MDA Distilled

Stephen J. Mellor
Project Technology, Inc.

http://www.projtech.com
+1-(520) 544-2881
steve@projtech.com

In 2000, the Object Management Group (OMG) published "Model Driven Architecture" (OMG 2000), a white paper that described a vision for software development that relied on linking object models together to build complete systems. This **model-driven architecture** (**MDA**) approach would employ existing technologies, which support existing and future OMG standards, to support model-driven development so that object models would become assets instead of expenses.

Today, however, models are precisely that: They're expenses. Once a model has been built, it must be transformed into code, and this is a tedious, error-prone, and above all expensive process. Once the interesting abstraction work has been done much of the coding work can be automated, yet once again we find that the cobblers' children have no shoes.

MDA is the result of the recognition that interoperability is a Good Thing and that modeling is a Good Thing too. MDA allows developers to build models without knowledge of other models in the system and then combine those models *only at the last minute* to create the system. This prevents the application from becoming intertwined with design decisions; it also leaves the application independent of its implementation so the application can be recombined with other technologies, as well as other application subject matters, at some later time. This is a kind of *design-time interoperability* of models; the result is that models become *assets.*

Does this sound too good to be true? Possibly. However, MDA doesn't require a one-step leap from a code-driven process to one driven by modeling. Instead, it offers features that allow the progressive adoption of the technology. You'll likely find that some of the technologies are in place in your project already—MDA simply fits them together and organizes them into a coherent *standard* whole.

Frequently Asked Questions

We begin by answering some frequently asked questions about MDA.

What is MDA? MDA is actually three things. It is:

- An OMG initiative to develop standards based on the idea that modeling is a better foundation for developing and maintaining systems.
- A brand for standards and products that adhere to those standards.
- A set of technologies and techniques associated with those standards.

What are these technologies and techniques? There are many. The most well known is probably the Unified Modeling Language (UML), which people use to capture abstract semantics models as well as the software structure of object systems. Others include the following:

- The Meta Object Facility (MOF), a modeling language for describing metamodels
- Mapping functions
- Marking models
- Executable models and Agile MDA

MDA defines these all concepts, and more importantly, how they relate to one another.

Is that all? No. At present, MDA is still in development. Some of the technologies exist, some of the technologies need to be developed further and standardized, while others need further definition.

If they're not defined, what good are they? The ideas behind MDA have been around for years; they're only just now in the process of formalization and standardization. For example, people have been building executable models, generating code, and refining and transforming models for some years now, and gaining significant benefits from doing so. But it wasn't standard.

It takes a long time to build standards. Should I wait? No. Much of the technology has been around for a while, and you may even have been using it. It also takes time to bring a new technology into an organization, and in any case, you can do so progressively.

I've heard about MDA in an IT context. Does MDA apply to me? The principles behind MDA apply to software development in general; they aren't specific to a certain kind of software. The ideas described here apply equally to real-time and IT systems; some of them, such as Executable and Translatable UML, were developed first for real-time systems.

Now that we have an understanding of how MDA fits into the world-at-large, we take a look at the basic technologies. We begin with the flagship concept—it's a part of the name after all—models.

Models

Models consist of sets of elements that describe some physical, abstract, or hypothetical reality. Good models serve as means of communication; they're cheaper to build than the real thing; and they can be transformed into an implementation. Models can run the gamut from rough sketches to fairly detailed blueprints to fully executable models; all are useful in the appropriate context.

Central to MDA is the notion of creating different models at different levels of abstraction and then linking them together to form an implementation. Some of these models will exist independent of software platforms, while others will be specific to particular platforms.

Each model will be expressed using a combination of text and multiple complementary and interrelated diagrams. The modeling language family of choice today is the UML. The existence of this standard, (reasonably) well-defined language reduces the likelihood of misinterpretation by the viewers of models, as well as machines. In the longer term, domain-specific languages (DSLs) using the MDA framework are likely to be an important alternative to the UML.

