MODEL DRIVEN ARCHITECTURE

Dr. Thomas Koch, mailto:koch@io-software.com
Axel Uhl, mailto:uhl@io-software.com
Dirk Weise, mailto:weise@io-software.com

November 28, 2001

Interactive Objects Software GmbH
http://www.io-software.com

INTRODUCTION

VOCABULARY

Introduction

This document is divided into five parts:

1. "Overview" describes the basic MDA notions and their relationships in UML.
2. "Vocabulary" gives definitions and explanations of MDA-related terms.
3. "Example" applies MDA principles to a sample business model and its mappings onto EJB and CORBA.
4. "An MDA Software Development Process Outline" proposes a software development process leveraging MDA.

Overview

Introduction and Goals

The OMG has defined the scope of Model-Driven Architecture (MDA) in its document ormsc/2001-07-01 (see http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01). An introductory whitepaper can also be found at http://cgi.omg.org/docs/omg/00-11-05.pdf. MDA provides a framework for software development that uses models to describe the systems to be built. The system descriptions that the models provide can be given at various levels of abstraction, each emphasizing certain aspects or viewpoints of the system.

A software system will eventually be deployed to one or more platforms, either used alternatively, or in conjunction. Platforms are subject to change over time. Most of all, they change at different, typically higher, rates than the "higher-level" models of the system that tend to grow increasingly independent of the target platforms.

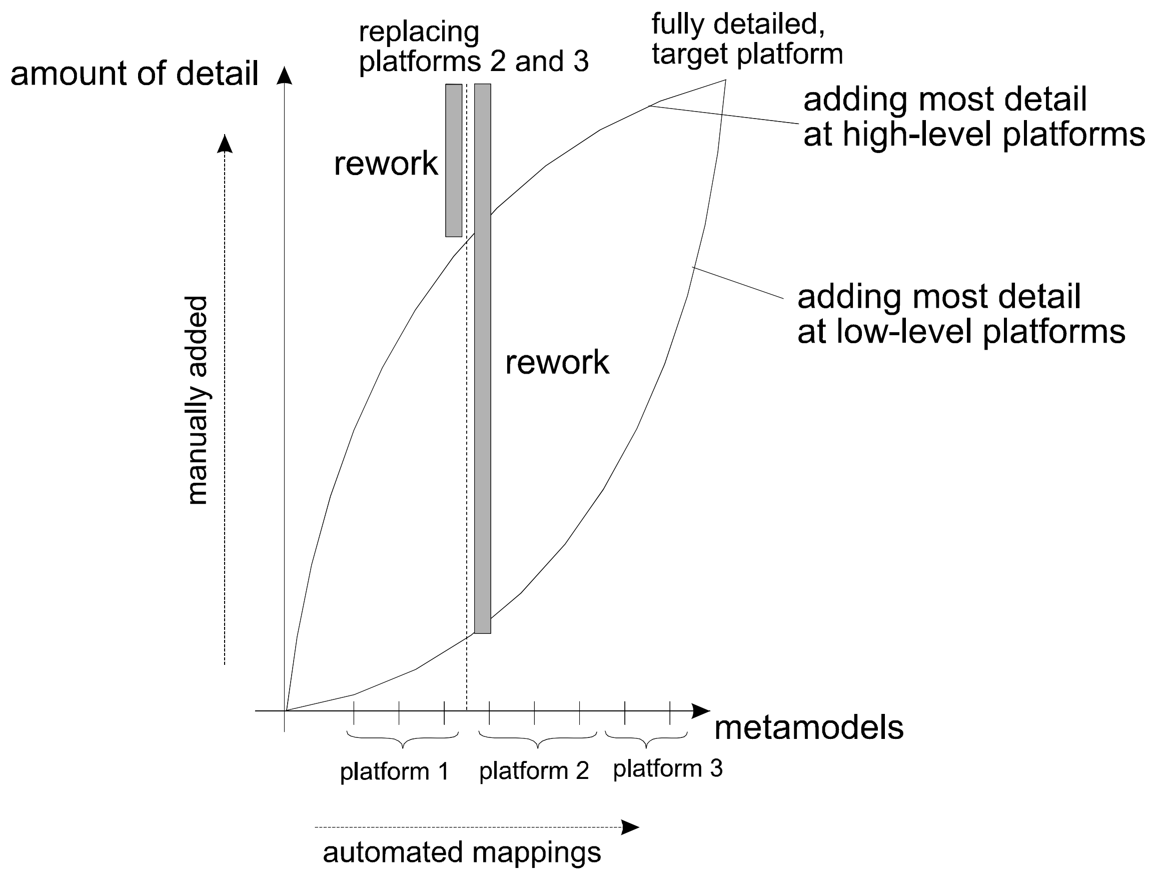


Figure 1 A walkthrough for an MDA instance

The level of abstraction at which a certain amount of system detail is expressed typically determines the ratio between the effort and the amount of detail that gets added. Best practices and accepted defaults allow for the definition of efficient and effective mappings to less abstract, i.e. more detailed levels.

For example, an assignment in C++ can be as easy as writing "x=5". Today we let compilers do the job of transforming this into assembly code that does something equivalent. In doing so, we rely on the quality of these transformations and trade the loss of flexibility against the gain of effectiveness in expressing system specifications. Only in rare cases do we still feel obliged to take control of the register assignment in order to make a loop perform that 1 percent better than the code the compiler would have created.

This pattern of relying on best practices continues to reach higher levels of abstraction. In addition to compilers, we now use tools that allow us, for example, to draw an association relationship between two business components. The tool will map such a simple line to literally thousands of lines of corresponding Java or C++ code and other artifacts such as configuration files, deployment descriptors etc.. These take care of appropriate persistence and transaction handling for the association, distribution, or delete propagation, to name a few.

As we come to accept available best practices and proven defaults for model transformations, it becomes much easier to express certain elements of a system in a model, because we can do so at a higher level of abstraction and let a tool do the transformation into a more detailed specification.

Figure 1 shows two different approaches to developing a system using MDA. The lower curve indicates that only little detail about the system is specified in more abstract, platform-independent metamodels, whereas most of the detail gets added on platform-specific levels. Compared to the upper curve, this has two serious drawbacks:

- The initial system creation requires more effort. This is because, as explained above, adding detail at a lower level of abstraction typically requires more effort than doing so at a higher level of abstraction and relying on an automated mapping.
- When the target platforms change, much more rework is required for porting the system to the modified platforms.

Models and Metamodels

According to MOF, models are instances of metamodels. A metamodel may make it possible to describe properties of a particular platform. In this case, the models that are instances of such a metamodel are said to be *platform-specific* with regard to this platform. Models that describe a system at a level of abstraction that is sufficient to use all their contents for implementing the system on different platforms are referred to as *platform-independent* with regard to these target platforms.

These relations are displayed in Figure 2.

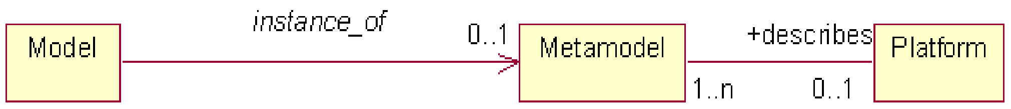


Figure 2 Overview

Mapping Between Models

Models and, by implication, their metamodels, may have a semantical relation to one another, for example if they describe the same system for the same platform at different levels of abstraction.

It is of course desirable to have mappings between those different but related models performed automatically. This makes it possible to express each aspect of a system at an appropriate level of abstraction while keeping the various models in sync.

A mapping between models is assumed to take one or more models as its input and produce one output model. Again, each of these models is an instance of a metamodel.

The rules for the transformation that is performed by the mapping are described in a mapping technique. They are described "at the metamodel level" in such a way that they are applicable to all sets of source models being instances of the source metamodels.

For example, if a mapping technique was to describe how to map a UML model of a Java application to a corresponding Java source code model, the description would contain rules like "a classifier maps to a Java class declaration, where the name of the class is taken to be the name of the classifier".

