Model Driven Software Development through the integration of three models

Doug Orr

Genasys Consulting Services
2290 Pathfinder Drive
Burlington, ON L7L 6N1, Canada

orrd@acm.org

ABSTRACT

This paper describes an approach to model-driven software development. It promotes a process of managing the specification of system requirements by separating the problem into three distinct model sets: functional models, non-functional models, and an instantiation model. The approach is illustrated through examples based on a prototype implementation using the Eclipse Modeling Framework as its underlying metadata manager.

Categories and Subject Descriptors

D.1.2 Automatic Programming;

D.2.1 Requirements/Specifications

D.2.3 Coding Tools and Techniques

General Terms

Documentation, Design, Languages

Keywords

Model Driven Software Development, Aspect-Oriented Programming, Eclipse Modeling Framework, Code Generation.

1. INTRODUCTION

Traditional software development methodologies manage requirements under two separate categories: functional requirements and non-functional requirements. Functional requirements describe ***what*** the system is expected to do. They cover things like the information that is to be maintained, the business processes that are supported, and the organization that will work with the system. Non-functional requirements express ***how*** these capabilities are to be achieved. They cover things like performance, capacity, security, usability, availability, quality-assurance, supportability, standards compliance, etc.. Many development organizations separate the processes for gathering these two sets of requirements: business analysts focus on the gathering and documentation of functional requirements; application architects focus on the non-functional requirements. The business case and delivery plan for any solution typically represents a proposal to implement some sub-set of the functional requirements and some sub-set of the non-function requirements. Figure 1 illustrates this dynamic relationship.

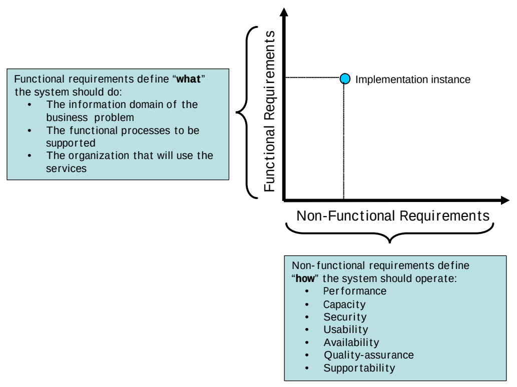

Functional requirements define "what" the system should do:
The information domain of the business problem
The functional processes to be supported
The organization that will use the services
Functional Requirements
Implementation instance
Non-Functional Requirements
Non-functional requirements define "how" the system should operate:
Performance
Capacity
Security
Usability
Availability
Quality-assurance
Supportability

Figure 1 - Dynamics of requirements interrelationship

Any application system can be thought of as the mapping of specific functional semantics to a set of non-functional heuristics.

There are many modeling paradigms that facilitate the capture and documentation of functional requirements. (This is the focus of the UML.) However, these models become complex and confusing, especially for non-technical business people, as layers on non-functional details are incorporated.

This paper presents an approach to managing the specification of system requirements by separating the problem into three distinct model sets: functional models, non-functional models, and an instantiation model. It also describes the software generation process that uses these three model sets.

The examples in this paper are based on a working prototype of a set of tools that use the Eclipse Modeling Framework (EMF) as the metadata manager for all of the models. However, the concepts could be implemented in any modeling environment that provides suitable programmable access to the underlying metadata.

2. FUNCTIONAL MODEL

The set of models that captures the functional features of a system can include a wide variety of notational structures. In UML this might be class diagrams, use cases, activity diagrams, interaction diagrams, etc. In any development setting, the choice of what type of model to use depends on how well it communicates to the key stakeholders (such as the business sponsor). For the examples in this paper we will concentrate on the simplified class model supported by the EMF but it is important to emphasize that any model semantics could be used.

The only stipulation imposed by our approach is that the underlying modeling framework must expose an API that permits programmable navigation of the metadata structures that comprise the model. For EMF, there is extensive documentation of the API and a good overview is provided here: http://www.awprofessional.com/content/images/0131425420/samplechapter/budinskych02.pdf.

