A Technique to Represent Product Line Core Assets 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. A Product Line (PL) is a set of products (applications) that share common assets in a domain. Product line engineering (PLE) supports the systematic development of a set of similar software systems by common and distinguishing characteristics. Core assets, the common assets, are created and instantiated to make products in PLE. Model Driven Architecture (MDA) emphasizes its feasibility with an automatically developing product. Therefore, we can get the advantages of two paradigms, PLE and MDA, as core assets are represented as PIM in MDA with a predefined automatic mechanism. The PLE framework in the PIM level has to be interpreted by MDA tools. However, we do not have a standard UML profile for representing core assets. The research representing the PLE framework is not enough to automatically make core assets and products. We represent core assets in the PIM level in terms of architecture, components, and decision models. Core assets are specified with our profile at the level of PIM, where they can be automatically transformed and instantiated. The method of representing the framework with PLE and MDA is used to improve productivity, applicability, maintainability and quality of products.

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 an enterprise system, using engineering methods and tools. The goals of MDA are portability, interoperability, and reusability by the architectural separation of concerns. 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.

A product line is a set of products (applications) that share common assets in a domain. Product Line Engineering is a set of principles, techniques, mechanisms, and processes that enables the realization of produce lines [2]. Core assets, the common assets, are created and instantiated to make products in PLE. The concepts in the

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

application domain are analyzed and used to build a product line architecture, which includes reference architecture for the systems in the domain. Applications can then be constructed largely by instantiating this product line architecture.

Therefore, if main constructs and mechanisms of the MDA are combined with the core mechanisms and characteristics of PLE, we can expect to have an effective software development approach. PLE supports this by reusing common assets derived through core asset engineering, and MDA supports this by generating applications on diverse platforms through a model transformation. However, previous researches about representing assets are not enough to automatically make core assets and products.

In this paper, we suggest techniques to improve the advantage of PLE and MDA. We define the elements of core assets and PIM. We compare the needing of core assets and supporting PIM to present. We suggest a UML profile for PLE to present the gaps that can not be presented by a general PIM. Product line architecture, components, decision models, and resolution models are designed by our proposed method in MDA. The design can be automatically transformed to a source code implementation by using MDA transformation mappings. Eventually application engineering of PLE can be automated using MDA tools. We also believe that the productivity, applicability, maintainability, reusability, and quality of an application can be greatly increased.

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 goals of MDA are portability, interoperability and reusability through architectural separation of concerns [1]. Some of the motivations of the MDA approach are: 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 helps to avoid duplication of effort and other needless waste [3][4].

2.2 Product Line Engineering (PLE)

PLE supports the systematic development of a set of similar software systems by understanding and controlling their common and distinguishing characteristics. Thus it is an approach for software reuse driven by the concepts from the real-world domain of software products, which are used to tackle main reuse challenges. The concepts in the application domain are analyzed and used to build a product line infrastructure, which includes reference architecture for the systems in the domain.

Concrete applications can then be constructed largely by instantiating and reusing this product line infrastructure [5].

2.3 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. It defines a set of UML extensions that define several standard extension mechanisms, including stereotypes, constraints, tagged values and icons [6]. 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 specific kinds of systems. If a model is not specified by a particular UML profile, the model can not be transformed automatically by the MDA mechanism.

The OMG has adopted a MOF metamodel of Java and EJB to complement the UML profile for EJB [7], a UML profile for modeling enterprise application integration [8] and a UML profile for CORBA [9] as well. However, there support implementation levels. The profiles do not present the component of a PIM level.

2.4 Gomaa's PLUS

Product Line UML-Based Software Engineering (PLUS) method extends the UML-based modeling methods that are used for single systems to address software product lines [10]. With PLUS, the objective is to model the commonality and variability in a software product line. Gomaa suggest stereotypes «kernel», «optional», «alternative», and «variant» to be used in UML diagrams. However, the elements are not explicitly identified in this model and no precise definition for the elements is suggested. These stereotypes about representing PLE framework are not enough to automatically make core assets and products.

2.5 Muthig's PLE Metamodel

The metamodel for the product line assets describes an information model for the assets capturing the explicit and integrated product line information [5]. The metamodel for product line assets consists of three packages. The *Asset* package contains the metamodel for general assets including assets used for single-system development.

