UNIVERZA V LJUBLJANI

Fakulteta za računalništvo in informatiko

ROK ŠTEBE

PROTOTIP ORODJA ZA MODELNO VODEN RAZVOJ JAVANSKIH PROGRAMOV

MAGISTRSKO DELO

Mentor: prof. dr. Viljan Mahnič

Ljubljana, december 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št.: 82-MAG-ISO/2007
Datum: 22. 6. 2007

Rok **Š T E B E**, univ. dipl. inž. rač. in inf.

L j u b l j a n a

Fakulteta za računalništvo in informatiko Univerze v Ljubljani izdaja naslednjo magistrsko nalogo

Naslov naloge: **PROTOTIP ORODJA ZA MODELNO VODEN RAZVOJ JAVANSKIH PROGRAMOV**

Tematika naloge:

Modelno voden razvoj (angl. Model Driven Development) je nova paradigma razvoja programske opreme, ki temelji na postopni pretvorbi modelov, s katerimi je opisan problem, v delujočo programsko kodo. Za oblikovanje modelov uporablja domensko specifične jezike, ki so definirani v okviru metamodela domene. V ta namen je bilo razvito orodje GME (Generic Modeling Environment), ki omogoča izdelavo metamodela domene in oblikovanje domensko specifičnih modelirnih jezikov.

Proučite koncepte modelno vodenega razvoja in s pomočjo orodja GME izdelajte prototip okolja za modelno voden razvoj javanskih programov. V nalogi predstavite opis vaše problemske domene z ustreznim metamodelom, opišite realizacijo omejitev nad metamodelom in izdelajte lasten interpreter za pretvorbo modela v javansko programsko kodo. Delovanje prototipa preizkusite na konkretnih primerih z vaj pri enem od programerskih predmetov in ugotovite, kako modelno voden razvoj vpliva na hitrost programiranja in kakovost programske kode.

Mentor:

prof. dr. Viljan Mahnič

Dekan:

prof. dr. Franc Solina

ZAHVALA

Mentorju Viljanu Mahniču, ki me je opomnil, kako pomembno je dokončati vsako začeto delo…

KAZALO VSEBINE

Kazalo slik

POVZETEK

V magistrski nalogi z naslovom *Prototip orodja za modelno voden razvoj javanskih programov* obravnavamo koncepte, priporočila, tehnike in metode modelno vodenega razvoja. Doseči smo želeli tri poglavitne cilje: preveriti uporabnost enega izmed najboljših odprtokodnih orodij za izdelavo domensko specifičnih jezikov, na njegovi osnovi izdelati prototip orodja za modelno voden razvoj javanskih programov ter preveriti dejansko uporabnost predstavljenih modelirnih konceptov z izbrano množico študentov prvega letnika.

V uvodnem delu magistrske naloge smo naredili obširen pregled področja modelno vodenega razvoja. Na začetku smo podrobneje opisali koncept metamodeliranja in domensko specifičnih jezikov. Sledil je opis najpogostejših pristopov izvedbe transformacij med modeli ter predstavitev načinov združevanja ročno napisane in avtomatsko generirane programske kode. Za konec uvodnega dela smo navedli še nekatere najbolj vplivne raziskovalne skupine s tega področja ter izpostavili potrebo po enotni uvedbi standardov, orodij in metodologij.

V nadaljevalnem poglavju smo izpostavili problematiko programiranja v Javi ter predstavili ideje za izvedbo praktičnega dela magistrske naloge. Opisali smo popularno odprtokodno okolja GME, s katerim izdelujemo domensko specifične jezike. Podrobneje smo se dotaknili tudi strukture njegovega ogrodja, obrazložili pomen omejitev OCL in predstavili vzroke za rabo interpreterjev.

V osrednjem delu smo podrobneje opisali prototip orodja za modelno voden razvoj javanskih programov, ki smo ga izdelali v okviru praktičnega dela magistrske naloge. Najprej smo obrazložili, kako smo koncepte programiranja v Javi opisali z ustreznim metamodelom, nato smo predstavili grafične simbole, s katerimi koncepte domene vizualiziramo v modele, definirali potrebne omejitve z jezikom OCL in se osredotočili na način izvedbe interpreterja, s katerim oblikovane modele pretvorimo v ustrezno javansko programsko kodo.

V zadnjem delu magistrske naloge smo predstavili še anketo, s katero smo od študentov pridobili informacije o izkušnjah glede uporabe prototipa. Dobljene rezultate smo predstavili in jih primerjali z ugotovitvami oz. sklepi, do katerih smo prišli sami.

KLJUČNE BESEDE

model, modelno voden razvoj (MDD), prototip orodja, javanski program, domensko specifični jezik, interpreter, omejitev OCL

ABSTRACT

The masters' dissertation titled *The Prototype of a Tool for Model-Driven Development of Java Applications* details concepts, recommendations, techniques and methods of model-driven development. We aimed to achieve three main goals: to establish the usability of one of the best open source tools for developing domain-specific languages, to develop the prototype of a tool for model-driven development of Java applications based on it, and to establish the actual usability of the presented modeling concepts with a selected group of 1st year students.

The introduction of the dissertation serves as a broad-reaching overview of the field of model-driven development. It begins with a detailed description of meta-modeling and domain-specific languages. That is followed by the description of the most frequent approaches to executing transformations between models and the presentation of methods of combining hand-written and automatically generated programming code. The introduction ends with the list of some of the most influential research groups from this field and stresses the need for unified standards, tools and methodologies to be introduced.

The following chapter focuses on problems of Java programming and presents the ideas for the execution of the practical part of the dissertation. It is followed by the description of GME, a popular open source environment for developing domain-specific languages. We describe the structure of this tool in detail, explain the significance of OCL constraints and present the reasons for using interpreters.

The central part of the dissertation has a detailed description of the prototype of the tool for model-driven development of Java applications we built within the scope of the practical part of our dissertation. It begins by detailing how the programming concepts were described with an appropriate metamodel, followed by the presentation of the graphic symbols for visualizing domain concepts into models, then a definition of the required limitations with the OCL constraint language and a focus on the method of executing the interpreter, which is used to transform the designed models into the appropriate Java programming code.

The final part of the dissertation describes the survey we conducted among 1st year students to gain information about experiences with using the prototype. We present the results and compare them to the findings and conclusions that we reached ourselves.

KEYWORDS

model, model-driven development (MDD), prototype tool, Java application, domain-specific language, interpreter, OCL constraint

1. UVOD

V zadnjih petih desetletjih so razvijalci programske opreme postopoma uveljavljali različne pristope, s katerimi so želeli hitreje in učinkoviteje razvijati kakovostno programsko opremo. V 50' in začetku 60' let je bila delovna intenzivnost na področju računalništva usmerjena predvsem v reševanje problemov obvladovanja strojne opreme. Že takrat je bilo jasno, da bi bilo za smotrno obvladovanje »strojnih« problemov, potrebno uporabiti določen nivo abstrakcije, s katerim bi se programerji-znanstveniki bolj osredotočili na obvladovanje programskih rešitev kot pa na poznavanje kompleksnega računalniškega oz. strojnega jezika. Prvi primerki takšnih abstrakcij so zagotovo zajeti v okviru prvih zbirnih jezikov, ki so strojni jezik predstavili na način, razumljiv širši množici programerjev. Namesto zaporedja znakov 0 in 1 so programerji uporabljali mnogo bolj razumljive besedne ukaze, s katerimi je bilo probleme mogoče rešiti bistveno hitreje in učinkoviteje.

Tudi realizacija prevajalnikov za tako imenovane višje programske jezike (FORTRAN, COBOL itn.), v katerih se probleme opiše v obliki, ki je podobna matematični, je bila postopna. Veliko programerjev je namreč verjelo, zaradi slabih prevajalnikov pogosto upravičeno, da so programi, ki jih napišejo v strojnem ali zbirnem jeziku, boljši od prevodov, v katere prevajalnik prevede v višjem programskem jeziku napisan program. S pojavitvijo vedno boljših prevajalnikov pa je strah pred slabo prevedenimi programi postopoma izginil in večina programerjev ni nikoli več slišala, niti želela uporabljati programskih jezikov druge generacije – zaupali so prevajalnikom in bili navdušeni nad enostavnostjo, ki jo jim je ponujala nova, bolj abstraktna programska koda.

Na podlagi izkušenj iz uvajanj prvotnih programskih jezikov je bilo razvitih še več različnih pristopov in tehnologij, s pomočjo katerih bi bilo mogoče nivo abstrakcije programske kode še povečati. Kot podpora razvoju IS so v osemdesetih letih nastala računalniška orodja, ki so jih nekateri obravnavali kot revolucionarna, danes znana pod imenom CASE (Computer Aided Software Engineering). Prvotni namen teh orodij je bil zmanjšati težave, ki so se pojavljale zaradi še vedno prevelike kompleksnosti programskih jezikov tretje generacije (npr. upravljanje s pomnilnikom, zahtevno razhroščevanje ipd.) in tako povečati produktivnost programerjev. Njihova revolucionarnost je temeljila na vpeljavi diagramskih tehnik, kot so diagram stanj, strukturni diagram in diagram podatkovnih tokov, s katerimi so obravnavano problemsko področje predstavili na višjem abstraktnem nivoju. Posamezni elementi diagrama (modela) naj bi programerju pomagali avtomatsko generirati programsko kodo, ki bi izoblikovala osnovno ogrodje računalniškega programa v razvoju.

Kljub prvotno obetavnim prednostim, ki naj bi jih razvijalci pridobili z uporabo CASE orodij, pa se orodja v praksi niso tako močno uveljavila, kot je bilo napovedano. Glavni problem teh orodij so bili togi modelirni jeziki (predstavljeni v obliki diagramskih tehnik), ki so avtomatsko generirano programsko kodo le s težavo preslikali na takratne (prehodne) operacijske sisteme (DOS, OS/2, Windows). Zaradi pomanjkanja (razumljivo šele kasneje

razvitih) potrebnih storitev kakovosti (varnosti, obravnavanja napak, transparentnosti razpečevanja kode itn.) obstoječih operacijskih sistemov, je morala biti generirana programska koda mnogo kompleksnejša, kot bi sicer lahko bila, saj je bilo nekako potrebno zapolniti vrzel neobstoječih podpornih storitev. Prav tako so bila orodja namenjena le izdelavi preprostih sistemov, ki so bila v domeni razvoja le enega razvijalca ali pa največ le manjše razvojne skupine, ki je zaradi takrat še nerazvitega koncepta repozitorija upravljala nad istim datotečnim sistemom projekta. [1] povzema, da je bil eden izmed glavnih problemov tudi ta, da nekaterih problemskih področij enostavno ni bilo mogoče modelirati, saj so bili univerzalni modelirni jeziki preveč togi in neprilagodljivi.

Napredki v razvoju različnih tehnologij, pojavitev objektno-usmerjenih metodologij in oblikovanje iCASE orodij so razvoj programske opreme približale načinu, ki je človeku resnično pisan na kožo. Z uporabo objektno-usmerjenih programskih jezikov (npr. C++ in JAVA) ter jezikov četrte generacije (SQL, Visual Basic idr.) je strukturo programov sedaj mogoče opisati na bolj izrazen oz. intuitiven način, kot je bilo to možno prej. Poleg tega so posamezni koncepti objektnega razvoja, kot sta npr. koncepta ponovne uporabnosti objektov in dedovanje, močno olajšali razvoj tudi najbolj kompleksnih računalniških sistemov. Razredne knjižnice podpornih storitev, kot so varnost, obravnavanje napak in upravljanja z računalniškimi viri, so zdaj že privzeto integrirane v posamezna ogrodja obravnavanih platform in s tem v CASE orodja, ki te plaforme podpirajo. Razvijalci se lahko tako neposredno osredotočijo na realizacijo konkretnih rešitev in se ne ubadajo z odvečnim obvladovanjem implementacijskega okolja.

Z diagramskimi tehnikami UML [13], na katerih temelji modeliranje večine današnjih CASE orodij, opišemo tako statične kot tudi dinamične vidike nastajajočega sistema. Modeliramo lahko praktično katerokoli problemsko področje in generirana visoko-abstraktna programska koda nam lahko resnično koristi pri realizaciji programske rešitve. Kljub vsemu pa so problemi iz preteklosti danes še vedno prisotni. Modelirni jezik UML je za nekatere problemske domene še vedno premalo prilagodljiv in z njim ne moremo učinkovito izraziti vsega. Vsaj ne v smislu, da bi izdelane modele lahko formalno z različnimi postopki tranformacij, avtomatizacij itn. pretvorili v končni izdelek. Pojavi novih tehnologij, letno oblikovanje najnovejših različic razvojnih platform, raznolikost standardov in nasploh vse večja kompleksnost računalniških sistemov prehitevajo napore tistih, ki bi radi vse to zajeli v vedno bolj izpopolnjena (modelirna) razvojna orodja. Zato se nam s tega stališča orodja v praksi še vedno in prepogosto zdijo neuporabna.

Vprašanja v zvezi s tem se pojavljajo tudi na področju metodološke podpore razvoju. Od devetdesetih let naprej manjše razvojne skupine prisegajo na podporo agilnih metodologij, ki zaradi svoje prilagodljivosti dinamičnemu razvojnemu procesu omogočajo obvladovati stalno spreminjajoče se zahteve naročnikov programske opreme, spodbujajo komunikativni duh znotraj projektne skupine ter zagotavljajo delujoče ter vedno bolj izpopolnjene verzije sistema v kratkih časovnih intervalih. Dobro uveljavljene agilne metodologije, kot je Ekstremno

programiranje [24], podajajo načela, tehnike in pristope, s katerimi lahko učinkovito obvladujemo predvsem razvoj programske kode. Ali lahko takšne metodologije skupaj z orodji, ki so nam na voljo oz. jih potrebujemo, s pridom uporabimo tudi pri razvoju z višje abstraktnimi (grafičnimi) modelirnimi jeziki? Ali je npr. tehnika programiranja v parih še vedno dovolj učinkovita oz. smotrna, ko bi radi modelirali dinamično komponento sistema - npr. z UML diagrami zaporedja? Ali je zaradi višje abstrakcije »kodiranja« (modeliranja) še vedno potreben programer-partner? Kako in če sploh bi bilo agilne metodologije potrebno prilagoditi, da bi jih zavoljo njihovih izjemnih rezultatov v zadnjih nekaj letih, lahko s pridom uporabljali tudi na najnovejših generacijah novodobnih razvojnih pristopov, tehnologij in orodij?

Za praktično uveljavitev prvotnega in glavnega načela CASE orodij, v okviru katerega poljubno problemsko področje modeliramo s pomočjo modelov in le te postopoma pretvorimo v delujoči izdelek, bi se morali razvoja lotiti na popolnoma drugačen način. Kaj če bi vsako problemsko področje definirali s formalno zapisanim skupkom modelov, ki bi natančno in nedvoumno opisovali vse vidike obravnavane domene? Problem, ki bi ga predstavili v obliki množice modelov, bi obravnavali v takšnem modelirnem jeziku, ki ga ne bi razumeli ali znali uporabljati le računalniški eksperti, pač pa tudi eksperti obravnavane domene. Govorili bi torej v domačem jeziku strokovnjakov domene, ne pa v nekem tujem jeziku, ki so si ga izmislili strokovnjaki drugega tehničnega in organizacijskega področja z namenom, da bi na razumljiv, vendar tuj način, obravnavali probleme iz njihove domene. Iz tako oblikovanih modelov domačega jezika bi potem postopoma, s formalno zapisanimi pravili, začetne modele preoblikovali v množico nižje abstraktnih modelov, podobno kot v prvem primeru - razumljive drugemu spektru poznavalcev, in jih s tem počasi preoblikovali v končni izvedbeni izdelek. Modele in njihove modelirne jezike bi s takšnim načinom razmišljanja postavili v ospredje razvoja programskih rešitev in tako resnično govorili o PRAVI revoluciji!

V nadaljevanju bomo predstavili paradigmo Model Driven Development, ki stremi k izboljšanju celotnega razvojnega cikla programske opreme, z zagotavljanjem bolj abstraktnega modelirnega načrtovanja in izdelave kompleksnih računalniških sistemov, s čimer bi odpravili slabosti programskih jezikov 3. Generacije [5].

2. MODELNO VODEN RAZVOJ

Zadnjih nekaj let je sedaj že obširnemu krogu snovalcev programskih rešitev dobro poznana paradigma, imenovana Model Driven Development (v nadaljevanju MDD ali razvoj po MDD) [2], [6] in [14]. Nekatere organizacije oz. posamezni avtorji jo imenujejo tudi Model Driven Engineering (MDE) [1] oz. pogosteje, Model Driven Software Development (MDSD) [3], [4], [7] in [8]. V okviru magistrske naloge bomo uporabljali prvi izraz, saj kljub temu, da je izjemno preprost, jasno izraža bistven namen paradigme – govori o modelno vodenem razvoju. Kljub temu, da je iz imena izpuščena besedna zveza »programska oprema« (ang. software), bi moralo biti jasno, da je govora o modelno vodenem razvoju programske opreme, programov oz. najsplošneje, programskih rešitev.

Kot smo zapisali že v uvodnem poglavju, razvoj programske opreme na osnovi modelov ni novo področje. Do danes so se tako v industriji kot tudi na različnih raziskovalnih področjih pojavljale številne načrtovalske-razvojne modelirne tehnike različnih notacij, pristopov in »revolucionarnih« rešitev. Med njimi se je zagotovo najmogočneje uveljavil modelirni jezik UML (Unified Modeling Language), ki je pod okriljem konzoricija Object Management Group [12] splošno priznan standard opredeljevanja (podajanja), vizualizacije, načrtovanja in dokumentacije poslovnih ter drugih računalniških sistemov. Podporo jeziku UML so v zadnjem desetletju priznale številne velike organizacije in združenja (IBM, Sybase Inc., NEC Corporation, Borland Software idr.), ki z vedno večjim tempom postopoma spoznavajo in resnično začenjajo verjeti v uspeh modelno vodenih tehnik razvoja programske opreme. Njihova prizadevanja lahko upravičimo z naborom številnih razvojnih orodjih različnih proizvajalcev [13], s pomočjo katerih je bilo v preteklosti zaključenih nešteto uspešnih projektov [25]. Ni težko ugotoviti, da je sprejetost standarda UML, in s tem sprejetosti njegovih modelirnih tehnik, pri tako velikem številu orodij, vse prej kot zanemarljiva.

Kljub dejstvu, da so (UML) modelirne tehnike v veliki večini prisotne v začetnih fazah oz. aktivnostih razvojnih projektov – takrat je abstraktnost obravnavanja problemskega področja najvišja in z modeliranjem lažje predstavljiva človeku, so izdelani modeli le redko postavljeni v ospredje razvojnih aktivnosti tako kot iz njih posredno ali neposredno izpeljana programska koda. V večini primerov se modelirno gradivo obravnava le kot dobro dokumentacijo sistema, ki jo razvijalci vzdržujejo in z njo upravljajo le na začetku, potem pa na njo bolj ali manj pozabijo ali je ne dopolnjujejo skladno s kasnejšimi funkcionalnimi spremembami. Kljub vsemu je očitna relacija med začetnim modelom in končno implementacijsko kodo, v takšnem primeru zgolj namerna, ne pa tudi formalna - kar pomeni, da od nekega trenutka naprej ne moremo več govoriti o enostavni ali celo enolični preslikavi modela v programsko kodo, kaj šele obratno. Prav tako je takšna neformalno zasnovana preslikava bolj ali manj odvisna od interpretacije posameznega razvijalca, ki na svoj in edinstven način implementira višje abstraktni model v nižje abstraktno programsko kodo. Posledično se vse večjo težo pripisuje programski kodi, ki je bližje končni rešitvi, modele pa se prisilno dopolnjuje z namenom, da

se vsaj do neke mere ujemajo s kodo in se lahko z njimi pohvalimo pred naročnikom. Takšen način dela od razvijalcev zahteva ogromno dodatnega dela, od katerega v večini primerov nimajo dejanske koristi. Zato se ni čuditi dejstvu, da je med razvijalci ogromno nasprotnikov razvoja, osnovanega na modelih, saj v njegovem bistvu ne vidijo drugega, kot pomoč pri zagonu projekta ali pa le atraktivno izdelano projektno dokumentacijo.

Obliko modelno vodenega razvoja, ki smo jo ravnokar opisali, bi lahko bolje označili kot modelno osnovan razvoj (ang. Model-Based Development), saj:

- Je model le orodje, s katerim si razvijalci »pripravijo teren« za izdelavo končne programske rešitve,
- se modeliranje uporablja zgolj zato, da se na enostaven način (predvsem zaradi boljše komunikacije z akterji projekta) zajame funkcionalne zahteve, opiše različne tokove dogodkov oz. postopkov, prikaže arhitekturno zasnovo sistema v razvoju ipd., dejanskega vpliva pa modeli na končno rešitev nimajo,
- je koncept modela manjvreden od koncepta obravnavane programske kode, saj ni določene formalne zasnove, s pomočjo katere bi bilo mogoče model v vsakem trenutku neposredno ali po korakih pretvoriti v programsko kodo,
- se modeliranje obravnava zgolj kot orodje za izdelavo dokumentacije.

Če bi želeli govoriti o pravem MDD razvoju, takšnem, ki razvoj povzdigne na višji in obljubljeno bolj učinkovit abstraktni nivo, moramo znati modele postaviti v ospredje razvojnega cikla. Pri tem moramo znati upoštevati vsaj naslednje osnovne zahteve, ki smo jih povzeli iz najrazličnejših virov dobrih praks modelno vodenega razvoja:

- Modeli določajo in opredeljujejo bistvo programsko-razvojnega projekta,
- so najpomembnejši sestavni del (ang. first-class citizens) razvojnega procesa, ki jih na vseh stopnjah razvoja obravnavamo enakopravno programski kodi,
- so formalno osnovani, kar pomeni, da lahko s pomočjo formalno definiranih pravil preverjamo pravilnost modela v izgradnji,
- s postopki tranformacije, avtomatizacije in standardizacije jih moramo znati enolično (pre)oblikovati iz prvotne (najbolj abstraktne) oblike v končno implementacijsko obliko - programsko kodo,
- z upoštevanjem prvih treh zahtev moramo povečati produktivnost razvijalcev, poenostaviti razvojni proces in izdelati tako dobro programsko rešitev, kot bi jo izdelali sicer, drugače MDD pristop, ki ga uporabljamo, ni smotrn. [2] opisano zahtevo potrjuje z izjavo: *»Still, with all the benefits of automation and standardization, model-driven methods are only as good as the models they help us construct.«*

Da bi dosegli navedene osnovne zahteve, MDD predpisuje različne koncepte, tehnike, načela, priporočila in najboljšo prakso (v nadaljevanju poimenovani elementi MDD), s pomočjo

katerih modelno osnovan razvoj nadgradimo v modelno voden razvoj. Za začetek bomo na kratko opisali le nekaj izmed najpomembnejših elementov, v kasnejših poglavjih pa bomo znanje o MDD nekoliko poglobili. Poleg vsakega izmed pomembnejših elementov MDD bomo navedli tudi referenco na njegov kasnejši, bolj podroben opis, da bomo v primeru nerazumevanja lažje in hitreje našli morebitno obrazložitev.

2.1 OSNOVNI POJMI MDD

V najbolj splošnem lahko rečemo, da je MDD razvojni pristop oz. paradigma, ki v ospredje celotnega življenjskega razvojnega cikla postavlja modelirne (največkrat grafično predstavljene) elemente, združene v okvir enega samega ali več modelov. Z razliko od ideologije prvotnih in morda celo večine trenutnih CASE orodij, ki se opirajo na modeliranje vsakršne domene z univerzalnim modelirnim jezikom (kot je npr. UML), MDD poudarja potrebo po uvedbi formalno osnovanega modelirnega jezika, imenovanega domensko specifični jezik (ang. Domain Specific Language) [podpoglavje *2.3 – Metamodeliranje in domensko specifični jeziki*]. Z jezikom DSL lahko natančno modeliramo le tisto problemsko področje (domeno), za katero obstaja formalni zapis ali zasnova, s katerim smo ta jezik definirali.

Če navedemo jasen primer, bi za potrebe vzpostavitve preprostega računalniškega omrežja lahko oblikovali specifičen modelirni jezik, v okviru katerega bi povezanost med različnimi komunikacijskimi napravami predstavili z grafičnimi simboli strežnika, delovne postaje, usmernika, mrežnega tiskalnika, mobilne naprave ipd. Za vsako izmed komunikacijskih naprav bi bilo formalno predpisano kako, s katerimi pred-pogoji, po-pogoji, posebnimi zahtevami, alternativami in kakšnimi »posledicami« jo lahko priključimo v omrežje. Delovne postaje z vključeno možnostjo DHCP (Dynamic Host Configuration Protocol) npr. ne bi mogli priključiti v omrežje, če bi imel usmernik omrežja nastavljeno možnost DHCP server = disabled, mrežni tiskalnik bi v omrežje lahko konfigurirali tako, da bi ga lahko uporabljale le naprave s točno predpisanih IP naslovov ipd.

Vzpostavitev in konfiguracijo opisanega problemskega področja bi lahko modelirali tudi z razširjeno oz. za to domeno posebej prilagojeno različico univerzalnega modelirnega jezika UML (npr. na način, kot to predpisujejo t.i. UML profili [16]), vendar bi pri tem skrajno ne-intuitivno (gledano s vidika eksperta domene) izrazili tisto, kar je resnično pomembno. Bolj bi se ukvarjali z vprašanjem KAKO to izraziti v modelu (ang. solution space), kot pa KAJ izraziti v modelu (ang. problem space).

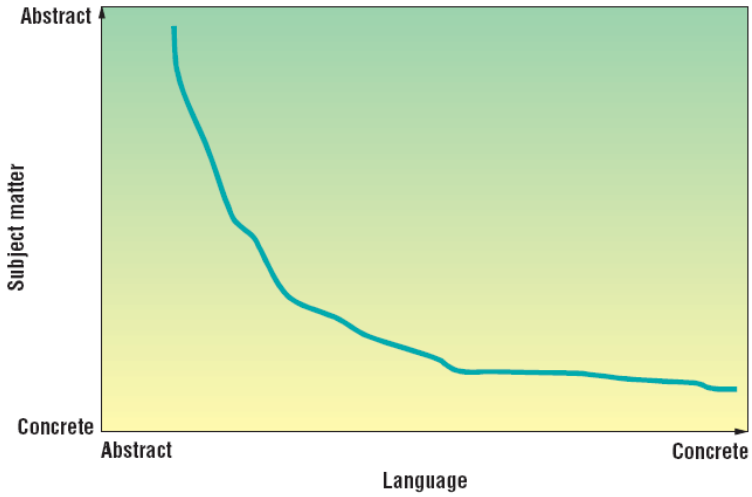
Če z domensko specifičnim jezikom modeliramo le tisto problemsko področje, za katero obstaja formalni zapis ali zasnova, s katerim smo ta jezik definirali, se je seveda potrebno dotakniti tudi pomena formalne definicije jezika. Metamodeliranje [podpoglavje *2.3 – Metamodeliranje in domensko specifični jeziki*] je eden izmed temeljnih konceptov razvoja po MDD, s katerim natančno opišemo ključne elemente obravnavane domene. Metamodel je

formalni načrt domene, v okviru katerega so zapisane lastnosti, omejitve in relacije elementov domene. Domensko specifični jezik, s katerim modeliramo konkretne primerke ali instance obravnavanega problemskega področja, mora upoštevati načrt domene – njegov metamodel. Le tako bodo oblikovani modeli pravilni v smislu domene, s čimer bo preprečeno oz. onemogočeno napačno razumevanje konkretnega problema na osnovi interpretacije modela. Če povemo drugače: s pomočjo izdelanih modelov bomo lahko nedvoumno in natančno opisali neko dejansko stanje iz obravnavanega problemskega področja.

Bistvo MDD ni v tem, da koncepte, abstrakcije in pravila iz realnega sveta (v obliki modelov) le zajamemo in opišemo z visoko-abstraktnim modelirnim jezikom. Oblikovani modeli nam sami po sebi praktično ne koristijo. Smotrno jih moramo znati preoblikovati v konkretno in uporabno programsko rešitev, ki nam bo poenostavila delo ter pomagala pri reševanju konkretnih problemov. Zato moramo modele s pomočjo formalno definiranih transformacij [podpoglavje *2.4 – Transformacije med modeli*] in postopki avtomatskega generiranja programske kode [podpoglavje *2.5 – Združevanje avtomatsko generirane in ročno napisane programske kode*] preoblikovati v takšno obliko (npr. datoteke DLL, Java bytecode, batch datoteke...), ki jo bo razumela izbrana implementacijska tehnologija oz. platforma. Na višjem abstraktnem nivoju kot platforma je, manj bomo imeli dela pri sami definiciji posameznih transformacij. Bolj kot ima bogato in raznoliko vsebino, višja bo produktivnost razvijalca ali programerja, ker se bo pri razvoju lahko opiral na že realizirane podporne storitve, ki si jih lahko predstavljamo kot črne škatle. Ni se nam treba ukvarjati z razumevanjem njihovega delovanja, znati jih moramo le uporabljati (kot npr. daljinec za televizijo). Bistveno je, da izberemo takšno platformo, v okviru katere bomo najbolj učinkovito in najhitreje poskrbeli za rešitev problema obravnavane domene. Najbrž ne bi bilo smiselno ročno spisati postopkov za krmiljenje tiskalnika ali neke druge komponente računalniškega sistema v strojnem jeziku, pač pa bi za to uporabili knjižnice višje-nivojskega jezika (npr. jezika C++), ki bi za abstraktno preslikavo ukazov v nižje-nivojske programske jezike »poskrbel sam«. Naj na tem mestu še enkrat poudarimo dejstvo, da je abstraktni nivo implementacijske tehnologije, ki razume končno obliko modela, popolnoma poljuben: lahko govorimo o specifičnih izvajalnih ogrodjih, lupinah operacijskih sistemov, interpreterjih, prevajalnikih, knjižnicah in paketih različnih programskih jezikov, storitvah spletnih strežnikov itn. Ne sme nas zavesti beseda »platforma«, ki se v praksi nanaša bodisi na enega izmed operacijskih sistemov (Mac OS, Linux, Windows), standarde kot sta npr. CORBA (Common Object Request Broker Architecture) in SOA (Service Oriented Architecture) ali pa najpogosteje na mogočna razvojno-izvajalna okolja (ogrodja), v katerih družino spadata že omenjena .NET in J2EE. Meni osebno najljubša definicija plaforme, ki bi jo lahko še najbolje umestil v kontekst razvoja po MDD, je: »*In general, platform has a generic meaning: anything on which one engages in some activity, with the goal of leveraging the work of others*« [15].

Povezanost med ravnjo abstrakcije obravnavanega problemskega področja in ravnjo abstrakcije modelirnega jezika lahko prikažemo s sliko 1. Bolj abstraktno kot z modeli

izrazimo določeno domeno (na sliki *Subject Matter*), bližje smo z modelirnim jezikom (na sliki *Language*) dejanskemu jeziku, ki ga razume ekspert domene. Bolj »konkreten« kot je ta modelirni opis, bližje smo dejanski implementacijski tehnologiji (recimo programskemu jeziku), ki je ekspertu domene tuja ali celo neznana. Kot primer lahko navedemo razvoj bančnega informacijskega sistema, katerega statični in dinamični vidik delovanja na začetku modeliramo z višje-abstraktnim modelirnim jezikom (kot so npr. različne diagramske tehnike jezika UML), da z naročnikom projekta in ključnimi uporabniki dosežemo skupen nivo razumevanja problema, potem pa ustrezno realizacijo delovanja sistema uresničimo s konkretnimi stavki programskega jezika (npr. v programskem jeziku C), ki ga razume izbrano razvojno okolje [6].



SLIKA 1: STOPNJA ABSTRAKCIJE DOMENE IN MODELIRNEGA JEZIKA

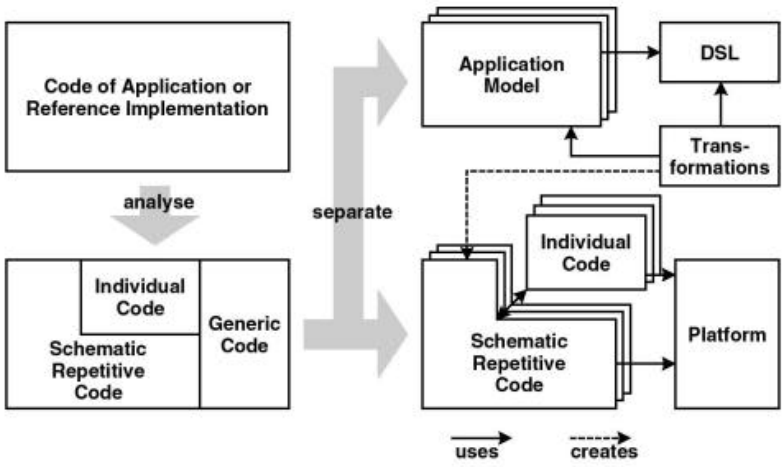
Nasprotniki razvoja po MDD so prepričani, da je programska koda, ki jo sami napišejo v enem izmed programskih jezikov (npr. C#, Java ipd.), mnogo boljša od tiste, ki jo je moč pridobiti na osnovi avtomatskega generiranja kode iz oblikovanih višje-abstraktnih modelov. Prav tako zatrjujejo, da so pri neposrednem kodiranju bolj učinkoviti, neobremenjeni z razvojnimi orodji in modelirnimi okolji ter, kar je najpomembneje, da dejansko poznajo in jasno razumejo svojo programsko kodo. Zagovorniki MDD jih pri takšnem »egoističnem« obnašanju istovetijo s tistimi, ki nekoč niso verjeli v začetek obdobja prevajalnikov. Avtor članka [2, str. 19-25] je mnenja, da je tehnologija modeliranja že dozorela in lahko v vseh pogledih ključno pripomore k razvoju programske opreme. Prav tako ugotavlja, da je z vedno večjim porastom različnih programskih področij večino programske kode mogoče generirati neposredno iz oblikovanih modelov. Bistvena dejavnika, ki bosta v bližnji prihodnosti verjetno še najbolj pripomogla k temu, sta standardizacija modelirnih tehnologij, razvojnih metod ter modelirnih orodij (okolij) in zrelost vse bolj raznovrstnih, močnih in visoko abstraktnih tehnoloških platform, ki bodo razvijalce-programerje ščitile pred odvečnim poznavanjem »vsega«.

Na tem mestu najbrž ni odveč izpostaviti dejstva, da razvoj po MDD ne pomeni le avtomatsko generiranje programske kode iz oblikovanih modelov, brez kakršnekoli ročne oz. »človeške intervencije«. Ravno nasprotno, avtorji [7], [4, str. 40-42], [8] predpisujejo različne načine oz. tehnike združevanja generirane in ročno napisane programske kode. Popolnoma nesmiselno je ročno programirati določene splošne ali generične dele končne programske rešitve, če lahko namesto tega uporabimo že izdelane programske module izbrane platforme (npr. razne skripte lupine OS, knjižnice za dostop do podatkovnih baz, spletnih storitev in ERP okolij, razrede za delo z grafičnim okoljem ipd.). Po drugi strani je nesmotrno porabljati časovna, finančna ali človeška sredstva za izdelavo prav vseh (ne)mogočih transformacijskih pravil, samo zato, da bomo izdelane modele »na silo« samodejno preoblikovali v programsko kodo. Nekatere specifične dele lahko dosti bolj elegantno, brez večjih stroškov in z relativno majhno verjetnostjo (predvsem človeških) napak neposredno sprogramiramo ročno (npr. v enem izmed programskih jezikov, ki jih obvladamo), potem pa jih s predlaganimi postopki združevanja »zlepimo« skupaj z generirano programsko kodo.

Razvoj po MDD tudi ne pomeni oblikovanje novih programskih rešitev vedno znova - od začetka (ang. from scratch). Spodbujena, če ne celo nujna, je (upo)raba že izdelanih projektov, programskih rešitev ali pa le nekaterih elementarnih funkcij, ki so skupni oz. enotni znotraj določene »družine« aplikacij. [7] to pojasnjuje s sliko 2: tipično programsko rešitev v razvoju si lahko predstavljamo kot skupek treh vrst programske kode:

- generične programske kode (na sliki *Generic Code*), ki je enotna (skupna) vsem programskim rešitvam (določene družine aplikacij),
- vzorčno ponavljajoče programske kode (na sliki *Schematic Repetitive Code*), ki jo med različnimi programskimi rešitvami med seboj lahko primerjamo po enakem vzorcu »obnašanja« (npr. uporabi istih arhitekturnih stilov ipd.) ter
- individualne programske kode (na sliki *Individual Code*), ki je specifična za točno določeno programsko rešitev in je ni mogoče posploševati.

Modelno voden razvoj stremi k sestavljanju (generiranju) vzorčno ponavljajoče programske kode, izključno iz oblikovanih modelov (na sliki *Application Model*) modeliranega problemskega področja. Ključni elementi, ki k temu pripomorejo, so domensko specifični jezik, transformacije med modeli in izbrana tehnološka platforma, ki jih je potrebno vsakič znova izoblikovati (določiti) za vsako domensko področje posebej. Kljub navideznemu kompleksnemu ločevanju navedenih tipov programske kode je takšen pristop mnogo boljši in celo enostavnejši, kot če bi vedno znova vse morali narediti od začetka: za vsako aplikacijo je potrebno posebej izoblikovati le njen specifični (manjšinski) del, medtem ko večinski del - tehnološko platformo in vzorčno ponavljajočo programsko kodo pridobimo na podlagi obstoječih rešitev in že izdelanih transformacij domensko specifičnih modelov.



SLIKA 2: APLIKACIJA KOT KONGLOMERAT RAZLIČNIH TIPOV PROGRAMSKE KODE

Preden preidemo na podroben opis do sedaj omenjenih elementov MDD, bomo poizkušali povzeti glavne prednosti modelno vodenega razvoja:

- S pomočjo razvoja po MDD je mogoče povečati učinkovitost dela razvijalcev. Temu se približamo predvsem z upoštevanjem načela avtomatizacije, v okviru katerega formalno zapisane modele s postopki transformacij in avtomatskega generiranja nižje-nivojskih programskih konstruktov enostavno, hitro in enolično preoblikujemo v implementacijsko programsko kodo.
- Zaradi (upo)rabe transformacij in formalno zapisanih modelirnih jezikov pridobimo na kakovosti izdelanih programskih rešitev. Avtomatsko generirana programska koda je neodvisna od sprotnih napak razvijalca in odraža dejansko sliko preverjenih ter vnaprej definiranih arhitekturnih vzorcev, določenih z modelirnim jezikom in elementrano oblikovanimi postopki transformacij.
- Višje abstraktni modelirni jeziki zagotavljajo bolj intenzivno »programiranje« (oz. uporabniško izkušnjo na višjem nivoju), kot je to mogoče doseči z npr. objektno-orientiranimi programski jeziki, na katerih temelji večina trenutnih razvojnih projektov. Zaradi razumevanja problemskega področja na višje-abstraktni ravni, ki jo je mogoče povsem prilagoditi obravnavani domeni, je lažje opisati kompleksne funkcionalne zahteve, upoštevati omejitve, obvladovati nižje-nivojsko implementacijsko tehnologijo ter omogočiti lažje iskanje in odpravljanje (človeških) napak razvijalcev.
- »Sveti gral« razvoja po MDD naj bi bil ravno v tem, da pri ustvarjanju programskih rešitev ne sodelujejo le računalniški eksperti, pač pa tudi strokovnjaki obravnavane domene, kot to v svojem članku pojasnjuje [14]. Z modelirnim jezikom opisano konkretno problemsko področje lahko eksperti razumejo prav zaradi tega, ker je modelirni jezik mogoče formalno osnovati in predstaviti v obliki, ki je podobna ali celo enaka njihovem ekspertnemu jeziku.

- Ko z modelirnim jezikom in elementarno oblikovanimi postopki transformacij določimo konkretne arhitekturne vzorce, jih lahko vsakič znova uporabimo pri oblikovanju posameznih družin aplikacij. Vsaka družina je sicer usmerjena v reševanje samosvojih (specifičnih) problemov, vendar za reševanje le-teh lahko uporablja skupne tehnološke oz. funkcijske gradnike. Koncept ponovne uporabe že izdelanih in dobro preizkušenih elementov je sicer že poznan iz sveta objektno usmerjenih tehnologij, vendar jih lahko v primeru razvoja po MDD uporabimo še bolj raznovrstno in na različnih (abstraktnih) nivojih.
- Z ločevanjem platform za razvoj aplikacij in iz njih oblikovanih konkretnih rešitev, zagotovimo uporabo standardnih funkcij, postopkov in mehanizmov obravnavanih platform in omogočimo razvoj dveh popolnoma ločenih razvojnih okolij.
- Z upoštevanjem načela standardizacije, v okviru katerega omogočimo prenosljivost delnih ali dokončnih programskih rešitev med orodji in različnimi platformami (ang. interoperability), zagotovimo široko programsko podporo različnih proizvajalcev, splošno sprejetost obravnavanih konceptov in povečamo možnost dolgoročne podpore različnih brezplačnih spletnih skupnosti in kakovostnih servisnih služb.

2.2 RAZUMEVANJE DOMENSKEGA PODROČJA

Če želimo modelirati neko problemsko področje, moramo znati abstraktne elemente domene opisati s pomočjo konkretnih gradnikov oz. elementov izbranega modelirnega jezika. Preslikavo iz konkretnega domenskega področja v formalno modelirno obliko jezika lahko imenujemo kar (modelirna) paradigma izbrane domene [10, točka 2].

Paradigma mora vsebovati vse bistvene sintaktične, semantične in predstavitvene informacije neke domene. Natančno moramo vedeti, katere koncepte domene bomo izbrali za izhodišče gradnje modelov, kakšne relacije obstajajo med temi koncepti, kako jih bomo združevali in predstavili uporabnikom-arhitektom iz različnih perspektiv, na kakšne načine (in ali sploh) jih bo mogoče razčleniti na manjše in bolj obvladljive enote, katere omejitve bomo pri tem upoštevali itn. Vsaka paradigma, ko je enkrat dobro definirana in upošteva vse želene zakonitosti obravnavane domene, lahko opiše poljubno mnogo konkretnih primerkov iz te domene. Če bi imeli npr. natančno definirano paradigmo dinamike letalskega prometa v nekem kontroliranem zračnem prostoru, pri čimer bi morali upoštevali vse možne kombinacije predvidenih in nepredvidenih dogodkov, do katerih lahko pride, bi za vsak dan v tednu, za vsako sekundo obratovanja v zraku in na tleh, lahko imeli nek vnaprej definiran posnetek stanja. Lahko si le predstavljamo, da bi na ta način lahko preprečili praktično vsako letalsko nesrečo. Žal pa je za konkreten primer to nemogoče izvesti. Paradigma bi enostavno vsebovala preveč konceptov in medsebojnih povezav, da bi jih na kakršenkoli mogoč način lahko predstavili v obvladljivi obliki. Tako skrajen primer smo navedli zgolj zato, da se bomo zavedali pomembnosti zajema vseh potrebnih informacij domene in dejstva, da se v okviru formiranja paradigme lahko osredotočimo le na konkretne dele domene. V primeru paradigme

letalskega prometa si lahko zelo jasno predstavljamo oblikovanje dela paradigme, kjer bi formalno opisali (modelirali) le zasedenost VFR (ang. Visual Flight Rules) točk prihodov in odhodov letal v kontroliran zračni prostor. Z vidika kompleksnosti opisa paradigme je to popolnoma obvladljiv nabor množice vnaprej definiranih stanj – vstopnih točk v kontroliran zračni prostor (Letališče Jožeta Pučnika ima npr. 3 VFR vstopne točke).

Pri oblikovanju paradigme izbrane domene morajo sodelovati tako eksperti domene, ki natančno poznajo obravnavano problematiko, kot tudi izkušeni načrtovalci modelov, ki so njihovo znanje sposobni predstaviti s pomočjo elementov izbranih modelirnih jezikov. Ni dovolj, da iz domenskega področja le zajamemo vse potrebne informacije. Znati jih moramo tudi učinkovito predstaviti oz. modelirati. Poleg tega se moramo obojega lotiti postopoma. Najprej je potrebno zajeti najbolj pomembne značilnosti in omejitve domene, potem pa še vse izjeme. Lahko pričakujemo iterativno-inkrementalen proces razvoja, v katerem se oblikovanje paradigme hitro spreminja v zgodnjih fazah razvoja in se stabilizira šele po nekaj ciklih testiranja ter dejanskem praktičnem modeliranju. Največji dejavnik, ki pripomore k temu pojavu je dejstvo, da eksperti domene na začetku niso sposobni ali pa ne znajo definirati, kako naj bi izgledalo modelirano okolje. Včasih je že težko opisati problem, kaj šele ga definirati in si ga jasno predstavljati z zahtevanimi abstraktnimi elementi modelirnega jezika. Sčasoma, ko se iz izgrajenih modelov začnejo jasno »kristalizirati slike iz realnega sveta«, tudi paradigma postane bolj stabilna. Razumljivo je, da bo vsaka različica paradigme zahtevala popolnoma osvežen nabor pripadajočih modelov in s tem tudi nov vidik modeliranja oz. nekoliko drugačne predstavitve problemske domene.

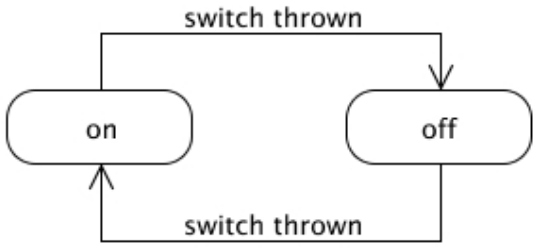
2.3 METAMODELIRANJE IN DOMENSKO SPECIFIČNI JEZIKI

Kot smo povedali že v prejšnjem poglavju, je vsako problemsko področje unikaten primerek svojevrstnih abstrakcij, različnih atomarnih oz. sestavljenih konceptov ter poslovnih pravil, ki omejujejo in določajo načine obnašanja med njimi. Če želimo problemsko področje modelirati z modelirnim jezikom, mora jezik znati implementirati omenjene abstrakcije, predstaviti posamezne koncepte statičnega in dinamičnega obnašanja domene ter upoštevati vse zakonitosti definiranih poslovnih pravil. To z drugo besedo pomeni, da morajo biti pri gradnji modela upoštevane prav vse posebnosti, lastnosti in omejitve, ki jih obravnavano problemsko področje zahteva.

Posamezne koncepte domene bi lahko zelo enostavno modelirali s stereotipnimi razredi modelirnega jezika UML. Pri tem lahko takoj podvomimo v smiselnost uporabe jezika v vseh mogočih (in nemogočih) primerih modeliranja, saj z uporabo diagramske tehnike razrednih diagramov ne moremo preprečiti asociacij med le nekaterimi stereotipnimi razredi. UML v sami osnovi tega ne omogoča. Torej bi v takšnem primeru morali oblikovati popolnoma nov DSL modelirni jezik (lahko tudi na osnovi razširjenega jezika UML), ki bi te omejitve upošteval.

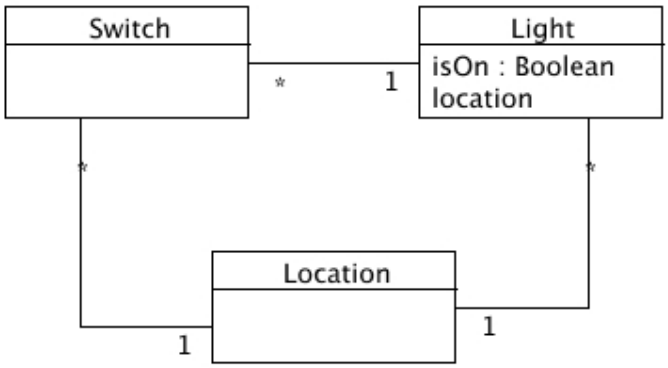
Zahtevam in omejitvam DSL modelirnega jezika se približamo z metamodeliranjem. Metamodel si lahko predstavljamo kot model, ki opisuje vse iz njega izpeljane modele. Iz tega sledi, da vsak DSL modelirni jezik vedno sledi pravilom in načelom, ki jih določa njegov metamodel. Pri modeliranju posebnih izjem, ki bi jih z oblikovanim metamodelom zelo težko izrazili (predvsem zaradi omejitev razvojnih okolij in samih predstavitvenih tehnik modela človeku), si lahko pomagamo s pomočjo omejitvenih jezikov, kot je npr. OCL (Object Constraint Language). S pomočjo takšnih pristopov dosežemo jasno razumljiv in nedvoumno definiran metamodel, ki strogo določa pravila oblikovanja oz. modeliranja vseh iz njega izpeljanih modelov, poleg tega pa za njegovo razumevanje ne potrebujemo veliko napora.

Povezavo med koncepti metamodela, modela in DSL modelirnega jezika bomo nazorno predstavili s spodnjim primerom [18]. Zamislimo si preprost sistem luči, v katerem lahko luč prižigamo in ugašamo z enim ali več stikali. Slika 4 prikazuje UML diagram stanj, s katerim modeliramo stanje luči. Možni sta dve različni stanji: prižgana luč in ugasnjena luč, prehod med obema stanjema pa dosežemo s preklopom stikala. Diagram stanj torej prikazuje način delovanja luči v odvisnosti med stikali. Če stikalo preklopimo, se luč prižge, če ga ponovno preklopimo, pa luč ugasne. Model si lahko predstavljamo kot preprost program, ki opisuje sistem luči. Preklapljanje med stikali in seveda posledično prižiganje in ugašanje luči, pa lahko razumemo kot dejansko izvedbo tega programa.



SLIKA 3: UML DIAGRAM STANJ SISTEMA LUČI

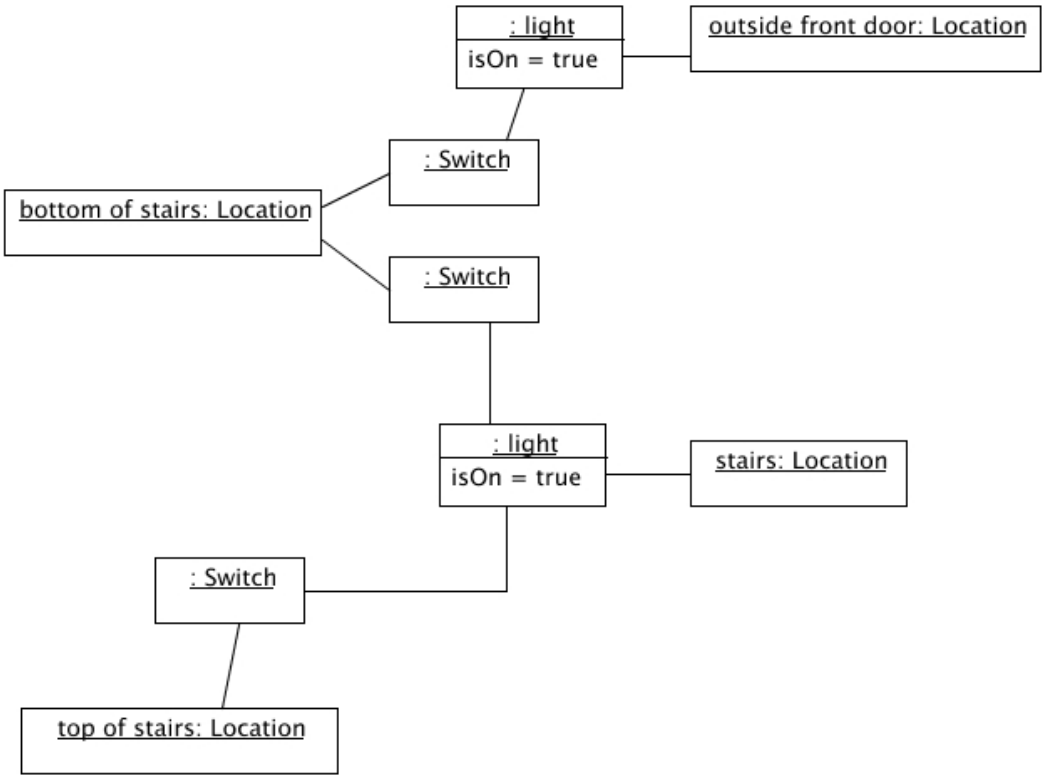
Z razrednim diagramom UML na sliki 4 lahko prikažemo statični vidik sistema. Razred *Switch* predstavlja koncept stikala, razred *Light* predstavlja koncept luči in je opisan z atributom *isOn* logičnega podatkovnega tipa, ki pove, ali je v nekem trenutku luč prižgana ali ugasnjena. Oba razreda med seboj združuje še razred *Location*, ki predstavlja nek točno določen prostor ali lokacijo znotraj hiše. Iz diagrama je jasno razvidno, da se v vsakem prostoru lahko nahaja več stikal in več različnih luči. Vsako stikalo prižge ali ugasne natanko eno luč, vsaka luč pa ima poljubno mnogo stikal.