Figure 2 illustrates a subset of the meta-model for the EMF.

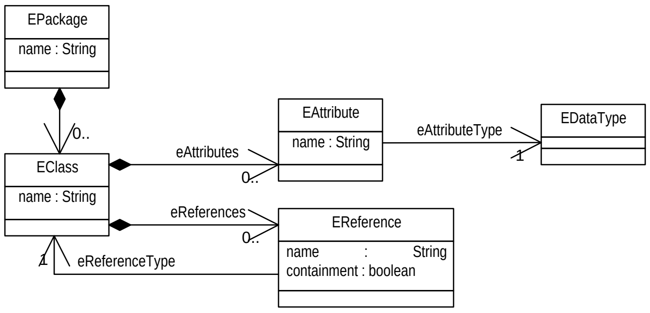
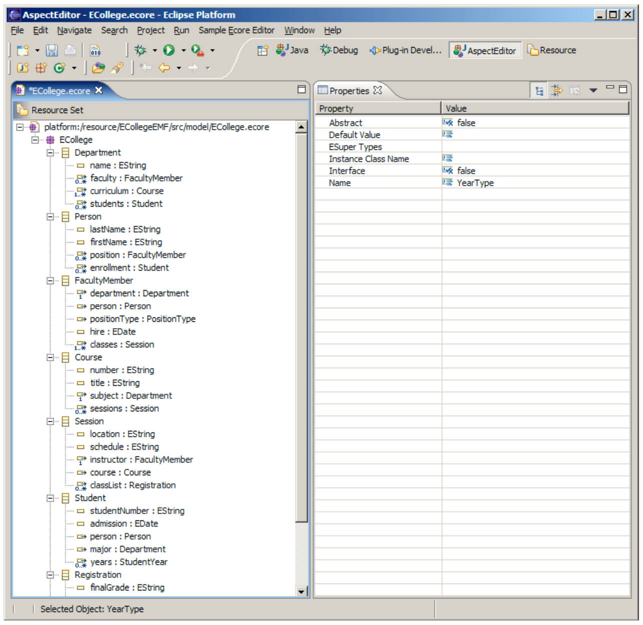


Figure 2 - Subset of the EMF ECore meta-model

Figure 3 illustrates a simple class model for an example business problem (college registration) as displayed by the EMF default model editor.


AspectEditor - ECollege.ecore - Eclipse Platform
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Java Debug Plug-in Devel... AspectEditor Resource
*ECollege.ecore
Properties

```
Resource Set
platform:/resource/ECollegeEMF/src/model/ECollege.ecore
  ECollege
    Department
      name : EString
      faculty : FacultyMember
      curriculum : Course
      students : Student
    Person
      lastName : EString
      firstName : EString
      position : FacultyMember
      enrollment : Student
    FacultyMember
      department : Department
      person : Person
      positionType : PositionType
      hire : EDate
      classes : Session
    Course
      number : EString
      title : EString
      subject : Department
      sessions : Session
    Session
      location : EString
      schedule : EString
      instructor : FacultyMember
      course : Course
      classList : Registration
    Student
      studentNumber : EString
      admission : EDate
      person : Person
      major : Department
      years : StudentYear
    Registration
      finalGrade : EString
```

Property	Value
Abstract	false
Default Value	
ESuper Types	
Instance Class Name	
Interface	false
Name	YearType

Selected Object: YearType

Figure 3 - Class model (College registration) as displayed by the EMF default editor

3. NON_FUNCTIONAL MODEL (or ASPECT MODEL)

Aspect-Oriented Programming has provided a vocabulary for describing how a system feature can be implemented as a collection of non-contiguous program statements embedded (or woven) into a number of modules within an application. The approach described in this paper does not rely on any implementation of AOP, but it borrows the concept to define a model for expressing the non-functional elements of a system.

Figure 4 illustrates a high-level class diagram of the Aspect model and Figure 5 shows an expanded representation of the model as implemented in the EMF prototype.