The *GenericAsset* package depends on the *Asset* package and defines genericity as an add-on characteristic for any asset. The main concepts in the *GenericAsset* package are variant asset elements, which are variation points concerned with variant information at local points in an asset.

The *DecisionModel* package builds on generic assets and extends the variation point concept to decisions. Decisions are variation points that typically constrain other variation points and provide a question that must be answered during application engineering. These questions and the constraints among them support application engineering by guiding the instantiation of generic assets. The constraint network of decisions is the decision model.

3 Elements of Core Assets

In this section, we define elements of core assets and each element is elaborated in detail. A core asset plays a key role in PLE. It consists of product line architecture (PLA) which is generic to products, a component model capturing components and interfaces, and a decision model defining variability realization, as shown in Fig. 1. PLA and component model can be represented to architecture specifications, component specifications, and interface specifications. Like the C&V (Commonality and Variability) model, the variability should especially be specified to decision description.

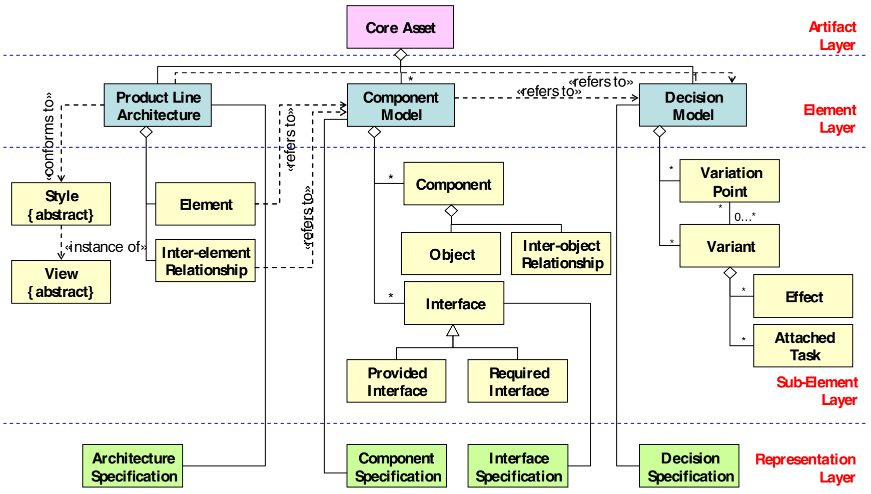


Fig. 1. Meta Model for Core Asset

Architecture specification effectively designs architecture. It is important that architecture specification include style, view, element and inter-element relationships. It describes the composition of software components and their functionalities in a conceptual level, while the component specification specifies objects and inter-object relationships.

Component specification specifies software components and their relationships with text or graphical notations such as UML diagrams [11]. The models can be represented by use case diagrams, class diagrams, and sequence diagrams in UML. Expression of Variability on the models has generally been proposed by using stereotypes. Interface specification specifies public interface through which two components communicate each other, and which consists of method signatures and semantics. In this specification, two types of the interfaces may exist, provided interface and required interface [11].

Decision specification is a realization of the variability of the C&V specification, and it consists of variation points, their associated variants, effects, and attached tasks. Several variants exist in one variation point, and each variant may have several effects and attached tasks.

4 Gap Analysis Between Core Asset and PIM Artifacts

MDA specification [12] and MDA guide [13] suggest the concept of PIMs. The PIM provides formal specifications of the structure and function of the system in a platform-independent manner using UML, MOF, and so on. We define the key elements of PIM in Fig. 2 to compare needing of core assets and supporting PIM to present.