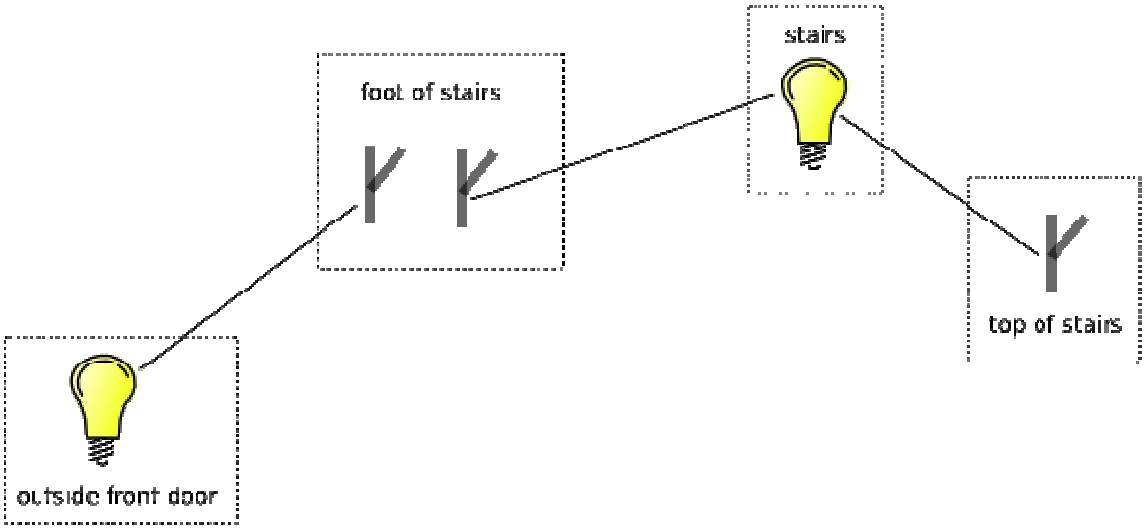
SLIKA 4: UML RAZREDNI DIAGRAM SISTEMA LUČI

Navedena modela opisujeta tako statični kot tudi dinamični vidik omejitev in zakonitosti obnašanja primerkov konkretne problemske domene. Z upoštevanjem pravil medsebojnega povezovanja stikal, luči in prostorov na sliki 5 opišemo preprost sistem luči, v katerem luč na stopnišču prižgeta natanko dve stikali: eno se nahaja ob vznožju, drugo pa na vrhu stopnic. Prav tako stikalo ob vznožju stopnic prižge luč pred vhodnimi vrati. Iz modela je še razvidno, da sta obe luči trenutno prižgani, vsako luč pa lahko z enim izmed pripadajočih stikal kvečjemu ugasnemo.

Opisani primerek sistema luči smo modelirali z diagramsko tehniko modelirnega jezika UML. To pomeni, da smo se pri izgradnji modela oprli na notacijo izbranega jezika. Model bi z vsemi zahtevanimi pravili povezovanja lahko realizirali tudi kako drugače. Na sliki 6 smo uporabili lastni DSL modelirni jezik, ki namesto UML razredov in povezav med njimi opisano problemsko področje modelira z grafičnimi simboli luči, zaporednim izrisom stikal in zapisom lokacije ob simbolu. S takšnim modelom lahko podani primer opišemo na popolnoma enak način, vendar dosti bolj razumljivo človeku.



SLIKA 5: PRIKAZ KONKRETNEGA PRIMERKA SISTEMA LUČI, OPISANEGA Z UML OBJEKTNIM DIAGRAMOM

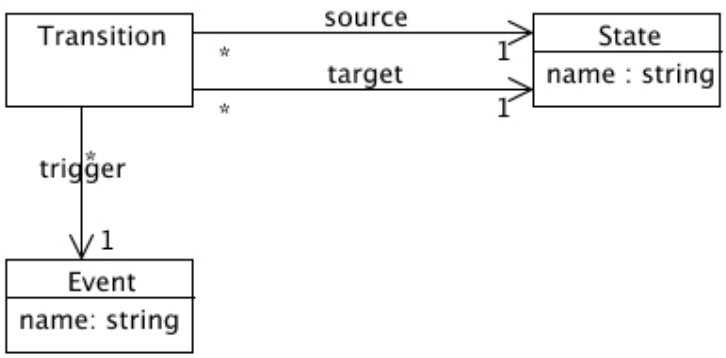


SLIKA 6: PRIKAZ KONKRETNEGA PRIMERKA SISTEMA LUČI, OPISANEGA Z LASTNIM DSL MODELIRNIM JEZIKOM

Z obema primerkoma modelov smo torej enake infomacije prikazali na dva različna načina. To je eden izmed ključnih konceptov razvoja po MDD. Vsak primerek modela lahko

realiziramo z različnimi metodami, pri čemer se lahko opremo na raznovrstne diagramske tehnike, razpredelnice, grafe, matematične sintakse, izpise v tekstualni obliki itn. Bistveno je, da za vsako domensko področje izberemo ali oblikujemo tak DSL modelirni jezik, s katerim bomo najlažje opisali konkretne probleme. Ne glede na izbrani modelirni jezik pa mora le-ta natančno slediti definiranim pravilom. Oba modela na slikah 5 in 6 ustrezata statičnemu vidiku sistema, ki smo ga opisali z UML razrednim diagramom sistema luči. Če povemo drugače, model sistema luči natančno definira oba konkretna primerka modelov. Torej predstavlja neke vrste model modelov ali lepše povedano - model vseh iz njega izpeljanih modelov. Kot smo že povedali v uvodu tega poglavja se je v terminologiji MDD za takšno poimenovanje uveljavil izraz meta-model.

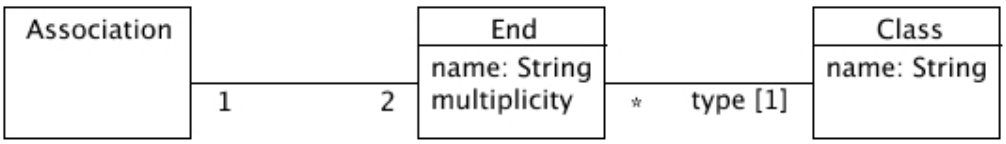
Če gremo še korak dlje, lahko tudi navedeni diagram stanj sistema luči opišemo z nekim metamodelom. Na tem mestu si velja zapomniti zlato pravilo razvoja po MDD: kadarkoli oblikujemo nek model, moramo zanj prej definirati njegov metamodel, ki nam pove, kako lahko model sploh zgradimo. Spodnja slika prikazuje metamodel za izgradnjo diagrama stanj sistema luči, ki smo ga opisali z modelom na sliki 3.



SLIKA 7: METAMODEL DIAGRAMA STANJ SISTEMA LUČI

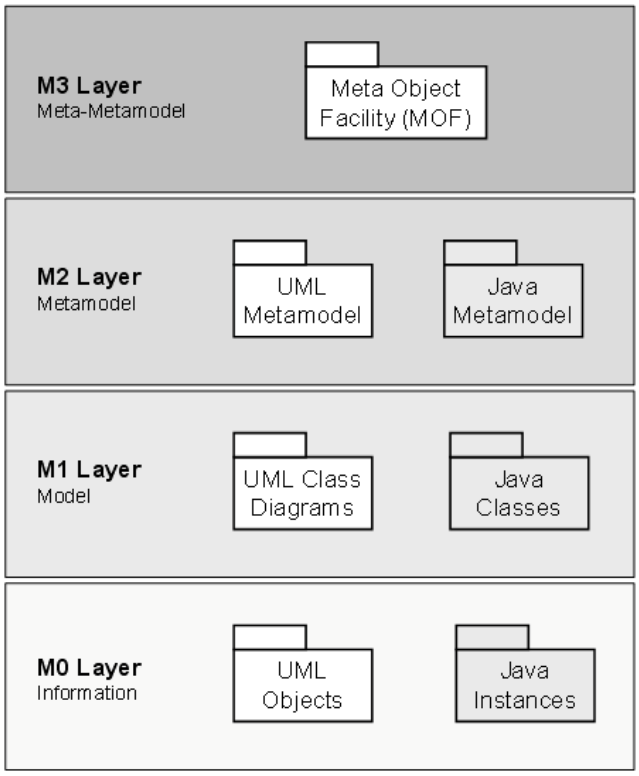
Za zgornji metamodel podajmo sliko 8, na kateri je prikazana poenostavljena različica njegovega meta-metamodela.

Ne moremo mimo dejstva, da takšen postopek lahko nadaljujemo v nedogled. V praksi načeloma ni tako. Na najnižjem nivoju opisani meta-meta-meta…modeli navadno opisujejo tudi samega sebe, tako da lahko postopek iskanja metamodelov v tistem trenutku zaključimo.



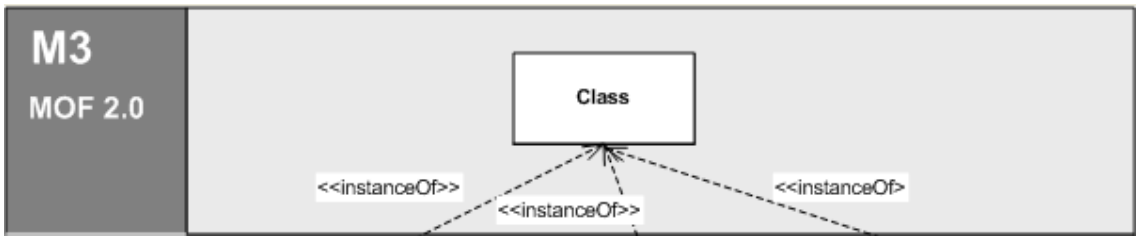
SLIKA 8: POENOSTAVLJEN META-METAMODEL, KI OPISUJE METAMODEL DIAGRAMA STANJ SISTEMA LUČI

Trenutno najbolj znan in v praksi največkrat uporabljen način, s katerim formalno opišemo poljuben metamodel, je standard MOF (Meta Object Facility) konzorcija OMG [3]. MOF je meta-metamodel, ki opisuje celotno razvojno arhitekturo programske opreme (npr. objektno usmerjene računalniške sisteme, različne tehnologije, itn.), vključno samega sebe (slika 9). Iz njega so izpeljani metamodel modelirnega jezika UML, metamodel komponentnih platform, kot sta npr. CCM (Corba Component Model) in EJB (Enterprise Java Bean) in vsi drugi nižje-nivojski (uporabniški) metamodeli. Če nadaljujem z metamodelom modelirnega jezika UML, le-ta nadalje opisuje gradnike UML diagramskih tehnik in relacij med njimi, z modeli, izpeljanimi iz njega, pa opišemo konkretne primerke različnih področij iz realnega sveta.



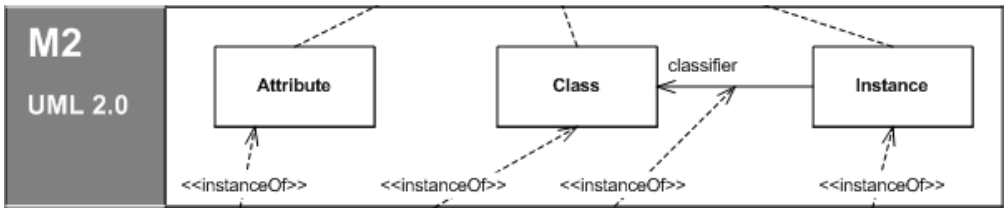
SLIKA 9: METAMODEL MOF IN NJEGOVE IZPELJANKE

Spodnje štiri slike bolj nazorno prikazujejo nivo abstrakcije posameznega člena verige na primeru metamodela UML, od najbolj abstraktnega MOF pa do t.i. nivoja M0, ki opisuje konkretne primerke objektov in povezav med njimi. Kot primer (slika 10) je prikazan koncept razreda, ki je na najvišjem nivoju predstavljen v obliki atomarnega elementa, opisanega z imenom *Class*.



SLIKA 10: KONCEPT OPISA RAZREDA NA NIVOJU M3

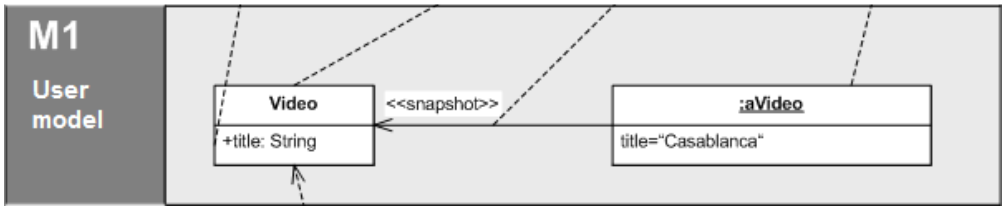
Na nivoju M2 se nahajajo trije elementi (slika 11), poimenovani *Class*, *Attribute* in *Instance* ter so neposredni primerki višje-nivojskega MOF elementa *Class*. Med elementoma *Class* in *Instance* obstaja še asociativna povezava, ki opisuje statično relacijo med tema dvema elementoma.



SLIKA 11: KONCEPT RAZREDA, ATRIBUTA IN PRIMERKA RAZREDA NA NIVOJU M2

Konkretne primerke razredov, ki sledijo pravilom in omejitvam na nivoju M2 opisanih konceptov, prikazuje slika 12. Uporabniški model vsebuje razred *Video*, neposredni primerek elemetna *Class* in je opisan z atributom *title*, neposrednim primerkom elementa *Atribute*. Prav tako je na sliki prikazan primerek razreda *Video*, poimenovan *aVideo* in je neposredni primerek elementa *Instance*, definiranega na nivoju M2.

Ob opisu atributa *title* ne smemo spregledati definiranega podatkovnega tipa *String*, katerega definicija je opisana v enem izmed zgornjih dveh nivojev M3 in M2, na sliki pa je ta definicija skrita.



SLIKA 12: OPIS KONKRETNEGA RAZREDA IN NJEGOVEGA PRIMERKA NA NIVOJU M1

Zaradi lažjega razumevanja lahko na nivoju M0 prikažemo še dejanski primerek iz realnega sveta, ki odraža primer neke konkretne informacije, definirane in opisane z uporabniškim razredom *Video* nivoja M1 (slika 13).

SLIKA 13: PRIKAZ PRIMERKA KONKRETNE INFORMACIJE IZ REALNEGA SVETA

Preverjanje skladnosti oblikovanega modela z njegovim metamodelom mora biti v okviru razvoja po MDD izvedeno samodejno že med samo gradnjo modela. Enkratno končno ali celo ročno preverjanje modela z njegovim metamodelom je popolnoma nesmiselno in se oddaljuje od bistva koncepta metamodeliranja in nenazadnje tudi načel MDD, ki poudarjajo potrebo po formalni izpeljavi (opisu) vsakega modela iz njegovega metamodela. Zaradi tega morajo biti v razvojna orodja vgrajeni mehanizmi preverjanja in različnih omejitev, ki razvijalcu omogočajo izgradnjo formalno pravilnega modela. Če bi razvijalcu prepustili proste roke, da bi preverjanje in ustrezno skladnost z metamodelom izvajal sam »ročno«, recimo ob zaključku modeliranja, bi tako izoblikovani model zagotovo vseboval preveč napak, s čimer bi povečal stroške razvoja in zmanjšal produktivnost svojega dela.

2.4 TRANSFORMACIJE MED MODELI

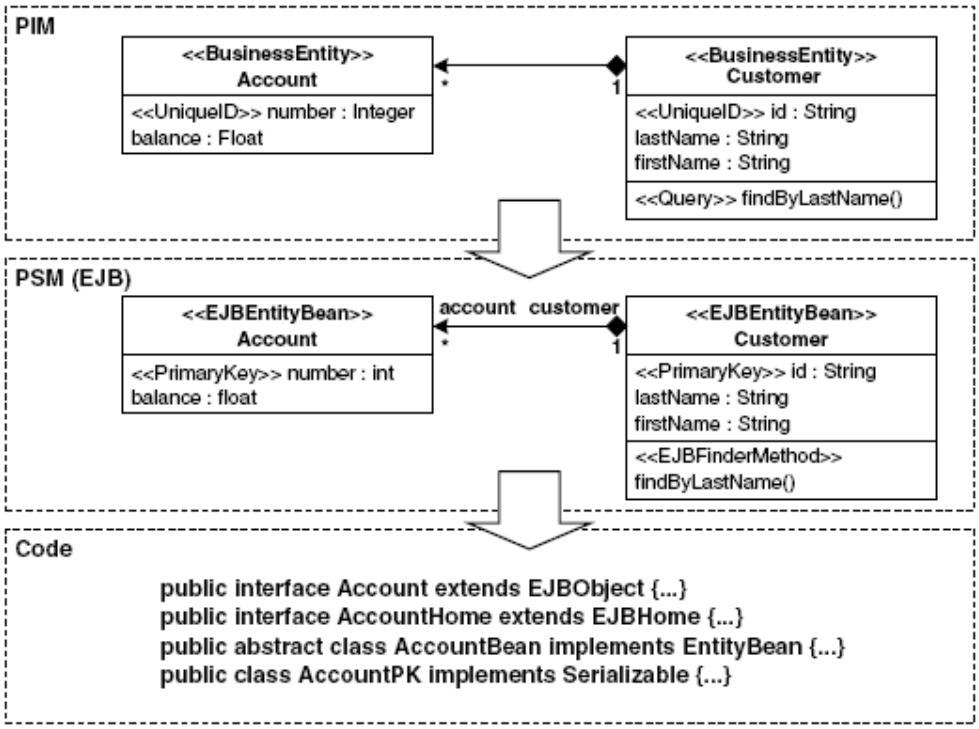
Z modeli mnogo bolj izrazno predstavimo konkretne scenarije problemskega področja, kot pa s pomočjo danes tudi najbolj popularnih in visoko abstraktnih (objektnih) programskih jezikov. Pri tem smo manj ali celo nič vezani na izbrano izvedbeno tehnologijo, s katero modele postopoma pretvorimo v končni izdelek. Takšne konceptualne modele v terminologiji MDD poznamo pod imenom platformsko neodvisni modeli (ang. Platform Independent Models - PIM). Enostaven primer konceptualnega modeliranja lahko izpostavimo iz področja načrtovanja podatkovnih baz. Načrtovalec modelov izbrano organizacijo podatkovnih struktur modelira na osnovi konceptualnega podatkovnega modela, pri čimer izdela načrt na osnovi visoko abstraktnih konceptov, kot so entitetni tip, atribut, relacija med entitetnima tipoma, enolični identifikator itn. Ti koncepti ne predstavljajo dejanskih podatkovnih struktur neke konkretne podatkovne baze (npr. Oracle, SQL Server, Informix ali MySQL), ampak le ustrezajo pravilom t.i. ER modela (ang. Entity-relationship model), s pomočjo katerega podatke abstraktno organiziramo na standardiziran način.

2.4.1 OBRAVNAVA PLAFORMSKO SPECIFIČNIH MODELOV

Da konceptualni modeli ne bi bili sami sebi namen, jih moramo prej ali slej preslikati v modele realizacije, ki so s svojimi modelirnimi elementi neposredno vezani na izbrano implementacijsko tehnologijo (npr. Java oz. .NET). Takšne modele imenujemo platformsko specifični modeli (ang. Platform Specific Models - PSM). Podobno analogijo lahko ponovno prikažemo s primerom iz področja načrtovanja podatkovnih baz. Vsak konceptualni

podatkovni model (ali bolje rečeno - načrt podatkovne baze) načrtujemo zato, da bomo v okviru neke domene v izbrani podatkovni bazi hranili želene podatke. Ker je konceptualni podatkovni model tehnološko neodvisna predstavitev organizacije podatkov, moramo njegove abstraktne elemente znati preslikati v konkretne gradnike izbrane podatkovne baze in le-te zajeti v okviru t.i. fizičnega podatkovnega modela. Če bi kot osnovo izbrali relacijsko podatkovno bazo Oracle, bi entitetne tipe konceptualnega podatkovnega modela preslikali v Oracle tabele, enolične identifikatorje v primarne ključe itn. Na srečo imamo prav za področje podatkovnega modeliranja danes na voljo množico naprednih razvojnih orodij, kot je npr. zelo popularen PowerDesigner, ki nam konceptualne podatkovne modele samodejno pretvorijo v fizične podatkovne modele in pri tem upoštevajo specifike posameznih baz.

Še en primer preslikave iz konceptualnega modela v fizični model in kasneje še v (model) javanske programske kode, prikazuje slika 14 [7]. Z UML PIM modelom je z razrednim diagramom opisana relacija komitent-bančni računi: en komitent ima lahko enega ali več različnih bančnih računov. Bančne račune (razred *Account*) med seboj ločimo na osnovi enolične identifikacijske številke *number*, ki jo opišemo z numeričnim podatkovnim tipom *integer*. Enoličnost navedenega atributa označimo z enoličnim identifikatorjem <<*UniqueID*>>. Na vsakem računu hranimo še podatek o saldu, kar opišemo z atributom *balance*. Podatkovni tip atributa je v tem primeru realno število, kar označimo s *Float*. Vsakega komitenta (razred *Customer*) opišemo z imenom (*firstName*), priimkom (*lastName*) ter enolično številko *id*. Vsi trije atributi hranijo podatke v obliki niza znakov, kar označimo s *String*. V končnem sistemu bomo iskanje komitentov realizirali s pomočjo iskalnega pogoja (po priimku komitenta), kar na modelu PIM označimo s <<*Query*>> *findByLastName()*. V tako podroben opis spodnjega primera smo zašli zato, da bo popolnoma jasna razlika med modeloma PIM in PSM. Do tega trenutka bi namreč morale biti razumljive oznake <<*BusinessEntity*>>, <<*UniquID*>> in <<*Query*>>, s katerimi smo z modelom PIM opisali zgolj konceptualno naravo posameznih elementov domene.



SLIKA 14: PRIKAZ PRIMERA TRANSFORMACIJE IZ PIM V PSM MODEL

V naslednjem koraku izgrajeni model PIM pretvorimo v model PSM izbrane tehnologije EJB (ang. Enterprise Java Bean). Poslovna razreda *Account* in *Customer* sta v izvedbenem modelu pretvorjena v razreda »javanskega fižolčka« (ang. EJB Entity Bean), enolični identifikatorji *<<UniqueID>>* v primarne ključe *<<PrimaryKey>>*, iskalni pogoj *<<Query>>* pa v metodo *<<EJBFinderMethod>>*. Poleg tega so konceptualni podatkovni tipi (*integer*, *Float* in *String*) pretvorjeni v konkretne javanske podatkovne tipe (*int*, *float* in *String*). Pravila za izvedbo transformacij iz PIM v PSM so ponavadi realizirane na osnovi vmesne standardizirane oblike (npr. XML), ki jo bomo opisali v nadaljevanju tega poglavja.

Model PSM (EJB) nato pretvorimo v ogrodje konkretne javanske programske kode, ki ni nič drugega kot še en model PSM s sintakso in semantiko javanskega programskega jezika. Če primerjamo slednjega s prvotnim modelom PIM, ni težko ugotoviti, da vsebuje očitne elemente izvedbene tehnologije EJB, ki nimajo nobene veze z modeliranim problemskim področjem – komitenti in njihovimi bančnimi računi. Na ta način smo začetni konceptualni opis domene zaščitili pred nepotrebnim »balastom«, ki je specifičen le za to tehnologijo. Če bi izbrali neko popolnoma drugo izvedbeno platformo, kot je npr ASP.NET, bi model PIM ostal isti, model PSM s pripadajočo programsko kodo pa bi vseboval elemente izbranega programskega jezika (npr. C#, Visual Basic ipd.). Ob tem lahko izpostavimo še nekaj. Model PIM bi v našem primeru načeloma lahko neposredno pretvorili v programsko kodo, brez vmesnega izvedbenega modela PSM. Toda s tem ne bi prav nič pridobili. Kvečjemu ogromno izgubili. Vmesno obliko modela PSM potrebujemo zato, da s pomočjo višje-abstraktne

predstavitvene logike vizualnega modela bolj enostavno in intuitivno modeliramo problem, katerega elementi so že postavljeni v ogrodje izvedbenega tehnološkega okolja. S tehnikami ločevanja in združevanja avtomatsko generirane ter ročno napisane programske kode pa potem oboje združimo v nižje-nivojski model PSM programske kode. Več o navedenih tehnikah bomo govorili v naslednjem poglavju.

Transformacije med modeli, od začetnega - najbolj »abstraktnega«, do končnega – najbolj »konkretnega«, lahko izvedemo postopoma, dokler ne pridemo do želene ciljne oblike. Ta je v svoji končni prezentaciji, kot smo jasno videli v zgornjem primeru, lahko tudi programska koda poljubnega programskega jezika. Odločitev, kako daleč bomo šli, je odvisna od primera do primera - nivo želene abstrakcije je v največji meri določen z znanji oz. izkušnjami arhitektov-programerjev, (dobre) prakse v organizaciji in zahtev trenutnega projekta. Bistveno je, da z modeliranjem prihranimo čas in zmanjšamo stroške izvedbe. Pomembno dejstvo, ki ga na tem mestu lahko omenimo je, da pri modeliranju in transformacijah modelov iz ene oblike v drugo nismo nujno vezani le na eno razvojno orodje. Modele lahko predstavimo na grafičen način: v obliki elementov in povezav med njimi, v obliki razpredelnic, kot strukturiran tekst ali kako drugače. Če smo izvorni model učinkovito izrazili v enem orodju (nekega proizvajalca), lahko ciljni model, v primeru medsebojne standardizirane povezanosti, predstavimo ali oblikujemo tudi v drugem. S tem se približamo konceptu prenosljivosti modelov med različnimi odprtokodnimi in komercialnimi okolji (»model interoperability«), s čimer se oddaljimo od prepogosto nujne vezanosti na nek konkreten produkt, okolje ali standard.

2.4.2 PRIMERI NAJBOLJ POGOSTIH NAČINOV TRANSFORMACIJ MED MODELI

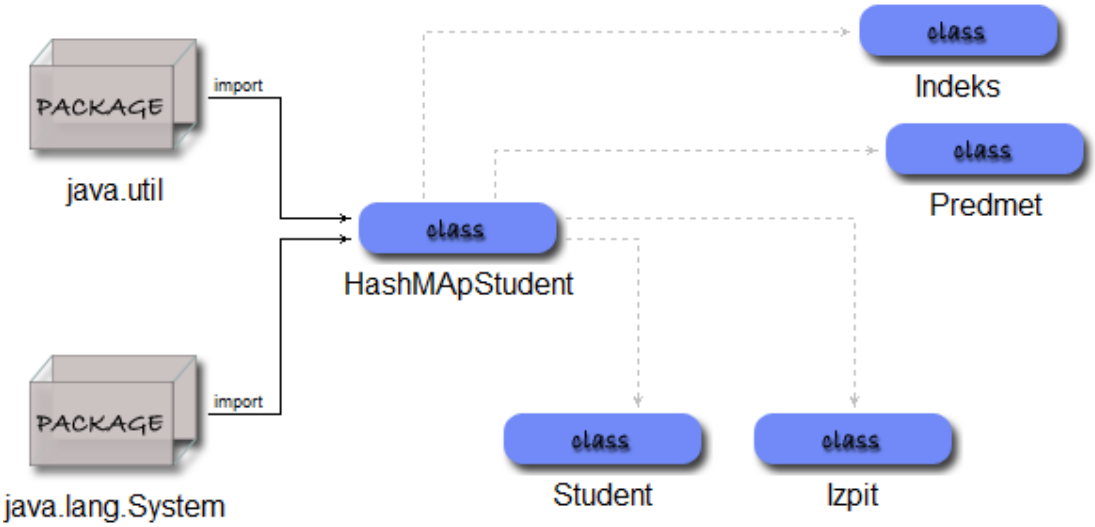
Pri realizaciji transformacij (med modeli) si lahko pomagamo z različnimi tehnikami in pristopi, ki so v praksi pokazali velik potencial [3]. Naj opišemo dva izmed najbolj pogostih pristopov:

- S postopki neposredne interakcije z modelom (ang. Direct Model Manipulation)

Preko vmesnikov, ki so realizirani v enem izmed programskih jezikov (npr. C++ ali Visual Basic), neposredno dostopamo do elementov modela in nato nad njimi izvajamo želene ukrepe (transformacije). Model preoblikujemo iz izvorne oblike v ciljno tako, da napišemo računalniški program, ki s pomočjo vmesnikov pridobi kontrolo nad posameznimi elementi izvornega modela in jih na želen način pretvori v ciljni model poljubne oblike. Programer mora sam poskrbeti za to, da oblika ustreza vsem pravilom izbrane platforme. Naj na tem mestu samo omenimo, da smo omenjeni način transformacij preizkusili tudi v praktičnem delu magistrske naloge, kar bomo tudi bolj podrobno opisali v poglavju *5.5 – Opis interpereterja*.

Na slikah 15, 16 in 17 si je v grobem mogoče ogledati koncept dostopa do elementov modela, da si opisani način transformacije lažje predstavljamo. Slika 15 prikazuje enega izmed izgrajenih uporabniških modelov, ki vsebuje elemente in povezave med njimi. Tako razredi in paketi (*java.util*, *java.lang.System*, *Student*, *Izpit*, *Indeks*…) kot tudi povezave (*import*, *Public – označene s črtkano povezavo*), so v modelu prikazani z različnimi tipi. Trenutno ni potrebno razumeti njihovega smisla, omenjeni so le zato, da bomo lažje razumeli izsek programske kode, v kateri v ponavljajoči FOR zanki iščemo elemente oz. povezave točno določenega tipa.

Na sliki 16 je prikazan del programske kode, iz katerega je razvidna implementacija ustreznega vmesnika, preko katerega si pridobimo dostop do elementov in povezav modela. Kot je razvidno iz primera, je implementacija vmesnika (public void InvokeEx(…)) v tem primeru realizirana preko javanske programske kode, lahko pa bi bila tudi v katerem izmed drugih programskih jezikov (npr. C++, C# itd.). Ustrezen nabor možnih kandidatov je seveda odvisen od proizvajalcev MDD orodij, praksa pa je v tem trenutku osredotočena v množico objektno usmerjenih programskih jezikov, ki so v zadnjem desetletju najbolj pridobili na popularnosti. Poleg tega se njihova objektna usmerjenost zelo lepo »pokrije« s samo naravo modelno opisanih problemov (metamodel = razred, model = objekt), kar še poveča njihovo ustreznost za modelno voden razvoj.



SLIKA 15: PRIMER IZGRAJENEGA UPORABNIŠKEGA MODELA

Slika 17 prikazuje metodo *public Vector najdiPublicRazrede()*, v kateri se sprehodimo čez drevesno strukturo modela in poiščemo ustrezne elemente. Kot je mogoče razbrati iz programske kode, se v zunanji FOR zanki sprehodimo čez vse elemente modela, v notranji pa čez vse povezave tipa »Public«. Če za neko povezavo ugotovimo, da je njen ponor določen element modela (zunanje zanke), le-tega preskočimo. V nasprotnem primeru pa je element ustrezen in ga shranimo v ustrezni vektor.

```
import java.util.*;
// Uvoz ustreznih knjižnic, preko katerih bomo dostopali do modela
import org.isis.gme.bon.*;
import org.isis.gme.meta.*;
import java.io.*;
import javax.swing.*;

// Implementacija vmesnika
public class JCP implements BONComponent
{
  Vector<JBuilderConnection> povezave;
  StringBuffer text=new StringBuffer();
  JBuilderModel model;

  //Zahtevana metoda vmesnika
  public void invokeEx(JBuilder builder, JBuilderObject focus, Collection selected,int param)
  {
   ...
   ...
    String trenutniPogled = new MgaMetaModel(focus.getMeta()).getAspects().getItem(0).getName();

    // S pomočjo spremenljivke "model" si pridobimo dostop do modela.
   if (trenutniPogled.equals("Koda")) model=((JBuilderModel)focus).getParent();
   else model = (JBuilderModel)focus;

    // Če model ne vsebuje elementov, končamo z izvajanjem programa.
   if (model.getModels().size()==0) return;

    // Poiščemo vse elemente modela
    Vector<JBuilderModel> publicElementi = najdiPublicRazrede();
   ...
   ...
  }
```

SLIKA 16: IZSEK IZ PROG. KODE ZA IZVEDBO TRANFORMACIJE (IMPLEMENTACIJA VMESNIKA)

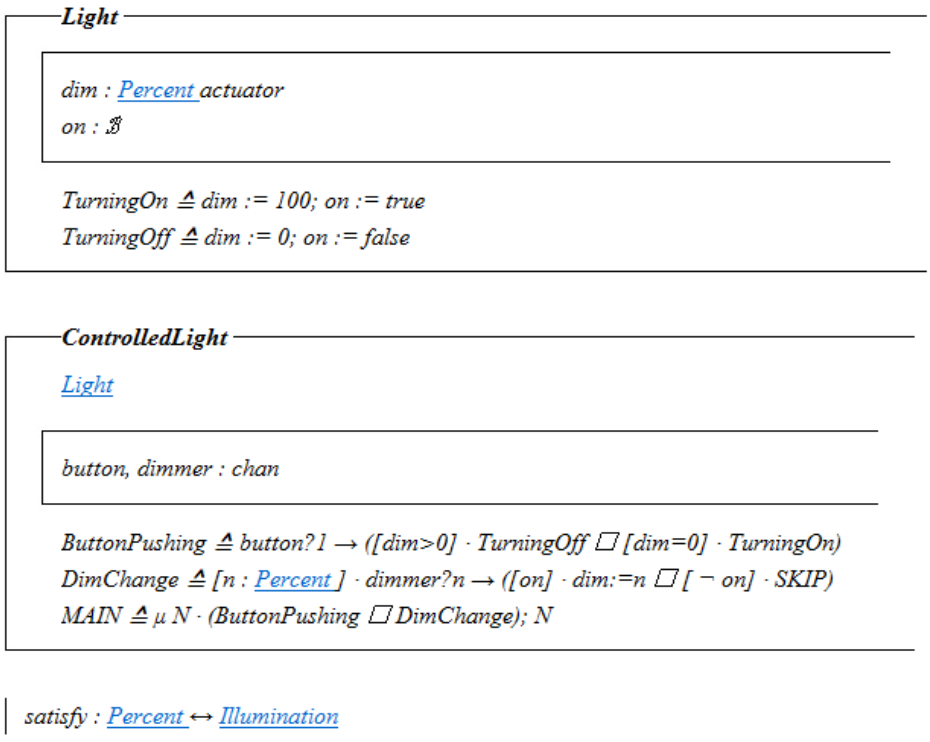
```
private Vector najdiPublicRazrede(){

    // V spremenljivko "elementi" pridobimo VSE elemente modela.
    Vector<JBuilderModel> elementi = model.getModels();

    // V spremenljivko "publicElementi" bomo shranili vse elemente
    // modela, ki jih potrebujemo v okviru te metode.
    Vector<JBuilderModel> publicElementi = new Vector<JBuilderModel>();
    boolean notPublic;

    // Sprehodimo se čez vse elemente
    for(JBuilderModel element: elementi){

        // Pridobimo ustrezne povezave med elementi modela
        povezave = model.getConnections("Public");
        notPublic = false;

        for(JBuilderConnection povezava: povezave){

            // Preverimo ustreznost elementa, ki ga iščemo.
            if (povezava.getDestination()==element){

                notPublic=true;
                break;

            }
        }
        // Če element ustreza iskalnim pogojem, ga shranimo.
        if (!notPublic) publicElementi.add(element);
    }
    return publicElementi;
}
```

SLIKA 17: IZSEK IZ PROG. KODE ZA IZVEDBO TRANFORMACIJE (SPREHOD ČEZ ELEMENTE MODELA)

- S standardiziranimi prehodnimi oblikami med modeli (ang. Intermediate representation)

S pomočjo ustreznih orodij izvorni model »izvozimo« v neko standardizirano obliko in ga nato kot takega »uvozimo« v ciljni model. Kot dober primer takšnega koncepta preslikav med modeli so razvijalci vajeni v okviru standarda XMI (XML Metadata Interchange), ki je bil oblikovan specifično za potrebe izvedbe transformacij med UML modeli na osnovi označevalnega jezika XML. Omenjena standardizirana prehodna oblika v tem primeru je torej zapis modela v obliko XML dokumenta, ki predstavlja povezovalno nit med izvornim in ciljnim modelom. Ni težko ugotoviti, da takšen pristop spodbuja koncept prenosljivosti modelov, kar je še dodaten razlog za dejansko uporabnost tega pristopa v praksi. Če je prehodna standardizirana oblika sprejeta s strani velike večine mogočnih korporacij in odprtokodnih organizacij, ki oblikujejo standard MDD, smo lahko prepričani v uspeh in bodočo podporo tej obliki, kot je npr. standard XMI.

[17] opisuje lep primer uporabe standarda XMI kot povezovalne vezi med dvema modeloma, čeprav gre v tem primeru zgolj za predstavitev istega statičnega dela sistema v dveh različnih notacijah: nepregledne in na prvi pogled kompleksne (matematične) oblike TCOZ (ang. Timed Communicating Object-Z) v obliko preprostega in preglednega UML razrednega diagrama.



SLIKA 18: DEL MODELA SISTEMA LUČI, OPISAN V NOTACIJI TCOZ

Slika 18 prikazuje del modela sistema za upravljanje z lučmi, opisanega z jezikom TCOZ. Na sliki ni prikazan celoten model sistema, pač pa le del, ki opisuje prižiganje in ugašanje luči. Celoten model si je mogoče ogledati na navedenem spletnem naslovu.

V podrobnejšo sintaktično in semantično obravnavo jezika se ne bi spuščali, naj na tem mestu omenimo le, da je namenjen modeliranju kompleksnih podatkovnih stanj, komunikacij med objekti, izvajanja v realnem času, vzporednega procesiranja in večnitnih obdelav informacij. Z jezikom je kompleksne sisteme mogoče razčleniti v več manjših in obvladljivih enot z namenom lažjega obvladovanja posameznih lokalnih delov sistema.

Na osnovi formalne definicije jezika TCOZ lahko določimo ustrezno XML shemo, s katero natančno opišemo strukturo nekega XML dokumenta – v našem primeru dokumenta, v katerem bo na strukturiran način mogoče opisati katerikoli model TCOZ. Slika 19 prikazuje del XML kode, ki opisuje na priloženem TCOZ modelu definiran element »Light«. Tudi če ne poznamo nobenega od obeh navedenih jezikov, lahko intuitivno primerjamo obe sliki in iz njih jasno razberemo »popis« istih atributov elementa.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="http://www.comp.nus.edu.sg/~sunjing/objectZ/objectzed.xsl"?>
<!DOCTYPE unicode SYSTEM "http://www.comp.nus.edu.sg/~sunjing/objectZ/myunicode.dtd">
<objectZnotation>
        <classdef layout="simpl" align="left">
          <name>Light</name>
          <state>
              <decpart>
                  <decl>
                      <name>dim</name>
                      <dtype>
                          <type ty="predefine">Percent</type>
                          <type ty="basic">&actuator;</type>
                      </dtype>
                  </decl>
                  <decl>
                      <name>on</name>
                      <dtype>
                          <type ty="basic">&bool;</type>
                      </dtype>
                  </decl>
              </decpart>
          </state>
          <op layout="calc">
                  <name>TurningOn</name>
                  <predicate>dim := 100; on := true</predicate>
          </op>
          <op layout="calc">
                  <name>TurningOff</name>
                  <predicate>dim := 0; on := false</predicate>
          </op>
        </classdef>
```

SLIKA 19: DEL XML KODE, KI OPISUJE TCOZ ELEMENT "LIGHT"

Ko imamo model enkrat zajet v obliki XML dokumenta, je potrebno le-tega skladno s predlaganim postopkom preoblikovati v vmesno standardizirano obliko XMI. To naredimo s

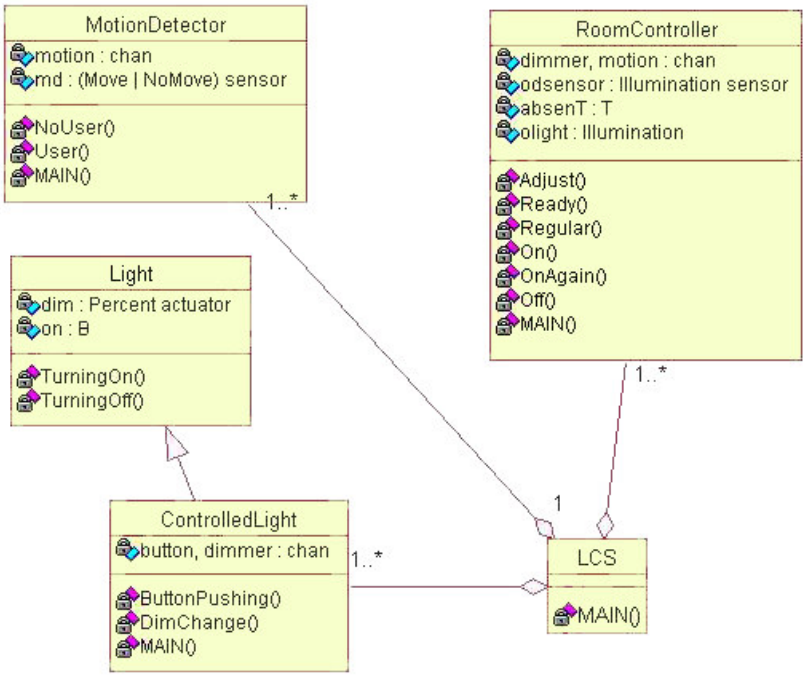
pomočjo pravil, definiranih v okviru XSL transformacije, ki vhodni XML dokument TCOZ modela pretvori v ciljni XML dokument oblike XMI, katere del je prikazan na sliki 20.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE XMI (View Source for full doctype...)>
- <XMI xmi.version="1.0">
  <XMI.header />
  - <XMI.content>
    - <Model_Management.Model xmi.id="G.1">
      <Foundation.Core.ModelElement.name>UML_XMI</Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility xmi.value="private" />
      <Foundation.Core.GeneralizableElement.isRoot xmi.value="false" />
      <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false" />
      <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false" />
      - <XMI.extension xmi.extender="Unisys.IntegratePlus.2">
        <XMI.reference xmi.idref="G.21" />
      </XMI.extension>
      - <Foundation.Core.Namespace.ownedElement>
        - <Foundation.Core.Class xmi.id="S.10001">
          <Foundation.Core.ModelElement.name>Light</Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.visibility xmi.value="public" />
          <Foundation.Core.GeneralizableElement.isRoot xmi.value="true" />
          <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false" />
          <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false" />
          <Foundation.Core.Class.isActive xmi.value="false" />
          - <Foundation.Core.ModelElement.namespace>
            <Model_Management.Model xmi.idref="G.1" />
          </Foundation.Core.ModelElement.namespace>
          - <Foundation.Core.GeneralizableElement.specialization>
            <Foundation.Core.Generalization xmi.idref="G.20" />
            <!-- { ControlledLight -> Light } -->
            <Foundation.Core.Generalization xmi.idref="G.20" />
            <!-- { ControlledLight -> Light } -->
          </Foundation.Core.GeneralizableElement.specialization>
          - <Foundation.Core.ModelElement.taggedValue>
            - <Foundation.Extension_Mechanisms.TaggedValue>
              <Foundation.Extension_Mechanisms.TaggedValue.tag>persistence</Foundation.Extension_Mechanisms.TaggedValue.tag>
              <Foundation.Extension_Mechanisms.TaggedValue.value>transient</Foundation.Extension_Mechanisms.TaggedValue.value>
            </Foundation.Extension_Mechanisms.TaggedValue>
          </Foundation.Core.ModelElement.taggedValue>
          - <Foundation.Core.Classifier.feature>
            - <Foundation.Core.Attribute xmi.id="S.10002">
              <Foundation.Core.ModelElement.name>dim</Foundation.Core.ModelElement.name>
              <Foundation.Core.ModelElement.visibility xmi.value="private" />
              <Foundation.Core.Feature.ownerScope xmi.value="instance" />
              <Foundation.Core.StructuralFeature.multiplicity>1..1</Foundation.Core.StructuralFeature.multiplicity>
              <Foundation.Core.StructuralFeature.changeable xmi.value="none" />
              <Foundation.Core.StructuralFeature.targetScope xmi.value="instance" />
              - <Foundation.Core.Attribute.initialValue>
```

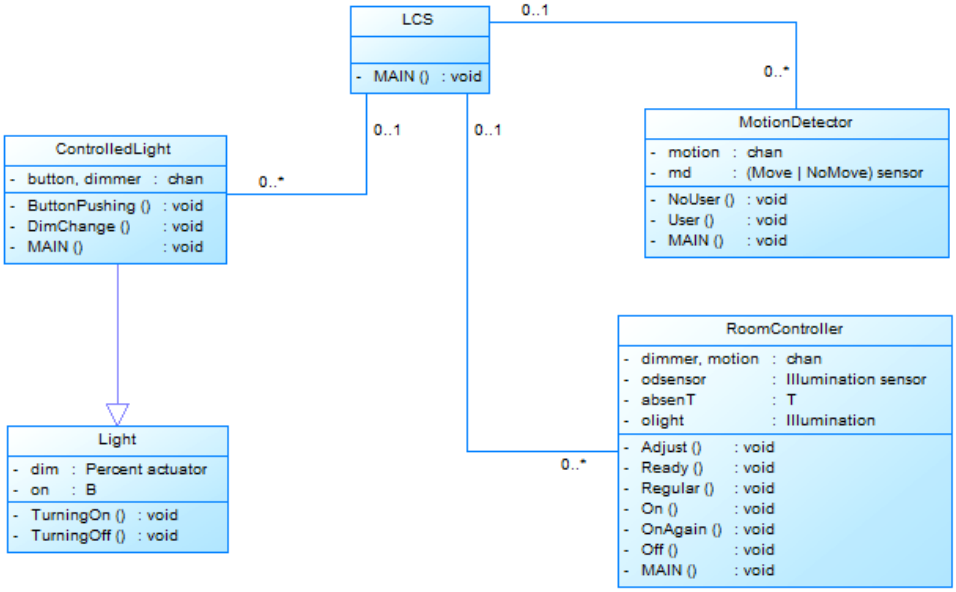
SLIKA 20: DEL XMI KODE TCOZ MODELA

XMI dokument predstavlja vmesno standardizirano obliko, ki jo v naslednjem koraku lahko »izvozimo« v ciljni model. Avtorji priloženega primera predlagajo orodje Rational Rose 2000 Enterprise, ki podpira standard XMI. Da pa bi pokazali bistvo uporabe standardiziranih oblike, smo XMI dokument uvozili tudi z orodjem Sybase PowerDesigner 12. Na sliki 14 in 15 je mogoče primerjati oba ciljna modela.

Avtorji so z navedenim primerom želeli prikazati statični del istega dela sistema s pomočjo dveh različnih modelov. Če bi z UML diagramskimi tehnikami želeli prikazati še dinamični vidik omenjenega sistema, bi lahko to modelirali z diagrami stanj. Tega zaenkrat avtorji primera še niso realizirali, nameravajo pa to storiti v prihodnje. Kljub vsemu upamo, da je priloženi primer dober za razumevanje opisanega koncepta transformacij med modeli in hkrati jasno poudarja potrebo po standardizirani izmenjavi dokumentov.



SLIKA 21: RAZREDNI DIAGRAM ORODJA RATIONAL ROSE



SLIKA 22: RAZREDNI DIAGRAM ORODJA POWERDESIGNER

2.5 ZDRUŽEVANJE AVTOMATSKO GENERIRANE IN ROČNO NAPISANE PROGRAMSKE KODE

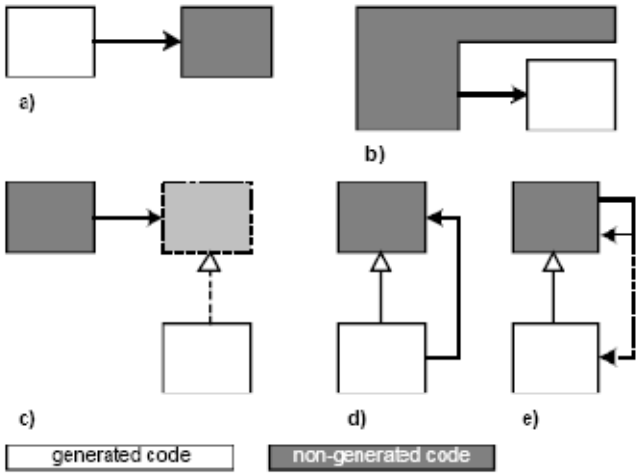
Koncept avtomatskega generiranja programske kode iz oblikovanega modela ni z vidika MDD nič drugega kot izvedba transformacije, ki model na osnovi vnaprej definiranih pravil preslika v tekstualni zapis ali bolje rečeno - programsko kodo. Zagovorniki paradigme MDD ta koncept pogosto enačijo z vlogo prevajalnikov, ki programerja ščitijo pred podrobnim poznavanjem komponent strojne opreme. Podobno kot so prevajalniki nadomestili programiranje v zbirnem jeziku, naj bi tudi koncept modeliranja (in postopnega avtomatskega generiranja) programske kode postopoma prevladal nad klasičnim programiranjem in naj bi ga v bližnji prihodnosti v celoti nadomestil.

Kljub vsemu pa bo z vidika produktivnosti, zahtev po poenostavitvi sistema ali časovnih omejitev za določene dele sistema še vedno potrebno ročno napisati programsko kodo. Bodisi bo zahtevana funkcionalnost preveč kompleksna, da bi jo lahko enostavno modelirali z višje abstraktnimi elementi modelirnih jezikov, bodisi bo modeliranje predrago ali pa enostavno ne bo potrebe, da bi prav vse realizirali po načelih in priporočilih MDD. Današnja razvojna orodja nam ponujajo različne tehnike, s katerimi lahko združujejo oz. ločujejo generirano (GPK) in ročno napisano (RNPK) programsko kodo. Najbrž ni težko razumeti želje, da moramo ta dva pojma obravnavati popolnoma ločeno, če želimo organizirano in obvladljivo razviti zahtevani sistem. Želja po fizičnem ločevanju GPK in RNPK je celo nujna, saj le tako lahko dosežemo potrebno implementacijsko neodvisnost med obema oblikama kode.

