A Technique to Represent and Generate Components in MDA/PIM for Automation*

Hyun Gi Min and Soo Dong Kim

Department of Computer Science,
Soongsil University,
511 Sangdo-Dong, Dongjak-Ku, Seoul, 156-743, Korea
hgmin@otlab.ssu.ac.kr, sdkim@ssu.ac.kr

Abstract. Component-Based Development (CBD) is an effective approach to develop software effectively and economically through reuse of software components. Model Driven Architecture (MDA) is a new software development paradigm where software is generated by a series of model transformations. By combing essential features of CBD and MDA, both benefits of software reusability and development automation can be achieved in a single framework. In this paper, we propose a UML profile for specifying component-based design in MDA framework. The profile consists of UML extensions, notations, and related instructions to specify elements of CBD in MDA constructs. Once components are specified with our profile at the level of PIM, they can be automatically transformed into PSM and eventually source code implementation.

1 Motivation

MDA is a new software development paradigm where a model plays a key role in automatic software development [1]. It provides a systematic framework to understand, design, operate, and evolve all aspects of enterprise system, using engineering methods and tools. The framework is based on modeling different aspects and levels of abstraction of such systems, exploiting interrelationships between these models.

A very common technique for achieving platform independence is to target a system model for a technology-neutral virtual machine. A model in PIM is reusable over different platforms. Hence, we regard PIM as neither executable unit nor implemented unit. PIM enables models to be traced and improves maintainability through modifying model and regeneration into PSM.

CBD is another promising approach to develop software system effectively and economically through reuse of software components. Especially, domain-common components provide a common set of features and functions in a domain, so that application members can utilize the components by customizing the behavior with minimum effect.

* This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD). (KRF-2004-005-D00172).

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 293 – 307, 2006.

Therefore, if MDA is combined with CBD approach, we can acquire a highly effective development environment where the commonality and variability(C&V) in a domain are modeled and developed as MDA compatible components, and software development can be greatly automated. Moreover, since C&V is reflected in designing PIM and code-level components are not limited to only one platform, the reusability of components is greatly increased.

In this paper, we suggest techniques to combine the advantage of CBD and MDA. We first define a component-based PIM (CB-PIM) and proposed a UML profile for specifying component design in MDA/PIM. The profile consists of UML extensions, notations, and related instructions to specify elements of CBD in MDA constructs. If components are designed by using the proposed method, the design can be automatically transformed into source code implementation, yielding benefits of reusability and automation.

2 Foundation

2.1 Model Driven Architecture (MDA)

MDA is an approach to using models in software development. The essence of MDA is making a distinction between Platform Independent Models (PIMs) and Platform Specific Models (PSMs). To develop an application using MDA, it is necessary to first build a PIM of the application, then transform this using a standardized mapping into a PSM, and finally map the latter into the application code by automation.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns [1]. Some of the motivations of the MDA approach are to reduce the time of adoption of new platforms and middleware, primacy of conceptual design, and interoperability. The MDA approach makes it possible to save the conceptual design and the MDA helps to avoid duplication of effort and other needless waste [2][3].

2.2 UML Profile

A UML profile defines standard UML extensions that combine and/or refine existing UML constructs to create a dialect that can be used to describe artifacts in a design or implementation model. The UML profile defines a set of UML extensions that capture the structure and semantics. It defines several standard extension mechanisms, including stereotypes, constraints, tagged values and icons [4]. When one defines a profile, it is common MDA practice to also define mappings that specify how to transform models conforming to the profile into artifacts appropriate to the kinds of systems. If a model is not specified by a particular UML profile, the model can not be transformed automatically by MDA mechanism.

The OMG has adopted a MOF metamodel of Java and EJB to complement the UML profile for EJB [5], a UML profile for modeling enterprise application integration [6] and a UML profile for CORBA [7] as well. However, these profiles only support implementation levels and do not present component of a PIM level.

2.3 Fontoura's UML-F

UML-F is an UML extension that supports working with object-oriented frameworks and allows the explicit representation of framework variation points [8]. A framework, UML-F assumes, is a collection of several fully or partially implemented components with largely predefined cooperation patterns between them.

