Elektrotehniški vestnik 73(5): 255–260, 2006
Electrotechnical Review, Ljubljana, Slovenija

Arhitektura za modelno voden razvoj in njen vpliv na proces razvoja programske opreme

Damjan Vavpotič, Marjan Krisper

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, Tržaška 25, 1000 Ljubljana, Slovenija
E-pošta: damjan.vavpotič@fri.uni-lj.si

Povzetek. Ideja modelno vodenega razvoja programske opreme ni nova in je na posameznih področjih uveljavljena že vrsto let (npr. podatkovno modeliranje). Kljub temu so bili v preteklosti poskusi idejo razširiti tudi na druga področja razvoja programske opreme večinoma neuspešni in se v praksi niso uveljavili (npr. CASE). Na podlagi pridobljenih izkušenj je bila pod okriljem Object management group (OMG) oblikovana celovita arhitektura za modelno voden razvoj programske opreme, t.i. model driven architecture (MDA). MDA podaja teoretično osnovo za dvig ravni abstrakcije, na katerem nastaja programska oprema, posledica tega pa so tudi pomembne spremembe v procesu razvoja programske opreme: močno se poveča pomen analize in načrtovanja, hkrati se zmanjša pomen programiranja in testiranja. V prispevku so najprej na kratko predstavljeni osnovni koncepti MDA, v nadaljevanju pa se posvetimo predvsem vplivu modelno vodenega razvoja in MDA na proces razvoja programske opreme in še posebej primerjavi modelno vodenega pristopa k razvoju programske opreme z agilnimi pristopi.

Ključne besede: metodologije razvoja programske opreme, MDA, modelno voden razvoj, proces razvoja programske opreme

Model-driven architecture and its impact on the software development process

Extended abstract. The idea of model-driven software development is not new. It has been practiced in certain development fields for several years (e.g. data modeling). The trials to spread the idea to other fields of software development have been quite unsuccessful and never widely used in practice (e.g. CASE). Based on the gained experience, the architecture for model-driven development of software systems, i.e. model-driven architecture (MDA), has been developed under the umbrella of the Object management group (OMG). MDA offers a theoretical basis that enables the development of software on a higher abstraction level. Its use also affects the software development process; on one hand, analysis and design become the most important parts of the development and on the other, implementation and testing turn to be less important. The first section of the paper briefly presents the basic concepts of MDA. The second section describes the impact of model-driven development and MDA on a software development process and especially focuses on a comparison between model driven development and agile processes.

Key words: software development methodologies, MDA, model-driven development, software development process

Prejet 19. oktober, 2005
Odobren 30. november, 2006

1 Uvod

Pri razvoju IS se pogosto srečamo z vprašanjem, katere tehnologije uporabiti, da bo aplikacija razvita v čim krajšem času, s čim nižjimi stroški ter pozneje razširljiva. Pri izbiri smo navadno omejeni na tehnologije, ki jih razvijalci obvladajo. S hitrim razvojem tehnologij postaja problem izbire še bolj pereč. Razvijalci morajo namreč poznati veliko tehnologij, ki se pogosto nadgrajujejo in menjavajo. Prehod na drugo tehnologijo navadno pomeni, da je treba aplikacijo razviti na novo ali pa so potrebne vsaj obsežne prilagoditve. V tem pogledu bi bistveno prednost pomenil razvoj aplikacij na višji ravni abstrakcije, ki ni odvisen od uporabljenih implementacijskih tehnologij.

