Modelinės architektūros naudojimas kuriant komponentines programų sistemas

Vaidas Giedrimas

Šiaulių universiteto lektorius
Lecturer of Šiauliai University
P. Višinskio g. 19, LT-77156 Šiauliai
Tel. +370 650 72042
El. paštas: vaigie@mi.su.lt

Modelinė architektūra gana paplitusi, tačiau jos taikymai komponentinei paradigmai yra tik daliniai – neatsižvelgiama į reikalavimą atskirti komponentų ir komponentinių sistemų kūrimo procesus. Šio straipsnio tikslas – aprašyti modelinės architektūros naudojimo automatizuotai kuriant komponentines programų sistemas iš binarinių komponentų metodą, atsižvelgiant į Išrink–Pritaikyk–Testuok gyvavimo ciklo reikalavimus. Straipsnyje atskleista komponentinių programų abstrakčiųjų ir konkrečiųjų modelių specifika, aprašytos jų transformacijos. Nustatyta, kad modelinė architektūra su pakeitimais gali būti naudojama ir komponentinių programų sistemų kūrimo iš binarinių komponentų procesui automatizuoti, pateiktos jos tobulinimo gairės.

Komponentinė paradigma užtikrina efektyvesnį programų sistemų kūrimą iš pakartotinai naudojamų binarinių komponentų, lengvesnį gautų sistemų testavimą, modifikavimą ir palaikymą. Kuriama ir tobulinama komponentinių programų sintezės sistema (Giedrimas, 2007), skirta šiam procesui automatizuoti. Viena iš šios sistemos problemų – tikslios specifikacijos problema. Egzistuoja didelis atotrūkis tarp dalykinės srities, kuriai kuriama programinė įranga, ir realizacinės srities (konkrečių komponento modelių, karkasų, operacinių sistemų ir t. t.) konceptų. Šis atotrūkis sunkina komponentinių sistemų specifikavimą ir realizavimą (konkrečiai – komponentų paiešką ir adaptavimą). Sąlyginai nauja modelinė architektūra (angl. *Model-Driven architecture – MDA*) sukurta siekiant tokį atotrūkį paversti pranašumu. Plačiai aprašomi tiek bendrieji modelinės architektūros principai (Kleppe, 2003; Mellor et al., 2004; Pastor, Molina, 2007), tiek jos taikymo konkrečiai programų kūrimo paradigmai atvejai (Alti et al., 2007; López-Sanz et al., 2008; Xiao, 2009). Tačiau (Alti et al., 2007; Kum, Kim, 2006) siūlomi MDA taikymai komponentinių programų sistemoms kurti nevisiškai atitinka komponentinę paradigmą. Pažeidžiamas vienas komponentinės paradigmos principų – komponentų ir komponentinių sistemų kūrimo procesų atskyrimas (Kaisler, 2005; Crnkovic, Larsson, 2003; Szyperski, 2002), nes naudojant (Alti et al., 2007; Kum, Kim, 2006) aprašytus metodus sistemos kuriamos iš čia pat generuotų komponentų. Šio straipsnio tikslas – aprašyti modelinės architektūros naudojimo automatizuotai kuriant komponentines programų sistemas iš binarinių komponentų metodą, atsižvelgiant į *Išrink–Pritaikyk–Testuok* gyvavimo ciklo reikalavimus.

Komponentinių programų sistemų gyvavimo ciklas

Komponentinių programų sistemų inžinerijoje skiriami du procesai: komponentų kūrimas ir komponentinių sistemų kūrimas (Crnkovic, Larsson, 2003; Szyperski, 2002). Komponentinių

programų sistemų (KPS) kūrimo procese pabrėžiamas ne suprojektuotos sistemos ir jos dalių realizavimas, bet jau sukurtų binarinių komponentų pakartotinis naudojimas. Komponentinių programų sistemų gyvavimo ciklą (KPSGC) apima reikalavimų analizės, komponentų atrankos, sistemos projektavimo, realizavimo, sistemos integravimo, verifikavimo, sistemos palaikymo etapai. KPSGC yra ypatingas tuo, kad palyginti su klasikiniu programų sistemų gyvavimo ciklu, keičiasi etapų turinys:

- Komponentų atrankos etapo svarba padidėja. Tiek programų sistemos kokybė, tiek kūrimo išlaidos tiesiogiai priklauso nuo šiame etape rastų komponentų savybių.
- Realizavimo etapo svarba ir trukmė sumažėja. Idealiu atveju sistemai sukurti pakanka vien pakartotinai naudojamų binarinių komponentų. Jei komponentai nevisiškai atitinka reikalavimus, jie konfigūruojami arba kuriamas nedidelės apimties jungiantysis programinis kodas (angl. *glue-code*).

