



Aonix

Architecture Component Development – MDA based Model Transformation

Ameos™ White Paper

Michael Benkel
Michael.Benkel@aonix.de



Abstract

The Unified Modeling Language (UML) provides a wide range of graphical notations and textual means to describe a software system in an abstract way. Depending on the architecture of the system, the detailed design is often complex and difficult to maintain. The Model Driven Architecture (MDA), OMG's latest approach to improve the software engineering process, helps to solve this problem. Shifting the focus of development from source code to the model will result in better software quality, faster iteration cycles and higher productivity. Although the "brand name" MDA is relatively new, the engineering problems that are being addressed have been known for a long time and various approaches have been taken to solve them. We can draw from those experiences to evaluate how to make the best use of MDA.

Degrees of abstraction

In the history of software engineering, an increase in complexity has always been addressed by finding more abstract means of specifying software. This is reflected in the history of programming languages. Assembler was introduced to remedy the intricacies of writing machine code. Higher level languages were introduced to hide the details of the underlying processor from the programmer. 4GL languages were invented to spare a lot of repetitive tasks in information systems. Figure 1 shows the increasing levels of abstraction.

Each step in this process of raising the level of abstraction requires two basic elements: a new language of describing the next higher abstraction with the accompanying semantics and an efficient transformation. The latter is required to convert or compile a program description in the new higher level abstraction to an existing abstraction closer to the target hardware. With UML and Model Driven Architecture (MDA) the OMG provides the standards to achieve this. UML is the current candidate for the next abstraction level and MDA provides the semantics and the transformations necessary to compile UML to program code.

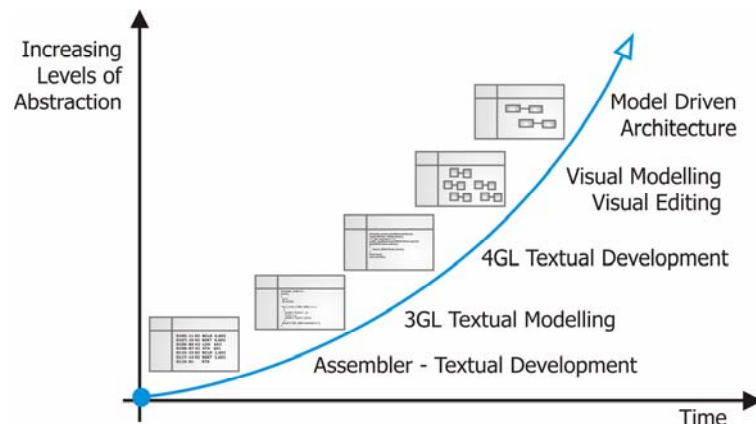


Figure 1: Increasing Levels of Abstraction

UML as defined by the OMG is the standard notation today. It provides several diagrams for modeling the different aspects of a system. Nevertheless the detailed meaning of the diagrams and the semantics are application-specific. This means that UML can be used to model a wide range of software systems, but very often an extension to the standard notation is needed to add additional semantics. The standard UML extension mechanisms are profiles which mainly consist of predefined stereotypes and tagged values.

MDA provides the semantics and the transformations necessary to compile UML to program code. Each MDA platform requires a UML Profile as its semantic underpinnings and it has an impact on the model transformation to the target system.

Visual modeling provides a lot of benefits compared to traditional development. UML helps to improve communication in the project team and to provide a clear interface to other stakeholders in the project. Architectural problems are detected much earlier in the development process. Therefore UML based projects are much easier to maintain. Today many developers spend most of their time in maintaining existing software systems instead of implementing new systems.

MDA is the next logical step in this evolution. The building of systems can be organized around a set of models by imposing a series of transformations between models.

MDA Overview

MDA modeling starts on a higher level of abstraction with the creation of a Platform Independent Model (PIM). A PIM is always focused on the domain of the project and mostly UML is used as the modeling notation. In the next step the PIM is mapped to one or more Platform Specific Models (PSM) by adding technical aspects. This mapping is done by a transformer and transformation rules implementing technical patterns. This PSM is then mapped to the implementation by using the same mechanism. Implementation patterns drive the transformation of the model to the target environment. Figure 2 gives an overview of MDA and the transformation rules.

