The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture

Dean Wampler, Ph.D.

Aspect Programming, Inc.

A fundamental challenge of software development is bridging the vision of an application to its realization. In practical terms, if we capture the vision in use cases and requirements, without the distractions of implementation details, how then do we generate conforming implementations on real platforms, while accounting for platform idiosyncrasies, limitations, and other constraints? Despite years of research, this generation process remains difficult. Fortunately, there are emerging technologies that bring software generation closer to reality.

Model-Driven Development is one approach to this challenge, where the problem domain is modeled at a high level of abstraction and the implementation is derived from these models. The Object Management Group (OMG) has articulated a particular vision of MDD called Model-Driven Architecture [1]. MDA will include standards for modeling problem domain specifications and mapping techniques for generating implementations from them.

However, the challenge for MDA and other MDD variants is to define all these standards and to make them work on real projects. This article describes how a new methodology, Aspect-Oriented programming (AOP) [2], helps make this possible.

What Is MDA?

In MDA, business processes and applications are specified using platform-independent models (PIM's) that define the required features at a level of abstraction above the details of possible implementation platforms. Standardized mapping techniques transform the PIM's into platform-specific models (PSM's) and ultimately into implementations. For example, standardized mappings are being defined for the .NET and J2EE platforms.

OMG uses UML [3] as the modeling language and hence most mapping approaches exploit OMG's Meta-Object Facility (MOF).

Models of *pervasive services,* such as security, transactions and persistence, are being developed, along with specialized models and transformations for a number of vertical

industries, such as telecom and finance. Both efforts will encapsulate industry standards and best practices for widespread reuse.

Conceptually, there can be several vertical levels of PIM's and PSM's, each of which is the product of a mapping from a higher-level model. The highest-level PIM is essentially the *analysis* model, as described in methodologies like the Rational Unified Process (RUP) [4], [5]. It represents what I will call the *core business logic*, the unique abstractions of object structure and behavior, divorced from all implementation concerns. Mappings to one or more PIM's at lower levels might incorporate specifications for vertical-industry domain knowledge and pervasive services, but still in a platform-independent way.

The PSM's might also include several levels, starting with high-level architectural models, followed by lower-level design models, all the way down to the implementation constructs.

Traversal from one level to the next lower level occurs through a mapping. The inputs at each step consist of one or more input models and optional parameters that configure the behavior of the mapping.

Figure 1 shows a conceptual example of MDA with a two-level PIM and a two-level PSM. The top-level PIM represents the core business logic for an online bank. It is passed through a mapping that incorporates security constraints to create a second-level PIM. This PIM is then passed through a J2EE-specific PSM mapping to create a J2EE design-level PSM. Finally, this PSM is mapped to the second-level PSM, the actual source code. Note that in principle, the build process that creates compiled Java byte code and deployable Java archives ("jar" files) could be considered another mapping and the build products could be considered a third-level PSM (not shown). To create a .NET version of the application, a PIM-to-.NET PSM mapping is substituted for the PIM–to-J2EE PSM mapping.