Figure 3 illustrates these concepts.

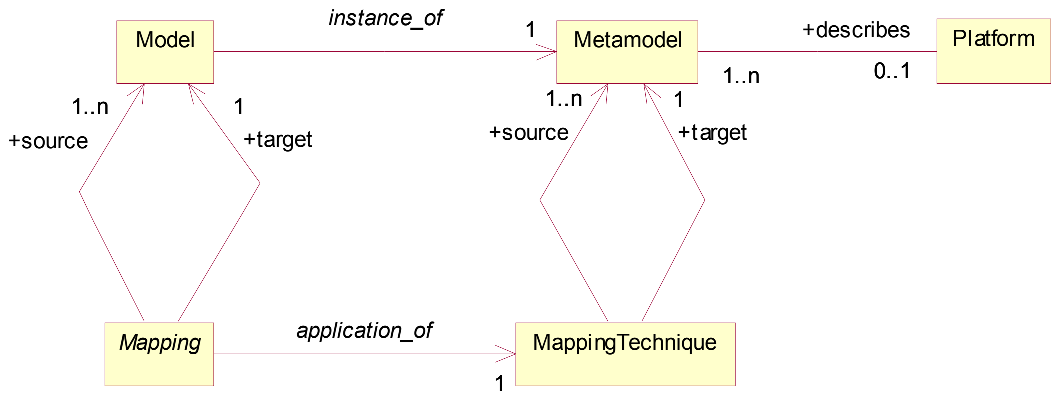


Figure 3 MDA model overview

The Platform Stack

A platform is the specification of an execution environment for models. For example, a microprocessor family specifies the environment for the execution of certain assembly code models, the Java Virtual Machine provides the specification of the environment for the execution of Java byte code, and Solaris specifies an environment also known as an operating system.

A platform is only considered in the context of MDA if there is at least one realization of it. A realization can be seen as the implementation of the specification that the platform represents.

A realization can in turn build upon one or more other platforms. For example, there is a Solaris implementation that is built on the SPARC system architecture platform which consists of the microprocessor platform and various other platforms such as the I/O subsystem. The example of Solaris shows that more than one realization may exist for the same platform. Solaris also has been ported to the PC architecture that is building upon the Intel processor platform and a standardized set of other subsystems such as the BIOS.

In theory, the platform stack can extend down to the level of quantum mechanics, but this is not very useful in most cases. Within the MDA context, platforms are only of interest as long as we want to create, edit, or view models that can be executed on them.
Of course, this is a relative definition. While somebody developing a payroll system typically will not get involved with the specifics of the assembly code of the processor family on which it is destined to run, they may have to address the level of the 3rd generation programming language (Java, C++, ...). In this case, the platform consisting of the programming language specifications and the available APIs is the last platform to be

addressed. This is done by defining corresponding metamodels. In other cases, developers may explicitly want to exploit specifics of the microprocessor the system will run on. Then they will have to create models that *do* address this lower level of abstraction.

The described concepts are illustrated in Figure 4.

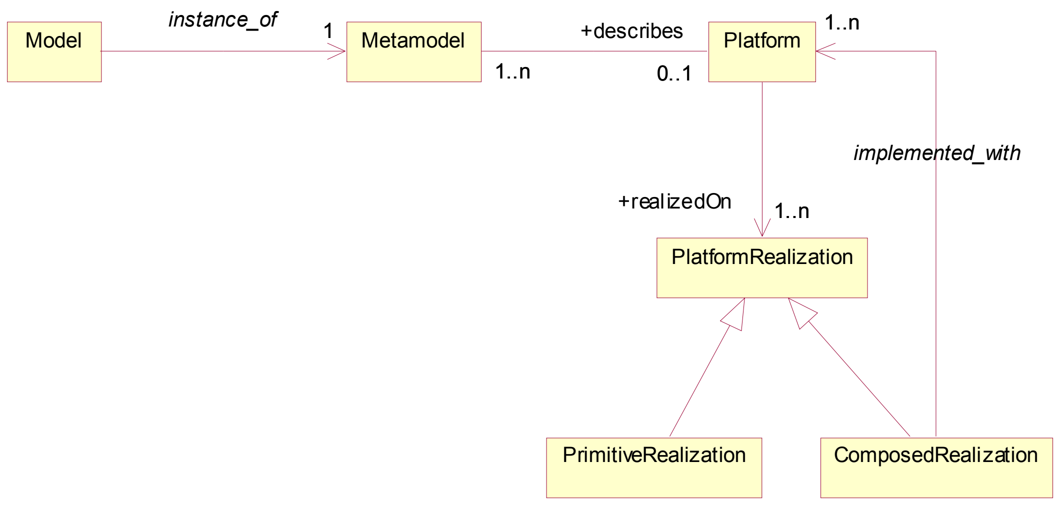


Figure 4 Models, metamodels, and platforms

Annotating Models

Incremental and iterative development has to be supported by the MDA. This requires that mappings between models are repeatable. Thus, if a mapping requires input in addition to the source models, this information has to be stored as being persistent. However, it must not be *integrated* into the source model because it is specific to the mapping, and several different mapping techniques may exist that each require different additional input. Integrating the additional mapping input with the model would make the model specific to the corresponding mapping technique, which is not desirable.

Instead, the additional mapping input is kept in annotations. A mapping may use several annotations on the source models, but it is also possible that a mapping does not require any annotations at all. A model may have several annotations that cater to several different mappings.

In the same way that a model is an instance of a metamodel, an annotation is an instance of an annotation model. These annotation models describe the structure and semantics of the annotations. A mapping technique specifies the annotation models of which it requires instances (annotations) on which instances of its source metamodels.

If a mapping technique can use more than one annotation model for a single source metamodel, then annotation models can be reused for several different mapping techniques. This, in turn, renders the corresponding annotations reusable for the different corresponding mappings.

Figure 5 illustrates the concept of annotations and annotation models. It should be noted that an annotation model in the role *requiredAnnotationModel* always belongs to one or more metamodels that are associated with the mapping technique in the *source* role. This fact is not expressed in the figure.

Constraints:

Let *M* be a mapping of the source models $S_1,\dots,S_n$ to a target model *T*, and *MT* be the mapping technique of which *M* is an application.

- For each source model *S* in *M* there must be a source metamodel *SM* in *MT* where *SM* is the metamodel of *S*. For each *SM* in *MT* there is an *S* in *M* where *SM* is the metamodel of *S*.

- The target metamodel *TM* in *MT* is the metamodel of the target model *T* in *M*.

- For each required annotation *R* in *M* there is a required annotation model *RM* in *MT* where *RM* is the annotation model of *R*. For each *RM* in *MT* there is a *R* where *RM* is the annotation model for *R*.

- For each provided annotation *P* in *M* there is a provided annotation model *PM* in *MT* where *PM* is the annotation model of *P*. For each *PM* in *MT* there is a *P* where *PM* is the annotation model for *P*.

- If *AM* is the annotation model for annotation *A* then the metamodel *Mm* owning *AM* must be the metamodel for the model *M* owning *A*.