Metamodels and Platforms

A **metamodel** is simply a model of a modeling language. It defines the structure, semantics, and constraints for a family of models. (Note that we're using the term *family* here to group models that share common syntax and semantics.)

A model is captured by a particular metamodel. For example, a model that employs UML diagrams is captured by the UML metamodel. The UML metamodel describes how UML models can be structured, the elements that they can contain, and the properties those elements exhibit. A metamodel may describe some properties of any particular platform, not just UML, and a platform's properties may be described by more than one metamodel.

A **platform** is the specification of an execution environment for a set of models. Examples of platforms include operating systems like Linux, Solaris, and Windows, the Java platform, and specific real-time platforms.

A platform has to have an implementation of the specification that the platform represents—in other words, at least one realization of it. A realization can in turn build upon one or more other platforms. A realization that stands on its own is a **primitive realization**; a realization comprised of one or more realizations is a **composed realization**. 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.

The UML metamodel is itself captured using the Meta-Object Facility (MOF), a facility specified and standardized by the OMG. This MOF meta-metamodel describes the structural and behavioral aspects of the UML metamodel, but it doesn't specify how UML models are graphically represented, or how they could be edited by multiple users simultaneously. These are the details that the MOF meta-metamodel abstracts out.

What the MOF *does* do is define how models can be accessed and interchanged, in terms of, for example, interfaces defined using the OMG's XML Metadata Interchange (XMI).

The MOF is far removed from embedded target code, but it is important in the context of MDA as a mechanism for capturing and interchanging models and metamodels.

Mappings Between Models

Models may have semantic relationships with other models; for example, when a set of models describes a particular system at different levels of abstraction. As code-driven developers, we construct one model from others by applying a set of implicit rules. Mapping functions capture this design expertise explicitly in such a manner that it can be performed automatically. It's desirable to have mappings between different but related models performed automatically.

MDA must support iterative and incremental development. This means that mappings between models must be repeatable. This makes it possible to express each aspect of a system at an appropriate level of abstraction while keeping the various models in synch. Such synchronization also enforces consistency between models.

A **mapping** between models takes one or more models as its input (these are the "sources") and produce one output model (this is the "target"). The rules for the mapping are described by **mapping rules** within a **mapping function**; the mapping is an application of the mapping

function. These rules are described at the metamodel level in such a way that they're applicable to all sets of source models that conform to the given metamodel.

For example, a mapping function that describes how to map a UML model of an application to a corresponding C source code model will have rules such as "A UML class maps to a C class declaration, where the name of the C class matches the name of the UML class." Note that this specification refers not to specific classes (elevator, cabin, etc), but to their *types* as defined by the metamodel classes.

There are several kinds of mapping, but two stand out. First, a *refining mapping* takes an abstract (analysis) model and transforms it onto one that is more oriented towards the implementation (a design model). The second major kind is a *merging mapping* that defines joins between models so that the combination can be transformed into an implementation.

These mapping rules can be expressed using a nascent MDA standard called QVT for *Queries, Views, and Transformations*.

Marking Models

A mapping rule that, for example, turns a UML attribute into a C data member on the heap, may be not always be appropriate. In some cases, there may also be a need for persistent attributes, so we need to have two mapping rules, and additional mapping inputs to select which one to apply.

These additional mapping inputs take the form of **marks,** which are lightweight, non-intrusive extensions to models that capture information required for model transformation without polluting those models. A mapping may use several different marks associated with the source models; conversely, a mark may cater to several different mappings. There also may be global marks that aren't necessarily related to individual model elements.

However, marks mustn't be *integrated* into the source model, because they're specific to the mapping, and several different mapping functions may exist, each of which requires different marks. Integrating the marks with the model would make the model specific to the corresponding mapping rules, which isn't desirable. You can think of marks as a set of "sticky notes" attached to the elements of a source model that direct the model transformer.