Ideja o dvigu ravni abstrakcije, na katerem nastajajo IS, pravzaprav ni nova in se je na posameznih področjih tudi precej uveljavila. Tako na primer že vrsto let poznamo orodja, ki omogočajo generiranje kode in skript za PB na podlagi konceptualnih podatkovnih modelov. Dobro poznana so tudi orodja CASE, ki so poskušala dvigniti raven abstrakcije, na katerem nastajajajo IS, vendar se v parksi niso uvel-

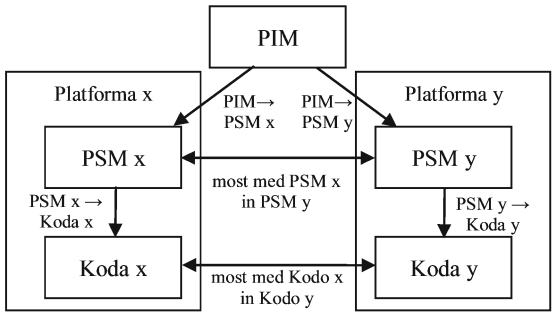
javila [1]. S standardizacijo jezika za vizualno modeliranje programske opreme (UML) in razvojem čedalje zmogljivejših orodij za generiranje kode je ideja v zadnjem času dobila nov zagon. Bistven napredek na tem področju je standardizacija arhitekture za razvoj programske opreme na podlagi modelov (MDA), ki je nastala v okviru OMG.

2 MDA - arhitektura za modelno voden razvoj

Temeljno vodilo modelno vodenega razvoja programske opreme ali Model Driven Development (MDD), kot ga predvideva standard MDA [2], je, da se razvijalci posvetijo predvsem funkcionalnosti sistema oziroma določijo njegove bistvene funkcije, namesto da v programskem jeziku opisujejo tehnološke podrobnosti njegovega delovanja. Razvijalci funkcionalnosti opišejo na visoki abstraktni ravni, v veliki meri z uporabo vizualnih modelov sistema. Osnovna zahteva za te visoko abstraktne modele je, da so splošni, kar pomeni, da so neodvisni od uporabljene platforme in tehnologije, ali z drugimi besedami, da opisujejo izključno funkcionalne zahteve. Prav tako MDA predvideva avtomatizirano transformacijo visoko abstraktnih modelov v modele na nižji ravni abstrakcije, ki so prilagojeni posameznim platformam in tehnologijam, ter nato avtomatizirano transformacijo teh modelov v delujočo kodo. MDA definira tri osnovne modele [3]:

- *PIM (platform independent model)* je model na visoki abstraktni ravni, na podlagi katerega so generirani nižjenivojski modeli in koda. PIM mora biti konsistenten, točen in mora vsebovati dovolj informacij o sistemu, da omogoča nadaljnje generiranje nižjenivojskih modelov PSM.
- *PSM (platform specific model)* je model na ravni, ki upošteva konstrukte implementacije, ki so na voljo v okviru izbrane implementacijske tehnologije oziroma platforme. S pomočjo pretvornika PIM transformiramo v enega ali več PSM. Število PSM, ki nastanejo na podlagi PIM, je odvisno od števila različnih implementacijskih tehnologij, ki jih uporablja sistem.
- *Programska koda* je model na najnižji ravni, ki jo je mogoče s pomočjo prevajalnika pretvoriti v izvedljivo kodo oziroma jo interpretirati. Gre za končni rezultat transformacij, ki nastane na podlagi PSM.

Slika 1 poleg opisanih modelov prikazuje tudi druge bistvene elemente MDA. Usmerjene puščice označujejo transformacije, ki omogočajo prehode med modeli na različnih ravneh abstrakcije. Na najvišji ravni je PIM, na podlagi katerega s pomočjo transformacij generiramo modela PSM x in PSM y, ki sta prilagojena Platformi x in Platformi y. V nadaljevanju za PSM x generiramo Kodo x, ki je prilagojena Platformi x, za PSM y pa generiramo Kodo y, ki je prilagojena za Platformo y. To pa še ni dovolj, saj je za delovanje sistema treba povezati tudi kode oziroma PSM na različnih platformah. To dosežemo s pomočjo mostu med PSM x in PSM y, ki ga prav tako generiramo na podlagi PIM. Most med PSM omogoča tudi generiranje fizičnega mostu med Kodo x in Kodo y. Na sliki je prikazano generiranje kode in PSM za dve platformi. Na enak način bi lahko potekalo tudi generiranje kode za več platform.



