo za relativno enostavne probleme standardne podpore poslovanju preslikati v končno delujočo programsko rešitev. Ta je večinoma sestavljena iz podatkovne baze, izdelane po generiranih skriptah, strežnika, ki nudi standardne storitve iskanja podatkov po podatkovni bazi in njihovega shranjevanja ter odjemalca, ki z uporabo strežnikovih storitev omogoča pregledovanje in urejanje podatkov. Razne nestandardne ali bolj zapletene zahteve, za katere orodja nimajo podpore, je mogoče dodelati v klasičnih programskih jezikih ali pa z izdelavo lastne modelno vodene podpore zanje.

Proizvajalci podpornih orodij za modelno vodeni pristop svoja orodja prikazujejo kot idealna orodja, s katerimi je možno iz nekaj standardnih diagramov z aktivacijo 'magičnega gumba' s preslikovanjem avtomatično izdelati končno delujočo rešitev. Razmišljanje o ponovni izvedbi že izvedenih srednje do visoko zapletenih projektov iz lastnih izkušenj, še bolj pa praktičen preizkus podpornih orodij, pokaže njihovo realno uporabnost, kjer uporaba podprtih izraznih zmožnosti zelo pospeši in poenostavi razvoj in tudi dosega želeni višji abstraktni nivo in

zmanjšanje izgube namena. Preseganje teh izraznih zmožnosti pa zahteva veliko truda v obliki klasičnega razvoja ali zelo zahtevnega izdelovanja dodatne podpore.

Že sama zdrava pamet pove, da je vse podrobnosti končne rešitve nujno treba podati. Določena dejstva so lahko podana implicitno v obliki privzetih vrednosti, vendar le vsa implicitno in eksplicitno podana dejstva skupaj definirajo vhod v preslikave, ki na izhodu ne morejo dati nič vsebinsko originalnega, če za izračun tega v vhodnih modelih ne obstaja ustrezna informacija.

V nalogi smo skozi orodja predstavili različne možnosti uporabe modelno vodenega pristopa; od možnosti izdelave lastnih podpornih orodij čisto od začetka, preko predstavitve uporabe raznih delnih rešitev, do opisa celovitih razvijalskih okolij za podporo modelno vodenemu razvoju. Opisali smo tudi možnosti uporabe idej modelno vodenega pristopa pri razvoju s klasičnimi programskimi jeziki. Izdelali in opisali smo tudi več prototipov in končnih rešitev za te možnosti.

Iz analize obstoječih orodij in praktičnega ukvarjanja s pristopom se je kot najboljši način razvoja celovitih podpornih orodij za modelno vodeni razvoj pokazala izdelava večih manjših rešitev za ozko usmerjene domene, seveda v obliki modelov in preslikav, kjer so te rešitve med seboj odvisne, v smislu naslanjanja naprednih rešitev na modele in rezultate preslikav osnovnih rešitev. Za boljše razumevanje podajamo primer rešitve *obstojnosti* (angl. persistence) razredov, ki je uporabljena v rešitvi storitvenih razredov za shranjevanje, branje in poizvedovanje po primerkih teh obstojnih razredov. Problem obstojnosti je mogoče rešiti z UML razrednimi diagrami z označevanjem obstojnih razredov s posebnim stereotipom, na primer ≪*Obstojen*≫. Problem storitvenih razredov za shranjevanje, branje in poizvedovanje podatkov pa je mogoče rešiti z modeliranjem UML razredov s posebnim stereotipom, na primer kar ≪*Storitev*≫, ki ima poizvedbe definirane s povezavami na obstojne razrede[3]. S tem so podane vse potrebne informacije za generiranje programske kode za shranjevanje, branje in poizvedovanje.

Modelno vodeni pristop tako že danes predstavlja smiselno in praktično uporabno možnost za razvoj programske opreme, predvsem v okvirih izraznih zmožnosti obstoječih orodij. Preseganje teh okvirov pa zahteva glede na čas, izkušnje in znanje precej zahtevno izdelavo dodatne podpore. Bistvena prednost obravnavanega pristopa je zapis navodil računalniku na ustreznem abstraktnem nivoju, kar rešuje problem *izgube namena*, pristop pa poleg tega še povečuje produktivnost, spodbuja oblikovanje ponovno uporabnih rešitev in podobno.

