IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITB12861

This paper appears in the book, *Emerging Trends and Challenges in Information Technology Management, Volume 1 and Volume 2* edited by Mehdi Khosrow-Pour © 2006, Idea Group Inc.

MDA-Based Design Pattern Components

Liliana Martinez, INTIA - Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil - Argentina, lmartine@exa.unicen.edu.ar

Liliana Favre, INTIA - Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil - Argentina, & CIC, Buenos Aires, lfavre@exa.unicen.edu.ar

ABSTRACT

The Model Driven Architecture (MDA) is an initiative of the Object Management Group (OMG) that promotes the use of models for developing software systems. It distinguishes at least three different kinds of models: Platform Independent Model (PIM), Platform Specific Model (PSM) and Implementation Specific Model (ISM). The concepts of models, metamodels and model transformations are at the core of MDA. In this paper we describe a meta-modeling technique to define design pattern components from a MDA perspective. In this context, we propose a "megamodel" for defining reusable components that integrates different kinds of models with their respective metamodels. We analyze metamodel-based model transformations among levels of PIMs, PSMs and ISMs. We illustrate the approach to define reusable design pattern components using the Observer pattern.

1 INTRODUCTION

The Model Driven Architecture (MDA) is an initiative of the Object Management Group (OMG) that promotes the use of models and model transformations for developing software systems (MDA, 2005). MDA distinguishes at least three kinds of models: Platform Independent Model (PIM), Platform Specific Model (PSM) and Implementation Specific Model (ISM). A PIM is a model that contains no reference to the platforms that are used to realize it. A PSM describes a system in the terms of the final implementation platform e.g., .NET or J2EE. An ISM refers to components and applications.

A model driven development is carried out as a sequence of model transformations that includes at least the following steps: construct a PIM; transform the PIM into one or more PSMs, and construct executable components and applications directly from the PSMs (Kleppe et al., 2003; MDA, 2005).

Metamodeling has become an essential technique to support model transformations. In MDA metamodels are expressed using MOF (Meta Object Facility) that defines a common way for capturing all the standard and interchange constructs. They are expressed as a combination of UML class diagrams and OCL constraints (UML, 2005; OCL, 2005). The 4 main core metamodeling constructs are classes, binary associations, data types and package.

MDA is a young approach and several technical issues are not adequately addressed. The success of MDA depends on the definition of model transformations and component libraries which make a significant impact on MDA-based tools. To date existing Case tools (CASE TOOLS, 2005) do not provide adequate support to deal with component-based reuse and MDA.

In this paper we analyze a technique to reach a high level of reusability and adaptability of MDA design pattern components. Design pattern describes solutions to recurring design problems. Arnout (2004) analyzes the popular Gamma´s design patterns (Gamma et al., 1995) to identify which ones can become reusable components in an Eiffel library. Their work hypothesis is that "design patterns are good, but the components are better" because they are reusable in terms of code. Our work takes up these ideas and contributes a metamodeling technique to built reusable design pattern components in a MDA perspective.

We propose a "megamodel" to define families of design pattern components in a way that fits MDA. A "megamodel" is a set of elements that represent and/or refer to UML-based models and metamodels at different levels of abstraction (Bezivin et al., 2004). Design Pattern components are described at three different abstraction levels: Platform Independent Component Model (PICM), Platform Specific Component Model (PSCM) and Implementation Component Model (ICM). The subcomponents in the different levels include model and metamodel specifications and their interrelations. Metamodels allow defining as many components as different pattern solutions can appear in a concise way. We analyze metamodel-based model transformations of both PIMs into PSMs, and PSMs into ISMs. We illustrate the approach by using the Observer pattern.

This paper is organized as follows. Section 2 describes a megamodel for defining design pattern components. Section 3 describes how to specify components at the metamodel level. Section 4 shows how to specify model transformations as OCL contracts. Section 5 deals with the related work and compares our approach with other existing ones. Finally, Section 6 considers conclusions and future work.