Če npr. modeliramo postopek izdelave poljubnega avtomobila v tovarni vozil, recimo kateri robot bo nek del avtomobila v določenem trenutku pritrdil na ogrodje vozila, lahko izberemo nek splošni in vnaprej definirani modelirni vzorec, po katerem se bo za vsako vrsto vozila na točno določeno mesto vedno najprej pritrdilo kolesa, nato zavorni sistem, žaromete itn. Takšen vzorec bi npr. enostavno modelirali z diagramu poteka podobnim jezikom, kjer bi za vsak korak (in vsako vrsto vozila) imeli vnaprej definirane ustrezne parametre. Če bi za točno določeno vrsto avtomobila želeli definirati, naj roboti namesto standardnih luči vgradijo posebne LED sisteme, ki zahtevajo popolnoma drugačen postopek vgradnje, bi želeni postopek vgradnje opisali (sprogramirali) popolnoma neodvisno in ločeno od izbranega splošnega vzorca. Tako bi poleg podpore za vgradnjo standardnih avtomobilskih delov, ki smo jo npr. dobili ob nakupu robotskega sistema, imeli možnost razširitev sistema za vgradnjo še poljubnih nestandardnih avtomobilskih delov.

Logično je, da programske module GPK in RNPK obravnavamo ločeno. Še več, smiselno je, da jih neodvisno razvijemo, hranimo in vzdržujemo posebej, obstajajo pa načini, kako jih lahko združujemo. Le tako lahko na ponovljiv, obvladljiv in smotrn način gradimo nove kompleksne sisteme, ki so osnovani na ponovljivih vzorcih dobrih izkušenj in so razširjeni z »ročno definirano« specifiko za točno izbran namen ali nalogo. GPK in RNPK lahko med seboj združujemo na različne načine [8] (slika 23):

- GPK se sklicuje na metode paketov oz. knjižnic, katerih vsebino definiramo v okviru RNPK (primer a). Obseg GPK je v tem primeru dosti manjši kot bi sicer bil, saj se zanašamo na realizirane funkcionalnosti ročno napisanih modulov.
- Možen je tudi obraten primer, v katerem RNPK oblikujemo tako, da kliče posamezne dele GPK (primer b). Nekoliko razširjeno različico tega izvedemo tako, da
- RNPK oblikujemo vzporedno z abstraktnimi razredi (vmesniki), ki jih realiziramo s pomočjo GPK (primer c).
- Razrede GPK lahko razširimo tudi z nadrazredi RNPK. Generične metode nadrazredov enostavno uporabimo znotraj metod podrazredov (primer d)
- Možna je tudi obratna situacija, v kateri se znotraj osnovnega abstraktnega RNPK razreda sklicujemo na abstraktne metode, ki bodo realizirane v okviru razširjenih GPK razredov. V tem primeru moramo seveda zagotoviti končne neabstraktne razrede, s katerimi bomo lahko ustvarili konkretne primerke teh razredov (primer e)



SLIKA 23: NAČINI ZDRUŽEVANJA PROGRAMSKE OPREME

Zgornji primeri prikazujejo samo nekaj izmed najbolj tipičnih načinov združevanja generirane in ročno napisane programske kode. Želje nekaterih, da bi postopoma popolnoma opustili ročno napisano programsko kodo in postali popolnoma odvisni od generirane programske kode, je najbrž pretirana. Popolnoma jasno je, da je nivo abstrakcije novodobnih programskih jezikov človeku že tako »blizu«, da bi njihova opustitev vsekakor prej pomenila nenadomestljivo izgubo kot pa kakršnokoli pridobitev.

2.6 PODPORA STANDARDOM, ORODJEM IN METODOLOGIJI

MDD razvoj programskih rešitev kljub sedaj že nekajletnemu prizadevanju posameznikov (Stahl Thomas, Völer Markus, Bettin Jorn) in nekaterih organizacij, med katerimi ima najvidnejšo vlogo OMG, v praksi še ni dodobra sprejet. K temu pripomore dejstvo, da je bilo do sedaj sprejetih relativno malo standardov, priporočil in razvojnih orodij, s katerimi bi razvijalci uspešno zaključili konkretne MDD projekte v praksi. Še najbolj odmevno pobudo v to smer že nekaj let prispeva konzorcij OMG s svojo različico MDD, imenovano MDA (Model driven Architecure). S splošno sprejetimi standardi in tehnologijami, kot so npr. UML (Unified Modeling Language), XMI, MOF, CORBA (Common Object Request Broker Architecture), OCL, CWM (Common Warehouse Metamodel) idr., bi jih lahko postavili v ospredje razvoja po MDD. Njihovi modelirni standardi so realizirani v orodjih različnih komercialnih proizvajalcev in odprtokodnih združenj [19], ki se trudijo, da bi ta orodja postala mehanizem novodobnega razvoja programske opreme. Pri tem je potrebno biti nekoliko previden, saj sem spadajo tako orodja, ki nudijo podporo le standardu UML, kot tudi orodja, ki s podporo raznovrstnim standardom za izmenjavo podatkov, splošno sprejetimi modelirnimi pristopi in neodvisnostjo od konkretnih proizvajalcev, želijo v celoti slediti načelom in priporočilom razvoja po MDD. Dejstvo je, da večina izmed orodij prve vrste zagotavlja modelirno podporo začetnim razvojnim fazam sistemov, medtem ko se v kasnejših fazah modele največkrat uporabi zgolj kot drugo-razredno dokumentirno sredstvo. Takšna orodja ne moremo obravnavati kot prava MDD orodja, saj z njimi ne moremo nuditi modelirne podpore celot¸nemu življenjskemu ciklu izdelka. Iz te množice lahko izvzamemo le orodja, ki se opirajo na standarde za izmenjavo modelov med orodij, kot je npr. že prej omenjeni XMI.

MDA mnogi očitajo, da v celoti ne sledi bistvenim konceptom in priporočilom razvoja po MDD. Glavni vzrok temu je v prvi vrsti enotni modelirni jezik UML, ki se, kakor smo povedali že prejšnjih poglavjih, bistveno razlikuje od DSL jezika v pravem pomenu besede. Kljub temu, da lahko tudi jezik UML posebej prilagodimo potrebam posameznega problemskega področja, naj bi bili pri tem še vedno preveč vezani na močne vezi konzorcija OMG, ki naj bi si prizadeval oblikovati standarde predvsem zaradi dodane vrednosti partnerskim proizvajalcem razvojnih orodij. V zvezi s tem konzorciju očitajo tudi mnogo premočen agnostičen pogled na odprtokodne rešitve in standarde.

Skupina uglednih posameznikov (Jorn Bettin, Craig Cleaveland, Ghica van Emde Boas, Markus Voelter) je že pred časom objavila obširen dokument [4], v katerem opisujejo svoj pogled oz. različico MDD, imenovano kar Model-Driven Software Development [20]. V dokumentu jasno navajajo kritike MDA in si za razliko od njih v celoti prizadevajo podpreti po njihovem mnenju glavne vrednote MDD:

- We prefer to validate software-under-construction over validating software requirements

- We work with domain-specific assets, which can be anything from models, components, frameworks, generators, to languages and techniques
- We strive to automate software construction from domain models; therefore we consciously distinguish between building software factories and building software applications
- We support the emergence of supply chains for software development, which implies domain-specific specialization and enables mass customization

Omenjena skupina združuje, opisuje in predpisuje množico različnih tehnik, pristopov, priporočil, vrednot in načel ter najboljših izkušenj, ki naj bi nudili podporo štirim razvojnim področjem: modeliranju problemskega področja, arhitekturi razvojnih orodij, podpori procesu razvoja ter oblikovanju okolij za izdelavo modelno usmerjenih programskih rešitev. Skupina poudarja potrebo po nujnosti upoštevanja odprtokodnih pristopov ter se sklicuje na nujnost spoštovanja vrednot in načel metodologij agilnega razvoja programske opreme. V tem pogledu skupina nasprotuje metodologiji RUP (Rational Unified Process), ki je po njihovem mnenju preobširna (pretežka), ne nudi resnične podpore vzporednemu razvoju modelirnih okolij in konkretnih izdelkov, ki jih iz teh okolij lahko izoblikujemo ter ne podaja dovolj praktičnih napotkov za realizacijo iterativno-inkrementalnega razvoja v praksi. Skupina podaja priporočila glede uporabe odprtokodnih MDD orodij za oblikovanje metamodelov domenskih področij (Eclipse Generic Modeling Environment, openArchitectureWare Generator, Generative Model Transformer), modelov, ki jih iz njih lahko izpeljemo, DSL modelirnih jezikov ter postopkov za izvedbo transformacij med modeli. V že prej navedenem dokumentu avtorji po posameznih razvojnih področjih navajajo tehnike ločevanja GPK in RNPK, opisujejo smiselnost »dvotirnega« iterativnega razvoja (Dual Track Development), podaja navodila za smotrno obvladovanje stroškov (Fixed Budget Shopping Basket) in učinkovito porazdeljevanje virov (Scope trading), predlagajo smernice za oblikovanje jasnih in kvalitetnih metamodelov (Talk Metamodel, Formal Metamodel...) itd.

V okviru projekta CALM/CANDENA [21] je bil oblikovan opisni jezik CALM (The Cadena Architecure Language with Meta-modeling), s katerim opišemo poljuben komponentni oz. modularni sistem. Modeliramo ga kot množico komponent s poslovno logiko, storitvami in povezavami med njimi. Ker je CALM zasnovan kot ne-grafični modelirni jezik, je bil kot dodatek orodja Eclipse razvito orodje CANDENA, ki s pomočjo različnih razpredelnic ter zaslonskih mask za zajem in izpis podatkov omogoča preprosteje oblikovati modele, nad njimi izvajati različne poizvedbe in jih pretvarjati iz ene oblike v drugo. Več o opisanem projektu si je mogoče prebrati na spletnih straneh projekta ali v članku [9].

S pomočjo orodja GME za metamodeliranje in oblikovanje DSL modelirnih jezikov je bil zasnovan neodvisni projekt C-SAW (Constraint-Specification Aspect Weaver) [22]. Namen projekta je oblikovati mehanizem, s katerim je v okolju GME oblikovane modele mogoče

preoblikovati (v ciljni model) na osnovi razširitve omejitvenega jezika OCL imenovanega ECL (Embedded Constraint Language). Z jezikom ECL torej opišemo potrebne postopke oz. korake, ki so zahtevani za izvedbo želene transformacije. Mehanizem C-SAW postopke izvede z naborom vmesnikov, s katerimi je omogočen dostop do posameznih elementov izvornega modela. Na domači strani projekta si je mogoče ogledati video predstavitve nekaterih zmožnosti opisanega GME dodatka.

Podjetje Microsoft je že v okviru svojega IDE razvojnega orodja Visual Studio 2005 dalo na tržišče orodje, imenovano Domain-Specific Language Tools. Orodje je namenjeno izdelavi DSL modelirnih jezikov, s katerimi posredno preko oblikovanih modelov generiramo končno programsko kodo. Microsoft je na ta način v enega izmed svojih najbolj popularnih razvojnih orodij pogumno vključil modelirno podporo razvoju in svojo očitno uspešno zgodbo nadaljeval tudi v novi različici orodja - Visual Studio 2008. Že od samega začetka pri Microsoftu jasno izražajo prepričanje, da je GPK vedno znova potrebno dopolniti z RNPK, ker le tako razvijalcu pustimo dovolj svobode pri ustvarjanju izdelka in hkrati povečamo njegovo produktivnost dela. Enega izmed najboljših spletnih priročnikov, vključno z nazornimi video predstavitvami za učenje z orodjem DSL tools, je mogoče najti na [23].

Upamo, da smo s pravkar navedenimi primeri pokazali kako dejavno je področje razvoja po MDD v zadnjih nekaj letih. Prav zaradi tega smo se v okviru magistrske naloge odločili, da bomo to področje tudi bolj podrobno raziskali. V naslednjih poglavjih bomo opisali prototip orodja za modelno voden razvoj javanskih programov, ki smo ga izdelali v praktičnem delu magistrske naloge in s katerimi smo priporočila, modelirne tehnike in najboljše izkušnje MDD preizkusili tudi v praksi. Novo poglavje bomo začeli z opisom problemske domene, iz katere smo črpali potrebno znanje in informacije za izdelavo omenjenega prototipa.

3. OPIS PROBLEMATIKE IN IDEJA ZA IZVEDBO PRAKTIČNEGA DELA MAGISTRSKE NALOGE

»A programming language is a system of notation for describing computations. A useful programming language must therefore be suited for both description (i.e., for human writers and readers of programs) and for computation (i.e., for efficient implementation on computers). But human beings and computers are so different that it is difficult to find notational devices that are well suited to the capabilities of both.« R.D. Tennent, Principles of Programming Languages

Učenje programskih jezikov je samo po sebi zahteven proces. Že dijaki višjih letnikov gimnazij in večine srednjih šol, ki se v času študija le v grobem srečajo z enim izmed programskih jezikov pravijo, da je programiranje »težko«. Če se sam spomnim svojih prvih gimnazijskih občutkov, ko sem se prvič spoznal z najenostavnejšimi programskimi konstrukti, ko so npr. preproste definicije spremenljivk, konstant, operatorjev, pogojnih stavkov itn., moram priznati, da je bil to zame popolnoma tuj svet: drugačen način razmišljanja, nova frazologija, opisovanje in reševanja nekih popolnoma »nesmiselnih« problemov. Vse te definicije in primeri skrajno primitivnih programov, ki niso znali poleg izpisovanja na ekran in ponavljajočega seštevanja izvajati nič kaj bolj pametnega, so bili po mojem mnenju le nujno zlo učiteljevega učnega programa, od katerega nisem dosti odnesel. Le neprijeten občutek, da programiranje ni enostavno, kaj šele smiselno. K temu je najbrž pripomoglo preprosto dejstvo, da me je takrat bolj kot programiranje zanimalo vse drugo: igranje računalniških iger in sestavljanje komponent strojne opreme.

Podobne zgodbe, le z različnimi scenariji, sem lahko opazoval v okviru študija prvega letnika Fakultete za računalništvo in informatiko. Na mojo veliko srečo, kot na vse skupaj gledam sedaj, z obeh strani: tako z vidika učenca, kot tudi učitelja, oz. bolje rečeno – asistenta. Naj najprej opišem prvega.

Po končani gimnaziji mi je bilo popolnoma jasno, da se bo moj pogled na računalništvo popolnoma spremenil. In seveda tudi programiranje. Vedel sem, da bo večinski del prvega letnika posvečen prav osnovam programiranja. Torej nadgradnji tistega »nesmisla«, ki smo se ga učili v srednji šoli. Vendar sem na vse skupaj takrat gledal že v nekoliko v drugačni luči. Vedel sem, da bo vse pridobljeno znanje del moje službe, ki me bo zelo verjetno spremljala še zelo dolgo. Zato sem se na prvo uro programiranja podal s skrajnim optimizmom in željo, da svoje neprijetno preteklo izkušnjo nadomestim z novo, tokrat pozitivno. Prvih nekaj ur je minilo brez posebnosti. Osvežil sem svoje preteklo znanje osnovnih programskih konceptov in ves nesmisel je kar naenkrat začel dobivati neko logiko. Počasi sem začel razumeti, kako »razmišlja« računalniški program in kakšen je dejanski pomen posameznih programskih konstruktov. Kljub prvotnemu navdušenju in naivnemu prepričanju, da sedaj jasno razumem vse, učenje ni potekalo tako, kot sem si sprva predstavljal. Iz preprostih osnov smo kar hitro prešli na bolj napredne in kompleksne programske strukture, ki jih v srednji šoli nismo

obravnavali. Če sedaj dobro pomislim, so se stvari začele zapletati najprej z večdimenzionalnimi tabelami in kasneje, predvsem s t.i. kazalci – betonskim zidom veličine študentov. Problemi, ki naj bi jih rešil s pomočjo računalniškega progama, so kar naenkrat postali drugorazredni. V prvi vrsti nisem razumel niti orodja, s katerim naj bi jih rešil. Minilo je kar nekaj časa, trde volje in predvsem ogromno praktičnih izkušenj, da sem počasi začel obvladovati tako programski jezik, njegovo sintakso in semantiko, kot tudi jasno razumel načine, kako lahko z njim rešim zastavljeni problem. Prav v živo se spominjam vikenda, ko mi je v glavi naredilo »klik«. Tudi če sem imel pred tem izoblikovano neko teoretično osnovo in mi je bilo popolnoma jasno, kako npr. napolnim večdimenzionalno tabelo, izvedem klic procedure po referenci ipd., nisem znal rešiti nalog, ki smo jih dobili od asistenta oz. profesorja. Po tistem vikendu pa sem znal, četudi kdaj narobe, stvari v večini primerov zapeljati v pravo smer. In kako zanimivo je kar naenkrat postalo programiranje…

Opisano izkušnjo (v prvi osebi) sem želel predstaviti predvsem zato, da bi dodobra razumeli kako razmišlja in reagira povprečen začetnik pri soočenju s svojim prvim programskim jezikom. Podobno je najbrž pri učenju vsake nove stvari. Najprej se pojavi neko nezaupanje oz. morebitni dvom o obvladovanju nepoznanega, sledi nekakšno začetno navdušenje, ki ga kar hitro zaduši poglobljeno poznavanje vsake podrobnosti obravnavanega in kasnejše postopno nabiranje izkušenj ter pozitiven občutek nečesa novega. Učenje programskega jezika je kot učenje katerega koli drugega tujega jezika. Potrebne so osnove, ki nas kasneje pripeljejo do vedno bolj kompleksne obravnave jezika. Četudi se nam zdijo prvotni koncepti in primeri, ki nam jih podajajo učitelji, na začetku nesmiselni, se na koncu izkaže, da prav z njihovo pomočjo hitro osvojimo osnovno znanje, na katerem lahko gradimo naprej.

Prav v vlogo učitelja-asistenta sem bil postavljen tudi sam. Študentom prvega letnika, tako univerzitetnega kot tudi visokošolskega študija, sem moral teoretično znanje programskega jezika Java, ki so ga pridobivali na rednih tedenskih predavanjih, dejansko prikazati še na praktičnih primerih. Odzivi na prve občutke ob soočenju s samostojnim programiranjem so bili v večini primerov podobni kot pri meni. Študenti so imeli v povprečju probleme že pri razumevanju prvih nekaj primerov: kakšna je razlika med osnovnimi podatkovnimi tipi, spremenljivkami in konstantami, kako sploh pognati program, kako v neko spremenljivko zapisati vrednost, kako dve vrednosti med seboj primerjati in večjo izmed njih izpisati na ekran, ipd. Bil sem presenečen, da so študenti »odpovedali« že pri najbolj osnovnih stvareh, ki so se meni osebno zdele popolnoma samoumevne – vsaj za nekoga, ki se je lotil študija računalništva. Nekaterim se je že samo obvladovanje orodja zdela težka stvar, kot npr. odpreti program Notepad, vanj zapisati preprost program, ga prevesti in ga na koncu še pognati. Kaj šele da bi razumeli osnovno strukturo programa, kako deluje in zakaj na takšen ali drugačen način z njim uspešno rešimo določen problem. Zanimive so mi bile predvsem reakcije popolnih začetnikov, če sem od njih želel izvedeti kaj za izvajajoči program pomeni določena preprosta sprememba: kaj se zgodi, če zamenjam vrstni red izvajanja FOR zanke, kaj pomeni zamenjava vrstnega reda vrednosti dveh spremenljivkam, zakaj ne morem v spremenljivko

shraniti vrednosti »3.0«, če spremenljivko podatkovnega tipa »double« spremenim v »int«, ki tudi hrani številčne vrednosti, ipd. Posamezni izrazi na obrazih in posledično izgovarjanje na začetniško neznanje so bili včasih prav zabavni, priznam.

Po skoraj triletni izkušnji poučevanja programskega jezika Java mi je popolnoma jasno, da si začetniki izjemno težko predstavljajo sam koncept programiranja in vzroke, zakaj ga sploh uporabljamo. Preden začnejo s študijem programiranje prevečkrat enačijo z nekimi grafičnimi simulacijami, matematičnimi obdelavami, sortirnimi algoritmi, če ne že skoraj raketno znanostjo. S takšnim prepričanjem se jim popolnoma razumljivo tudi najbolj preprosti »šolski« primeri, kot je npr. izpis trikotnikov zvezdic na ekran, polnjenje tabel, rekurzivni postopki ipd., zdijo popolnoma absurdni. Ne zavedajo se dejstva, da jih prav ti preprosti in mogoče kdaj res »neumni« primeri na koncu lahko pripeljejo do programa, ki bo resnično krmilil pogonske motorje rakete.

Osvojitev nekega osnovnega znanja še vedno ne pomeni obvladovanje celotnega problemskega področja. Poleg začetnikov-programerjev sem v treh letih imel priložnost na vajah učiti tudi študente, ki so po opravljenem predmetu iz osnov programiranja, študij nadaljevali s predmetom, v okviru katerega so spoznali še višje-nivojske programske konstrukte in se naučili še bolj zahtevnih tehnik programiranja. Kljub prvotno osvojenemu znanju, ki je bilo včasih celo vprašljivo, so imeli študenti še vedno velike težave pri razumevanju objektno usmerjenih konceptov programskega jezika Java. Probleme so že znali nekako rešiti, vendar je njihovo abstraktno razumevanje objektno usmerjene zasnove programa velikokrat šepalo. Enostavno je bil most med razumevanjem problema v realnem svetu in preslikavo oz. rešitvijo problema v obliki realizacije objektno usmerjenega programa, ki bo ta problem rešil, enostavno prevelik. Dokler se študenti niso dodobra poglobili v razumevanja koncepta razredov, objektov, dedovanja, vmesnikov ipd., nikakor nismo mogli pokazati zadovoljiv nivo znanja. Četudi so jim bile osnove programiranja bolj ali manj jasne, celotne slike niso razumeli. Včasih je bila slika še nekoliko slabša. Zaradi nerazumevanja tudi najbolj osnovnih konceptov strukture programa, npr. kako so razredi organizirani po različnih datotekah ali pa katere (osnovne) knjižnice moramo »uvoziti«, da bomo npr. lahko obdelovali nize znakov, so študentje enostavno obupali ali pa le s težavo nadaljevali. Podobno kot v navedenih dveh primerih so imeli študenti tudi velike težave pri sami predstavi koncepta dedovanja. Vprašanja, ki so mi jih pogosto zastavljali, so bila: »Kaj se zgodi, če nekdo deduje od nekoga?«, »Kako so ti razredi med seboj povezani?«, »Zakaj ta deduje od tega, ta pa od tega?«, »Kakšna je razlika med abstraktnim razredom in vmesnikom?«, ipd.

Trenutno sem zaposlen v podjetju, ki se med drugim ukvarja tudi z lastnim razvojem programske opreme. Pred dobrim mesecem sem bil skrajno presenečen, ko sem sodeloval pri razgovorih za novo delovno mesto programerja v objektno usmerjenem programskem jeziku C#. Izmed dvajsetih kandidatov smo jih cca. dvanajst najbolj resnih povabili na kratek pogovor. Glede na to, da so kandidati jasno vedeli za katero delovno mesto se potegujejo, je bil popolnoma nejasen njihov nivo zahtevanega znanja. Izmed vseh so le štirje poznali razliko

med objektom in razredom, trije pa le razliko med metodo objekta in metodo razreda. Očitno nekatere stvari res ostanejo skrivnost za vedno. Žalostno je le, da takšnih primerov ni malo. Še bolj sem bil presenečen pred nekaj dnevi, ko sem se pogovarjal z enim izmed naših najboljših programerjev. Ko je pogovor nanesel na vmesnike in abstraktne metode, je sodelavec omenil, da jih sicer zna uporabljati, vendar dodobra ne pozna njihovega pomena. Pač samo to, da jih mora implementirati oz. »povoziti«. Žalostno.

V času priprav za praktični del magistrske naloge sem najprej razmišljal o najbolj primernem okolju oz. orodju, s katerim bi lahko preizkusil priporočila in tehnike razvoja po MDD. Ko sem orodje enkrat našel (več o njem bom zapisal v naslednjem poglavju), sem bil postavljen pred dejstvo, da si moram izbrati še neko primerno problemsko področje, v okviru katerega bom lahko učinkovito zajel in modeliral vse koncepte obravnavane domene. Ker sem ravno v tistem času pisal seminarsko nalogo na temo transformacij med modeli, sem nekako začel razmišljati v tej smeri. Kljub vsemu pa nisem našel prave ideje, ki bi lahko resnično pomenila dodano vrednost k praktičnemu delu moje magistrske naloge. Dobra ideja iz tega področja se mi je posvetila neko popoldne, ko se je študent prvega letnika oglasil na mojih govorilnih urah in me prosil, če mu lahko še enkrat obrazložim nalogo, ki bi jo moral rešiti za domačo nalogo. Namen naloge je bil spoznati abstraktne razrede, abstraktne metode, koncept dedovanja, redefinicije abstraktnih metod itd. Študent mi je predložil primer iz predavanj, ki ga je za domačo nalogo moral dopolniti. Kljub vsej dokumentaciji, dobri razlagi profesorja na predavanjih ter praktičnem prikazu povedanega še na vajah si študent omenjenih konceptov nikakor ni predstavljal. Ko sva vse skupaj še enkrat ponovila in koncept razredov oz. objektov narisala še na papir po dolgem in počez, se je študentu končno posvetilo. Pa tudi meni – seveda, vsakemu na svoj način. Ljudje smo bitja, ki informacije obdelujemo na osnovi vizualne percepcije. Bolj kot percepcija odraža podobe in opise iz realnega sveta, lažje in hitreje obdelujemo informacije ter jih tudi bolj intuitivno razumemo. Zagotovo se je že vsakemu izmed nas kdaj pripetil primer, ko sogovornika nismo razumeli, dokler nam problema ni narisal na papir. Mogoče ravno zaradi tega študent ni razumel problematike, ker si je ni znal dobro predstavljati. Zelo preprosta risba mu je odprla pogled, ker je na problem začel gledati iz popolnoma drugega zornega kota.

Omenjeno spoznanje in dejstvo, da sem ravno takrat študiral o transformacijah med modeli, me je pripeljalo do tega, da sem začel razmišljati o izgradnji prototipa za modelno voden oz. usmerjan razvoj javanskih programov. S tem bi »ubil tri muhe na en mah«. Kot prvo bi preizkusil, kako dobro je eno izmed najboljših odprtokodnih okolij za MDD razvoj, osredotočil bi se na domensko področje, ki ga dobro poznam, poleg tega pa bi prototip lahko preizkusili še študenti sami in povedali, ali bi jim takšno orodje v prihodnosti lahko pomagalo pri samem programiranju oz. razumevanju samih konceptov programiranja. Pri tem so se takoj pojavila vprašanja, na katera nisem znal ali si celo upal odgovoriti:

- V kolikšni meri je smiselno modelirati obravnavano problemsko področje, da bi študenti z uporabo prototipa še vedno znali »programirati«, kot se to od njih pričakuje

v 1. letniku, poleg tega pa ravno toliko »modelirali«, da bi jim to koristilo pri programiranju?

- Ali bo izbrano modelirno okolje (orodje) pravo za ta primer oz. ali bom z njim lahko izrazil vse tisto, kar bi potreboval?
- Ali imam dovolj časa, da izdelam prototip in ga »preizkusim« na izbrani množici študentov?

Največji problem je bil v tem, da kljub najboljši izbiri modelirnega okolja, še vedno nisem točno vedel, kako daleč bom lahko prišel s tem okoljem. Eno je brati opise in vtise ljudi o nečem, drugo pa je to zadevo dejansko preizkusiti še na lastnem primeru. Preden sem sploh pomislil na možnost, da bi z okoljem res lahko razvil to, kar sem potreboval, sem naredil nek podoben, vendar dosti bolj enostaven primer-pilot. Za njegovo izdelavo sem potreboval dobrih 10 dni in kljub temu, da sem naletel na nekaj konkretnih »min«, mi je bilo počasi jasno, kakšne so omejitve in prednosti okolja oz. kaj z njim lahko pričakujem in kaj ne.

Na koncu sem se odločil, da je modelirno orodje prava izbira in da se z njim lahko lotim praktičnega dela magistrske naloge. Na vprašanje, do kolikšne mere naj v okviru naloge modeliram izbrano problemsko domeno, pa sem takrat že lahko v grobem odgovoril: v tolikšni meri, da si bo z modelirnimi elementi programer-začetnik lažje predstavljal objektno povezanost med razredi, koncepte dedovanja, implementacijo vmesnikov, ločevanje med abstraktnimi in dejanskimi metodami, notranje razrede, vključenost knjižnic v obravnavani program in organizacijo razredov po datotekah. Postopek programiranja bi načeloma lahko modeliral z diagramu zaporedja podobnim modelom, vendar to nikakor ni bil moj namen. Takšnega načina »programiranja« si enostavno nisem smel privoščiti. Študent 1. letnika mora še vedno znati programirati »po starem«, le način kako si predstavlja celotno zasnovo programa sem skušal poenostaviti (predvsem povezanost med razredi v primeru dedovanja). Moj namen tudi ni bil, da v okviru okolja zajamem celotno definicijo programske jezika Java, ker bi bil to za praktičen del magistrske naloge enostavno preobsežen zalogaj. To tudi ne bi bilo smotrno, ker bi že z dovolj manjšim primerom pokazal smiselnost razvoja po MDD oz. kakovost in zrelost izbranega okolja, da z njim obravnavano problemsko domeno mogoče modelirati do neke smiselne mere. Moja največja želja je bila: *izdelati prototip orodja za modelno voden razvoj javanskih programov tako dobro, da bodo študentje z njim lahko v celoti izdelali vse primere javanskih programov, ki jih bodo morali izdelati na predavanjih, vajah in katere bodo morali realizirati v okviru vseh domačih nalog, poleg tega pa zaradi njegove uporabe ne bodo prikrajšani pri samem »programiranju«*.

Ura je začela teči, imel sem dobre 4 mesece, da izdelam prototip in ga v začetku semestra predam izbrani množici študentov…

4. OPIS OKOLJA GME

V prejšnjih poglavjih smo navedli, da smo v okviru magistrske naloge izdelali prototip orodja za modelno voden razvoj javanskih programov, s pomočjo katerega smo želeli preizkusiti tisto, kar smo podrobneje raziskali v teoretičnem delu naloge, to je, koncepte, smernice, priporočila in tehnike razvoja po MDD. Pri tem smo se že na začetku želeli opreti na enega izmed najboljših obstoječih razvojnih okolij, ki omogočajo izgradnjo takšnih orodij. Po podrobnih raziskavah in prebranih člankih [10], [11], [9] in [5] smo se odločili, da nam bo za potrebe magistrske naloge najbolj ustrezalo okolje GME (Generic Modeling Environment), ki so ga razvili na inštitutu Software Integrated Systems univerze Vanderbilt v Nashvillu [26].

GME je modelirno okolje za izdelavo domensko specifičnih jezikov. Poimenovali smo ga »okolje« in ne »orodje«, čeprav bi bilo slednje poimenovanje morda bolj smiselno. Dejansko gre pri GME za programsko-modelirno »orodje«, s katerim izdelujemo nova (specifična) okolja, na osnovi katerih z domensko specifičnimi jeziki izdelujemo modele. Ker je rezultat magistrske naloge prototip konkretne programske rešitve, izoblikovane s pomočjo »orodja« GME, smo za njegovo interpretacijo rajši izbrali besedo »okolje«, da pri poimenovanjih ne bi prihajalo do zmede. V strokovnih člankih GME enačijo z orodjem, ker kot tako zagotovo je.

Glavna prednost okolja GME je v tem, da lahko z njim zajamemo sintaktične, semantične in predstavitvene informacije praktično kateregakoli domenskega področja. Pri zajemu informacij se odločamo, kaj točno želimo modelirati: na katere koncepte domene se bomo oprli pri izgradnji modela, katere relacije med njimi bomo pri tem morali upoštevati, kako bomo koncepte medsebojno organizirali in katere so tiste zakonitosti in omejitve, ki jih bomo pri modeliranju morali upoštevati. Poleg tega moramo natančno vedeti tudi to, kaj bomo z oblikovanimi modeli sploh počeli. Katere obdelave oz. analize bomo izvajali nad njimi in zakaj. Vsa takšna vprašanja in odgovore nanje moramo formalizirati že takoj na začetku, da se bomo v kasnejših fazah znali pravilno odločati. Končni rezultat zajema in obdelave informacij je t.i. paradigma domene, na podlagi katere gradimo modele. Drugače povedano: ko imamo enkrat definirano ustrezno paradigmo domene, jo lahko uporabimo kot orodje za modeliranje izbranega domenskega področja. In prav v tem je čar okolja GME: z istim okoljem, s katerim definiramo izbrano domeno, lahko takoj za tem gradimo modele, ki ustrezajo sintaktičnim, semantičnim in predstavitvenim zakonitostim te domene.

Ko imamo enkrat izoblikovano osnovno sliko, kako naj bi izgledala paradigma obravnavane domene, izdelamo ustrezen metamodel, s katerim formalno zajamemo semantična pravila domene. Z metamodelom najprej definiramo koncepte domene, ki jih bo mogoče obravnavati pri modeliranju. V najbolj grobem ločimo koncepte v obliki atomarnih delov in sestavov, ki lahko vsebujejo več različnih podsestavov in atomarnih delov. Nato določimo atribute, s katerimi jih opišemo in natančno predpišemo dovoljene relacije med njimi. Prav tako predvidimo različne perspektive oz. aspekte prikaza sestavov in atomarnih delov na ekranu, s čimer obravnavo problemov med postopkom modeliranja razbijemo na več medsebojno

povezanih in obvladljivih celot. Nad elementi metamodela določimo tudi potrebne omejitve oz. pravila, ki se jih bomo morali držati med izgradnjo modelov. Le-te definiramo s pomočjo razširjenega predikatnega jezika OCL, ki vsebuje določene specifike in posebnosti okolja GME. Več o jeziku OCL in načinih njegove uporabe bomo povedali v podpoglavju *4.2 – OCL omejitve*.

Po končani prvi iteraciji izgradnje metamodela skupaj s pripadajočimi OCL omejitvami lahko izoblikovano paradigmo uporabimo kot modelirno orodje za izgradnjo modelov. V ta namen osnovno GME okolje avtomatsko zgenerira domensko specifično GME okolje, ki ga, kot smo že povedali, v magistrski nalogi poimenujemo kar »orodje«, s katerim na osnovi (metamodela in z OCL omejitvami definiranega) domensko specifičnega jezika gradimo želene modele. Popravki ali spremembe nad metamodelom oz. posameznimi OCL omejitvami zahtevajo izvedbo ponovne razvojne iteracije, ki kot posledico povzroči ponovno generacijo nove različice domensko specifičnega GME okolja. Sedaj ni več težko ugotoviti, kakšen je v najbolj grobem pomenu dejanski rezultat praktičnega dela magistrske naloge. Z okoljem GME smo definirali posplošen metamodel *modelno usmerjene izgradnje javanskih programov*, nato pa smo isto okolje uporabili kot orodje za izgradnjo *modelov konkretnih javanskih programov*.

V praksi bi paradigmo domene morala skupaj oblikovati ekspert problemskega področja, ki natančno pozna obravnavano domeno in načrtovalec modelov, ki bo znal njegovo znanje zajeti z elementi okolja GME. Poleg tega mora načrtovalec modelov zajeto znanje eksperta domene predstaviti tako dobro, da bo ekspert domene na učinkovit način znal uporabiti rezultate njegovega dela – torej, probleme opisati (modelirati) z »njegovim« orodjem. Le na ta način bo ekspertu domene izoblikovani modelirni jezik resnično v pomoč pri njegovem delu. Da bi to dosegel, mora načrtovalec modelov modelirni jezik izoblikovati postopoma. Tako kot pri ostalih programskih rešitvah tudi paradigma dozori šele po nekajkratnem testiranju, povratnih razvojnih ciklih in dejanski uporabi v praksi. V primeru da načrtovalec modelov modelirnega jezika ne izdela dovolj dobro, kar se ponavadi pokaže že takoj po prvi razvojni iteraciji, saj si ekspert domene z modelirnim jezikom ne more dosti pomagati ali pa z njim ne more intuitivno izraziti želenega, morata oba ponovno izvesti revizijo jezika in predlagati izboljšave. Sledijo popravki in spremembe nad posameznimi koncepti metamodela, njihovimi atributi in povezavami ter aspekti, na osnovi katerih so koncepti domene in relacije med njimi vizualno predstavljeni ekspertu domene. Poleg tega je potrebno izvesti še prilagoditve nad posameznimi OCL omejitvami, ki še dodatno definirajo načine in pravila izgradnje modelov. Rezultat razvojne iteracije je nova različica orodja, ki gre nato lahko v postopek potrditve ekspertu domene.

Skladno s spremembami paradigme se razumljivo spreminjajo tudi modeli, ki jih gradimo na osnovi njenih pravil in zakonitosti. Koncept dodelave in dokončno izoblikovanje paradigme ter posledično vseh modelov, ki izhajajo iz posameznih vmesnih različic paradigme, lahko enačimo z različnimi razvojnimi stopnjami programske kode. Okolje GME ponuja nekakšen

repozitorij, v katerem imamo vse različice paradigme shranjene v zaledni podatkovni bazi. V vsakem trenutku lahko prikličemo poljubno različico paradigme in na njeni osnovi nadaljujemo z izgradnjo modelov. Popolnoma jasno je, da modeli neke različice paradigme ne bodo nujno ustrezali pripadajočemu metamodelu ali OCL omejitvam druge različice in bodo zato z vidika slednje obravnavani kot neustrezni ali napačni. Kljub vsemu pa orodje vsebuje napredne mehanizme samodejne prilagoditve modelov novejšim oz. starejšim različicam paradigme, če le razlike med eno in drugo različico niso prevelike ali celo nasprotujoče. V takšnih primerih prilagoditev modelov med posameznimi različicami paradigme niso mogoče.

Ko imamo enkrat izdelano »stabilno« različico paradigme, s katero je na enostaven in intuitiven način v večji meri mogoče modelirati želeno problemsko področje, lahko začnemo s postopkom izgradnje t.i. interpreterja. Slednjega si lahko predstavljamo kot poseben program, ki ga poženemo nad izoblikovanim modelom, da nad njim izvedemo določeno akcijo. Nekateri interpreterji so že privzeto (ang. out of the box) vključeni v osnovno namestitev okolja GME. Z njihovo pomočjo je npr. mogoče samodejno razvrstiti elemente metamodela na ekranu tako, da izgledajo »lepo«. Tako načrtovalcu modelov ni potrebno skrbeti za izgled metamodela, saj ga na njegovo zahtevo samodejno prilagodi (uredi) interpreter sam.

V naslednjih podpoglavjih bomo opisali glavne gradnike in načine uporabe okolja GME. V poglavju ki sledi, bomo podrobneje opisali paradigmo modelno vodenega razvoja javanskih programov, ki smo jo skupaj z izdelanim interpreterjem, s katerim je oblikovane modele paradigme mogoče transformirati v javansko programsko kodo, zaradi lažje predstavitve poimenovali kar »prototip orodja za modelno voden razvoj javanskih programov«. Naj ob tem samo omenimo, da bi opis celotnega okolja GME močno presegal meje magistrske naloge, poleg tega pa bi s tako podrobnim opisom zajeli tudi ogromno možnosti in elementov, ki jih pri oblikovanju paradigme sploh nismo potrebovali. Celotno specifikacijo si je mogoče ogledati na [28].

4.1 OSNOVNI GRADNIKI METAMODELIRANJA

Najprej si oglejmo posamezne modelirne elemente okolja GME, s katerimi oblikujemo (meta)modele domen problemskih področij. Nato se bomo osredotočili še na način realizacije OCL omejitev in opisali, zakaj v okolju GME potrebujemo interpeterje ter predstavili načine kako jih implementiramo v praski.

4.1.1 SESTAVI (ANG. MODELS)

Najbolj osnoven gradnik okolja GME je t.i. model. Z njim abstraktno predstavimo poljubno stvar ali pojem iz realnega sveta oz. nekega problemskega področja. Kaj točno predstavlja nek model je popolnoma odvisno od domene, katere vsebino želimo zajeti v okviru metamodela. Naj naštejemo nekaj primberov:

- Če želimo modelirati komunikacijo med posameznimi elementi v komunikacijskem omrežju, z GME gradnikom modela predstavimo usmerjevalnik omrežja, osebni računalnik, delovno postajo, tiskalnik, mrežno diskovno polje itd.
- Škatlo s čevlji, torbo, lonček za vlaganje, sesalec ali kitaro si lahko predstavljamo kot modele, s katerimi opišemo zasedenost delovnega prostora ali polic v njem.
- Z motornim zmajem, ultralahkim letalom, večmotornim letalom ali jadralnim letalom lahko v obliki modelov simuliramo zasedenost zračnega prostora nad nekim letališčem.

V smislu modeliranja si modele lahko predstavljamo kot nekaj, kar lahko »obdelujemo«. V nadaljevanju bomo model poimenovali kar z besedo »sestav«, da ne bo prihajalo do zmede z istoimenskim poimenovanjem z meta(modelom). Sestav ima v vsakem trenutku neko stanje, obnašanje in identiteto. Med vsemi gradniki metamodeliranja imajo sestavi najvidnejšo vlogo. Ostali gradniki so od njih odvisni ali pa so z njimi na nek način povezani. Sestavi so tudi najpomembnejši element v smislu obdelave celotnega (meta)modela. Na njih se v prvi vrsti nanašajo programi v obliki interpreterjev, ki preko njih dostopajo do ostalih gradnikov (meta)modela.

Različne paradigme lahko vsebujejo različne tipe sestavov:

- V primeru domene procesne obdelave signalov imamo lahko enostavne sestave, s katerimi modeliramo osnovne operatorje med signali in kompleksne sestave (sestavljene iz enostavnih ali kompleksnih sestavov), s katerimi modeliramo sestavljene operatorje med signali.
- V večprocesorski arhitekturi imamo lahko sestave, s katerimi predstavimo osnovna računska vozlišča ali sestave, ki vsebujejo kompleksno mrežo med seboj povezanih računskih vozlišč.

Sestav tipično vsebuje dele – ostale gradnike GME, vsebovane v sestavu. Lahko jih razdelimo v naslednje kategorije:

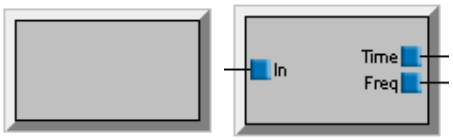
- atome oz. atomarne dele (ang. Atoms)
- druge sestave
- kazalce, ki si jih lahko predstavljamo kot »naslove« drugih objektov oz. delov (ang. References)
- množice (ang. Sets), ki niso nič drugega kot sestavi, le da vsak vsebovani element lahko vsebujejo natanko enkrat in
- povezave (ang. Connections), ki med seboj povezujejo modelirne gradnike

Če sestav vsebuje ostale gradnike rečemo, da je oče svojih gradnikov. Vsak del opišemo z množico lastnosti, ki jih imenujemo atributi. Vsak atomarni del vsebuje atribut, s katerim povemo, ali naj ga obravnavamo kot povezovalni del sestava (ang. Link part). Povezovalni

deli igrajo posebno vlogo pri medsebojnem povezovanju sestavov – lahko si jih predstavljamo kot konektorje povezav v obliki asociacije, odvisnih povezav ali katerih koli drugih vrst relacij med njimi. Preko povezovalnega dela vstopamo v določen sestav ali pa iz njega izstopamo. Sestave, ki vsebujejo druge sestave, imenujemo kompleksni sestavi (ang. Compound models). Sestave, ki vsebujejo druge gradnike, vendar ne drugih sestavov, pa imenujemo enostavni sestavi (ang. Primitive models).

V okolje GME vsak gradnik predstavimo z nekim simbolom. Sestav privzeto predstavimo s simbolom, ki je prikazan na spodnji sliki - levo. Če sestav vsebuje povezovalne dele, so le-ti prikazani na simbolu sestava (spodnja slika - desno). Privzeti simbol lahko nadomestimo z drugim (svojim) simbolom ali sliko, vendar pri tem izgubimo možnost prikaza povezovalnih delov sestava.

Kot poseben tip sestavov v okolju GME obravnavamo množice (ang. Sets). Ker jih v okviru praktičnega dela magistrske naloge nismo potrebovali, jih v nadaljevanju tudi ne bomo posebej opisovali.

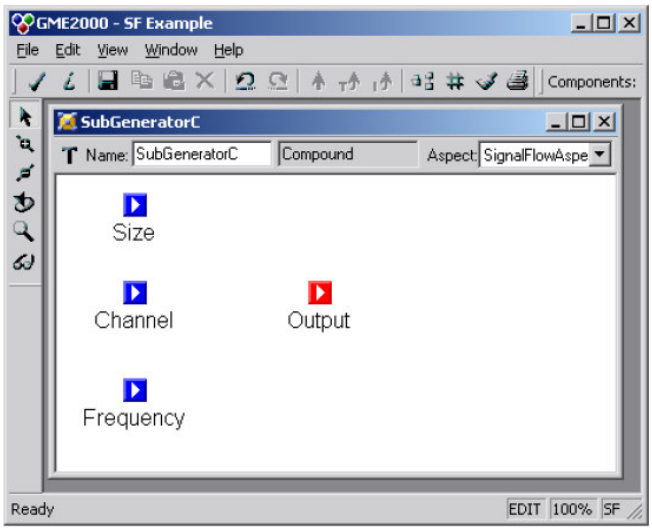


SLIKA 24: PRIVZETI SIMBOL SESTAVA (LEVO) IN SIMBOL SESTAVA S POVEZOVALNIMI DELI (DESNO)

4.1.2 ATOMI (ANG. ATOMS)

Atomi oz. atomarni deli so modelirni elementi, ki nimajo lastne interne strukture, jih pa lahko opišemo z množico enega ali večih atributov. Z atomi modeliramo homogene ali elementarne objekte iz izbranega problemskega področja, ki lahko obstajajo le pod okriljem nadrejenega očeta. V okolju GME to pomeni, da morajo biti atomi vedno vsebovani v vsaj v enem enostavnem ali kompleksnem sestavu, sami pa kot taki ne morejo obstajati. Slika 25 prikazuje privzeti simbol atoma, na sliki 26 pa je prikazan enostaven sestav SubGeneratorC, ki vsebuje štiri atome (atomi so dveh tipov, kot je razvidno iz simbolov modre in rdeče barve).

SLIKA 25: PRIVZETI SIMBOL ATOMA



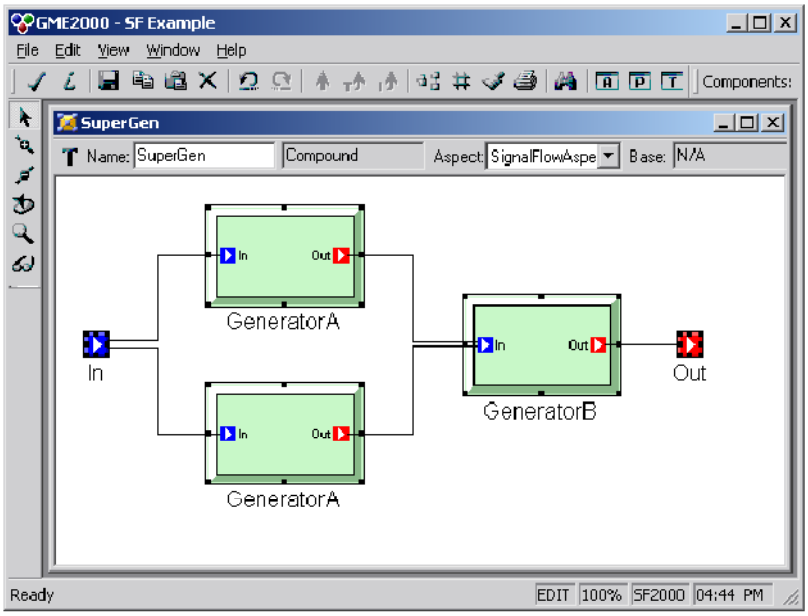
Slika 26: Enostaven sestav s štirimi povezovalnimi deli

Da si bomo lažje predstavljali, kaj lahko v praksi izrazimo z atomi, si oglejmo spodnje primere:

- Če govorimo o paradigmi opisa strojne opreme osebnega računalnika, lahko atomi predstavljajo povezovalne dele (pine) procesorja, s katerim le-te povežemo na matično ploščo.
- V primeru modeliranja delovnega toka proizvodnje lahko posamezen nastavitveni parameter procesa predstavimo z atomom.
- Z atomi lahko predstavimo tudi posamezne dele motorja, če želimo podrobneje opisati posamezne sestave avtomobila.

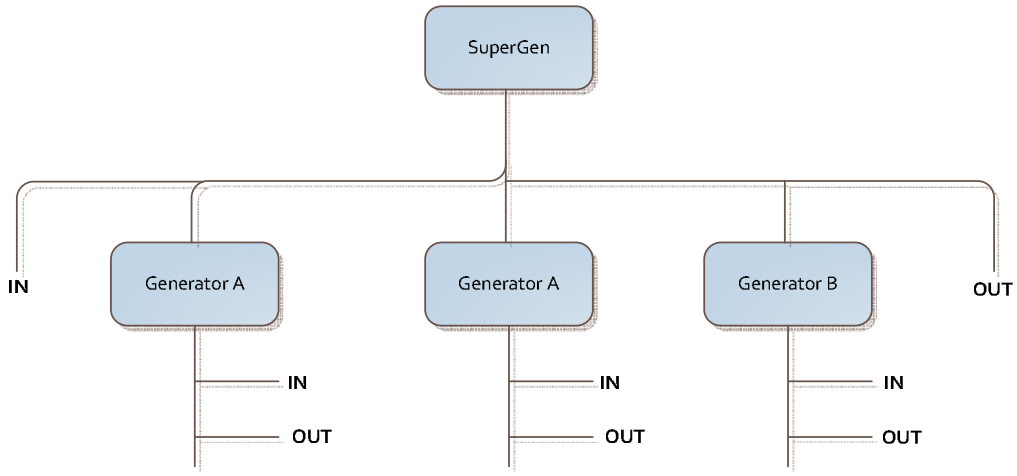
4.1.3 HIERARHIJA SESTAVOV (ANG. MODEL HIERARCHY)

Kot smo že povedali, lahko sestavi vsebujejo druge sestave kot svoje (neatomarne) dele. V takšnem primeru govorimo o hierarhiji sestavov, ki bi jo lahko predstavili v obliki drevesne strukture. Koren drevesa bi predstavljal sestav najvišje v hierarhični lestvici. Na prvem nivoju bi se nahajali listi drevesa, ki bi predstavljali posamezne atomarne dele sestava in sestavi, preko katerih bi sestopali na drugi hierarhični nivo drevesa. In tako naprej, dokler ne bi prišli do najnižje veje drevesa. Primer hierarhije sestavov prikazujeta sliki 27 in 28.