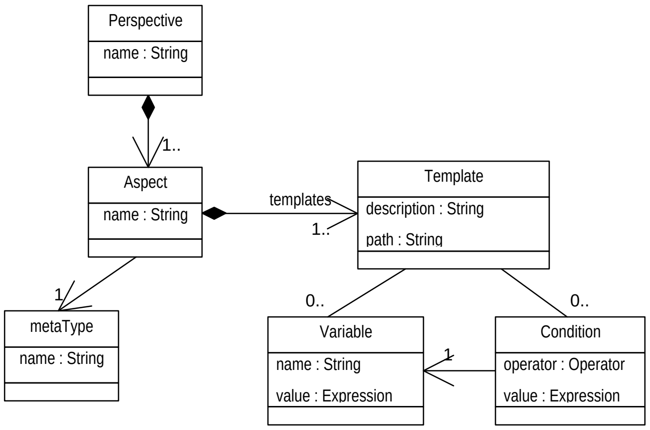
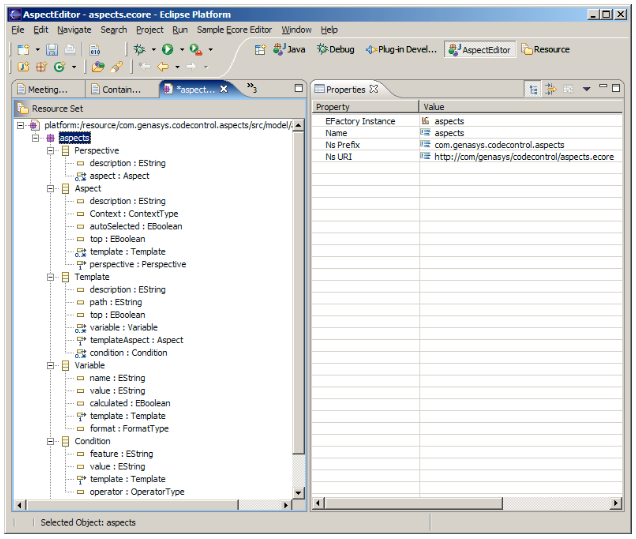


Figure 4 - Subset of the Aspect model


AspectEditor - aspects.ecore - Eclipse Platform
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Java Debug Plug-in Devel... AspectEditor Resource
Meeting... Contain... *aspect... »3
Properties

```
Resource Set
platform:/resource/com.genasys.codecontrol.aspects/src/model/
  aspects
    Perspective
      description : EString
      aspect : Aspect
    Aspect
      description : EString
      Context : ContextType
      autoSelected : EBoolean
      top : EBoolean
      template : Template
      perspective : Perspective
    Template
      description : EString
      path : EString
      top : EBoolean
      variable : Variable
      templateAspect : Aspect
      condition : Condition
    Variable
      name : EString
      value : EString
      calculated : EBoolean
      template : Template
      format : FormatType
    Condition
      feature : EString
      value : EString
      template : Template
      operator : OperatorType
```

Property	Value
EFactory Instance	aspects
Name	aspects
Ns Prefix	com.genasys.codecontrol.aspects
Ns URI	http://com/genasys/codecontrol/aspects.ecore

Selected Object: aspects

Figure 5 - Aspect Model as displayed by the EMF default editor

The following is a brief narrative description of the entities that comprise the Aspect model:

3.1 Perspective

The root of the Aspect model is a Perspective. A particular implementation may support many different Perspectives. A Perspective represents a general computer system implementation paradigm or framework (for example J2EE CMP paradigm, or .NET Web Services paradigm).

From a technical point of view, a Perspective is a container for the set of *Aspects* that comprise the implementation features of the particular application paradigm represented by the Perspective.

3.2 Aspect

Similar to the notion in AOP, an Aspect is a container for the "advice" that can be used to implement logic in a set of point-cuts. In this model the advice is stored as a collection of *Templates*.