2 A "MEGAMODEL" FOR DEFINING DESIGN PATTERN COMPONENTS

Most current approaches to reusability in the context of MDA are based on empirical methods focusing on reuse of code models; however the most effective forms of reuse are generally found at more abstract levels of design. Reusability is difficult because it requires taking many different requirements into account, some of them are abstract and conceptual, whereas others such as efficiency are concrete. A good approach for reusability must reconcile them. This work proposes an approach for defining reusable design pattern components that integrate high level specifications, which are independent of any implementation technology, specifications targeted at different platforms and implementations.

To define reusable components we propose a "megamodel" that integrates PIMs, PSMs and code with their respective metamodels. Fig. 1 shows the different correspondences that may be held between several models and metamodels.

We define MDA components at three different levels of abstraction: Platform Independent Component Model (PICM), Platform Specific Component Model (PSCM) and Implementation Component Model (ICM). The PICM includes a UML/OCL metamodel that describes a family of all those PIMs that are instances of the metamodel. A PICM-metamodel is related to more than one PSCM-metamodels. The PSCM includes UML/OCL metamodels that are linked to specific platforms and

Figure 1. Defining MDA components

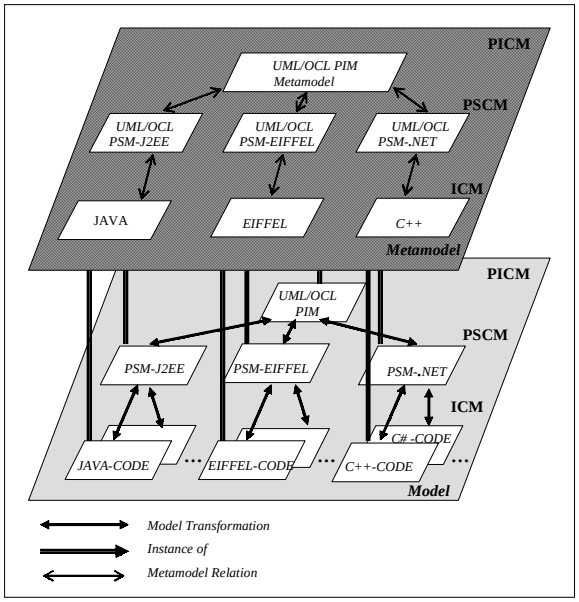
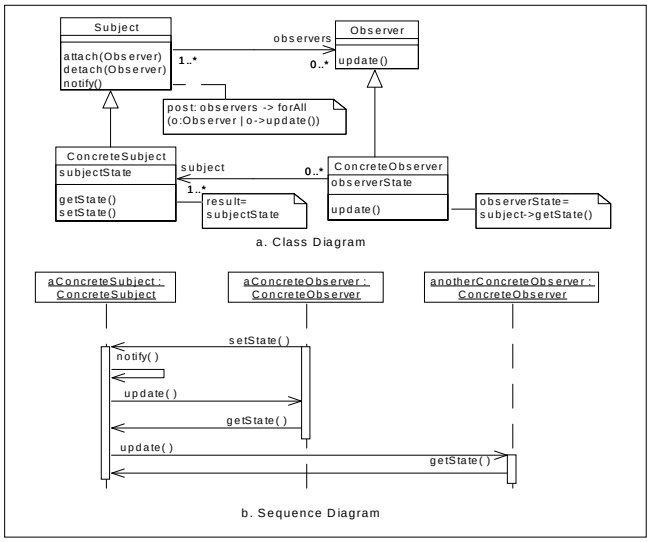


Figure 2. Observer design pattern



a family of PSM-models that are instances of the respective PSCM-metamodel. Every one of them describes a family of PSM instances. PSCM-metamodels correspond to ICM-metamodels (Fig.1).

Metamodels are expressed as MOF-metamodel whose their main core metamodeling constructs are classes, binary associations, data types and package. A model transformation is a specification of a mechanism to convert the elements of a model, that are instances of a particular metamodel, into elements of another model which are instances of the same or different metamodels.

We define Design Pattern components at three different abstraction levels: the PICM includes a PIM metamodel whose instances are several pattern solutions. PIM-metamodels describe all the concepts of the structural and interaction view of pattern solutions. The PSCM level includes metamodels for particular platforms and, the ICM level includes metamodels for different programming languages.