The gaps that are differentiated between core asset artifacts and PIM are covered by UML profile for specifying PLE in section 5. If the gaps are covered by PIM with UML profile for PLE, the core asset contents can be designed by PIMs. Applications

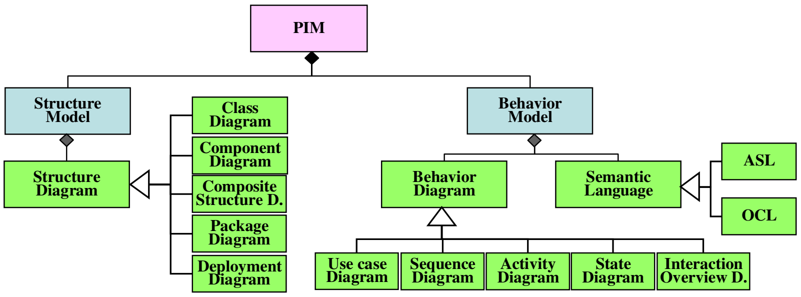


Fig. 2. Metamodel of PIM

Table 1. Gaps between Core Assets and PIM Artifacts (○: Support, △: Partially Support, –: Not supported)

Needed by Core Assets			Supporting by PIM (UML and ASL)	Gap
Element	Sub Element			
PLA Model	Module		Class, Component, Composite Structure, Package Diagram	△
	Relationship		Dependency, Association, Generalization, Realization, etc.	○
Compo-nent Model	Compo-nent	Intra-Object	Class, Composite Structure Diagram	○
		Relationship	Relationship	○
	Interface		Provided and Required Interface	○
	Functional View		Use case and activity Diagram Action Semantic Language	△
	Static View		Class, Component Diagram	○
	Dynamic View		Sequence Diagram Communication Diagram Interaction Overview Diagram	○
Decision Model	Variation Point		–	–
	Variant		–	–
	Effect		–	–
	Task		–	–

can be generated by the PIM using automation. We identify the gaps of PLA, components, and decision model as Table 1.

Product line architecture must provide a link between the standard organization, or communications protocols, and the individual components to permit clients to understand easily how the components will fit into their respective architecture [2]. The product line architecture model that consists of module and relationship between modules needed by core assets is covered by PIM, such as UML and ASL. However, the detailed types for the module are not specified by them.

The component model represents a design of components themselves which are represented with structural and behavioral models of objects, inter-object relationships, and interfaces. The component model of core assets is nearly supported by class, composite structure, and a component diagram.

The decision model is a specification of variations in core assets and includes variation points, variants, effects, and the attached task. Core assets are instantiated for an application by the decision model and resolution model. However, the elements of the decision model are not supported by PIM.

5 Method to Present the Core Asset in PIM

In this section, we specify methods to represent the core asset in PIM of MDA. The PIMs that are specified by our methods can be transformed into PSM and code sources using the automation of MDA. The method to present core assets consists of product line architecture, components, variations, and decision models. The method is showed by examples about core assets for rental application.

5.1 Method to Present Product Line Architecture

Software architecture realizes both functional and non-functional requirements. With the requirements and product line analysis model derived from the requirements, several kinds of views are proposed and used such as module view, C&C (Component and Connector) view and deployment view [14]. Several *styles* of a view can be applied to PLA. Architecture design begins with choosing the most appropriate architectural styles which can realize both types of requirements.

Element and *inter-element relationships* in PLA are directly derived from the requirements and especially inter-element relationships are guided by styles. Therefore, it is fair to state that elements and the relationships effectively implement functional and non-functional requirements. Hence the style is not a constituent of product line architecture, but an abstract element to which the architecture conforms.

The UML profile for specifying PLA can be described as shown in Table 2. The profile can specify module views and C&C views [14]. Conceptual components have conceptual relationships between each other. Relationships are abstracted interfaces between units. The styles of the C&C viewtype consist of the pipe-filter, shared-data, publish-subscribe, client-server, peer to peer, and communicating-processes styles.

Types of messages are control, data, and uses from QADA which are methods of standing for Quality-Driven Architecture Design Analysis method [15]. Passing *control* means, that one component controls a given aspect of the system. Passing *data* means, that one component inputs data to another component, but is still able to continue processing without waiting for a reply or return value. The situation is reversed when talking about *uses* relationships, where a component has a kind of subcontract with the component it uses [15]. Fig. 3 is an example of PIM for