SLIKA 27: PRIMER HIERARHIJE SESTAVOV

Koncept hierarhije lahko razložimo z različnimi nivoji abstrakcije, ki jih želimo modelirati. Sestav, ki vsebuje druge sestave, predstavlja višji abstraktni nivo, kjer nas podrobnosti o »sestavnih delih« le-tega ne zanimajo ali pa jih želimo skriti. Na podoben način si lahko razlagamo najnižji abstraktni nivo, na katerem podrobno opisujemo vse elementarne oz. atomarne dele nekega enostavnega sestava. Na ta način omogočimo ekspertu domene, da se lahko z različnih nivojev kompleksnosti loti modeliranja problema. Če ga zanimajo samo sestavi nekega modela, si to »ogleda« z vidika najvišjih hierarhičnih nivojev. Globlje kot vrta v globino, bolj se mu razkrivajo podrobnosti in načini delovanja nekega konkretnega koncepta.



SLIKA 28: HIERARHIJA SESTAVOV PREDSTAVLJENA Z DREVESNO STRUKTURO

Primere hierarhične strukture modelov si poglejmo z naslednjima dvema alinejama:

- Če nas zanimajo samo osnovni deli avtomobila, to modeliramo s sestavi motorja, izpušnega sistema, elektronike, sistema krmiljenja itd. Če želimo podrobnejše informacije o posameznem sestavu, npr. motorju, vrtamo v globino sestava, kjer se nam razkrijejo informacije o glavni pogonski gredi, cilindrih, batih itn. Glede na želen nivo abstrakcije lahko tudi sam motor bolj kompleksno opišemo kot drevesno strukturo sestavov in atomarnih delov v malem.
- V primeru modeliranja komunikacijskega omrežja nekega podjetja bi na najvišjem hierarhičnem nivoju predstavili glavno komunikacijsko omrežje, sestavljeno iz posameznih sestavov – poddomen. Podrobnejše informacije o omrežju neke enote ali oddelka bi razkrili na najnižjih hierarhičnih nivojih, kjer bi med seboj lahko videli natančne povezave med omrežnimi usmerjevalniki, osebnimi računalniki, tiskalniki itn.

4.1.4 KAZALCI (ANG. REFERENCES)

Kazalci predstavljajo »bližnjice« do elementov modela. V celoti jih lahko enačimo s pojmom kazalcev v objektno usmerjenih programskih jezikih. Kadar modeliramo kompleksno problemsko področje, katerega sestave in atomarne dele opišemo z obsežnimi hierarhičnimi strukturami, moramo znati iz enega sestava poljubnega hierarhičnega nivoja dostopati do želenega atomarnega dela ali pod-sestava, ki se nahaja na nižjem oz. višjem hierarhičnem nivoju. Če kot primer vzamemo posplošen diagram podatkovnih tokov, kjer na nekem nivoju definiramo ponor podatkov, bi iz drugega višje-nivojskega diagrama do želenega ponora podatkov lahko dostopali le postopoma, preko ustreznih procesov (sestavov) in podatkovnih tokov (povezav med njimi). Kar pa je v obsežnih diagramih z večimi nivoji lahko dokaj zamuden in neroden postopek.

Okolje GME ponuja boljšo rešitev – kazalce. Kazalci so posebni modelirni elementi, s katerimi dostopamo do drugih elementov modela. Kot že samo ime pove, je njihova naloga zgolj »kazanje« in nič drugega. Lahko si jih predstavljamo kot bližnjice, s katerimi neposredno dostopamo do enostavnih ali kompleksnih sestavov, atomarnih delov in celo drugih kazalcev poljubnega hierarhičnega nivoja. Popolnoma logično je, da kazalci ne morejo obstajati brez elementov, na katere kažejo. Kar v praksi pomeni, da moramo vedno najprej imeti element, da lahko napravimo bližnjico do njega. V okolju GME lahko realiziramo tudi t.i. prazne bližnjice (ang. Null references). Njihov namen je vzpostaviti možnost nadaljnje širitve modela, če v času gradnje modela ne znamo ali ne želimo vzpostaviti strukture posameznih (pod)sestavov ali pa še ne vemo, kateri del bomo v konkretnem primeru potrebovali. Prazne bližnjice lahko kot take vključimo v model, njihovo vsebino oz. bolje rečeno definicijo dejanske poti do vsebine, pa določimo naknadno.

Privzeti simbol kazalca prikazuje slika 29.

SLIKA 29: PRIVZETI SIMBOL KAZALCA

4.1.5 POVEZAVE (ANG. CONNECTIONS)

Obstoj sestavov in njihovih delov ni dovolj, da opišemo model nekega sistema. Definirati moramo še pojem povezav, s katerimi opišemo relacije med njimi. Okolje GME pozna več načinov, s katerimi izrazimo relacije med elementi. Najenostavnejšo med njimi realiziramo preko t.i. povezave (ang. Connection). V času modeliranja si povezavo lahko predstavljamo kot črto ali linijo med dvema gradnikoma, ki ima določeni vsaj dve lastnosti oz. atributa:

- izgled povezave (Različni tipi povezav imajo drugačen izgled, da jih med seboj lažje prepoznamo. Ob definiciji povezave v metamodelu določimo ime tipa povezave, naziv ob izvoru povezave, naziv ob ponoru povezave, barvo povezave, način izrisa črte povezave, obliko izvora povezave in obliko ponora povezave.)
- smer povezave (Z definirano smerjo ugotovimo morebitni izvorni in ponorni element povezave. Dovoljene sta obe smeri ali pa le ena izmed njih.)

Poleg omenjenih dveh lastnosti lahko v definiciji metamodela za vsako povezavo definiramo še več različnih atributov, s katerimi natančno opišemo relacijo med obema udeleženima elementoma. Nabor potrebnih oz. zahtevanih atributov je v celoti odvisen od obravnavane domene.

V času modeliranja je dejanska semantika povezave med elementoma določena na osnovi konkretne paradigme. Ko v času modeliranja ekspert domene med seboj poveže dva elementa, se ustreznost povezave glede na njeno definicijo v metamodelu preveri v dveh korakih:

- Najprej se preveri, ali je povezava med obema tema tipoma elementov sploh dovoljena.
- Nato se preveri še dejansko pravilnost smeri povezave. Kot pravilno lahko določimo dvosmerno povezavo ali pa le enosmerno povezavo od elementa A do elementa B.
- Pravilnost povezave med elementoma se preveri tudi na osnovi definiranih OCL omejitev, ki jih bomo obravnavali nekoliko kasneje.

Če dodajmo povezavo med sestavom in nekim drugim elementom, se lahko povezavo vzpostavi z enim izmed njegovih povezovalnih delov. V ta namen se ob simbolu sestava pojavijo njegovi povezovalni deli (glej sliko 24), na katere se »priključi« izvorni oz. ponorni element na drugem koncu povezave. Kot smo že povedali, se lahko vrsta povezave glede na posamezni povezovalni del razlikuje skladno z definiranim metamodelom domene.

4.1.6 ATRIBUTI (ANG. ATTRIBUTES)

Sestave, atomarne dele, kazalce, množice in povezave med elementi lahko opišemo z enim ali več atributi oz. lastnostmi. Katere atribute bo določen element vseboval, definiramo z metamodelom domene. Če v okviru metamodela za nek tip sestava definiramo, da bomo za vsak njegov atomarni del hranili podatek o njegovi *barvi*, *obliki* in *materialu*, bomo pri modeliranju lahko rekli, da imamo škatlo, v kateri se nahajajo:

- *Steklena* (material) *kroglica* (oblika) *rdeče* (barva) barve
- B*ela* (barva) *papirnata* (material) škatla *pravokotne* oblike
- *Plastičen prozoren valj*

Za zgornji primer ni težko ugotoviti, da so *barva*, *oblika* in *material* definirani atributi vsakega dela sestava, ki jih moramo določiti za vsak primerek posebej. Kako veliko atributov bomo o nekem gradniku hranili, je v celoti odvisno od obravnavanega problemskega področja. Ena izmed pomembnejših nalog eksperta domene naj bi bila ravno v tem, da zna načrtovalec modelov za vsakega izmed vplivnejših konceptov domene povedati vse tiste lastnosti, ki bi jih pri modeliranju morali obravnavati.

V okolju GME obstaja tri vrste atributov, ki jih privzeto uporabljamo pri definiciji gradnikov metamodela:

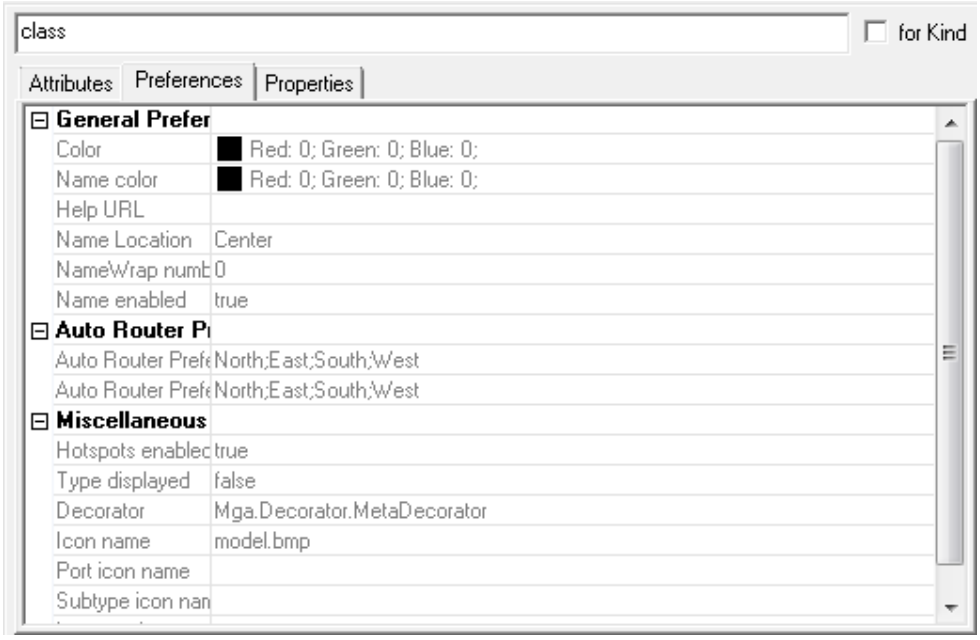
- Logični atribut (ang. Boolean attribute) z možnima naboroma vrednosti *true* ali *false*. Lahko določimo privzeto vrednost atributa in ga opišemo z nekaj »podpornimi« lastnostmi, kot je npr. kratka pomoč v obliki niza znakov, ki se v času modeliranja izpiše ekspertu domene, vidnost atributa itn. To vrsto atributa se uporablja takrat, ko želimo o nekem gradniku vedeti, ali vsebuje neko lastnost ali ne. Kot primer lahko navedemo atribut »Je žival«, z možnima vrednostima da/ne.
- Opisni atribut (ang. Field Attribute), ki mu določimo podatkovni tip (string, integer, double), privzeto vrednost in opcijo, ali lahko v atribut vnesemo enovrstične ali večvrstične vrednosti. Podobno kot logični atribut vsebuje nekaj podpornih lastnosti. Načeloma se to vrsto atributa uporablja najbolj pogosto, saj z njim na tekstualen način opišemo neko lastnost gradnika. Najbolj enostaven primer je atribut »lokacija«, z vrednostjo »Tržaška cesta 25, 1000 Ljubljana«.
- Opcijski atribut (ang. Enum Attribute), ki mu vnaprej definiramo nabor možnih vrednosti. Poleg tega eno izmed teh vrednosti določimo kot privzeto, atribut pa podobno kot pri ostalih dveh vrstah opišemo še nekaj dodatnimi podpornimi lastnostmi.

Pri metamodeliranju vsakega izmed obravnavanih gradnikov (sestav, atom, povezavo, kazalec in množico) opišemo z naborom posebnih atributov, imenovanim »nastavitve«

(»Preferences«). Nastavitve so prilagojene vsakemu gradniku posebej. Podedujejo jih iz definicije meta-metamodela okolja GME. Mednje sodijo lastnosti, kot so:

- Spletni naslov pomoči (ang. help URL), na katerem se nahaja opis elementa
- Barva elementa
- Simbol elementa
- Izpis pozicije naziva
- Itd.

Tipično okno z nastavitvami gradnika prikazuje slika 30.



SLIKA 30: TIPIČNE NASTAVITVE GRADNIKA OKOLJA GME

4.1.7 POGLEDI (ANG. ASPECTS)

S hierarhijo sestavov in tehniko »vrtanja v globino« dosežemo ustrezno kompleksnost obravnave informacij nekega problemskega področja. Nižje kot se spuščamo po hierarhični lestvici, bolj podrobne informacije dobimo o nekem konkretno obravnavanem elementu domene. Včasih bi zaradi še vedno prevelike kompleksnosti obravnavanega problema na nekem konkretnem hierarhičnem nivoju, nekatere informacije želeli skriti, bolj pomembne pa prikazati in jih jasno izpostaviti. V ta namen je v okolju GME vpeljan koncept pogledov, s katerimi skrivamo ali prikazujemo določene informacije (sestave, atomarne dele, povezave ipd.) o modelu. Bolj natančno, za vsak modelirni gradnik lahko definiramo en glavni in več pomožnih pogledov, v katerih ga prikazujemo skupaj z ostalimi gradniki – seveda le tistimi, ki nas v nekem trenutku zanimajo. En gradnik lahko prikažemo le v enem, lahko pa tudi v več različnih pogledih. Na ta način dosežemo to, da gradnik vedno obravnavamo le v okviru

želene vsebine. Glavni pogled gradnika je namenjen dodajanju oz. odvzemanju gradnika v/iz modela, medtem ko v pomožnih pogledih gradnik le povezujemo z ostalimi, mu določamo različne atribute itn. V vsakem trenutku je lahko aktiven le en pogled. Če pogled »preklopimo«, se trenutni pogled skrije (deaktivira), v novem (aktivnem) pa se pokažejo le tisti gradniki modela, ki so za ta pogled ustrezno definirani. Nekaterim gradnikom lahko v okviru posameznih pogledov, v katerih obravnavamo neko specifično vsebino, popolnoma spreminjamo izgled. Ne glede na to, da gre v vseh primerih za isti modelirni element, ga vsakič znova prilagodimo obravnavani vsebini in s tem dosežemo želeni vizualni efekt.

4.2 OCL OMEJITVE

Kot smo že povedali v prejšnjih poglavjih, z metamodeliranjem opišemo neko domensko področje in definiramo načine povezovanja med gradniki domene. Metamodele v okolju GME definiramo z UML razrednimi diagrami. Kljub temu, da so razredni diagrami zelo močna diagramska tehnika, s katero zelo dobro določimo omejitve in dovoljene zakonitosti med posameznimi objekti, nekaterih semantičnih omejitev med objekti nikakor ne moremo definirati. Če bi npr. želeli povedati, da povezavo med dvema objektoma lahko dodamo le v primeru, če ima prvi izmed obeh gradnikov pet sinov, med katerimi mora vsaj eden biti povezan še vsaj s petimi drugimi gradniki, poleg tega pa mora njegov atribut »barva«, ki smo ga definirali s pripadajočim metamodelom, biti »modra« ali »rdeča«, tega nikakor ne moremo opisati s statičnimi relacijami med objekti. Potrebujemo nek močan in dinamičen mehanizem, ki bo na enostaven način preveril želeno semantiko med objekti.

V ta namen je bil razvit predikatni omejitveni jezik OCL, katerega specifikacijo v različici 1.4 implementira tudi okolje GME. Z jezikom je na naraven in formalen način mogoče enostavno preveriti semantično vsebino objektov in njihovo medsebojno povezanost. Ker bi jezik z nekaj vaje moral biti razumljiv povprečnemu načrtovalcu modelov, ni potrebe bo strogem matematičnem znanju, ki bi se ga načeloma morali priučiti pri podobnih formalnih jezikih. OCL ni programski jezik, zato znotraj njegovih izrazov ne moremo pričakovati programskih blokov ali proceduralnih izrazov, kot smo to morebiti vajeni iz programskih jezikov 3. generacije. Vsak OCL izraz oz. poizvedba vedno vrne neko vrednost. Ne spreminja ali posega v vsebino modela, nad katerim izvajamo povpraševanje. To pomeni, da z OCL izrazom nikoli ne bomo spremenili stanja modela, ampak se z njim zgolj vprašali o določeni vsebini modela, ki nas trenutno zanima.

Konkretne primere realizacije OCL omejitev si bomo pogledali v naslednjem poglavju v okviru opisa OCL omejitev paradigme magistrske naloge. Več o sami definiciji jezika si je mogoče prebrati na [27].

4.3 INTERPRETERJI

Do sedaj smo modelirali le statični vidik nekega problemskega področja. S pomočjo sestavov, atomov, povezav, kazalcev in drugih gradnikov okolja GME smo zajeli vse bistvene koncepte domene, prikazali njihove korelacije in podrobno obravnavali ostalo statično semantiko domene. S pomočjo interpreterjev pa izrazimo tudi dinamično komponento vsebine, ki jo oblikujemo na osnovi oblikovanega modela in vseh z njim predstavljenih informacij. Interpreterji niso nič drugega kot računalniški programi, napisani v enem izmed programskih jezikov, s katerimi preko vmesnikov dostopamo do informacij modela. GME trenutno podpira vmesnike, realizirane v programskem jeziku Java in C++. Za potrebe magistrske naloge smo se odločili za programski jezik Java, predvsem zaradi njegove popularnosti v zadnjih nekaj letih in dejstva, da bomo jezik že tako ali drugače v smislu modelirne paradigme podrobno obravnavali v naslednjem poglavju.

Ko je interpreter enkrat izdelan, z njim »pridobimo« dostop do modela in iz njega »izvlečemo« želene informacije. S tako pridobljenimi podatki imamo praktično neomejeno možnosti, kaj želimo narediti. Naj navedemo samo nekaj najbolj zanimivih primerov:

- Podatke lahko iz izvorne (modelirne oblike) pretvorimo v poljubno ciljno obliko. Lahko si zamislimo XML strukturirano obliko dokumenta, ki ga zapišemo v ustrezno datoteko. Na ta način lahko model, ki smo ga modelirali v okolju GME, prenesemo v poljubno drugo predstavitveno obliko, ki jo nato kot izvor uporabimo v drugem orodju oz. okolju.
- Prebrane podatke lahko nekoliko preuredimo, dopolnimo in jih ponovno zapišemo v model. Tako lahko na generičen način spreminjamo vsebino modela. Če bi imeli npr. v podatkovni bazi zapisane potrebne dopolnilne podatke o nekem modelu, bi na ta način s pomočjo interpreterja že izoblikovani model semantično dopolnili s popolnoma drugega vira vsebine. Lahko si predstavljamo model komunikacijskega omrežja, ki ga je IT ekspert s standardnimi elementi omrežja oblikoval za neko konkretno podjetje. Programer je napisal interpreter, ki podatke o posameznih vozliščih omrežja (npr. IP naslove, tipe naprav, konkretne nastavitve naprav, kot so recimo nastavitve omrežnega tiskalnika, ipd.), ki jih je s pomočjo vmesnikov mogoče pridobiti iz posebnega IT nadzornega sistema, zapiše v ustrezne gradnike modela.
- Interpreter lahko napišemo tudi tako, da nam samodejno preoblikuje model, ki smo ga ročno modelirali sami. Na ta način lahko program zastavimo tako, da nam predlaga najboljšo razporeditev elementov modela, dopolni manjkajoče podatke atributov itn. Takšnega načina (programiranja) smo vajeni pri klasičnih programskih jezikih z orodji, ki nam npr. samodejno zamikajo vrstice, barvajo kodo, pripravijo definicijo metod ipd.

Pokazali smo, kako močan mehanizem za obdelavo modelov so lahko interpreterji. Kakšen interpreter bomo za konkretno paradigmo potrebovali, je popolnoma odvisno od

tega, zakaj smo se modeliranja sploh lotili že na samem začetku. Najbrž je popolnoma nesmiselno, da je model samemu sebi namen. Kot poudarjamo že tekom celotnega dela magistrske naloge, je bistvo MDD ravno v tem, da znamo z modeli oblikovati konkretne programske rešitve, ki so nam v pomoč pri izvajanju vsakodnevnih (delovnih) opravil.

V naslednjem poglavju bomo kot skupek oblikovane paradigme in namensko izdelanega interpreterja predstavili »prototip orodja za modelno voden razvoj javanskih programov«. S prototipom smo v okviru praktičnega dela magistrske naloge želeli preveriti moč odprtokodnega modelirnega okolja za modelno voden razvoj GME ter preveriti koncepte, tehnike in priporočila MDD v praksi.

5. PROTOTIP ORODJA ZA MODELNO VODEN RAZVOJ JAVANSKIH PROGRAMOV

V 3. poglavju magistrske naloge smo opisali želje in razloge za izdelavo prototipa orodja za modelno voden razvoj javanskih programov (v nadaljevanju ga bomo poimenovali kar »prototip«). Sklepna ugotovitev pri odločitvi izdelave je bila: *izdelati prototip orodja za modelno voden razvoj javanskih programov tako dobro, da bodo študentje z njim lahko v celoti izdelali vse primere javanskih programov, ki jih bodo obravnavali na predavanjih, vajah in katere bodo morali izdelati v okviru vseh domačih nalog, poleg tega pa zaradi njegove uporabe ne bodo prikrajšani pri samem »programiranju«*. Pred začetkom izdelave prototipa smo morali sprejeti nekaj konkretnih odločitev, da smo se sploh lahko lotili izdelave paradigme obravnavane domene. S prototipom smo želeli modelirati razvoj javanskih programov, vendar le do te mere, da študente pri tem ne bomo prikrajšali pri programiranju, torej pisanju programske kode na klasičen način. Vedeli smo, da se v sam tok programa ne bomo smeli spuščati, ker bi s tem ogrozili način obravnave programske kode. Poleg tega nam je bilo jasno, da zaradi samih omejitev modelirnega okolja GME ne bomo mogli v celoti realizirati prav vsega, kar bi si želeli. Morali smo najti neko srednjo pot, ki bi dovolj dobro pokrila vse zastavljene cilje.

5.1 ZAJEM ZAHTEV OBRAVNAVANE DOMENE

Kako oblikovano programsko kodo razdelimo na dva dela? Z enim delom na vizualen oz. grafičen način modeliramo želeno strukturo programa, z drugim pa na običajen način programiramo stvari po starem. Še pred začetkom izdelave prototipa smo dobro poznali okolje GME. V grobem nam je bilo jasno, kaj vse lahko pričakujemo od okolja in česa z njim ne bomo mogli realizirati. Sam sem bil v tistem času tudi asistent s področja programiranja, tako da sem zelo dobro vedel kako programirati v jeziku Java. Iz tega razloga lahko upravičeno trdimo, da smo se znašli v vlogah dveh različnih oseb: na eni strani eksperta domene, prav tako pa tudi nekakšnega načrtovalca modelov okolja GME. Kljub očitni prednosti, ki bi si jo v praksi želeli pri vsakem zajemu zahtev poljubnega domenskega področja, smo bili postavljeni pred vprašanja, s katerimi se vedno znova soočata oba - tako ekspert domene kot tudi načrtovalec modelov. Kaj iz domenskega področja bi želeli modelirati in na kakšen način? Kateri so tisti ključni koncepti domene, ki bi jih z modeli želeli obravnavati in kako naj jih na dovolj dober način vizualno predstavimo ekspertu-domene? Ker ekspert do potankosti razume le svojo domeno, od njega ne moremo pričakovati odgovorov, kako naj se vsebina domene odraža v svetu modelov. Glavna naloga načrtovalca modelov bi torej bila, da od njega pridobi ustrezne informacije in na njihovi osnovi začne s prvo iteracijo izgradnje metamodela obravnavane domene.

Pri sprejemanju odločitve, kateri so tisti ključni koncepti domene, ki bi jih v okviru modelno vodenega razvoja želeli modelirati, smo se zgledovali po [4, str. 42-45] oz. [8].

Najprej smo zapisali vse pojme, samostalnike, koncepte, trditve, vprašanja itn., ki jih lahko pripišemo obravnavanemu problemskemu področju:

- programska koda
- struktura programske kode
- datoteka
- razred
- notranji razred
- metoda
- dedovanje
- abstraktna metoda
- abstraktni razred
- parametri metode
- vmesnik (interface)
- kaj metoda vrača
- atribut
- knjižnica
- katere knjižnice so potrebne pri katerem razredu
- kako so razredi organizirani po datotekah
- kako razredi in vmesniki dedujejo med seboj
- podatkovni tipi
- vidljivost metode
- parametri metode
- vidljivost razreda
- določilo *static*
- konstante
- inicializacija metode
- inicializacija atributa
- vidljivost atributa
- določilo *final*
- generični tipi

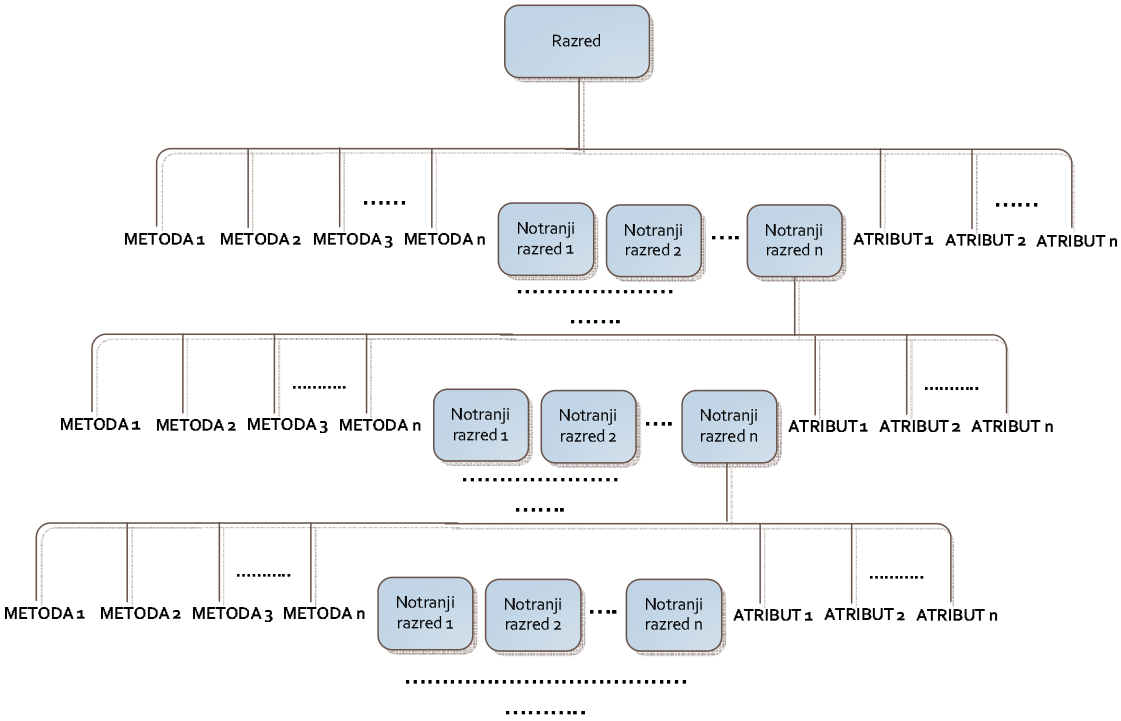
Nato smo spisek prečistili, združili določene sorodne vsebine in jih zajeli v spodnjo tabelo. V levem stolpcu smo poimenovali posamezne skupine elementov, ki predstavljajo sorodno vsebino, oz. bi jih lahko obravnavali v okviru neke skupne entitete. V desnem stolpcu smo navedli elemente skupine iz levega stolpca. Ker posameznih skupin med seboj nismo mogli obravnavati povsem neodvisno, smo določene elemente iz spiska vključili v več različnih skupin, predvsem tam, kjer se nam je to zdelo najbolj smiselno.

Skupina	Elementi skupine
Koda	• programska koda • struktura programske kode
Datoteke	• datoteka • »kako so razredi organizirani po datotekah«
Razredi	• razred • notranji razred • abstraktni razred • kako so razredi in vmesniki organizirani po datotekah • vidljivost razreda • določilo *final*
Metode	• metoda • abstraktna metoda • parametri metode • kaj metoda vrača • določilo *final* • določilo *static* • vidljivost metode • inicializacija metode
Atributi	• atribut • določilo *final* • določilo *static* • vidljivost atributa • inicializacija atributa • konstante
Dedovanje	• dedovanje • abstraktni razred • abstraktna metoda • vmesnik (interface) • kako razredi in vmesniki dedujejo med seboj
Knjižnice	• knjižnica • katere knjižnice so potrebne pri katerem razredu
Razno	• podatkovni tipi

	• generični tipi

Na podlagi tabele se je bilo potrebno odločiti, kaj bi bilo s prototipom sploh smiselno modelirati in na kakšen način. Glede na to, da programske kode jeder metod nismo imeli namena opisovati s kakšnim diagramu poteka podobnim modelom, smo za skupino »Koda« praktično že imeli rešitev. Telesa metod bi študenti še vedno morali obravnavati tako, kot so se učili na predavanjih oz. vajah. Na klasičen način. S tekstualnim vnosom programske kode. Brez kakršnegakoli modeliranja poteka korakov programa. Študentom bi vnos telesa metod zagotovili v okviru modelirnega elementa »Opisni atribut« (glej prejšnje poglavje), ki omogoča vnos vsebine poljubne dolžine. V veliko prednost si lahko štejemo možnost integracije kateregakoli zunanjega urejevalnika besedila v okolje GME, kot so npr. Notepad, Notepad++ ali JBuilder, s čimer bi študentom zagotovili vnos telesa metod z njihovim najljubšim urejevalnikom besedila.

Naslednji dve skupini, »Razredi« in »Dedovanje«, smo na nek način želeli povezati med seboj že od prvotne pojavitve ideje o izdelavi prototipa. Tekom izvajanja vaj iz programiranja smo ugotovili, da koncept objektno usmerjene predstavitve problemov študentom povzroča še največ težav. Želeli smo najti načine, kako bi jim te abstraktne gradnike objektno usmerjenega jezika Java predstavili na najbolj nazoren in intuitiven način. Z modelirnimi gradniki okolja GME bi statično strukturo javanskega progama lahko modelirali zelo enostavno. Ker si razrede lahko predstavljamo kot nekakšne sestavljene elemente, ki vsebujejo posamezne metode, atribute in morebitne notranje razrede, jih v okolju GME lahko enačimo s sestavi. Notranji razredi ponovno vsebujejo druge notranje razrede, atribute in metode. S tem lahko opišemo hierarhično strukturo kompleksnih sestavov, kot to prikazuje slika 31. Iz slike je razvidno, da bi metode in atribute lahko obravnavali kot atomarne dele, ki jih naprej ni več mogoče razdeliti na še manjše gradnike. Ker vsakega izmed modelirnih gradnikov lahko opišemo z naborom različnih lastnosti, lahko predpostavimo, da bi metode v okolju GME zagotovo opisali z njihovim najpomembnejšim atributom - telesom metode.

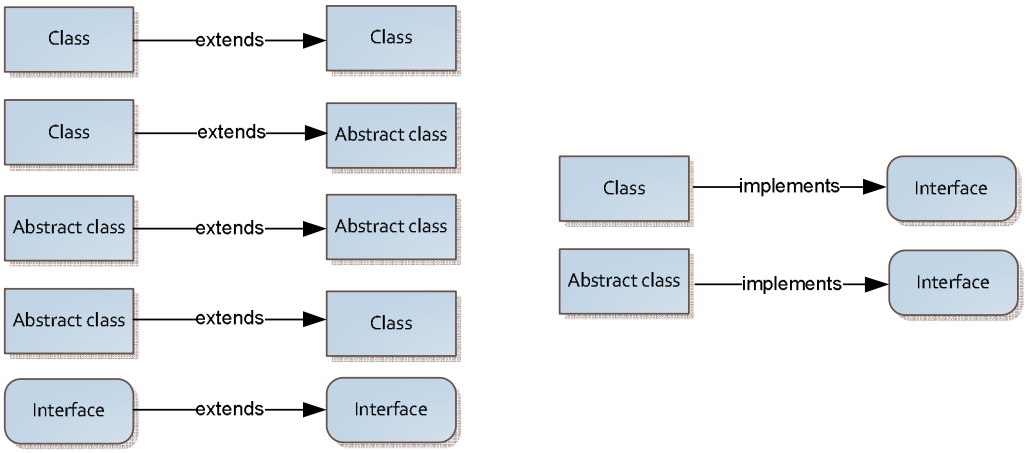


SLIKA 31: PRIKAZ HIERARHIJE RAZREDOV, ATRIBUTOV IN METOD

Ko smo enkrat določili kako obravnavati razrede, je bilo potrebno definirati način realizacije koncepta dedovanja. Ker smo razrede obravnavali kot sestave, bi njihovo medsebojno povezanost v smislu dedovanja razredov oz. implementacije vmesnikov lahko prikazali z različnimi tipi povezav med njimi. Na osnovi identificiranih elementov skupine »Dedovanje« smo določili naslednji različici povezav:

- *extends*
  - (abstraktni) razred razširja nek drug (abstraktni) razred
  - vmesnik razširja nek drug vmesnik
- *implements*
  - (abstraktni) razred implementira vmesnik

Oba tipa povezav bi lahko prikazali kar znotraj enega GME pogleda, s čimer bi študentu na enem mestu nazorno predstavili vse možne razširitve razredov, kot tudi implementacije vmesnikov. Tako bi med posameznimi razredi in vmesniki znotraj enega pogleda obstajale možne povezave, ki so prikazane na sliki 32.



SLIKA 32: MOŽNI NAČIN POVEZAV MED RAZREDI IN VMESNIKI

Ko smo metamodel domene, ki ga bomo podrobno obravnavali v naslednji točki, postopoma oblikovali na iterativno-inkrementalen način, kot je to predlagano v [5] in [28, str. 12], smo v določeni iteraciji naleteli na vrsto problemov, zaradi katerih smo morali način obravnave razredov, abstraktnih razredov, vmesnikov ter način povezovanja med njimi nekoliko dopolniti. Kot je razvidno iz slike 31, bi vse v modelu definirane razrede obravnavali kot sestave, kar pomeni, da imajo neko notranjo strukturo. V praksi bi v postopkih modeliranja to pomenilo, da bi za vsak razred lahko obravnavali vse njegove metode, atribute in notranje razrede - kar je načeloma popolno pravilno in logično. Težava se pojavi, ker moramo razlikovati med tistimi razredi, ki jih bomo modelirali (napisali) sami in tistimi, ki jih bomo v okviru modeliranega programa-modela le uporabili. Lep primer so razredi (npr. *AbstractCollection*, *JPanel*, idr.) in vmesniki (npr. *MouseListener* in *ActionListener*) standardnih knjižnic (*java.util*, *java.awt*, *javax.swing*, *java.awt.event*…), ki jih študentje uporabljajo pri večini svojih programov. Drug primer so razredi, ki smo jih nekoč že napisali in bi jih sedaj le želeli uporabiti, nimamo pa njihove izvorne kode. Imamo le prevedeno različico programske kode. Na vajah študenti velikokrat dobijo že prevedeno različico razreda *BranjePodatkov*, ki ga pri ostalih programih uporabljajo kot pomoč pri branju podatkov iz tipkovnice.

Jasni, vendar popolnoma različni, sta dve rešitvi problema:

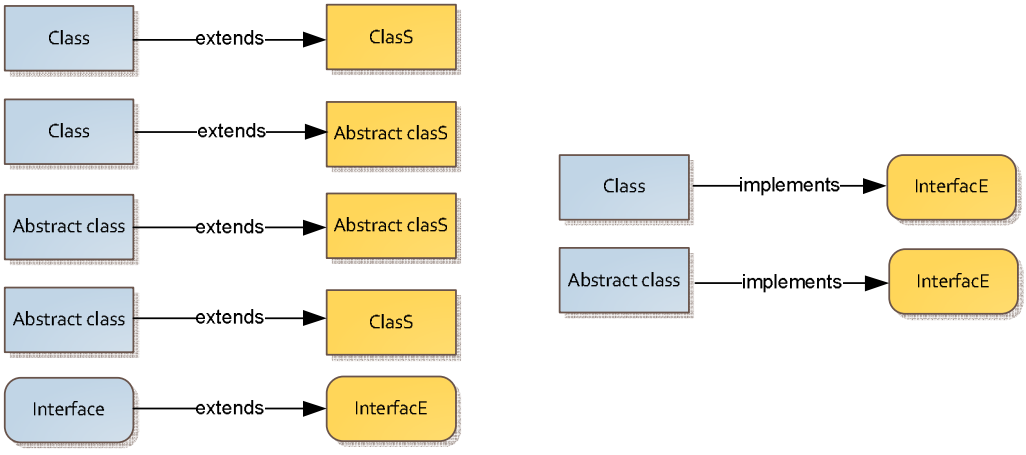
- Takšne razrede in vmesnike obravnavamo kot črne škatle, v katere nimamo vpogleda, poznamo pa njihov način obnašanja oz. delovanja in jih kot take lahko le uporabljamo.

- Vsebino teh razredov in vmesnikov »zaklenemo«, da imamo le vpogled v njihovo strukturo (npr. atribute razreda, glave metod - lahko tudi njihova telesa, notranje razrede itn.), ne pa tudi možnost njenega spreminjanja. Na noben način ne bi namreč smeli dovoliti, da nekdo na kakršenkoli način spreminja razrede, metode ali atribute že

prevedenih razredov, kot so npr. *String*, *Vector*, *JPanel* itn. Bi bilo pa seveda zelo dobrodošlo, če bi lahko njihovo vsebino videli v obliki med seboj povezanih modelov.

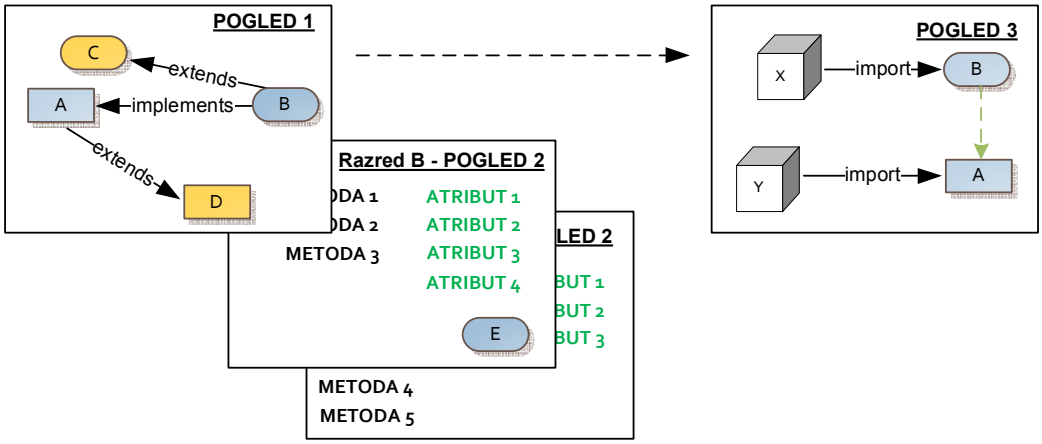
Druga rešitev bi sicer funkcionalnost prototipa močno izboljšala, ker bi se na ta način zavedali vsebine vseh že realiziranih razredov oz. vmesnikov, vendar je v obsegu magistrske naloge nikakor ne bi bili sposobni narediti. Če bi želeli uporabljati razrede, katerih izvorne kode nimamo, imamo pa prevedeno programsko kodo, bi morali izvesti t.i. »Reverse engineering» prevedenih *class* datotek, da bi dobili potrebne informacije o dejanski strukturi razreda. Težave pa bi se tukaj šele začele. Tekstualno obliko zapisane programske kode bi morali pretvoriti v obliko hierarhično povezanih modelov, za to pa bi morali napisati kompleksen program ali interpreter, ki bi izvedel potrebne transformacije programske kode v vizualne modele. Ker bi morali biti izgrajeni modeli semantično pravilni, bi morali metamodel domene oblikovati tako dobro, da bi program pri izvajanju preslikav upošteval in preverjal vse zakonitosti programskega jezika Java. Da bi tako kompleksen metamodel sploh lahko začeli oblikovati, bi potrebovali poglobljeno ekspertno znanje močne ekipe ljudi, s katerim bi bili sposobni zajeti in opisati celotno semantiko programskega jezika Java.

Iz zgornje razlage je najbrž popolnoma jasno, zakaj smo se odločili za prvi primer. Kljub enostavnosti predlaganega koncepta je realizacija rešitve pomenila delno spremembo metamodela in drugačno obravnavo razredov in vmesnikov, ki jih bomo v modelu le uporabili (v nadaljevanju jih bomo poimenovali »že realizirani« razredi oz. vmesniki). Poleg modelirnih gradnikov *Class*, *Abstract class* in *Interfance* smo vpeljali še atomarne gradnike *ClasS*, *Abstract clasS* in *InterfacE*. Dopolnitev slike 32 predstavlja slika 33. Novi atomarni gradniki so na sliki označeni z oranžno barvo. Veljavne povezave med opisanimi modelirnimi elementi so vse, ki so prikazane na obeh slikah.



SLIKA 33: MOŽNI NAČINI POVEZAV MED LASTNIMI IN ŽE IZDELANIMI RAZREDI OZ. VMESNIKI

Ko smo imeli enkrat razdelan način predstavitve razredov in določen način povezovanja med njimi, smo morali definirati še način modeliranja ostalih informacij razredov. Kot smo se že odločili, smo razrede in njihove *extends* oz. *implements* povezave predstavili znotraj enega GME pogleda. Notranje razrede, atribute in metode razreda pa kot sestave in atomarne dele znotraj drugega GME pogleda - en hierarhični nivo nižje. Za boljšo predstavo si oglejmo spodnjo skico.



SLIKA 34: KONCEPT POGLEDOV PROTOTIPA

Predstavljajmo si program, v okviru katerega želimo modelirati razred B. Razred implementira metode vmesnika A, ki ga bomo prav tako morali modelirati. Vmesnik A razširja vmesnik D (npr. *MouseListener*), ki je že realiziran v okviru standardne knjižnice Y (npr. *java.awt.event*). Razred B razširja abstraktni razred C (npr. *AbstractQueue*) standardne knjižnice X (npr. *java.util*). Vse povezave med omenjenimi razredi in vmesniki opišemo z modelom pogleda »POGLED 1«. Če želimo modelirati notranje razrede, metode in atribute razreda B, to naredimo v okviru pogleda »POGLED 2«, en hierarhični nivo nižje. Model na tem nivoju vsebuje atomarne metode (metoda1, metoda2 in metoda 3) in atribute (atribut 1, atribut 2, atribut 3 in atribut 4) razreda B, poleg tega pa še notranji sestav - razred E. Na naslednjem hierarhičnem nivoju na podoben način modeliramo metode in atribute razreda E, ponovno v okviru pogleda »POGLED 2«.

Na zgornji skici smo prikazali še model pogleda »POGLED 3«, ki ga do sedaj še nismo obravnavali. Iz tabele skupin elementov smo v okviru še enega pogleda želeli združiti skupini »Datoteke« in »Knjižnice«, saj smo bili prepričani, da bi koncept porazdelitve razredov po posameznih datotekah in koncept uvoza paketov v javanski program lahko modelirali na enem skupnem pogledu. V eni datoteki se lahko nahaja kvečjemu en sam razred z določilom *public*, ostali razredi znotraj iste *java* datoteke pa tega določila ne smejo imeti. Če se tega pravila ne držimo, prevajalnik javi napako. Prav tako znotraj ene datoteke navedemo vse javanske knjižnice, ki jih bomo potrebovali v okviru definiranih razredov datoteke. Na podlagi podobnosti obeh konceptov, v katerih znotraj vsake datoteke lahko obstaja le en

razred z določilom *public*, poleg tega pa vključenost javanskih knjižnic v datoteki navedemo samo enkrat, smo za »POGLED 3« definirali naslednja pravila:

- Znotraj pogleda je mogoče videti vse razrede in vmesnike modeliranega javanskega programa, ki smo jih predhodno določili znotraj pogleda »POGLED 1«. Vidni naj bodo le tisti razredi oz. vmesniki, ki jih bomo dejansko modelirali v okviru obravnavanega programa. Že realizirane razrede oz. vmesnike znotraj tega pogleda ni smiselno prikazovati. Nove razrede oz. vmesnike znotraj pogleda »POGLED 3« ni mogoče dodajati. To pomeni, da je z vidika razredov in vmesnikov »POGLED 1« definiran kot glavni pogled, »POGLED 3« pa kot edini pomožni pogled, ker lahko od tam sestave le opazujemo, ne moremo pa jih dodajati ali celo brisati.
- Usmerjena (na skici označena črtkana) povezava od razreda A do razreda B pomeni, da bosta v okviru ene datoteke obstajala oba razreda, pri čemer bo imel razred A v svoji definiciji navedeno določilo *public*, razred B pa tega določila ne bo imel. Ime datoteke naj bo enako imenu *public* razreda. Če med razredoma ali vmesnikoma ni določene povezave, vsak razred oz. vmesnik obstaja v svoji datoteki.
- V model moramo imeti še možnost dodajati modelirne elemente, ki predstavljajo že realizirane (standardne) knjižnice, kot so npr. *java.util*, *javax.swing* ipd.
- Usmerjena povezava od knjižnice do razreda oz. vmesnika pomeni, da se bo v datoteko razreda dodalo *import* definicijo navedene knjižnice oz. paketa. Povezave med knjižnicami in razredi oz. vmesniki brez določila *public*, naj ne bodo dovoljene.

Do tega trenutka smo za izbrane skupine elementov »Datoteke«, »Razredi«, »Metode«, »Dedovanje« in »Knjižnice« sprejeli bistvene odločitve glede vizualizacije njihovih elementov znotraj modeliranega javanskega programa. Glede na zmožnosti in omejitve modelirnega okolja GME, časa, ki smo ga imeli na voljo za izdelavo prototipa, ter jasno zastavljenih ciljev na začetku magistrske naloge menimo, da boljše porazdelitve javanskih konceptov med posameznimi pogledi okolja ne bi mogli doseči:

- Znotraj prvega pogleda na enostaven in pregleden način definiramo vse razrede in vmesnike, ki jih bomo potrebovali v okviru modeliranega javanskega programa. Nato s povezavami med njimi in povezavami z že realiziranimi razredi oz. vmesniki določimo koncepte dedovanja, ter tako na enem mestu prikažemo celovito shemo programa in medsebojna razmerja med njegovimi sestavnimi deli.
- Na drugem pogledu, ki se nahaja en nivo nižje po hierarhični lestvici, lahko vidimo notranjo strukturo izbranega razreda oz. vmesnika: njegove atribute, metode in morebitne notranje razrede. Vrtanje v globino še nižje po hierarhični lestvici je mogoče za vsak na novo definiran notranji razred, pri čemer njegovo strukturo ponovno prikažemo znotraj drugega pogleda.

- Na tretjem pogledu za vse razrede oz. vmesnike, definirane v okviru prvega pogleda, določimo, kako bodo razporejeni po posameznih *java* datotekah. Poleg tega znotraj istega pogleda modeliramo še uvoz knjižnic oz. paketov po teh datotekah.

Kljub podrobnemu opisu načina realizacije najbolj osnovnih modelirnih elementov paradigme, na tem mestu še vedno pogrešamo celovit opis slike problema, brez katerega si modeliranja javanskih programov niti ne moremo zamisliti:

- Nikjer nismo predvideli, kje razredom oz. vmesnikom definiramo določilo *final* in opredelimo morebitne generične tipe, notranjim razredom pa še definicijo določila *static* in njihovo vidljivost.
- Manjkajo definicije metod – kaj metode vračajo, kateri so njihovi parametri, kakšna je njihova vidljivost, ali imajo metode definirana določila *static* oz. *final*, kakšne napake metode »vržejo« in kar je najpomembnejše, nikjer nismo v metode vključili njihovega telesa – programske kode. Prav tako nikjer nismo obravnavali posebnega tipa metod, imenovanega *konstruktorji* razreda.
- Atributom nismo določili vidljivosti, podatkovnega tipa, določil *static* in *final*, prav tako jih nismo nikjer inicirali.

Vse navedene pomanjkljivosti bomo rešili v okviru posebnih elementov, imenovanih atributi modelirnih gradnikov, ki smo jih opisali že v prejšnjem poglavju. Vsakemu sestavu (razred, vmesnik) ali atomarnemu delu (atribut, metoda, paket, že realiziran razred oz. vmesnik) lahko v postopkih metamodeliranja določimo nabor atributov, s katerimi ga opišemo v celoti. Obravnavo atributov znotraj posameznega modelirnega elementa si bomo podrobneje ogledali v točki *5.3 Metamodel prototipa*. Skupaj z obravnavanimi OCL omejitvami, opisanimi v točki *5.4 Realizacija OCL omejitev,* bomo imeli v celoti definirano statično semantiko paradigme razvoja javanskih programov, s katero bomo realizirali prototip magistrske naloge.

V naslednji točki si bomo ogledali, kako smo za boljšo uporabniško izkušnjo vizualizirali posamezne modelirne gradnike, s katerimi smo predstavili koncept razreda, vmesnika, metode, abstraktne metode, konstruktorja, atributa, paketa ter že realiziranega razreda oz. vmesnika. Prav tako bomo predstavili način vizualizacije povezav med modelirnimi gradniki, s katerimi smo modelirali koncept dedovanja, uvoz standardiziranih knjižnic in porazdelitev razredov oz. vmesnikov po posameznih *java* datotekah.

5.2 IZBIRA GRAFIČNIH SIMBOLOV

Vsakemu modelirnemu elementu v okolju GME lahko določimo nek grafičen simbol, ki ga bo predstavljal v oblikovanem modelu. Že pred začetkom izdelave prototipa nam je bilo v grobem jasno, s kakšnimi simboli bi želeli predstaviti posamezne javanske koncepte. Dejstvo je, da si ljudje vizualne podobe zapomnimo dosti hitreje in bolj učinkovito od monotonih tekstualnih zapisov. Bolj kot so te podobe enostavne in enobarvno označene, hitreje si jih

človeški možgani »zapomnijo« in po potrebi prikličejo nazaj v zavest. Za primer si lahko predstavljamo 10.000 predmetov približno enake velikosti, ki jih naključno razmečemo po sobi. Najhitreje bomo našli tiste, ki so enobarvni in homogenih oblik, ker možgani najhitreje obdelajo preproste informacije. Prav tako bomo najhitreje razlikovali med posameznimi predmeti, če si med seboj ne bodo podobni. Zaradi omenjenih dejstev smo želeli gradnike modela javanskega programa predstaviti na tem bolj enostaven in nazoren način. Različne tipe smo med seboj sicer želeli jasno razlikovati, vendar pri tem nismo želeli biti preveč »agresivni«. Dovolj bi bila že enostavna vizualna karakteristika, s katero bi dosegli želeni učinek. V ta namen smo uporabili kombinacije dveh osnovnih geometrijskih oblik in treh homogenih barv: modre, rdeče in zelene, kot to prikazuje slika 35.

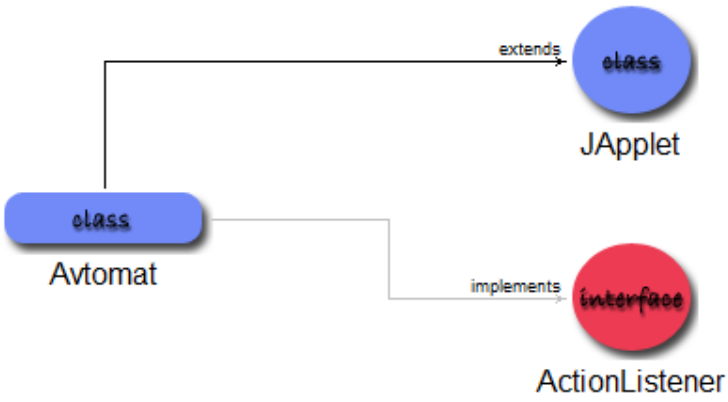
SLIKA 35: SIMBOL RAZREDA, ABSTRAKTNEGA RAZREDA IN VMESNIKA

Simbole razreda, abstraktnega razreda in vmesnika smo predstavili z obliko zaobljenega štirikotnika, pri čemer smo jih med seboj ločili z barvo. Poleg tega smo znotraj simbola navedli še naziv *class*, *abstract class* oz. *interface*. Glede na to, da gre v vseh treh primerih za razrede z določenimi specifikami, smo jih zaradi lažje predstave študentom prikazali z enotno geometrijsko obliko.