3 SPECIFYING DESIGN PATTERN COMPONENTS

A design pattern describes a family of solutions for recurring design problems. In this section we analyze the Observer pattern (Gamma et al., 1995). We present a description of the Observer Pattern, a PIM-metamodel and a PSM-metamodel based on an Eiffel platform.

3. 1 Observer Pattern Description

The design pattern Observer "defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" (Gamma et al., 1995, p. 293). The pattern Observer involves the following participants (Gamma et al., 1995):

- *Subject*: Any number of *observer* objects may observe a *subject*. It keeps a collection of observers and provides an interface for add and remove observer objects (*Attach and Detach*). Whenever its state changes (typically the values of some of its attributes change), the subject will notify its observers (*notify*).
- *Observer*: It can observe one or more subjects. It defines an updating interface for objects that should be notified of changes in a subject (*update*).
- *Concrete Subject (ClockTimer)*: It stores state of interest to *ConcreteObserver* objects and sends a notification to its observers when its state changes.
- *Concrete Observer (AnalogClock and DigitalClock)*: It maintains a reference to a *ConcreteSubject* object. It stores state that should stay consistent with the subjects and implements the observer updating interface to keep its state consistent with the subjects (*Update*).

Fig. 2 shows a UML class diagram and a UML sequence diagram of a typical application using Observer pattern.

3. 2 PIM-Metamodel of the Observer Pattern

The Observer pattern metamodel at PIM level specifies the structural and behavior views of this pattern in a platform independent pattern model. That is, it specifies the classes that participate, its operations and attributes and the relationship between classes.

There are four essential participants: *Subject*, *Observer*, *ConcreteSubject* and *ConcreteObserver*. So, these four classes must be specified in the metamodel, as well as the relationship between them and their interactions.

The specialized UML metamodel of the Observer pattern is partially shown in Fig. 3. The shaded metaclasses correspond to metaclasses of the UML metamodel, whereas the remaining corresponds to the specialization of the UML metamodel of the Observer pattern. Fig. 4 partially shows some well-formedness rules in OCL for the metamodel.

Semantics

AbstractObserver. This metaclass specifies the characteristics of Observer class inside the Observer pattern. It should have at least an operation with the characteristics of *Update*. Each instance of this metaclass can be an abstract class or an interface. If the instance is an abstract class, a concrete observer inherits its behavior, therefore there is an inheritance relationship with the concrete observer. If the instance is an interface, there is a realization relationship with the concrete observer.

ConcreteObserver. This metaclass specifies the characteristics of a concrete observer. It knows the subject (or the subjects), then it is associated to *ConcreteSubject* through a unidirectional association navigable away from that end.

AbstractSubject. Each instance of this metaclass can be an abstract class or an interface and it has at least three operations specified by *Attach*, *Detach* and *Notify*. If the instance of this metaclass is an abstract