Predlog: izvedljiva specifikacija

Pri zgoraj opisanem ovrednotenju modelno vodenega pristopa in možnosti njegove praktične uporabe smo ugotovili tudi določene slabosti, predvsem pomanjkanje dejanskih rešitev, vprašljivost absolutne idealnosti grafičnih konkretnih zapisov ter problem dvojnosti dela v smislu specifikacije in izvedbe programske rešitve. Na podlagi teh ugotovitev smo oblikovali predlog nadgradnje modelno vodenega pristopa.

[3]Rešitev poizvedovanja je opisana v razdelku 3.4.2 na strani 93.

Osnovna ideja podanega predloga *izvedljive specifikacije* je oblikovanje popolnoma novega enotnega zapisa, ki je primeren za formalno specifikacijo programskih rešitev, hkrati pa je direktno izvedljiv. Na konceptualnem nivoju je to zapis, ki je zaradi organiziranosti, možnosti ponovne uporabe, razumljivosti in obvladljivosti razbit na smiselne vsebinske enote, ki se lahko sklicujejo ena na drugo, vsebina enot pa je popolnoma formalen jezik, ki se zaradi težnje po intuitivnosti in minimalnosti potrebnega znanja za razumevanje bere kot naravni jezik.

Na podlagi opisane konceptualne ideje *izvedljive specifikacije* smo razvili tudi dejanske izrazne zmožnosti. Zaradi obširnosti izdelave celovite podpore smo se omejili na podporo splošnemu podajanju dejstev po zgledu objektnega pristopa kot najboljše teoretične rešitve za to področje. Poleg tega pa smo razvili še zapis za preprost grafični vmesnik kot primer podpore specifični domeni. Primer zapisa v splošnih izraznih zmožnostih je '*Avto* is a concept, *it* is in the *avtomobilizem* domain, *it* has *stevilo* named *hitrost*, *it* knows how to *pospeši*: *hitrost* is *hitrost* plus *5*.'. Primer zapisa v izraznih zmožnostih grafičnega vmesnika pa je '*Površina* is a panel, *it* contains a *15* chars wide *ime* text field, *it* contains button with text *"Potrdi"*. *Testna grafična aplikacija* is a GUI application titled *"Test"* using *površina*.'. Zapisi so sestavljeni iz posameznih povedi, katerih stavki morajo ustrezati definiranim izraznim zmožnostim z določenimi, v zgornjih primerih kurzivno izpisanimi, spremenljivimi deli.

Izdelali smo tudi orodje za direktno izvajanje *izvedljivih specifikacij*. Poenostavljeno povedano se za zagon določene izvedljive specifikacije orodju v ukazni vrstici za parameter poda datoteko s to izvedljivo specifikacijo. Orodje s svojo modularno zasnovo omogoča enostavno dodajanje novih izraznih zmožnosti. Posamezne izrazne zmožnosti so definirane z objektnim razredom in s konkretnimi zapisi. Objektni razred, ki mora ustrezati predpisanemu vmesniku, definira abstraktni zapis, pomen in preslikavo izrazne zmožnosti. Konkretni zapisi določajo zapis izrazne zmožnosti v obliki stavkov s spremenljivimi deli in s skriptami za razčlenjevanje teh oblik v prej definiran objektni razred.

Bistvo ideje *izvedljive specifikacije* je agregacija obstoječe ogromne količine znanja in idej na temo razvoja programskih rešitev v kompakten visokonivojski specifikacijski jezik brez nepotrebnih tehničnih podrobnosti, ki ga je mogoče direktno izvajati.

Ideja nam na konceptualnem nivoju deluje precej zanimivo. Dobra se nam zdi tudi rešitev organizacije in konceptualne oblike vsebine zapisa, problem so pomanjkljive izrazne zmožnosti, delno zaradi obsežnosti oziroma nerealnosti pričakovanja podpore za poljubno domeno, delno pa zaradi že večkrat omenjenega problema pomanjkanja dobrih teoretičnih rešitev za specifikacijo poljubnih domen.