Že realizirane razrede oz. vmesnike smo označili z geometrijsko obliko kroga, pri čemer smo za vsak tip uporabili enake barve kot v zgornjem primeru. Poleg tega smo ime tipa znotraj simbola še prečrtali, da je razlika med zgornjimi razredi, abstraktnimi razredi in vmesniki še bolj očitna (glej sliko 36).

SLIKA 36: SIMBOL ŽE REALIZIRANEGA RAZREDA, ABSTRAKTNEGA RAZREDA IN VMESNIKA

Ko smo enkrat imeli oblikovane simbole osnovnih modelirnih gradnikov, smo glede na zmožnosti okolja GME lahko določili še različne prikaze povezav med gradniki. Povezave tipa *extends* smo prikazali s črno neprekinjeno črto, povezave tipa *implements* pa s sivo. Poleg tega smo nad vsako povezavo definirali tudi izpis naziva *extends* oz. *implements*, kot to prikazuje slika 37. Iz slike je razviden še primer ločevanja med razredom, ki ga bomo realizirali sami (razred *Avtomat*), in že realiziranim razredom (*JApplet*), ter vmesnikom (*ActionListener*) standardnih knjižnic.

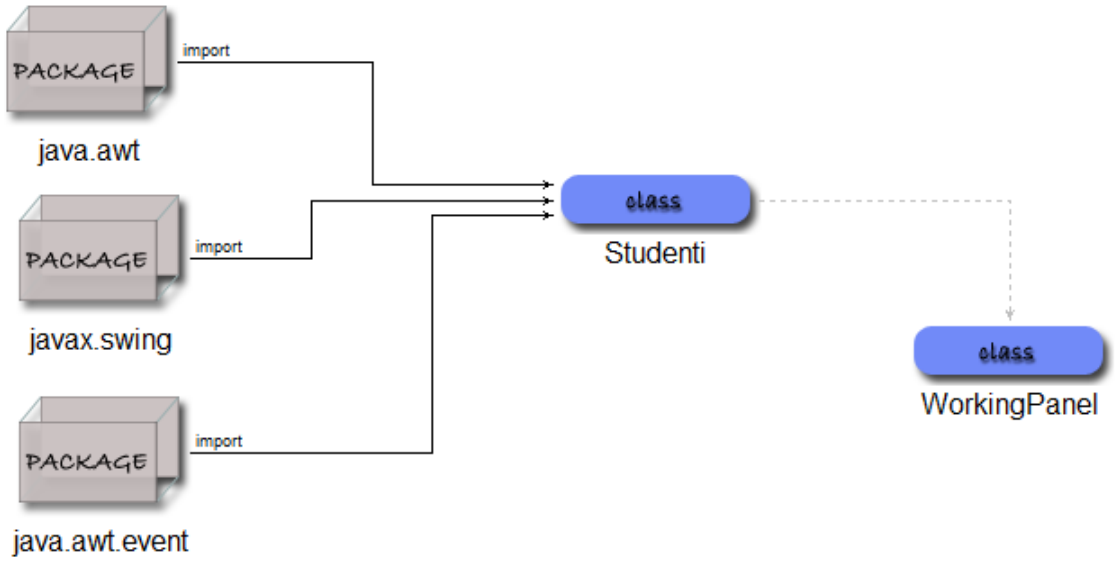


SLIKA 37: NAČIN IZPISA POVEZAV EXTENDS IN IMPLEMENTS

Metode in atribute razredov smo prikazali s simbolom pravokotnika, kot je to prikazano na sliki 38. Vizualna razlika med atributom in metodo je v tem, da ima atribut »odškrbljen« levi zgornji rob, prav tako pa je prikazan v rumeni barvi. Kot posebna tipa metod smo definirali konstruktor razreda in ga označili z rdečo barvo ter določili abstraktno metodo, ki smo jo označili z nekoliko temnejšo modro barvo.

Razlog, zakaj smo se tudi v primeru metod in atributov odločili za štirikotno obliko gradnikov, je v tem, da smo želeli doseči neko vizualno usklajenost med vsemi gradniki modela. Ker se gradniki vedno prikazujejo znotraj ločenih pogledov modela, ki jih poleg tega obravnavamo na drugem hierarhičnem nivoju modela, ni bojazni, da bi jih med seboj kakorkoli pomešali.

V okviru zadnjega pogleda paradigme smo morali določiti še simbole za standardne knjižnice oz. pakete in oblike povezav med gradniki. Kot simbol paketa smo izbrali preprosto obliko kvadra, z nazivom *PACKAGE*. Povezave med knjižnico in razredom oz. vmesnikom smo predstavili s preprosto črno neprekinjeno usmerjeno povezavo, nad katero smo določili izpis *import*. Povezavo, s katero določimo vključenost razreda v *java* datoteko želenega *public* razreda, smo nevpadljivo označili s sivo črtkano črto. Primer obeh povezav in način povezovanja med paketom ter razredom prikazuje spodnja slika.



SLIKA 38: SIMBOL PAKETA IN NAČIN IZPISA POVEZAV MED NAVEDENIMI GRADNIKI

5.3 METAMODEL PROTOTIPA

Na podlagi ugotovitev in sprejetih sklepov v prejšnjih dveh točkah bomo v tej točki opisali metamodel obravnavane domene, ki smo ga na iterativno-inkrementalen način oblikovali v procesu modelno usmerjene izdelave prototipa. Metamodel bomo v magistrski nalogi predstavili z razrednimi diagrami orodja *PowerDesigner*. Prvotne oblike metamodela iz okolja GME ne bomo prikazovali, ker bi bili v dokumentni obliki preveč nepregledni.

5.3.1 RAZRED, ABSTRAKTNI RAZRED IN VMESNIK

Sestave prototipa lahko predstavimo z razrednim diagramom, ki je prikazan v *Prilogi 8.3.1 – Sestavi javanskega programa*. Koncept javanskega razreda opišemo z abstraktnim razredom UML, ki mu določimo stereotip »model«. Abstraktni razred realizirata dva dejanska razreda, *class* in *innerClass*. Kot je že iz imena razvidno, razreda predstavljata javanski razred in javanski notranji razred. Kot dva ločena razreda smo ju definirali zato, ker ju bomo pri modeliranju obravnavali vsakega nekoliko drugače. Obema razredoma smo z določenim stereotipom pripisali lastnosti sestava.

Na podoben način smo opisali tudi koncept javanskega abstraktnega razreda in vmesnika, in sicer z definiranima razredoma *abstractClass* in *interface*. Na prvi pogled nepotrebna abstraktna razreda *abstractClassA* in *interfaceA* sta vpeljana zgolj zaradi morebitne kasnejše nadgradnje prototipa. Za potrebe magistrske naloge in zahtevanih domačih nalog, ki so jih v okviru rednega programa morali pripraviti študentje 1. letnika, ni bilo primerkov, v katerih bi bilo potrebno sprogramirati notranji abstraktni razred ali notranji vmesnik.

V *Prilogi 8.3.3 – Atributi modelirnih gradnikov* so definirani atributi opisanih sestavov. Vsakemu razredu določimo morebitni generični tip, ki ga definiramo z razredom *GenericniTip* stereotipa »FieldAttribute« (v nadaljevanju bomo razrede atributov večkrat poimenovali »atributi«). Poleg tega bomo za razred tudi povedali, ali vsebuje določilo *final*, kar je v metamodelu opisano z atributom *Final* stereotipa »EnumAttribute«. Nabor vrednosti atributa sta »/« in »da«. Ime razreda je podedovano že iz meta-metamodela okolja GME, zato te lastnosti razreda v metamodelu ni potrebno obravnavati posebej. Notranje razrede opišemo še z dvema dodatnima atributoma, *Vidljivost* in *Static*. Nabor vrednosti atributa *Vidljivost* so *private, public*, *protected* in *package access* (v programski kodi vidljivost ne bo eksplicitno navedena), atributa *Static* pa »/« in »da«.

Za lažjo predstavitev atributov razredov si poglejmo spodnje tri razrede:

```
public class LinkedQueueG<T> extends AbstractQueueG<T>{
…
…
}
public final class Test{
        public static void main(String[] args){
        }
        private class Notranji{
        }
}
```

Atributi zapisanih razredov bi vsebovali naslednje vrednosti:

Razred LinkedQueue

- Ime: *LinkedQueue*
- GenericniTip: *T*
- Final: */*

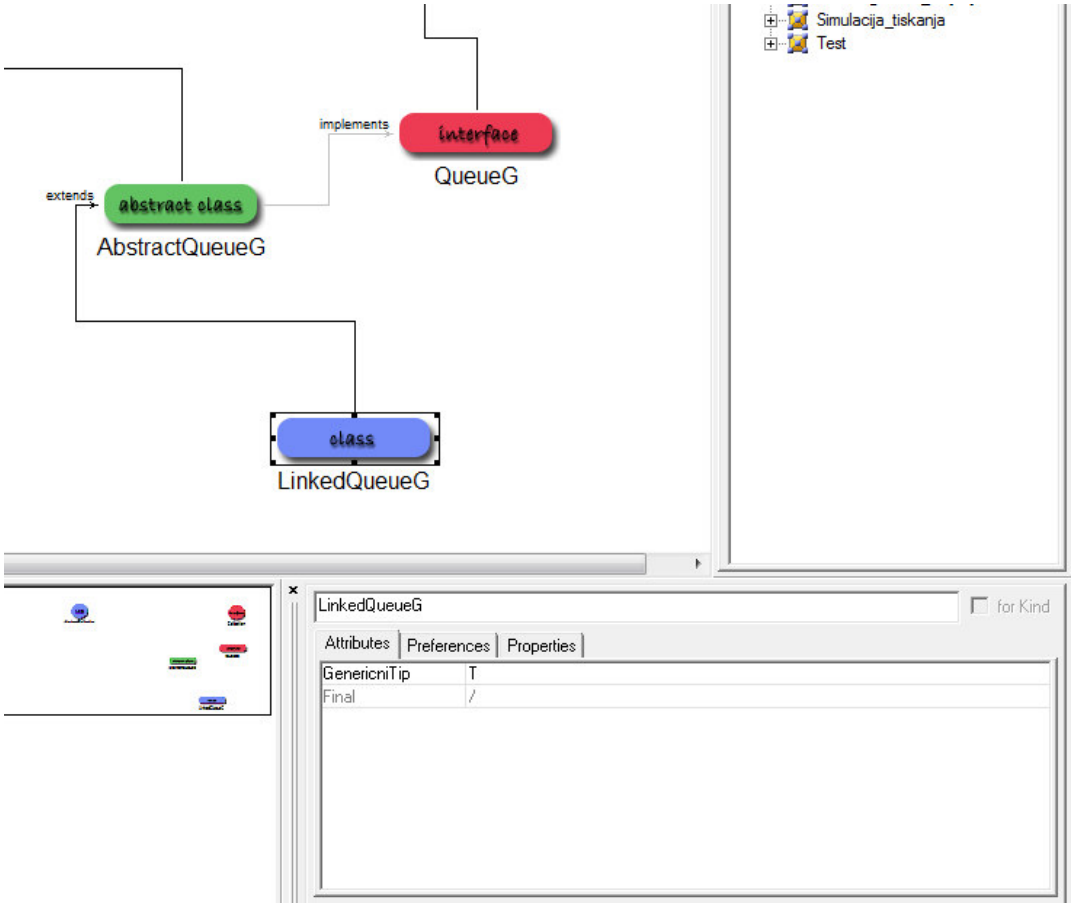
Razred Test:

- Ime: *Test*
- GenericniTip:
- Final: *da*

Notranji razred Notranji:

- Ime: *Notranji*
- GenericniTip:
- Final: */*
- Vidljivost: *private*
- Static: */*

Kako bi določitev atributov za razred *LinkedQueueG* izgledala v okolju GME, prikazuje slika 39.



Slika 39: Opis atributov razreda LinkedQueueG v okolju GME

V metamodelu smo za koncepta abstraktni razred in vmesnik, predstavljena z razredoma *abstractClass* in *interface,* definirali le atribut *GenericniTip*, ki ga oba podedujeta od svojih abstraktnih razredov *abstractClassA* in *interfaceA*. Podobno smo atribut *GenericniTip* pripisali že realiziranim razredom, abstraktnim razredom in vmesnikom, ki smo jih v metamodelu opisali z razredi clasS, abstractClasS in interfacE, kar je razvidno iz *Priloge 8.3.2 – Atomarni deli javanskega programa*. Modelirne gradnike za razliko od opisanih sestavov obravnavamo kot atomarne dele, kar v metamodelu definiramo s stereotipom razreda »Atom«. Kljub njihovemu bistvenemu razlikovanju s sestavi smo jih prav tako navedli na tem mestu, saj v njihovem primeru še vedno govorimo o razredih in vmesnikih, ki jih podrobneje obravnavamo v tej točki.

Z asociacijama kompozicije med razredoma *innerClass* in *abstractClassA* ter *innerClass* in *classA* povemo, da razred lahko vsebuje enega ali več notranjih razredov. V okolju GME navedena vsebovanost pomeni »vrtanje v globino« oz. sprehod nižje po hierarhični lestvici.

Na podoben način je realizirana tudi vsebovanost razreda, abstraktnega razreda in vmesnika v krovni model *Naloga*, ki si ga lahko predstavljamo kot prvi oz. nični model v hierarhični lestvici. Za lažjo predstavitev lahko rečemo, da je vsaka javanska naloga predstavljena v obliki programa, ki lahko vsebuje razrede, abstraktne razrede in vmesnike. Da bo v prejšnjem poglavju podrobno opisan koncept paradigme popoln, morajo biti z asociacijami kompozicije s krovnim sestavom *Naloga* povezani še atomarni elementi *clasS*, *abstractClasS* in *interfacE*, ker jih prav tako kot sestave obravnavamo znotraj modelirne javanske naloge *Naloga*.

5.3.2 METODA, ABSTRAKTNA METODA, KONSTRUKTOR IN ATRIBUT

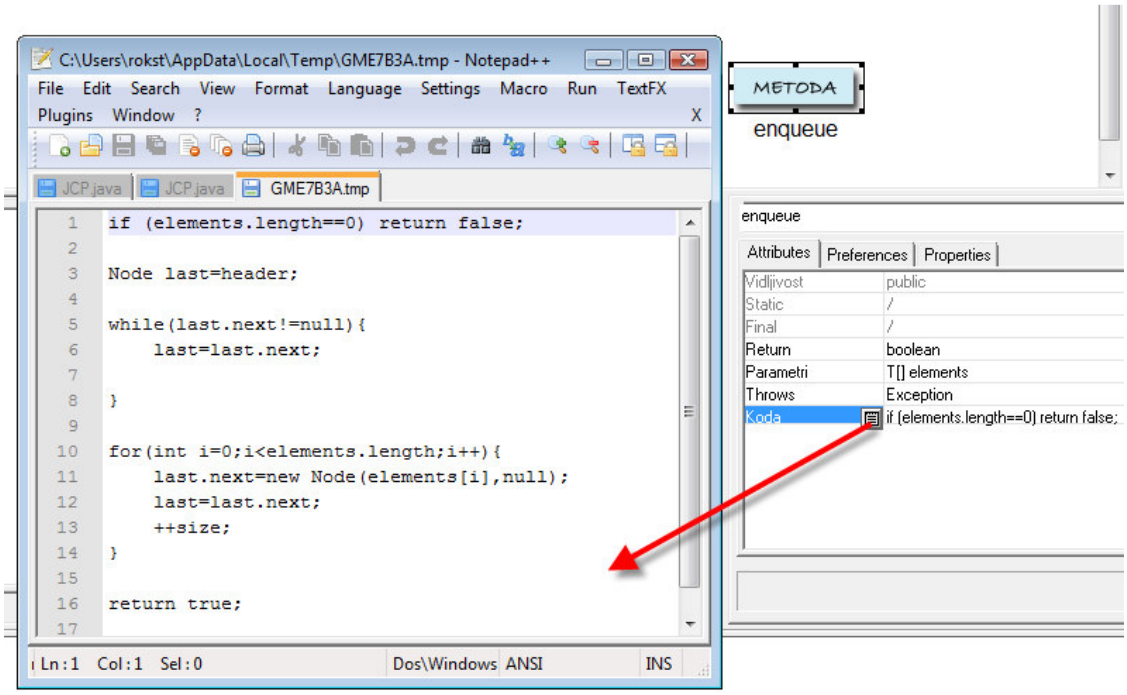
Koncepti metod, abstraktnih metod, konstruktorjev in atributov razredov so prikazani v *Prilogi 8.3.2 – Atomarni deli javanskega programa*. Konceptom z določenim stereotipom »Atom« pripišemo lastnosti atomarnih modelirnih gradnikov. Metode in abstraktne metode definiramo z razredoma *Metoda* in *AbstraktnaMetoda*. Oba razreda podedujeta splošne atribute abstraktnega razreda *MetodaA*. Atributa *Vidljivost* in *Static* smo že opisali v prejšnji točki. Atribut *Return* pove, kateri podatkovni tip metoda vrača. Z Atributom *Parametri* podamo parametre metode, z atributom *Throws* pa povemo, kako metoda obravnava napake.

Neabstraktne metode opišemo še z dvema atributoma: *Final* in *Koda*. Atribut *Koda* lahko obravnavamo kot najpomembnejšega izmed vseh lastnosti neabstraktnih metod, v okviru katerega podamo celotno programsko kodo metode. Ker okolje GME podpira integracijo poljubnega urejevalnika besedila, vnos podatkov v atribut samodejno odpre urejevalnik besedila, preko katerega na klasičen način vnesemo programsko kodo. Celoten nabor atributov atomarnih modelirnih gradnikov je prikazan v *Prilogi 8.3.3 – Atributi modelirnih gradnikov.*

Za lažjo predstavitev atributov metod si poglejmo primer na sliki 40. Na podlagi vrednosti atributov metode *enqueue*, ki so prikazani na sliki, bi s pomočjo interpreterja, podrobno ga bomo opisali v točki *5.5 Opis intepreterja*, zgenerirali naslednjo programsko kodo:

```
public boolean enqueue(T[] elements) throws Exception{
        if (elements.length==0) return false;
        Node last=header;

        while(last.next!=null){
                last=last.next;
        }
        for(int i=0;i<elements.length;i++){
                last.next=new Node(elements[i],null);
                last=last.next;
                ++size;
        }
        return true;
}
```

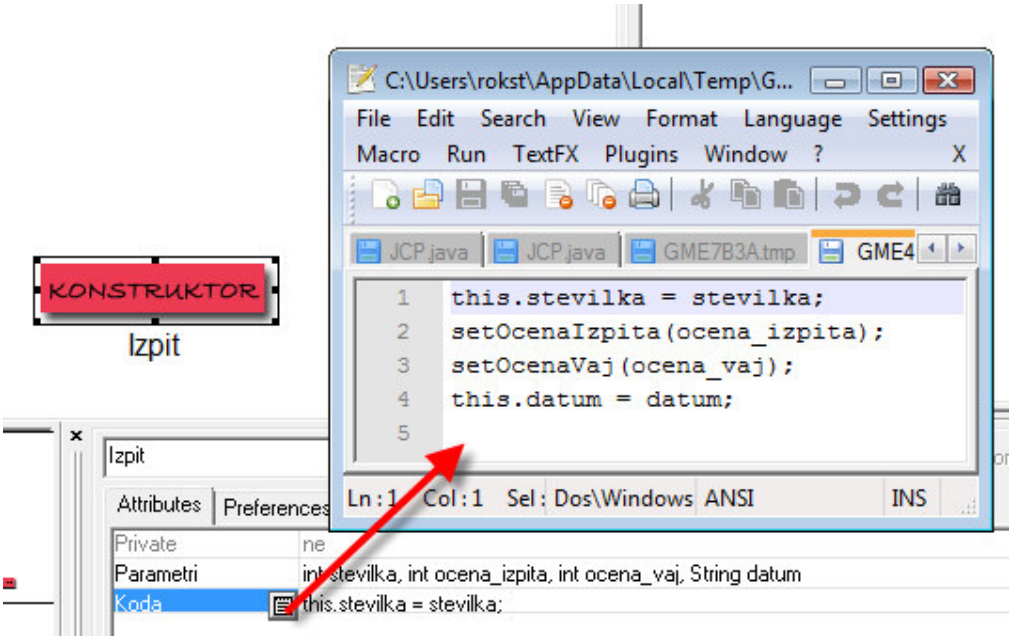
Na sliki je tudi prikazan način vnosa programske kode atributa *Koda* preko urejevalnika besedila Notepad++, ki smo ga kot privzetega določili v sistemskih nastavitvah okolja GME.



SLIKA 40: VREDNOSTI ATRIBUTOV METODE ENQUEUE

Konstruktorje razredov v metamodelu obravnavamo posebej, ločeno od opisa metod. Dejstvo, zakaj smo izbrali takšen pristop, je zaradi kasnejšega lažjega ločevanja metod in konstruktorjev v postopkih interpretacije modela, pa tudi metamodel je s takšno zasnovo bolj pregleden. Konstruktorje opišemo z atributoma *Parametri* in *Koda*. Poleg tega z atributom *Private* določimo še vidljivost konstruktorja – z možnima vrednostima »da« in »ne«. Za primer si poglejmo sliko 41. Definiranim vrednostim atributov konstruktorja enakovredno ustreza naslednja programska koda:

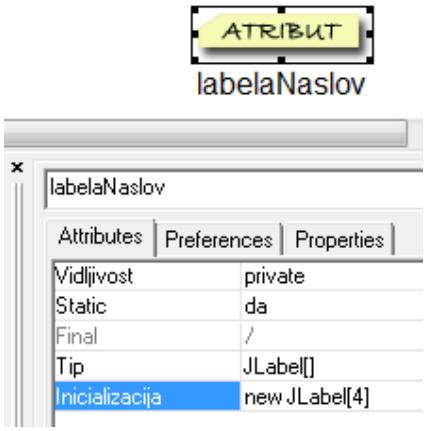
```
Izpit(int stevilka, int ocena_izpita, int ocena_vaj, String datum){
        this.stevilka = stevilka;
        setOcenaIzpita(ocena_izpita);
        setOcenaVaj(ocena_vaj);
        this.datum = datum;
}
```



SLIKA 41: VREDNOSTI ATRIBUTOV KONSTRUKTORJA IZPIT

Poleg metod znotraj razreda obravnavamo tudi njegove atribute. V metamodelu je koncept atributa opisan z lastnostmi *Vidljivost*, *Static*, *Final*, *Tip* in *Inicializacija*. Z lastnostjo *Tip* definiramo podatkovni tip atributa, kot je npr. *String*, *int*, *Razred* itn. Z atributom *Inicializacija* atribut iniciramo oz. mu ob definiciji določimo še začetno vrednost. Če je vrednost atributa prazna, atribut samo definiramo. Primer inicializacije atributa je prikazan na sliki 42. Generirana programska koda bi v tem primeru bila:

private static JLabel labelaNaslov = new JLabel[4];



SLIKA 42: DOLOČENE LASTNOSTI ATRIBUTA LABELANASLOV

Neabstraktne metode se lahko pojavljajo le znotraj razredov, notranjih razredov in abstraktnih razredov. Zaradi tega sta v metamodelu definirani asociaciji kompozicije med razredoma *Metoda* in *classA* ter *Metoda* in *abstractClassA*. Abstraktne metode se lahko pojavljajo le znotraj abstraktnih razredov in znotraj vmesnikov. Zaradi tega sta v metamodelu definirani še dve asociaciji kompozicije in sicer med razredoma *AbstraktnaMetoda* in *interfaceA* ter *AbstraktnaMetoda* in *abstractClassA*. Poleg metod lahko v razredih, abstraktnih razredih in vmesnikih modeliramo tudi atribute. V metamodelu smo to označili s tremi asociacijami kompozicije med razredi *Atribut*, *classA*, *abstractClassA* in *interfaceA*.

5.3.3 DEDOVANJE

Koncept dedovanja je realiziran s pomočjo dveh povezav - *extends* in *implements*. Ker se obe povezavi lahko pojavljata le med modelirnimi gradniki znotraj krovnega sestava *Naloga*, smo v metamodelu določili asociaciji kompozicije med sestavom *Naloga* in razredom *Extends* ter *Naloga* in razredom *Implements*, kot je prikazano v *Prilogi 8.3.4 – Povezave med modelirnimi gradniki*. Oba razreda smo označili s stereotipom »Connection«, kar pomeni, da koncepta povezave ne moremo uvrščati niti med sestave niti med atomarne modelirne gradnike.

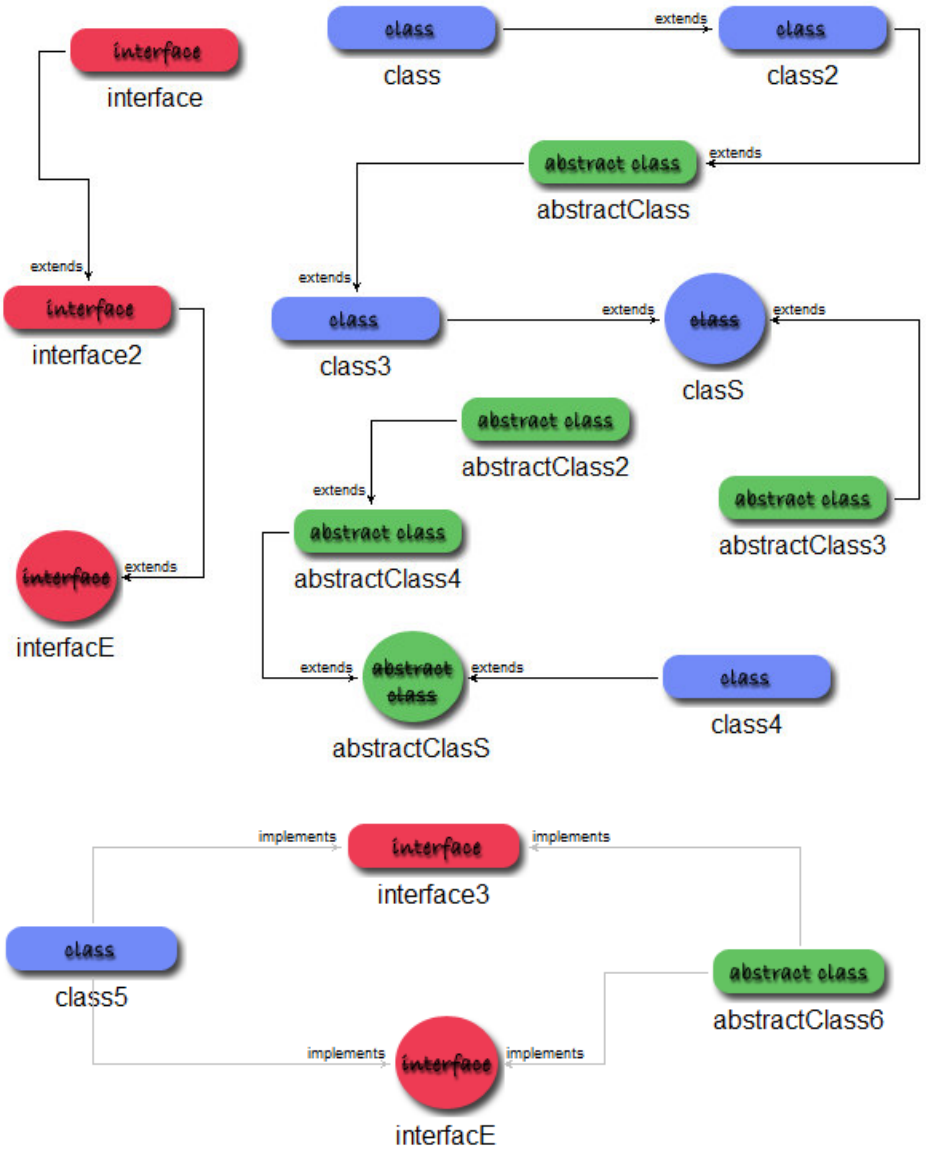
Dedovanje med razredi in vmesniki, v smislu razširjanja nekega razreda ali vmesnika z drugim razredom oz. vmesnikom, lahko opišemo na način, kot je definiran v metamodelu *Priloge 8.3.5 – Povezava – extends*:

- Razred lahko razširja nek drug razred, kar je v metamodelu označeno z rekurzivno povezavo razreda *class*. Povezavo realizira razred asociacije *Extends*. Namesto tega lahko razred razširja nek abstraktni razred (povezava med *class* in *abstractClass*), že realizirani razred (povezava med *class* in *clasS*) ali abstraktni razred (povezava med *class* in *abstractClasS*).
- Na podoben način abstraktni razred razširja nek drug abstraktni razred, kar v metamodelu označimo z rekurzivno povezavo razreda abstractClass. Prav tako lahko razširja razred (povezava med abstract*Class* in *class*), že realizirani abstraktni razred (povezava med *abstractClass* in *abstractClasS*) ali razred (povezava med *abstractClass* in *clasS*).
- Vmesnik lahko razširja bodisi nek drug vmesnik bodisi že realizirani vmesnik. Obe povezavi sta v metamodelu prikazani med razredoma *interface* in *interfacE*. Vmesniki razredov oz. abstraktih razredov ne morejo razširjati, zato povezave med temi koncepti v metamodelu niso definirane.

V *Prilogi 8.3.6 – Povezava – implements* je na podoben način prikazana še realizacija implementacije vmesnikov:

- Razred lahko implementira bodisi enega ali več vmesnikov bodisi enega ali več že realiziranih vmesnikov, kar označimo s povezavama med razredi *class* in *interface* ter *class* in *interfacE*. Povezavo realizira razred asociacije *Implements*.
- Prav tako lahko abstraktni razred implementira enega ali več vmesnikov oz. enega ali več že realiziranih vmesnikov, kar označimo s povezavama med razredi *class* in *interface* ter *class* in *interfacE*.

Vse možne opisane *extends* in *implements* povezave med modelirnimi gradniki prikazuje spodnja slika.

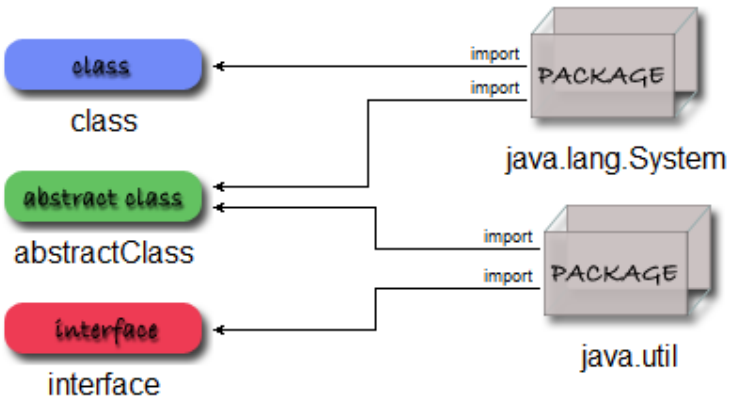


SLIKA 43: MOŽNE POVEZAVE DEDOVANJA MED RAZREDI IN VMESNIKI

5.3.4 UVOZI PAKETOV V DATOTEKE

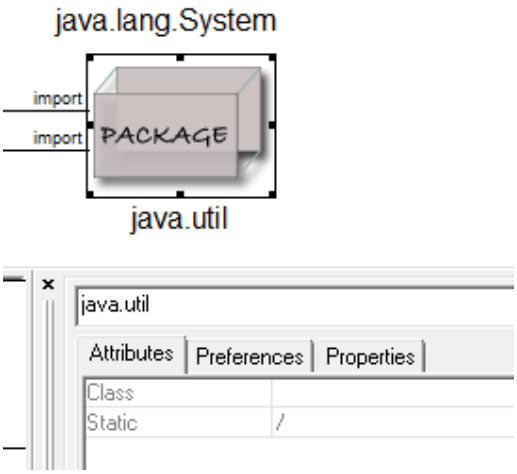
Vsak javanski program temelji na uporabi že realiziranih gradnikov (razredov, abstraktnih razredov in vmesnikov), kot smo to že prikazali v prejšnjih točkah. Skupino med seboj povezanih gradnikov iste vsebine imenujemo paket. Ker si kompleksnejših javanskih programov sploh ne moremo omisliti brez uporabe funkcionalnosti že realiziranih knjižnic, kot sta npr. standardni knjižnici *java.util* in *java.lang.System*, je bilo v metamodel paradigme potrebno vključiti še koncept uvoza paketov. Kot smo že nakazali v prejšnjem poglavju, uvoz paketov v razrede, abstraktne razrede in vmesnike modeliramo znotraj istega pogleda, v katerem prikazujemo tudi razvrščenost teh gradnikov po posameznih datotekah. Ker smo model uvrstili na prvi hierarhični nivo, torej en nivo nižje od že obravnavanega krovnega sestava *Naloga*, je v metamodelu definirana asociacija kompozicije med sestavom *Naloga* in atomarnim elementom *Package*. Povezava kompozicije je prikazana v *Prilogi 8.3.2 – Atomarni deli javanskega programa*. Ker smo uvoz paketa v nek razred, abstraktni razred oz. vmesnik modelirali z usmerjeno povezavo med paketom in enim izmed modelirnih sestavov, je bilo v metamodelu potrebno definirati tudi koncept t.i. povezave *import*. Povezavo v obliki razreda *Import* stereotipa »Connection« smo ravno tako kot modelirni gradnik *Package* z asociacijo kompozicije vključili v krovni sestav *Naloga*. Povezava je prikazana v *Prilogi 8.3.4 – Povezave med modelirnimi gradniki.*

Formalno pravilne povezave med sestavi paradigme in obravnavanim atomarnim gradnikom *Package* smo definirali z delom metamodela, ki je prikazan v *Prilogi 8.3.7 – Povezava - import.* V obliki končnega modela so dovoljene *import* povezave prikazane na sliki 44.



SLIKA 44: VKLJUČENOST KNJIŽNIC V RAZREDE IN VMESNIKE

Za konec moramo še povedati, katere so lastnosti vsakega paketa. Iz *Priloge 8.3.3 – Atributi modelirnih gradnikov* je razvidno, da je vsak paket opisan z dvema atributoma, *Class* in *Static*. Z atributom *Class* navedemo uvoz posameznega razreda iz paketa, z atributom *Static* (vrednosti »/« in »da«) pa prevajalniku povemo, ali bi radi uvozili le statične elemente razreda. Praktičen primer določitve vrednosti omenjenih atributov je prikazan na sliki 45.



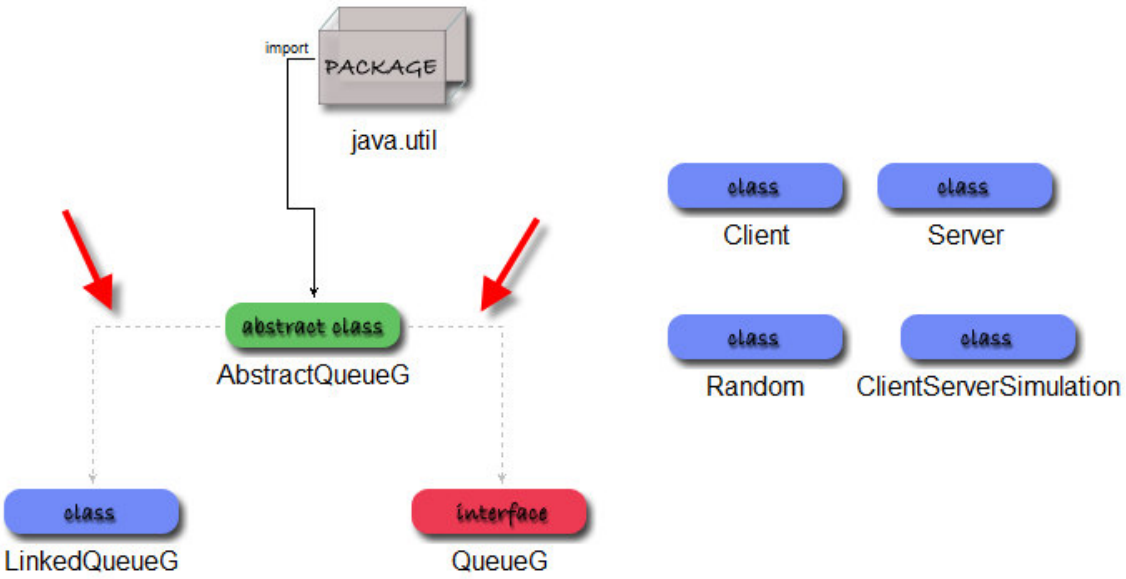
SLIKA 45: PRIKAZ ATRIBUTOV PAKETA

5.3.5 RAZVRŠČENOST RAZREDOV PO DATOTEKAH

Razrede, abstraktne razrede in vmesnike lahko razvrstimo v več različnih datotek. Znotraj ene datoteke ima lahko le eden izmed njih določilo *public*, ostali pa tega določila nimajo. V končnem modelu to definiramo z usmerjeno črtkano povezavo od sestava, ki naj ima določilo *public*, do sestava, ki bo v isto datoteko vključen brez določila *public*. Če pogledamo primer na sliki 46 lahko ugotovimo naslednje:

- Razredi *Client*, *Server*, *Random* in *ClientServerSimulation* bodo vsi z določilom *public* vključeni v svojo *java* datoteko z istim imenom, kot je ime razreda.
- Abstraktni razred *AbstractQueue* se bo z določilom *public* nahajal v istoimenski *java* datoteki, poleg tega pa se bosta v tej datoteki nahajala še razred *LinkedQueue* in vmesnik *QueueG*, oba brez določila.
- Med drugim bomo v okviru istega modela s povezavo *import* med paketom *java.util* in razredom *AbstractQueue* modelirali še uvoz standardne knjižnice v to datoteko.

Povezavo, ki določenemu razredu priredi določilo *public*, smo v metamodelu označili z razredom *Public* stereotipa »Connection«. Podobno kot vse druge povezave, smo tudi to povezavo z asociacijo kompozicije povezali s krovnim sestavom *Naloga*, kot je prikazano v *Prilogi 8.3.4 – Povezave med modelirnimi gradniki.* Definirane možne relacije povezave *Public* med posameznimi razredi, abstraktnimi razredi in vmesniki so navedene v *Prilogi 8.3.8 – Povezava – public.*



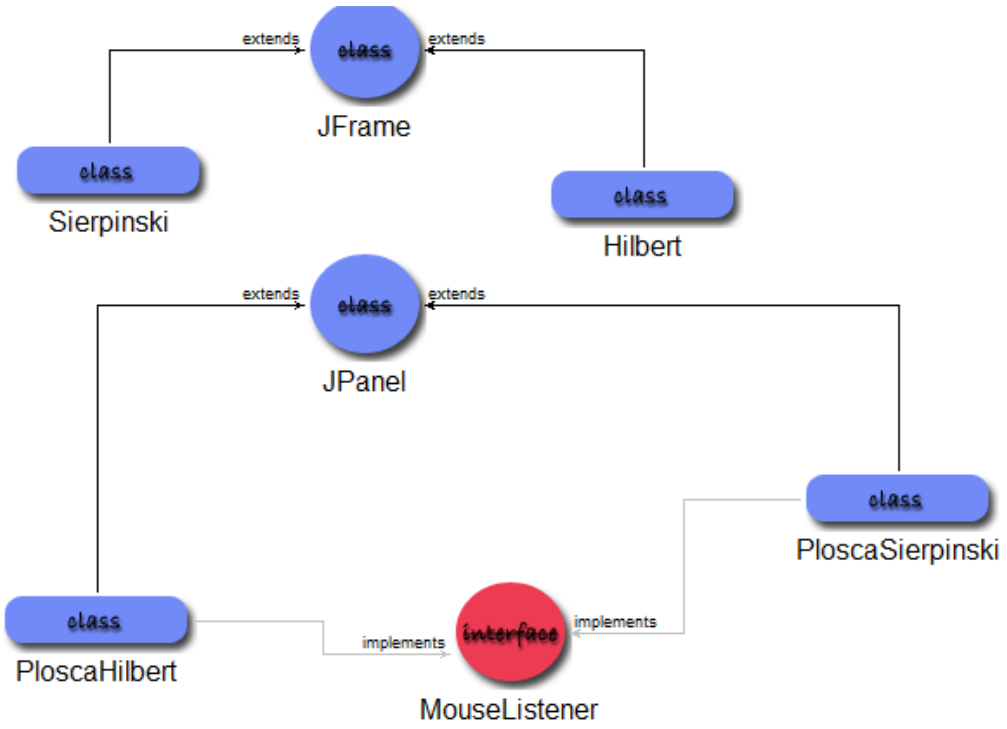
SLIKA 46: PRIMER RAZVRSTITVE RAZREDOV IN VMESNIKA PO JAVA DATOTEKAH

5.3.6 POGLEDI PROTOTIPA

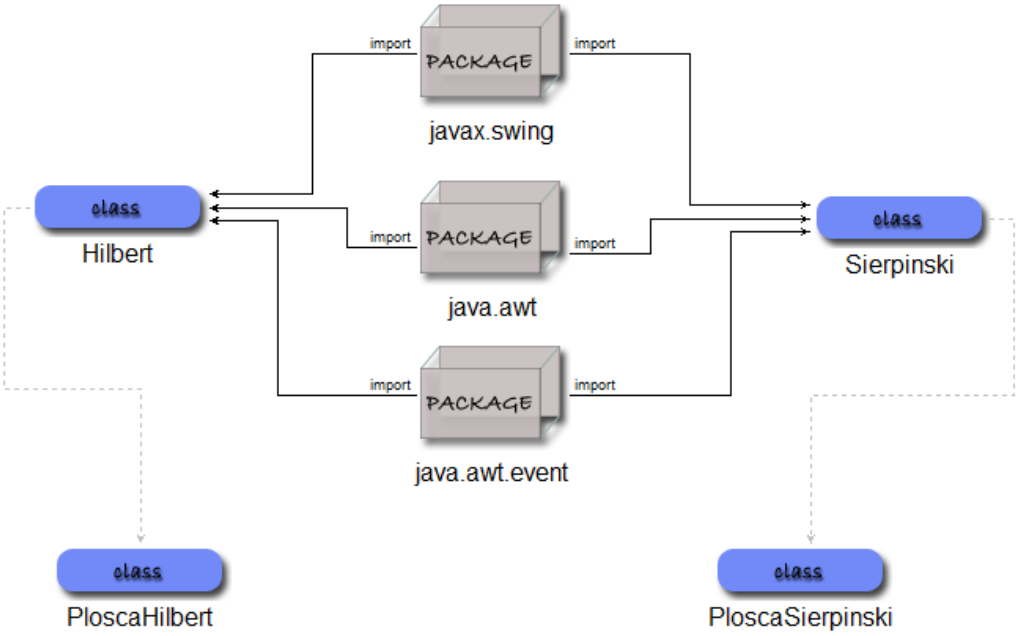
V definicijo metamodela smo zajeli tudi način predstavitve modelirnih gradnikov paradigme. Kot smo navedli že v prejšnjem poglavju, smo v okviru prototipa želeli obravnavati tri različne poglede:

- »POGLED 1«, ki smo ga v metamodelu poimenovali *Dedovanje*. Znotraj pogleda v modelirani javanski program dodajamo nove razrede, abstraktne razrede in vmesnike. Med njimi določimo še povezave, ki realizirajo koncept dedovanja med razredi oz. vmesniki.
- »POGLED 2«, ki smo ga v metamodelu poimenovali *Koda*. Pogled obstaja za vsak sestav razreda, abstraktnega razreda in vmesnika. Znotraj njega modeliramo atribute, metode, konstruktorje in notranje razrede obravnavanega sestava.
- »POGLED 3«, ki smo ga v metamodelu poimenovali *Vidljivost*. Namenjen je obravnavi porazdelitve razredov, abstraktnih razredov in vmesnikov po posameznih *java* datotekah ter uvozu paketov v te datoteke.

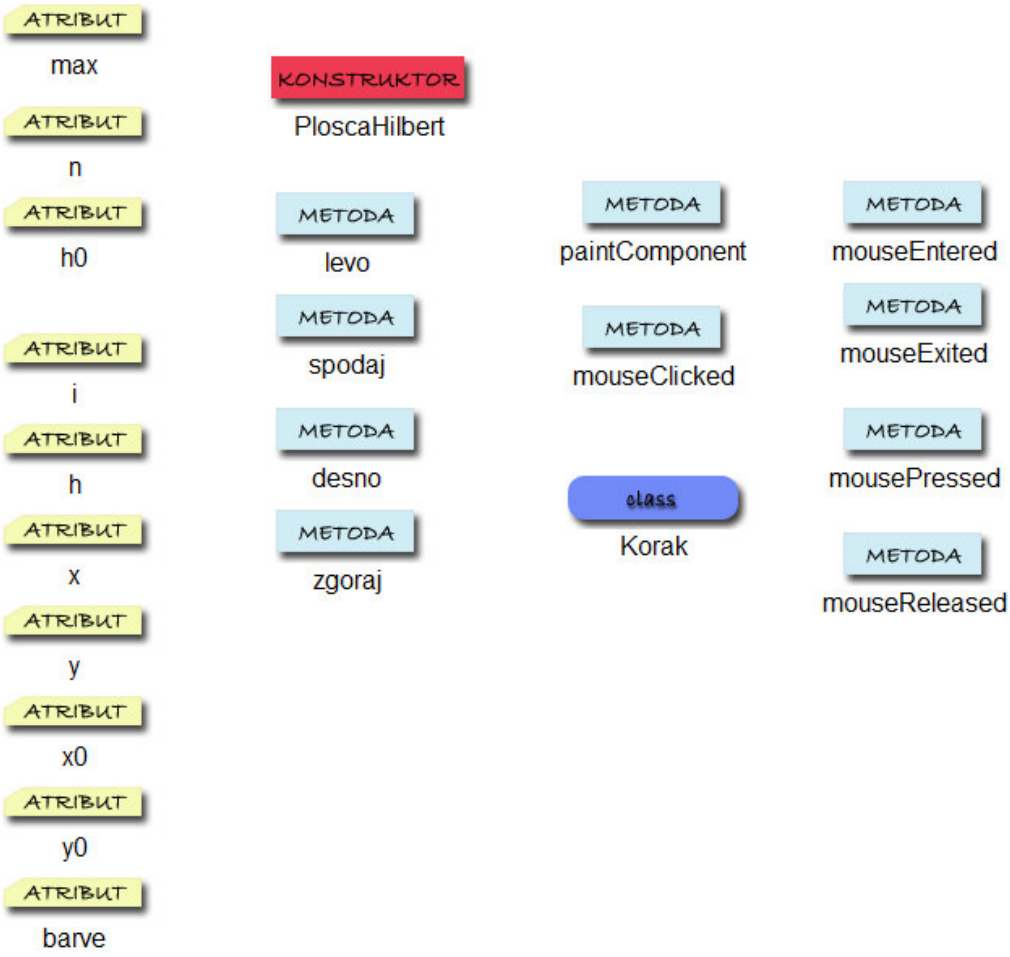
Primer obravnave modelirnih gradnikov javanskega programa (4. naloga in dodatna naloga A*), opisan v *Prilogi 8.2 – Laboratorijske vaje iz Osnov programiranja II*, je po posameznih pogledih prikazan na slikah 47, 48 in 49.



SLIKA 47: PRIMER POGLEDA DEDOVANJE



SLIKA 48: PRIMER POGLEDA VIDLJIVOST



SLIKA 49: PRIMER POGLEDA KODA (RAZREDA PLOSCAHILBERT)

V metamodelu smo poglede definirali tako, kot je prikazano *v Prilogi 8.3.9 – Pogledi*. Pogleda *Dedovanje* in *Vidljivost*, predstavljena z istoimenskima razredoma UML stereotipa »Aspect«, sta kot asociaciji kompozicije določena v okviru krovnega sestava *Naloga* - vsaka javanska naloga vsebuje natanko en tak pogled. Pogled *Koda*, definiran z istoimenskim razredom UML stereotipa »Aspect«, obstaja za vsak sestav razred, notranji razred, abstraktni razred in vmesnik, kar smo definirali z ustreznimi asociacijami kompozicije. Potrebno je bilo določiti še vidljivost modelirnih gradnikov znotraj posameznih pogledov.

Pogled *Dedovanje* vsebuje naslednje modelirne gradnike (glej *Prilogo 8.3.10 – Pogled Dedovanje)*:

- Sestave *class*, *abstractClass* in *interface*
- Atomarne gradnike clasS, abstractClasS in interfacE
- Povezavi Extends in Implenets

Znotraj pogleda Koda obravnavamo naslednje modelirne gradnike (glej *Prilogo 8.3.11 – Pogled Koda)*:

- Sestav *innerClass*
- Atomarne gradnike *Metoda*, *AbstraktnaMetoda*, *Konstruktor* in *Atribut*

V pogled *Vidljivost* smo vključili naslednje modelirne gradnike (glej *Prilogo 8.3.12 – Pogled Vidljivost)*:

- Sestave *class*, *abstractClass* in *interface*. Obravnavani pogled smo za navedene sestave na posebnem mestu okolja GME definirali kot pomožnega, kar pomeni, da sestavov znotraj pogleda ne moremo dodajati oz. brisati, pač pa jih lahko samo uporabljamo.
- Atomarni gradnik *Package*
- Povezavi *Import* in *Public*

5.4 REALIZACIJA OCL OMEJITEV

Z metamodelom smo definirali statični vidik paradigme. Določili smo nabor možnih modelirnih gradnikov, jih opisali z različnimi atributi in predvideli možne povezave med njimi. Da bi bili končni modeli pravilni še v smislu dejanskih vrednosti objektov domene, smo morali predpisati še določena pravila omejevanja teh vrednosti.

Z omejitvenim jezikom OCL v okolju GME definiramo omejitve vrednosti modelirnih gradnikov(atomarnih delov, sestavov ipd.), ki jih preverjamo ob različnih dogodkih, kot so npr.:

- Kreiranje modelirnega gradnika
- Brisanje modelirnega gradnika
- Definiranje povezave med dvema gradnikoma
- Sprememba vrednosti atributov
- Dodajanje gradnika v sestav

Namen preverjanja omejitev ob posameznih dogodkih je dodatno zagotoviti semantično pravilnost grajenih modelov. Kljub temu, da z omejitvami OCL pokrijemo praktično vse prepovedane kombinacije relacij modelirnih gradnikov in vrednosti njihovih atributov, smo nekatere napake oz. nekonsistentnosti pri gradnji modelov morali enostavno spregledati. Glavni razlog temu so omejitve okolje GME, zaradi katerih ne moremo realizirati vsega, kar bi si želeli. Primer takšne omejitve je generično dodeljevanje imena konstruktorja, ki mora biti enako imenu razreda: ko v postopku modeliranja v model razreda dodamo konstruktor, bi pričakovali, da bo konstruktor samodejno podedoval ime razreda. Ker v okolju GME tega ne moremo preprosto implementirati, niti z metamodelom niti z OCL omejitvami, moramo ime konstruktorja vedno znova določiti sami. Kot bomo videli v naslednjem podpoglavju, takšne

in podobne pomanjkljivosti rešujemo z interpreterjem, v katerem obravnavamo še vse tiste nekonsistentnosti modela, ki jih prej ne moremo.