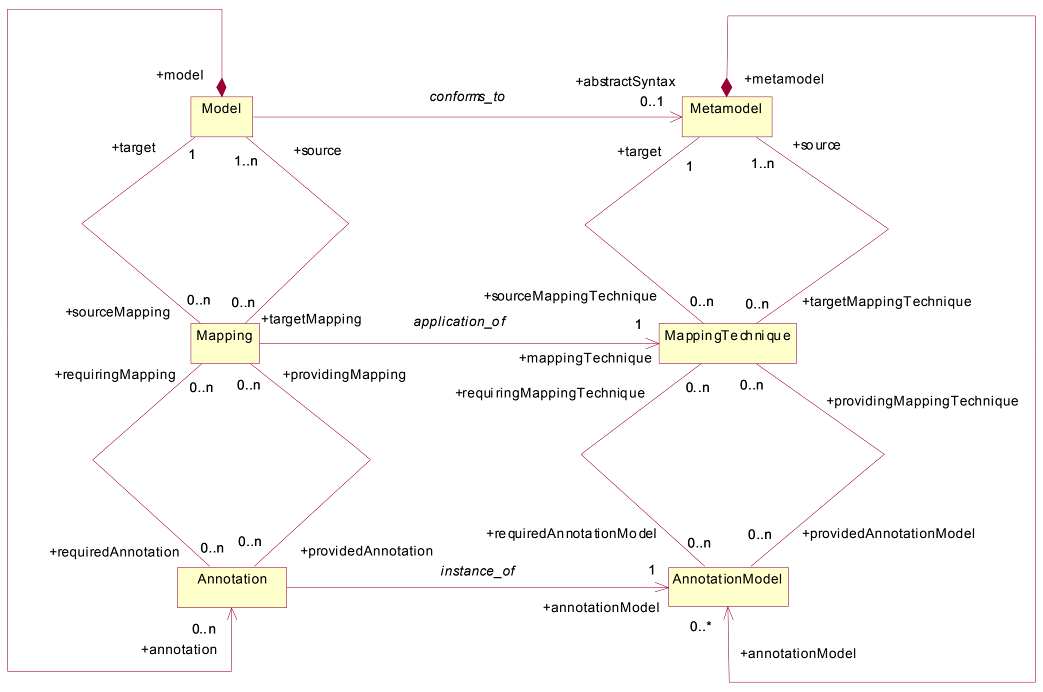


Figure 5 Annotations and annotation models

Figure 6 illustrates the idea of annotating a model for different mappings leading to different platform-specific models (PSM). The source model for these mappings is a platform-independent model (PIM) with regard to the two target platforms *A* and *B*. The annotations do not pollute the PIM which allows the PIM to be mapped repeatedly to two different PSMs.

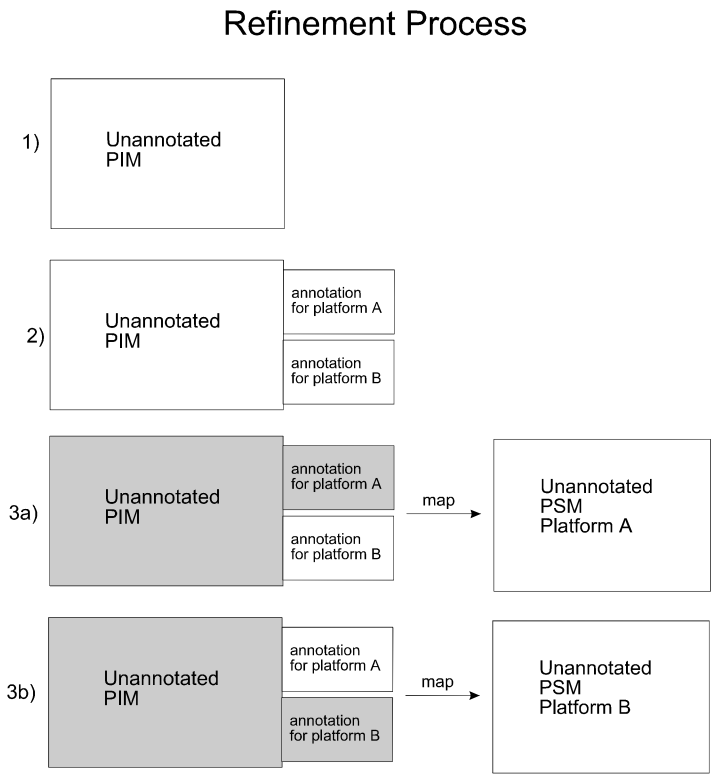

Refinement Process
1)
Unannotated PIM
2)
Unannotated PIM
annotation for platform A
annotation for platform B
3a)
Unannotated PIM
annotation for platform A
annotation for platform B
map
Unannotated PSM Platform A
3b)
Unannotated PIM
annotation for platform A
annotation for platform B
map
Unannotated PSM Platform B

Figure 6 Models and their annotations

Representing Models

A model has to be represented in some way. For example, the model of a Java source is typically represented in an ASCII file, whereas a UML model could be represented in the graphical UML notation or in ASCII using the UML textual notation.

While the representations in these examples are suited for editing by humans, there may be other representations of the same model that are better suited for transformation through programs, for example. Thus, a Java compiler would first transform the ASCII Java source into an abstract syntax tree before continuing with the semantic analysis. And a generator designed to transform a UML model into something else would use a representation of the model that provides a MOF-compliant interface such as the MOF CORBA APIs or JMI.

Figure 7 shows examples of different base metamodels (EJB Compact Bean, EJB Expanded Bean, and Java Source) along the x-axis, and different representations for each of them along the y-axis. For example, the metamodel *EJB Expanded Bean* can be represented in UML, using UML's standard extensibility mechanisms (stereotypes, properties, and constraints), while a UML model can in turn be represented either in the graphical or the textual UML notation.

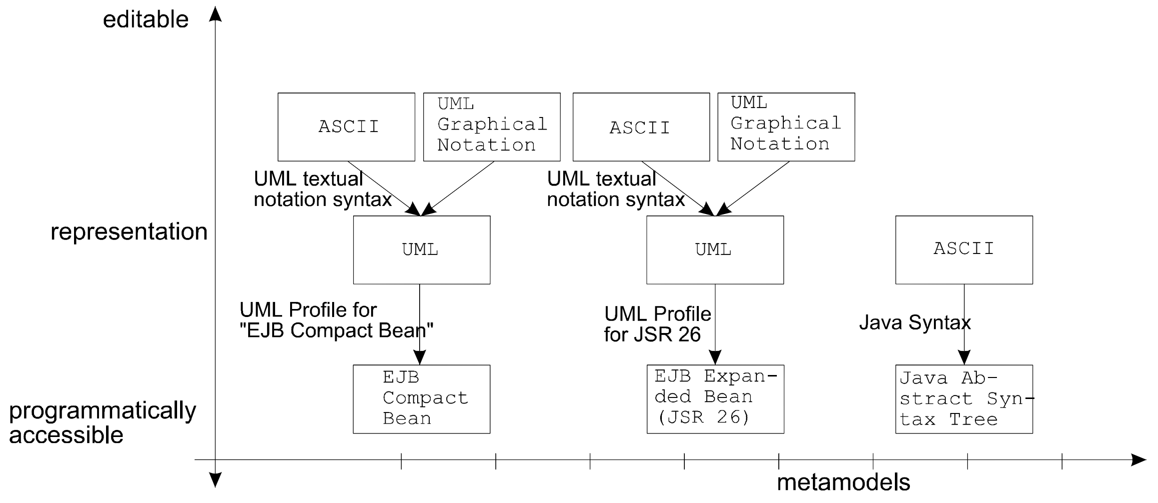

editable
representation
programmatically accessible
metamodels
ASCII
UML Graphical Notation
ASCII
UML Graphical Notation
UML textual notation syntax
UML textual notation syntax
UML
UML
ASCII
UML Profile for "EJB Compact Bean"
UML Profile for JSR 26
Java Syntax
EJB Compact Bean
EJB Expanded Bean (JSR 26)
Java Abstract Syntax Tree

Figure 7 Model representation

A representation of a model is a model in itself, and therefore is an instance of a metamodel. For example, the ASCII representation of a Java source complies with the ASCII metamodel (which simply provides an alphabet). In this case, the mapping is described by the Java syntax. In the example of the mapping between *UML* and *EJB Expanded Bean* the mapping is described by a *UML profile* that defines how elements of the metamodel are expressed in UML by the use of stereotypes and other extension mechanisms.

Figure 8 shows how the representation relation between models can be described in terms of models, metamodels, and mapping techniques.