Table 2. Elements of UML Profile for Specifying PLA

Element	Presentation	Applies to	Remarks
Unit	«module»	Class, Component, Package	
Connector	«connector»	Class, relationship	
Use Dependency	«use»	Relationship	Use UML2.0
Control Relationship	«control»	Relationship	
Data Relationship	«data»	Relationship	
Layer	«layer»	Package and class	
Filter	«filter»	Class, Component	
Pipe	«pipe»	Relationship	
Repository	«repository»	Class, Component	
Publish	«publish»	Relationship	
Subscribe	«subscribe»	Relationship	
Client	«client»	Class, Component	
Server	«server»	Class, Component	

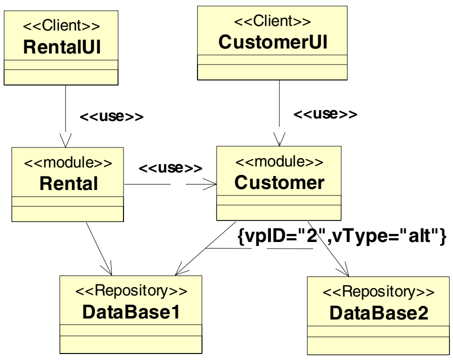


Fig. 3. Example of PLA PIM Design for Rental Application

specifying PLA. This unit and its relationships can have variation and can be instantiated.

5.2 Method to Present Components

The component model represents component designs themselves which are represented with structural and behavioral models of objects, inter-object relationships, and interfaces as Table 3. Generally, component-base development (CBD) is based on object-oriented development (OOD) and in this section we assume that functionality of the core asset is realized by its components. The component model is based on PLA.

Table 3. Elements of UML Profile for Specifying Components

Element	Presentation	Applies to	Remarks
Component	«component»	Component, Package	Use UML 2.0
Transient Class	«Transient»	Class	
Persistence Class	«Persistence»	Class	Default
Primary key filed	«UniqueId»	Attribute	
Synchronous Message	«Sync»	Operation	Default
Asynchronous Message	«Async»	Operation	
Relationships Between Components	Dependency, Association, Generalization, Realization, etc.	Relationship	Use UML 2.0
Interface	«Interface»	Interface	Use UML 2.0
Provided Interface	«ProvidedInterface»	Interface	Use UML 2.0
Required Interface	«RequiredInterface»,	Interface	Use UML 2.0
Signature	name (param : Type): return type	Operation	Use UML 2.0
Constraints	{ }, pre:, post:, inv:	Class, Method, Relationship, etc.	OCL
Algorithms	Use Text	Method	OCL, ASL

The components are identified by architecture models. The components are nearly described by UML 2.0. The structural model of the components can be represented by the composite structure diagram of UML 2.0. The behavior can be represented by sequence, communication, activity, state machine diagrams, and Action Semantic Language (ASL).

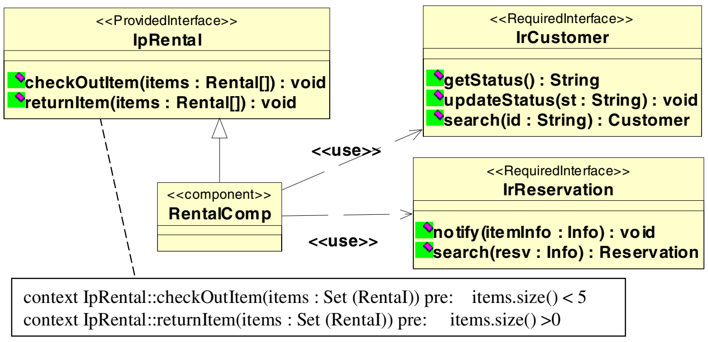
Components are the fundamental units of packaging related objects [16], hence we need to specify the related objects in a component in core asset PIM. A port is a connection point between a classifier and its environment. Connections from the outside world are made to ports according to what is provided and required.

Persistency objects that should be stored in the database or file systems are represented by a stereotype «Persistence». If some objects such as value objects [17] for transforming data are not persistent, a stereotype «Transient» is used. Asynchronous messages use the 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 ASL.

A component provides its component-level interface, i.e. the protocol for accessing the service of the component. An interface is clearly separated from the component implementation to increase the maintainability and replaceability [16]. Hence, we need to specify some interfaces as well as component units.