Menimo, da bi moral biti idealen nabor izraznih zmožnosti sestavljen iz dobrih splošno-namenskih izraznih zmožnosti po vzoru objektnega pristopa in iz modularno razširljivih specifičnih izraznih zmožnosti za pogosto uporabljane domene oziroma teoretične rešitve kot so grafični vmesniki, obstojnost podatkov, preverjanje podatkov, poizvedbe, prenos podatkov med sistemi in drugi standardni vzorci.

5.2 Možnosti nadaljnjih raziskav

Modelno vodeni razvoj oziroma širše področje razvoja programske opreme seveda ponujata veliko možnosti za nadaljnje raziskave.

Ena najpomembnejših smeri raziskovanja je iskanje novih in izboljševanje obstoječih teoretičnih načinov oziroma vzorcev podajanja navodil računalniku, ki bi nadomestili nerodne načine podajanja poljubnih domen s splošno-namenskimi izraznimi zmožnostmi. Neobstoj teh teoretičnih rešitev je namreč osnovni vzrok problemov *izgube namena*, preseganja začrtanih okvirov projektov in podobnih. Izumljene teoretične rešitve je nato mogoče uporabiti v sicer malo bolj nerodni obliki v programskih jezikih oziroma v boljši obliki pri modelno vodenem pristopu ali v prihodnjih naprednejših zapisih, med katere sami subjektivno uvrščamo tudi podani predlog *izvedljive specifikacije*.

Drugo pomembno področje raziskovanja je konkretizacija trenutno dobre konceptualne zasnove modelno vodenega pristopa v dejanske rešitve in orodja. Trenutno že obstajajo precej dobre rešitve, kot je predstavljeno okolje OptimalJ, vendar ne nudijo dovolj široke podpore. To pomanjkanje fleksibilnosti v večini primerov rešujejo precej nerodno z uporabo klasičnih programskih jezikov ali z omogočanjem izdelave lastne dodatne podpore, kar se večinoma izkaže za zelo zahtevno nalogo. Pogrešamo standardizacijo dejanskih rešitev; organizacija OMG v okviru standarda MDA izdeluje specifikacije posameznih domen, katerih uporabnost se mora v praksi še potrditi, podobno ponudniki orodij ponujajo različno nestandardno podporo. Pogrešamo tudi nov pogled na oblikovanje izraznih zmožnosti, ki se neobremenjen s trenutnimi rešitvami sprašuje, kako bi bilo moč določena dejstva podati najbolj idealno, ne pa, kako obstoječe, zaradi slabših izraznih zmožnosti neidealne rešitve, nadgraditi v nov način podajanja navodil računalniku.

V ožjem okviru naloge so najbolj smiselne in zanimive nadaljnje raziskave podanega predloga *izvedljive specifikacije*, predvsem v smislu dopolnjevanja obstoječih in dodajanja novih izraznih zmožnosti. Nekatere konceptualne predloge in smeri izboljšav *izvedljive specifikacije* smo že podali v poglavju 4.4.

Eno najbolj zahtevnih vprašanj predstavlja vprašanje oblike idealnih izraznih možnosti v smislu iskanja pravega kompromisa med danes vidnima možnima skrajnostima. Ena izmed teh skrajnosti so zelo obširne splošno-namenske zmožnosti in veliko število domensko-specifičnih zmožnosti v vseh možnih različicah, kar ni dobro ravno zaradi zahtevanega poznavanja te neobvladljive množice možnosti. Druga skrajnost pa je relativno omejen nabor osnovnih izraznih zmožnosti, ki po eni strani zahteva manj znanja, po drugi pa gradnjo specifičnih izraznih zmožnosti iz osnovnih.

Literatura

[1] A. Abouzahra, J. Bézivin, M. D. D. Fabro, in F. Jouault. A Practical Approach to Bridging Domain Specific Languages with UML profiles. V *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05*, San Diego, California, USA, 2005.

[2] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, in S. Neema. Developing Applications Using Model-Driven Design Environments. *Computer*, 39(2):33–40, 2006.

[3] J. Bloch. *Effective Java, Second Edition*. Prentice Hall PTR, 2008.

[4] M. Bohlen. QVT und multi-metamodelltransformationen in MDA. *OBJEKTspektrum*, 2, 2006. (Angleški prevod dostopen na: http://galaxy.andromda.org/jira/secure/attachment/10780/ QVT article mbohlen 2006.pdf).

