

# Model-Driven Architecture

Stephen J. Mellor  
Kendall Scott  
Axel Uhl  
Dirk Weise

**Abstract.** This paper offers an overview of the basic MDA terms and concepts and the relationships among them, with the latter expressed in terms of the UML, and also an outline of a proposed software development process that would leverage MDA. The article also discusses the idea of “accelerated” MDA, which involves using one metamodel to do mappings of metamodels to multiple platforms.

## 1 Goals

Model-Driven Architecture (MDA) provides a framework for software development that uses models to describe the system to be built. The system descriptions that these models provide can be expressed at various levels of abstraction, with each level emphasizing certain aspects or viewpoints of the system.

The driving force behind the MDA is the fact that a software system will eventually be deployed to one or more platforms, used separately or together. Platforms are subject to change over time—and they change at different, typically higher, rates than the higher-level models of the system, which in turn tend to grow increasingly independent of the target platforms. This paper provides an introduction to the MDA’s response to this conundrum.

## 2 Abstraction

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

Relying on best practices continues to enable modelers to reach higher levels of abstraction. Tools are available that let one create a UML association between two business components, and subsequently map that simple line to literally thousands of lines of corresponding Java or C++ code and other artifacts such as configuration files and deployment descriptors. 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.

## 3 Models and Metamodels

The OMG’s Meta Object Facility (MOF) defines a model as an instance of a **metamodel**. 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*, while models that describe a system at a level of abstraction, one that’s sufficient to allow use of their entire contents for implementing the system on different platforms, are referred to as *platform-independent*. Figure 1 illustrates these relationships.

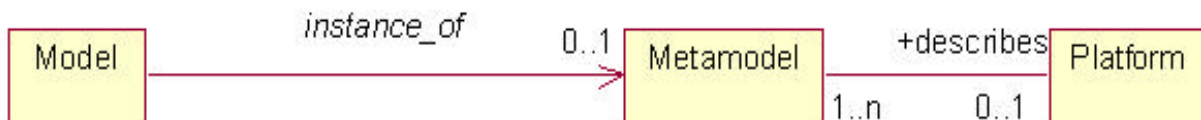


Figure 1 Model, Metamodel, and Platform

## 4 Mapping Between Models

Models may have semantic relationships with other models; for example, a set of models may describe a particular system at different levels of abstraction. It's desirable to have mappings between 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 synch.

A **mapping** between models is assumed to take one or more models as its input and produce one output model. The rules for the transformation performed by the mapping are described in a **mapping technique**. These rules are described at the metamodel level in such a way that they're applicable to all sets of source models. A modeler can automate a mapping technique by providing an executable implementation of its specification. Such an implementation allows for the automation of important steps of an MDA-driven process. However, this implementation isn't necessarily required as long as one can verify any manually conducted mapping against the specification that the mapping technique provides. For example, a mapping technique that describes how to map a UML model of a Java application to a corresponding Java source code model would have rules such as "A UML classifier maps to a Java class declaration, where the name of the class matches the name of the classifier." Figure 2 illustrates these concepts.

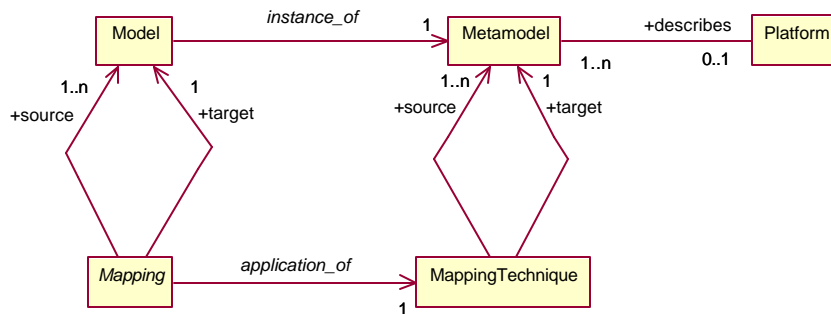


Figure 2 Overview of MDA Modeling

## 5 Platform Stack

A **platform** is the specification of an execution environment for models. A platform is relevant in the context of MDA only if there is at least one realization of it—in other words, an implementation of the specification that the platform represents. Note that a realization can in turn build upon one or more other platforms. In theory, this **platform stack** can extend down to the level of quantum mechanics, but for our purposes, platforms are only of interest as long as we want to create, edit, or view models that can be executed on them. Figure 3 illustrates these concepts in the context of models and metamodels.