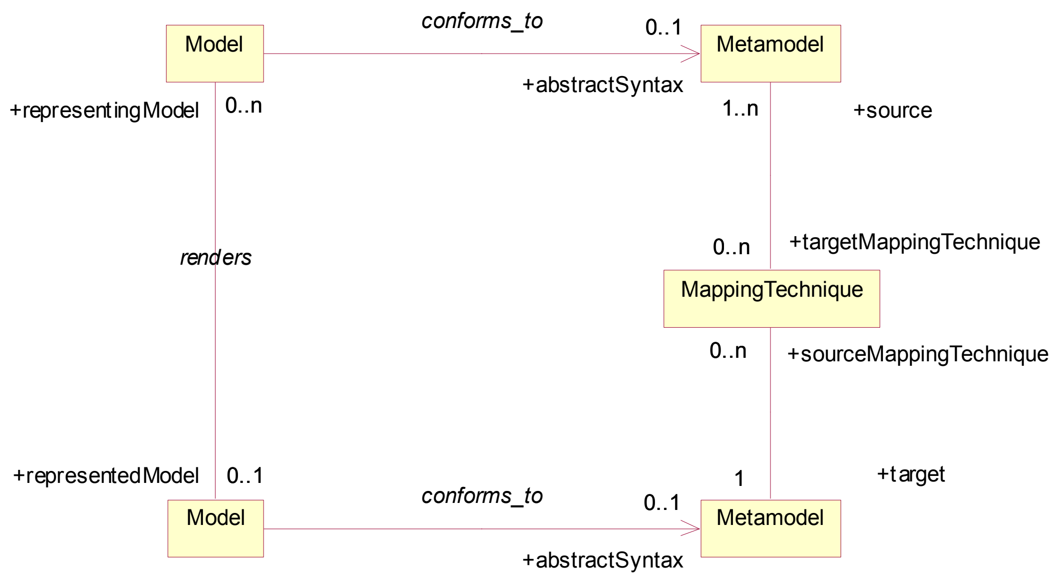


Figure 8 Model representation

Vocabulary

Model

Definition

A model abstracts from a domain.

Rationale

In MOF terminology, a model constitutes the M1 level, at which its domain constitutes the corresponding M0 level.

See Also

- Metamodel

Metamodel

Definition

A metamodel is a model whose domain consists of models.

Rationale

A metamodel describes the syntax and semantics of models.

In MOF terminology, a metamodel constitutes the M2 level, at which its models constitute the corresponding M1 level.

See Also

- Platform

Platform

Definition

A platform is a set of protocols, interfaces, and semantics specifications.

Rationale

The platform notion is introduced in order to define model semantics with respect to an IT domain. It is refined by the notions of formalism and technology.

Note:

> Formalism and technology are context-sensitive notions. They are used for clarification purposes and have not been introduced into the MDA model.

A platform is expressible by one or more metamodels that describe how models for this platform have to be structured and how they map to the semantics specified for this platform. A platform thus represents a well-defined level of abstraction.

If a platform specifies execution semantics, then the models for this platform can be executed on this platform.

A platform may use one or more other platforms to implement the execution of its models (cf. Platform Realization). This relation leads to a tree of platforms.

The leaves of this tree have to be MDA-approved platforms that permit useful execution of the corresponding models. Examples for such platforms are the Java language or CORBA.

Other lower-level platforms may be used in turn to implement these MDA-approved platforms, but it is not in the scope of MDA to describe, create, transform, or modify models at the level of these platforms. Whether a platform is considered lower-level or not depends upon the context of the problem to be solved. Seen from the perspective of creating business systems, the Java VM and the Intel Pentium III microprocessor are both examples of such lower-level platforms.

A platform defines the semantics which are part of all metamodels describing the platform.

See Also

- Formalism
- Technology
- Platform Realization

Formalism

Definition

A formalism is a platform that is unable to execute models with the degree of effectiveness and efficiency required to solve the problem.

Rationale

Whether a platform is a formalism or a technology depends on the problem to be solved.

Example: Executable UML is considered a technology if model simulation is an adequate solution to a given problem. Executable UML is considered a formalism if integration with legacy systems is required to solve a given problem.

UML and the CRC technique are typical examples of formalisms.

See Also

- Technology

Technology

Definition
A technology is a platform that is able to execute models with the degree of effectiveness and efficiency required to solve the problem.

Rationale
A model is executed effectively if the desired effects are reached in the specified target environment. A model is executed efficiently if the required resources (time, memory consumption, processor load) meet their specified limits.

CORBA, RDBMS, and Java are typical examples of technologies.

See Also
- Formalism

Mapping Technique

Definition
A mapping technique is a model whose domain consists of mappings.

Rationale
A mapping technique describes how elements of its source metamodels (and their source annotation models) map to elements of its target metamodel (and its target annotation model).

Models are not mapped to one another in an ad hoc manner. The mapping of a source model to a target model follows mapping rules. These rules are expressed in terms of the metamodels describing the source and target models, respectively.

Example of a mapping:

> There is a source UML model containing the classes "Customer" and "Account". The source model is mapped to a CORBA target model containing the interfaces "Customer" and "Account".

Example of a mapping technique:

> There is a mapping technique describing how to map UML elements to CORBA elements. This mapping technique states that the UML element "Class" maps to the CORBA element "Interface". The example mapping is an application of the example mapping technique.

A mapping technique uses annotation models to specify how to structure additional mapping-related information.

See Also

- Mapping
- Annotation Model
- Annotation

Mapping

Definition

A mapping is the process of transforming one or more source models into one target model.

Rationale

The rules to be applied in the course of a mapping are defined by a mapping technique. The source and target models of the mapping and their required and provided annotations correspond to the source and target metamodels and their required and provided annotation models.

It is sufficient to define a mapping as a mapping to one target metamodel because a simultaneous refinement to several target metamodels can be broken down into multiple mappings, each describing the mapping to exactly one target metamodel.

However, this does not apply in the case of a mapping from multiple source metamodels to one target metamodel. Such a mapping technique cannot be equivalently expressed as the combination of multiple separate mapping techniques.

Explanation:

> The effect is comparable to a mathematical function with a vector with n elements as input and a vector with m elements as output. Such a function can easily be looked at m functions, each with a vector with n elements as input. It cannot, though, be expressed in terms of n functions.

A mapping can be divided into two steps:

1. **Annotating the source models for the mapping technique.** This step creates all required annotations (see also Figure 5). Thus, if the mapping technique does not specify any required annotation model, this step is not required.

- **Executing the mapping technique.** This step results in an instance of the target metamodel. Annotations may be provided for the target model, if the mapping technique specifies a provided annotation model.

Annotation Model

Definition

An annotation model is a model whose domain consists of annotations.

Rationale

An annotation model defines the structure and semantics of annotations.

An annotation can take two roles with regard to a mapping – required annotation and provided annotation. As a consequence there are also two roles an annotation model can take with regard to a mapping technique – required annotation model and provided annotation model.

If the mapping technique neither requires nor provides any annotations, it does not have to provide an annotation model.

Multiple mapping techniques may share annotation models. This allows for the reuse of annotations.

See Also

- Mapping Technique
- Required Annotation Model
- Provided Annotation Model

Annotation

Definition

An annotation is additional information that is attached to a model.

Rationale

A mapping may require further information in addition to the source and target models. This additional information is called an annotation. There are two types of annotations with regard to mappings – required annotation and provided annotation.

Multiple annotations can be aggregated by a source model, allowing it to be subject to multiple mapping techniques. This is illustrated in Figure 6.

Usually an annotation will be structured into element-specific parts.

See Also

- Required Annotation
- Provided Annotation
- Annotation Model

Required Annotation

Definition

A required annotation is an annotation that is attached to the source model of a mapping in order to facilitate an unambiguous mapping to a target model.

Rationale

A required annotation can be provided in the preparation of the mapping or can be recorded in the course of an interactive mapping.

It is an instance of one of the required annotation models of the applied mapping technique.

A required annotation anticipates detail to be added in the target model and thus allows for traceability between source model elements and the addition of detail in the target model. As a result, the mapping becomes repeatable.