A key distinction between the notion of Aspect presented here and the notion of Aspect in AOP is that there are no technology or language assumptions in this model. An Aspect can cut through descriptor files, Java code, database DDL, job-scheduling scripts or any other artifact of the resulting system. An Aspect can also point-cut through other Aspects recursively. The ultimate goal is to define Aspects that can implement every feature of the target P*erspective*.

3.3 MetaType

The context for an Aspect is driven by the associated MetaType. The metaType is the reference that connects an Aspect to a particular metadata type in the Functional Model.

In the EMF example this means that an Aspect can be associated with EPackage, EClass, EAttribute, or EReference structures.

3.4 Template

As already mentioned, Templates contain the "advice" snippets of the container Aspect. As also mentioned, the content of a Template may be a mix of a variety of textual syntax: XML for descriptor files, Java code, etc.

A key point to emphasize about the Template concept presented here is that there is virtually no specialized templating syntax. All of the controls associated with processing Templates is externalized in the *Variable* and *Condition* structures of the model.

3.5 Variable

A Variable defines a substitution that will be applied to the Template text at generation time.

3.6 Condition

A Condition describes a rule that will be enforced by default to control the selection of Templates that will be used at generation time.

4. INSTANTIATION MODEL (or ATTACHMENT MODEL)

Perspectives defined in a Non-Functional Model (or Aspect Model) are associated with Functional Models through the MetaType. That is, Aspects with a MetaType of "EClass" would be associated with each of the EClass entities in a particular EMF class model. However the interconnections of the functional and non-functional models may need to be customized to reflect the unique requirements of a particular business setting. This customization is facilitated by the Instantiation Model. Because the Instantiation Model represents the "attachment" of Aspects of the Non-Functional model to Entities in the Functional model, this model is also referred to as the Attachment Model. The Attachment model can be thought of as a representation of how a particular computer system paradigm (the Perspective of the Aspect model) would apply to a particular functional model.

Figure 6 illustrates a high-level class diagram of the Attachment model and Figure 7 shows an expanded representation of the model as implemented in the EMF prototype.

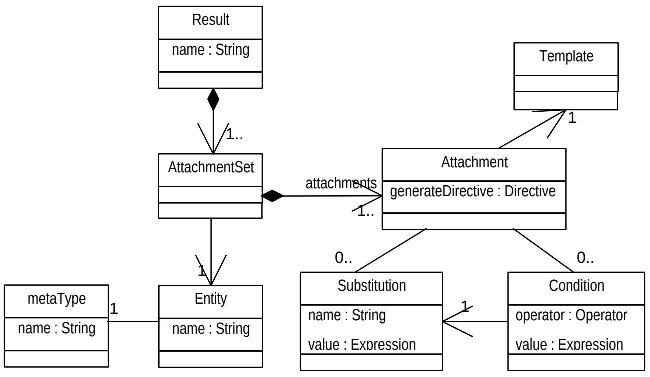
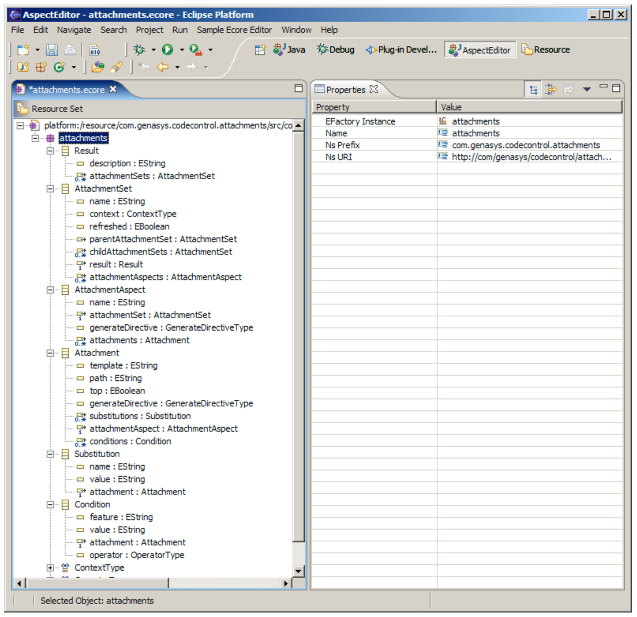


