UNIVERSITY OF CALIFORNIA
RIVERSIDE

A Study of Model Driven Architecture Approaches

A Project submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Maxime Najim

June 2005

Project Committee:
Dr. Doug Tolbert
Dr. Thomas Payne

The Project of Maxime Najim is approved:

____

____
Project Chairperson

University of California, Riverside

ACKNOWLEDGEMENTS

I would like to thank my project advisor

Dr. Doug Tolbert

for his support and encouragement and for opening my eyes to the world of MDA

I would also like to thank my mentor

Dr. Walid Najjar

for encouraging, advising, and supporting me for the past two years

A special thank you to

Dr. Thomas Payne

for his willingness to be part of this project

DEDICATIONS

This project is dedicated to my parents

Great Economos Dr. Michel Najim

and

Eva Najim

and to my brother and sister

Theodore Najim

and

Lydia Najim

and to all my friends and family

ABSTRACT OF THE PROJECT

A Study of Model Driven Architecture Approaches

by

Maxime Najim

Master of Science, Graduate Program in Computer Science
University of California, Riverside, June 2005
Dr. Doug Tolbert, Chairperson

Software production remains to this day abstract, complex, and constantly subject to pressures of change. With the advent of new and revolutionary tooling ideas and technologies initialized by the Object Management Group (OMG) termed Model Driven Architecture (MDA), software developers may now have the ability to create software systems entirely with models so to insure high productivity, maintainability and technological independence. For MDA to reach its full potential, the process of generating code from models must be fully automated. Several different approaches to executing model-to-code generation exist and each has a potential to carry out the MDA vision to its fullest. This project thesis classifies currently available MDA tools into two separate camps, namely Elaboration and Translation, according to their implemented model-to-code generation. Based on this classification, six tools are evaluated for their ability to generate a sample application. The elaboration approach proves superiority in code generation over the translation approach but lacks the PIM-centric methodology that may make MDA the silver bullet.

TABLE OF CONTETS

1. Introduction – MDA and Software Engineering.. 1
1.1 The Essence Problems of the Software Development Process................................ 1
1.1.1. Complexity.. 1
1.1.2. Conformity.. 2
1.1.3. Changeability.. 3
1.1.4. Invisibility... 4
1.2 The MDA Solution... 4
1.2.1 Complexity Solution.. 4
1.2.2 Conformity Solution.. 6
1.2.3 Changeability Solution.. 7
1.2.4 Invisibility Solution... 8
2. Preliminaries.. 9
2.1 Mapping between Models.. 9
2.2 Marking Models... 10
2.3 Extending UML.. 10
2.3.1 Stereotypes... 11
2.3.2 Tagged Values.. 11
2.3.3 Constraints.. 12
3. MDA Approaches.. 13
3.1 Elaboration.. 13
3.2 Translation.. 15
4. Evaluating MDA Approaches... 19
4.1 Case Study: Pet Store... 19
4.1.1 Model-View-Controller Design Pattern.. 20
4.1.2 Pet Store Evaluation... 21
4.2 The Tools.. 22
4.3 Evaluation Method.. 23
5. Results.. 26
6. Conclusion.. 28

LIST OF FIGURES

Figure 2.1 – MDA in a Nutshell..9
Figure 3.1 – Elaboration Approach...14
Figure 3.2 – Translation Approach..15
Figure 4.1 – MVC Pet Store Architecture...21
Figure 4.2 – Pet Store Use-Case Diagram...22
Figure 4.3 – 4.7 – Point Scheme Examples...22, 23
Figure 5.1 – MDA Evaluation Results...26
Figure 6.1 – Spectrum of Model-Code Relationship...28

1. Introduction – MDA and Software Engineering

Fredrick Brooks, in his influential essays written in the 1960s and 70s, collectively entitled The Mythical Man-Month, presented several key problems in software engineering which hinder software reliability and longevity. With the initiation of MDA, OMG envisions developers creating software systems entirely with models so to insure high productivity, maintainability and technological independence. This section inspects the software engineering problems presented by Brooks and their possible solutions in the MDA process.

1.1 The Essence Problems of the Software Development Process

Software production may seem to be quite innocent and straightforward at first glance, but like a werewolf, it can become horrific in the shape of late deadlines, exceeded budgets, and residual specification and design faults not detected during testing [Schach 45]. Brooks eloquently argued, in his historic article entitled “No Silver Bullet”, that there exists inherent difficulties in software production that no technique can nor ever will slay the software werewolf. To quote Brooks, “building software will always be hard. There is inherently no silver bullet.” Brooks listed four aspects of software production that remain inherently difficult, using the terms complexity (section 1.1.1), conformity (section 1.1.2), changeability (section 1.1.3), and invisibility (section 1.1.4).