This framework implements the software architecture for a family of applications with similar characteristics, which are derived by specialization through application-specific code. UML-F suggests constraint *{appl-class}, {variable}, {extensible}, {static}, {dynamic}, {incomplete}, {for all new methods}* and *{optional}*.

However, elements are not explicitly identified in this model and no precise definition for the elements is suggested. Only the overall meaning of a framework that UML-F reference is explained.

2.4 Exertier's Component Design PIM

Exertier suggests a 'Component Design PIM' that represents a platform-independent solution expressed in terms of software components [9]. The modeling of the distributed components PIM includes four major activities. *Partition the system:* The architecture of a software subsystem identifies a set of architectural elements, here components, which collaborate to achieve the system's functional and non-functional requirements. The objective of this activity is to specify this decomposition. *Perform the component boundary design*: As defined by UML2.0, a component is a modular, deployable and replaceable (pluggable) part of a system. It encapsulates its internal part and exposes a set of interfaces. *Perform the component internal design*: When the boundary of a component has been defined, its internal design can be performed. *Perform the components logical deployment*: Components collaborate to reach functional and non-functional requirements of the subsystem.

This research only suggests four activities for designing component as a PIM. However, it does not deal with how to specify each activity and variability of component for PIM.

2.5 Kim's Variation Types

Kim's work establishes a theoretical foundation on variability in component based development by defining five types of variability and three kinds of variability scope [10]. In this, various variability-related terms are defined such as *Variation Point (VP)*, *Variant*, and *Variability*. Also, five types of variability in CBD are identified; variability on *Attribute*, *Logic*, *Workflow*, *Interface* and *Persistence*. Attribute is defined as an abstract storage to store values, and it is realized as constants, variables, or data structures.

Attribute variability denotes occurrences of *variation points* on attributes. Logic describes an algorithm or a procedural flow of a relatively fine-grained function. *Logic variability* denotes occurrences of *variation points* on the algorithm or logical procedure. *Workflow variability* denotes occurrences of *variation points* on the sequence of method invocations. Persistency is maintained by storing attribute values of a component in a permanent storage so that the state of the component can alive over

system sessions. *Persistency variability* denotes occurrences of *variation points* on the physical schema or representation of the persistent attributes on a secondary storage.

3 Elements of Component Design

In this section, we define elements of a component design and each element is elaborated in details.

A *component* is defined as a set of related *classes*, and it provides a relatively coarse-grained functionality as Fig. 1. All the *classes* in a component are related in some way; association, inheritance, aggregation, composition, and dependency. *Operations* available through the *interface* of components are generally larger-grained than methods in a class. The behavior of these operations is modeled as a *workflow*, which is a sequence of method invocations among the objects/classes in a component.

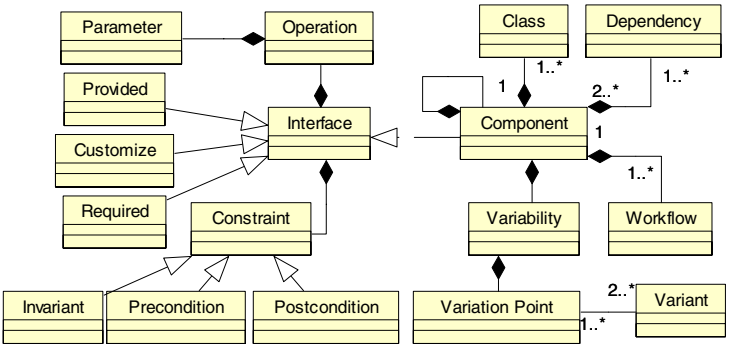

Fig. 1. The metamodel of component

An *interface* has one or more operations, and each operation is given a signature that consists of the operation name, input parameters and a return type. Semantics of each operation should be described to define the behavior and constraints of the operation. It is described by a pre-condition, a post-condition, an invariant, side effects, and constraints. A post-condition describes the state of an object that should be met after an operation finishes execution. A side effect of an operation is any additional changes in the state of related objects besides the main object.

Variabilities are characteristics that may vary from application to application. In general, all variabilities can be described in terms of alternatives. Variability is defined as variation points and variants. Modeling and realizing variability is one of the unique features of CBD. Variability is characterized by a number of variations within the common requirement. It consists of variation points and all their valid variants for variable requirement that is determined to have a minor and detailed difference among some family members by relevant stakeholders.