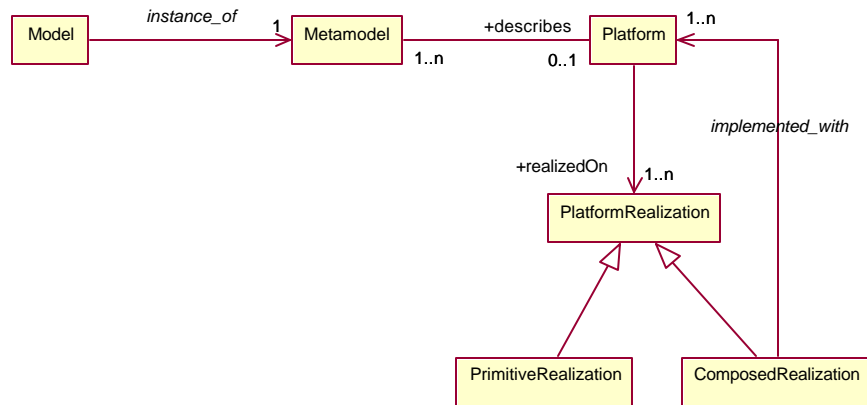


Figure 3 Models, Metamodels, and Platforms

## 6 Annotating Models

MDA must support incremental and iterative development. This means that mappings between models must be repeatable. So, if a mapping requires input in addition to the source models, this information must be persistent. However, it mustn't be *integrated* into the source model, because it's specific to the *mapping*, and several different mapping techniques may exist, each of which require different additional inputs. Integrating the additional mapping input with the model would make the model specific to the corresponding mapping technique, which is not desirable.

These additional mapping inputs take the form of **annotations**. A mapping may use several annotations on the source models; conversely, an annotation may cater to several different mappings.

Just like a model is an instance of a metamodel, an annotation is an instance of an **annotation model**. This model describes the structure and semantics of the annotation. A mapping technique, therefore, specifies the annotation models of which it requires instances (annotations) on the instances of its source metamodels. If a mapping technique can use more than one annotation model for a single source metamodel, then one can reuse annotation models for several different mapping techniques. This, in turn, renders the corresponding annotations reusable for the different corresponding mappings. Figure 4 illustrates these concepts.

Figure 5 illustrates the idea that annotating a model for different mappings leads to different **platform-specific models (PSMs)**. The source model for these mappings is a **platform-independent model (PIM)** with regard to the target platforms *A* and *B*. The annotations don't pollute the PIM, which allows the PIM to be mapped repeatedly to two different PSMs.