1.1.1. Complexity

Brooks claimed that complexity is an integral part of the software development process. Brooks likened software production practices to the biblical account of building the tower of Babel, which failed due to the lack of communication, and its consequence,

disorganization. Good communication between developers in large teams is essential to reducing complexity. Early and continuous communication can give the architect good cost readings and the builder confidence in the design, without blurring the clear division of responsibility [Brooks 54]. Thus, as Brooks suggested, teams should communicate with one another in as many ways as possible [Brooks 74] so to avoid the tower of Babel syndrome.

Brooks advocated the use of a workbook which he defined as a structure imposed on documents produced by a project [Brooks 75]. In Brooks' mind, all documents of a project would be part of this structure, including objectives, specifications, and standards, just to name a few. Workbook restrictions included continual maintenance, complete distribution to all developers, and a comprehensive summary of changes prepared daily [Brooks 78]. However, with teams reaching to hundreds and even thousands of developers, such tasks would be, in itself, a great undertaking. How would one maintain such a structure successfully and remain productive?

1.1.2. Conformity

Brooks identifies two types of conformity issues wherein software acquires an unnecessary degree of complication [Brooks 184]. The first type occurs when a piece of software has to interface with an existing system. The second type occurs due to the misconception that software is the most conformable component in a system (hence the term "soft"). This leads to development of a system in which software is the last to be implemented so to conform to other components. Problems caused by this forced conformity cannot be removed by redesigning the software because the complexity is not

due to the structure of the software itself, but rather by the interfaces, to humans or to hardware, imposed on the software designer [Schach 47]. Both types are difficult software engineering problems.

1.1.3. Changeability

Similar to the previous, Brooks pointed out continual pressures to change software [Brooks 184]. Repairing design defects, adding functionality, and adapting to technological changes are all valid reasons for the need of software changeability. The issue of changeability therefore becomes the issue of maintainability. Brooks advocated the importance of planning a system for change. In fact, he went so far as to state that one should plan to throw the first implementation away as part of the learning process [Brooks 116]. To quote Brooks, "the throw-one-away concept is itself just an acceptance of the fact that as one learns, he changes the design" [Brooks 117]. Brooks saw program maintenance consisting chiefly of changes that repair design defects. He estimated that such changes capture 40%+ of the cost of developing software [Brooks 120].

Brooks suggested many relevant means of planning and organizing software development with change in mind. The most important included the use of high-level languages and self documenting techniques [Brooks 118]. However, Brooks admitted that changeability remains a problem. All repairs done on a system tends to destroy the structure and therefore increase the entropy, or disorder, of the system [Brooks 122]. Brooks stated, "Program maintenance is an entropy-increasing process, and even its most skillful execution only delays the subsidence of the system into unfixable obsolescence" [Brooks 123].

1.1.4. Invisibility

Brooks believed that there are no acceptable ways to represent either a complete product or some sort of overview of the product. Software is "invisible and unvisualizable" [Brooks 185]. To Brooks, programmers work vaguely detached from reality and build "castles in the air" by the application of their imagination [Brooks 7]. In that regard, software is somehow detached from any real representation. This inability to represent software visually not only makes software difficult to comprehend, but it also severely hinders communication among software developers [Schach 49].

1.2 The MDA Solution

Brooks' software engineering problems may have a modern solution with the advent of new and revolutionary tooling ideas and technologies initialized by the Object Management Group (OMG) termed Model Driven Architecture (MDA). This section will examine each of Brooks' inherent difficulties in software engineering and their possible solutions present in the visionary technology of OMG's MDA.

1.2.1 Complexity Solution

Anneke Kleppe, et al., in a book entitled MDA Explained, The Model Driven Architecture: Practice and Promise, stressed, along with many other MDA supporters, that documentation is viewed as a cumbersome chore that is impractical for software developers. As found in current practice, documentation is considered an overhead and usually remains undone until the end of a project. Programmers are seen productive when they write code and not when they write models or documents [Kleppe, et al. 4].