A variation point identifies one or more locations in a software asset at which the variation will occur [11]. Griss defines variation point as an explicitly designated location within a component at which a variability mechanism may be used to create a

customized component [12]. A Variation Point is a place in software where the minor difference occurs for variable requirement. A Variant is a value or instance that can validly fill in a variation point, i.e. a variant resolves a variation point.

A software quality model is a specification of software quality attributes and their relationship. ISO 9126 is a representative quality model for generic software [13]. A quality attribute is a non-functional characteristic of a component or a system, such as integrability, usability, efficiency, modifiability, reliability, security, transaction, flexibility or availability. Also, deployment of component affects performance, reliability, security, availability, capacity and bandwidth. The component is an executable unit. Therefore, we need not only functionality of component but also extra functional information that supports components deploying and operating.

4 Component Development Process Using UML Profiles

In this section, we propose a component development process using UML profile for specifying components to improve the applicability of PIM of component level as Fig. 2. Analysis process extracts functional and non-functional requirements. The analyzed requirements are represented using UML 2.0 by object oriented design process. This process yields PIMs based on objects.

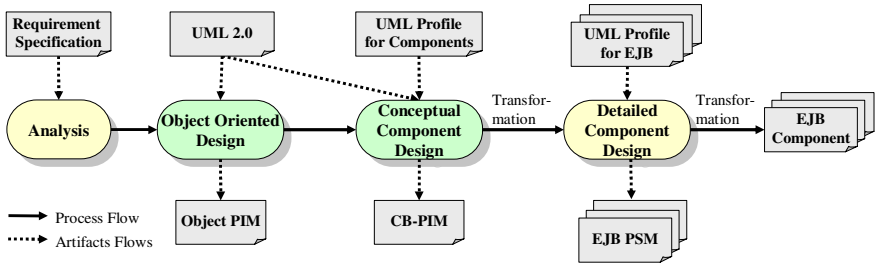

Fig. 2. Component Development Process using UML Profile for Components

In the conceptual component design, the PIMs of object level transform into component-based PIM (CB-PIM) that presents general component information. The general component information that is units, interfaces, variability, and environments of components does not depend on component platforms such as EJB, CORBA, etc. This process identifies the general component information. None of these can be represented by UML 2.0 [19]. Therefore, we need to UML profile for specifying components to present these. The UML profile will be introduced later. Object PIM transforms into CB-PIM that is not dependent on component platform such as EJB and CORBA.

In the detailed component design, the CB-PIM can be automatically transformed into each PSM using the UML profile for component platforms such as UML profile for EJB. Finally, the generated PSMs are transformed into each component source. Therefore, traditional MDA process reuses the object level of PIM. Our MDA process

reuses the component level of CB-PIM. Once components are specified with our profile at the level of PIM, they can be automatically transformed into PSM and eventually source code implementation.

5 UML Profile for Specifying Components

In this section, we suggest a UML profile for specifying components. Our UML profile to represent CB-PIM is based on the UML 2.0. Elements from UML 2.0 and EDOC are used in our profile; the elements for CB-PIM that are not supported by UML 2.0 [14] are extended from MOF. Our UML profile is MOF. Therefore, the CB-PIM that is specified by our profile can be presented by common MDA tools.

5.1 UML Profile for Specifying Component Units

In CBD, a component is the fundamental unit of packaging related objects [12], hence we need to specify the related objects in a component in PIM. A port is a connection point between a classifier and its environment. Connections from the outside world are made to ports according to provided and required [15]. Workflow in a component can be designed by sequence and communication diagrams according to UML 2.0. The UML profile for specifying component units is presented as Table 1.