Example of anticipated added detail:

> The CRC technique states that a class has responsibilities. UML states that a classifier has features. CRC responsibilities cannot be mapped to UML features because of the obvious ambiguities. On the other hand, a UML feature might correspond to a CRC responsibility. Annotating a CRC responsibility with the information required for an unambiguous mapping to an UML feature anticipates that UML features may be added to an UML classifier. If an annotated CRC model is repeatedly mapped to an UML model, the UML features corresponding to CRC responsibilities will be present as a result of the mapping and will not have to be added again.

See Also

- Required Annotation Model
- Provided Annotation

Provided Annotation

Definition

A provided annotation is an annotation that is attached to the target model of a mapping in order to accommodate surplus information contained in the source model.

Rationale

A provided annotation holds source model information that otherwise would be lost after the application of a mapping technique. A mapping is not required to create target annotations.

It is an instance of the provided annotation models of the applied mapping technique.

A provided annotation can hold information that facilitates unambiguous backward traceability.

Example of backward traceability:

> A UML association end is mapped to a set of CORBA operations. It is impossible to tell whether a CORBA operation has resulted from the mapping of a UML

association end unless it is annotated with the UML association end it corresponds to. This information has to be provided as the result of a mapping.

A provided annotation can hold information that is contained in or attached to the source model of a mapping and will be of use for further mappings.

Example of anticipated information:

A CRC card is annotated with the name of a table in the existing corporate database. The CRC card is then mapped to an EDOC entity. The mapping from CRC to EDOC does not require the annotated table name. However, the annotated table name can be used for the mapping from EDOC to RDBMS. It is useful to provide the annotated CRC table name to the EDOC entity in order to avoid repeated specification.

A provided annotation helps preserve source model information that otherwise could not be mapped to the target model.

Example of surplus detail:

A UML class can have an invariant. When a UML class is mapped to a CORBA interface, the constraint information will be lost unless the CORBA interface is annotated, e.g. with a comment complying with a defined syntax.

See Also

- Provided Annotation Model
- Required Annotation

Required Annotation Model

Definition

A required annotation model is an annotation model whose domain consists of required annotations.

Rationale

In some cases the source metamodel information is not sufficient to define an unambiguous mapping technique to the target metamodel. A required annotation model describes the required additional information.

Example of a required annotation model:

Suppose there is a mapping technique from UML to EJB 1.1. A UML class can be mapped either to a session bean or to an entity bean. This ambiguity can be resolved by annotating each UML class with its EJB kind (either session bean or entity bean). The required annotation model for the mapping technique would state that a UML class has an annotated EJB kind.

See Also

- Required Annotation
- Mapping Technique
- Provided Annotation Model

Provided Annotation Model

Definition

A provided annotation model is an annotation model whose domain consists of provided annotations.

Rationale

In some cases the target metamodel of a mapping technique is unable to accommodate all information contained in the source metamodels and their required annotation models. A provided annotation model describes how to accommodate the information that would otherwise be lost.

Example of a target annotation model:

> Suppose there is a mapping technique from UML to CORBA. A UML association end maps to a number of CORBA operations. Those operations are unaware of the association end they correspond to. This problem can be solved by annotating each CORBA operation with its corresponding UML association end (if any). The provided annotation model for the mapping technique would state that a CORBA operation can have an annotated UML association end.

See Also

- Provided Annotation
- Mapping Technique
- Required Annotation Model

Refinement

Definition

Metamodel B is said to be a refinement of Metamodel A if a "reasonable" (semantic-preserving) surjective mapping technique (or mapping in the algebraic sense) from A to B cannot be provided.

Rationale

This means that after mapping semantics to the greatest degree possible, the target metamodel still allows for the addition of more detail ("non-surjectiveness", cf. Figure 9).

Regarding operational semantics of the two metamodels, there may be equivalent transformations between the two (e.g. between Java source code and Java byte code).

There may be multiple transformations of this kind, varying only in terms of details which are not visible or expressible in the target metamodel.

As a result, if one metamodel does not allow for the specification of more detail than another, it cannot be a refinement of the other. In this case it could, at most, define another view of the same platform, if both metamodels describe models for the same platform.

See Also

- Abstraction

Abstraction

Definition

Metamodel B is said to be an abstraction of Metamodel A if a "reasonable" (semantic-preserving) surjective mapping technique (or mapping in the algebraic sense) from A to B can be provided .

Rationale

An abstracting metamodel shows one or more or all aspects covered by the abstracted metamodel in less detail (more abstract), suppressing detail that is considered insignificant for the level of abstraction represented by the abstracting metamodel.

Examples of such abstractions are specific types of source code visualizations. One abstraction could simply be the set of declared classes and interfaces together with their inheritance relationships. Another one could show the set of all operations together with their usage dependencies.

Another example is the abstraction of the detailed EJB metamodel describing a component as a Home-interface, a Remote-interface, and a Bean-class into a compacted metamodel where this component is represented as only one model element.

See Also

- Refinement

Adding Detail

Definition

The set of models complying with the target metamodel of a mapping technique can be completely partitioned into a set of disjoint equivalence classes, with each such equivalence class containing exactly one element from the result set of the mapping technique. There may be multiple different equivalence class partitionings.

Adding detail to a model is then defined as modifying the model in such a way that the unmodified and modified models are both elements of the same equivalence class (cf. Figure 9).

Rationale

It is evident that the concept of the "addition" of detail is most meaningful if the equivalence classes are structured according to some reasonable measure of similarity defined in the target metamodel so that for all elements in an equivalence class there is no mapping result that is more similar than the one mapping result that is contained in the class.

Example:

> If a target metamodel adds an attribute to the source metamodel, then a reasonable equivalence class definition is one that considers two elements as equivalent if and only if they vary exclusively in the value of the additional meta-attribute.

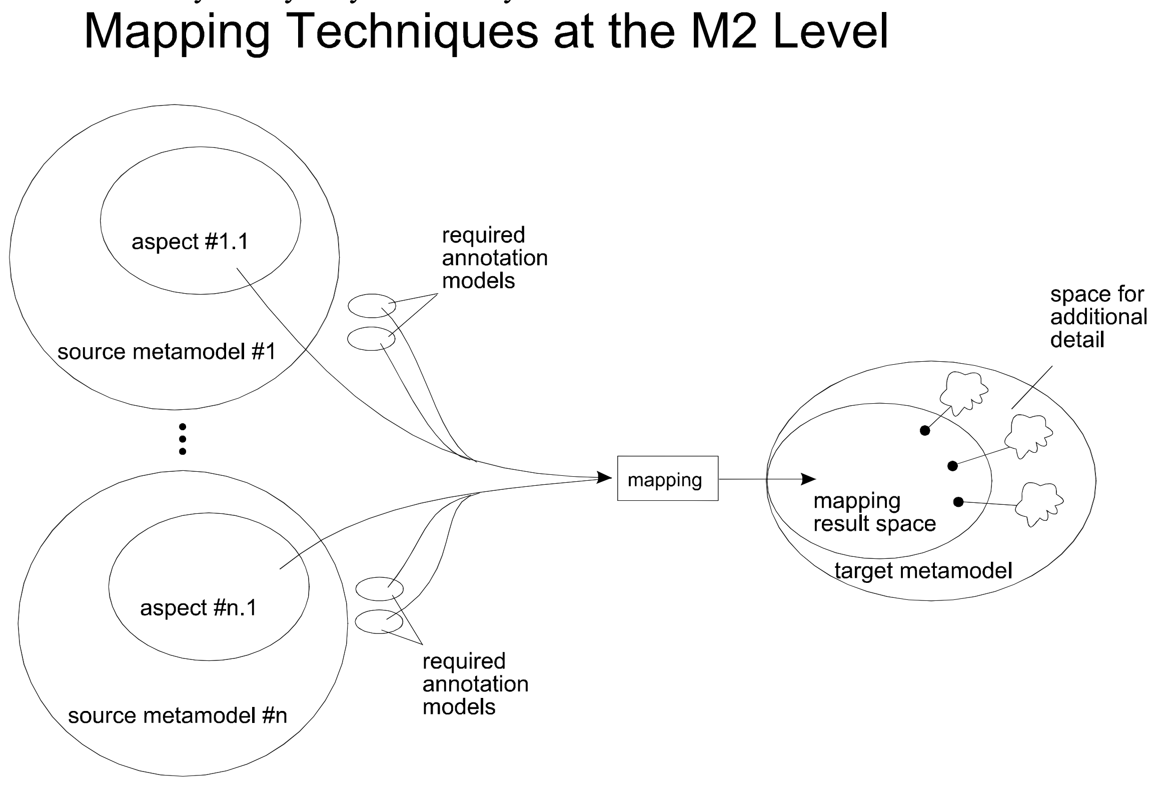

Mapping Techniques at the M2 Level
aspect #1.1
source metamodel #1
required annotation models
aspect #n.1
source metamodel #n
required annotation models
mapping
space for additional detail
mapping result space
target metamodel

Figure 9 Mapping Technique Details

Platform Realization

Definition

A platform realization implements the protocols, interfaces, and semantics defined by a platform.