Pastebėtina, kad komponentų atrankos, sistemos projektavimo, realizavimo bei sistemos integravimo etapai iteratyviai kartojami tol, kol programų sistema maksimaliai tenkins ne tik funkcinius, bet ir nefunkcinius reikalavimus. Šis gyvavimo ciklas dar vadinamas *Išrink–Pritaikyk–Bandyk* (angl. *Select–Adapt–Test*) ciklu (Crnkovic, Larsson, 2003).

Modelinė architektūra

MDA – tai OMG konsorciumo palaikoma programinės įrangos kūrimo iniciatyva, pabrėžianti modelių svarbą programinės įrangos kūrimo procese. Programų sistemų, kuriamų naudojant MDA, gyvavimo ciklą, kaip ir klasikinį gyvavimo ciklą, sudaro: reikalavimų rinkimas, analizė, projektavimas, kodavimas, testavimas bei tiražavimas ir palaikymas (Kleppe, 2003). Tačiau skiriasi kiekvieno iš šių etapų automatizavimo laipsnis ir rezultatai.

Analizės etapo rezultatas – abstraktusis modelis (angl. *Platform-Independent Model – PIM*). Kuriant programų sistemos PIM modelį operuojama tik dalykinės srities sąvokomis, nesigilinant į realizacines detales, tokias kaip programavimo kalba, DBVS ir pan.

Projektavimo etapo rezultatas – konkretusis modelis (angl. *Platform-Specific Model – PSM*). PSM modelyje operuojama jau realizacinėmis sąvokomis, detalizuojami nuo konkrečios operacinės sistemos, karkaso, programavimo kalbos ir kt. veiksnių priklausomi sprendimai.

Darbe (Mellor et al., 2004) pabrėžiama, kad šiuolaikinė programinė įranga yra labai sudėtinga, todėl MDA procesas gali apimti ne vieną, o keletą PIM ir keletą PSM modelių.

Dar vienas MDA pranašumas – galimybė automatizuoti programų sistemų kūrimo procesą. Visiško arba dalinio automatizavimo objektai yra (GMT project, 2009; Pastor, Molina, 2007):

- abstrakčiojo modelio neprieštaringumo tikrinimas;
- abstrakčiojo modelio transformacija į konkretųjį modelį;
- konkrečiojo modelio transformacija į programinį kodą;
- vieno konkrečiojo modelio transformacija į kitą (pvz., kitai OS skirtą) konkretųjį modelį.

Taip pat kuriami ir atvirkštinių transformacijų įrankiai.

Automatizuotas pakartotinis PSM modelio ir atitinkamai programinio kodo generavimas pasikeitus PIM modeliui sudaro galimybes operatyviai reaguoti į reikalavimų pokyčius.

Komponentinių programų abstraktieji modeliai

Abstraktusis modelis gali būti aprašomas tik dalykinės srities sąvokomis, tačiau siekiant, kad modelis atitiktų komponentinę paradigmą, tikslinga ir jame naudoti komponento konceptus. Akivaizdu, kad čia būtų operuojama dalykinės srities (verslo) komponentais, o ne programiniais komponentais (Lupeikienė, 2006).

Z. Stojanovič (2005) skiria du abstrakčiuosius modelius: *verslo komponentų modelį* ir *taikomosios programos modelį* (angl. *application component model*). S. Gobel (2005) siūlo naudoti

adaptyviuosius komponentus, kurie gali būti konfigūruojami ir pritaikomi tiek jų kūrimo, tiek paskirstymo, tiek vykdymo metu.