This is evident in the incremental and iterative version of the software process, wherein documents and models are produced in the requirements, specification and design phases. Documents and diagrams created in these first three phases rapidly lose their value once coding begins [Kleppe, et al. 2]. This is especially true when a system is changed over time. The distance between code and models becomes larger since these changes are often done at the code level only. The value of updating models after several iterations seems questionable since new changes are already present in the code. This complicates the crucial maintenance phase, which holds two-thirds of the total software effort [Schach 47]. Product maintainers, who are often different programmers then those that developed a product, need to rely on accurate documentation to accurately understand the system in order to correctly maintain it. Poor and inaccurate documentation is often the cause of incorrectly performed maintenance [Schach 47].

In order to reduce such situations, the MDA development life cycle places models (and therefore documentations) as an integral part of the development cycle. In MDA, the software development process is driven by the activity of modeling a software system [Kleppe, et al. 6]. The major difference between the traditional and the MDA software development cycle lies in the artifacts created during the development process. In the former, the artifacts are informal diagrams and text, where as in the latter, the artifacts are formal models that can be understood by computers. By the use of MDA tools, these models can be generated into a fully functioning software product.

This drastically decreases the apparent complexity of a software system. Since tools generate the code for the developer, there is less effort and confusion that are

inherent in coding. This improvement is even more effective since developers can now shift their focus from coding to designing and thus pay more attention to solving the business problem at hand. This results in a less complex system that fits much better with the needs of the end users. Such a gain can only be reached by use of fully automated model-to-code generations.

1.2.2 Conformity Solution

UML and other modeling languages are capable of defining interfaces more accurately than programming languages [Frankel 34]. This is true since modeling languages are semantically rich. A given model can be more easily conformed then a programming language. Models give the meaning of the system which can be interpreted to several interfaces. In beginning a new project, conformity can be solved by initiating the software development before the hardware and other such developments [Schach 48]. With models, this would mean modeling software interfaces before hardware interfaces so to insure conformity.

MDA realizes an additive solution, which allows reuse of each layer independent of the others. This mechanism of bridging between layers localizes the interface so that an interface can be changed and subsequently propagated throughout the code [Mellor, et al. 10]. The ability to transform one model from a specific technology to one that conforms to another technology is at the heart of the MDA process. This idea of interchanging technologies to better ensure conformity is discussed further in the next section.

1.2.3 Changeability Solution

The MDA framework was built with change in mind. The MDA development life cycle consists of two types of models, namely, Platform Independent Models (PIM), and Platform Specific Models (PSM). PIMs are models with a high level of abstraction that are independent of any implemented technology. A PIM is concerned with the business logic at hand and includes all specification of the system free from implementation bias. A PSM, on the other hand, is a model tailored to one specific implementation technology.

MDA achieves changeability by focusing on the development of PIMs, which can be automatically transformed into multiple PSMs for different platforms. Therefore, in the MDA process, the systems' architecture is imposed only at the last moment [Mellor, et al. 10]. This allows for complete technological flexibilities in response to pressures of change. This scheme may indeed be a significant improvement in maintenance. Changes made to the system will eventually be made by changing the PIM and regenerating the PSM and its associated code. Thus repairing design defects, adding functionality, and adapting to technological changes can be easily achieved by modification of the PIM.

Because it is the source of the generated system, the PIM fulfils the function of a high-level document that is required for the maintenance phase. Software maintenance will, thus, become design maintenance and therefore alleviate inherit problems of software changeability. Brooks had suggested maintaining documentation by incorporating it to the source program [Brooks 169]. In MDA, this is accomplished by making the documentation itself the central source of the program.

1.2.4 Invisibility Solution

A model is a simplification or an abstraction of a system under study (SUS), and in no way encompasses all the detailed aspects of a system. In the eyes of MDA, a model is simply a set of true statements about a SUS [Seidewitz]. A good model accurately reflects the reality of the SUS while at the same time omits technological information in order to help the viewer more clearly see the issues at hand and verify the correctness of the SUS [Mellor, et al. 26]. In the MDA process, once a model is interpreted, by mapping its elements to the element of the SUS, it becomes an exact description of the SUS [Seidewitz]. This is analogous to the PIM to PSM transformation. Therefore, a model allows developers to view, manipulate and reason about a SUS, and so to help them understand its inherent complexity, without a full visualization of the system [Mellor, et al. 25]. Therefore, although software is invisible, it can nonetheless be modeled at varying levels of abstractions, wherein these abstractions can be transformed into an actually system. In the MDA vision, systems are comprised of many small, manageable models rather than a single massive model.