Rationale

A platform realization represents a set of one or more platforms that can be used together to implement another platform.

Examples are the platforms C++ and Solaris that can be used to implement the Java Virtual Machine, which, in turn, is a platform.

See Also

- Platform

PIM/PSM

Definition

A model is said to be independent of a set of platforms

1. if its metamodel abstracts from those platforms and
2. if for each abstracted platform there is a sequence of mapping techniques from its metamodel to a metamodel describing this platform.

A model is said to be specific to a platform if its metamodel describes this platform.

Rationale

In view of the above definition of platform, a model is always an instance of exactly one metamodel which in turn describes exactly one platform. The model and its metamodel are said to be "specific to this platform".

Model *M* and its metamodel *MM*, which are both specific to platform *A*, are said to be independent of the set of platforms B_1, ..., B_n if for all $1 \leq i \leq n$ there is a mapping technique from *MM* to one of B_i's metamodels, say MM_i (please note that a corresponding annotation model may support each such mapping technique).

In some cases, a set of platforms may be grouped together and referred to by means of a single term. In this case, if a model and its metamodel are independent of all the platforms in this set, then the model and its metamodel can be said to be independent of this platform group.

Example:

> If the set of operating systems for which there is an implementation of the Java Virtual Machine is termed "operating system platform", then each of these platforms is used in a realization of the Java platform, and the Java source code model of an application is independent of all the operating systems in this group. Therefore, the Java source code model can be said to be "independent of the operating system platform".

Example

The Business Object Model

The example is based on a very simple business model illustrated in Figure 10. The business logic behind this model can be expressed as follows:

- A customer knows its name and may have an arbitrary number of accounts.
- An account knows its number and balance and it must be associated with at least one customer.
- An arbitrary amount can be transferred between two accounts.

For the sake of simplicity, we have kept the semantics of the business object model (BOM) to a minimum. A real model would probably contain further restrictions such as "source and target account must be different", would impose limitations on the amount, etc.

Our description and the UML diagram of the business model do not contain any information about the IT system that could be used for an implementation. This model is platform-independent with respect to any IT implementation. It is the highest level PIM in this example. According to the MDA, the term platform refers to technological and engineering details that are irrelevant to the fundamental functionality of a software component.

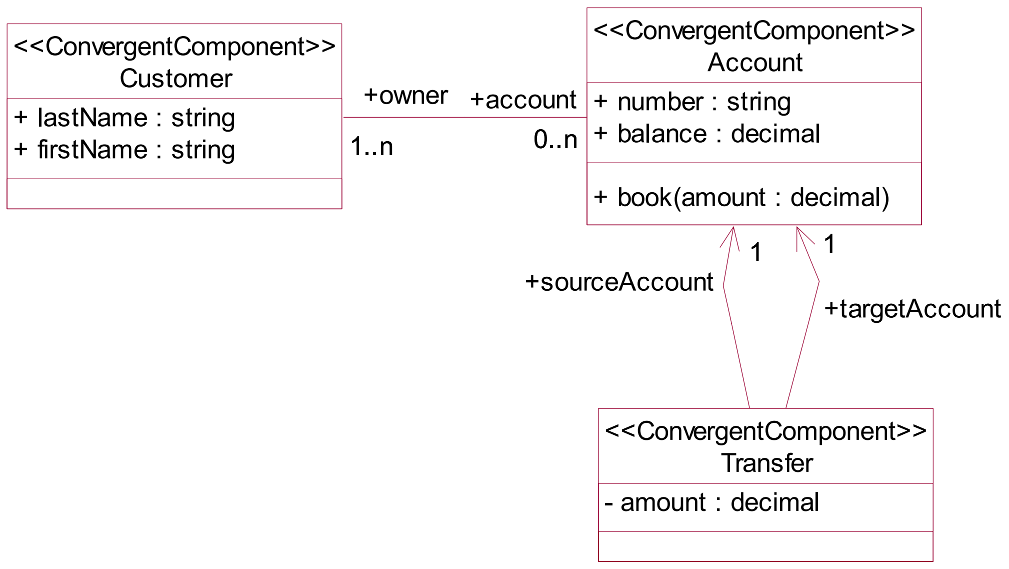


Figure 10: Business Object Model of a simple customer-account relationship.

All objects in this BOM are characterized by the stereotype "ConvergentComponent". The name of this stereotype is derived from the concept of a convergent model, a model in which the business and IT views are converged into a single model. All artifacts of the BOM are described in a UML profile of the corresponding metamodel. We could have used a different profile with other stereotypes. The important issue here is that the stereotypes do not imply a specific IT implementation. The indication of attribute and operation visibility (the +/- signs) is optional and does not presume any specific implementation.

The type "decimal" used in this BOM denotes an element of the platform-independent metamodel, whose semantics are those of an arbitrary precision signed decimal number. Decimal is a specialization of UML DataType. Again, no assumption is made about the IT implementation of this type, decimal just expresses the business requirement of a precise amount. As will be seen in the following sections, different platforms may use different mechanisms to provide the required accuracy. Similarly, the attribute type „string" denotes a character sequence of arbitrary length.

The "EJBcompact" Model

Our BOM is now mapped to an IT platform. This entails the initial decision for an IT platform, in this case the J2EE platform, and the choice of a metamodel and a corresponding UML profile that reflects the specific features of this platform. There might be several different metamodels available for the same target platform, each providing its own UML profile. As will be seen later in this example, different metamodels for the same target platform may provide varying degrees of detail and expressiveness. Currently, various tool vendors support metamodels having different degrees of detail and provide appropriate tool suites for their preferred modeling styles. In this case, the "EJBcompact" UML profile is used, and the resulting model is illustrated in Figure 11.