Siekiant užtikrinti kuo didesnį modelio tikslumą siūloma naudoti UML 2.0 kalbos poaibį – *vykdomąją UML* kalbą (Mellor et al., 2004). Pagrindinis šios kalbos pranašumas – galimybė itin išsamiai aprašyti esybių dinamiką, ir tai sudaro sąlygas gautus vykdomuosius modelius įvykdyti ir patikrinti jau šiame etape. Be tradicinėmis tapusių UML kalbos priemonių, vykdomojoje UML įvesta *klasių veiksmų* (angl. *Class Actions*) semantika (Mellor, Balcer, 2002). Klasių (šiuo atveju – esybių) veiksmai aprašomi specialia *klasių veiksmų kalba* (KVK) (OMG, 2002). Klasių veiksmų naudojimas padeda aprašyti dalykinės srities logiką, o vėliau – vykdymo logiką, abstrahuojantis nuo konkrečių platformų ir kitų detalių. Pažymėtina, kad kaip veiksmų aprašymo kalbą galima naudoti ne tik KVK, bet ir kitas veiksmų aprašymo kalbas, pavyzdžiui, Schlaer-Mellor (Mellor, Balcer, 2002) kalbą. Ribojimus, kuriuos privalo tenkinti dalykinės srities komponentai, tikslinga aprašyti OCL kalba.

Vykdomoji UML kalba sukurta profiliavimo būdu praplečiant UML 2.0 kalbos bazę (Mellor et al., 2004) ir gali būti lengvai modernizuojama sukuriant kitą profilį, labiau pritaikytą komponentinei paradigmai. Be to, naudojant MOF (angl. *Meta Object Facility*) metamodelių kūrimo konstrukcijų aibę (OMG, 2006) galima kurti konkrečių dalykinių sričių modeliavimo priemones.

Komponentinių programų konkretieji modeliai

Konkretusis modelis, aprašytas vykdomąja UML kalba, suvokiamas kaip modelio kompiliavimo proceso parametras (Mellor, Balcer, 2002). Šiame modelyje jau atsispindi komponento modelio, karkaso ir kt. realizacinės detalės. Realizuojant programų sistemą gali tekti atsižvelgti į įvairius jos aspektus, todėl tikslinga naudoti keletą konkrečiųjų (vienas iš kito gaunamų) modelių, kurių skaičius priklauso nuo programų sistemos sudėtingumo (Kleppe 2003; Mellor et al., 2004). Pavyzdžiui, S. Gobel (2005) išskiria tris konkrečiuosius modelius: kūrimo, paskirstymo ir vykdymo. Po atskirą konkretųjį modelį turėtų būti kuriama ir kiekvienam kitam komponento modeliui. Pavyzdžiui, nusprendus naudoti nebe CORBA, o .NET komponentus, būtina generuoti naują konkretųjį modelį.

Konkrečiajam modeliui aprašyti tikslingiausia taip pat naudoti vykdomąją UML kalbą (Kleppe, 2003), nors galimi ir kitų kalbų (Fabresse et al., 2008; Pastor, Molina, 2007) naudojimo atvejai.

Pažymėtina, kad automatizuotu būdu iš abstrakčiojo modelio transformacijos būdu gautas konkretusis modelis gali būti neišsamus, neatitikti realizacinės aplinkos. Pavyzdžiui, jame aprašyti kol kas neegzistuojantys komponentai, arba kelių realių komponentų funkcijos aprašytos kaip vieno. Modeliui patikslinti siūloma naudoti *detalizavimo* (angl. *Model elaboration*) priemones.

Abstrakčiojo modelio transformacija

S. Gobel (2005) nagrinėja adaptyviuosius komponentus, be kitų interfeisų turinčius ir valdymo bei parametrizavimo interfeisus, kurie naudojami komponento konfigūravimui ir adaptavimui. Manoma, kad kiekvienas abstrakčiajame modelyje naudojamas dalykinės srities komponentas turi konkrečius realizacinius atitikmenis. Tokiu atveju modelių transformacija yra gana paprasta. Tačiau iš tikrųjų dalykinės srities ir realizacinės dalies komponentų prasmė, skaičius ir kitos savybės gali įvairuoti, vienareikšmė atitiktis įmanoma tik idealiu atveju, todėl darbe (Gobel, 2005) išdėstytas požiūris yra netikslus.