Table 1. The Elements of UML Profile for Component Units Design

Element	Presentation	Applies to	Remarks
Component	«component»	component	Use UML 2.0
System Component	«SysComponent»	component	
Business Component	«BizComponent»	component	
Transient Class	«Transient»	class	
Persistence Class	«Persistence»	class	Default
Primary Key	«UniqueId»	attribute	
Synchronous Message	«Sync»	method	Default
Asynchronous Message	«Async»	method	
Message Call	«use», «call», etc.	dependency	Use UML 2.0
Relationships	association, inheritance, composition, aggregation, dependency	relationship	Use UML 2.0
Constraints	{ }, pre:, post:, inv:	class, method, relationship, etc.	Use OCL
Algorithm	Use Text	method	Use OCL, ASL

Components are in general classified into *system* components and *business* components [16]. A system component interacts with client programs and manages client transactions by coordinating message flows among participating components and/or objects which mostly manipulate data. System components provide a system service that is the external representation of the system, providing access to the

services of the system. This service acts as a façade and a mediator for the business service [17]. A business component consists of persistent objects which handle persistent business data. Hence, business components execute upon the requests from system components. To denote two types of components in PIM, we use stereotypes; «SysComponent» and «BizComponent».

Persistency objects that should be stored in database or file systems are represented by a stereotype «Persistence». If some objects such as value objects [17] for transforming data are not persistency, a stereotype «Transient» is used. Asynchronous messages use a stereotype «Async» that are described at methods in class, sequence, and communication diagrams. Constraints and algorithms can be expressed by Object Constraints Language (OCL), and Action Semantic Language (ASL).

As Fig. 3 shows, the *LoanMgr* component is denoted as a system component with «SysComponent» stereotype, and composed of one class. The *LoanAccount* component is denoted as a business component with «BizComponent» stereotype, and its two member classes are shown.

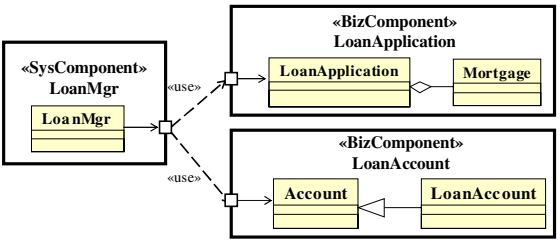

Fig. 3. Example of Expressing Component Units

5.2 UML Profile for Specifying Interfaces

A component provides its component-level interface, i.e. the protocol for accessing the service of the component. In CBD, an interface is clearly separated from component implementation to increase the maintainability and replaceability [12]. Hence, we need to specify some interfaces as well as component units in CB-PIM as Table 2.

Table 2. The Elements of UML Profile for Interface Design

Element	Presentation	Applies to	Remarks
Interface	«Interface»	Interface	Use UML 2.0
Provided Interface	«ProvidedInterface»	Interface	Use UML 2.0
Customize Interface	«CustomizeInterface»	Interface	
Required Interface	«RequiredInterface»,	Interface	Use UML 2.0
Signature	operationName(param:Type): ReturnType	Operation	Use UML 2.0
Constraints	{ }, pre:, post:, inv:	Class, Method, Relationship	OCL
Algorithm	Use Text	Method	OCL, ASL

In CBD, three types of interface can be modeled; *provided*, *customize* and *required* interfaces. The *provided* interface specifies the services provided by a component and it is invoked by other components or client programs at runtime. The stereotype «ProvidedInterface» is used to denote this interface, and the name of the *provided* interface is defined by using 'Ip' prefix name.

Components often provide mechanisms to tailor the behavior of the components through an interface designed especially for this purpose. A *customize* interface consists of methods that are used to assign a *variant* to a *variation point* [18]. To specify *customize* interface, we use a stereotype «CustomizeInterface» and 'Ic' prefix on the name of the *customize* interface.

The *required* interface specifies external services invoked by the current component, i.e. a specification of external services required by the current component [18]. By specifying the *required* interface for a component, we can precisely define the services invoked by the current component. This information can be later used in integrating related components into an application or a component framework. The *required* interface can be specified with a stereotype «RequiredInterface». An interface consists of operation signatures and their semantics. The semantics can be expressed in terms of pre- and post-conditions and invariants using OCL.

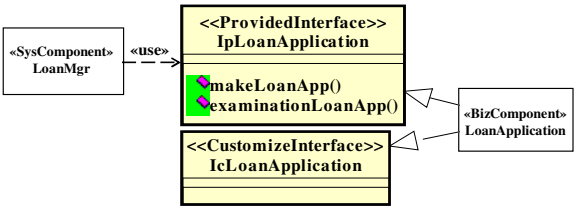

Fig. 4. Example of Expressing Interfaces

Fig. 4 shows an example of expressing interfaces and their realized components in CB-PIM, where a *LoanApplication* component is realized by IpLoanAppication interface and IcLoanApplication interface. The IpLoanApplication interface is a required interface of LoanMgr component. The LoanMgr component requests services of the LoanApplication component. The required interface of the LoanMgr is the IpLoanApplication.