V spodnjih alinejah smo strnili vse najpomembnejše omejitve, ki jih je bilo mogoče obravnavati z omejitvenim jezikom OCL:

- Vmesniki vsebujejo definicije metod, ki jih realiziramo s posameznimi razredi. Ker vidljivost metod v prototipu obravnavamo z atributom *Vidljivost*, z naborom možnih vrednosti *public*, *private*, *protected* in *package access*, je bilo potrebno zapisati omejitev, s katero smo v vmesnikih prepovedali privatne metode*:*
  self.parent().kindName<>"interface" || self.Vidljivost<>#private
- Na enak način smo morali v okviru vmesnikov prepovedati še definicije privatnih atributov:
  self.parent().kindName<>"interface" || self.Vidljivost<>#private
- Za *extends* povezave smo definirali naslednje omejitve:
  - Razred oz. vmesnik lahko razširja natanko en razred oz. vmesnik.
    self.connectionPoints("src") ->
    theOnly().target().attachingConnections("src","Extends") -> size <=1
  - Razred oz. vmesnik ne more razširjati samega sebe. Lahko bi se namreč zgodilo, da bi sestav z *extends* povezavo povezali s samim sabo:
    self.connectionPoints("src") -> theOnly().target() <>
    self.connectionPoints("dst") -> theOnly().target()
  - Dedujemo lahko le od razredov, ki niso označeni kot končni (ne vsebujejo določila *final*)
    if (self.Final=#da) then
    self.attachingConnections("src","Extends") -> isEmpty()
    else
    true
    endif
- Za *implements* povezave je bilo potrebno definirati omejitev, s katero lahko razred implementira nek vmesnik natanko enkrat:
  let destination=self.connectionPoints("dst") -> theOnly().target() in
  self.connectionPoints("src") ->
  theOnly().target().attachingConnections("src","Implements") -> select(b |
  b.connectionPoints("dst") -> theOnly().target() = destination) -> size <= 1
- Za *import* povezave smo določili naslednje omejitve:
  - Paket je mogoče uvoziti le enkrat:
    let destination=self.connectionPoints("dst") -> theOnly().target() in
    self.connectionPoints("src") ->
    theOnly().target().attachingConnections("src","Import") -> select(b |
    b.connectionPoints("dst") -> theOnly().target() = destination) -> size <= 1

- Import povezavo lahko realiziramo le med paketom in razredom z določilom *public*. Kot smo definirali v metamodelu, so kandidati vsi tisti razredi, ki niso določeni kot ciljni sestav nobene izmed možnih *public* povezav:
  self.connectionPoints("dst") ->
  theOnly().target().attachingConnections("dst","Public") -> size = 0

- *Public* povezave smo opisali z naslednjimi omejitvami:
  - Dvojna *public* povezava med istim izvornim in ciljnim razredom oz. vmesnikom je prepovedana:
    let destination=self.connectionPoints("dst") -> theOnly().target() in
    self.connectionPoints("src") ->
    theOnly().target().attachingConnections("src","Public") -> select(b |
    b.connectionPoints("dst") -> theOnly().target() = destination) -> size <= 1
  - Krožna *public* povezava, s katero sestav povežemo s samim sabo, ne more obstajati:
    let source=self.connectionPoints("src") -> theOnly().target() in
    self.connectionPoints("dst") ->
    theOnly().target().attachingConnections("src","Public") -> select(b |
    b.connectionPoints("dst") -> theOnly().target() = source) -> size = 0
  - *Public* povezava med izvornim in ciljnim razredom oz. vmesnikom ne more obstajati, če slednji vsebuje določilo *public*:
    self.connectionPoints("dst") ->
    theOnly().target().attachingConnections("dst","Import") -> size = 0

5.5 OPIS INTERPRETERJA

V prejšnjem poglavju smo povedali, da si v okolju GME z ustrezno izdelanim interpreterjem pridobimo dostop do informacij modela. V primeru paradigme magistrske naloge smo z interpreterjem modeliran javanski program preoblikovali v klasično obliko tekstualno zapisane javanske programske kode in jo po končanem postopku še prevedli z ustreznim prevajalnikom. Za dostop do informacij modela okolje GME ponuja javanske programske vmesnike, ki smo jih uporabili pri realizaciji postopka interpretacije modela. Vmesniki so združeni v dva standardna paketa, *org.isis.gme.bon* in *org.isis.gme.meta*. Preko ustreznih metod je mogoče dostopati do posameznih elementov modela, njihovih atributov in želenih povezav izbranega tipa (*extends*, *implements*, *public*, *import*). Način dostopa do informacij modela sledi priporočilom modelno vodenega razvoja, ki smo ga obravnavali v poglavju 2.4 - v razdelku s postopki neposredne interakcije z modelom.

S tehniko rekurzivne obdelave hierarhične strukture modela, smo na enostaven način postopoma obdelali vse sestave, njihove pod-sestave in atomarne elemente, kot so metode, atributi ter že realizirani razredi oz. vmesniki. Če skrajno poenostavimo postopek

pregledovanja drevesne strukture modela, lahko korake transformacij iz modela v javansko programsko kodo, ki jo na koncu še prevedemo, opišemo takole:

1. V modelu poiščemo vse takšne razrede oz. vmesnike, ki bodo v končnih *java* datotekah imeli določilo *public*.
2. Za vsak takšen razred oz. vmesnik:
    a. Poiščemo vse njegove *import* povezave in izpišemo ustrezne uvoze paketov.
    b. Izpišemo glavo paketa oz. vmesnika.
    c. Glede na *extends* oz. *implements* povezave izpišemo razširitve razreda oz. vmesnika ter vse implementacije vmesnikov.
    d. Poiščemo vse notranje razrede in za njih z nekoliko drugačno obravnavo ponovimo postopek rekurzije (npr. pri notranjih razredih ne izpisujemo uvoza paketov, ipd.).
    e. Poiščemo vse konstruktorje razreda in jih izpišemo
    f. Poiščemo vse metode in abstraktne metode razreda oz. vmesnika in jih izpišemo.
    g. Če se nahajamo v najbolj zgornjem koraku rekurzije:
        i. Poiščemo morebitne razrede oz. vmesnike, ki bodo z obravnavanim *public* razredom oz. vmesnikom skupaj zapisani v isto *java* datoteko. Za njih ponovimo postopek rekurzije, ponovno z nekoliko prirejenim zaporedjem korakov.
        ii. Za obravnavani razred ustvarimo *java* datoteko in vanj vključimo izpisano programsko kodo.
3. Generirano programsko kodo *java* datotek še prevedemo z ustreznim prevajalnikom in uporabniku prikažemo morebitne napake interpreterja oz. prevajalnika.

Podrobnejši način realizacije intepreterja prikazuje spodnja psevdo koda. Pri tem je potrebno poudariti, da smo nekatere izseke programske kode zaradi boljše preglednosti izpustili. Sem sodijo ukazi za izpis zamika programske kode, obravnava sistemskih napak, izpis praznih vrstic ipd.

```
Definiraj globalno spremenljivko za izpis: StringBuffer text;
Definiraj globalno spremenljivko naloga: JBuilderObject naloga;
Definiraj globalno spremenljivko javacPath;
Definiraj globalno spremenljivko pathKrovni;
Definiraj globalno spremenljivko pathStandardPaketi;

InvokeEx(…, JBuilderObject GMEnaloga,…)
{
    Odpremo datoteko z uporabniškimi nastavitvami okolja GME in jih zapišemo v
    globalne spremenljivke.
    {
        javacPath  = Preberi pot do javacPath;
```

pathKrovni *= Preberi pot krovnega direktorija, v katerega naj se po posameznih poddirektorijih generirajo razredi modelirane javanske naloge;*
pathStandardPaketi *= Preberi pot do direktorija, kjer se nahajajo že realizirani uporabniški paketi, razredi in vmesniki (npr. BranjePodatkov);*
}

Globalni spremenljivki ***naloga*** *priredi vrednost parametra GMEnaloga, ki vsebuje naslov modela javanske naloge:* ***naloga****=GMEnaloga;*

S pomočjo GME API se postavi na prvi nivo drevesne strukture modela javanske naloge ***naloga****.*

Definiraj Vektor ***publicSestavi****=****najdiPublicRazrede()****;*

//Rekurzivno izpišemo vse razrede, njihove metode, atribute, notranje razrede…
izpisiRazrede(publicSestavi, -1);

//Nad kreiranimi java datotekami zaženemo prevajalnik in prevedemo obravnavani javanski program
String napaka = Compile(publicSestavi)*;*

če (napaka==prazna)
Končaj z izvajanjem interpreterja;
sicer
Preko posebne statične metode uporabniku izpiši opis napake ***napaka****, ki jo je javil prevajalnik.*
}

//Poiščemo vse sestave (razrede, abstraktne razrede, vmesnike), ki se v nobeni izmed Public povezav ne nahajajo v vlogi ciljnega sestava. V nadaljevanju bomo takšnim sestavom rekli public sestavi.
najdiPublicRazrede()
{
Definiraj Vector ***publicSestavi****;*
Definiraj Vector ***sestavi****=S pomočjo GME API pridobi vse sestave obravnavane javanske naloge* ***naloga****;*
*Sprehodi se čez vse sestave(****x****;* ***x*** *je element* ***sestavi****)*
{
Definiraj Vector ***povezave****=S pomočjo GME API pridobi vse public povezave obravnavane javanske naloge* ***naloga****;*

Definiraj pomožno spremenljivko indikator=0;
*Sprehodi se čez vse najdene public povezave(****p****;* ***p*** *je element* ***povezave****)*
{
Če se obravnavani sestav ***x*** *nahaja v vlogi ciljnega sestava povezave* ***p****, nastavi indikator=1;*
}

```
            če (indikator==1)
                  Dodaj sestav x v Vektor publicSestavi;
      }
      Vrni publicSestavi;
}

// Rekurzivno izpiši vse razrede, njihove metode, atribute, notranje razrede…
izpisiRazrede(sestavi, stopnjaRekurzije)
{
      Sprehodi se čez vse sestave(x; x je element sestavi)
      {
            V pomožne spremenljivke (a1, a2, a3,…, an) preberi vrednosti atributov sestava
            x;

            //Če je obravnavani razred vrhnji(public) razred
            če (stopnjaRekurzije == -1){

                  //Izpišemo vse uvoze uporabniških in standardnih paketov
                  izpisiImportePaketov(x);

                  Glede na tip sestava (razred, abstraktni razred, vmesnik) in vrednosti
                  atributov a1, a2, a3,…, an izpiši glavo sestava (npr. public final class
                  <T> Test ). Izpis pomeni text.append(…);
            }
            sicer{
                  Glede na tip sestava (razred, abstraktni razred, vmesnik, notranji
                  razred) in vrednosti atributov a1, a2, a3,…, an izpiši glavo sestava. Izpis
                  pomeni text.append(…);
            }

            //Izpišemo razširitve razreda oz. vmesnika
            izpisiExtende(x);

            //Izpišemo implementacije vmesnikov razreda
            izpisiImplemente(x);

            //Izpis začetka razreda oz. vmesnika
            text.append("{\r\n");

            //Izpis notranjih razredov sestava
            če(stopnjaRekurzije == -1)
                  izpisiRazrede(najdiInnerRazrede(x),stopnjaRekurzije +2);
            sicer
                  izpisiRazrede(najdiInnerRazrede(x),stopnjaRekurzije +1);

            //Izpišemo atribute razreda oz. vmesnika
            izpisiAtribute(x);
```

```
            //Izpišemo konstruktorje razreda oz. vmesnika
            izpisiKonstruktorje(x);

            //Izpišemo metode razreda oz. vmesnika
            izpisiMetode(x);

            če (stopnjaRekurzije == -1){

                  //Izpis konca razreda oz. vmesnika
                  text.append("\r\n}");
                  izpisiRazrede(najdiRazrede(x),0);

                  //Zapiši vsebino pomožne spremenljivke text v java datoteko. Ime java
                  datoteke naj bo enako imenu razreda x. Direktorij, kamor naj se
                  datoteka zapiše, naj bo enako imenu obravnavane javanske naloge
                  (poseben atribut krovnega sestava Naloga))
                  WriteFile(naloga.imeNaloge, x.imeRazreda, text);

                  Sprazni vsebino pomožne spremenljivke text;

            }
            else
                  //Izpis konca razreda oz. vmesnika
                  text.append("\r\n}");
      }

}

//Izpišemo vse uvoze uporabniških in standardnih paketov
izpisiImportePaketov(x)
{
      Definiraj Vector razredi;
      Definiraj Vector povezave=S pomočjo GME API pridobi vse import povezave
      obravnavane javanske naloge naloga;

      Sprehodi se čez vse najdene import povezave(p; p je element povezave){
      {
            Če se obravnavani sestav x nahaja v vlogi ciljnega sestava povezave p, izpiši
            uvoz paketa (skupaj z določenimi vrednostmi atributov paketa), ki je definiran
            kot izvorni atomarni gradnik import povezave (npr. import java.util.*;). Izpis
            pomeni text.append(…);
      }
}

//Izpišemo razširitve razreda oz. vmesnika
izpisiExtende(x)
{
```

*Definiraj Vector **povezave**=S pomočjo GME API pridobi vse extend povezave obravnavane javanske naloge **naloga**;*

*Sprehodi se čez vse najdene extend povezave(**p**; **p** je element **povezave**){*
{
*Če se obravnavani sestav **x** nahaja v vlogi izvornega sestava povezave **p**, izpiši razširitev razreda oz. vmesnika (skupaj z določenimi vrednostmi atributov razreda oz. vmesnika), ki je definiran kot ciljni element extend povezave (npr. extends Collection<T>). Izpis pomeni **text**.append(…); Izpis se za določen razred oz. vmesnik izvede samo enkrat, ker razredi oz. vmesniki ne morejo razširjati več drugih razredov oz. vmesnikov.*
}
}

//Izpišemo implementacije vmesnikov razreda
izpisiImplemente(x)
{
*Definiraj Vector **povezave**=S pomočjo GME API pridobi vse implement povezave obravnavane javanske naloge **naloga**;*

*Sprehodi se čez vse najdene implement povezave(**p**; **p** je element **povezave**){*
{
*Če se obravnavani sestav **x** nahaja v vlogi izvornega sestava povezave **p**, izpiši implementacijo vmesnika (skupaj z določenimi vrednostmi atributov vmesnika), ki je definiran kot ciljni element implement povezave (npr. implements QueueG<T>). Izpis pomeni **text**.append(…);*
}
}

*//Za obravnavani razred **x** poiščemo vse razrede, ki se kot ciljni sestavi nahajajo v public povezavah, kjer je razred **x** naveden kot izvorni sestav. Drugače povedano - poiskati moramo vse tiste razrede, ki bodo skupaj z razredom **x** določila public obravnavani v eni java datoteki.*
najdiRazrede(x)
{
*Definiraj Vector **povezave**=S pomočjo GME API pridobi vse public povezave obravnavane javanske naloge **naloga**;*

*Sprehodi se čez vse najdene public povezave(**p**; **p** je element **povezave**){*
{
*Če se obravnavani sestav **x** nahaja v vlogi izvornega sestava povezave **p**, dodaj ciljni sestav **w** (razred oz. vmesnik) povezave **p** v vektor **razredi**;*
}
*Vrni **razredi**;*
}

*//Za obravnavani razred **x** poišči vse njegove notranje razrede*
najdiInnerRazrede(x)
{

*Definiraj Vector **notranjiRazredi**=S pomočjo GME API pridobi vse notranje razrede (sestave tipa innerClass) razreda **x**;*

*Vrni **notranjiRazredi**;*

}

*//Za obravnavani razred oz. vmesnik **x** izpiši vse njegove atribute*
izpisiAtribute(x)
{

*Definiraj Vector **atributi** = S pomočjo GME API pridobi vse atribute (razredi stereotipa »Attribute«) razreda oz. vmesnika **x**;*

*Sprehodi se čez vse najdene atribute razreda oz. vmesnika (**a**; **a** je element **atributi**)*
{

*Izpiši atribut razreda oz. vmesnika (skupaj z določenimi vrednostmi atributov atributa). Izpis pomeni **text**.append(…). Primer: public static String param1 = new String();*

}

}

*//Za obravnavani razred **x** izpiši vse njegove konstruktorje*
izpisiKonstruktorje(x)
{

*Definiraj Vector **konstruktorji** = S pomočjo GME API pridobi vse konstruktorje (sestavi tipa »Konstruktor«) razreda oz. vmesnika **x**;*

*Sprehodi se čez vse najdene konstruktorje razreda (**k**; **k** je element **konstruktorji**)*
{

*Če ime konstruktorja **k** ni enako imenu razreda **x**, potem končaj izvajanje interpreterja in uporabniku preko posebne statične metode izpiši napako »Ime konstruktorja mora biti enako imenu razreda«;*

*Izpiši konstruktor razreda (skupaj z določenimi vrednostmi atributov konstruktorja). Izpis pomeni **text**.append(…). Primer: public Test(int a, int b, String c);*

}

}

*//Za obravnavani razred **x** izpiši vse njegove metode*
izpisiMetode(x)
{

*Definiraj Vector **metode** = S pomočjo GME API pridobi vse metode (sestavi tipa »Metoda«) razreda **x**;*

*Sprehodi se čez vse najdene metode razreda (**m**; **m** je element **metode**)*
{

*Izpiši metode razreda (skupaj z določenimi vrednostmi atributov metode). Izpis pomeni **text**.append(…). Primer: public static void izpisiPodatke(Object a);*

}

*Definiraj Vector **abstraktneMetode** = S pomočjo GME API pridobi vse abstraktne metode (sestavi tipa »AbstraktnaMetoda«) razreda oz. vmesnika **x**;*

*Sprehodi se čez vse najdene abstraktne metode razreda (**am**; **am** je element **abstraktneMetode**)*

{

*Izpiši metode razreda (skupaj z določenimi vrednostmi atributov metode). Izpis pomeni **text**.append(…). Primer: public abstract void izpisiPodatke(Object a);*

}

}

//Zapišemo vsebino parametra text v java datoteko. Ime java datoteke je enako vrednosti parametra imeRazreda. Direktorij, kamor naj se datoteka zapiše, je enako vrednosti parametra imeNaloge)

WriteFile(imeNaloge, imeRazreda, text)

{

*Znotraj krovnega direktorija **pathKrovni** ustvari direktorij javanske naloge **imeNaloge**;*

*V direktoriju **imeNaloge** ustvari java datoteko z nazivom **imeRazreda**;*

*Zapiši vsebino parametra **text** v datoteko **imeRazreda**;*

}

//Nad kreiranimi java datotekami zaženemo prevajalnik in prevedemo programsko kodo

Compile(publicSestavi)

{

*Definiraj pomožno spremenljivko: String **sestavi**;*

*Sprehodi se čez vse sestave (**s**; **s** je element **publicSestavi**)*

{

Sestavimo niz znakov iz imen razredov oz. vmesnikov – za ta imena obstajajo istoimenske java datoteke, ki smo jih ustvarili v metodi WriteFile

***sestavi = sestavi +** s.imeRazreda;*

}

*//Prevedemo javanski program in v primeru napake v spremenljivko **napaka** zapišemo opis napake*

*String **napaka** = Poženi zunanji program (prevajalnik), ki v kombinaciji s spremenljivkami **sestavi**, **javacPath**, **pathKrovni** in **pathStandardPaketi** prevede obravnavani javanski program.*

*Vrni **napaka**;*

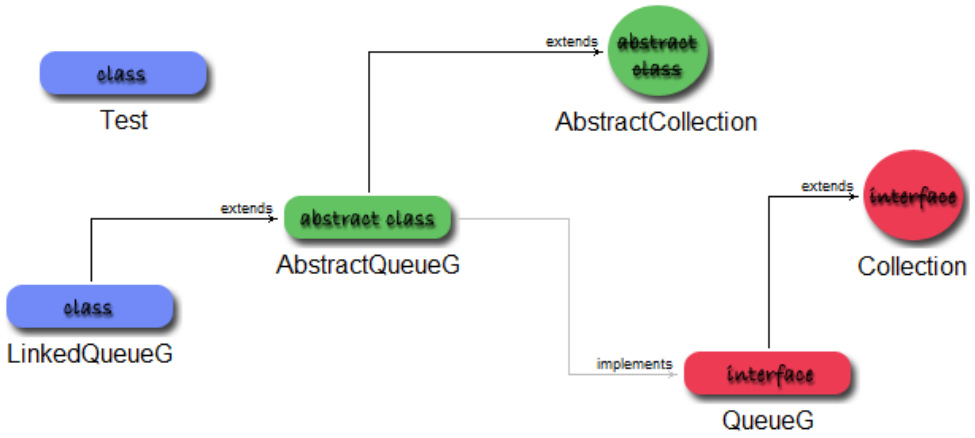
}

5.6 PRAKTIČEN PRIKAZ DELOVANJA PROTOTIPA

Za konec poglavja si oglejmo še praktičen primer uporabe prototipa, kljub temu, da smo do sedaj pokazali že kar nekaj segmentov njegove uporabe, vendar nikoli v celoti.

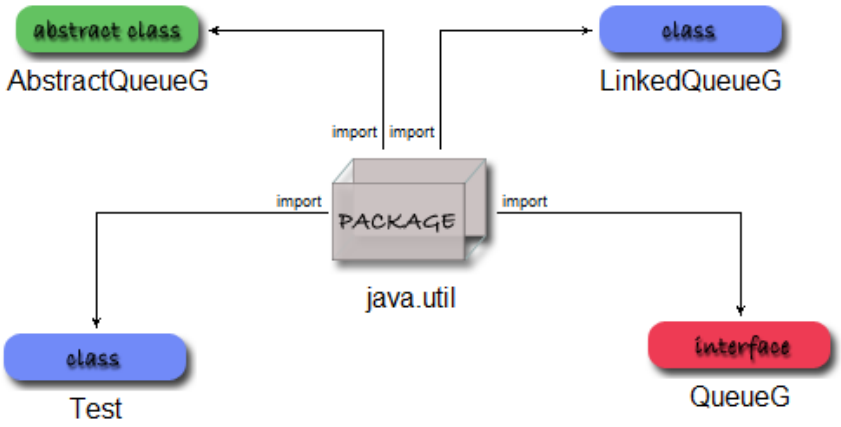
V *Prilogi 8.2 – Laboratorijske vaje iz Osnov programiranja II* je pod točko 5. opisana naloga za izdelavo javanskega programa, ki realizira vrsto kot povezan linearni seznam objektov, pri čimer naj bo vrsta predstavljena kot generični razred.

Najprej smo v pogledu Dedovanje opisali ogrodje javanskega programa in v njem definirali glavni razred *LinkedQueue*, ki razširja abstraktni razred *AbstractQueueG*, ta pa implementira vmesnik *QueueG*. Abstraktni razred razširja že realizirani abstraktni razred *AbstractCollection* paketa *java.util*, vmesnik pa razširja že realizirani vmesnik *Collection*, prav tako paketa *java.util*. Ker moramo v nalogi realizirati generično vrsto, se vsi omenjeni razredi v svoji definiciji sklicujejo na določila generičnega tipa, kar opišemo z vrednostjo *T* atributa *GenericniTip*. Poleg vseh navedenih razredov v model vključimo še razred *Test*, v katerem bomo prikazali uporabo generične vrste. Zastavljeno modelirano ogrodje javanskega programa prikazuje slika 50.



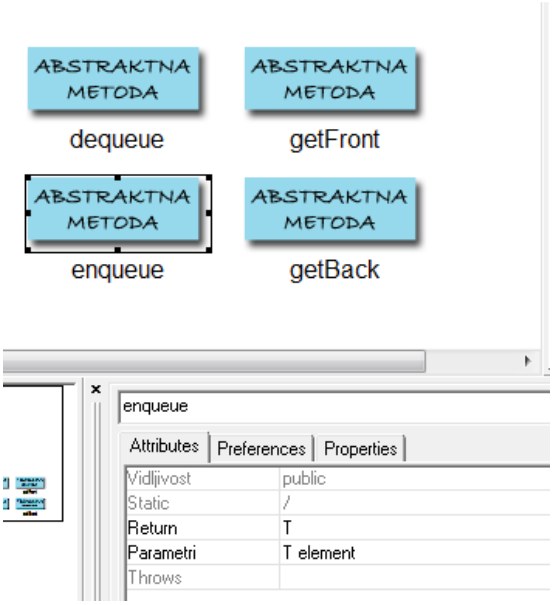
SLIKA 50: MODELIRANO OGRODJE JAVANSKEGA PROGRAMA, KI REALIZIRA GENERIČNO VRSTO

Ker smo se pri modeliranju programa sklicevali na že obstoječa abstraktni razred in vmesnik paketa *java.util*, smo morali uvoz paketa modelirati znotraj pogleda *Vidljivost* (slika 51). Prav tako smo določili še razporejenost razredov in vmesnika po posameznih *java* datotekah. Najbolj smiselno je bilo, da smo vsak razred zapisali v svojo datoteko, zato v modelu ni videti črtkanih povezav med modelirnimi gradniki, ki predstavljajo opisan koncept *public* povezave.



SLIKA 51: UVOZ PAKETA JAVA.UTIL IN RAZPOREJENOST RAZREDOV PO DATOTEKAH

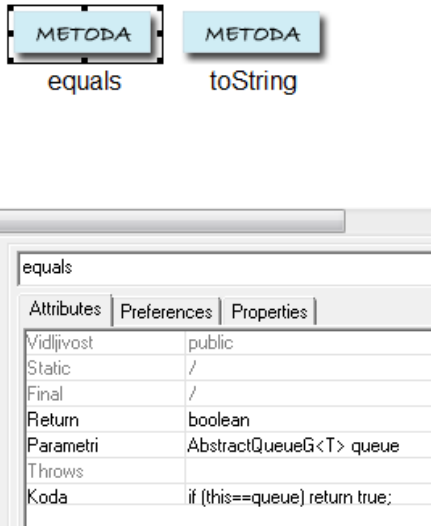
Modelirane definicije metod skupaj z določenimi vrednostmi atributov metode *enqueue* vmesnika *QueueG* prikazuje slika 52. Na sliki 53 sta prikazani abstraktni metodi abstraktnega razreda *AbstractQueueG*. Poleg tega so na sliki prikazane še vrednosti atributov metode *equals*. Programske kode metod (vrednost atributa *Koda*) znotraj te točke ne bo bomo obravnavali.



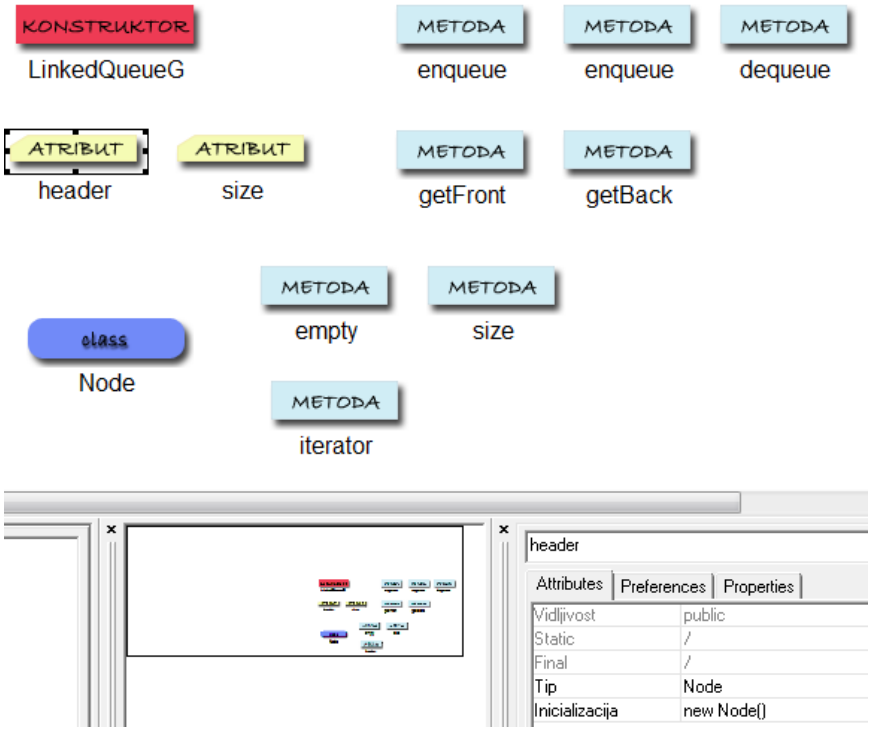
SLIKA 52: ABSTRAKTNE METODE VMESNIKA QUEUEG

Metode, atribute in notranji razred *Node*, s katerim smo implementirali iterator že realiziranega vmesnika *Collection*, razreda *LinkedQueueG* smo oblikovali z modelom, ki je prikazan na sliki 54. Iz slike je lepo razvidna ena izmed glavnih prednosti vizualne

predstavitve javanskih konceptov, v katerih smo posamezne modelirne gradnike, ki so si po vsebini ali funkcionalnostih podobni, skupaj združili na enem mestu.

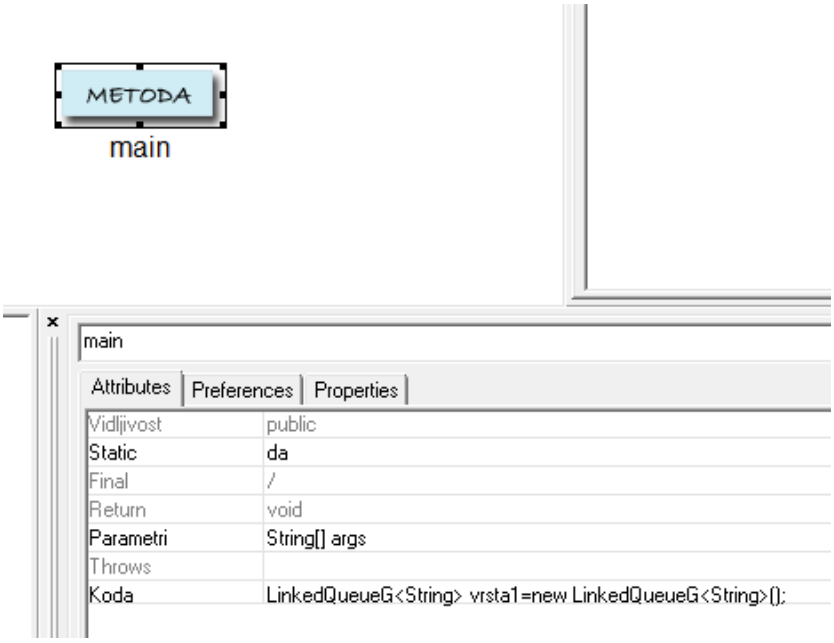


SLIKA 53: METODE ABSTRAKTNEGA RAZREDA ABSTRACTQUEUEG



SLIKA 54: METODE, ATRIBUTI IN NOTRANJI RAZRED NODE RAZREDA LINKEDQUEUE

Edina metoda razreda *Test*, s katero smo prikazali uporabo generične vrste *LinkedQueueG*, je prikazana na sliki 55.



SLIKA 55: MAIN METODA RAZREDA TEST

6. UPORABA PROTOTIPA V PRAKSI

V tem poglavju se bomo osredotočili na dejansko uporabnost prototipa v praksi. Najprej bomo v kratkem opisali, kakšne osebne izkušnje imamo z rokovanjem s prototipom, katere pomanjkljivosti smo našli tekom razvoja in testiranja prototipa, oz. katere so tiste ideje, s katerimi bi izboljšali trenutno različico prototipa. Nato bomo povedali, kako smo prototip predstavili študentom in na katere stvari smo jih ob njegovi uporabi opozorili. V zadnjem delu poglavja bomo predstavili še rezultate študentske ankete in se osredotočili na tiste pomanjkljivosti prototipa, na katere so nas kot najbolj sporne opozorili študenti.

6.1 OSEBNE IZKUŠNJE Z UPORABO PROTOTIPA

Kot smo navedli že v poglavju *5.1 Zajem zahtev obravnavane domene,* smo pri izgradnji prototipa sodelovali tako v vlogi eksperta domene kot tudi načrtovalca modelov. Zaradi tega smo v vsakem trenutku vedeli, kaj lahko od prototipa pričakujemo in česa ne. Ker smo v postopkih iterativno-inkrementalnega razvoja sproti odpravljali pomanjkljivosti trenutnih različic prototipa, smo proti koncu dobili dovolj dobro ogrodje, ki je zadostilo večini naših prvotnih pričakovanj. Nekatere pomanjkljivosti, ki jih zaradi omejitev okolja GME nismo mogli odpraviti, smo kot take sprejeli že pred začetkom izdelave prototipa. Najpomembnejše izmed njih lahko na kratko opišemo v naslednjih nekaj alinejah:

- Prototip se v času modeliranja »ne zaveda« semantične pravilnosti javanske kode telesa metod. Pravilnost oz. nepravilnost je mogoče ugotoviti šele v času prevajanja programa. Ena izmed večjih praktičnih slabosti tega je nezmožnost sprotnega podčrtavanja oz. barvanja nepravilno grajene programske kode, kar razvijalcu-programerju močno olajša delo. Zaradi istega razloga tudi ni mogoče realizirati funkcije *IntelliSense*, ki nam pomaga s sprotnimi sugestijami oblikovanja javanske programske kode. Argument v prid neobstoju teh funkcionalnosti je dejstvo, da začetniki prepogosto postanejo popolnoma odvisni od pomoči orodja in so brez nje popolnoma izgubljeni.
- Z uvozom paketov znotraj pogleda *Vidljivost* se prototip ne zaveda vsebine paketa. V praksi bi s prisotnostjo te funkcionalnosti znotraj pogleda *Dedovanje* uporabniku lahko ponudili nabor že realiziranih razredov oz. vmesnikov, ki so vsebovani znotraj uvoženega paketa. S tem bi dodano vrednost prototipa podvojili. V 5. poglavju, v katerem smo omenjeno pomanjkljivost prototipa posredno že izpostavili, smo povedali, da bi realizacija takšnega koncepta močno presegala obseg magistrske naloge. Poleg tega v tem trenutku ne moremo potrditi, ali bi z okoljem GME to sploh lahko realizirali. Če se tako kot v prejšnji točki postavimo v prid neobstoju te funkcionalnosti, lahko rečemo, da morajo začetniki zaradi tega bolj poznati javanski API in natančno vedeti, kateri že realizirani razredi oz. vmesniki spadajo v kateri

standardni paket. Vse to osnovno znanje zagotovo pričakujemo od začetnika, ki bi se v nasprotnem primeru preveč zanašal na ponujene informacije razvojnega okolja.

- Zaradi enotnega načina predstavitve modelov znotraj okolja GME izgubimo možnosti drugačne predstavitve informacij. V pogledu *Koda* bi bilo npr. koristno imeti informacijo o modelu na enem hierarhičnem nivoju višje, kar lahko ugotovimo le s sprehodom navzgor po modelu. Dobrodošla bi bila npr. vizualizacija v obliki pol-prosojnega modela višje stopnje, ki bi se prikazal s pritiskom določene kombinacije tipk. Na podoben način znotraj telesa posameznih metod včasih potrebujemo informacijo o programski kodi metode nekega drugega razreda oz. vmesnika. Pri klasičnem programiranju do te informacije zelo enostavno pridemo z iskalnikom besedila ali s hitrimi premiki navigacijskega koleščka na miški. V okolju prototipa pa se do te informacije dokopljemo šele po nekajkratnih sprehodih med modeli in njihovimi elementi ter različnimi okni programske kode zunanjega urejevalnika besedila. Glavna pomanjkljivost prototipa v tem primeru je torej način predstavitve informacij podrejenega okolja, ki ne omogoča preproste vizualne razširitve njegovega ogrodja.
- Tudi dinamika okolja GME se je prevečkrat pokazala kot izjemno toga. Veliko stvari se sicer da doseči s pomočjo interpreterjev, kljub vsemu pa bi včasih potrebovali tudi kakšne bolj enostavne prijeme za rokovanje z modeli. Kot najbolj očiten primer tega naj navedemo nezmožnost avtomatskega generiranja konstruktorjev ter *get()* in *set()* metod v trenutku, ko ustvarimo nek razred oz. vmesnik. Ali pa nezmožnost avtomatskega ustvarjanja *main()* metode in določil vseh njenih atributov v primeru kreiranja *public* razreda. Mehanizmi za interakcijo z modelom v obliki javanskih knjižnic *org.isis.gme.bon* in *org.isis.gme.meta,* v tem trenutku tega ne omogočajo.

Najboljše ideje za izboljšanje prototipa se nahajajo prav v ravnokar obravnavanih alinejah. Zaradi kompleksnosti zahtevanih nalog in omejitev okolja GME se lahko vprašamo, katere izmed njih bi v praksi sploh lahko uresničili. Če želimo ostati na trdnih tleh, lahko predlagamo konkretne izboljšave, ki bi jih resnično lahko dosegli v relativno kratkem času:

- Pogled *Koda* bi lahko razširili z možnostjo dodajanja notranjih abstraktnih razredov in notranjih vmesnikov. Takšnih zahtev v okviru rednih domačih nalog študentov 1. letnika zaenkrat nismo zasledili.
- V pogled *Koda* bi naknadno lahko dodali še možnost razširjanja oz. implementacije notranjih razredov in vmesnikov, kot to lahko sedaj počnemo znotraj pogleda *Dedovanje*. Tudi takšnih potreb v okviru rednega programa zaenkrat nismo zasledili.
- OCL omejitve bi lahko razširili z novimi omejitvami, ki uporabnikom ne bi nekaj prepovedovali, ampak bi jih le opozarjali na določene pomanjkljivosti v modelu. Okolje GME takšne možnosti omogoča. Kot primer lahko izpostavimo opozorilo, ki nas opozarja na neujemanje imena konstruktorja z imenom njegovega razreda. Na

takšne anomalije nas trenutno opozarja interpreter, bolj učinkovito rešitev pa bil imeli, če bi se anomalij zavedali že med gradnjo modela.

- Po opravljeni analizi študentske ankete, ki jo bomo predstavili v naslednji točki, bi morali še enkrat premisliti glede oblikovanih simbolov javanskih konceptov. Nekateri študenti so v anketi izrazili skrb, da so simboli še vedno premalo razumljivi in uporabnikom vzamejo preveč časa, da se seznanijo z njimi.

Kljub navedenim pomanjkljivostim je prototip v trenutni različici dovolj dobro izdelan, da lahko z njim dokaj enostavno modeliramo katerokoli javansko nalogo, ki jo študenti po rednem programu obravnavajo na predavanjih in laboratorijskih vajah. Preden smo prototip predali v ocenjevanje študentom, smo vse domače naloge, ki bi jih študenti v okviru domačih nalog morali opraviti do konca semestra, z orodjem v celoti modelirali tudi sami. Le v nekaterih primerih jeder metod, v katerih je bilo potrebno programirati res veliko število vrstic programske kode, se je rokovanje s prototipom izkazalo kot nekoliko nerodno. V večji meri predvsem zaradi prekomernega preklapljanja med okni urejevalnika besedila, zamudnega sprehajanja med hierarhičnimi nivoji modela in nestabilnega delovanja programa. Če pogledamo na izdelek kot celoto, lahko rečemo, da smo z organiziranim iterativno-inkrementalnim razvojem zadovoljivo zaobšli nekatere omejitve modelirnega okolja GME in z delujočim prototipom jasno predstavili in preizkusili vse tiste koncepte modelno vodenega razvoja, ki smo si jih kot cilij zastavili raziskati na začetku. Kljub temu, da prototip ni idealen, smo ga z veliko mero zaupanja predali v ocenjevanje manjši skupini študentov, da o njem izrazijo mnenje.

6.2 POIZKUS VPELJAVE PROTOTIPA NA LABORATORIJSKIH VAJAH

Eden izmed pomembnejših ciljev praktičnega dela magistrske naloge je bil, da izdelani prototip preizkusi izbrana množica študentov 1. letnika študija. V prvem semestru so študentje pri predmetu Osnove programiranja I spoznali temeljne koncepte programiranja v Javi in se seznanili z objektno usmerjeno strukturo programa. Ker so imeli že neko osnovno znanje iz področja programiranja, so bili zelo primerni kandidati za podajo ocene. V prvem semestru so se z naprednimi orodji za urejanje programske kode naučili programirati na klasičen način. Vedeli so, kaj jim je pri učenju povzročalo največ težav, poleg tega pa so do takrat že spoznali vse tiste javanske koncepte, ki smo jih v prototipu z drugačno obliko percepcije želeli predstaviti na bolj enostaven in intuitiven način.

Predvidevali smo, da bodo študentje brez težav identificirali vse tiste elemente prototipa, ki bi začetnikom nudili pomoč pri razumevanju objektno usmerjenega programiranja. Hkrati smo upali, da bodo znali izpostaviti vse tiste njegove pomanjkljivosti, ki bi programerja-začetnika ovirali pri modeliranju javanskih programov. Kljub jasno zastavljenima ciljema smo bili prepričani, da bodo študentje odgovore na obe vprašanji zelo težko podali. Glavni razlog temu so očitne omejitve in pomanjkljivosti okolja GME, ki smo jih v okviru razvoja identificirali že

sami. Zaradi tega ni bilo mogoče jasno predvideti, ali se bodo študenti v prototipu zavedali vseh prednosti, ki jim jih lahko ponudi modelno voden razvoj, ali bodo v prototipu samo iskali pomanjkljivosti, ki jim jih s prototipom nismo mogli ponuditi. Dejstvo, da gre samo za prototip, katerega namen je predstaviti določene koncepte, je dosti težje razumeti kot občutiti nestabilnost programa, pisati programsko kodo brez barvanja kode, ne imeti funkcije *IntelliSense* ipd.

Prototip smo na delovnem sestanku predstavili izbrani množici študentov. Vsega skupaj je v poizkus privolilo 5 študentov, od skupaj 11. K sodelovanju smo povabili vse tiste študente, ki so v prvem semestru pri predmetu Osnove Programiranja I redno hodili na vaje, delali domače naloge in na splošno dajali vtis zglednih študentov – ne glede na njihovo dejansko znanje iz programiranja v Javi. Na delovnem sestanku smo jih s pomočjo primerov seznanili z vsemi funkcionalnostmi prototipa in še posebej izpostavili tiste, ki jih morajo zaradi narave prototipa vzeti v zakup – kot ne najboljše. Poleg tega smo jih opozorili na vse tiste stvari, ki jih prototip v primerjavi z ostalimi naprednimi razvojnimi orodji nima, bi jih pa načeloma v svoji končni (ne-prototipni) različici lahko imel. S tem smo jim želeli obrazložiti s katerimi stvarmi naj se ne ukvarjajo oz. naj jih spregledajo in na katere naj bodo pozorni.

Študenti so morali prototip preizkusiti v šestih javanskih domačih nalogah, ki so jih tako kot ostali študentje po rednem programu morali opraviti do konca semestra. Ker smo študentom prototip predstavili nekoliko pozno, nekje v času zagovorov druge domače naloge, smo jih prosili, naj kljub vsemu s prototipom modelirajo tudi prvi dve nalogi. Poleg tega smo jim na koncu čas za oddajo zadnje naloge nekoliko podaljšali. Celoten nabor zahtevanih nalog je naveden v *Prilogi 8.2 Laboratorijske vaje iz Osnov programiranja II.*

6.3 IZKUŠNJE ŠTUDENTOV IN REZULTATI ANKETE

Pogoj za sodelovanje pri testiranju prototipa je bila tudi obvezna izpolnitev ankete ob koncu semestra. Namen ankete je bil najti pomanjkljivosti prototipa in ugotoviti, ali modelno voden razvoj javanskih programov lahko začetnikom pomaga pri razumevanju konceptov objektno usmerjenega programskega jezika Java. Anketo smo oblikovali v dveh delih:

- Najprej smo od študentov želeli izvedeti njihov dosedanji nivo znanja iz programiranja v Javi. Zanimalo nas je tudi, ali v splošnem poznajo modelno voden razvoj, ali se zavedajo njegovih prednosti in ali bi imeli takšen koncept izdelave programov rajši od klasičnega.
- V drugem delu smo od študentov želeli izvedi nekoliko več o konceptualni zasnovi prototipa: ali jim je jasna porazdelitev modelirnih gradnikov znotraj treh obravnavanih pogledov – *Dedovanje*, *Vidljivost* in *Koda*, kaj jih v okviru posameznega pogleda moti, katere funkcionalnosti pogrešajo, ali jim hierarhična struktura modela oz. t.i. tehnika »vrtanja v globino« poenostavita razumevanje objektne zasnove programa, ipd.

Vprašanja ankete so bila v večini zastavljena tako, da bi od študentov dobili tem več koristnih informacij v zvezi s koncepti modelno vodenega razvoja, ki smo jih predstavili v prototipu. Prav tako smo želeli izvedeti, kakšna je uporabna vrednost prototipa v praksi. Nabor vseh vprašanj ankete se nahaja v *Prilogi 8.1 – Anketa – Prototip orodja za modelno voden razvoj javanskih programov*. Kljub temu, da smo študente jasno opozorili na pomanjkljivosti in omejitve prototipa, ki zaradi njegove testne narave ne bi smeli vplivati na končno oceno, smo pričakovali kar nekaj kritik tudi s tega naslova. Vrnjeno in izpolnjeno anketo smo dobili od vseh sodelujočih študentov. Rezultati ankete se nahajajo v *Prilogi 8.4 – Rezultati ankete študentov*. Povzetek rezultatov ankete smo strnili v nadaljevanju poglavja.

Predhodno znanje iz programiranja v Javi med študenti je bilo porazdeljeno, dva sta znala programirati že od prej, trije študenti pa so bili čisti začetniki. Kljub temu so bili vsi v večini prepričani, da znajo odlično uporabljati znanje, ki so si ga pri programiranju v Javi pridobili v prvem semestru. Po svojem občutku so bili študentje mnenja, da znajo programirati nekoliko bolje od svojih vrstnikov v letniku. S sklepanjem bi lahko ugotovili, da so bili ravno zaradi tega študenti pravi kandidati za ocenitev prototipa. Dobro so poznali osnove programiranja ter večino pasti, na katere so naleteli kot začetniki. Te pasti so po našem mnenju ključnega pomena za ocenitev kakovosti prototipa, saj smo jih nekaj prav z modelno zastavljenimi koncepti želeli odpraviti ali pa le bolj opozoriti na njih.

Kar se tiče uporabe prototipa (v anketi smo ga poimenovali prototip JCP), so bili študentje različnih mnenj. Trije izmed njih so s prototipom izdelali vse zahtevane naloge. Dva sta prototip uporabljala celo pri nalogah, ki niso bile predpisane v okviru rednih domačih nalog. Ostala dva študenta sta imela s prototipom nekaj težav, na katere sta opozorila šele po koncu opravljenih vaj. Prototip sta uporabljala le deloma - dosti sta si pomagala s klasičnim programiranjem, potem pa sta zapisano programsko kodo prekopirala v prototip. Eden izmed glavnih razlogov, kot sta ga tudi navedla v nadaljevanju ankete, je nezmožnost barvanja programske kode metod prototipa. Težavo bi zelo enostavno rešili z integracijo drugega urejevalnika teksta. Na dejstvo, da je pri prototipu potrebno precej »klikati z miško«, saj določenih omejitev okolja GME pač ne moremo zaobiti, smo opozorili že na delovnem sestanku. Prototip so študentje res začeli uporabljati nekoliko kasneje, kot je bilo prvotno predvideno, vendar to ne upraviči lenobe, da dva izmed študentov prvih dveh nalog nista modelirala tudi kasneje.

Študentje so o modelno vodenem razvoju programske opreme pred uporabo prototipa že slišali, vendar se z njim še niso srečali v praksi. Vsi so bili enotnega mnenja, da so se z njegovo uporabo zavedali prednosti, ki jim jih lahko ponudi modelno usmerjeno programiranje oz. bolje rečeno modeliranje. Na vprašanje, ali bi se programiranja v prihodnje še vedno lotili na klasičen način in ali bi začetnikom priporočili uporabo modelno vodenega razvoja, so študentje odgovarjali dokaj porazdeljeno - tako v prid enemu kot tudi drugemu pristopu. Tisti, ki poleg programskega jezika Java poznajo tudi druge objektno usmerjene jezike, so bili po večini mnenja, da je način modelno vodenega razvoja, ki smo ga predstavili

s prototipom, popolnoma smiseln za programski jezik Java. Na vprašanje, ali bi raje še naprej uporabljali klasična razvojna orodja ali modelno vodena orodja z vsemi najpomembnejšimi funkcijami klasičnih razvojnih orodij (kot sta npr. barvanje kode in t.i, IntelliSense), so študentje odgovorili v prid slednjim.

Način prototipne modelne zasnove javanskega programa se je študentom v celoti zdel smiseln. S tremi zastavljeni pogledi (*Dedovanje*, *Koda* in *Vidljivost*) so si na logičen način jasno predstavljali modelirani program. Bili so mnenja, da si s pomočjo vizualne percepcije posamezne objektne koncepte predstavljajo dosti lažje in se na njih osredotočijo bolje kot pri klasičnem programiranju. Eden izmed študentov je povedal, da se v primeru telesa metod, kjer je še vedno potrebno programirati na klasičen način, bolje znajde pri klasičnih razvojnih orodjih. S tem se z njim v celoti strinjamo. Kot smo povedali že v prejšnji točki, bi z večjo dovršenostjo prototipa in vključenostjo najpomembnejših funkcij klasičnih razvojnih orodij lahko odpravili tudi to pomanjkljivost.

Glede pogleda *Dedovanje* študenti niso imeli večjih pripomb. V večini so se strinjali, da je s pomočjo usmerjenih povezav mogoče jasno prikazati koncepta dedovanja in implementacije vmesnikov. Nekaj pozitivnih mnenj je bilo izraženih tudi glede uporabe barv pri posameznih modelirnih gradnikih. Dva študenta sta imela pomisleke glede prečrtanih imen pri že realiziranih razredih oz. vmesnikih. Po njunem mnenju bi morali to predstaviti na drugačen način oz. koncept vizualizacije bolj poenostaviti. Če smo prav razumeli komentar enega izmed študentov, bi se dejanska realizacija *extends* in *implements* povezav morala odražati tudi znotraj pogleda *Koda*, kot je npr. avtomatsko generiranje potrebnih metod implementiranega vmesnika, ipd. Tudi s to pripombo študenta se v celoti strinjamo, saj smo navedeno hibo omenili kot eno izmed največjih pomanjkljivosti trenutne različice prototipa.

Pri pogledu *Vidljivost* se študenti enotno niso strinjali s trditvijo, da je s pomočjo sivih črtkanih povezav mogoče jasno prikazati organiziranost razredov po datotekah. Ravno nasprotno so bili enotnega mnenja, da je z usmerjenimi *import* povezavami mogoče jasno predstaviti uvoze različnih paketov po posameznih datotekah. Eden izmed študentov je le opozoril na morebitno nepreglednost modela v primeru velikega števila povezav, kot bi lahko to opazili na slikah 15 in 48.