Abstrakčiojo modelio transformacijos į konkretųjį modelį etape būtina atsižvelgti į *Išrink–Pritaikyk–Testuok* gyvavimo ciklo specifiką. Tradiciniu būdu taikant MDA šiame etape generuojamas konkretusis modelis, kuris parodo, kokia sistema turėtų būti *sukurta*. Šiuolaikiniuose komponento modeliuose komponentas suprantamas kaip „juodoji dėžė“ su minimaliomis konfigūravimo priemonėmis, todėl konkretusis modelis turi parodyti, kokie esami komponentai ir kaip

turi būti panaudoti. Iš to išeina, kad transformacijos metu privalo būti atliekami dar ir papildomi veiksmai:

- esamų realizacinių komponentų paieška ir atranka saugykloje;
- atskirų realizacinių komponentų konfigūracijų kitimo ribų tyrimas ir komponentų konfigūravimas (jei įmanomas);
- trūkstamų realizacinių komponentų identifikavimas.

Transformacijos uždavinys patenka į grupę uždavinių, kurie jau yra sprendžiami kuriamoje komponentinių programų sistemų sintezės sistemoje (Giedrimas, 2007). Tai sudaro prielaidas naudoti MDA kuriant komponentines programų sistemas, nes pakanka sukurti specialius abstrakčiojo modelio transformacijos įrankius naudojančius struktūrinės sintezės ir induktyvųjį generavimo metodus.

Konkrečiojo modelio transformacija

Komponentinės programos konkrečiojo modelio transformacija į programinį kodą ypatinga tuo, kad realizaciniai komponentai yra binariniai:

- Generuojamo kodo apimtis yra daug mažesnė nei tradicinėje MDA. Generuojamas tik jungiantysis kodas ir trūkstami realizaciniai komponentai identifikuoti ankstesnės transformacijos metu (jei tokių buvo).
- Būtina patikrinti, ar realizaciniai komponentai („juodosios dėžės“) ir jų veiksmų semantika atitinka dalykinės srities komponentams keliamus reikalavimus (įskaitant ir veiksmų semantiką). Šiam palyginimui atlikti reikalingas specialus įrankis. Pavyzdžiui, .NET komponentų atveju toks įrankis turėtų formuoti atitiktis tarp refleksijos būdu apklausto komponento.
- IL kodo ir vykdomosios UML veiksmų semantikos.

Pažymėtina, kad tiek abstrakčiojo modelio transformacija, tiek konkrečiojo modelio transformacijos gali būti iteratyviai kartojamos.

Išvados

Analizė parodė, kad modelinės architektūros metodas gali būti naudojamas automatizuotam KPS kūrimui ir iš binarinių komponentų. Atsižvelgiant į *Išrink–Pritaikyk–Testuok* gyvavimo ciklo specifiką siūlomi šie MDA pakeitimai:

1. Abstrakčiojo modelio transformacijos į konkretųjį modelį etape būtina atlikti papildomus veiksmus: esamų realizacinių komponentų paiešką ir atranką saugykloje, atskirų realizacinių komponentų konfigūracijų kitimo ribų tyrimą ir komponentų konfigūravimą, trūkstamų realizacinių komponentų identifikavimą. Siekiant procesą automatizuoti, būtina sukurti tarpusavyje suderinamų įrankių rinkinį.

2. Konkrečiojo modelio transformacijos į programinį kodą etape būtina patikrinti, ar realizaciniai komponentai atitinka dalykinės srities komponentams keliamus reikalavimus. Komponentų semantikos palyginimui atlikti reikalingas specialus įrankis.

3. Būtina sudaryti galimybes iteratyviai kartoti abstrakčiojo ir konkrečiojo modelių transformacijas tol, kol programų sistema maksimaliai tenkins funkcinius ir nefunkcinius reikalavimus. Konkrečiajam modeliui patikslinti siūloma naudoti detalizavimo priemones.