Figure 6 - Subset of the Attachment model



AspectEditor - attachments.ecore - Eclipse Platform
File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Java Debug Plug-in Devel... AspectEditor Resource
*attachments.ecore
Properties

```
Resource Set
platform:/resource/com.genasys.codecontrol.attachments/src/co
  attachments
    Result
      description : EString
      attachmentSets : AttachmentSet
    AttachmentSet
      name : EString
      context : ContextType
      refreshed : EBoolean
      parentAttachmentSet : AttachmentSet
      childAttachmentSets : AttachmentSet
      result : Result
      attachmentAspects : AttachmentAspect
    AttachmentAspect
      name : EString
      attachmentSet : AttachmentSet
      generateDirective : GenerateDirectiveType
      attachments : Attachment
    Attachment
      template : EString
      path : EString
      top : EBoolean
      generateDirective : GenerateDirectiveType
      substitutions : Substitution
      attachmentAspect : AttachmentAspect
      conditions : Condition
    Substitution
      name : EString
      value : EString
      attachment : Attachment
    Condition
      feature : EString
      value : EString
      attachment : Attachment
      operator : OperatorType
    ContextType
```

Property	Value
EFactory Instance	attachments
Name	attachments
Ns Prefix	com.genasys.codecontrol.attachments
Ns URI	http://com/genasys/codecontrol/attach...

Selected Object: attachments

Figure 7 - Attachment Model as displayed by the EMF default editor

The following is a brief narrative description of the entities that comprise the Attachment model:

4.1 Result

The root of the Attachment model is a Result. A Result is a container for the mapping of the functional model to the non-functional (Aspect) model. The name "Result" refers to the notion that this is the output from the first phase of the generation process.

4.2 Attachment

The core of the model is the set of Attachments. Each Attachment represents the association of each Template of the Aspect model with each associated entity of the Functional Model. As noted above, an Aspect points to the metaType with which it is associated. This reference is used to control the "attachment" of Templates.

The *generateDirective* attribute of the Attachment is used to override default conditional behaviour during the second phase of the generation process.

4.3 AttachmentSet

The AttachmentSet is a convenience class that collects all of the Attachments that are associated with a specific entity of the Functional Model.

4.4 Substitution

This structure is carried over from the *Variable* entity of the Aspect Model by the generation process. These substitutions will be applied to the Template text during the second phase of the generation process.

4.5 Condition

This structure is carried over from the *Condition* entity of the Aspect Model by the generation process. A Condition describes a rule that will be enforced to control the selection of Templates during the second phase of the generation process.

5. OVERVIEW OF THE GENERATION PROCESS

The process of code generation based on the models described in this paper consists of five logical steps. Figure 8 provides a high-level illustration of the process.

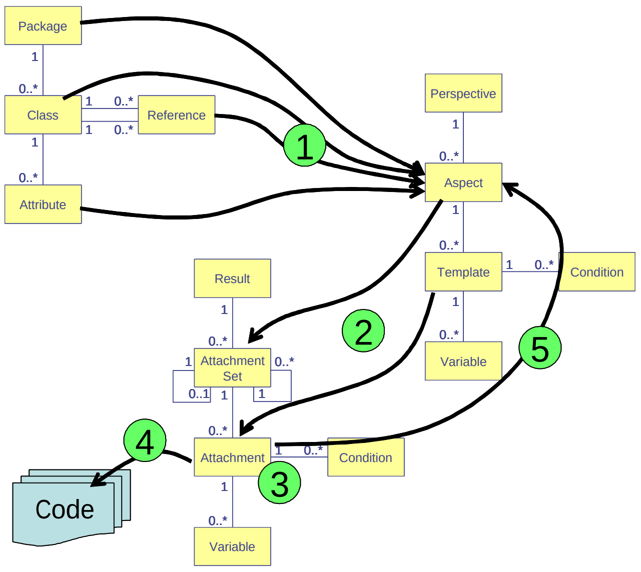


Figure 8 - High-level flow of the generation process

The prototype automates portions of this process in two phases: Phase one consists of steps 1 & 2. Phase 2 automates step 4. Step 3 is a manual process. Step 5 could be automated but this has not been done for the prototype. The following narrative briefly describes each of the steps.