Slika 1. Različni nivoji abstrakcije v MDA in transformacije
Figure 1. Different levels of abstraction in MDA and transformations

Preden lahko MDA uporabimo, je treba izbrati jezike, v katerih bodo zapisani posamezni modeli. MDA ne predpisuje konkretnih jezikov, pomembno je predvsem, da gre za dobro definirane jezike, ki so primerni za avtomatizirane transformacije. Za delovanje MDA je bistvenega pomena jezik za zapis PIM. PIM mora biti namreč zapisan na visoki abstraktni ravni, ki ni odvisna od tehnologij, hkrati pa mora vsebovati dovolj podrobnosti, da omogoča nadaljnje generiranje PSM in kode. Ena izmed možnosti za zapis PIM, ki jo predvideva OMG, je uporaba UML in Object Constraint Language (OCL). V ta namen je OMG razvila novi različici UML 2.0 in OCL 2.0, ki vključujeta konstrukte za podporo MDA.

3 VPLIV MDA NA PROCES RAZVOJA PROGRAMSKE OPREME

3.1 Primerjava tradicionalnega razvoja in MDD

Kot smo že omenili, se proces razvoja MDD v posameznih aktivnostih precej razlikuje od danes na-

jpogostejšega načina razvoja IS. V primerjavi z danes običajnim razvojem, kjer je navadno velik del razvojnega časa namenjenega kodiranju, je v MDD končna izvorna koda generirana s pomočjo orodij. Večji del razvojnega časa v MDD je namenjen analizi, v okviru katere nastane PIM, ki mora biti točen, konsistenten ter dovolj podroben. Vzporedno z analizo funkcionalnih zahtev lahko poteka analiza nefunkcionalnih zahtev, v okviru katere izberemo najustreznejše platforme, arhitekture, vzorce oziroma implementacijske tehnologije, ki jih bo uporabljal sistem. Na podlagi PIM in analize nefunkcionalnih zahtev generiramo PSM, v naslednjem koraku pa še kodo. Slika 2 prikazuje primerjavo izdelkov in aktivnosti, ki nastopajo pri klasičnem razvoju in pri MDD.

V MDD sta torej bistveni izdelava PIM in izbira ustreznih implementacijskih tehnologij, precej pa se zmanjša pomen klasičnega kodiranja. Tako spremenjen proces tudi pomembno vpliva na znanja in vloge, potrebne pri razvoju IS. Poveča se pomen analitičnih vlog in hkrati zmanjša pomen vlog, ki nastopajo pri implementaciji. Poglejmo si nekaj novih vlog in kakšna znanja so potrebna:

- *Analitik in načrtovalec* PIM na podlagi izdelkov zajema zahtev in izdelkov poslovnega modeliranja izdelata model PIM. Potrebna znanja obsegajo poznavanje problemske domene, branje izdelkov zajema zahtev - funkcionalne zahteve, določanje logične zgradbe sistema (npr. v jeziku UML), podrobno opredeljevanje poslovnih pravil v PIM (npr. v jeziku OCL) itd. Ker je model PIM neodvisen od platforme, analitiku in načrtovalcu PIM ni treba poznati posebnosti ciljne platforme.

- *Izdelovalec PSM* s pomočjo orodja za transformiranje PIM pretvori v enega ali več modelov PSM. Pred transformacijo mora izbrati ciljne platforme, arhitekture, vzorce, tehnologije itd., ki jih mora dobro poznati. Odločiti se mora na primer, ali izbrati .NET ali J2EE kot ciljno platformo, ali je potrebna 3-nivojska arhitektura ali zadošča arhitektura odjemalec-strežnik, itd. Osnova za njegove odločitve so predvsem nefunkcionalne zahteve, identificirane v aktivnosti zajema zahtev. Izdelovalec PSM je torej v veliki meri odgovoren, da bo izdelani sistem ustrezal nefunkcionalnim zahtevam, kot so zmogljivost, varnost ipd.