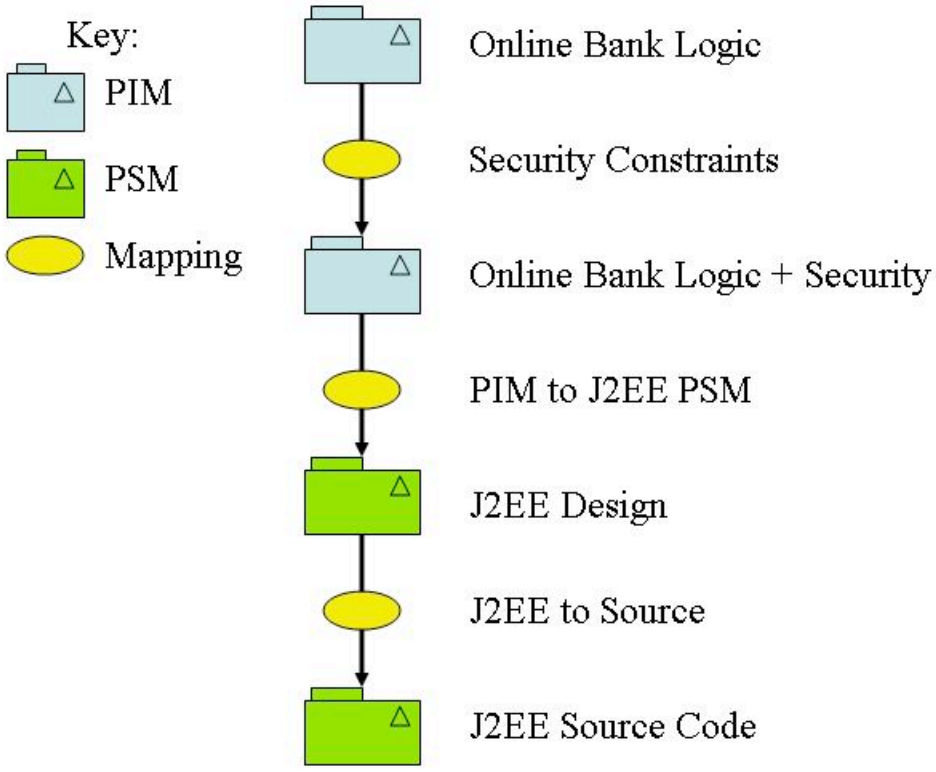

Figure 1 – An MDA Example Showing Layers of PIM's and PSM's

Parts of this vision exist today, of course. Many Rapid Application Development (RAD) environments generate applications from high-level specifications that are often defined in a proprietary fashion. RAD tools are most suitable for somewhat constrained problem domains and when rapid delivery is paramount.

Some UML model-to-model transformations have been defined [3]. For example, a standard UML representation of EJB's exists [6]. Also, Rational XDE™ [7] has a flexible "patterns" engine that can be used to implement UML model-to-model transformations and to save them as reusable assets.

If achieved, MDA will make it possible to create implementations on one or more platforms using a single platform-neutral specification. Furthermore, the expertise of implementing applications on particular platforms and in some vertical industries will be encapsulated in models and platform-specific mappings, which will make this expertise easier to share and reuse.

The Challenge for MDA

While the vision of generating applications from higher-level models is not new [1], [8], the majority of applications are still developed by writing platform-specific source code,

often with little or no modeling. This gap between the vision of MDD/MDA and the reality of current practice reflects several fundamental issues that MDA must address.

- Modeling and mapping constructs must be expressive enough to specify the application completely and unambiguously.
- Model-to-Model mapping technology must be comprehensive, robust, and suitable for automation.
- The MDA development process must be efficient enough to entice developers away from hand-written source code.
- The MDA development process must satisfy real world concerns including concurrent (team) development, configuration management, maintenance, and evolution.

Modeling constructs, combined with configurable mapping processes, must be expressive enough to specify the application in enough detail that the implementation can be generated from the PIM's with minimal manual intervention. Complete automation isn't essential for success, but the attractiveness of MDA will diminish if extensive manual intervention is required.

While UML is expressive enough in principle to fully specify applications, in practice many developers find that it is still faster to work with source code directly at lower levels of detail. MDA advocates must either address this perception or embrace source code as a PSM that is integral to the process.

Model mapping must generate complete and robust models from higher-level models. The PIM-to-PSM mapping process is particularly challenging, because the resulting PSM must closely match what a domain expert would create manually. Developers won't use these PSM's unless they meet minimum standards for quality, efficiency, legibility (for software inspection), *etc*.

A practical challenge for mapping is maintaining traceability between models and using it for automatic synchronization of changes between models. One-time mapping between models is not difficult, but changes are inevitable in iterative development and they must be synchronized between all the affected models. Manual synchronization is difficult and time consuming. In most real projects, it is quickly abandoned, thereby undermining the value of modeling. Furthermore, concurrent changes made by team members must be differenced and merged reliably. In fact, these practical issues are probably the most important reasons why MDD has seen limited adoption, to date.

Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming [2], [9] is an emerging family of techniques which address a central limitation of Object Oriented Programming, namely the breakdown of modularity that results when the core business logic is combined with other "concerns",

such as security, data persistence, transactions, high availability, logging, error handling, *etc*. as part of the implementation process.

OOP is a success because it effectively decomposes the core business logic into minimally-coupled modules and classes, which can be developed separately and then combined to create applications. This is a "divide and conquer" strategy.

The core business logic is represented by an object structure. However, additional logic for the other concerns cuts across the boundaries of the core objects. Therefore, these concerns are called *crosscutting concerns*. (Much of the terminology comes from [10].)

Hence, in the source code of an application, the pristine object model is often obscured by code that implements the crosscutting concerns. This *tangling* of code reduces modularity, increases complexity, and impacts quality. Maintenance becomes difficult because modifications in crosscutting concerns must be made across many code modules. In short, crosscutting concerns undermine the benefits of OOP.

The vision of AOP is to restore overall modularity by extracting the crosscutting concerns, modularizing them as *aspects*, and then *weaving* them back into the application at well defined locations, called *join points*.