Two types of interfaces can be modeled; *provided* 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 *provided* interface is defined by using 'Ip' prefix name. The *required* interface specifies external services invoked by the current component, i.e. a specification of external services required by the current component [10]. By specifying the *required* interface for a component, we can precisely define the services invoked by the current component.

This information can later be used in integrating related components into core assets. The *required* interface can be specified with a stereotype «RequiredInterface». An interface consists of operation signatures and their semantics. The semantics can



<<ProvidedInterface>>
IpRental
checkOutItem(items : Rental[]) : void
returnItem(items : Rental[]) : void
<<RequiredInterface>>
IrCustomer
getStatus() : String
updateStatus(st : String) : void
search(id : String) : Customer
<<use>>
<<component>>
RentalComp
<<RequiredInterface>>
IrReservation
notify(itemInfo : Info) : void
search(resv : Info) : Reservation
<<use>>
context IpRental::checkOutItem(items : Set (RentaI)) pre: items.size() < 5
context IpRental::returnItem(items : Set (RentaI)) pre: items.size() >0

Fig. 4. Example of Components PIM Design for Rental Application

Table 4. Elements of Action Semantic Language

Semantics	Usage
Creating Object	**create object instance** <object reference> **of** <class>;
Deleting Object	**delete object instance** <object reference>;
Searching Object	**select [one|any|many]** <object reference set> **from instances of** <class> **where** <where clause>;
Writing Attribute	<object reference>.<attribute name> = <expression>;
Reading Attribute	…<object reference>.<attribute name>;
Sending Message	**generate** signal action to <class>

be expressed in terms of pre- and post-conditions and invariants using OCL. Fig. 4 is an example of component PIM design of a component RentalComp.

Behavior can be represented by behavior diagrams, ASL such as Executable UML [18], and OCL. The action semantic is described in Table 4. The behavior can generate source code using automation of MDA because the syntax can be read by MDA tools.

Fig. 5 is an algorithm of returnItem( ). If a customer should pay a penalty fee, the penalty fee is deducted from his or her mileage. If customers return items early, points are added to his or her mileage.

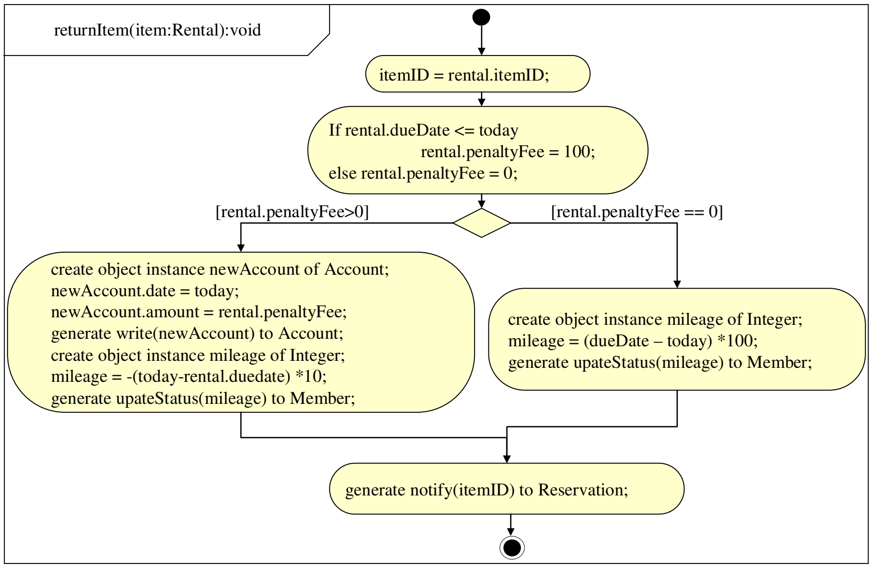

returnItem(item:Rental):void
itemID = rental.itemID;
If rental.dueDate <= today
rental.penaltyFee = 100;
else rental.penaltyFee = 0;
[rental.penaltyFee>0]
[rental.penaltyFee == 0]
create object instance newAccount of Account;
newAccount.date = today;
newAccount.amount = rental.penaltyFee;
generate write(newAccount) to Account;
create object instance mileage of Integer;
mileage = -(today-rental.duedate) *10;
generate upateStatus(mileage) to Member;
create object instance mileage of Integer;
mileage = (dueDate – today) *100;
generate upateStatus(mileage) to Member;
generate notify(itemID) to Reservation;