5.1 Step 1 - Relate aspects to specific features of the business model

This step is automated by the prototype generator. A functional model and a non-functional (Aspect) model are passed to the generator. It walks through each model element (based on the metaType specified for each Aspect) and builds a collection of all of the relationships. For each of these relationships it walks through each variable of each associated template and calculates what the default substitution would be.

5.2 Step 2 – Build the Attachment model

Still within phase one of the generation process, the tree of associations from the previous step are stored in an Instantiation model (also referred to as an Attachment model). Substitutions are created for each Variable, and the default Conditions are copied.

5.3 Step 3 - Refine attachments if necessary

This is an optional manual step to apply customizations and overrides to the Attachment model before the final code is generated. Overrides can consist of altering default Substitutions or Conditions, or replacing the Template. The prototype captures and remembers these customizations so that they are not lost during subsequent regenerations.

5.4 Step 4 - Generate code based on attachment conditions

Phase 2 of the automated generation process walks the Attachment model and produces source code and other application artefacts from the provided Templates, Substitutions, and Conditions.

5.5 Step 5 - Upgrade aspects based on refinements

This is an optional step. Attachment customizations can be ported to the Aspect model so that they become integrated parts of the Non-Functional model. It is possible that some portion of this step could be automated (as the reverse of Step 2) but this has not been built into the prototype.

6. Perceived benefits of the approach

By separating the MDSD into three separate models we believe we achieve a number of significant benefits:

6.1 Simplified business view

The Functional Model is focused on capturing only the functional requirements of the business in the simplest terms possible. There is no need to "clutter" the model with constraints or other non-functional features that confuse or distract the business sponsor from validating the model.

6.2 Separation of concerns

The role of business analyst, responsible for capturing and documenting functional requirements, can be cleanly separated from the role of application architect. The focus of the business analyst is the functional models. The focus of the application architect is the non-functional model. Obviously there is value in each role understanding the deliverables of the other, but, to a great extent, they can work independently of each other.

This separation may not be appropriate for all development teams. On small teams a developer may play both the business analyst and the architect role. Nevertheless it encourages a boundary between the definition of business functions and the architectures that will be used to implement the functions. This separation provides other benefits as noted below.

6.3 Architecture life-cycle management

MDA is all about the ability to manage changes in architecture without corrupting or compromising the objectives of the business. The separation of the functional and non-functional models facilitates the introduction and management of architectural changes completely independently of the definition of the business' functional requirements. In fact, multiple target architectures (such as J2EE and .NET, or Struts and JSF) can be maintained as distinct, parallel Aspect models supporting different implementations of the same functional model.

7. ISSUES AND CONCERNS

The prototype demonstrates the significant productivity that comes with MDSD approaches. At the same time it highlights some of the challenges that still need to be addressed.

7.1 Separation of Concerns = Loss of Context

Debugging a system requires an understanding of both the functional features and the non-functional (architectural) features of the system. By documenting the requirements in separate models it can be difficult to bring together an understanding of the entire context of the system. It is possible that this could be addressed by generating appropriate documentation for all of the features of the system but this implies a significant level of discipline in the development of the Aspect model, because the documentation becomes its own set of Aspects.

7.2 Developer discipline

One of the problems common to most code generation facilities is the risk that a developer will abandon the approach because of lack of understanding or lack of patience with the processes imposed. To some extent this problem will be addressed as the tooling and training for MDSD matures. Nevertheless it should also be acknowledged that MDSD is changing the role of the business system developer, away from being a coder, and toward being a modeller.