5.3 UML Profile for Specifying Variation

The commonality and variability is made explicit through variation points and variants in the components and other reusable component elements [12]. The goal is to create a set of reusable components that expresses commonality and variability appropriate to the family of applications.

The variability can increase the reusability of component. However, the UML does not support notations of variability. Therefore, variability is designed by non-standard stereotypes, tagged values, or note elements [20]. If the variability is presented by standard notation, MDA tools identify variation points by the

variability. The variation points of PIM or PSM can be filled by other design artifacts and automation tools.

We define types of variation that are attribute variability, logic, workflow, persistency and interface variability as in [10]. To express variation points of a component in CB-PIM, we propose «VP-Attr», «VP-Logic», «VP-WF», «VP-Persistency» and «VP-Interface» stereotypes as in Table 3.

Table 3. UML Profile for Variation Design

Element	Presentation	Applies to	Remarks
Variation Point (VP)	«VP»	Attribute, Method	
Attribute VP	«VP-Attr»	Attribute, Use case	
Logic VP	«VP-Logic»	Method	
Workflow VP	«VP-WF»	Method	
Interface VP	«VP-Interface»	Operation	
Persistency VP	«VP-Persistency»	Operation, Method	
Variant	«Variant»	Class, Operation, Method	
Variation Scope	{vScope = value}	Variation Point	Close, Open
VP ID	{vpID = value}	Variation Point, Variant	
Variant ID	{varID = value}	Variant	
Constraints	{ }, pre:, post:, inv:	Class, Method, Relationship	OCL
Algorithm	Use Text	Class, Method	OCL, ASL

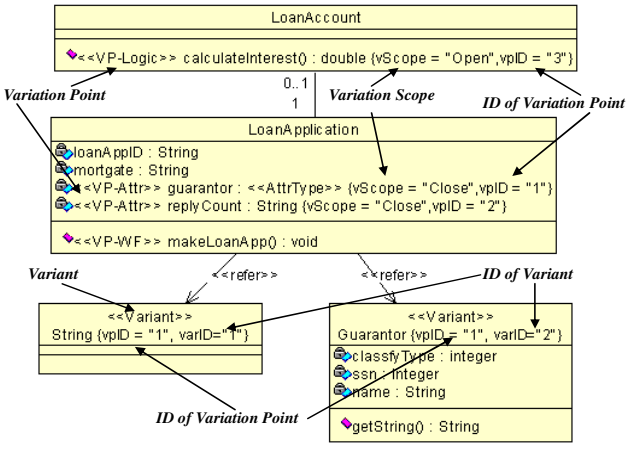

Fig. 5. Example of Expressing Variation

We present two kinds of scope of variation points. *Open* scope of variation point has any number of variants which are already known and additional variants which

are currently unknown but can possibly be found later at customization or deployment time. In constraint, *close* scope of variation point has two or more variants which are already known [10].

Fig. 5. shows an example of expressing variability in CB-PIM. The logic of *calculateIntereste()*can be changed by each family member. The class 'LoanApplication' has two variation points which are guarantor and replyCount. Two variants of the attribute guarantor are a type String and a class Guarantor. The attribute guarantor has variation that has two variants; String and object Guarantor. If the variant string is set as {varID="1"}, the attribute has string data type to store guarantor's ID. If the object Guarantor is set, the data type of the attribute becomes Guarantor. In the implement process, the variation will be implemented by the value of varID later.

5.4 UML Profile for Specifying Extra-Function

A component is an executable unit. We need not only functionality of components but also extra functionality of components that supports components deploying and operating. The extra functional properties extensions are motivated more by the desire to ensure that interface specifications are sufficiently complete to ensure correct integration than by the desire to extend the scope of information hiding to additional properties. Both ends are served by these extensions [21].

To specify extra functional information in CB-PIM more practically, we classify properties into four types; deploy property, runtime property, transaction property and security property as Table 4 . A deploy property captures information for deploy on server. A runtime property specifies runtime environment for component instances. A transaction property defines method of transaction. A security property manipulates strategy about usage of component.