In terms of modeling, UML as defined by the OMG is the standard notation today. It provides several diagrams for modeling the different aspects of a system. Nevertheless the detailed meaning of the diagrams and the semantics are application-specific. This means that UML can be used to model a wide

range of software systems, but very often an extension to the standard notation is needed to add additional semantics. The standard UML extension mechanisms are profiles which mainly consist of predefined Stereotypes and Tagged Values. UML 2.0 describes Profiles and defines how to model them in UML notation.

The MDA approach allows the separation of concerns in the project and therefore a big ROI on intellectual properties that is now captured in platform specific models. Another goal of MDA is to make the reuse of design models easier. Since platform dependencies are added later, the same design model can be used in many different settings. Platforms can be much more than just the underlying processor or operating system; rather they can be entire environments, such as middleware, component platforms, or libraries, thus adding to the flexibility of the design model.

Required Tool support

Each MDA platform requires a UML Profile as its semantic underpinnings and it has an impact on the model transformation to the target system. Therefore it is important to describe and document Profiles properly. This requires a specific editor to define new Stereotypes and Tagged Values, assign them to an element of the UML metamodel and to reuse these definitions easily in the user's model.

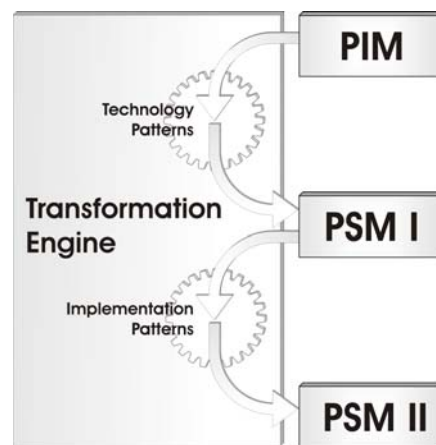


Figure 2: MDA Overview

The used profiles also have an impact on the model transformation and the refinement of the model. This refinement of the model is a critical point in the project that will decide whether the model and generated code will be consistent or if they will diverge. There is a major flaw in many modeling tools:

- The code generation is too fine-grained. Each and every attribute that is going to be generated has to be added to the model manually, otherwise it will not show up. This is a manual transformation from high-level to low-level design and involves a lot of tedious and error-prone work. The manual labour also increases the cost of infrastructure changes. But the most costly disadvantage is the loss of abstraction.
- The code generation is hard coded and not customisable. Most code generators will allow you to set hundreds of options – but as Murphy 's Law strikes, the option that you need won't be there. This kind of code generator is expensive, if not impossible to modify. But since we want a running system in the end, generating code from the model is a must to reduce costs.

Traditional code generators give you a one-to-one mapping from the design model to code. For each class that is represented in the model there is one class specified in the code. Hence, more implementation code is needed to complete your system. So, to get this extra implementation code, you must either represent extra details in your model or input the code manually. This leads to models which are constructed to achieve maximum code generation, rather than accurately representing the business or user requirements in a maintainable way.

What is really needed is a powerful transformation engine that can be seen as a model compiler. This allows the mapping from higher level descriptions in UML notation to existing abstractions closer to the target system. Modeling can then be done on a higher level of abstraction with all the benefits described in the previous chapter. Nevertheless it is crucial for the acceptance of a modeling approach that the transformation rules can be easily adapted to project specific needs.

Ameos and MDA

Ameos is a complete UML and MDA environment. It provides full UML support, a specific UML 2.0 Profile Editor and a MDA transformation engine named Architecture Component Development (ACD).

Profiles are an easy way to extend standard UML and are widely used in the Market. Extensions mainly consist of domain specific Stereotypes and Tagged Values to adapt UML to project specific needs. UML 2.0 describes Profiles and defines how to model them in UML notation. The Ameos Profile Editor allows stereotypes and tagged values to be defined and assigned to model elements of the UML Metamodel, ensuring that profiles are well designed, documented and easy to use for the entire project team.