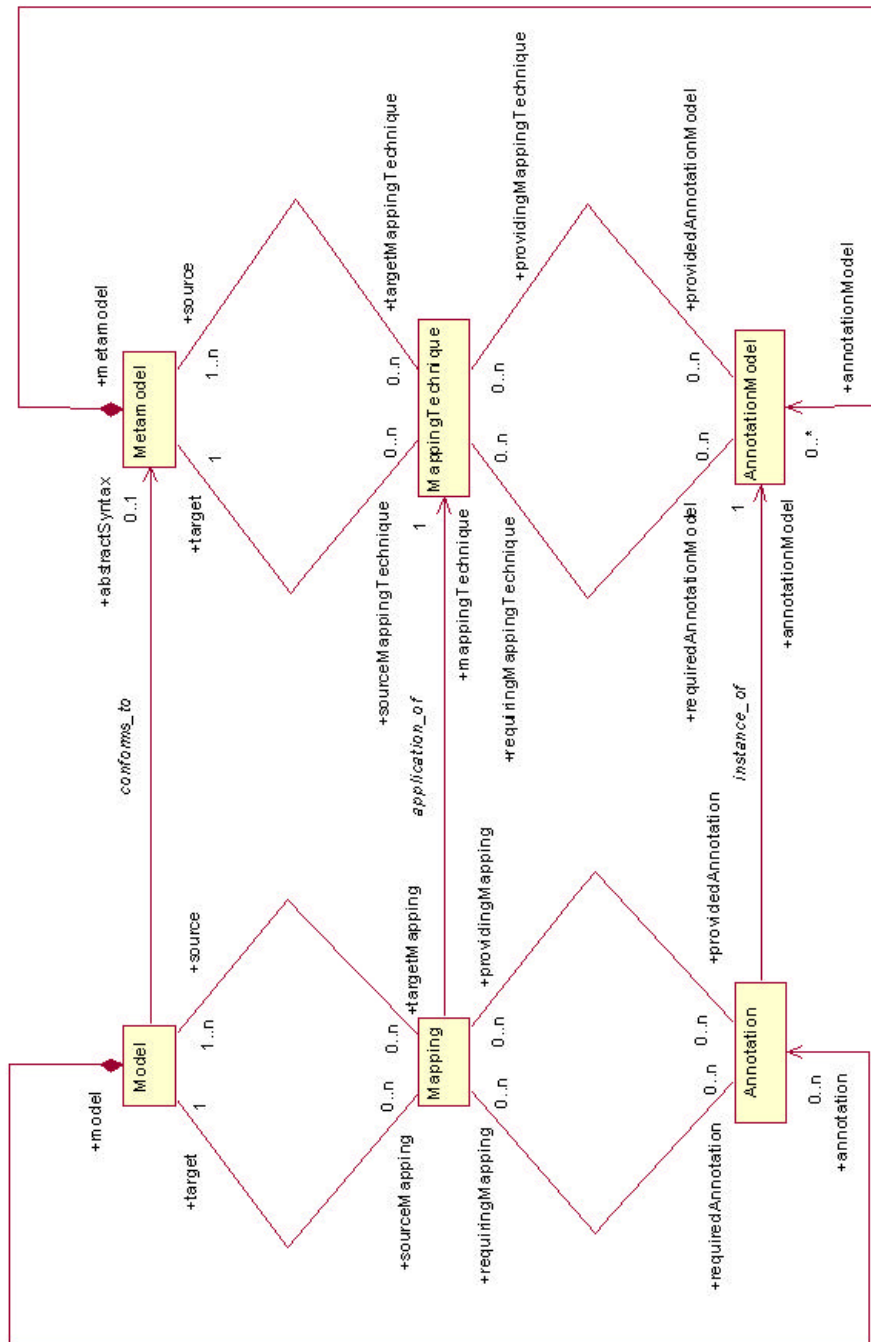


Figure 4 Model Refinement Process

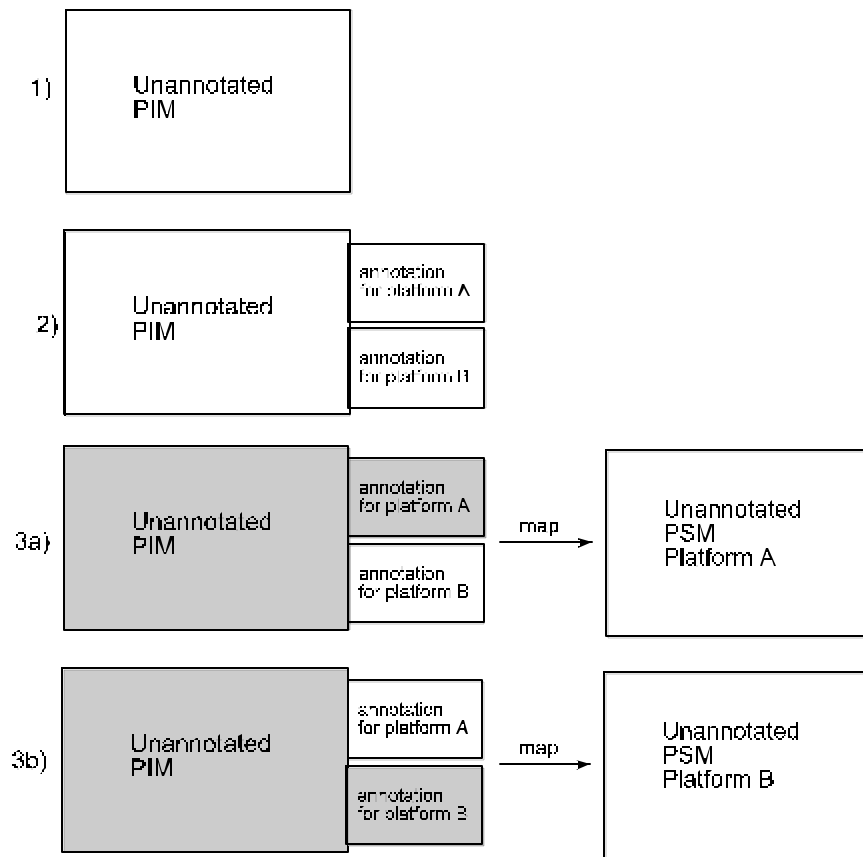


Figure 5 From PIMs to PSMs

## 7 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, while a UML model could be represented in the graphical UML notation or in ASCII using UML textual notation. While these representations are suited for editing by humans, there may be other representations of the same model that are better suited for transformation through programs—think of a Java compiler transforming the ASCII Java source into an abstract syntax tree before continuing with the semantic analysis, or 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 JMI.

A representation of a model is a model in itself, and therefore is an instance of a meta-model. For example, the ASCII representation of a Java source complies with the ASCII metamodel (which simply provides an alphabet). Figure 6 shows how the representation relationships among models can be described in terms of models, metamodels, and mapping techniques.

## 8 Outline of MDA-Compatible Software Development Process

The following seven process steps, taken together, offer a simple yet robust way to incorporate MDA into a software development project.