2. Preliminaries

The first section described the importance of the MDA framework in relation to the software engineering essence problem as described by Brooks. This section elaborates on terms and concepts that are found at the heart of the MDA process. This section acts as a preliminary to topics found in the third section. These topics include mappings (2.1), markings (2.2), and modeling extending (2.3).

2.1 Mapping between Models

The MDA process consists of a set of models (i.e. PIMs and PSMs) which describe a particular system at different levels of abstraction. Models are constructed from other models by applying a set of implicit rules or mappings. In other terms, a mapping constitutes an interpretation of one model's element to another. Since MDA must support iterative and incremental development, these mappings must be repeatable. A mapping between models is assumed to take one or more models as input to produce output models [Mellor, et al. 16]. Mappings can specify the change in a PIM data type to a PSM data type where a PIM is built using a platform independent model types.

FIGURE 2.1 - MDA in a Nutshell

Mapping rules can also be described at the meta-model level where a PIM and PSM conform to the same meta-model. Recall, that a meta-model is a specification describing

a modeling language. That is, a meta-model makes statements about what can be expressed in valid models of a certain modeling language. OMG has developed standards for describing meta-models [MOF] and meta-model mappings [QVT].

2.2 Marking Models

Models are marked to aid in mapping. Markings in a PIM model indicate how model elements should be transformed into the target PSM. For example, a UML PIM is mapped to an Enterprise Java Bean (EJB) diagram by the aid of marks that indicate which class is transformed to an EJB. Each type of model element in the PIM is marked to indicate its mapped model type in the PSM. Marks, which can be platform specific, are not part of the PIM, but are rather lightweight, non-intrusive extensions. Marks indicate which template to apply and identify parameters for the template. In current MDA tools, marks are specified using an extension to UML.

2.3 Extending UML

The Unified Modeling Language (UML) is equipped with a built-in extension mechanism to define and use additional model constructs beyond those defined by basic or plain UML. A set of such extensions, which constitutes a dialect of UML, are called profiles. A UML profile is a definition of a set of stereotypes (2.3.1), tagged values (2.3.2), and constraints (2.3.3) that extend elements of the UML meta-model. The following subsections give a brief introduction to these concepts.

2.3.1 Stereotypes

A stereotype is an extension mechanism that defines a new kind of model element based on an existing model element. Thus, a stereotype is similar to an existing element, with some extra semantics that are not present in the former model element. In OMG terminology, a stereotype is defined by extending a UML meta-class element. UML notation allows for the assignment of a stereotype to a model element via a UML keyword surrounded by guillemets (i.e. <<*keyword*>>). In the MDA process, a model element is annotated with a stereotype so to aid the mapping function in generating corresponding platform specific elements.

To further explain, consider an illustration. Classes in a class model can be stereotyped as control, boundary, and entity. These stereotypes are used to increase semantics of classes and their usage. These mentioned stereotypes are based on the model-view-controller design pattern, where the entity is a model, the control is the controller, and the boundary is the view (MVC is discussed further in section 4.1.1). Marking a UML class with the <<entity>> stereotype aids a J2EE transformation function to transform a <<entity>> class into an EJB class along with other supporting classes.

2.3.2 Tagged Values

A tagged value explicitly defines an element property as a name-value pair where the name is referred to as the tag. Each tag represents a particular kind of information property applicable to one or more kinds of elements. Generally, the notation for a property is {tag1 = value1, tag2 = value2, etc.}. A tagged value can be associated with a

particular stereotype. For example, a <<UniqueId>> stereotype of an attribute could define a tag that indicates whether the identifier's value is user-assigned or system-assigned. The tag {isUserAssigned = true} next to a UniqueId attribute indicates that it is value assigned by the user.

2.3.3 Constraints

A constraint is a restriction on an element that limits usage of the element or the semantics of the element. Constraints add valuable information into a model and thus allow for a more complete and correct implementation of the system. Constraints in UML are usually expressed in the Object Constraint Language (OCL). OCL is a standard and sanctioned method for articulating constraints on UML models. Among many other constraints, OCL provides syntax to allow a modeler to specify invariants, pre-conditions, and post-conditions. (Section 3.1 contains a further discussion relating to OCL and the MDA process). An invariant is a condition that the class-implementer guarantees to be true throughout the execution of the system. A pre-condition is one that the caller of an operation guarantees to be true before the operation is called, and the post-condition is one that the operation-implementer guarantees to be true after the operation has been executed. These three key constraints are often used together to implement the technique knows as Design by Contract (DBC) [Frankel 79]. In order for an operation to function, the class-implementer, the caller, and the operation-implementer must hold up their part of the contract.