The example in Figure 3 shows the definition of a profile. The Stereotypes HIApexBlackboard and HIApexBuffer are assigned to the UML Meta-Class UMLAssociation. Based on this definition, these two Stereotypes can only be assigned to UML notation elements of type UMLAssociation in the users model. Depending on the used stereotype an association is implemented by additional classes of type Blackboard or Buffer.

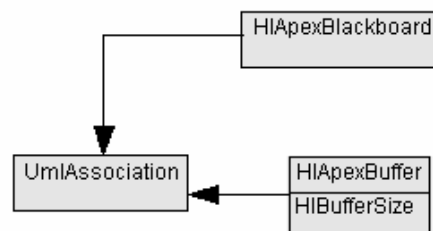


Figure 3: Profile Example

The class diagram in Figure 4 illustrates how defined profiles are applied. This example consists of two independent classes (tasks) and a communication mechanism. Both classes are periodic threads with certain priorities. To exchange data between these two threads we use a “buffer” structure, so that they can send and receive messages.

‘FireSensor’ writes data into a circular buffer of fixed size and ‘AlarmManager’ reads data from it in a FIFO order.

Using the profile definition in Figure 3 it is very easy to model the communication pattern. To use a buffer as the communication mechanism, the association between two tasks is marked with the stereotype <<HIApexBuffer>>.

The association class Alarm is used to define the type of information exchanged between the two classes.

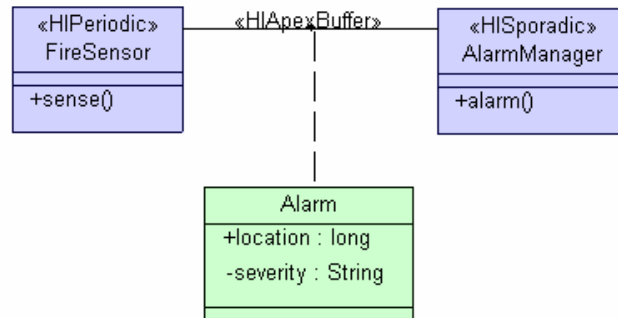


Figure 4: Buffer Example

Instead of employing a one-to-one mapping from model to code, the ACD approach allows the level of abstraction in UML models to be raised. These domain-specific models can then be mapped to the target system and ACD templates generate the technical infrastructure. These types of models are much easier to understand and to maintain than traditional UML Models.

ACD Overview

ACD is a model compiler that provides a more powerful and flexible approach for automatically generating solutions from your Ameos/UML models. Using ACD, a majority of the implementation code for your application can be generated automatically from your model. ACD implements a template-based approach that allows you to specify the architectural aspects of the system under construction. To do this, it provides a specific Meta Model and a transformation language named Transformation Description Language (TDL).

The ACD UML metamodel is based on the standard Object Management Group (OMG) metamodel. It comprises a set of rules for ACD on how to read UML models and extract information from them to generate code. The metamodel also provides the paths and directions you need when creating a new template.

The transformation language TDL is a simple, declarative template language and enables access to the UML modeling elements. It is much simpler but also much more abstract than the scripting languages used by current visual modeling tools. Compared to code generation script implemented in Visual Basic only 1/10 of the lines of code in TDL are necessary.

All templates start with the keyword “template” followed by the template name and end with “end template”. All lines between template delimiters are written literally to the output. This means that anything you write is written directly to the output file, e.g. ‘protected void set’ in the example below. Everything enclosed by square brackets [] is replaced by the modeling information, like the name of the attributes or the data type.

The following example shows the implementation of a pattern to generate set() and get() methods.

```

template genAccessMethod(MClass)
[loop(MClass->MAttribute)]
[/* get operation */]
public [getDataType([MAttribute.type])] get[MAttribute.name]()
{
    return [MAttribute.name];
}
[/* set operation */]
protected void set[MAttribute.name]([getDataType([MAttribute.type])] value)
{
    [MAttribute.name] = value;
}
[end loop]
end template

```

Although this get/set pattern is very simple and most code generators implement it to some degree, it is a good example to demonstrate the power of Ameos TDL. The template is easy to understand, the indentation of the generated code is handled by the transformation engine and the template can easily be adapted to project specific needs.