- *Razvijalec definicij transformacij* je vloga, ki skrbi za izdelavo definicij transformacij za transformacijska orodja. Razvijalec definicij transformacij je zelo zahtevna vloga, saj zahteva dobro poznavanje jezikov za zapis PIM in PSM, transformacijskih jezikov in transformacijskih orodij, različnih platform in tehnologij. Večino definicij transformacij bo v prihodnosti najverjetneje mogoče kupiti od izdelovalcev transformacijskih orodij. Kljub temu se bo verjetno pokazala potreba po prilagajanju transformacij in razvoju povsem specifičnih transformacij, ki na trgu ne bodo dostopne.

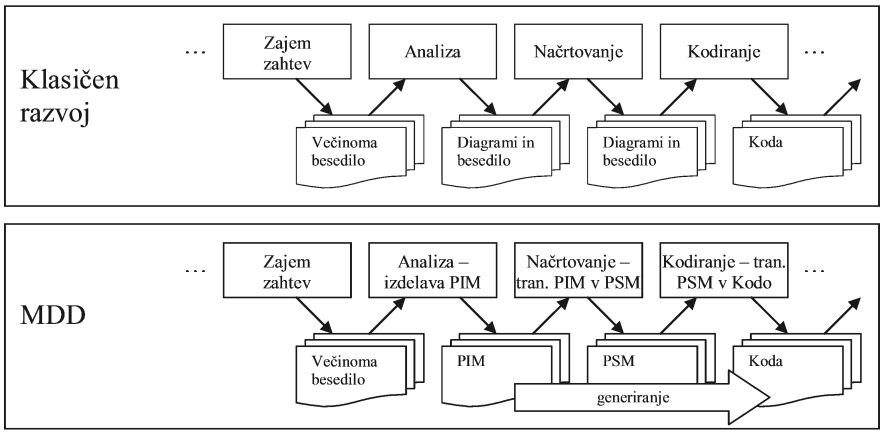
Idealno bi bilo, da bi bila koda v celoti generirana, ročno kodiranje ali ročno spreminjanje PSM pa bi povsem odpadlo. Današnja orodja to žal omogočajo le v omejenem obsegu. Tako je treba pri bolj kompleksnih delih aplikacij ročno dodelati generirano kodo oziroma PSM. Poleg tega v razvojnih orodjih, ki le delno podpirajo MDA, PIM poleg abstraktnih elementov pogosto vsebuje tudi konkretne, platformsko odvisne elemente (npr. metode, zapisane v navadnem programskem jeziku). To je tudi posledica dejstva, da trenutno še nimamo zares primernega jezika za zapis PIM, čeprav se je položaj na tem področju s prihodom UML 2.0 in OCL 2.0 precej izboljšal.

Šele s pomočjo dobro definiranih jezikov in transformacij ter učinkovitih orodij postane ideja MDD zares uporabna tudi v praksi. Čeprav trenutno ni mogoče najti orodja, ki bi v celoti podpiralo MDA, pa je na voljo kar nekaj orodij, ki so usklajena vsaj z delom MDA. To so na primer orodja ArcStyler, OptimalJ, Rational Rapid Developer itd. Idejo MDA so vsaj delno sprejeli tudi nekateri znani izdelovalci razvojnih orodij, kot npr. Borland, ki v svojih novejših orodjih podpira MDA s pomočjo ogrodja Enterprise Core Objects.