Aspects can be discussed for applications that are not object-oriented, but AOP and OOP complement each other nicely. In a sense, AOP is a "superset" of OOP. The core business logic and the aspects can be developed separately using OOP techniques, suitably extended with AOP concepts. Then, AOP describes how to weave the logic and aspects together to create the application. OOP models provide a number of natural join points into which aspects can be woven.

Aspects appear in all levels of the development process. For example, security, persistence of data, transactions, and high availability concerns appear at the requirements level all the way down to the implementation. Conversely, logging and language-specific error handling are implementation concerns. AOP is a useful tool for quality-assurance issues. White-box testing can be implemented using test aspects that are woven into applications to drive test cases, inject faults, examine program state, *etc*.

Clearly an essential facet of AOP is the ability to weave aspects into applications, hopefully in an automated fashion. Automation is important for maintaining the modularity of the aspects. Without automation, changes to an aspect would require manual "reweaving", a tedious process that developers would quickly abandon.

AOP is still relatively new and evolving. It is more mature in the implementation arena; see for example AspectJ [10] for Java. For a list of AOP projects and tools, see [9].

How AOP Helps MDA

As stated before, to get from vision to reality, MDA must define comprehensive modeling standards and mapping technologies. AOP helps meet this challenge with its modularization and weaving concepts.

AOP Allows Crosscutting Concerns to Be Developed Independently

With AOP, the core business logic and the cross-cutting aspects can be developed as separate, modular PIM's and PSM's. They are woven together at the appropriate points to generate other PIM's and PSM's. The MDA proposals for *pervasive services* are ideal candidates for aspects and the models for vertical industries are ideal frameworks upon which to implement the core business logic. In fact, it is likely that parts of these frameworks should also be developed as aspects.

For example, in a financial framework, regulatory constraints could be implemented as aspects that are interleaved into core processes. The constraints would programmatically ensure compliance and the framework would remain adaptable to evolving laws through modification of the corresponding aspect modules.

Without incorporating AOP, it will be difficult to define PIM standards and mappings that properly and effectively handle these divergent concerns. AOP provides a natural divide-and-conquer strategy for these large and diverse problem spaces.

AOP Specifies How Aspects Are Woven Together

A key promise of AOP is the ability to weave aspects and the core business logic together. This capability aligns with MDA's need for processes to combine input PIM's and to generate PSM's from them.

In a nutshell, AOP handles the *horizontal* decomposition of the problem domain, design space, and the implementation into aspect-oriented models, as well as the subsequent weaving together of these models within the same level. OOP is the most logical choice for modeling *within* a single aspect model and OOP technologies like MOF provide the tools for *vertical* mapping of a model from one level to another level.

However, to fully exploit AOP in MDA, AOP technology for higher-level models needs further definition. Possible approaches include UML enhancements [11], stratified frameworks [12], and use-case oriented approaches [13], among others [9].

The language-level approach of AspectJ suggests how higher-level AOP might work [10]. AspectJ provides a way of defining the source code that implements an aspect, called an *advise*, and where to insert it in the target source code. The target points are defined by a *pointcut,* which is a collection of logically-related *join points,* such as packages, classes, method calls, return statements, and variable modifications. ([10] defines the terms in greater detail). The AspectJ compiler "ajc" inserts an *advise* in the

target source code at the *join points* in the *pointcut*. (Byte code weaving is under development.)

UML also contains *join points*. In use cases, the extension mechanism, «extend» [4], [13], preconditions and postconditions, and scenario steps are *join points*. UML models have many of the same *join points* as source code. For less structured information, like textual requirements and use case descriptions, it is harder to weave them together using automated tools. However, mapping them to lower-level models and then weaving those models together is probably a better strategy anyway.

Techniques for OOP-based vertical mapping, such as MOF-based transformations [1] and patterns [7] are more mature, although more work is needed on the practical issue of traceability, as discussed previously.

Aspect-Oriented MDA

Let's reconsider the earlier MDA example, an online banking application with PIM's and PSM's in *vertical* levels. The top-level PIM specified the core business logic and the second-level PIM added security concerns. Using AOP, those *vertical* PIM levels are best refactored into *horizontal* PIM's, which together constitute the complete specification.

Ideally, all the necessary information can be specified in the set of top-level PIM's and the complete application can be generated automatically. An alternative and more realistic strategy is to do aspect weaving at different levels of the PIM/PSM hierarchy.

Figure 2 illustrates this strategy of using AOP in MDA, using a revised version of the initial example. Now, the online banking application logic and the security aspect are separate high-level PIM's. For illustration purposes, additional PIM's have been added for transaction and persistence aspects.