Študenti so bili bolj ali manj enotni pri odgovorih glede pogleda *Koda*. Strinjali so se, da si je z modeliranjem strukturo programa sicer dosti lažje predstavljati, vendar pri tem težko dosežemo enako ali celo večjo raven učinkovitosti kot pri klasičnem programiranju. Morda bi z že omenjenimi izboljšavami prototipa produktivnost nekoliko povečali, vprašanje pa je, če bi s tem res presegli produktivnost klasičnih razvojnih orodij. Glede tehnike t.i. »vrtanja v globino« so študenti potrdili trditev, da je edino tako mogoče smiselno prikazati hierarhično strukturo programa na grafičen način. Prav tako so bili mnenja, da je z modeliranjem metode, atribute in notranje razrede smiselno združevati glede na sorodnost vsebine, ker je s tem struktura programa dosti bolj razumljiva.

Študenti so kot slabost uporabe prototipa večkrat omenjali nezmožnost barvanja kode. Naj ob tem povemo, da pri tem najbrž ni bilo mišljeno samo barvanje kode (npr. FOR zank z eno barvo, nizov znakov z drugo, spremenljivk s tretjo itn.), saj smo način integracije zunanjih urejevalnikov besedila, ki omogočajo barvanje kode, pokazali na delovnem sestanku. Študenti so ob tem najbrž želeli izpostaviti dejstvo, da kljub uporabi zunanjih urejevalnikov besedila ni mogoče prikazati (pobarvati) semantičnih napak znotraj programske kode, kot so to vajeni pri klasičnem programiranju. Ker celotno strukturo javanskega programa oblikujemo šele na koncu, ko poženemo interpreter, se prototip v času gradnje modela nikakor ne more zavedati pravilnosti oz. nepravilnosti programske kode telesa metod. Ker sta generirana in ročno napisana programska koda združeni šele v času interpretacije modela, bi bile potrebne ogromne dopolnitve in sprememba prototipa, da bi se semantične pravilnosti programske kode prototip lahko zavedal že v postopku modeliranja. Zaradi omejitev okolja GME iz posameznih vrednosti atributov metod prav tako ni bilo mogoče avtomatsko generirati glav metod v urejevalniku besedila, kar bi zelo povečalo dodano vrednost prototipa. Iz istega razloga tudi ni bilo mogoče samodejno generirati konstruktorjev, *get()* in *set()* metod razredov, kar so študenti ponovno označili kot eno izmed glavnih pomanjkljivosti prototipa. Izpostavljena je bila tudi slabost okolja GME, zaradi katere ni mogoče urejati več vrednosti atributov hkrati. To bi v praksi pomenilo, da bi lahko imeli odprtih več oken s programsko kodo jeder metod, s čimer bi povečali preglednost nad modeliranim programom. Z razširitvijo funkcionalnosti okolja GME bi tudi izboljšali obravnavo napak, ki jih ob prevajanju preko interpreterja javi prevajalnik. Strinjamo se s trditvijo enega izmed študentov, da v trenutni različici prototipa napak znotraj programske kode, ki jih javi prevajalnik, ni enostavno razumeti. Ker je napaka obravnavana v vrstici dokončno združene - ročno napisane in generirane programske kode, težko ugotovimo, na katerem mestu (v kateri vrstici telesa metode, ki jo obravnavamo znotraj atributa *Koda* neke metode) je dejansko prišlo do napake in kako naj napako odpravimo.

Kot najbolj zanimivi ideji za izboljšanje prototipa so študenti predlagali možnost tehnike »Reverse Enginering« in uporabo hitrih tipk. Kot dodatni funkcionalni modul prototipa bi lahko napisali interpreter, ki bi postopek gradnje javanskega programa obrnil – tako bi že napisano javansko kodo lahko preoblikovali v modele znotraj vseh treh pogledov prototipa. Na ta način bi začetnikom, ki ne razumejo obravnavanih programov na predavanjih in laboratorijskih vajah, ponudili še eno možnost več. Kljub ogromni pridobitvi prototipa bi v praksi realizacijo omenjene tehnike izjemno težko dosegli. Razloge za to smo jasno navedli v 5. poglavju, v katerem smo obravnavali koncept že realiziranih razredov oz. vmesnikov. Bolj enostavno bi bilo zagotoviti nekakšno razširitev okolja GME, ki bi omogočala uporabo hitrih tipk. Na ta način bi študenti z pritiskom kombinacije dveh ali treh tipk po hitrem postopku izvedli to, kar morajo sedaj npr. klikati z miško. Takšne hitre tipke bi npr. omogočale dodajanje nove metode, atributa ali notranjega razreda, vrtanje v globino modela, preklapljanje med pogledi ipd. Ena izmed idej študentov je bila tudi, da bi bilo javanske programe mogoče poganjati neposredno iz prototipa, npr. po končanem postopku prevajanja.

Kljub temu, da smo to idejo želeli uresničiti že med izdelavo prototipa, nam tega zaradi specifičnega načina delovanja operacijskega sistema Windows ni uspelo realizirati. Vsak poizkus zaganjanja programa neposredno iz okolja GME je namreč zagnal svoj proces operacijskega sistema, ki se je izvajal v ozadju aktivnih procesov. Zaradi tega ni bilo mogoče izvesti aktivne komunikacije s programom (npr. z vnosom znakov preko tipkovnice), s čimer smo idejo o neposrednem zaganjanju programa morali opustiti.

Študenti v okviru testiranja prototipa niso izpostavili niti vsebinskih niti tehničnih napak. Grajali so edino nestabilnost (»sesuvanje«) modelirnega okolja, za kar je posledično ponovno krivo ogrodje okolja GME. Na vprašanje, ali bi še kdaj sodelovali pri podobnem projektu, so vsi odgovorili pritrdilno.

7. SKLEP

Namen magistrske naloge je bil preveriti kakovost modelirnega okolja GME, ki velja za eno izmed najboljših odprtokodnih modelirnih orodij za izdelavo domensko specifičnih jezikov, z njim izdelati *Prototip orodja za modelno voden razvoj javanskih programov* in preveriti dejansko uporabnost prototipa v praksi s pomočjo študentov 1. letnika študija Računalništvo in informatika. Bistvo preverjanja uporabnosti prototipa je bilo, da ugotovimo smiselnost predstavljenih konceptov modelno vodenega razvoja, s katerimi bi študentom-začetnikom pomagali pri razumevanju osnov objektno usmerjenega programiranja v Javi. Na podlagi prototipa smo prav tako želeli izvedeti, kako prilagodljivo oz. dinamično je okolje GME in ali bi z njim lahko modelirali poljubno problemsko področje.

Glede na lastne izkušnje, ki smo si jih pridobili med iterativno-inkrementalnim razvojem in testiranjem prototipa, lahko rečemo, da je okolje GME preseglo vsa naša pričakovanja, ki smo jih imeli pred začetkom raziskovanja modelirnega okolja. Kot največji adut okolja se je izkazal način opisa problemskega področja, v našem primeru *Razvoj javanskih programov*, s katerim lahko na zelo enostaven način podrobno zajamemo večino sintaktičnih, semantičnih in predstavitvenih informacij obravnavane domene. Metamodeliranje je izvedeno s pomočjo standardiziranega modelirnega jezika UML, ki ga mora znati razumeti vsak načrtovalec modelov. S tem se že takoj na začetku znebimo dodatnih stroškov izobraževanja ustreznih kadrov, ki bi jih morebiti potrebovali pri izbiri drugačnega modelirnega jezika. Metamodeliranje je omejeno na diagramsko tehniko razrednih diagramov, na osnovi katerih lahko zelo podrobno, a enostavno, zajamemo statičen vidik nekega problemskega področja. Ker vsak koncept domene opišemo s pomočjo poljubnega števila atributov, nam ni potrebno skrbeti, da bi zaradi omejitev okolja bili nesposobni opisati katerega izmed pomembnejših konceptov domene. Temu v prid govori tudi dejstvo, da lahko definiramo poljubne tipe relacij med posameznimi modelirnimi gradniki, določimo želene stereotipne razrede in jih obravnavamo kot sestave ali kot nerazdružljive atomarne gradnike domene. Ker lahko v primeru sestavov s pomočjo tehnike »vrtanja v globino« probleme obravnavamo na različnih hierarhičnih nivojih bolj ali manj podrobno, lahko skeptike že takoj na začetku prepričamo v zagotovljeno zmožnost obravnave različno kompleksnih problemov. Prav tako lahko probleme obravnavamo še z različnih zornih kotov oz. t.i. pogledov, ki jih v postopkih izgradnje paradigme natančno definiramo v ustreznem metamodelu. V praktičnem delu magistrske naloge smo pokazali in si celo upamo trditi, da si kateregakoli drugega modelirnega okolja brez obeh načinov obravnave zahtevnih problemov sploh ne moremo zamisliti.

Kljub temu, da je v predstavitveni dokumentaciji okolja GME govora o enostavnosti in učinkovitosti omejitvenega jezika OCL, s katerim v okviru okolja definiramo potrebne omejitve vrednosti konceptov domene, smo glede uporabnosti tega mehanizma v praksi nekoliko skeptični. Kljub temu, da je omejitveni jezik v svoji osnovi relativno enostaven za

opis preprostih omejitev, smo za izdelavo bolj kompleksnih omejitev, ki med seboj povezujejo več različnih vrednosti modelirnih gradnikov, potrebovali kar nekaj časa. Glavni vzrok temu je slaba realizacija urejevalnika OCL omejitev, ki načrtovalcu domene prav nič ne pomaga (z različnimi sugestijami, predlogami ipd.) pri sprotnem oblikovanju omejitev. Največkrat smo dejansko (ne)pravilnost omejitev morali preizkusiti na konkretnem primeru - modeliranem javanskem programu. Bolj enostavno bi bilo, če bi že med načrtovanjem lahko vedeli, ali smo omejitev načrtovali pravilno ali ne. Prav tako nismo našli dobre podpore (dokumentacije) obravnavanih knjižnic, ki bi nam z nekaj preprostimi primeri lahko pokazala način najboljše uporabe omejitev. Določene omejitve, ki jih zaradi načina predstavitve informacij v okolju GME ni bilo mogoče realizirati z jezikom OCL, smo enostavno in hitro izdelali v okviru interpreterja. Ravno v primeru omejitev OCL smo naleteli na nekaj večjih pomanjkljivosti okolja, zaradi česar nam je izdelava prototipa vzela dalj časa, kot smo sprva predvideli.

S pomočjo interpreterjev v okolju GME neposredno dostopamo do informacij modela in jih uporabimo oz. preoblikujemo na poljuben način. Ker smo na domačih straneh okolja GME našli dovolj dokumentacije o uporabi javanskih vmesnikov, preko katerih dostopamo do želenih gradnikov modela, nismo imeli večjih težav pri izdelavi interpreterja. Lahko rečemo, da smo s postopki neposredne interakcije z modelom, ki so v splošnem obravnavani kot ena izmed najbolj pogostih tehnik transformacij med modeli, modelirane javanske programe na preprost in učinkovit način preoblikovali v ustrezno javansko programsko kodo. Pri tem smo hierarhično strukturo javanskega modela obdelovali z rekurzivnimi metodami sestopanja, kar je učinkovitost opisanega postopka še povečalo.

Zaradi pričakovanih omejitev okolja GME in prevelike kompleksnosti posameznih zahtev obravnavane domene, nekaterih stvari v okviru prototipa magistrske naloge nismo bili sposobni izvesti. Med njimi je najbolj očitna nezmožnost zavedanja semantične in tudi sintaktične pravilnosti programske kode telesa metod v času modeliranja javanskega programa. Zaradi tega v prototip nismo mogli integrirati funckije *IntelliSense*, ki jo programerji poznajo kot eno izmed najuporabnejših funkcij dobrih razvojnih orodij. Velika slabost prototipa je tudi ta, da se z uvozom standardiziranih knjižnic oz. paketov prototip ne zaveda njihove vsebine. Kljub temu, da bi takšna funkcionalnost dodano vrednost praktičnega izdelka magistrske naloge zelo povečala, si te rešitve zaradi izjemne kompleksnosti implementacije najbrž ne bi mogli privoščiti. Kot zadnjo večjo slabost smo omenili predstavitev modelirnih informacij prototipa, ki jih na enostaven način ne moremo dodatno razširiti s katerimi drugimi »vizualnimi« pripomočki. Takšen primer bi npr. bili vizualizacija hierarhično nadrejenega modela, preprost iskalnik informacij na nivoju celotne modelirane javanske naloge, hiter vpogled v druga telesa metod razredov ipd. Nekaj manjših pripomb smo imeli tudi v zvezi z dinamiko okolja GME. Javanski vmesniki, preko katerih dostopamo do informacij modela, nam ne omogočajo nikakršne interakcije z modelom. Kot primer lahko izpostavimo možnost premikanja posameznih modelirnih gradnikov z enega mesta na drugo,

generično dodeljevanje vrednosti atributom modelirnih gradnikov, ipd. Takšne možnosti že sedaj omogoča vmesnik okolja GME, ki ga lahko realiziramo s pomočjo programskega jezika C++.

Študenti so v okviru ankete, ki so jo izpolnili ob koncu uporabe prototipa, tudi sami navedli nekaj pomanjkljivosti prototipa. Poleg že omenjenih so kot eno izmed največjih pomanjkljivosti izpostavili nezmožnost barvanja programske kode telesa metod. Pri tem je še enkrat potrebno povedati, da je tukaj bolj kot samo barvanje kode mišljena že omenjena neprisotnost funkcije *IntelliSense*, ki so jo študentje očitno resnično pogrešali. Kljub strinjanju z njimi, smo morali izpostaviti skrb, da takšna lagodnost modelirnih orodij lahko programerju-začetniku prej škoduje kot koristi. Razlog temu je prevelika pomoč razvojnih orodij, od katere lahko začetniki prepogosto postanejo odvisni. Kot naslednjo večjo slabost prototipa so študenti navedli njegovo nestabilno delovanje (sesuvanje), kar lahko v celoti pripišemo nedovršeni podpori okolja GME in posameznim različicam operacijskega sistema Windows. Nekaj očitkov je bilo izpostavljenih tudi na račun obravnave in prikaza napak interpreterja, kar bi z manjšimi razširitvami okolja GME lahko tudi izboljšali. Dva izmed študentov je motila tudi izbira modelirnih simbolov nekaterih javanskih konceptov in sta bila mnenja, da njihovo razumevanje uporabnikom vzame preveč časa.

Kljub navedenim slabostim oz. pomanjkljivostim smo od študentov dobili pozitiven odziv glede uporabe prototipa. Študenti so s prototipom v večini modelirali vse zastavljene naloge. Še boljše rezultate bi dosegli z večjo stabilnostjo okolja GME in nekaj odpravljenimi pomanjkljivostmi, ki so študente spravljale v slabo voljo. Kljub temu, da pred uporabo prototipa študenti niso dobro poznali modelno vodenega razvoja, so se z njegovo uporabo jasno zavedali prednosti, ki jim jih pred tekstualno zapisano programsko kodo ponujajo višje abstraktni modeli. Modelirne koncepte javanskih programov, ki smo jih predstavili s prototipom, so sprejeli pozitivno. Tisti, ki poznajo enega ali več objektno usmerjenih jezikov, so bili mnenja, da je predstavljeni modelirni način razvoja smiseln za programski jezik Java.

Glede zasnove prototipa študenti niso imeli večjih pripomb. Bili so mnenja, da si s pomočjo vizualne percepcije posamezne objektne koncepte predstavljajo dosti lažje in se na njih osredotočijo bolje kot pri klasičnem programiranju. Kljub vsemu so bili prepričani, da se pri dejanskem programiranju telesa metod glede na trenutno zasnovo prototipa znajdejo hitreje pri klasičnih razvojnih orodjih, s čimer smo se strinjali. Prototip je pač prototip, z vsemi svojimi slabostmi, ki jih lahko postopoma odpravljamo le s povratnimi informacijami ob njegovi dejanski uporabi v praksi. Glede tehnike t.i. »vrtanja v globino« so se študentje strinjali z nami, da je to edini smiseln način prikaza hierarhične strukture modeliranega javanskega programa. Prav tako so bili mnenja, da je v okviru modeliranja metode, atribute in notranje razrede smiselno združevati glede na sorodnost vsebine, s čimer se zagotovo strinjamo tudi mi.

Skupaj s študenti lahko predlagamo še nekaj realnih izboljšav prototipa, ki bi jih bilo mogoče doseči z nadaljnjim raziskovalno-razvojnim projektom. Najbolj zanimivo bi bilo izdelati interpreter, s pomočjo katerega bi javansko programsko kodo z obratnim postopkom t.i. tehnike »Reverse Enginering« preoblikovali v ustrezne modele javanskega programa. Na ta način bi pomagali programerjem-začetnikom, ki si objektne koncepte programskega jezika Java na predavanjih in laboratorijskih vajah težko predstavljajo. Z uporabo hitrih tipk ali t.i. bližnjic bi zagotovili lažjo in hitrejšo interakcijo s prototipom. Pogled *Koda* bi lahko razširili z možnostjo dodajanja notranjih abstraktnih razredov in vmesnikov ter omogočili razširjanje oz. implementacije notranjih razredov in vmesnikov. Omejitve OCL bi lahko razširili z nekoliko manj strogimi pravili, ki bi uporabnike opozarjali na določene pomanjkljivosti modelirane javanske naloge. Poleg tega bi s ponovnim sodelovanjem s študenti lahko oblikovali še bolj jasne simbole modelirnih gradnikov, ki bi programerjem-začetnikom pomagali hitreje razumeti modelirane javanske koncepte.

Za konec magistrske naloge lahko še povemo, da smo z izdelano nalogo dosegli vse cilje, ki smo si jih zastavili na začetku. Dodobra smo se spoznali s koncepti modelno vodenega razvoja programske opreme, naučili smo se rokovati z odličnim odprtokodnim orodjem GME za razvoj domensko specifičnih jezikov in z njim izdelali prototip orodja za modelno voden razvoj javanskih programov, ki dobro služi svojemu namenu. Prototip smo povrhu vsega predali v testiranje izbrani množici študentov 1. letnika, ki so z zastavljeno anketo podali svoje mnenje glede njegove uporabe v praksi. Po opravljeni anketi smo naredili podrobno analizo in rezultate povzeli v zaključku naloge.

V tem trenutku si lahko rečem samo: »Uspelo mi je!« ;)

8. PRILOGA

8.1 ANKETA – PROTOTIP ORODJA ZA MODELNO VODEN RAZVOJ JAVANSKIH PROGRAMOV

Anketa

PROTOTIP ORODJA ZA MODELNO VODEN RAZVOJ JAVANSKIH PROGRAMOV

ocenitev prototipa JCP (Java Class Project)

Za vsako izmed spodaj navedenih vprašanj izberite ustrezen odgovor (vedno le enega, le če je navedeno drugače). Izbrani odgovor obkrožite ali ga kako drugače označite. V primeru opisnega odgovora podajte jasne, kratke in razumljive stavke, trditve oz. dejstva. Odgovore lahko podate tudi v ločeni datoteki (z oznakami posameznih vprašanj)!

I DEL – SPLOŠNO O VAŠEM ZNANJU IZ PROGRAMIRANJA IN MODELNO USMERJENEM RAZVOJU

1. Ali ste znali programirati že pred prihodom na fakulteto?
    a. DA, že v srednji šoli sem se naučil(a) programirati z vsaj enim programskim jezikom
    b. NE, programirati sem se naučil(a) šele na fakulteti
2. Kako bi ocenili svoje znanje programskega jezika Java po končanem **prvem semestru**?
    a. Odlično sem razumel(a) in znal(a) uporabljati zahtevano znanje iz Osnov programiranja I
    b. Težje snovi (razredi, dedovanje, prilagajanje tipa, rekurzija, ipd.) iz OP I nisem razumel(a) v celoti, sem pa poznal(a) osnove
    c. Imel(a) sem težave pri razumevanju snovi iz OP1 – programski jezik Java sem spoznal(a) le v grobem
3. Kako bi ocenili to znanje glede na ostale kolege v letniku?
    a. Znal(a) sem mnogo več kot ostali
    b. Vedel(a) sem nekoliko več od ostalih
    c. Menim, da sem bil(a) po znanju nekje v zlati sredini
    d. Znal(a) sem nekoliko manj od ostalih
    e. Drugi so znali občutno več od mene

4. Ali ste s prototipom JCP **sami** izdelali vse od vas pričakovane naloge?
    a. DA
    b. NE

    Zakaj NE ?:

    _____
    _____
    _____
    _____
    _____

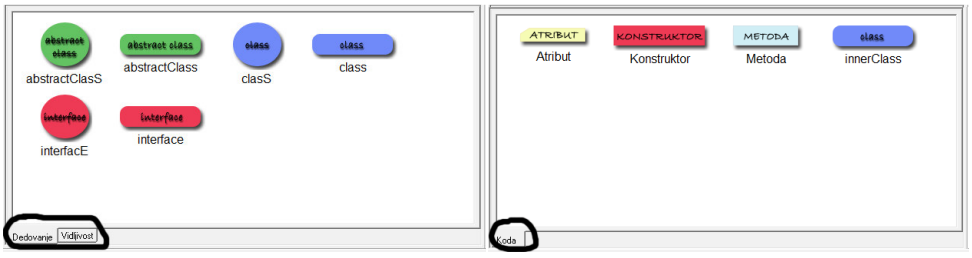
5. Ali ste naloge izdelali brez pomoči »klasičnega« programiranja?
    a. DA, vsaj 5 nalog sem izdelal izključno s prototipom JCP
    b. NE, naloge sem sprogramiral na klasičen način, in jih nato prekopiral v prototip (s t.i. »copy/paste« načinom).
6. Ali ste prototip uporabljali tudi sicer?
    a. DA, kar nekajkrat
    b. DA, enkrat ali dvakrat
    c. NE, s prototipom sem izdelal(a) le zahtevane naloge
7. Ali ste pred uporabo prototipa JCP že kaj vedeli o modelno usmerjenem razvoju programske opreme?
    a. DA, o tem pristopu oz. načinu razvoja programske opreme sem vedel(a) že kar nekaj (ali sem se z njim že srečal(a))
    b. Slišal(a) sem že za ta pristop, kaj več pa ne
    c. NE, prej nisem vedel(a) popolnoma nič o modelno usmerjenem razvoju
8. Ali menite, da ste se z uporabo prototipa JCP zavedali vsaj nekaterih prednosti (pred klasičnim programiranjem), ki bi vam jih ponuja modelno usmerjen razvoj?
    a. DA
    b. NE
    c. NE VEM
9. Ali vam je **SAM KONCEPT** modelno usmerjenega razvoja, ki ste ga spoznali ob uporabi prototipa JCP (torej predstavitev ogrodja javanskega programa s pomočjo modelov in grafičnih simbolov, združevanja ročno napisane in generirane programske kode, poudarek na vizualni predstavitvi programa ipd.), v spošnem bolj uporaben, nazoren in učinkovit od klasičnega programiranja (pri tem v zakup vzamite slabosti prototipa, kot so nezmožnost funkcije *IntelliSense* (glej: http://en.wikipedia.org/wiki/IntelliSense), barvanja programska kode, vizualne dovršenosti prototipa, reverse engineeringa ipd.)?
    a. DA, menim, da bi na takšen način mnogo hitreje in bolj učinkovito razvijal(a) programsko kodo

b. NE, »čisto« klasično programiranje mi je še vedno ljubše
c. Težko ocenim, saj s prototipom nisem mogel(mogla) ugotoviti dejanske uporabnosti/vrednosti takšnega načina razvoja programske opreme

10. Ali menite, da bi se programer-začetnik (z nekim osnovim predznanjem iz programiranja, recimo iz Osnov programiranja I) z dobro zastavljenim modelno usmerjenim orodjem za razvoj javanskih programov (takšnim, ki razširja funkionalnosti prototipa JCP) naučil **naprednega** programiranja (takšnega, ki ste ga spoznali pri Osnovah programiranja II) hitreje kot s klasičnim programiranjem?
    a. Zagotovo, saj ljudje vizualne informacije pomnimo in obdelujemo bolje in hitreje kot tekstualne
    b. NE, klasično programiranje (z dobrim orodjem) je zagotovo boljše – modelno usmerjen razvoj bi programerja-začetnika samo oviral
    c. NE VEM
11. Če poznate še ostale programske jezike, tudi tiste, ki ne temeljijo na objektnem programiranju, ali menite, da je modelno usmerjen razvoj (takšen, ki sledi načelom prototipa JCP) sploh smiselen za programski jezik Java?
    a. DA
    b. NE
    c. NE ZNAM OCENITI
    d. NE MOREM OCENITI, ker ne poznam drugih programskih jezikov
12. Kaj bi rajši uporabljali: klasično razvojno orodje (kot je npr. Eclipse oz. JBuilder) ali modelno usmerjeno razvojno orodje, ki sledi načelom prototipa JCP?
    a. Klasično razvojno orodje
    b. Orodje za modelno usmerjen razvoj, ki vključuje tudi **najpomembnejše** funkcionanosti klasičih razvojnih orodij (npr. barvanje kode in funkcijo IntelliSense)

II DEL – PODROBNO O PROTOTIPU JCP

13. Ali se vam zdi smiselen način predstavitve in razvoja programa s pomočjo treh v prototipu JCP predstavljenih vidikov (Dedovanje, Vidljivost, Koda)?



    a. DA, vidiki so smiselno zastavljeni tako, da si je z njimi na LOGIČEN način mogoče JASNO predstavljati program v razvoju. Vsak vidik posebej obravnava svoj del strukture programa (ali delov programa), na katerega se kot

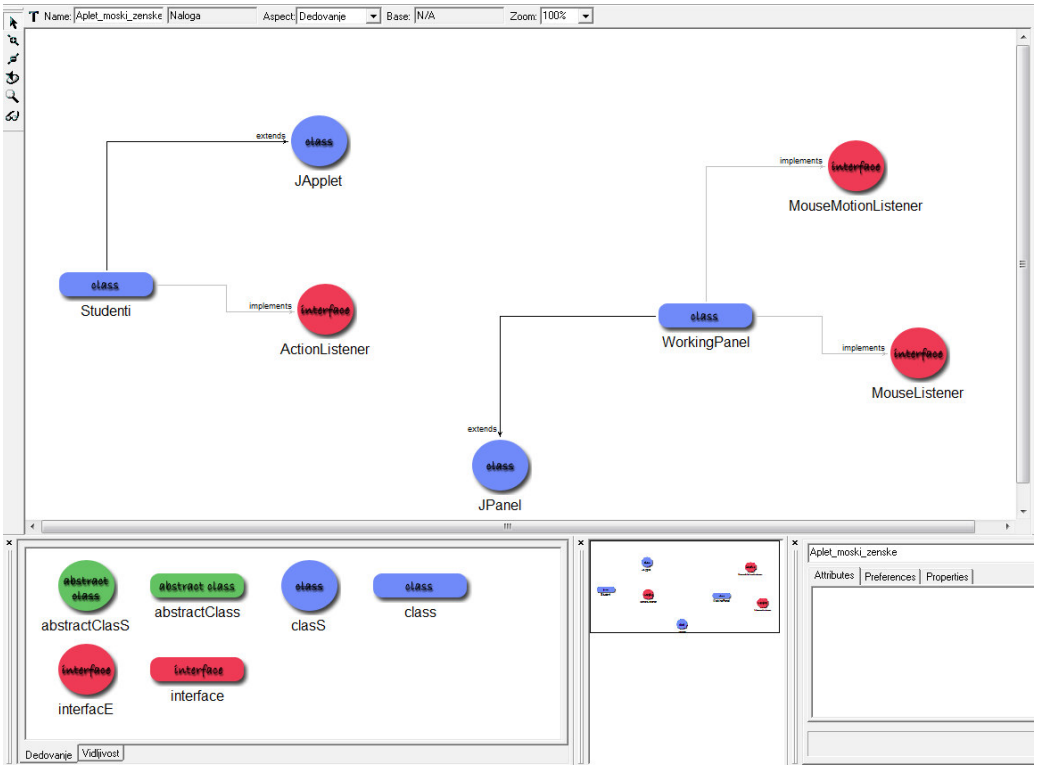
na neko zaokroženo celoto lahko osredotočimo bolj kot pri klasičnem programiranju (npr. če opazujem metode, se osredotočim na vidik »Koda«; če me zanima dedovanje med razredi, uporabim vidik »Dedovanje«; če me zanima organiziranost razredov po datotekah, to ugotovim z vidikom »Vidljivost«, ipd.)

b. NE, vidiki so zastavljeni NELOGIČNO oz. NEPOVEZANO – na takšen način bi v praksi zelo težko razvijal uporabne programe. Bolj kot iz modelov in različnih vidikov se znajdem le iz pobarvane in dobro strukturirane programske kode

Sam bi stvari zastavil(a) nekoliko drugače (**LAHKO** dopolnite, če ste se odločili za odgovor **b**):

_____
_____
_____
_____
_____
_____
_____
_____
_____

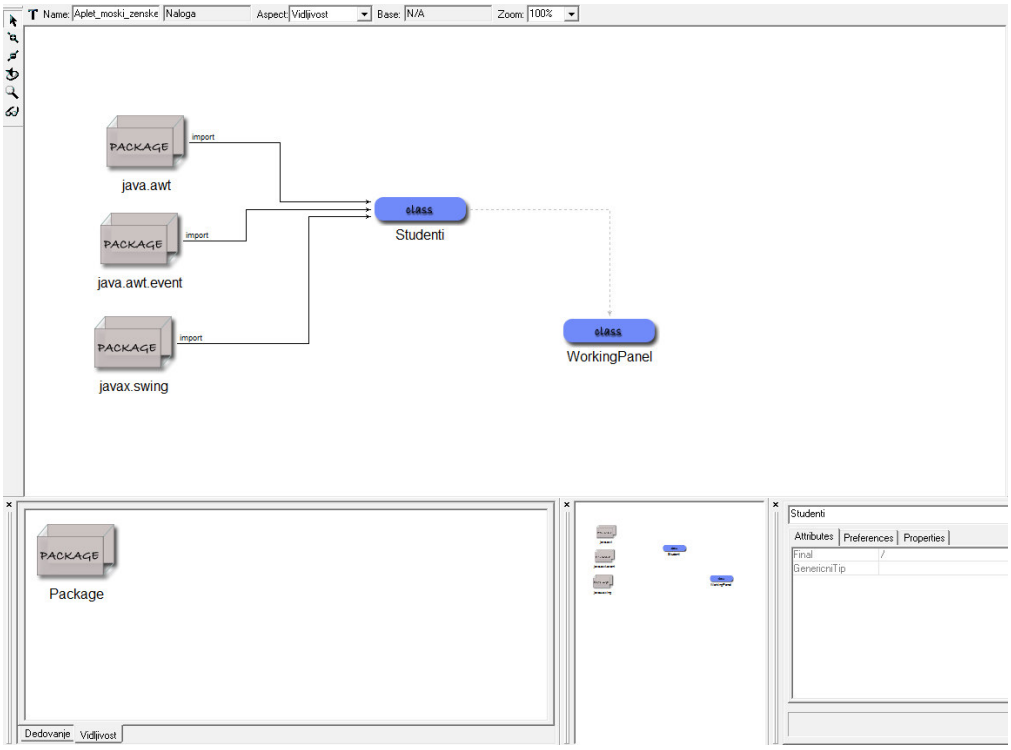
14. Kako bi ocenili vidik »Dedovanje« **(izberete lahko nič, enega, dva ali tri odgovore)**



a. S pomočjo usmerjenih povezav je mogoče jasno prikazati koncepta dedovanja in implementacije vmesnikov
b. Dobra in uporabna se mi zdi predstavitev z različnimi barvami: razredov z modro barvo, abstraktnih razredov z zeleno barvo in vmesnikov z rdečo barvo
c. Smiselna se mi zdi uporaba različnih simbolov (kroga ter elipse) in prečrtanih imen ~~class~~, ~~abstract class~~ in ~~interface~~ za ločevanje med ročno napisanimi in uporabljenimi razredi oz. vmesnilki: elipsa predstavlja razrede oz. vmesnike, ki jih bom razvil sam, krog (s prečrtanim imenom) pa razrede oz. vmesnike, ki jih bom v svojem programu le uporabil (in so že na voljo v drugih paketih)

Sam bi stvari zastavil(a) nekoliko drugače (**LAHKO** dopolnite ne glede na vaše odgovore):

15. Kako bi ocenili vidik »Vidljivost« **(izberete lahko nič, enega ali dva odgovora)**

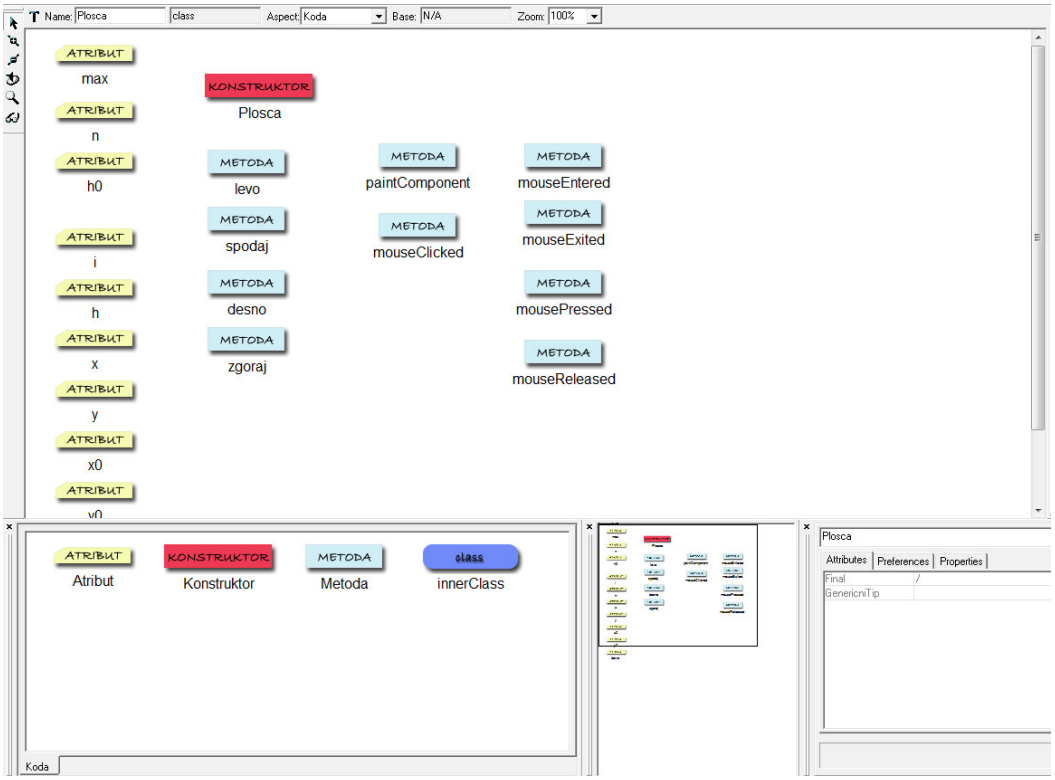


a. S pomočjo črtkanih usmerjenih povezav je mogoče jasno prikazati organiziranost razredov po datotekah
b. S pomočjo polnih usmerjenih povezav se lepo predstavi vključenost različnih paketov po razredih (npr. java.util, java.awt ipd.)

Sam bi stvari zastavil(a) nekoliko drugače (**LAHKO** dopolnite ne glede na vaše odgovore):

_____
_____
_____
_____
_____
_____
_____
_____
_____

16. Kako bi ocenili vidik »Koda«



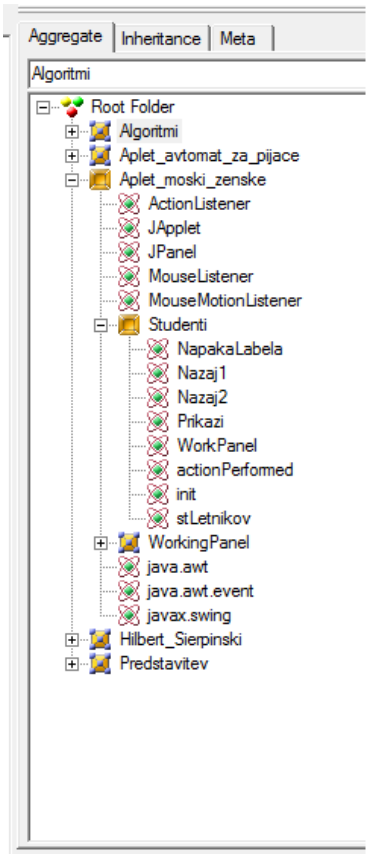
   a. S pomočjo vidika »Koda« je programsko kodo mogoče razvijati bolj učinkovito kot pri klasičnem programiranju: iskano metodo, atribut ali notranji razred najdem hitreje (in bolj pregledno določim njihove lastnosti, kot so npr. vidiljivost, statičnost ipd.), prav tako pa si lažje predstavljam celotno strukturo programa
   b. S takšnim načinom »programiranja« si je strukturo programa sicer mogoče predstavljati bolj jasno, vendar pri tem težko dosežem enako ali celo večjo raven učinkovitosti kot pri klasičnem programiranju
   c. Iz dobro napisane, pobarvane in komentirane programske kode (klasičnega programiranja) se znajdem hitreje in bolje kot pri grafični predstavitvi metod, atributov in notranjih razredov

17. Ali se vam znotraj vidika »Koda« zdi smiselna uporabna tehnike t.i. »vrtanja v globino« (npr. znotraj razreda s svojimi metodami in atributi se lahko naredi notranji razred: klik nanj prikaže njegove metode in atribute, znotraj njega se spet lahko naredi notranji razred: klik nanj prikaže njegove meotde in atribute...)
   a. DA, edino tako je mogoče smiselno prikazati hierarhično strukturo programa na grafičen način
   b. NE, tehnika »vrtanja v globino« kompleksnost izdelave programa samo poveča

Sam bi stvari zastavil(a) nekoliko drugače (**LAHKO** dopolnite, če ste se odločili za odgovor **b**):

_____
_____
_____
_____
_____
_____
_____
_____
_____

18. Ali menite da je posamezne grafične simbole metod oz. atributov razreda znotraj vidika »Koda« med seboj smiselno združevati po skupinah (npr. združevanje kontruktorjev, SET() metod, GET() metod, metod za izpis podatkov, ipd.)
    a. DA, tako je hitreje in lažje vidna struktura programa (in njegovih sestavnih delov)
    b. NE, metode in atribute razreda sem do sedaj dodajal kar enega zraven drugega, brez kakršnegakoli reda
19. Ali se vam zdi uporabna navigacija po posameznih elementih programa (metod, atributov, notranjih razredov...) v desnem navigacijskem meniju?



a. DA
b. DA, vendar do sedaj navigacijskega menija še nisem uporabljal (ni bilo potrebe)
c. NE

20. Ali menite da bi bil t.i. reverse engineering pri modelno usmerjenem orodju za razvoj javanskih programov nujen (program bi na podlagi vhodnih *.JAVA datotek zgeneriral modele)?
    a. DA
    b. NE
    c. NE VEM
21. Ali ste pri uporabi prototipa JCP kaj pogrešali oz. se vam kaj ni zdelo logično/učinkovito/enostavno?

_____
_____
_____
_____
_____

22. Kakšni bi bili vaši napotki ali ideje za izboljšanje prototipa JCP?

23. Ali ste pri testiranju prototipa JCP našli kakšen BUG ali večjo napako?

24. Ali bi v prihodnje želeli sodelovali še na kakšnem podobnem projektu?
    a. DA
    b. NE

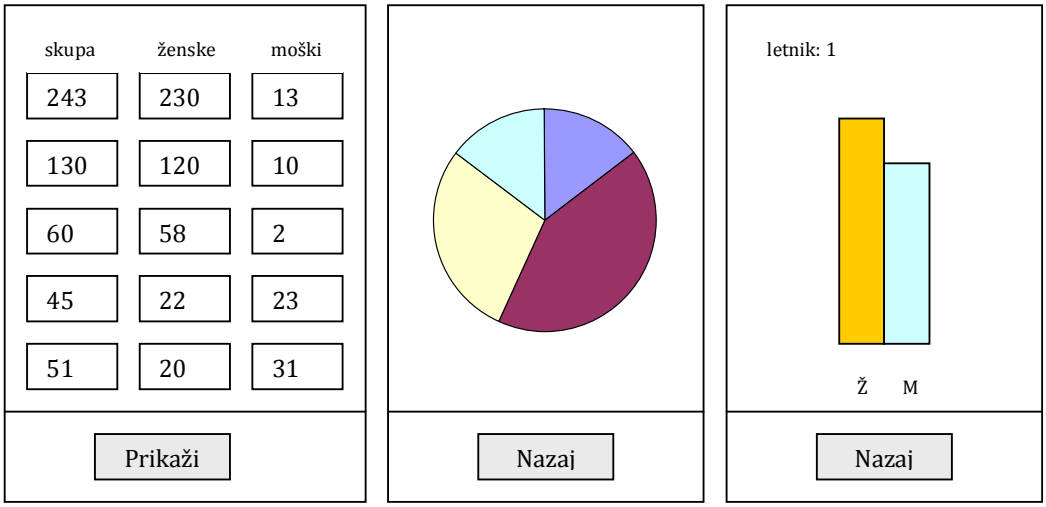
Osnove programiranja II (IŠRM)

Laboratorijske vaje

1. Struktura vpisanih študentov

Napišite aplet, s katerim bo mogoče na grafični način prikazati strukturo števila študentov po posameznih letnikih nekega študijskega programa. Aplet naj najprej prebere število letnikov študijskega programa (veljavne vrednosti so od 3 do 6) iz parametra z imenom STEVILO_LETNIKOV, ki je podan na HTML strani. Na podlagi števila letnikov naj se na začetku pojavi ustrezno število vnosnih polj, ki omogočajo zajem skupnega števila študentov letnika, število moških in število žensk. Pri zajemu podatkov je treba preverjati ustreznost podatkov (skupno število študentov letnika je enako vsoti števila moških in žensk).

Ob kliku na gumb »Prikaži« naj aplet prikaže zajete podatke o skupnem številu študentov v letniku v obliki krožnega grafa (angl. pie chart). S klikom na določen odsek tega grafa naj se prikaže še delež moških in žensk in sicer v obliki stolpičnega grafa. Oba grafična prikaza naj vsebujeta tudi gumb »Nazaj«, ki omogoča vrnitev na predhodni
izpis



2. Sortirni algoritmi

Napišite program v Javi, ki prikazuje sortiranje tabele objektov za enega izmed spodnjih razredov. Ti predstavljajo objekte različnih skupin oseb, pri čemer je vsak razred razširitev razreda Oseba, ta pa je razširitev abstraktnega razreda Element. Za vsako osebo poznamo ime, priimek, datum rojstva (objekt tipa Datum, ki vsebuje dan, mesec in leto) in spol, medtem ko imajo pripadniki posameznih skupin še

dodatne lastnosti. **Vsak študent naj izbere skupino, ki je določena kot njegova vpisna številka MOD 4** (recimo študent z vpisno številko 63050987 naj realizira razred Varčevalec – 63050987 MOD 4 = 3).

1. **Sportnik:** sport, prvo leto nastopanja, trenutni status, podatek, če je športnik še aktiven
   Primer: ("Andrej", "Bikar", (3,11,1969),"M"), "tek", 1982, "vrhunski športnik", false)
2. **Popotnik**: klub, leto prvega potovanja, tip popotnika, podatek, če popotnik uporablja avto
   Primer: ("Jana", "Pintar", (3,3,1981),"Z"), "Krizkraz", 1999, "štopar", false,)
3. **Varcevalec:** banka, leto odprtja računa, tip varčevalca, podatek, če ima varčevalec kredit
   Primer: ("Marija", "Perko", (19,3,1951),"Z"), "Banka Koper", 1975, "fizična oseba", false)
4. **Glasbenik**: skupina, leto prvega nastopa, tip glasbenika, podatek, če je včlanjen v SAZAS
   Primer: ("Jan", "Drofenik", (29,02,1976),"M"), "TheBand", 1997, "slovenski glasbenik", false)

Program naj omogoča uporabo naslednjih naslednjih algoritmov za sortiranje:

- **navadno vstavljanje (straight insertion)**
- **dvojiško vstavljanje (binary insertion)** – mesto za vstavljanje elementa iščemo s pomočjo bisekcije
- **navadno izbiranje (straight selection)**
- **sortiranje s porazdelitvami (quicksort)**
- **sortiranje s porazdelitvami – iterativno (quicksortI)**

Program naj omogočajo tako **naraščajoče** (od najmanjše do največje vrednosti) kot **padajoče** sortiranje z vsakim od naštetih algoritmov. Sortiranje pa naj bo mogoče **po vsaki lastnosti** razreda (torej po imenu, priimku, …). Izjemoma naj bo pri datumih sortiranje omogočeno le po datumu kot celoti.

Primer izpisa:

```
Razred: Varčevalec
Atribut: Leto odprtja računa
Metoda: Navadno vstavljanje
Urejanje: Naraščajoče
Število elementov: 6

»1975« »1980« »1955« »1960« »1999« »1989«
»1975« »1980« »1955« »1960« »1999« »1989«
»1955« »1975« »1980« »1960« »1999« »1989«
»1955« »1960« »1975« »1980« »1999« »1989«
»1955« »1960« »1975« »1980« »1999« »1989«
»1955« »1960« »1975« »1980« »1989« »1999«
```

3. Leksikografsko urejanje nizov

Napišite program v Javi, ki prikazuje **leksikografsko urejanje tabele nizov** in sicer tako **naraščajoče** (od A do Z) kot **padajoče** (od Z do A). Pred urejanjem tabelo nizov napolnite iz izbrane datoteke ASCII znakov (besedila), ki naj ne vsebuje več kot 1000 nizov. Program naj nato najprej izpiše neurejeno tabelo nizov, po (naraščajočem ali padajočem) urejanju pa še pravilno urejeno tabelo nizov. Urejeno tabelo naj program tudi prepiše v izbrano ASCII datoteko.

Zaradi enostavnosti predpostavite, da ne ločimo med velikimi in malimi črkami, besedilo pa je sestavljeno le iz 26-ih črk angleške abecede (A - Z) ter iz 10-ih cifer (0 - 9). Vse ostale znake (presledke, ločila, ipd.) pri prepisovanju iz tabele upoštevajte kot zaključek trenutnega niza. Pri

prepisovanju si zapomnite tudi dolžino najdaljšega niza, ki jo nato upoštevajte pri leksikografskem urejanju.

Datoteka:
Tezko je razloziti, kaj vse je java 1.5.
V enem stavku bi lahko rekli: java je podlaga za nov rod programske opreme.

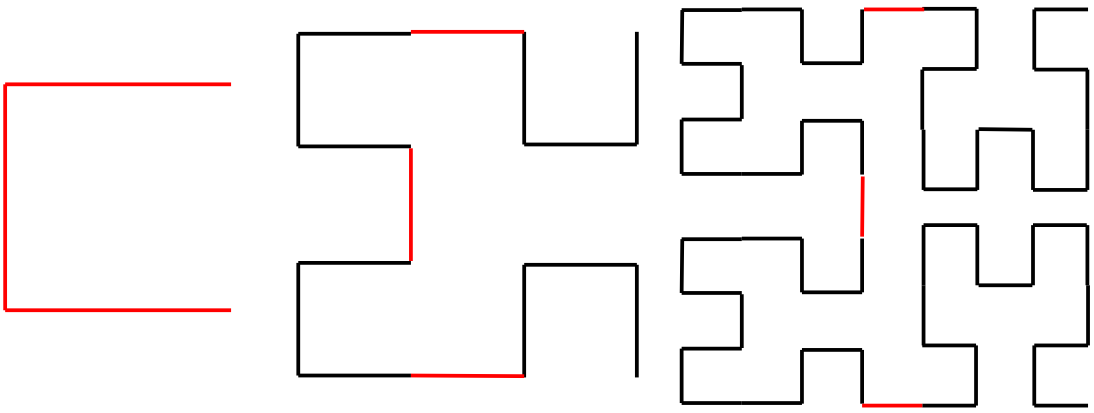
Neurejena tabela nizov:
TEZKO JE RAZLOZITI KAJ VSE JE JAVA 1 5 V ENEM STAVKU BI LAHKO REKLI JAVA JE PODLAGA ZA NOV ROD PROGRAMSKE OPREME

Naraščajoče urejena tabela nizov (in datoteka):
1 5 BI ENEM JAVA JAVA JE JE JE KAJ LAHKO NOV OPREME PODLAGA PROGRAMSKE RAZLOZITI REKLI ROD STAVKU TEZKO V VSE ZA

4. Rekurzivno risanje krivulj

Napišite program v Javi, ki na rekurziven način izriše Hilbertove krivulje i-tega reda (označeno s H_i), kot jih je opisal njihov izumitelj D. Hilbert (1891). Hilbertovo krivuljo dobimo s sestavljanjem štirih ustrezno orientiranih krivulj H_{i-1} polovične velikosti, ki so medsebojno povezane s tremi ravnimi črtami. Krivuljo H_1 si lahko predstavljamo kot štiri "prazne" krivulje H_0 povezane s tremi ravnimi črtami.

Na spodnji sliki so predstavljene Hilbertove krivulje od prvega do tretjega reda. Povezave so prikazane v rdeči barvi.



H_1 H_2 H_3

Namig: ker je vsaka krivulja H_i sestavljena iz štirih krivulj H_{i-1} polovične velikosti, lahko napišemo metodo za risanje H_i kot štiri rekurzivne klice metod za izris H_{i-1} v ustrezni velikosti in orientaciji in treh ustreznih povezovalnih črt. Različne velikosti določimo s parametrom, medtem ko štiri različno usmerjene krivulje realiziramo s štirimi metodami. Krivulja H_1 na zgornji sliki je levo obrnjena krivulja in je sestavljena iz :

- navzdol obrnjene krivulje enega reda manj,
- kratke vodoravne črte,
- levo obrnjene krivulje enega reda manj,
- kratke navpične črte,
- levo obrnjene krivulje enega reda manj,
- kratke vodoravne črte in
- navzgor obrnjene krivulje enega reda manj.

5. Generična vrsta

Napišite program v Javi, ki realizira **vrsto** kot **povezan linearni seznam** objektov, pri čemer naj bo vrsta predstavljena kot **generični razred**. Pri tem izhajajte iz vmesnika *Queue* in razredov *AbstractQueue* ter *LinkedQueue*, kjer slednjega ustrezno spremenite oziroma dopolnite. Upoštevajte, da imajo objekti v linearnem seznamu povezavo samo s svojim naslednikom (in ne s predhodnikom), generični razred pa omogoča shranjevanje objektov izbranega tipa.

Rešitev naj vsebuje konstruktor, ki ustvari prazno vrsto, ustrezno realizacijo *iteratorja* ter štiri osnovne operacije:

- dodajanje na konec vrste (*enqueue*),
- izločanje prvega elementa iz vrste (*dequeue*),
- izpis prvega elementa vrste (*getFront*) in
- izpis zadnjega elementa vrste (*getBack*).

Poleg osnovnih operacij realizirajte še nekaj drugih, ki omogočajo (enostavnejši) prikaz uporabe:

- dodajanje večjega števila elementov iz tabele, podane kot vhodni parameter, na konec vrste,
- praznjenje vrste,
- primerjavo enakosti dveh vrst (vrsti sta enaki, če vsebujeta enake elemente, ki v obeh vrstah nastopajo v enakem vrstnem redu),
- izpis števila elementov v vrsti,
- izpis vseh elementov vrste (z uporabo iteratorja).