Fig. 5. Example of Behavior Modeling using ASL for Rental Application

5.3 Method to Present Variation and Decision Model

The instantiation of components in core assets of PLE differs from the customization of components in CBD. Binary components include customization mechanisms to set variation [19]. The variation is chosen by the component consumer after the transition. The core assets are instantiated by the resolution of the decision model in the application engineering before the transition. The unnecessary elements are removed or deselected for target application by the decision models. The decision model is a specification of variations in core assets and includes variation points, variants, effects, and the attached task [20].

A variation point is a place where slight differences among members may occurr. The variation point may be exposed in the architectural style, architectural elements and relationships, and component internals. *Variants* are valid values which can appropriately fill in a variation point. As the types of variation points and variability types such as optional and alternative, the variants may be designed into various formats.

The *effect* means a range of relationships among variations points, and is represented with dependencies and constraints. For example, some variations should be selected with some variations, but some variations are not. The relationships are an essential problem which should be specified and resolved in product line engineering [21]. Therefore, for one variant of a variation point, the effect can be represented as the post-conditions of setting the variant. The *Attached Task* is a set of activities to resolve a variation point for one selected variant, that is, to instantiate. Through the attached tasks, post-conditions of the instantiated variation point should satisfy defined effects for the variation point.

Table 5. Elements of UML Profile for Specifying Components

Element	Presentation	Applies to	Remarks
Variation Point	«VP»	Attribute, Method	Default
Attribute VP	«VP-A»	Attribute in Class	
Logic VP	«VP-L»	Method in Class, Sequence, etc.	
Workflow VP	«VP-W»	Method in Class, Sequence, etc.	
Interface VP	«VP-I»	Operation in Class Diagram	
Persistency VP	«VP-P»	Operation in Class Diagram	
Variation Scope	{vScope = value}	Variation Point	Close, Open, Unknown
ID of VP	{vpID = value }	Variation Point	Unique ID
Type of VP	{vpType = value }	Variation Point	opt, alt
Constraints	{ }, pre:, post:, inv:	Class, Method, Relationship, etc.	OCL
Algorithms	Use Text	Method	OCL, ASL

We define types of variation as attribute variability, logic, workflow, persistency and interface variability [22]. To express variation points of core assets, we propose stereotypes that are «VP-A», «VP-L», «VP-W», «VP-P» and «VP-I». The *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.

We present two kinds of variation point scopes. The o*pen* scope 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 the customization or deployment time. In constraint, the *close* scope variation point has two or more variants which are already known. The types of variation points consist of *opt* and *alt*. The *opt* is an optional selection of variants at the variation point. The variation cannot be used. The a*lt* is one of which could be selected at the variation point. Stereotype variation points have a tagged value about variation scope, ID of a variation point, and Type of a variation point.

The decision model for the returnItem ( ) in Fig. 6 is shown as Table 6. Decision models may be described using the table form in the tool. It is easily read by people. However, the table form cannot be read by tools for instantiation. Therefore, if the decision models are saved as XML, the file is easily read by tools.

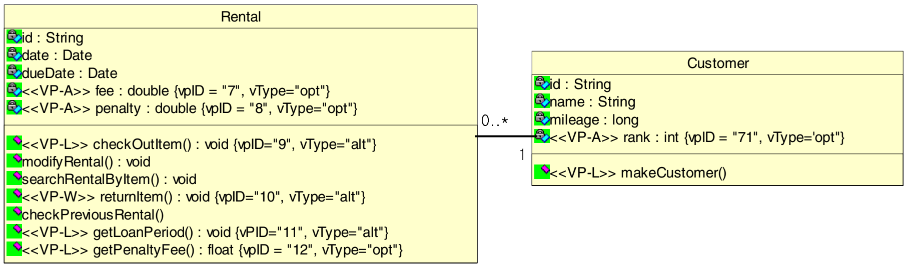

Rental
id : String
date : Date
dueDate : Date
<<VP-A>> fee : double {vpID = "7", vType="opt"}
<<VP-A>> penalty : double {vpID = "8", vType="opt"}
<<VP-L>> checkOutItem() : void {vpID="9", vType="alt"}
modifyRental() : void
searchRentalByItem() : void
<<VP-W>> returnItem() : void {vpID="10", vType="alt"}
checkPreviousRental()
<<VP-L>> getLoanPeriod() : void {vPID="11", vType="alt"}
<<VP-L>> getPenaltyFee() : float {vpID = "12", vType="opt"}
0..*
1
Customer
id : String
name : String
mileage : long
<<VP-A>> rank : int {vpID = "71", vType='opt"}
<<VP-L>> makeCustomer()