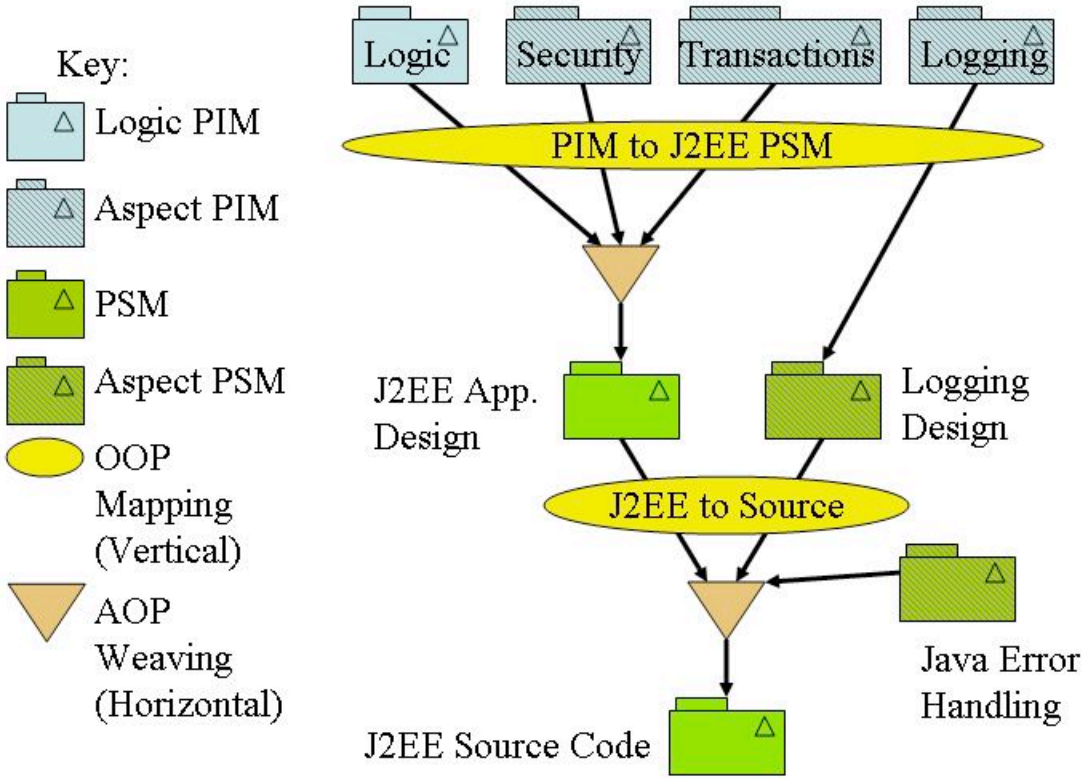

Figure 2 – A Reworked MDA Example Using AOP

For illustration purposes, the PIM's are mapped to J2EE Design PSM's and then all but the persistence PSM are immediately woven together to get a J2EE Application Design PSM and a Persistence Design PSM. Finally, the two design PSM's are mapped to source code and woven together, along with a Java-specific Unit Testing PSM.

The PIM's are standard UML models and other artifacts that are driven by the use cases and requirements for the "domains" they cover. The Logic PIM represents the business logic classes and behaviors. The Security, Transactions, and Persistence PIM's are aspect-oriented PIM's that capture industry best practices for these concerns, expressed in UML and applicable to any enterprise-class application. They are excellent candidates for standardization and reuse.

The design model for the Persistence PIM is kept separate from the merged J2EE Application Design model to suggest that some aspect PIM's will need further customization and elaboration at the PSM level. Indeed, not all aspects are best captured as PIM's. Unit testing tends to be very language and application specific, so the Unit Testing PSM shown is a reusable aspect model of Java-specific constructs.

The OOP in AOP

The PIM's and PSM's in the two figures are shown here using the UML "folder" notation for models. However, these folders might contain multiple UML models for use cases, object models, *etc*. The aspects and the core business logic are modeled using mature OOP technologies, like UML, with extensions to represent aspect-oriented constructs.

These extensions are the subject of active research (see for example [11]). They must cover modeling AOP concepts themselves and the weaving of aspects together with other models. The details are beyond our scope here, but a few general comments are in order.

Use Cases and Iterative Development

It is interesting to note that *use cases* loosely fit the definition of aspects! A use case describes how an external *actor* uses the application to achieve a particular goal. When realizing a use case in models, the use case typically cuts across the boundaries of the model elements. In other words, a single use case is realized by several collaborating objects and each object may help realize several use cases. This suggests that AOP may provide a way to solve a long-standing problem: how to map the set of use cases to the design and implementation models in a robust and "algorithmic" fashion. Normally, this is done manually, but maintaining traceability through the application lifecycle is difficult, so it is hard to ensure that each use case is properly realized. If each use case is treated like an aspect and modeled separately, then perhaps aspect weaving can be used to combine the use case models to create a complete application PIM. Notice that now the "core business logic" is really a set of aspect models.