Let's assume there are some coding guidelines to implement, for example, the visibility of access methods should be the same as the visibility of the attribute and there should always be an uppercase letter after get and set in the method name. There are only a few modifications to the existing templates, in [blue](#), that are required to implement these guidelines

```

template genAccessMethod(MClass)
[loop(MClass->MAttribute)]
[/* get operation */]
[MAttribute.access] [getDataType([MAttribute.type])] get[string_capitalize(MAttribute.name)]()
{
    return [MAttribute.name];
}
[/* set operation */]
[MAttribute.access] void set[string_capitalize(MAttribute.name)]([getDataType([MAttribute.type])]
value)
{
    [MAttribute.name] = value;
}
[end loop]
end template

```

In this way, it is possible to generate a large part of the source code automatically from the UML metamodel. Most traditional code generators just generate code from static class diagrams. The example source in Figure 5 shows the access methods for the two attributes of class Alarm.

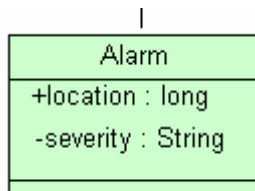


Figure 5: Getter/Setter Example

```

public class Alarm {
    // -----
    // instance attributes
    // -----
    private long location;
    private String severity;
    ...
    //Automatically generated <code>set</code>-method
    public void setLocation(long location) {
        this.location = location;
    }
    //Automatically generated <code>get</code>-method
    public int getLocation() {
        return Location;
    }
    //Automatically generated <code>set</code>-method
    private void setSeverity(String severity) {
        this.minute = minute;
    }
    //Automatically generated <code>get</code>-method
    private String getSeverity() {
        return Severity;
    }
    ...
}

```

The listing shows the automatic generated 'set' and 'get' operations based on the tdl template above. Although attribute 'location' is public in the model, it shows up as private in the source code, but the get and set methods for 'location' are public. That is how the coding guidelines are implemented in the transformations rules.

Another TDL template implements a communication pattern for a buffer. The listings in figure 6 show this template on the left hand side and the implementation in Java on the right.

<pre> template BufferAssoc (MAssociation) [loop (MAssociation->MAssociationEnd As FromRole->MClass As SenderCI)] [loop (MAssociation->MAssociationEnd As ToRole->MClass As ReceiverCI Where [ToRole.id] != [FromRole.id])] import com.aonix.hidoors.tools.ApexBuffer; public class [SenderCI.name][ReceiverCI.name]Buffer { [loop (MAssociation->AssociationClass As MessageCI)] // New buffer queue [loop(MAssociation->TaggedValue As TV Where [TV.tag] == "HIBufferSize")] private ApexBuffer queue = new ApexBuffer([TV.value]); [end loop] // send and receive methods public [MessageCI.name] receive() { return ([MessageCI.name] queue.get()); } public void send ([MessageCI.name] element) { queue.put(element); } } [end loop end loop end loop] end template </pre>	<pre> import com.aonix.hip.tools.ApexBuffer; public class FireSensorAlarmManagerBuffer { // New buffer queue private ApexBuffer queue = new ApexBuffer(512); // send and receive methods public Alarm receive() { return (Alarm) queue.get(); } public void send(Alarm element) { queue.put(element); } ... </pre>
--	--

Figure 6: Template and implementation of a buffer pattern

This example also shows the WYSIWYG effect in the Ameos transformation templates. Everything not enclosed in square brackets is pure Java. Therefore existing transformation templates are easy to read, easy to maintain and easy to adapt to project specific needs.

Since TDL can access the ACD UML metamodel, it is possible to generate source code or test cases from State and Sequence Diagrams as well.

One of the most powerful concepts of ACD is *type mapping*, which allows the de-coupling of types used in the analysis and design from the types needed in the implementation. Type mapping also helps to generate all the necessary include, forward or import statements allowing the creation of 100% compilable code. Furthermore, technology specific type handling, e.g. the mapping of the CORBA-types to the respective implementation language types in the CORBA environment, and even project specific type conventions, can be supported in the framework of type mapping. Type mapping makes it possible to generate different programming languages like C++, Ada and Java from the same model.