Figure 3. A simplified observer pattern metamodel

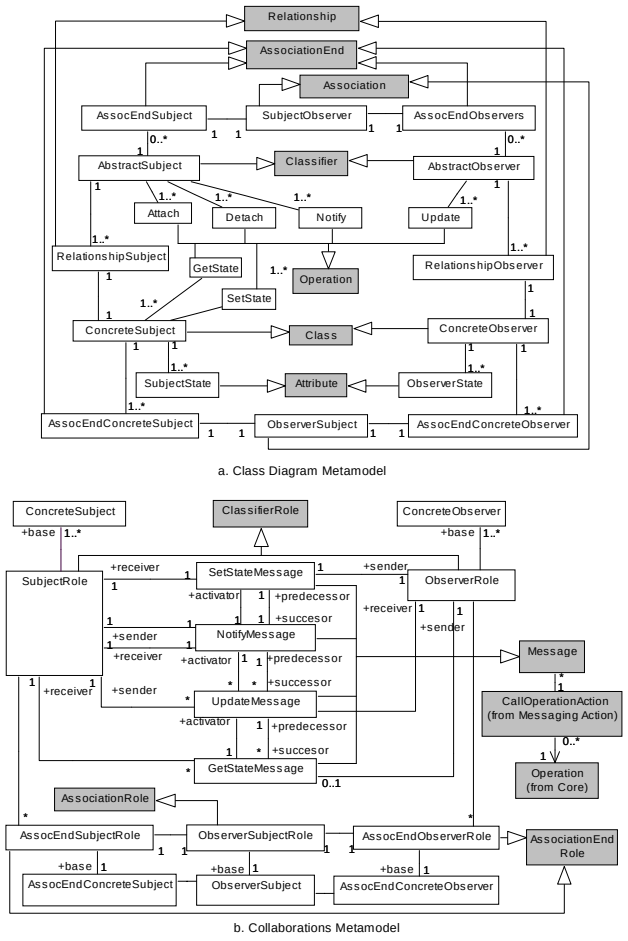


Figure 4. Observer pattern metamodel: Some well-formedness rules

Well-formedness rules – Class Diagram Metamodel

context *AbstractObserver* **inv***:*
(self.oclIsTypeOf(Class) **and** self.isAbstract = #true) **or** self.oclIsTypeOf(Interface)

context *AbstractSubject* **inv:**
(self.oclIsTypeOf(Class) **and** self.isAbstract = #true) **or** self.oclIsTypeOf(Interface)
and self.oclIsTypeOf(Interface) **implies** self.assocEndSub ->isEmpty( )

context *AssocEndConcreteSubject* **inv:**
self.isNavigable= #true **and** (multiplicity.range.lower= 0 **or** multiplicity.range.lower> 0)
and (self.multiplicity.range.upper > 0 **or** self.multiplicity.range.upper = #unlimited)

context *AssocEndSubject* **inv:**
self.isNavigable= #false **and** (multiplicity.range.lower= 0 **or** multiplicity.range.lower> 0)
and (self.multiplicity.range.upper > 0 **or** self.multiplicity.range.upper = #unlimited)

context *Attach* **inv:**
self.isQuery= #false **and** self.parameter->notEmpty() **and** self.parameter->select(param |
param.kind= #in **and** param.type= oclIsKindOf(AbstractObserver)) -> size( ) = 1

context *ConcreteObserver* **inv:**
self.oclIsTypeOf(Class) **and** self.isAbstract = #false

context *ConcreteSubject* **inv:**
self.oclIsTypeOf(Class) **and** self.isAbstract = #false

context *RelationObs* **inv:**
self.oclIsTypeOf(Generalization) **or**
(self.oclIsTypeOf(Abstraction) **and** self.stereotype.name= 'realize')
and self.oclIsTypeOf(Generalization) **implies** (self.parent.oclIsKindOf(Class) **and**
self.parent.oclIsTypeOf(AbstractObserver) **and** self.child.oclIsTypeOf(ConcreteObserver))
and self.oclIsTypeOf(Abstraction) **implies** (self.supplier.oclIsKindOf(Interface) **and**
self.supplier.oclIsTypeOf(AbstractObserver) **and** elf.client.oclIsTypeOf(ConcreteObserver))
…

class, all concrete subjects inherit its behavior, therefore there is an inheritance relationship with the concrete subject.

If the instance is an interface, there is a realization relationship with the concrete subject. If the instance of *AbstractSubject* is an abstract class, it is associated to an instance of *AbstractObserver* through a unidirectional association navigable away from that end.

ConcreteSubject. This metaclass specifies the characteristics of a concrete subject. It has at least two operations specified by GetState and SetState and its internal state is specified by the ObserverState metaclass.

3. 3 PSM-Metamodel of the OBSERVER Pattern

For each design pattern at the PIM level there are a number of metamodels corresponding to different platforms. The metamodel of the observer pattern in an EIFFEL platform is described below. Fig. 5 partially shows the UML specialized metamodel of the Observer pattern in the Eiffel platform (structural view) and some OCL rules. The shaded metaclasses correspond to metaclasses of the Eiffel metamodel.

4 SPECIFYING METAMODEL-BASED MODEL TRANSFORMATIONS

A model transformation is a specification of a mechanism to convert the elements of a model, that are instances of a particular metamodel, into elements of another model which can be instances of the same or different metamodels. Metamodel transformations are a specific type of model transformations that impose relations between pairs of metamodels. They can be used in the specification stages of the MDA-based developments to check the validity of a transformation.

We specify metamodel-based model transformations as OCL contracts that are described by means of a transformation name, parameters, local operations, preconditions and postconditions. The precondition states the conditions that must hold whenever the transformation is applied. The postcondition states the properties that the transformation guarantees when it was applied.