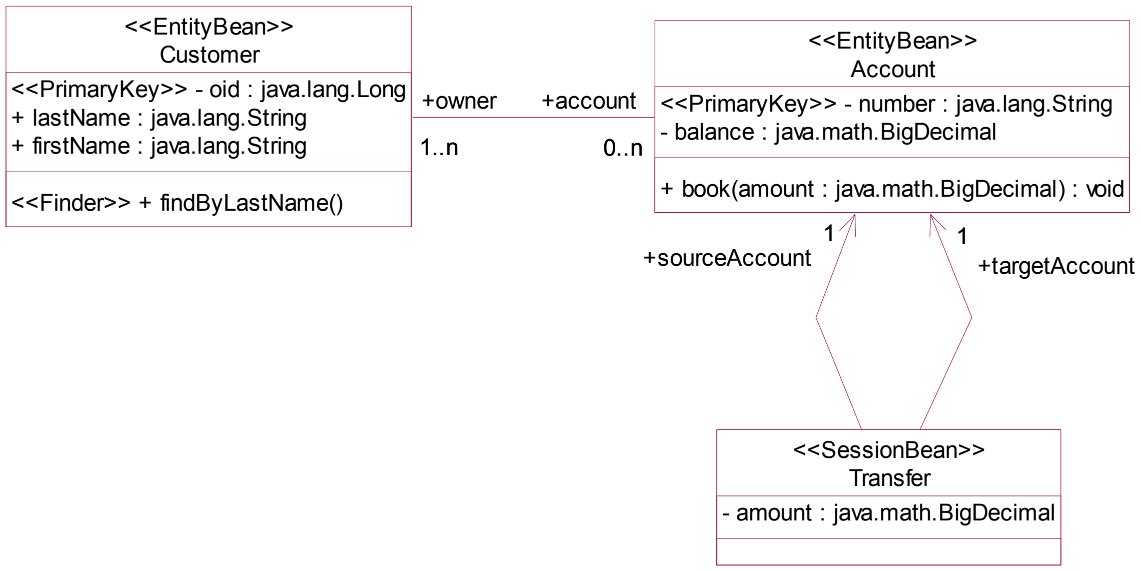

<<EntityBean>>
Customer
<<PrimaryKey>> - oid : java.lang.Long
+ lastName : java.lang.String
+ firstName : java.lang.String
<<Finder>> + findByLastName()
+owner
+account
1..n
0..n
<<EntityBean>>
Account
<<PrimaryKey>> - number : java.lang.String
- balance : java.math.BigDecimal
+ book(amount : java.math.BigDecimal) : void
1
+sourceAccount
1
+targetAccount
<<SessionBean>>
Transfer
- amount : java.math.BigDecimal

Figure 11: The "EJBcompact" model, a mapping of the BOM according to the "EJBcompact" mapping rules.

The "EJBCompact" model in Figure 11 is the PSM in the mapping process from the BOM in Figure 10 (in this case, as the PIM) to an "EJBCompact" model. A comparison of both models shows great similarities in the overall structure. The number of classes and associations are identical as are the names of the objects. This is an advantage of the "EJBcompact" UML profile: it is close to the BOM and therefore the mapping rules are easy and can be applied in a straightforward manner. Technical details are kept to a necessary minimum, such as bean types (defined with stereotypes), language-specific data types or additional stereotypes for persistent storage in a database (<<PrimaryKey>>).

While the PIM and PSM in this example look very similar, one should be aware that certain design decisions are made in the mapping process. The kind of bean (entity or session) is defined for each component. The decision whether a ConvergentComponent maps to an EJB session bean (either stateless or stateful) or to an EJB entity bean depends upon its required scope and lifetime. The necessary information has to be provided as an annotation in the PIM or by human interaction during the mapping process. Entity beans need a primary key for identification in the database and a factory operation that takes the modeled primary key attribute (if any) as its single argument. This can be an existing attribute or a new attribute added specifically for this purpose. It is assumed that the customer id introduced during the mapping process is maintained by the system. Furthermore, a finder method is defined for the customer bean to allow a name search for a customer.

In general, a PSM element does not have to result from a predefined mapping rule. Though it is desirable to have a clean path of refinement throughout all stages of modeling, PIM or PSM designers are free to introduce new elements. Since there is no rule with a unique result that can be applied to the information in the PIM, these design decisions require human intervention in the mapping process, . Tools are available for supporting the "EJBcompact" mapping, but the decisions must be made by a human system designer. The tools offer support for reasonable choices and a verifier checks that the required minimum of technical information was added in the mapping process (e.g. a primary key for each entity bean).

There are two important remarks on the "EJBcompact" model:

1. The model presented in Figure 11 clearly assumes the role of a PSM when the "EJBcompact" mapping is applied to the BOM. But the very same model assumes the role of a PIM in the next step towards implementation. The "EJBcompact" UML profile does not provide any types or constructs with information about technical features of a specific EJB product, so it is perfectly independent with regard to EJB implementations from different vendors.
2. The model presented in Figure 11 contains sufficient technical information for an automated mapping to an EJB product. Again, in this next step the "EJBcompact" model is the PIM and the information provided is used by code generation tools for the creation of computable Java code and EJB implementation artifacts (such as bean descriptors). Additional coding may be required for the implementation of some methods (e.g. a vendor-specific query expression for the search method).

In the "EJBcompact" model, an EJB component is modeled as a single element. There is no distinction between the technical artifacts of home and remote interfaces and the bean implementation itself. This makes for a clearer modeling style if only simple components are used. Required EJB artifacts (such as the bean description and the different interfaces) are automatically generated by the code generator according to a predefined set of mapping rules as explained above. The drawback of this modeling style is a restricted degree of freedom for the application designer. The following section describes a different modeling style for EJB applications in which experienced designers can define every detail of the technical implementation.

The "EJBexpanded" Model

Sometimes experienced designers may want to define the details of the technical EJB implementation explicitly. This is not supported in the "EJBcompact" modeling style. In this section another metamodel and UML profile for J2EE applications with the name "EJBexpanded" are introduced. Figure 12 shows the relations between classes in the PIM and the PSM in an "EJBexpanded" model. Every ConvergentComponent from the PIM is represented by three different classes in the PSM. This provides maximum flexibility for the designer, but requires more work in the mapping process.

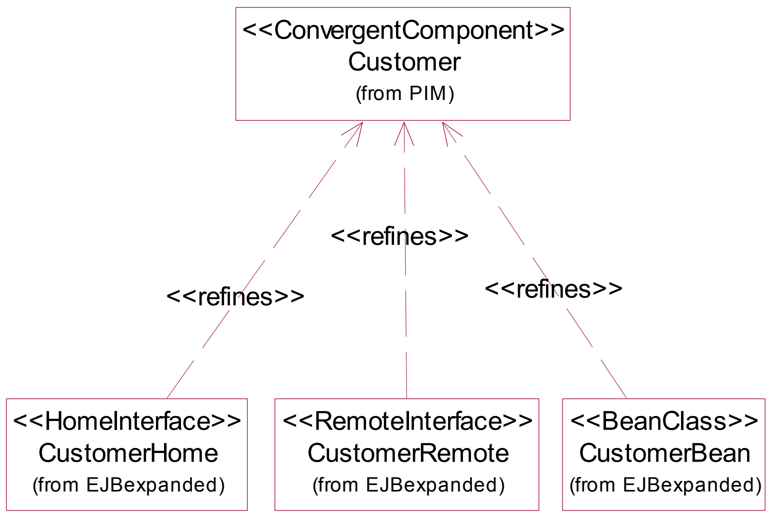
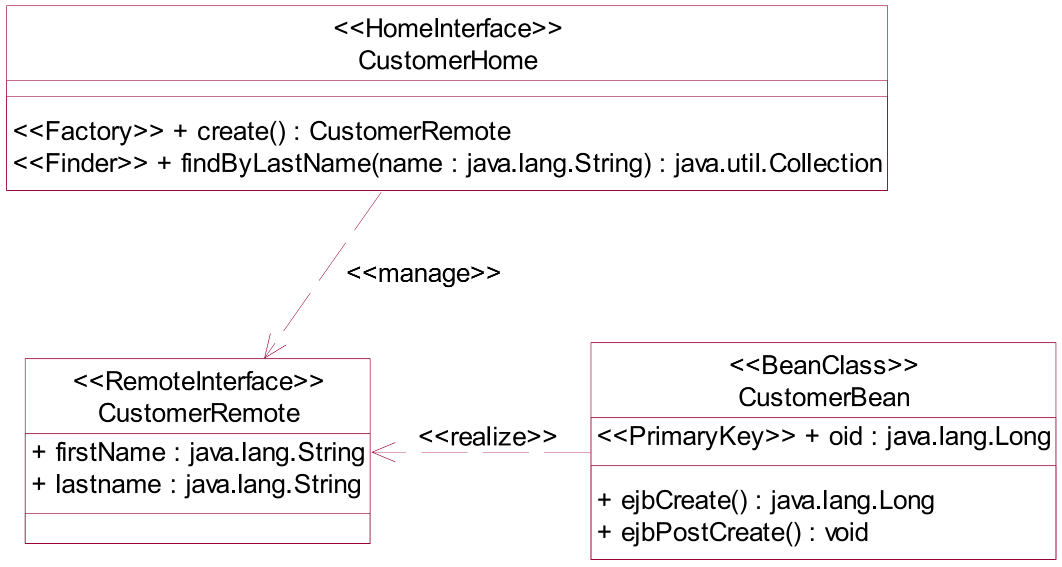


Figure 12: Relations between PIM and PSM artifacts in the “EJBexpanded” modeling style.