Benefits of ACD

The major goal of ACD is to reduce the modeling effort, but increase the amount of generated code. In order to translate a 1 to 1 design model into source code, it is necessary, subject to architecture, conventions and programming language, to include many details in the model, which are actually redundant. Redundant because these details can be derived from the architecture or design conventions or from standard design patterns, and furthermore because these details are typically used identically in many classes.

Obviously, the effort to describe the transformation rules in the ACD template language is small compared to the effort of the manual implementation of the code, especially when we take into account that a template implements more than one model element. Think of a template implementing a state machine. Each class in a model, described by an additional State Diagram, will be implemented by this template. Depending upon the architecture one can expect that a large part of the source code for an application can be generated automatically.

It is important that the code generation follows the rules defined in the project. It does not help to generate code that is filled with macros and therefore difficult to read, or to generate additional operations that are not used. ACD code generation is not hard coded but defined by transformation rules in ASCII files. Therefore the code generation is transparent for the user and can be adapted easily.

Each manually implemented code line is a possible source for bugs and therefore code re-work and must be changed manually when modification is necessary. Generated code has a higher quality, is consistent with the model, and can be very easily adapted, through modification of the templates. Therefore, with ACD, dramatic improvements in productivity and quality are the result. Especially during later project stages, like maintenance or further development, the advantages of ACD become even more noticeable. Because of the fact that architecture specifics can be implemented by using ACD-templates, it is even possible to reuse architectures.

ACD also helps in making more effective use of technology specialists. Many software development projects and IT departments of larger companies do not have enough technology specialists to implement the business and user requirements using state of the art technology. For IT departments it is already hard enough to understand all the requirements to support the business's core needs. To be up to date on all the new developments in technology parallel to this is almost impossible. This, in turn, results in application and domain specialists being forced to get involved with database design and to trouble themselves with Architectures like CORBA. With ACD, the application specialists can concentrate on the business view. The few technology specialists write templates and develop architectural concepts as well as develop central technical classes. There is little need to document design and coding rules in project manuals, which are usually hard to communicate and check anyway. Instead, the templates automatically take care of many of these rules. Models created with Ameos are not tied to the target language. Therefore different target languages can be generated from a single model. Transformation rules for all supported languages like C, C++, Ada95 and Java are part of the Ameos Developer package.

Summary

UML helps to handle the growing complexity in modern software systems. Model Driven Architecture is a new way to describe systems independent from the final implementation and defines transformations between models in different levels of abstraction. Transformation is a big step forward and stands for the automation in the software industry.

Ameos and the transformation engine Architecture Component Development (ACD) provide mechanisms and tools which allow the use of UML in a software development project as a consistent and up-to-date information base. There is no need to create and maintain a detailed design model which is overburdened with implementation details.

ACD helps to realise many sought after benefits in software projects today. Bringing in projects on schedule, whilst meeting budgets, requires functionality and quality objectives with the available staff.

Templates can be constructed to support many languages, and different architectures, thus a number of templates have been put together to support the generation of C, C++, Ada and Java code from UML models.

To obtain more information, please contact Aonix at www.aonix.com or your local Aonix office.

North America

Phone: (800) 97-AONIX
Fax: (858) 824-0212
E-mail: info@eonix.com



France

Phone: +33 (0) 1 41 48 10 00
Fax: +33 (0) 1 41 48 10 20
E-mail: info@eonix.fr

United Kingdom

Phone: +44 (0) 1491 415000
Fax: +44 (0) 1491 571866
E-Mail: info@eonix.co.uk

Germany

Phone: +49 (0) 7 21/9 86 53-0
Fax: +49 (0) 7 21/9 86 53-98
E-mail: info@eonix.de

Sweden

Phone: +46 (0) 8 601 9491
Fax: +46 (0) 8 601 9499
E-mail: info@eonix.se

Copyright © 2005 Aonix Corporation. All rights reserved. Aonix and Ameos are registered trademarks of Aonix Corporation. All other company and product names are trademarks of their respective companies.