3.2 Primerjava agilnih metodologij in MDD

Ugotovimo lahko, da se MDD in danes zelo popularne agilne oziroma lahke metodologije v marsičem razlikujejo. Agilne metodologije so v nasprotju s kompleksnimi in obsežnimi metodologijami precej racionalne pri formalizaciji postopkov analize in načrtovanja ter procesa razvoja nasploh [4]. Kot trdijo zagovorniki agilnih metodologij, so proces razvoja, orodja, projektna dokumentacija, pogodbe med izvajalci in naročniki ter projektni načrti vsekakor pomembni vidiki, ki jih metodologija mora obravnavati, vendar pa je tedaj, ko je treba stvari postaviti na tehtnico, pomembnejše poskrbeti za zadovoljstvo in motivacijo posameznikov, učinkovito komunikacijo med njimi, delujočo programsko opremo, vključevanje uporabnika v proces razvoja in upoštevanje sprememb.

Tabela 1 prikazuje bistvene razlike med MDD in agilnimi metodologijami. Vidimo, da na eni

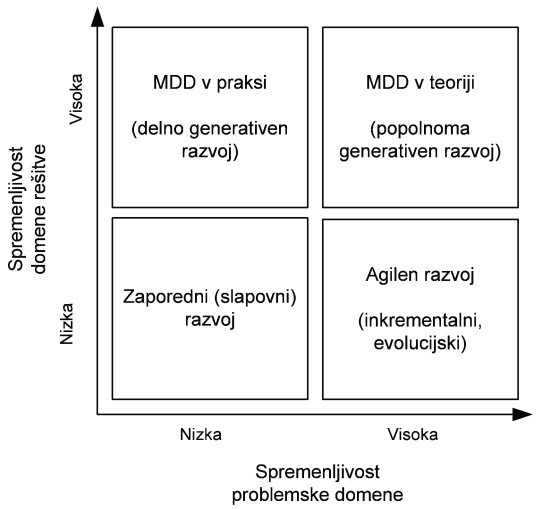


Slika 2. Primerjava izdelkov in aktivnosti pri tradicionalnem razvoju in modelno vodenem razvoju
Figure 2. Comparison of artefacts and activities used in the traditional and model-driven development

strani agilne metodologije dajejo velik poudarek ljudem, ki so udeleženi v procesu razvoja programske opreme, na drugi strani pa se MDD opira predvsem na napredno tehnologijo, ki omogoča generiranje.

Posledica predstavljenih razlik (tabela 1) je med drugim tudi različna primernost obeh pristopov za razvoj programske opreme (slika 3) [5]. Kot smo že omenili, MDD temelji na rigorozni analizi, ki je podlaga za generiranje kode. Naknadno spreminjanje uporabniških zahtev oziroma problemske domene (horizontalna os na sliki), se mora odražati v analizi (PIM), načrt (PSM) in kodo pa je treba na njeni podlagi ponovno generirati. Kot vemo, trenutno nimamo orodij, ki bi znala v popolnosti generirati PSM in kodo, zato jih je treba delno še vedno izdelati ročno. Pri ponovnem generiranju oziroma spremembi problemske domene je treba ročno dopolnjene načrte (PSM) in ročno izdelano kodo tudi ročno spremeniti ali v skrajnem primeru celo izdelati na novo. Posledica tega je, da lahko MDD v praksi obravnavamo kot pristop, ki ga je mogoče uporabiti le pri stabilni problemski domeni. Vendar pa to ne velja za MDD v teoriji, ki naj bi podpiral popolno generiranje PSM in kode.

Prednost MDD pa je njegova prilagodljivost domeni rešitve (vertikalna os na sliki). Gre za prilagodljivost platformam, tehnologijam, jezikom ipd., ki jih uporablja programska rešitev. Ker je analiza, ki je osrednji izdelek v okviru MDD, neodvisna od izbranih tehnologij, je mogoče le-te med seboj menjavati oziroma nadomeščati z alternativnimi tehnologijami ter nato prilagojene PSM in kodo generirati. Pri tem orodja za generiranje PSM in kode ponovno igrajo pomembno vlogo, vseeno pa je MDD v primerjavi z drugimi pristopi, bolj odporen na spremenljivost domene rešitve, saj je tudi pri nepopolnih orodjih osnovni izdelek še vedno analiza.