You can use marks in two contexts: as additional *inputs*, which you can use to anticipate design decisions or reuse design decisions across transformation functions, and as additional *outputs*, which serve as a kind of record of the transformation process from a source model to a target model.

A mark is associated with a **marking model** that describes the structure and semantics of a set of types of marks. A mapping function, therefore, specifies the marking models whose types of marks it requires on the instances of its source metamodels.

The combination of mapping functions/rules and marks/marking models constitutes a *bridge.*

Building Languages

In the normal course of development, people build languages on a regular basis. For instance, when they use a subset of the UML for "analysis" and a larger subset for "design," and when they specify what the elements of these subsets actually mean, they've defined two new languages, each with a different purpose.

There are two major reasons to seek relatively formal definitions of new languages in the context of MDA. The first is communication among team members. There needs to be agreement on whether to include things like persistence, for example, in a certain model, and on how to represent those things. The second is communication with machines. Defining languages formally allows for mappings between models expressed in those languages.

One way to define a language involves using the MOF (see "Metamodels and Platforms"). The MOF supports several important concepts that can serve as the foundation for a new language. Since the metamodel for the UML is already defined using these core concepts, it's relatively straightforward to use the MOF to define a language this way. Another way involves extending the UML via **profiles**, which are mechanisms for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. The key elements of profiles are **stereotypes**, which extend the basic vocabulary of the UML, and **constraints**, which specify conditions within a model that must hold true for the model to be "well-formed".

Note that the "definition" of a language is, strictly speaking, the *abstract syntax*, which addresses the structure of the language separated from its concrete notational symbols. Defining a graphical notation for a new language—in other words, a *concrete syntax* and notation to use in creating, editing, and maintaining the models expressed in that language—is a separate problem. Fortunately, there are straightforward ways to represent a new language graphically using the MOF and the UML; these include mapping functions and marks in addition to models and the underlying metamodel.

(Note: We do not discuss this in class.)

Model Elaboration

Model elaboration is the idea that a model can be modified after it has been generated. Generally, this means adding code to the model, but it can also mean editing the generated model itself. This possibility of elaboration of target models is an advantage of the MDA framework because it allows developers to ease into model-driven development, rather than take a step function from a code-driven process to a model-driven one.

To get the most out of model elaboration, it's important to follow certain principles:

- Don't elaborate a model if you don't have to.
- Don't elaborate "intermediate" models that aren't meant to be exposed
- Localize elaboration and avoid redundancy in elaborating locations.

If it's done carefully, elaborating models can be a perfectly acceptable practice in the context of MDA.

Of course, the primary manner in which people will want to elaborate models involves adding code to them. When the target model is regenerated, one needs to be certain that this added code

isn't replaced by the regenerated code. A simple approach is the concept of protected areas: If an area of the source model is protected, the mapping function can preserve the manually-entered contents. Detecting manual changes in target models, preserving/ merging manual changes during mapping, and avoiding the loss of manually-created information are critical success factors in this area.

Executable Models

The next logical step is to **executable models**, which are complete in that they have everything required to produce the desired functionality of a single domain. These models are neither sketches nor blueprints; as their name suggests, models run. This allows us to deliver a running system in small increments in direct communication with customers.

Executable models act just like code, in a sense, though they also provide the ability to interact directly with the customer's domain, which is something code doesn't do well. They're not exactly the same as code, though, because they need to be woven together with other models (for example, a meaningful user interface) to produce a system. This is generally done by a **model compiler**. As each model is complete in itself, though, once the weaving is done, the system is complete.

Just as programming languages conferred independence from the hardware platform, executable models confer independence from the software platform, which makes executable models portable across multiple development environments. Contrast this with adding code bodies to models. Such code bodies are inherently dependent on the structure of the platform for which the code is intended.