If so, this has important implications for iterative development processes, like RUP [5] and Agile Methods [14]. Both emphasize delivering high priority use cases (or *user stories* in AM parlance) in early iterations and lower-priority use cases later. If an AOP approach to use case development makes it easy to weave together models for different use cases, then it reduces the temptation to overbuild early use case realizations in anticipation of future needs. Instead, the developer can focus on implementing one use case at a time and composing the application through weaving.

Common Architectural and Implementation Aspects

Some very common aspects include security handling, high-availability and fault tolerance, transactions, and persistence of data. They are re-implemented repeatedly and unnecessarily. In AOP, they can be modularized and developed separately as PIM's and PSM's.

Analysis, Design, and Implementation Patterns [14, 15] are an important source of modularized, reusable expertise. Unfortunately, with the exception of a few tools (for example, [7]), developers usually have no automated way to incorporate these assets into their code. AOP reformulations of patterns [16] address this problem and also provide a new approach to the development of reusable assets.

Similarly, implementation-level aspects include error handling strategies, where language-specific constructs for exception handling need to be considered, logging of application and object state, and quality improvement strategies such as Design by Contract [17]. Aspects for unit testing and quality assurance were discussed previously. Note that aspect weaving allows debugging and testing aspects to be inserted during development, but removed for production builds.

Once modularized, all these aspects can be modeled to capture the industry best practices. The models make this knowledge accessible to a wider developer community as reusable assets.

Conclusions

Aspect-Oriented Programming is a powerful new tool that improves program modularity in ways where Object Oriented Programming is weak. In AOP, the different concerns in real applications are modularized, developed separately, and then woven together to create applications. As such, AOP addresses a fundamental problem faced by the Model-Driven Architecture, how to define separate models for concerns and domains, combine those models, and finally generate applications from them.

References

[1] Object Management Group, "The Architecture of Choice for a Changing World." Available http://www.omg.org/mda/.

[2] Tzilla Elrad, *et al.*, Special Issue on Aspect-Oriented Programming", *Communications of the ACM*, vol. 44, issue 10, Oct. 2001.

[3] Object Management Group. "UML Resource Page." Available http://www.omg.org/uml/.

[4] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Engineering*, ACM Press, Addison-Wesley, Wokingham, England, 1992.

[5] Rational Software, "The Rational Unified Process." Available http://www.rational.com/products/rup/faq.jsp.

[6] Dean Wampler, *et al.*, "UML/EJB Mapping Specification", Java Community Process. Available http://www.jcp.org/en/jsr/detail?id=26.

[7] Rational Software, "Rational XDE™: Liberated Development." Available http://www.rational.com/xde/.

[8] K. Czarnecki and U. W. Eisenecker, *Generative Programming*, Addison-Wesley, Boston, Mass., 2000.

[9] Aspect-Oriented Software Development Steering Committee, "Aspect-Oriented Software Development." Available http://aosd.net/

[10] AspectJ Development Team, "AspectJ Project." Available http://www.eclipse.org/aspectj/.

[11] Siobhán Clarke, "Extending Standard UML with Model Composition Semantics", *Science of Computer Programming*, vol 44, issue 1, July 2002, pp.71-100. Also available http://www.dsg.cs.tcd.ie/~sclarke/ThemeUML/.

[12] Colin Atkinson and Thomas Kühne, "Aspect-Oriented Development with Stratified Frameworks", *IEEE Software*, vol. 20, no. 1, Jan./Feb. 2003, pp.81-89.

[13] Ivar Jacobson, "Use Cases and Aspects – Working Seamlessly Together", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 7-28.

[14] R. Martin, J. Newkirk, and R. Koss, *Agile Software Development*, Prentice-Hall, Upper Saddle River, New Jersey, 2003.

[15] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, Massachusetts, 1995.

[16] Jan Hannemann and Gregor Kiczales, "Design Pattern Implementation in Java and AspectJ", *Proceedings of OOPSLA 2002,* pp. 161-173.

[17] Eiffel Software, "Building bug-free OO software: An introduction to Design by Contract™." Available http://www.eiffel.com/doc/manuals/technology/contract/.

Acknowledgments

The author gratefully acknowledges helpful feedback from Ivar Jacobson, Grant Larsen, Grady Booch and Jochen Seeman.