For example, the stereotype «DeployProperty » specifies information for deploy environment. An attribute deployedName as align is called and managed by component middleware server. When the component is running in a server, the mechanism of the component server may use the align name. The components are packaged automatically by the artifactName attribute.

Table 4. UML Profile for Extra Functional Design

Element	Presentation	Applies to	Remarks
Deployment Property	«DeploymentProperty»	Component	Stereo type
	deployedName, artifactName		Tagged Value
Runtime Property	«RuntimeProperty»	Component	Stereo type
	virtualClientsPerInstance, instancePerComponent, instanceTimeToLive componentTimeToLive, instanceInactivityTimeout		Tagged Value
Transaction Property	«TXProperty»	Component, Interface, Class, Method	Stereo type
	useTX, TXAttrType, TXIsolation, TXTimeOut		Tagged Value
Security Property	«SecurityProperty»	Component, Interface, Class, Method	Stereo type
	userRoleName		Tagged Value

Our UML profile represents an activation policy that describes how a client gains access to the component, whether it has exclusive access to the component, and certain lifetime restrictions on the component. The stereotype «RuntimeProperty» specifies information for runtime environment and lifetime restrictions. The activation constraints that can currently be specified in a runtime property are: limits on the number of clients per-instance and per-component, restrictions on the number of instances per-component, limits on the time an instance or a component may exist, including an inactivity timeout, the name by which clients may activate the component and activation operations which allow parameterized activation of the component.

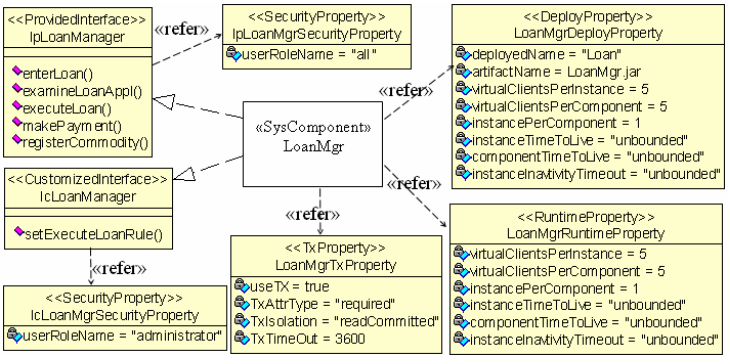

Fig. 6. PIM showing Deploy Property of LoanMgr Component

The stereotype «TxProperty» specifies strategy about transaction of component. If useTX attribute is false, other transaction attributes are ignored by PSM or Code level. The attribute TxAttrType has a TxAttrTypeKind enumeration type. The TxAttrTypeKind enumeration type consists of required, requiredsNew, supports, mandatory, notSupported and never value. The attribute of TxIsolationType has a TxIsolationTypeKind enumeration type. The TxIsolationTypeKind enumeration type consists of readUncommitted, readCommitted, repeatableRead and serializable. The TxTimeOut is a timeout period for transaction operation. If the transaction access time is over TxTimeOut, the transaction should be rolled back.

The stereotype «SecurityProperty» contains strategy about security of component. The attribute of userRoleName is the permitted role name of a component's caller. The role of the component is assigned by this userRoleName attribute. The customize interface may used by an administrator. In this case the userRoleName attribute of the «SecurityProperty» is an 'administrator'. The provided interface may used by all customers. This userRoleName is an 'all'. This attribute may apply to <security-role-ref>, <security-role> and <method-permission> in the deployment descriptor at PSM level for EJB. Fig. 6 shows an example of expressing extra functional property of a LoanMgr system component in CB-PIM.

6 Assessment

The Fountoura's UML-F [8] is based on object framework. Elements are not explicitly identified in this model and no precise definition for the elements is suggested. UML-F reference only explained the overall meaning of a framework. Exertier's research [9] only suggests four activities for designing components with a PIM. This research does not deal with how to specify each activity and the variability of components with a PIM. The UML 2.0 and UML profile for EDOC [22] does not fully present the profiles for specifying general component.