One way to express executable models involves the use of **Executable UML,** a profile of the UML that defines an execution semantics for a carefully selected streamlined subset of the UML. The subset is computationally complete, so an executable UML model can be directly executed. Executable UML defines groupings of data and behavior ("classes"), the behavior over time of instances ("state charts"), and precise computational behavior ("actions") *without* prescribing implementation.

Agile MDA

Agile MDA is based on the notion that code and executable models are operationally the same. It employs executable models so they may be immediately tested and verified by running them, which provides for immediate feedback to customers and domain experts from running models—a key feature of agility.

The need for immediate feedback is a cogent criticism of models-as-blueprints. When models don't run, there is no way to verify that a model is correct or even in the ballpark.

Agile MDA addresses the potential conflict between MDA and agile methods, which propose to address the problems associated with the "verification gap" (which comes about when one writes documents that can't be executed) by delivering small slices of working code as soon as possible. This working functionality is immediately useful to the customer, who can interact with it; this might result in improved understanding on the customer's part of the system that needs to be

built. As these delivery cycles can be short (say, a week or two), the systems' development process is able to adapt to changing conditions and deliver just what the customer wants.

In Agile MDA, each model necessarily conforms to the same metamodel, because all models are equal—there are no "analysis" or "design" models. Models are linked together, rather than transformed, using mapping functions, and all of them will then be mapped to a single combined model that is subsequently translated into code according to a single system architecture.

Building an MDA Process

After all of the discussion of the various aspects of MDA, at some point, one needs a process. We started this paper by talking about models, which of course are at the heart of MDA. At the heart of defining an MDA process, by extension, is the identification of these models.

Think of the gap that separates a problem statement from a coded system. If the gap is narrow enough, you can simply hop across, but if it's wider, you need some stepping-stones. The stones represent the models you select; each step from one stone to another represents a mapping function. The path from one side to the other constitutes a particular **mapping chain** for this project. Deciding where to place the stones, and planning the journey from one side to the other, constitutes a definition for an **MDA process**. The selection of the models and the mapping functions between them must fit together to form the specific process you apply on your MDA project.

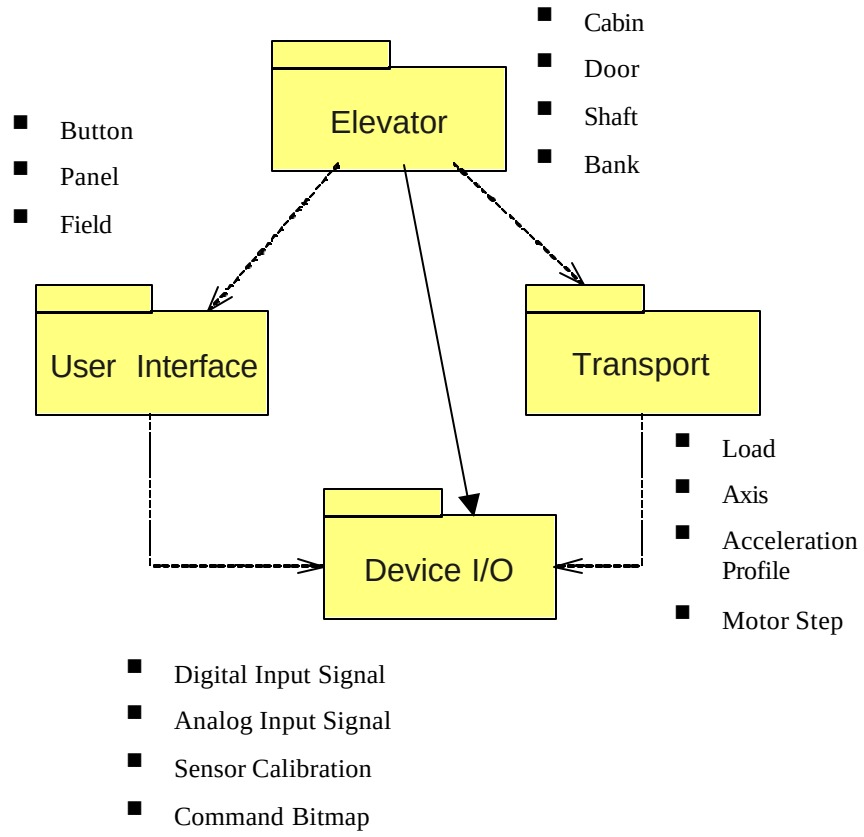
The best approach to building an MDA process involves a combination of two approaches: (1) finding models that exist at a single level of abstraction, and (2) finding an optimal length for each "hop" in the chain. A "single-hop" approach tends to simplify the mapping functions and exposes an intermediate model (which sits between the source model and the target model) so the mapping functions can be reused. The choice of intermediate domains may also depend on what is available for reuse (possibly from third parties).