Next, we partially exemplify a transformation from a PIM to an Eiffel-based PSM that is included in an Observer Pattern component. The definition of the transformation from PIM to PSM uses both the specialized UML metamodel of the Observer pattern and the UML metamodel of an Eiffel platform as source and target parameters respectively. The source metamodel describes a family of packages whose elements are only classes and associations. The postconditions establish correspondences among classes, their superclasses, parameters, operations, and associations. The transformation specification guarantees, for instance, that for class *sourceClass* in the source model exists a class *targetClass* in the target model, both of them with the same name, the same parent classes, the same child classes and so on. (See Box A.)

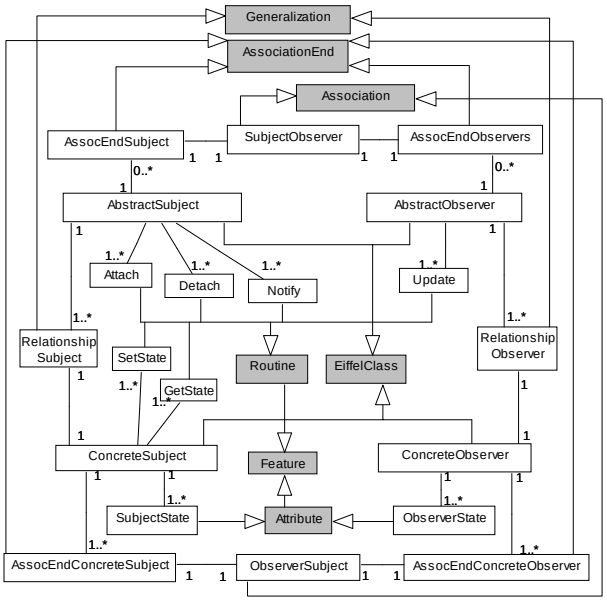
5 RELATED WORK AND DISCUSSION

In Budinsky et al. (1996) a tool to automatically generate code of design patterns from a small amount of information given by the user is described. This approach has two widespread problems. The user should understand "what to cut" and "where to paste" and both cannot be obvious. Once the user has incorporated pattern code in his application, any change that implies to generate the code again will force it to reinstate the pattern code in the application. The user cannot see changes in the generated code through the tool.

Florijn et al. (1997) describes a tool prototype that supports design pattern during the development or maintenance of object-oriented programs.

Albin-Amiot & Guéhéneuc (2001a) describes how a metamodel can be used to obtain a representation of design patterns and how this representation allows both automatic generation and detection of design patterns. The contribution of this proposal is the definition of design patterns as entities of modeling of first class. The main limitation of this

Figure 5. Observer pattern metamodel - EIFFEL platform



context *AbstractObserver* **inv:**
self.isAbstract = #true

context *AbstractSubject* **inv:**
self.isAbstract = #true

context *RelationshipObserver* **inv:**
self.parent.oclIsTypeOf(AbstractObserver) **and** self.child.oclIsTypeOf(ConcreteObserver)

context *RelationshipSubject* **inv:**
self.parent.oclIsTypeOf(AbstractSubject) **and** self.child.oclIsTypeOf(ConcreteSubject)
...