Fig. 6. Example of Variation Design for Rental Application

Table 6. Example of Decision Model of returnItem()

VP ID	Variant ID	Effect	Task

10	1	While returning items, points are assigned by each customer rank.	{if vpID==71 then varID == 1} //customer object should have an attribute rank
			generator notifyReservation() to Reservation;
			Point = 10 * customer.rank;
			generator IncPoint() to Member;

	2	While returning items, 100 points are assigned to each customer.	Point = 100;
			generator IncPoint() to Member;
			generator notifyReservation() to Reservation;

Decision specification can be represented to UML extensions and XML as Fig. 7. While variability occurs in components of a component-based development, it can occur in the component model or PLA of PLE. A variability listed in the decision specification is eventually reflected and realized in the architecture specification, component specification and interface specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<DecisionModel xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
          xsi:noNamespaceSchemaLocation=". \DecisionModel.xsd">
    <VariationPoint ID="10">
     <Variant ID="1">
          <Effect> While returning items, points are assigned by each customer rank.</Effect>
          <AttachedTask>{if vpID==71 then varID == 1}</AttachedTask>
          <AttachedTask>generator notifyReservation() to Reservation;</AttachedTask>
          <AttachedTask>Point = 10 * customer.rank;</AttachedTask>
          <AttachedTask>generator IncPoint() to Member;</AttachedTask>
     </Variant>
     <Variant ID="2">
          <Effect> While returning items, 100 points are assigned to each customer.</Effect>
          <AttachedTask>Point = 100;</AttachedTask>
          <AttachedTask>generator IncPoint() to Member;</AttachedTask>
          <AttachedTask>generator notifyReservation() to Reservation;</AttachedTask>
     </Variant>
    </ VariationPoint>
</DecisionModel>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="DecisionModel">
     <xs:annotation>
      <xs:documentation>Comment describing your root element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
       <xs:element name="VariationPoint" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Variant" minOccurs="0" maxOccurs="unbounded">
           <xs:complexType>
            <xs:sequence>
              <xs:element name="Effect" minOccurs="0"/>
              <xs:element name="AttachedTask" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="ID" type="xs:string " use="required"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
         <xs:attribute name="ID" type="xs:string" use="required"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
  </xs:schema>
```

Fig. 7. Decision Model and XMI Schema for Decision Model

6 Assessment

The UML 2.0 and Gomaa's PLUS do not sufficiently detail the profile for specifying architecture, components, and decision models. These stereotypes about the representing PLE framework are not enough to automatically make core assets and products. We suggest a UML profile that supports them. The core asset PIM can be present by our UML profile. The core asset PIM can be instantiated by MDA mapping rules for instantiation.

As shown in Fig. 8, the MDA transformation mechanism can be used to map a core asset to an instantiated core asset if the decision models and application specific decisions including variants are expressed in XMI. To automate the instantiation process, mapping rules that map elements of the core asset to the elements of the instantiated core asset are required, as in Fig. 8.

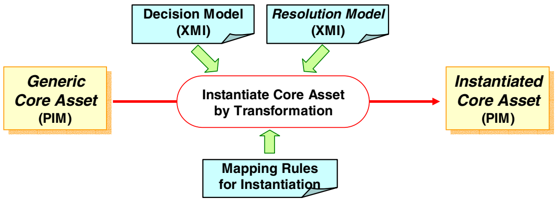

Decision Model (XMI)
Resolution Model (XMI)
Generic Core Asset (PIM)
Instantiate Core Asset by Transformation
Instantiated Core Asset (PIM)
Mapping Rules for Instantiation

Fig. 8. Instantiation by MDA Transformation

7 Conclusion Remarks