1. Identify the set of target platforms.
2. Identify the metamodels that you want to use for describing models for these platforms, and also the modeling language/profile in which you'll express your models.
3. Find proper abstracting metamodels from the ones that are aligned along the lines of expected platform changes (with the goal being platform independence).
4. Define the mapping techniques you'll use with your metamodels so that there are full paths from the most abstract metamodels to the metamodels of all of your target platforms.
5. Define the annotation models that these mapping techniques require.
6. Implement your mapping techniques either by using tool support or by describing the steps necessary to manually carry out the techniques.
7. Conduct iterations of the project. Each iteration will add detail to one or more models describing the system at one or more levels of abstraction. You'll map these additional details all the way down to the target platforms.

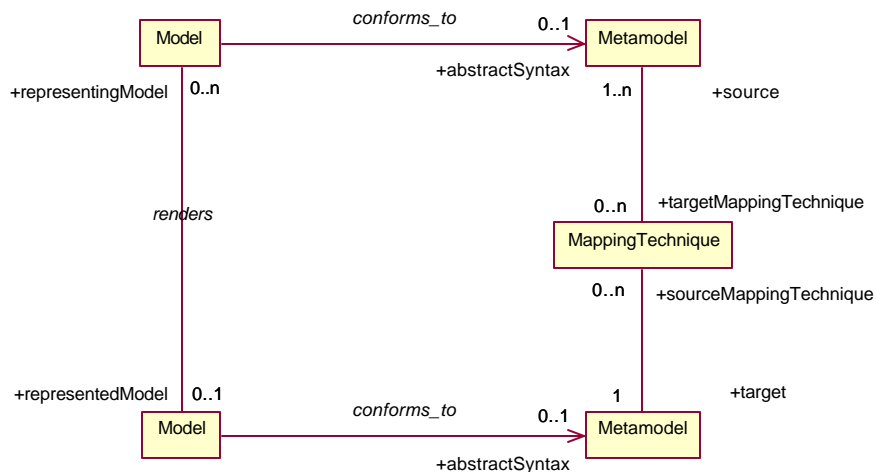


Figure 6 Model Representation

## 9 Accelerated MDA

Figure 6 shows instances of PIM and PSM models and metamodels and how these instances relate to one another. These models can themselves be populated with instances. The PSM model instance contains all of the semantic information of the original PIM model instance, as well as all of the information added as a result of the mapping technique. Because the PSM metamodel describes a platform, the PSM also includes—at least by reference—all of the elements that comprise the platform.

However, the PSM may *not* be the bottom of the platform stack. The PSM can itself be modeled as a PIM, which in turn is mapped to another PSM via a different mapping technique. The resulting PSM, which we'll call PSM', now contains all of the semantic information of the original PIM model instance, the information added as a result of the application of the first mapping technique, *and* the information added by the application of the second mapping technique. Because each PSM metamodel describes a platform, the second-level PSM also includes the elements that comprise *both* platforms.

The recursion ends when the last PSM (a PSM with some number of primes) is directly implementable, either by generating code or by being directly executable on the virtual machine modeled by the last PSM. The platform stack, and the distinction between the last platform (a PrimitiveRealization) and those platforms above it, are illustrated in Figure 3.

This approach is ultimately flexible. Any model, regardless of exactly how platform-independent it is, can have a metamodel defined for it, and developers can easily model applications using their favorite notations, conceptually optimized for the platform types to which the models are to be mapped. However, this approach requires the definition of multiple mapping techniques to preserve semantics at each level. The result is that platforms can become “silos” in which certain mappings can only be applied under a certain set of assumptions. For example, once a particular kind of inheritance has been applied in one metamodel, it will be difficult to use a different kind of inheritance in a subsequent model.

An alternative is to use the *same* metamodel to capture each platform. In this formulation, a model is always expressed using the same concepts, enough to execute the model. In addition, the modeler chooses the concepts such that the model can be translated into any arbitrary platform. This “Executable and Translatable UML” doesn't contain any information about software structure, only about behavior of the subject matter under study. The metamodel describing Executable and Translatable UML is therefore much smaller than UML.

When an Executable and Translatable UML model is rendered as an implementation, it is annotated as described above and woven together with the platform specifics by a mapping technique that is completely general—it needs only be written once. Platform-specific mappings, of course, will vary, but that's inherent in the fact that platforms differ.

This accelerated approach bypasses the PSM. There's no need to capture the conjoining of the developer model and the platform as a distinct model. The developer model and the platform model are woven together directly into an implementation. This approach also increases the applicability of tools, because there's only one metamodel on which to operate, rather than multiple metamodels (one for each platform).

## **10 Conclusion**

MDA is the OMG's next step in solving integration problems through open, vendor-neutral interoperability specifications. It is constantly evolving based on the experiences of OMG members in creating standards for implementation language-independent models in CORBA and developing standards such as UML and MOF. The key concepts described in this paper form the foundation for ongoing exploration of what MDA will be, while the process outline provides a healthy start toward the definition of how software development teams can maximize the benefits of MDA.