Napišite tudi testni razred *TestVrste*, ki omogoča interaktiven prikaz uporabe vseh operacij nad objekti tipa, ki ste ga uporabljali v drugi seminarski nalogi (*Sportnik*, *Popotnik*, *Varcevalec*, *Glasbenik*). Testni razred naj omogoča prikaz delovanja vseh operacij na interaktiven način tako, da zaporedoma izvajamo operacije v poljubnem vrstnem redu nad izbrano vrsto.

Za preverjanje enakosti vsebine dveh vrst potrebujemo vsaj dve vrsti, zato je potrebno deklarirati tabelo vrst in nato pred vsako operacijo izbrati tisto vrsto, nad katero naj se operacija izvede.

6. Elektronski indeks

Napišite program v Javi, ki s pomočjo razredov **HashMap** in **LinkedList** omogoča hrambo podatkov iz indeksov študentov v elektronski obliki. Zapisi oblike **<ključ, vrednost>** naj se hranijo v zaprti razpršeni tabeli (razred HashMap), kjer je ključ posameznega zapisa objekt razreda **Student**, vrednost pa je objekt razreda **Indeks**.

Razred Student naj vsebuje vpisno številko študenta, njegovo ime in priimek. Dva objekta tega razreda sta enaka, če se ujemata po vpisnih številkah.

Razred Indeks naj vsebuje dva linearna seznama (objekta razreda LinkedList). V prvem seznamu naj se nahajajo vsi predmeti (razred **Predmet**), ki jih ima študent v predmetniku, v drugem pa so zabeležena vsa opravljanja izpitov (razred **Izpit**) pri teh predmetih.

Razred Predmet je opisan s številko predmeta, imenom predmeta, letnikom v katerem se predmet predava, trenutno veljavno oceno izpita in vaj ter datumom zadnjega opravljanja izpita (zaradi enostavnosti predstavite datum kot razred String). V kolikor študent izpita pri predmetu še ni opravljal sta obe oceni in datum zadnjega opravljanja izpita nedefinirani.

Razred Izpit je opisan s številko predmeta, oceno izpita, oceno vaj ter datumom opravljanja izpita.

Ko se študentu v predmetnik doda nov predmet, ta še nima ocene izpita in vaj (vrednost nastavite na *-1*) ter datuma zadnjega opravljanja izpita (vrednost nastavite na null).

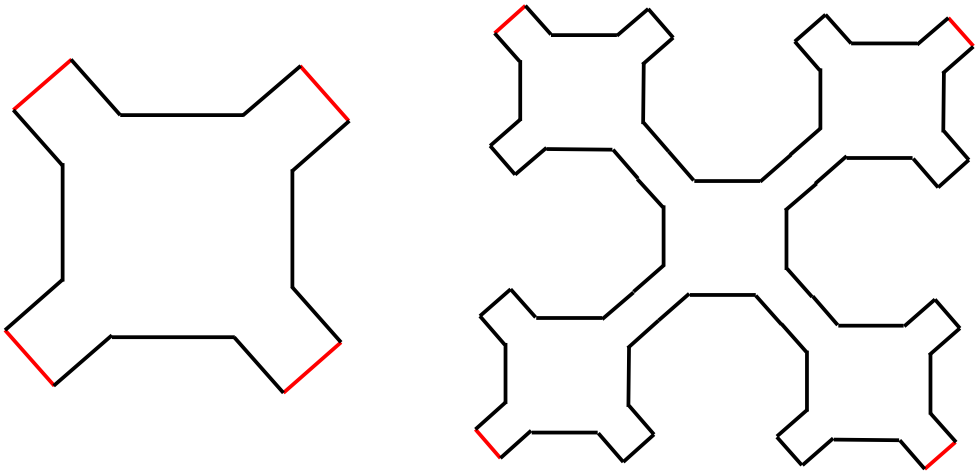
V primeru prvega oz. ponovnega opravljanja izpita se le-to zabeleži v seznam opravljanja izpitov (z doseženo ocena izpita, oceno vaj ter datumom opravljanja izpita), popravita pa se tudi obe oceni in datum zadnjega opravljanja izpita ustreznega predmeta v predmetniku.

Program naj omogoča:

- dodajanje podatkov novega študenta v hrambo (ustvariti je potrebno objekt tipa Student z imenom, priimkom in vpisno številko ter objekt tipa Indeks s praznim predmetnikom in praznim seznamom opravljanja izpitov),
- dodajanje novega predmeta v predmetnik izbranega študenta (dodajanje je možno le v primeru, če študent tega predmeta še nima v predmetniku),
- dodajanje opravljanja izpita študenta (opravljanje izpita se doda le v primeru, če ima študent ta predmet v predmetniku; hkrati se poskrbi tudi za prenos podatkov o trenutno veljavnih ocenah in datumu opravljenega izpita v ustrezen predmet iz predmetnika),
- iskanje študenta po vpisni številki in izpis vseh njegovih ocen in polaganj,
- izpis podatkov vseh shranjenih študentov (vpisna številka, ime in priimek),
- izpis povprečne ocene pri izbranem predmetu (pregleda se vsa opravljanja izpitov predmeta pri vseh študentih, ne glede na dobljeno pozitivno oz. negativno oceno).

A*. Rekurzivno risanje krivulj Sierpinskega

Napišite program v Javi, ki, podobno kot četrta naloga, na rekurziven način izriše krivulje Sierpinskega i-tega reda. Krivulji prvega in drugega reda sta prikazani na spodnji sliki.



S_1 S_2

B*. Simulacija sistema strežnik-odjemalec s pomočjo vrste FIFO

Napišite program v Javi, ki s pomočjo vrste prikazuje simulacijo klasičnega sistema strežnik-odjemalec. Strežnike lahko predstavimo kot mrežne tiskalnike, ki sprejemajo zahtevke za tiskanje. Zahtevki (odjemalci) čakajo na streženje v čakalni vrsti FIFO (First In First Out).

Mrežnih tiskalnikov je lahko več (označimo jih s črkami A,B,C,D...), vsak pa ima natančno predpisano hitrost tiskanja papirja, ki naj bo med 0,5 in 1 stran na sekundo (parameter določite z generatorjem naključnih števil ob ustvarjanju objekta-tiskalnika).

Zahtevki za tiskanje se pojavljajo na vsakih 1 do 6 sekund in se takoj postavijo v vrsto. Število strani, ki morajo biti natisnjene v okviru posameznega zahtevka, naj bo med 5 in 20. V primeru, da je poljuben tiskalnik prost, se iz vrste vzame ustrezen (prvi v vrsti) zahtevek in se preda tiskalniku v izvajanje. Čas tiskanja je odvisno od števila strani zahtevka in hitrosti tiskanja tiskalnika. Program naj izpiše stanje v diskretnem času, ki je omejen na eno sekundo natančno. Izpis naj bo takšen, kot to prikazuje spodnji primer. Celoten čas izvajanja simulacije določimo ob zagonu programa.

Če med izvajanjem simulacije število zahtevkov v vrsti doseže neko vnaprej določeno zgornjo mejo, se do konca simulacije zahtevki ne ustvarjajo več.

Za FIFO vrsto uporabite generično vrsto, ki ste jo izdelali v okviru 5 domače naloge (LinkedQueueG).

Primer izvajanja simulacije prikazuje naslednji izpis:

```
Stevilo tiskalnikov: 3

Hitrost tiskanja tiskalnikov (strani/sekundo):

    A – 0.51
```

B – 0.64

C – 0.81

Cas simulacije: 22 s

Max. število zahtevkov v vrsti: 5

cas: 0

cas: 1

cas: 2

cas: 3

cas: 4

cas: 5

Zahtevek #1 prispe ob casu 5, s stevilom strani: 14

V vrsti se nahajajo zahtevki: [#1(14)]

Tiskalnik A(0.51) prevzame zahtevek #1 ob casu 5

Vrsta je prazna

cas: 6

A(0.51) tiskanje koncano cez 27 sekund

cas: 7

A(0.51) tiskanje koncano cez 26 sekund

cas: 8

Zahtevek #2 prispe ob casu 8, s stevilom strani: 7

V vrsti se nahajajo zahtevki: [#2(7)]

A(0.51) tiskanje koncano cez 25 sekund

Tiskalnik B(0.64) prevzame zahtevek #2 ob casu 8

Vrsta je prazna

cas: 9

A(0.51) tiskanje koncano cez 24 sekund

B(0.64) tiskanje koncano cez 10 sekund

cas: 10

A(0.51) tiskanje koncano cez 23 sekund

B(0.64) tiskanje koncano cez 9 sekund

cas: 11

Zahtevek #3 prispe ob casu 11, s stevilom strani: 14

V vrsti se nahajajo zahtevki: [#3(14)]

A(0.51) tiskanje koncano cez 22 sekund

B(0.64) tiskanje koncano cez 8 sekund

Tiskalnik C(0.81) prevzame zahtevek #3 ob casu 11

Vrsta je prazna

cas: 12

A(0.51) tiskanje koncano cez 21 sekund

B(0.64) tiskanje koncano cez 7 sekund

C(0.81) tiskanje koncano cez 17 sekund

cas: 13

Zahtevek #4 prispe ob casu 13, s stevilom strani: 9

V vrsti se nahajajo zahtevki: [#4(9)]

A(0.51) tiskanje koncano cez 20 sekund

B(0.64) tiskanje koncano cez 6 sekund

C(0.81) tiskanje koncano cez 16 sekund

cas: 14

A(0.51) tiskanje koncano cez 19 sekund

B(0.64) tiskanje koncano cez 5 sekund

C(0.81) tiskanje koncano cez 15 sekund

cas: 15

A(0.51) tiskanje koncano cez 18 sekund

B(0.64) tiskanje koncano cez 4 sekund

C(0.81) tiskanje koncano cez 14 sekund

cas: 16

A(0.51) tiskanje koncano cez 17 sekund

B(0.64) tiskanje koncano cez 3 sekund

C(0.81) tiskanje koncano cez 13 sekund

cas: 17

A(0.51) tiskanje koncano cez 16 sekund

B(0.64) tiskanje koncano cez 2 sekund

C(0.81) tiskanje koncano cez 12 sekund

cas: 18

Zahtevek #5 prispe ob casu 18, s stevilom strani: 5

V vrsti se nahajajo zahtevki: [#4(9), #5(5)]

A(0.51) tiskanje koncano cez 15 sekund

B(0.64) tiskanje koncano cez 1 sekund

C(0.81) tiskanje koncano cez 11 sekund

cas: 19

A(0.51) tiskanje koncano cez 14 sekund

Tiskalnik B(0.64)konca z tiskanjem zahtevka #2 ob casu 19

C(0.81) tiskanje koncano cez 10 sekund

cas: 20

A(0.51) tiskanje koncano cez 13 sekund

Tiskalnik B(0.64) prevzame zahtevek #4 ob casu 20

V vrsti se nahajajo zahtevki: [#5(5)]

C(0.81) tiskanje koncano cez 9 sekund

cas: 21

A(0.51) tiskanje koncano cez 12 sekund

B(0.64) tiskanje koncano cez 14 sekund

C(0.81) tiskanje koncano cez 8 sekund

cas: 22

Zahtevek #6 prispe ob casu 22, s stevilom strani: 18

V vrsti se nahajajo zahtevki: [#5(5), #6(18)]

A(0.51) tiskanje koncano cez 11 sekund

B(0.64) tiskanje koncano cez 13 sekund

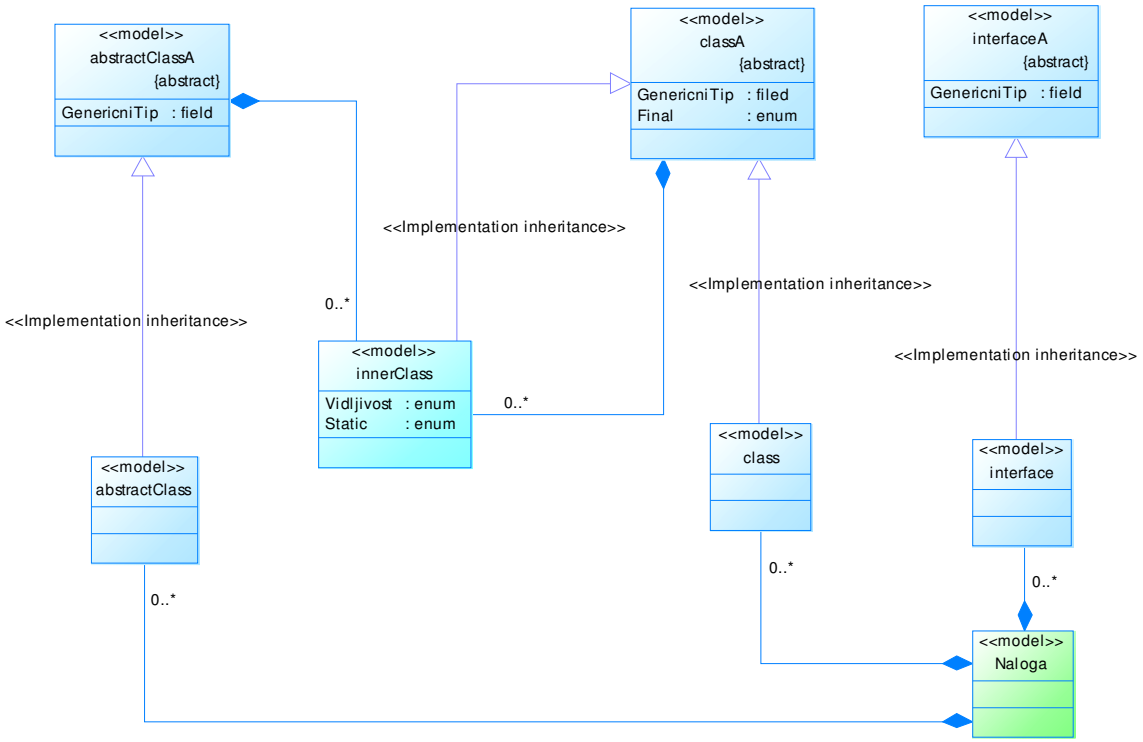
C(0.81) tiskanje koncano cez 7 sekund

NAVODILA:

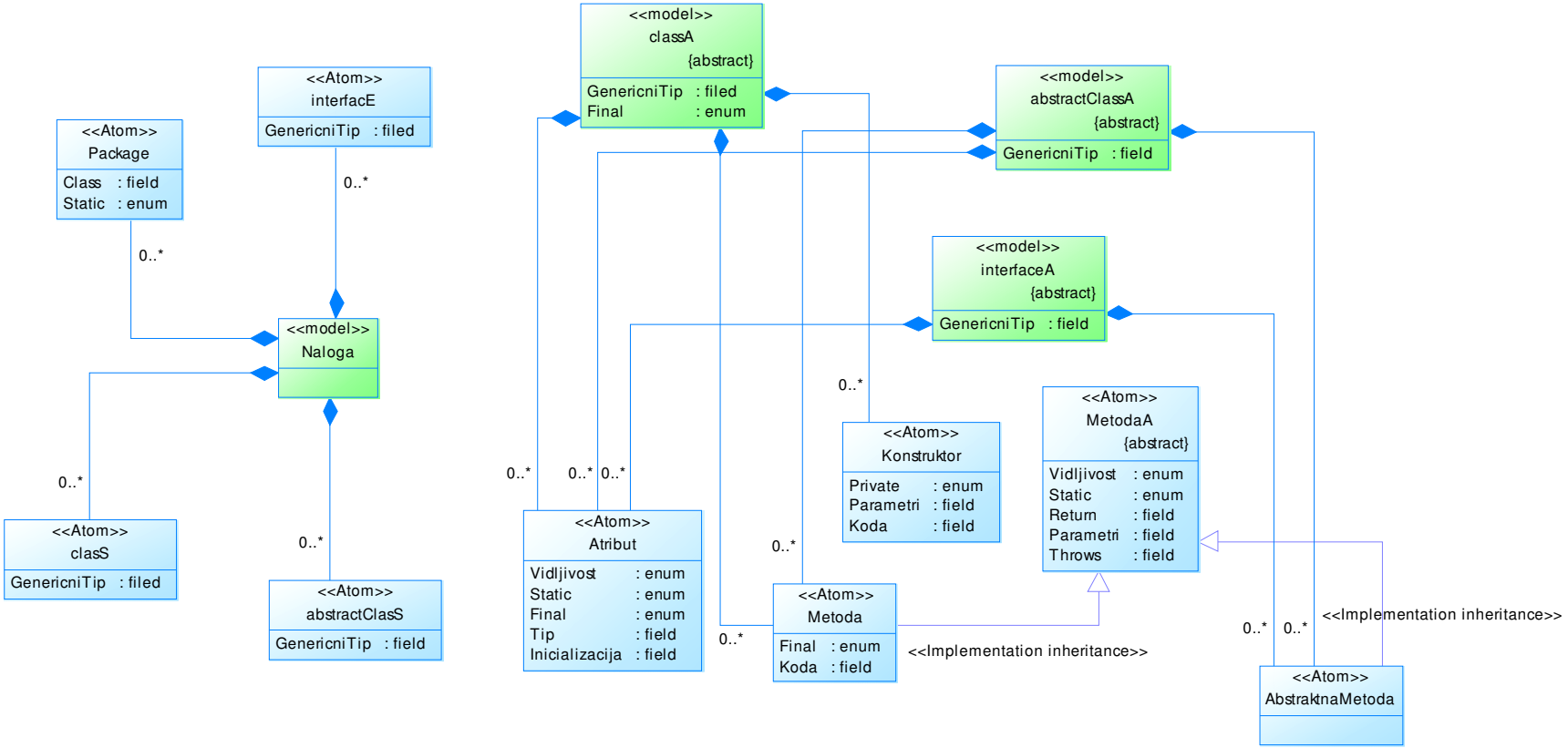
- Naloge lahko delate doma, vendar jih morate zagovarjati na laboratorijskih vajah do predpisanih rokov (ti so navedeni ob vsaki nalogi). V kolikor na določen rok odpadejo (prazniki), morate naloge zagovarjati na naslednjih vajah.
- Striktno se držite razporeda laboratorijskih vaj.
- Študenti, ki iz upravičenih razlogov (na podlagi pisnega dokazila o bolezni ipd.) ne bodo pravočasno zagovarjali naloge, morajo to storiti na prvem naslednjem roku.
- V poletnem izpitnem odbobju bodo lahko opravljali izpit samo študenti, ki bodo pravočasno opravili vseh 6 obveznih nalog.
- Na prvem jesenskem roku bodo lahko opravljali izpit študenti, ki bodo pravočasno opravili vsaj 3 obvezne naloge.
- Ostali študenti bodo lahko opravljali izpit šele na drugem jesenskem roku.
- Nalogi A* in B* sta težji in nista obvezni. Študenti, ki bodo opravili ti nalogi do predpisanega roka, bodo dobili za vsako nalogo (največ) 5 dodatnih točk. Te se bodo upoštevale (samo) pri prvem polaganju pisnega izpita v poletnem izpitnem obdobju.
- Naloge niso obvezne za študente, ki so ponovno vpisani v 1. letnik. Narediti jih morajo le, če želijo pridobiti dodatne točke s pomočjo nalog A* in B* (v tem primeru morajo zagovoriti tudi vse obvezne naloge).
- Študent, ki zamudi rok za zagovor obvezne naloge, lahko namesto te naloge naredi eno izmed dodatnih nalog (tako, ki ji še ni potekel rok za oddajo), vendar se v tem primeru ne upoštevajo dodatne točke

8.3 METAMODEL MODELNO VODENEGA RAZVOJA JAVANSKIH PROGRAMOV

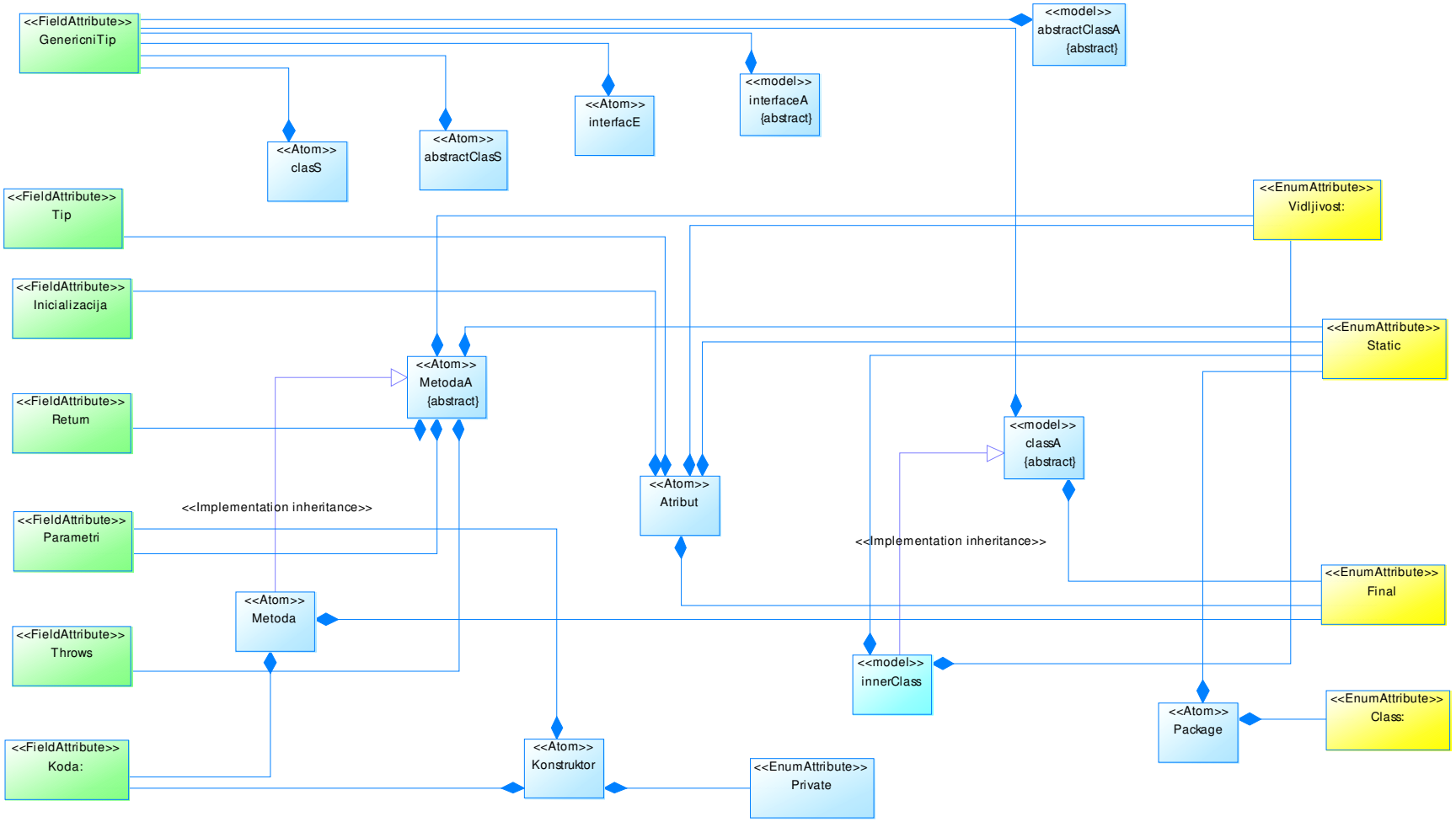
8.3.1 SESTAVI JAVANSKEGA PROGRAMA



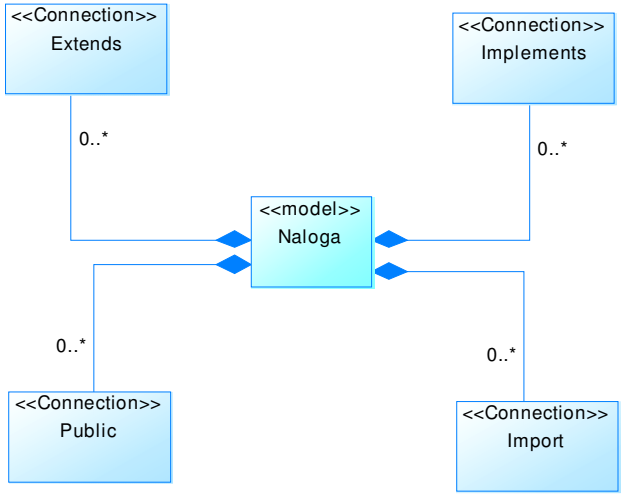
8.3.2 ATOMARNI DELI JAVANSKEGA PROGRAMA



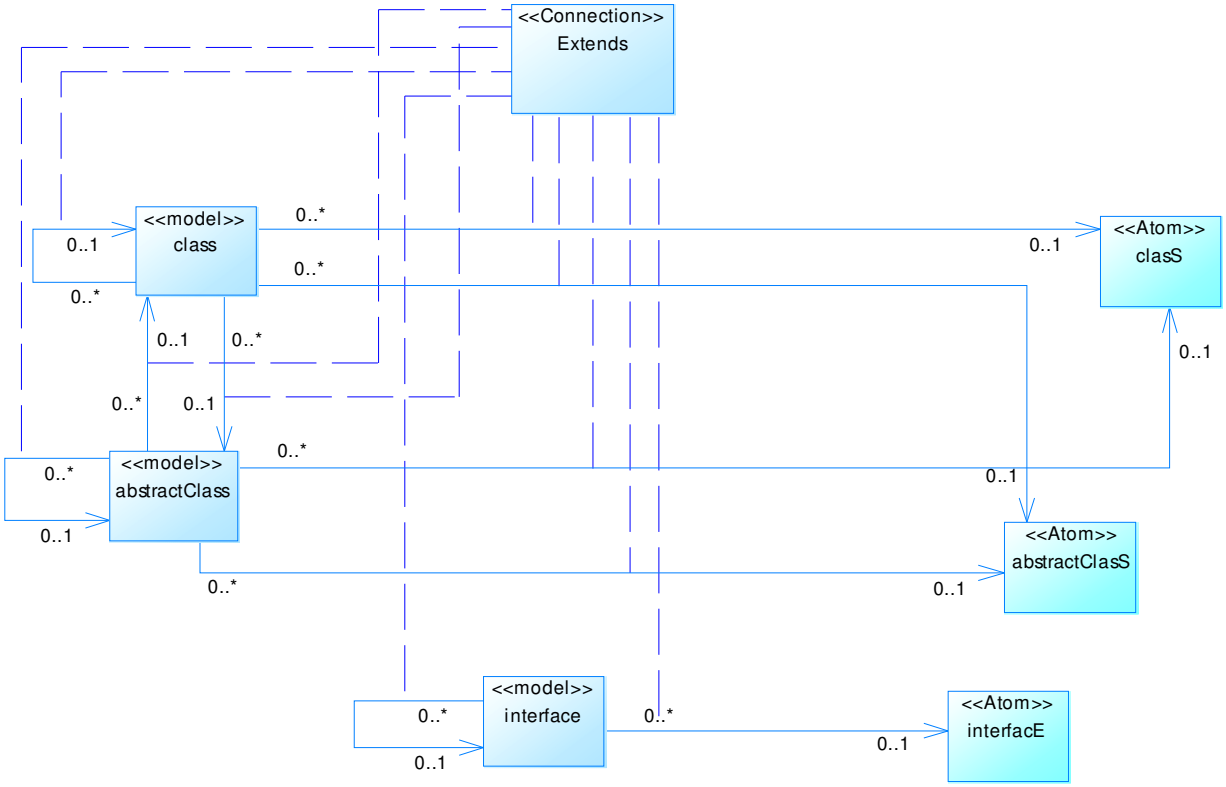
8.3.3 ATTRIBUTI MODELIRNIH GRADNIKOV



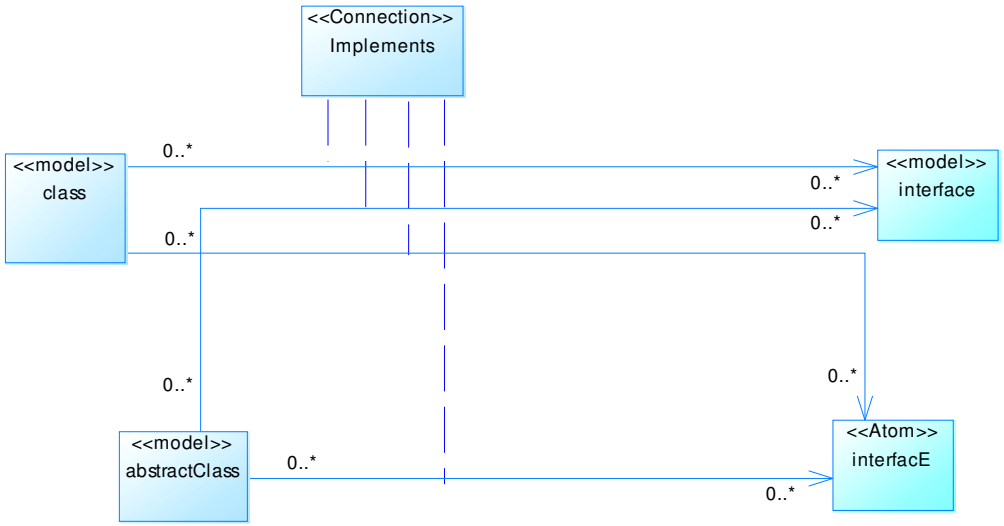
8.3.4 POVEZAVE MED MODELIRNIMI GRADNIKI



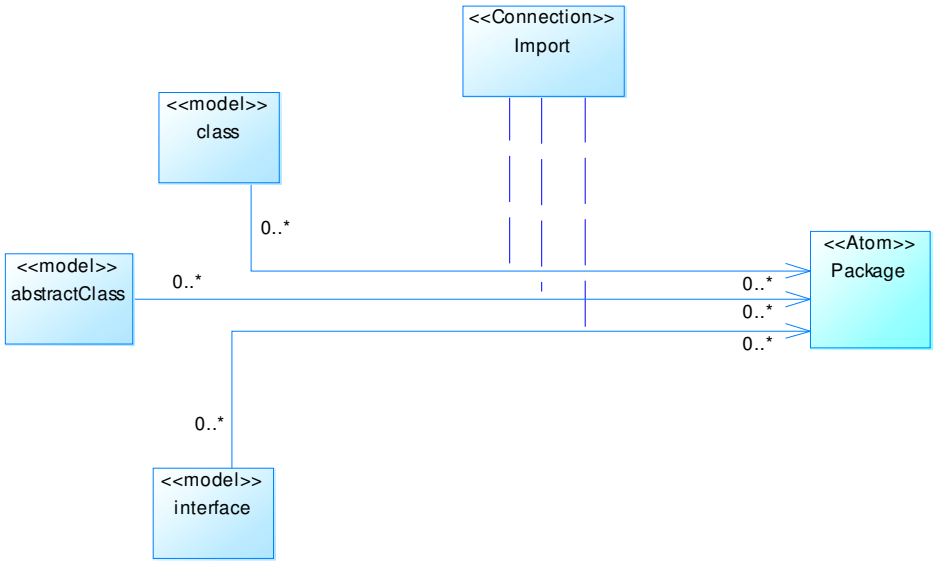
8.3.5 POVEZAVA - EXTENDS



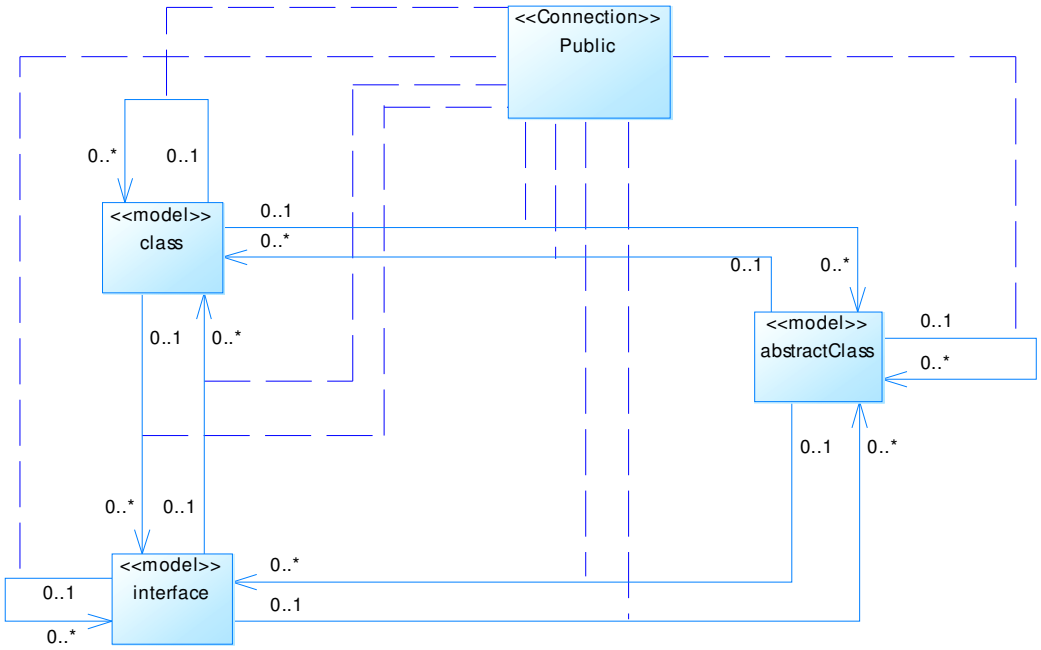
8.3.6 POVEZAVA – IMPLEMENTS



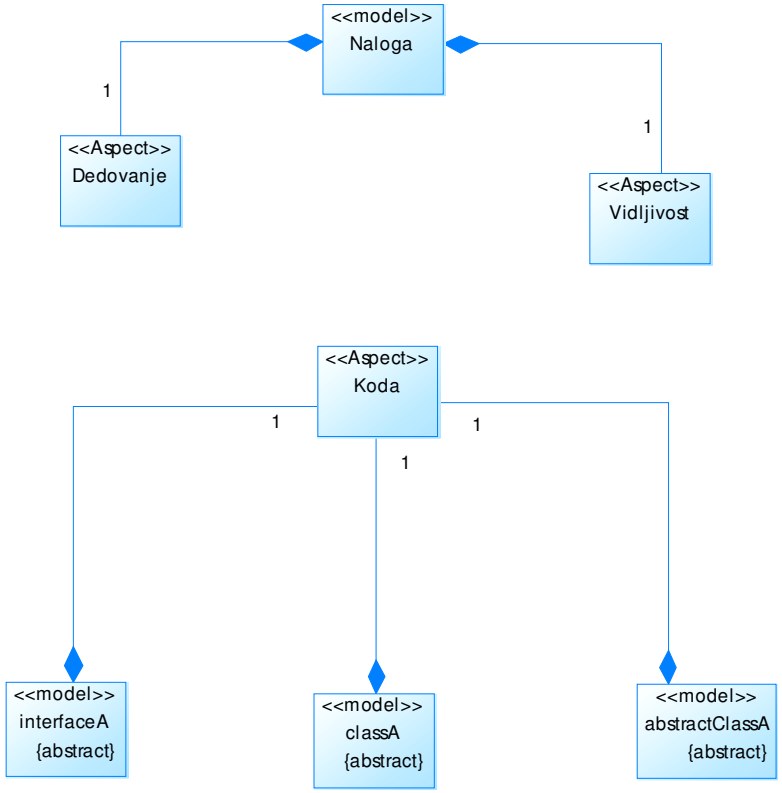
8.3.7 POVEZAVA - IMPORT



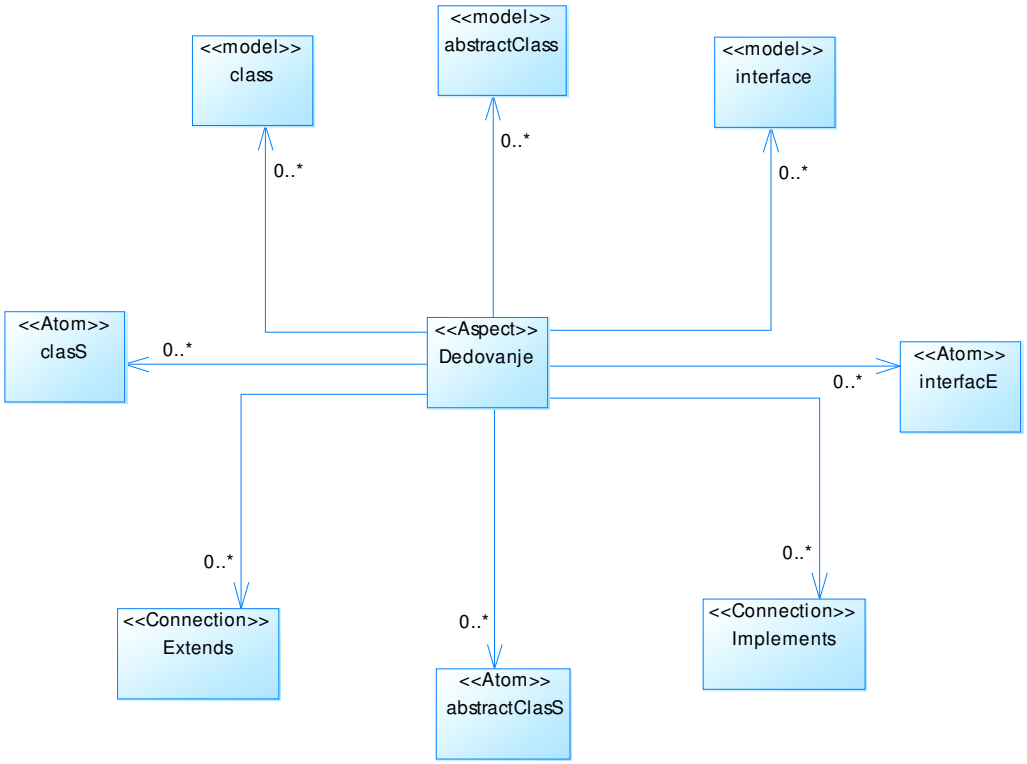
8.3.8 POVEZAVA - PUBLIC



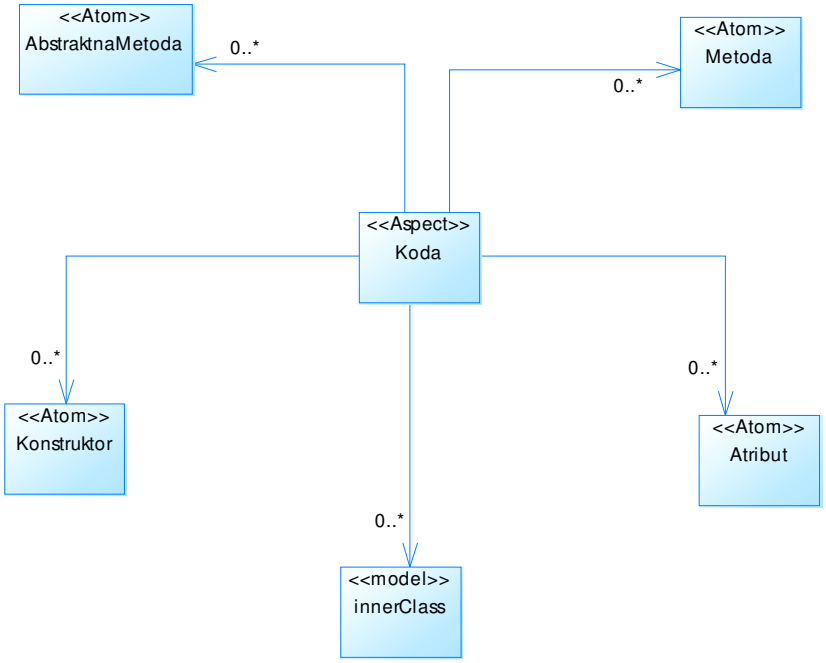
8.3.9 POGLEDI


<<model>>
Naloga
1
1
<<Aspect>>
Dedovanje
<<Aspect>>
Vidljivost
<<Aspect>>
Koda
1
1
1
<<model>>
interfaceA
{abstract}
<<model>>
classA
{abstract}
<<model>>
abstractClassA
{abstract}

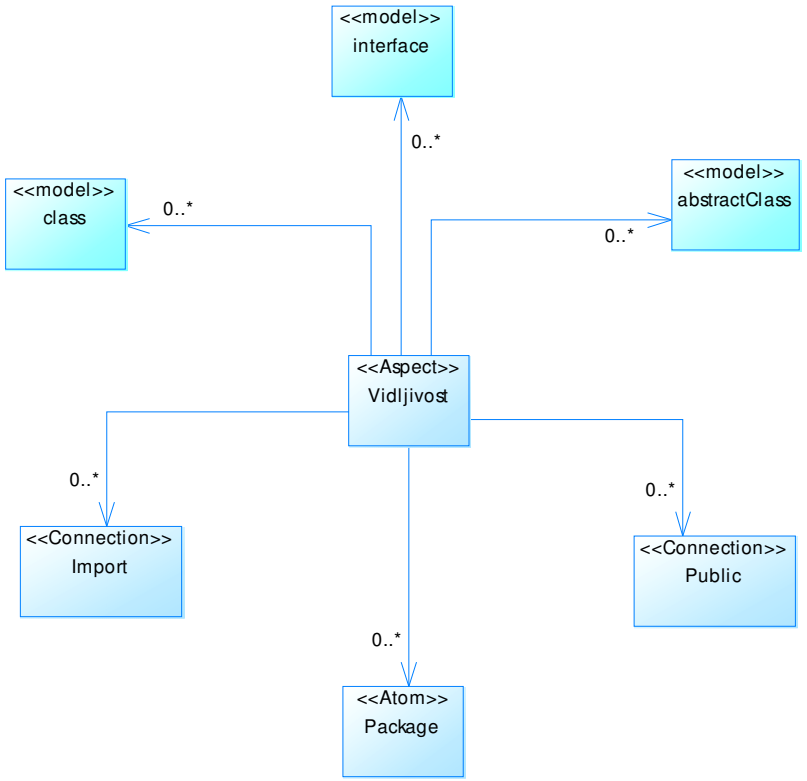
8.3.10 POGLED - DEDOVANJE



8.3.11 POGLED - KODA



8.3.12 POGLED - VIDLJIVOST



8.4 REZULTATI ANKETE ŠTUDENTOV

Vprašanje	Odgovori	Dodatni opisi študentov
I DEL – SPLOŠNO O VAŠEM ZNANJU IZ PROGRAMIRANJA IN MODELNO USMERJENEM RAZVOJU		
1.	2xA, 3xB	• Naučil sem se pred srednjo šolo, ker na gimnaziji programiranja ni bilo v programu.
2.	4xA,1xB	
3.	1xA,2xB,2xC	
4.	2xA,3xB	• Sama uporaba JCP prototipa vzame preveč časa za izdelavo obširnejših nalog (npr. 3. naloga), saj pri večjih projektih zgubiš preveč časa za samo navigacijo med kodo in posameznimi objekti. • NE, Ker smo s projektom začeli na sredini predavanj! • Ta program smo začeli uporabljat šele po prvih dveh domačih nalogah, poleg tega mi je GME nekaj nagajal, tako da so se podatki o projektu izgubili.
5.	3xA,2xB	
6.	2xB,3xC	
7.	4xB,1xC	
8.	5xA	
9.	1xA,3xB,1xC	
10.	2xA,2xB,1xC	
11.	3xA,1xB,1xD	
12.	2xA,3xB	

II DEL – PODROBNO O PROTOTIPU JCP		
13.	5xA	• Mogoče boljši navigator trenutnega nahajanja. • Komentar na odgovor A: na tak način se da zelo dobro videti kaj program vsebuje. Če te pa zanima izvorna koda, se težje znajdeš kot pri dobro strukturirani in obarvani izvorni kodi.
14.	5xA, 3xB, 2xC	• Puščice pri extends povezavah bi usmeril proti trenutnemu razredu (npr Studenti, WorkingPanel), saj se v novem razredu združijo funkcionalnosti interface-ov ter razreda, katerega razširja. • Simbolika prečrtanih imen ni najboljša, ampak to je lepotne narave. • Morda bi za splošna imena posameznih objektov uporabili imena, ki takoj povedo ali je to npr razred, ki ga moramo sami napisati ali že obstoječi razred: namesto »class« bi lahko bil »nov razred«, namesto »clasS« pa »obstoječi razred«. Sam sem bil na začetku malo zmeden, ker mi ni bila takoj jasna razlika.
15.	1xA, 5xB	• Pri veliki količini razredov postane preveč črt (npr. program s 5-imi okni, vsako v svoji datoteki, vsa uporabljajo java.awt). • Jaz bi dal neusmerjene povezave.
16.	4xB, 1xC	
17.	4xA, 1xB	
18.	4xA, 1xB	
19.	4xA, 1xB	
20.	5xA	• IntelliSense. Manjkale so sposobnosti obstoječih programskih orodij.
21.		• Obarvanost kode • Pri pisanju kode bi se lahko videla tudi glava metode. • Pri ustvarjanju razreda bi se lahko konstruktor in/ali main() metoda sama ustvarila. • Lepo bi bilo, če bi se lahko videla koda celotnega razreda naenkrat-kot da bi pisali navadno, in bi bile potem spremembe vidne tudi vizualno (če bi npr. dodajali nove metode). • Ko extendamo/implementiramo razred bi bilo lepo, če bi se metode, ki jih moramo napisati sami, kreirale z telesom ☺. • Ni bilo funkcije za direktni zagon programa. • Precej sem se navadil na uporabo NetBeansov, tako da sem resnično pogrešal intelliSense in obarvanost kode, saj

		sem se pri pisanju le s težavo znašel, če je bila katera od metod malo daljša. Tudi kadar sem moral kaj popraviti, oziroma sem dodajal nove stvari, je bilo ravno zaradi tega to težje kot v netBeans ali pa v Notepad++. Velik problem je bil zame tudi to, da nisem mogel imeti odprtih več metod za pisanje kode hkrati. Če sem se spomnil, da sem nekje kaj pozabil, sem moral trenutno metodo zapreti in jo pozneje še enkrat odpreti. Velik problem se mi zdi tudi to, da v primeru, ko ti prevajalnik javi napako, ni možno pogledati kje natanko je napaka, ne da bi si ogledal izvorno kodo, ki jo je generiral JCP. Ravno zaradi tega sem raje delal v navadni beležnici oziroma v katerem od zgoraj omenjenih programov.
22.		• Poglobitev v pomen objektov, ter temu primerne grafične predstavitve. Menim, da pri tistih, ki so novi v programiranju, pojmi kot so abstract, static, itd. predstavljajo težavo pri razumevanju, pri čemer bi precej pomagala dobra vizualizacija (če bi se že iz izgleda static atributa razbralo, da je ta atribut skupen vsem instancam razreda). • Da ne bi bilo več treba programa zaganjati iz konzole. • Da ti prikaže celotno kodo programa. • Autogeneriranje get/set metod. • Podobne stvari kot jih imajo sedaj drugi IDE-ji (npr.: Eclipse, NetBeans). • Da je kompatibilen za Windowse in ostale platforme. • Da se ne sesuva. • Da je več funkcij dostopnih preko tipkovnice(short-cuti) in manj klikanja. • Najprej bi se mi zdelo smiselno odpraviti trenutne napake, in popraviti stabilnost programa. Smiselno se mi zdi vključiti tudi možnost reverse Engineering-a, saj bi to orodje lahko služilo kot odlična predstavitev delovanja programov začetnikom. Morda bi lahko vključili tudi več bližnjičnih tipk za manjšo odvisnost od miške, ki te precej upočasni. Uporabna bi tudi bila možnost direktnega zagona programov, namesto zaganjanja s konzole.
23.		• Compile se ne izvede na ravni projekta. • Ne, ob »pravilni uporabi « se je samo nekajkrat sesul. • Razen tega, da se je program občasno sesedel (največkrat kadar sem po izpisu napak fokusiral drugi program), nisem opazil nobene večje napake.
24.	5xA	

9. VIRI IN LITERATURA

[1] SCHMIDT C., Douglas: Model-Driven Engineering, Computer, February, 2006

[2] SELIC Bran: The Pragmatics of Model-Driven Development, Software, 2003, vol. 20, ed. 5

[3] Sendall Shane, Kozaczynski Wojtek: Model Transformation: The Heart And Soul of Model-Driven Software Development, Software, 2003, vol. 20, ed. 5, str. 42-45

[4] Bettin Jorn: Model-Driven Software Development: An emerging paradigm for Industrialized Software Asset Development, SoftMetaWare, junij 2004, ver. 0.8

[5] Krishnakumar Balasubramanian, Aniruddha Gokhale in drugi: Developing Applications Using Model-Driven Design Environments, Computer, februar, 2006, str. 33- 40

[6] Mellor J. Stephen, Clark N. Anthony, Futagami Takao: Model-Driven Development, IEEE Software, vol. 20, ed. 5, 2003

[7] Stahl Thomas, Völer Markus in drugi: Model-Driven Software Development, John Wiley & Sons, 2006

[8] Völter Markus, Bettin Jorn: Patterns for Model-Driven Software-Development, ingenierbüro für softwaretechnologie Heidenheim, Germany, SoftMEtaWare Auckland, New Zeland, ver. 14, Maj 2004

[9] Childs Adam, Greenwald Jesse , Jung Georg in drugi: CALM and Cadena: Metamodeling for Component-Based Product-Line Development, Computer, 2006, vol. 39, ed. 2, str. 42-50

[10] Akos Ledeczi, Maroti Miklos, Bakay Arpad in drugi: The Generic Modeling Environment, IEEE International Workshop on Intelligent Signal Processing, 2001, Budimpešta, Madžarska

[11] Akos Ledeczi, Balogh Gyorgy, Molnar Zoltan in drugi: Model Integrated Computing in the Large, IEEE Aerospace, CD Rom, Big Sky, MT, Marec 2005.

OSTALI VIRI

[12] The Object Management Group, 29.08.2009: http://www.omg.org/

[13] UML® Resource Page, 15.7.2009: http://uml-directory.omg.org/

[14] BRUNELLI, Mark: The Holy Grail of model-driven development, SearchWebServices.com, 10.8.2004: http://searchwebservices.techtarget.com/qna/0,289202,sid26_gci999474,00.html

[15] Define a platform please, BLOG, 24.04.2007: http://blogs.msdn.com/jim_glass/archive/2007/04/24/define-a-platform-please.aspx

[16] Catalog of UML Profile Specifications, 06.09.2009: http://www.omg.org/technology/documents/profile_catalog.htm

[17] Formal specification to UML transformation demo, 30.08.2009: http://nt-appn.comp.nus.edu.sg/fm/zml/

[18] The MDD view of the Language Landscape, 11.12.2007: http://martinfowler.com/dslwip/MDDview.html

[19] MDA tools, 13.09.2009: http://www.modelbased.net/mda_tools.html

[20] Model-Driven Software Development, 08.11.2008: http://www.mdsd.info

[21] Cadena, 14.09.2009: http://projects.cis.ksu.edu/gf/project/cadena/

[22] C-SAW project, 01.09.2009: http://www.gray-area.org/Research/C-SAW/

[23] Domain-Specific Development with Visual Studio DSL Tools,

05.09.2009:
http://www.domainspecificdevelopment.com/

[24] Exreme Programming: A gentle introduction.,
14.09.2009:
http:// www.extremeprogramming.org/

[25] Sucess Stories,
14.09.2009:
http://www.uml.org/uml_success_stories/index.htm

[26] GME: Generic Modeling Environment,
20.09.2009:
http://www.isis.vanderbilt.edu/projects/gme/

[27] Object Constraint Language Specification, Version 2.0
12.10.2009:
http://www.omg.org/technology/documents/formal/ocl.htm

[28] GME 6, User's manual, version 6