Table 5. Comparing the suggested UML Profile with others (✔: Supported)

Technique / Factor	Comp. Spec.	UML 2.0	EDOC Profile	Our Profile	Remarks
Component Units	✔	✔	✔	✔	«SysComponent», etc.
Provided Interface	✔	✔	✔	✔	«ProvidedInterface»
Required Interface	✔	✔		✔	«RequiredInterface»
Customize Interface	✔			✔	«CustomizeInterface»
Variation Point	✔			✔	«VP-Attr», etc.
Variant	✔			✔	«Variant», etc.
Non Functional Design	✔			✔	«TXProperty», etc.
Workflows	✔	✔	✔	✔	Sequence Diagram, etc.
Reusing Model Level		✔	✔	✔	PIM Level

Our profile covers variability and extra functional designs as well as the four designs of Exeriter's component design PIM such as partition of the system, component boundary design, component internal design, and components logical deployment as in Table 5. Therefore, once components are specified with the suggested UML profile for specifying components at the level of CB-PIM, they can be automatically generated each source code implementation as shown in Fig.7. A CB-PIM can be reused into diverse platforms.

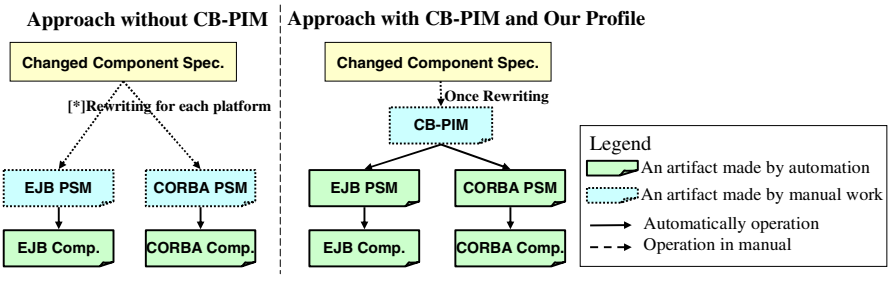

Fig. 7. An advantage of CB-PIM and UML Profile for Specifying Components

If the mechanism for implementing components which is shown in Fig.7 is supported with a tool, various components such as EJB and CORBA can be effectively implemented by using the seamless method and tools. To make our approach more practical and useful, we are developing a prototype development tool based on Eclipse as Fig. 8. The prototype will support all our UML profile and the mechanism.

Eclipse can plug modules such as our component designer and code generator prototypes as in Fig. 8. The component designer based on Graphical Editor Framework (GEF) [23] stores the PIM models to extended UML2 file for our profile. UML2 [24] is an EMF-based implementation of the UML™ 2.0 metamodel for the Eclipse platform. The code generator transfers the UML2 file to codes by using XMI schema. Eclipse basically includes a code editor. Therefore, components can be specified with our UML profile for specifying components at the level of CB-PIM. CB-PIM can be automatically transformed into each PSM and eventually each source code implementation by use the tool.

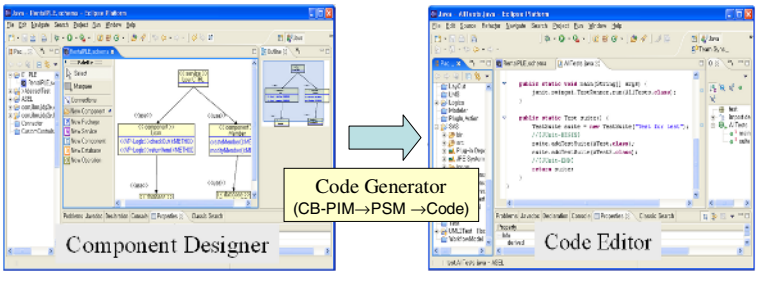

Fig. 8. Component Designer based on Eclipse

7 Conclusion Remarks

CBD is to develop software system effectively and economically through reuse of software components. Effective components should be designed using interfaces, component units, variability, and non-functional factors for components. As a basic reuse unit, components often come in black-box form, only exposing well-defined interface while hiding internal details.

MDA is a n approach to using models in software development. The essence of MDA is making a distinction between PIM and PSM. To develop an application using MDA, it is necessary to first build a PIM of the application, then transform this using a standardized mapping into a PSM, and finally map the latter into the application code by automation.