Since every component in the PIM is now represented by three objects in the PSM, the complete “EJBexpanded” model of our sample application contains at least nine objects and several associations. To avoid overloading the diagram, we display only the customer part of the “EJBexpanded” model in Figure 13. The account and transfer components are represented by a similar set of objects, applying the pattern from Figure 12.



<<HomeInterface>>
CustomerHome
<<Factory>> + create() : CustomerRemote
<<Finder>> + findByLastName(name : java.lang.String) : java.util.Collection
<<manage>>
<<RemoteInterface>>
CustomerRemote
+ firstName : java.lang.String
+ lastname : java.lang.String
<<realize>>
<<BeanClass>>
CustomerBean
<<PrimaryKey>> + oid : java.lang.Long
+ ejbCreate() : java.lang.Long
+ ejbPostCreate() : void

Figure 13: Part of the “EJBexpanded” model showing all artifacts for the customer component.

The “EJBexpanded” UML profile results in a much more complex PSM than the “EJBcompact” approach. Each component of an EJB (bean, home and remote interface) is explicitly described. Predefined relations between these components ensure adherence to the EJB standard. For example, the factory operation in the customer’s home interface corresponds to the remote interface and to the ejbCreate() and ejbPostCreate() methods in the customer’s implementation class.

An experienced designer can use this UML profile for individually optimized solutions whenever the application requirements justify the additional modeling effort. The increased complexity should be mitigated by tool-assisted modeling and model verification. Verification support, in particular, helps to ensure that the model is consistent with the definitions of the metamodel.

The “EJBexpanded” UML profile has another important property that can be observed in most metamodels on different abstraction levels. One element of the PIM maps to many elements in the PSM. Figure 12 clearly illustrates this “fan out” characteristic between model elements of the PIM and PSM. Each component in the PIM fans out into several components in the PSM. While this is straightforward for the mapping process from PIM to PSM, it makes the reverse mapping from PSM to PIM much more difficult. Contradictions are very likely to occur if a designer changes single elements of the PSM and then tries to perform a reverse mapping back into the PIM. This restriction generally holds for reverse mappings between PSM and PIM due to the difference in the abstraction levels of these models. The mapping between the BOM and the “EJBexpanded” modeling style is a typical example of the significant change of the abstraction level. Within a set of related MDA models, it is important that any changes are applied to the model on the appropriate abstraction level. If the business semantics need to be changed, one should first adapt the BOM and then re-apply the mappings to the lower level models.

The model in Figure 13 contains a much higher degree of IT detail, but it is still independent of any specific EJB implementation. The increased level of technical detail in the EJBexpanded model simplifies the automated mapping of this model to different EJB implementations. But the EJBexpanded model is still a PIM in terms of EJB implementations. Again, the very same model which first assumed the role of a PSM, assumes the role of the PIM in the next refinement step.

Another modeling approach uses the EJBcompact model as an intermediate step from the BOM to an EJBexpanded model. This approach requires a mapping between the EJBcompact and EJBexpanded modeling styles. Depending on the viewpoint, this could be considered as a PIM to PIM or a PSM to PSM mapping. It is not another PIM to PSM mapping, since both models are on a similar abstraction level and both are independent with respect to the same platform. Such a mapping between PIMs or PSMs is quite common. It simplifies the whole refinement process. The iteration cycle from one model to the next is shortened and the mapping is less error-prone.

The CORBA Model

In this section we will use our BOM as a PIM in a mapping to a CORBA PSM. This illustrates the major benefits of the MDA approach: the very same model can serve as a PIM for many different platform-specific models. The BOM is not restricted to implementation with a J2EE platform, but can also be realized in a CORBA environment. Figure 14 shows the corresponding PSM resulting from a mapping of the BOM to a CORBA UML profile.

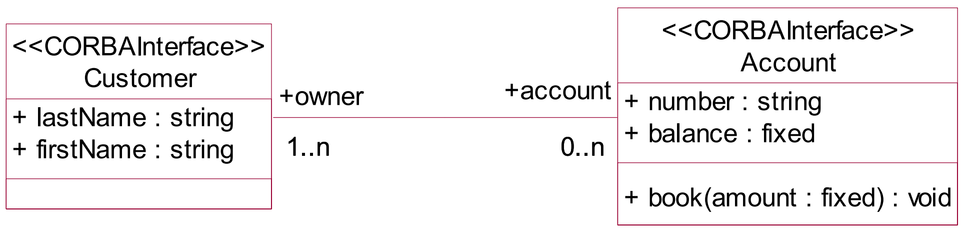


Figure 14: The CORBA model, a CORBA PSM resulting from a mapping of the BOM in Figure 10.

The CORBA PSM is still quite similar to the BOM. Each convergent component from the BOM is mapped to a CORBA interface. This is probably the most straightforward mapping to CORBA.

As we have shown for the J2EE platform, many different UML profiles, varying in degree of detail or in terms of notions, are feasible for the CORBA platform as well.

The CORBA model from Figure 14 can be mapped automatically to the following IDL code.

```
interface Customer {};

interface Account {
	typedef sequence <Customer> owner_def;

	attribute string number;
	attribute owner_def owner;
	readonly attribute fixed balance;

	void book ( fixed amount );
};
interface Customer {
	typedef sequence <Account> account_def;

	attribute string lastName;
	attribute account_def account;
	attribute string firstName;
};
```

An MDA Software Development Process Outline

Process

1. Identify the set of target platforms
2. Identify the metamodels you want to use for describing models for these platforms, together with the modeling language/profile to use to express the models in.
3. Find proper abstracting metamodels of those that are aligned along the lines of expected platform changes (striving for platform independence).
    - Too many metamodels in which additional detail is expressed may be a hindrance because of the many mapping techniques to be applied.
4. Define the mapping techniques between the metamodels in such a way that there are full paths from the most abstract metamodels to the metamodels of all target platforms.
    - Mapping techniques are most useful in the direction towards the metamodels of the target platforms, thus allowing for as much added detail as possible on as high a level of abstraction as possible and then automatically mapping down to more concrete metamodels.

    - Abstracting mapping techniques can be useful for viewing models from a more abstract perspective. If modifying the results of abstraction is to be permitted, then the corresponding refining mapping technique has to match the semantics of the abstracting mapping technique.
    - Abstracting mapping techniques can also be useful for the reverse engineering of models.
    - A refining mapping technique into a metamodel whose models are to be used for adding detail can be of practical use only if the mapping technique also works incrementally and thus allows for iterative development of the source and target models. This means that detail added to a “refined” model must not be overwritten or removed by repeated application of the mapping technique to the “abstract” models.
5. Define the annotation models required by the previously defined mapping techniques.
    - If the mapping technique requiring annotation models "skips" one or more "implicit metamodels" or one or more platforms, then it may be helpful to modularize the annotation models according to these implicit and skipped metamodels and platforms. This improves the reusability of the annotation models for other mapping techniques.
6. Implement the mapping techniques either by using tool support or by describing the steps necessary to manually carry out the mapping technique.
7. Conduct the iterations of the project. Each iteration will add detail to one or more models describing the system at one or more levels of abstraction. These additional details are mapped all the way down to the target platforms by applying the available mapping techniques.

   In particular, this process consists of three basic steps, each of which can be applied to any mapping used in the process:
    1. Add/edit detail in one or more of the mapping's source models
    2. Add/edit contents of the mapping-specific annotations to one or more of the mapping's source models
    3. Perform the mapping, which results in a new or changed target model

   These three steps can be repeated as often as required for each mapping technique used in the process during a single iteration. Each iteration will increase the amount of detail contained in the models for the target platforms.