3. MDA Approaches

The MDA guide [Miller] suggests strategies and techniques for transforming between models by use of marking and mapping functions. However, it does not give details regarding how PIMs, PSMs and their transformations are to be implemented. This section will examine current approaches taken by tool vendors. These approaches can be segmented into two groups, namely Elaboration (2.1) and Translation (2.2).

3.1 Elaboration

The elaboration approach is characterized by the gradual definition of the application throughout the MDA process. The PIM model, in this approach, is generally understood to be a plain UML class model. The class model describes the static view of a system in terms of classes and relationship among the classes and their behavior. This class model is used as a foundation for other models and may be prepared using a platform independent UML profile, which may be transformed into a PSM expressed using a second, platform specific UML profile. The transformation may involve marking the model using marks provided with the platform specific profile.

Once the PIM has been developed and marked, the MDA tool can generate an incomplete or first-cut PSM. This PSM can be further elaborated with additional platform details. This same approach is taken to generate code from the PSM, which can also be further elaborated. With this approach, there must be a synchronizing between lower layer models to higher layer models since a model, in the eyes of MDA, must be consistence at all levels of abstraction. This is why elaboration tools generally support

the capability to regenerate higher-level models from lower-level models. This round-trip engineering can be aided by the use of markings and the concept of protected areas [Mellor, et al 85]. Undoubtedly, this approach requires that generated PSM and Code to be human readable and understandable to the developer. Figure 3.1 illustrates this approach.

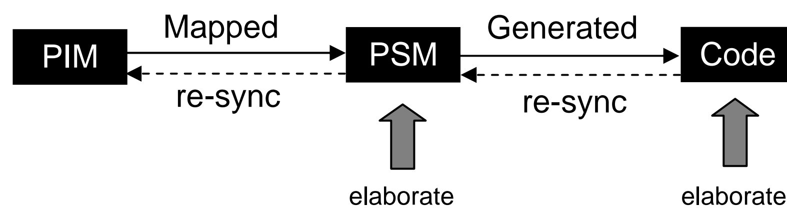

FIGURE 3.1 – Elaboration Approach

Also native to this approach is the use of patterns. That is, marked elements of the PIM are transformed according to some design pattern. These patterns comprise a set of templates for generating the PSM, where the marks are a way of binding template parameters [Miller]. Tools come with pre-defined set of transformations that indicate overall application architecture (e.g. J2EE architecture).

At the model layer, behavior is specified by the definition of pre-conditions, and post-conditions. These assertions are specified in a formal language, namely OCL. The operation is thus inferred and not actually specified in the PIM. For relatively simple operations the body of the operation might be generated from the OCL conditions, but most of the time the body of the operation must be written in the PSM [Kleppe, et al. 36]. Thus, in general, the PIM is not an executable artifact. To illustrate how OCL is used, consider the following OCL code:

```
context ShoppingCart::add( item: Thing ): Real
pre: not self.cart->includes( item )
post: self.cart->includes( item ) and result = item.price
```

where the special keyword `result` denotes the return value of the add operation. The behavior of the add operation can be inferred to mean that when the operation add`(item)` is called, the `item` is added to the `cart` list and its `price` is returned to the caller.

3.2 Translation

The other interpretation of the MDA process can be called translational in the sense that the PIM is translated directly into the final code of the system by model compilation. The translation of the PIM into final code is performed by a sophisticated code generator, sometimes referred to as a model compiler. Models in this approach are computationally complete and contain everything required to produce the desired functionality of a single problem domain. The translational approach is driven by Generation Rules that describe how the elements of the PIM are to be represented in the final code. Figure 3.2 illustrates this approach.

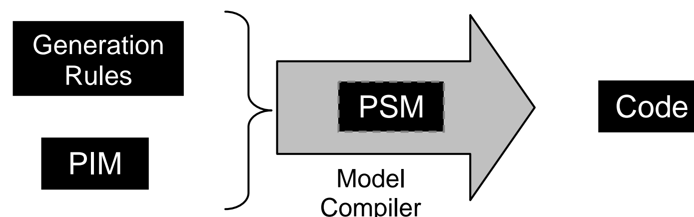

FIGURE 3.2 – Translation Approach

Contrary to the elaboration approach, the translational approach, as part of its feature, does not allow for further elaborations of the PSM or code. In fact, the PSM is internal to

the code generator and generally not visible or editable by the developer. The PIM with the addition of the Generation Rules is the full source of the generated system. Changes are only made at the PIM and are propagated down to the PSM and code by regeneration. For this reason, tools do not support synchronization and round-trip engineering.