Box A.

```
Transformation PIM-UML TO PSM-EIFFEL {

parameters
    sourceModel: Design Pattern Metamodel :: Package
    targetModel: Design Pattern Metamodel-EIFFEL :: Project
local operations
    equivalentType (a_type: Design Pattern Metamodel::Classifier,
      another_type: Design Pattern Metamodel-EIFFEL::Classifier): Boolean
 ...
pre:
    sourceModel.importedElement  isEmpty

post:
    sourceModel.ownedElement  select(oclIsTypeOf(Class))  size() +
        sourceModel.ownedElement  select(oclIsTypeOf(Interface))  size() =
        targetModel.ownedElement  select(oclIsTypeOf(EiffelClass))  size()
post:
    sourceModel.ownedElement  select(oclIsTypeOf(Class))  forAll ( sourceClass /
        targetModel.ownedElement.(oclIsTypeOf(EiffelClass))  exists ( targetClass /
        sourceClass.name = targetClass.name  and
        sourceClass.generalization.parent = targetClass.generalization.parent  and
        sourceClass.specialization.child = targetClass.specialization.child  and
        sourceClass.templateParameter = targetClass.templateParameter  and
        sourceClass.feature  select (oclIsTypeOf(Attribute))  forAll ( sourceAtt /
                targetClass.feature  select (oclIsTypeOf( Attribute))  exists ( targetAtt /
            sourceAtt.name = targetAtt.name  and
            sourceAtt.visibility = targetAtt.visibility and
            equivalentType(sourceAtt.type, targetAtt.type) ))  and
        sourceClass.feature  select (oclIsTypeOf(Operation))  forAll (sourceOp /
            targetClass.feature  select (oclIsKindOf(Routine))  exists (targetOp /
            targetOp.name = sourceOp.name and
            equivalentType (targetOp.type, sourceOp.type) and
            sourceOp.parameter  size() = targetOp.parameter  size()  and
                Sequence {1..( sourceOp.parameter  size())}  forAll ( index:Integer /
                targetOp.parameter  at(index).name =  sourceOp.parameter  at(index).name and
                equivalentType (targetOp.parameter  at(index).type, sourceOp.parameter  at(index).type) )
            ))
    ))
    ...
}
```

approach concerns the integration of the generated code with the user code.

Albin-Amiot & Guéhéneuc (2001b) presents two tools (Scriptor and PatternsBox) that help the developers to implement large applications and large frameworks using design patterns. In Scriptor the developers have little or no control on the generated code, once the code is generated, there is no form of locating what design pattern has been applied and where it has been applied. PatternsBox (preservative Generation) tool allows us to instantiate design patterns. The developers need to write most or great part of the code by hand.

Judson et al. (2003) describes an approach to rigorous modeling of pattern-based transformations that involves specializing the UML metamodel to characterize source and target models.

Kim et al. (2003a) describes a metamodeling approach to specify design patterns using roles. They analyze the characteristics of object-based roles and generalize them. Based on the generalized notion of a role, they define a new notion of a model role which is played by a model element. The approach is intended to be easy to use and practical for the development of tools that incorporate patterns into UML models.

Kim et al. (2003b) describes a metamodeling approach that uses a pattern specification language called Role-Based Modeling Language (RBML). A pattern specification defines a family of UML models in terms of roles, where a role is associated with a UML metaclass as its base. RBML uses visual notations based on the UML and textual constraints expressed in OCL to specify patterns properties. The RBML allows specifying various perspectives of design patterns such as static structure, interactions and state-based behavior.

France et al. (2004) presents a technique to specify pattern solutions expressed in the UML. The specifications created by this technique are metamodels that characterize UML design models of pattern solutions. The patterns specification consists of a Structural Pattern Specification (SPS) that specifies the class diagram view of pattern solutions, and a set of Interaction Pattern Specification (IPSs) that specifies interactions in pattern solutions. A UML model conforms to a pattern specification if its class diagram conforms to the SPS and the interactions described by sequence diagrams conform to the IPSs.

Our motivation is to integrate design patterns with MDA. The following advantages between our approach and some existing ones are worth mentioning. A design pattern metamodel allows detecting the presence of a pattern in a model. If there were no metamodels, a library of models specifying each one the ways in that the design pattern can appear should be necessary (this is expensive). Also, it should be necessary to compare the model that is analyzed with the models of the library to see if matching exists. On the other hand, the specification of the metamodels in the three levels allows us to refine pattern model step-by-step. In the context of MDD, model-to-model transformations can achieve a more complete code generation using design patterns.

In a Model Driven Development (MDD) different tools could be used to validate/ verify models at different abstraction levels (PIMs, PSMs, or implementations). In this direction we have formalized UML/OCL metamodels and metamodel-based model transformations. We use a metamodeling notation NEREUS that is independent of any formal language and can be translated to specific ones. A detailed description may be found at Favre (2005b).

6 CONCLUSIONS AND FUTURE WORK

In this paper we analyze a metamodeling technique to reach a high level of reusability and adaptability of MDA-based design pattern components. We propose a megamodel for defining components that integrates PIMs, PSMs, and code models with their respective metamodels. We use our approach to specify standard design patterns.