[5] G. Booch, J. Rumbaugh, in I. Jacobsonn. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[6] K. Czarnecki in U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley, New York, NY, USA, 2000.

[7] K. Czarnecki in S. Helsen. Classification of Model Transformation Approaches. V *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[8] M. Dalgarno in M. Fowler. UML vs. Domain-Specific Languages. *Methods & Tools*, Summer, 2008. Dostopen na: http://www.methodsandtools.com/PDF/mt200802.pdf.

[9] M. Fowler in K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2002.

[10] R. B. France, S. Ghosh, T. Dinh-Trong, in A. Solberg. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer*, 39(2):59–66, 2006.

[11] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley, New York, NY, USA, 2002.

[12] D. S. Frankel in J. Parodi. *The MDA Journal*, poglavje 12. Meghan-Kiffer Press, 2004.

[13] J. Greenfield, K. Short, S. Cook, in S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley, 2004.

[14] A. Hulshout in J.-P. Tolvanen. Modeling for full code generation. *Embedded Computing Design*, august, 2007.

[15] R. Johnson in J. Hoeller. *Expert One-on-One J2EE Development without EJB*. John Wiley, 2004.

[16] S. Kelly in J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*, poglavje 1. IEEE Computer Society, 2008.

[17] A. G. Kleppe, J. Warmer, in W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2003.

[18] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, in P. Volgyesi. The Generic Modeling Environment. V *Proceedings of the Workshop on Intelligent Signal Processing at WISP'01*, 2001.

[19] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[20] S. J. Mellor in M. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2002.

[21] S. J. Mellor, A. N. Clark, in T. Futagami. Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18, September 2003.

[22] T. O. Meservy in K. D. Fenstermacher. Transforming Software Development: An MDA Road Map. *Computer*, 38(9):52–58, 2005.

[23] J. Miller in J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003. Dostopen na: http://www.omg.org/docs/omg/03-06-01.pdf.

[24] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Technical report, Object Management Group, 2007. Dostopen na: http://www.omg.org/docs/ptc/07-07-07.pdf.

[25] OMG. UML 2.0 Superstructure, v2.1.2. Technical report, Object Management Group, 2007. Dostopen na:
http://www.omg.org/docs/formal/07-11-02.pdf.

[26] T. J. Parr. Enforcing strict model-view separation in template engines. V *WWW '04: Proceedings of the 13th international conference on World Wide Web*, strani 224–233, New York, NY, USA, 2004.

[27] R. Pawlak. Spoon: annotation-driven program transformation — the AOP case. V *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, New York, NY, USA, 2005.

[28] R. Pawlak. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distributed Systems Online*, 7, 2006.

[29] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[30] R. Soley. Model Driven Architecture. Technical report, Object Management Group, 2000. Dostopen na:
www.omg.org/docs/omg/00-11-05.rtf.

[31] J. Warmer in A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition.* Addison-Wesley, 2003.

[32] B. A. Myers, J. F. Pane in A. Ko. Natural programming languages and environments. *Communications of the ACM*, 47(9), 2004.

Ostali viri

[33] Object Management Group, *http://www.omg.org/*.

[34] Borland Together, *http://www.borland.com/us/products/together/*.

[35] Rational Rose, *http://www-01.ibm.com/software/awdtools/developer/rose/*.

[36] AndroMDA, *http://www.andromda.org/*.

[37] Hibernate, *http://www.hibernate.org/*.

[38] Struts, *http://struts.apache.org/*.

[39] Velocity, *http://velocity.apache.org/*.

[40] OptimalJ, *http://frontline.compuware.com/javacentral/*.

[41] MagicDraw, *http://www.magicdraw.com/*.

[42] Graphical Editing Framework, *http://www.eclipse.org/gef/*.

[43] Groovy, *http://groovy.codehaus.org/*.

[44] Natural Programming, *http://www.cs.cmu.edu/~NatProg/*.

Izjava o samostojnosti dela

Izjavljam, da sem magistrsko delo izdelal samostojno pod vodstvom mentorja prof. dr. Viljana Mahniča. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.

David Vidrih