An effective approach to finding which models to build is to divide up the system into independent subject matters, or **problem domains**. These subject matters can be displayed on a **domain chart,** which shows the domains and the bridges—in other words, the mapping functions, marks, and marking models—between them. The domain chart forms the basis for defining the MDA process for your project, as shown in the figure on the next page.

Executing an MDA Process

Once one has an MDA process in place, one naturally wants to execute it. Broadly speaking, this comes down to two main activities. (1) formalizing knowledge of a subject matter and then rendering that knowledge as an implementation, and (2) mapping that formalized knowledge onto a target platform that can execute.

Knowledge formalization, in the context of MDA, involves, as you might expect, gathering requirements relevant to the domain of interest, abstracting that knowledge into some set of concepts, and then expressing those concepts formally in a model. What MDA brings to the table is the concept of testing the model for correctness—preferably by executing it.



Once the models start coming together, the next step is to build bridges among them. This involves specifying and verifying mapping functions, building marking models, and then transforming the models. Once the models are marked, and the mapping function specifications are complete, one can transform the formalized, marked, and verified knowledge into other models or source code comprising the system's implementation.

The result of executing the MDA process is a system. Whoo hoo!

Acknowledgements and Credits

This paper was derived from the Preface and Chapter 2 of MDA Distilled, by Mellor, Scott, Uhl and Weise, published by Addison-Wesley in 2004 (Mellor et al., 2004) with permission of the authors. The Bibliography was ripped off wholesale. The domain chart of the sole figure was conceived by Leon Starr.

References

(Mellor and Balcer, 2002) Stephen J. Mellor and Marc J. Balcer: *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.
(Mellor et al., 2004): Stephen J. Mellor, Kendall Scott, Axel Uhl and Dirk Weise: *MDA Distilled: Principles of Model-Driven Architecture.* Addison-Wesley, 2004.

Further Reading

(Atkinson, 2003) Atkinson and Kühne: "Model-Driven Development: A Metamodeling Foundation," *IEEE Software,* September/October 2003.

(Beck, 2000) Kent Beck: *Extreme Programming Explained*. Addison Wesley, 2000.

(Bock, 2003) Conrad Bock: "UML without Pictures," *IEEE Software*, September/October 2003.

(Frankel, 2003) David S. Frankel: *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

(Garlan, 1994) David Garlan: *An Introduction to Software Architecture*. Research Access Inc., 1994.

(HUTN, 2002) Human Usable Textual Notation, revised submission, Object Management Group, 1 April 2002; **www.omg.org/cgi-bin/doc?ad/2002-03-02**

(Kleppe, 2003) Anneke Kleppe, Jos Warmer, and Wim Bast: *MDA Explained: The Model Driven Architecture—Practice and Promise*. Addison-Wesley, 2003.

(Selic, 2003) Bran Selic: "The Pragmatics of Model-Driven Development," *IEEE Software*, September/October 2003.

(Stachowiak, 1973) Herbert Stachowiak: *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

(Weis *et al*, 2003) Torben Weis, Andreas Ulbrich, and Kurt Giehs: "Model Metamorphosis," *IEEE Software,* September/October 2003.