We are validating the technique through rigorous forward engineering processes based on design patterns that integrate formal specifications and UML/OCL (Favre, 2005a).

A crucial problem is how to detect sub-diagrams which can be matched with a pattern. Metamodeling helps in the identification of design patterns by signature matching and semantic matching. The metamodel establishes what elements should be present in the pattern model and their restrictions. Metamodeling allows us to check models against a set of rules to ensure precise and consistent transformations.

REFERENCES

Albin-Amiot H., Guéhéneuc Y. (2001a). Meta-modeling Design Patterns: application to pattern detection and code synthesis. Proceeding of ECOOP Workshop on Automating Object-Oriented Software Development Methods.
Available: www.yann-gael.gueheneuc.net/Work/Publications/Documents/ECOOP01AOOSDM.doc.pdf

Albin-Amiot H., Guéhéneuc Y. (2001b). Design Pattern Application: Pure-Generative Approach vs. Conservative-Generative Approach. Proceedings of OOPSLA Workshop on Generative Programming, Florida, USA. Available: www.yann-gael.gueheneuc.net/Work/Teaching/Documents

Arnout, K. (2004). From Patterns to Components. Ph. D. Thesis, Swiss Institute of Technology (ETH Zurich).

Bezivin, J., Jouault, F., Valduriez, P. (2004) On the need for Megamodels. In: J. Bettin, G. van Emde Boas, A. Agrawal, M. Volter, & J. Bezivin (Eds.). Proceedings of Best Practices for Model-Driven Software Development (MDSD 2004). OOSPLA 2004 Workshop. Available:www.softmetaware.com/oopsla2004/bezivin-megamodel.pdf

Budinsky, F., Finni, M., Vlissides, J., Yu, P. (1996) Automatic code generation from design patterns. IBM System Journal, Vol 35, N° 2. 151-171.

CASE TOOLS (2005). Available: www.objectbydesign.com/tools

Favre, L. (2005a). Foundations for MDA-based Forward Engineering. Journal of Object Technology (JOT). (ETH) Zurich, Swiss Federal Institute of Technology, ISBN 1660-1769. Vol 4, N° 1, Jan/Feb. 129-153.

Favre, L. (2005b). A Rigorous Framework for Model Driven Development. In: T. Halpin, J. Krogstie and K. Siau (Eds.). Proceedings of CAISE´05 Workshops. EMMSAD ´05 Tenth International Workshop on Exploring Modeling Method in System Analysis and Design Porto, Portugal: FEUP Editions. 505-516.

Florijn, G., Meijers, M., van Winsen, P. (1997). Tool support for object-oriented patterns. Proceeding of ECOOP'97 (European Conference on Object Oriented Programming), Jyväskylä, Finland. 472-795.

France, R., Kim, D., Ghosh, S., Song, E. (2004). A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering. Vol. 30, Nº3, March, 2004. IEEE Computer Society. 193-206.

Gamma, E., Helm, R., Johnson, R.,Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Judson, S., Carver D., France, R. (2003). A metamodeling approach to model transformation. OOPSLA Companion 2003. 326-327.

Kim, D., France, R., Ghosh, S., Song, E. (2003a). A Role-Based Metamodeling Approach to Specifying Desing Patterns. Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03). IEEE Computer Society. 452-457.

Kim, D., France, R., Ghosh, S., Song, E. (2003b). A UML-Based Metamodeling Language to Specifying Design Patterns. Wisme@UML'2003-UMLWorkshop (Workshop in Software Model Engineering). San Francisco, USA. Available: www.cs.colostate.edu/~georg/aspectsPub/WISME03-dkk.pdf

Kleppe, A., Warner, J. & Bast, W. (2003). MDA Explained. The Model Driven Architecture: Practice and Promise. Addison Wesley.

MDA (2005). The Model Driven Architecture. Available: www.omg.org/mda

OCL (2005). OCL 2.0 Specification. Version 2.0. Formal document : ptc/05-06-06. URL: www.omg.org

UML (2005). UML 2.0 Superstructure Specification. OMG Available Specification: formal/05-07-04. Available: www.omg.org.