Both PLE and MDA are emerging as effective paradigms for building a family of applications in a cost effective way. PLE supports this by reusing common assets derived through core asset engineering, and MDA supports this by generating applications on diverse platforms through model transformation. However, previous researches about representing the PLE framework are not enough to automatically make core assets and products.

In this paper, we propose the UML profile for specifying PLE. Our UML profile consists of UML extensions, notations, and related instructions to specify elements of PLE in MDA PIM models, which can be presented by general UML and MDA design tools. We introduce a concept of integrating MDA and PLE in software engineering. Once core assets are specified with our profile at the level of PIM, they can be automatically transformed and instantiated. Eventually, the application engineering of PLE is automated by MDA tools. We also believe that the productivity, applicability, maintainability, reusability, and quality of the application can be greatly increased.

References

[1] OMG, "MDA Guide Version 1.0.1," *omg/2003-06-01*, June 2003.
[2] Clements P, Northrop L, Software Product Lines, Addison Wesley, 2002.
[3] Flater., D., "Impact of Model-Driven Architecture," *In Proceedings of the 35th Hawaii International Conference on System Sciences*, January 2002.
[4] Frankel, D. and Parodi, *The MDA Journal, Model Driven Architecture Straigth from the Masters*, Meghan-Kiffer Press, 2004.
[5] Muthig, D. and Atkinson, C., "Model-Driven Product Line Architectures," *SPLC2 2002, LNCS Vol. 2379*, pp. 110–129, 2002.
[6] Frankel, D., *Model Driven Architecture™:Applying MDA™ to Enterprise Computing*, Wiley, 2003.
[7] Java Community Process , "UML Profile For EJB_Draft," 2001.

[8] OMG, "UML™ Profile and Interchange Models for Enterprise Application Integration(EAI) Specification," 2002.

[9] OMG, "UML Profile for CORBA Specification V1.0, OMG," Nov. 2000.

[10] Gomaa, H., Designing Software Product Lines with UML from Use Cases to Pattern-based Software Architectures, Addison-Wesley, 2004.

[11] Rumbaugh, J., Jacobson, I, and Booch, G., The Unified Modeling Language Reference Manual Second Edition, Addison-Wesley, 2004.

[12] Object Management Group, Model Driven Architecture (MDA), July 2001.

[13] OMG, MDA Guide Version 1.0.1, omg/2003-06-01, June 2003.

[14] Clements, P., et al., Documenting Software Architectures Views and Beyond, 2003.

[15] Matinlassi, M., Niemela, E., and Dobrica, L., "Quality-driven architecture design and quality analysis method: A revolutionary initiation approach to a product line architecture," *VTT publication 456, VTT Technical Research Center of Finland, ESPOO2002,* 2002.

[16] Heineman, G. and Councill, W., *Component-Based Software Engineering*, Addison Wesley, 2001.

[17] Roman, E., *Mastering Enterprise JavaBeans™ and the Java™2 Platform*, Enterprise Edition, WILEY, 1999.

[18] Mellor, S. and Balcer, M., Executable UML: A Foundation for Model-Driven Architecture, Addison Wesley, 2002.

[19] Kim, S., Min, H., and Rhew, S., "Variability Design and Customization Mechanisms for COTS Components," *Proceedings of The 2005 International Conference on Computational Science and its Applications (ICCSA 2005), LNCS Vol. 3480,* pp. 57–66, 2005.

[20] Kim, S., Chang, S., and Chang, C., "A Systematic Method to Instantiate Core Assets in Product Line Engineering," *Proceedings of Asian-Pacific Software Engineering Conference 2004*, Nov. 2004.

[21] Sinnema, M., Deelstra, S., Nijhuis, J., and Bosch, J., "COVAMOF: A Framework for Modeling Variability in Software Product Families," *Proceedings of the Third Software Product Line Conference (SPLC 2004), LNCS Vol. 3154*, August 2004.

[22] Kim, S., Her, J., and Chang, S., "A Theoretical Foundation of Variability in Component-based Development," *Information and Software Technology*, Vol. 47, pp. 663–673, July 2005.