Slika 3. Primerjava primernosti MDD in agilnih pristopov za razvoj programske opreme
Figure 3. Comparison of MDD and agile approaches to software development

Kot smo že omenili, lahko med MDD v teoriji in agilnimi metodologijami najdemo tudi stične točke. Čeprav se morda zdi, da sodi MDD po svoji naravi v skupino pristopov, ki dajejo večjo težo analizi in načrtovanju (frontloaded methodologies), pa to ne drži v celoti. Tako MDD v teoriji podpira tudi nekatera načela agilnih metodologij.

- *Delujoča programska oprema je pomembnejša od popolne dokumentacije.* Klasične metodologije razvoja zahtevajo redno vzdrževanje izdelkov vseh ravni abstrakcije (na primer za vzdrževanje podatkovne baze moramo vzdrževati konceptualni podatkovni model, relacijski podatkovni model in fizično podatkovno bazo). V praksi pa se pogosto dogaja, da se izdelki na višjih

	Agilne metodologije	MDD
Ljudje	Ljudje imajo najvišjo prioriteto. So osrednji in najpomembnejši dejavnik pri razvoju programske opreme. Zelo pomemben je sociološki vidik.	Definirane so nekatere nove vloge. Ljudje se obravnavajo predvsem s tehnološkega vidika oziroma kot izvajalci/uporabniki procesa. Sociološki vidik ni pomemben.
Proces razvoja programske opreme	Proces ni podrobno definiran. Poudarek je na aktivnostih kodiranja in testiranja. Navadno se uporablja inkrementalen in iterativen življenjski cikel.	Nekatere tradicionalne aktivnosti in postopke procesa izvedejo orodja za generiranje (npr. kodiranje in podrobno načrtovanje). Poudarek je na aktivnostih analize. Inkrementi in iteracije imajo zaradi generiranja kode manjši pomen; življenjski cikel je bliže zaporedni-slapovni obliki.
Tehnologija in orodja	Tehnologija ima najnižjo prioriteto. Zaželeno je, da so orodja in tehnologija čim bolj preprosta.	Tehnologija in orodja imajo najvišjo prioriteto, saj omogočajo generiranje kode, kar je bistvo MDD.

Tabela 1. Osnovni podatki o domenah

Table 2. Basic description of data sets

ravneh abstrakcije v poznejših fazah zanemarijo in se ne vzdržujejo več. Agilni pristopi to upoštevajo in priporočajo vzdrževanje le izbranih izdelkov [4]. MDD temu sledi tako, da zahteva vzdrževanje na najvišji ravni abstrakcije, za vse druge ravni pa uporablja generator. V nasprotju s klasičnimi pristopi, kjer je kodiranje tista raven, ki pripelje do fizičnih rezultatov (delujoče programske kode), rezultati analize in načrtovanja pa so v začetku osnova, pozneje pa le še dokumentacija, je pri MDD analiza in načrtovanje tista raven, ki jo ves čas vzdržujemo ter iz nje generiramo programsko kodo.