LITERATŪRA

ALTI, A. et al. (2007). Integrating software architecture concepts into the MDA platform with UML profile. *Journal of Computer Science,* vol. 3 (10), p. 93–802.

CRNKOVIC, I.; LARSSON, M. (2003). *Building Realiable Component-Based Software Systems.* Eds. Crnkovic I., Larsson M. London: Artech House.

FABRESSE, L. et al. (2008). Foundations of a simple and unified component-oriented language. *Computer Languages, Systems & Structures,* vol. 34, issues 2–3, p. 130–149.

GIEDRIMAS, V. (2007). Generavimo metodų panaudojimas kuriant .NET komponentines programų sistemas. *Informacijos* mokslai, t. 42–43, p. 246–250.

GMT Project (2009). Prieiga per internetą: http://www.eclipse.org/gmt/ [žiūrėta 2009 m. gegužės 15 d.].

GOBEL, S. (2005). An MDA Approach for Adaptable Components. Iš *Model driven architecture – foundations and applications: first European conference, ECMDA-FA 2005 proceedings,* p. 74–87.

KAISLER, S.H. (2005). *Software paradigms*. Wiley-interscience. ISBN 0-471-48347-8.

KLEPPE, A. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston: Addison-Wesley.

KUM, D. K., KIM, S. D. (2006). A Systematic Method to Generate .NET Components from MDA/PSM for Pervasive Service. Iš *Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*, p. 324–331.

LÓPEZ-SANZ, M. et al. (2008). Modelling of Service-Oriented Architectures with UML. *Electron. Notes Theor. Comput. Sci.,* 194, 4 (Apr. 2008), p. 23–37.

LUPEIKIENĖ, A. (2006). Integrated Enterprise Information System Development through Component Abstraction. Iš *Proceedings of the Seventh International Baltic Conference on Databases and Information Systems*, O. Vasilecas, J. Eder, A. Čaplinskas (eds.), Institute of Electrical and Electronics Engineers, p. 168–174.

MELLOR, J.; BALCER, M. J. (2002). *Executable UML: A Foundation for Model-Driven Architecture*. Boston: Addison-Wesley. ISBN 0-201-74804-5.

MELLOR, J. et al. (2004). *MDA Distilled*: *Principles of Model-driven Architecture*. Addison-Wesley. ISBN 0-201-78891-8.

OMG (2002). *Unified modelling language Specification (Action Semantics). Version 2.0* [žiūrėta 2009 m. gegužės 2 d.]. Prieiga per internetą: http://www.omg.org/cgi-bin/doc.cgi?ptc/02-01-09.pdf.

OMG (2006). *Meta Object Facility (MOF) Core Specification. Version 2.0* [žiūrėta 2009 m. gegužės 2 d.]. Prieiga per internetą: http://www.omg.org/spec/MOF/2.0.

PASTOR, O.; MOLINA, J. C. (2007). *Model-Driven Architecture in Practice*. Springer. ISBN 978-3-540-71867-3.

STOJANOVIČ, Z. (2005). *A Method for Component-Based and Service-Oriented Software Systems Engineering*. PhD thesis. Delft University of Technology. ISBN 90-9019100-3.

SZYPERSKI, C. (2002). *Component Software: Beyond Object-Oriented Programming*. 2nd. Addison-Wesley Longman Publishing Co., Inc.

XIAO, L. (2009). An adaptive security model using agent-oriented MDA. *Inf. Softw. Technol.*, vol. 51, no. 5, p. 933–955.

THE APPLICATION OF MDA FOR COMPONENT-BASED SOFTWARE DEVELOPMENT

Vaidas Giedrimas

Summary

The Model-Driven Architecture is relatively widely used but the conjunctions of MDA and Component-Oriented Paradigm are still partial only. According Component-Oriented Paradigm there is clear division of component-based software engineering to the component development and component-based development, but unfortunately in existing approaches this principle is avoided. The aim of this article is to describe the method of MDA in automated component-based software synthesis from binary components process, having in mind the singularities of *Select-Adapt-Test* lifecycle. The key points of modified component-based MDA are covered including platform-independent models, platform-specific models and transformations.