If a component's middleware is changed but requirement is not modified, the related components should be redesigned and re-implemented because components platforms are diverse. If component specifications are designed at MDA/PIM, we can automatically create the components that are satisfied by the component platform of an application. Therefore, we need the UML profile for specifying components to make machine-understandable design for MDA tools.

In this paper, we proposed a UML profile for specifying component-based design and component development process in MDA framework. Our UML profile consists of UML extensions, notations, and related instructions to specify elements of CBD in MDA constructs. It can be presented by general UML and MDA design tools. Once components are specified in the proposed profile at the level of PIM, they can be automatically transformed into PSM and eventually source code implementation by MDA tools. By using the UML profile for specifying components, we believe that the productivity, reusability, applicability, and maintainability of components can be greatly increased by automation.

References

[1] OMG, "MDA Guide Version 1.0.1," *omg/2003-06-01*, June 2003.
[2] Flater., D., "Impact of Model-Driven Architecture," *In Proceedings of the 35th Hawaii International Conference on System Sciences*, January 2002.
[3] Frankel, D. and Parodi, *The MDA Journal, Model Driven Architecture Straigth from the Masters*, Meghan-Kiffer Press, 2004.
[4] Frankel, D., *Model Driven Architecture™:Applying MDA™ to Enterprise Computing*, Wiley, 2003.
[5] Java Community Process, UML Profile For EJB_Draft, 2001.
[6] OMG, "UML™ Profile and Interchange Models for Enterprise Application Integration(EAI) Specification," 2002.
[7] OMG, "UML Profile for CORBA Specification V1.0, OMG," Nov. 2000.
[8] Fontoura, M., Pree, W., and Rumpe, B., "UML-F: A Modeling Language for Object-Oriented Frameworks," *Lecture Notes in Computer Science Vol. 1850*, 2000.
[9] Exertier, D., Lnaglois, B., and Roux, X., "PIM Definition and Description," *Proceedings of* 1st *European Workshop, Model-Driven Architecture with Emphasis on Industrial Applications(MDA-IA 2004),* 2004.
[10] Kim, S., Her, J., and Chang, S., "A theoretical foundation of variability in component-based development," *Information and Software Technology, Vol. 47*, *p.663-673*, July, 2005.
[11] Muthig, D. and Atkinson, C., "Model-Driven Product Line Architectures," *Lecture Notes in Computer Science Vol. 2379*, pp.110-129, 2002.
[12] Heineman, G. and Councill, W., *Component-Based Software Engineering*, Addison Wesley, 2001.
[13] Kim, S. and Park, J., "C-QM: A Practical Quality Model for Evaluating COTS Components", *IASTED International Conference on Software Engineering (SE'2003),* 2003.
[14] Choi, S., Chang, S., and Kim, S., "A Systematic Methodology for Developing Component Frameworks," *In Proceedings of 7th Fundamental Approaches to Software Engineering (FASE'04) Conference, Lecture Notes in Computer Science Vol. 2984*, 2004.
[15] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual, Second Edition*, Addison-Wesley, 2004.
[16] Cheesman, J. and Daniels, J., *UML Components*, Addison-Wesley, 2001.
[17] Roman, E., *Mastering Enterprise JavaBeans™ and the Java™2 Platform*, Enterprise Edition, WILEY, 1999.
[18] Kim, S., Min, H., and Rhew, S., "Variability Design and Customization Mechanisms for COTS Components," *Lecture Notes in Computer Science Vol. 3480,* pp. 57-66, 2005.

[19] OMG, *Unified Modeling Language: Superstructure version 2.0,* ptc/03-08-02, 2003.
[20] Geyer, L. and Becker, M., "On the Influence of Variabilities on the Application-Engineering Process of a Product Family," " *Lecture Notes in Computer Science Vol. 2379*, 2002.
[21] Bachman, F. and Bass, L., "Volume II:Technical Concepts of Component-Based Software Engineering," *CMU/SEI-2000-TR-008*, May 2000.
[22] OMG, *UML Profile for EDOC V1.0*, 2004.
[23] Eclipse Project, *Graphical Editor Framework (GEF),* at URL: http://www.eclipse.org/gef/
[24] Eclipse Project, *UML2*, at URL: http://www.eclipse.org/uml2/