- *Upoštevanje sprememb je pomembnejše od sledenja planu.* Spremembe so eden od razlogov, ki ga pogosto povezujemo z neuspešnimi projekti. Številni avtorji ugotavljajo pomen obvladovanja sprememb in temu sledijo tudi moderne metodologije. Dinamika poslovnih okolij povzroa številne spremembe, zato je naivno pričakovati, da bomo lahko v začetnih fazah projekta zajeli vse zahteve. Projektni plani so koristen element, vendar morajo omogočati spremembe, vsaj v smislu preureditve prioritet znotraj dogovorjenega okvira. Poudariti velja, da sta planiranje in sledenje planu koristna, vendar le do stanja, ko se to ne razlikuje preveč od dejanskega. Najslabše je slediti planu, ki je zastarel. Generiranje kode po MDD omogoča hiter odziv na spremenjene uporabniške zahteve. Spremembe vnesemo le v model, kodo pa generiramo. Spremembe v model na visoki abstraktni ravni vnesemo laže, kot če bi morali neposredno spreminjati programsko kodo. Pri tem ne smemo pozabiti, da se sprememba, ki jo vnesemo v PIM, hkrati prenese na vse platforme in dele kode.
- *Sodelovanje uporabnika je pomembnejše kot pogajanje na podlagi pogodb.* Stalna prisotnost in sodelovanje uporabnika v procesu razvoja je ena temeljnih zahtev, na katere opozarjajo vse agilne metodologije. Za uspešnost projekta je ključnega pomena, da se naročnik in izvajalec dobro ujameta. Medtem ko lahko dober odnos med naročnikom in izvajalcem v veliki meri olajša še tako tehnološko zahteven projekt, lahko strog pogodbeni odnos in nerazumevanje med naročnikom in izvajalcem zamajeta tudi preproste projekte. MDD kot pristop temu ne daje posebne pozornosti, vendar pa po svoji naravi odpira možnosti za vključevanje uporabnika. Upoštevati moramo, da je uporabnik v večini primerov primeren le za testiranje delujočih delov aplikacije, razni modeli in programska koda pa mu niso blizu. MDD ima ravno tu prednost, saj omogoča zelo hiter prehod z modela na delujočo aplikacijo. Podoben učinek je povzročila uporaba CASE orodij kot pripomočka pri zajemu zahtev. Uporabniki so lahko že v prvi fazi videli prototipe zaslonskih mask, ki so bile generirane neposredno iz podatkovnega modela, ter podali svoje pripombe.

MDD v osnovi ni pristop, ki bi ga lahko enačili z agilnimi metodologijami (gre za dva različna fenomena), kljub razlikam pa lahko rečemo, da se pristopa ne izključujeta. Priznati pa moramo, da to za zdaj velja le teoretično, saj so za uporabo MDD na zgoraj opisani način potrebna orodja, ki omogočajo generiranje celotne aplikacije na podlagi PIM, brez ročnega kodiranja. Uporaba MDD brez učinkovitih orodij

ni smiselna, kar je tudi glaven očitek kritikov tega pristopa.

4 Sklep

MDD lahko razumemo kot odgovor na naraščajočo kompleksnost in edalje hitrejše pojavljanje novih tehnologij in razvojnih okolij, ki jih razvijalci le stežka obvladujejo. Razvoj aplikacij na ravni, ki je neodvisna od tehnologije, v tem pogledu prinaša vrsto prednosti [6, 7, 8, 9]:

- Povečanje produktivnosti kot posledica generiranja kode. Meritve, opravljene na referenčnih projektih, kažejo, da naj bi se z uporabo MDD produktivnost povečala celo do 40 odstotkov.
- Izboljšanje arhitekture nastalih aplikacij kot posledica razvoja na višji ravni abstrakcije. Razvijalci - analitiki se lahko bolj posvetijo arhitekturnim vprašanjem ter izdelajo robustno arhitekturo, ki jo bo pozneje mogoče nadgrajevati.
- Izboljšanje konsistenčnosti nastale kode kot posledica uporabe generatorjev kode. Pogosta težava pri klasičnem razvoju je, da programerji niso usklajeni. Pri generiranju ta težava odpade, saj generator vse dele modela obravnava na enak način ter generira enotno in konsistentno kodo.

Navedene prednosti naj bi bile dosežene že z orodji in standardi, ki so dostopni danes. V prihodnosti pa naj bi po mnenju zagovornikov MDD deloval podobno, kot danes delujejo prevajalniki za programske jezike tretje generacije, ki znajo generirati učinkovito strojno kodo. Tako kot danes za izdelavo programske kode le redkokdo uporablja zbirnik ali strojni jezik, naj bi po njihovem mnenju v prihodnosti le redkokdo uporabljal sedanje programske jezike tretje generacije. Razvoj naj bi potekal na visoki ravni abstrakcije, generirani programi pa naj bi bili v povprečju ravno tako učinkoviti kot programi, ki so napisani ročno.

Predstaviti velja tudi argumente kritikov MDD, ki zatrjujejo, da je MDD oz. MDA le vizija, ki stoji na trhlih temeljih. V realnosti namreč nimamo niti ustreznih orodij za generiranje kode niti ustreznega jezika za zapis PIM, brez česar ideja MDD ne more delovati. MDD brez stoodstotno generirane kode, ki je neposredno uporabna, po njihovem mnenju ne prinaša nobenih bistvenih prednosti pred klasičnim razvojem programske opreme. Po njihovem mnenju naj bi bile negotove tudi primerjave produktivnosti klasičnega razvoja in MDD. treba je namreč upoštevati, da je z uporabo ogrodij, sodobnih razvojnih orodij in procesov mogoče produktivnost povečati tudi v okviru klasičnega razvoja.

Uveljavitev MDA bi pomenila precejšnje spremembe pri razvoju programske opreme, pa tudi spremembe v vlogah in potrebnih znanjih. Čeprav ni mogoče napovedati, ali bo MDD oz. MDA v praksi uspešen, pa lahko zatrdimo, da sta glavna pogoja za njegovo uspešnost nadaljnji razvoj jezika za zapis PIM in razvoj učinkovitih transformacijskih orodij. Čeravno so tako standardi kot orodja za podporo MDD trenutno še v začetni fazi razvoja, pa njihovo aktualnost dokazuje tudi dokaj velika podpora izdelovalcev razvojnih orodij, ki si v svojih najnovejših razvojnih orodjih prizadevajo podpreti vsaj del MDD oz. MDA.

5 Literatura

[1] J. Iivari, Why are CASE Tools not used?, *Communications of the ACM*, Vol. 39, Num. 10, pp. 94-103, 1996.

[2] A. Kleppe, J. Warmer, W. Bast, MDA Explained: The Model Driven Architecture: Practice and Promise, *Addison Wesley*, 2003.

[3] MDA Guide, *http://www.omg.org/docs/omg/03-06-01.pdf*.

[4] D. Vavpotič, M. Bajec, Vzdrževanje in uporabna vrednost modelov in dokumentacije v življenjskem ciklu informacijskih sistemov, *Proceedings of INDO*, Portorož, 2003.

[5] H. Wegener, Agility in Model-driven software development? Implications for Organizations, Process and Architecture, *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[6] L. Fernandez-Sanz, Software Engineering. A Dream Coming True?, *Upgrade*, Vol. 4, Num. 4, pp. 2-4, 2003.

[7] D. Flatner, Impact of Model-Driven Standards, National Institute of Standards and Technology, *http://www.omg.org/mda/mda_files/mda_1.7_cleanformat.pdf*.

[8] The middleware company, Model Driven Development for J2EE Utilizing a MDA Approach, 2003.

[9] A. Tolk, Avoiding another Green Elephant - A Proposal for the Next Generation HLA based on the Model Driven Architecture, *Simualtion Interoperability Workshop*, Orlando, Florida, september 2002.

Damjan Vavpotič je asistent na Fakulteti za računalništvo in informatiko v Ljubljani. Raziskovalno se ukvarja z informacijskimi sistemi, metodologijami razvoja informacijskih sistemov in modelno vodenim razvojem programske opreme.

Marjan Krisper je docent na Fakulteti za računalništvo in informatiko v Ljubljani in predstojnik Katedre za informatiko. Raziskovalno se ukvarja z informacijskimi sistemi, razvojem in prenovo informacijskih sistemov, strateškim planiranjem in elektronskim poslovanjem.