Since UML is the main modeling language used by many MDA tools, this approach has also been called Executable UML (or xUML). xUML, as described in Model Driven Architecture with Executable UML by Chris Raistrick, et al, is a carefully selected, streamlined subset of UML, with clearly defined model structures, a precise semantics for actions, and a compliant action specification language. The primary models in xUML are domain, class, and state diagrams. Each domain in the domain diagram represents an independent subject matter in the system with respect to the customer's viewpoint. The class diagram is part of the static model and captures key abstractions and associations within a domain. It is the foundation of the domain model. The state chart captures dynamic behavior of active classes. These diagrams are further supported by use-case, sequence, and object diagrams. Use cases provide an informal description of required behavior from a user's perspective. The domain level sequence diagram shows how a use case is realized by the cooperation of various domains in the system. The lifelines on the sequence diagram correspond to domains. Within each domain an object level sequence diagram shows how use cases will be realized by interaction between objects. The lifelines on the sequence diagram correspond to objects. The interactions between objects serve to indicate which operations a class must provide along with the messages to send.

The xUML process differs greatly from the process found in tools using the elaboration approach. The xUML process is focused on developing a complete PIM in various phases which contain various parallel steps. In the first phase, called the Inception Phase, developers capture requirements with use cases, partition the system into domains, identify domain interactions and produce a class diagram for each domain to be modeled. In the second phase, termed the Modeling Phase, developers produce the sequence diagram, develop state charts for classes, add operations to classes and specify bridges between domains. As the last step implies, models are marked to indicate bindings between the different PIMs. This is contrary to the elaboration approach where models are marked to aid in their transformation from PIM to PSM. In addition, at each state and operation definition, an action language can be used to specify behavior. This action language is a key component of xUML and is the primary specification of behavior in the translation approach. Action Language, unlike OCL, is an imperative language. The state machines specify states that an object can have in its lifecycle, and how an object moves from state to state as events take place. Activities, specified in an action language, identify what an object does when its state changes. Defining state machines and activities within the PIM, results in a PIM that is executable, and therefore testable.

OMG has adopted a Precise Action Semantics for UML which provides an unambiguous semantic set of operations required to specify the behavioral aspects of a UML model. The semantic set includes operations that support manipulation of objects such as create, read, write, delete, and generation and handling of events or signals, and

logical constructs that support the specification of algorithms. Unfortunately, the Precise Action Semantics for UML is a semantic standard only, not a syntactic standard. This means that tools can implement their own Action Language syntax. This paper does not attempt to present in detail the specification of the Precise Action Semantics, but rather refers the reader to [Action Semantics Consortium] as an excellent source.

The final phase in the xUML process, called the Construction Phase, is when implementation-independent models specified in the previous phases are combined with archetypal design and code patterns. These patterns illustrate how attributes and operations from the analysis classes will map into a given language archetype (e.g. C++, Java, etc.). The model compiler extracts information from the application models and then selects and fills appropriate design patterns for each model element. The result is a fully coded model component.

Although the translation approach derives mainly from work on real-time and embedded systems, it is now being repositioned as a mainstream MDA technique, suitable also for business information and transactional systems. The next sections analyze the two approaches based on a case study and draws a conclusion as to which is a more suitable MDA approach.

4. Evaluating MDA Approaches

Currently, there exists no MDA tool that executes 100% model-to-code generation, however, as MDA tools become more sophisticated, the advantage of each of the approaches have become more apparent. This project classifies MDA tools by their MDA approach, Elaboration or Translation, and proceeds by evaluating each approach by a metric based on completeness. Three tools per classification were chosen. This section presents the project's case study (4.1) as well as a list of tools used for the evaluation (4.2) The method used to evaluate the mentioned MDA approaches is presented in subsection 4.3. Results of these methods are presented in section 5.

4.1 Case Study: Pet Store

Each tool was given the same task, namely to generate from models Sun's Java Pet Store blueprint application [Singh] (Pet Store). The Pet Store is a sample application using the Java 2 Platform, Enterprise Edition (J2EE) and illustrates the typical design decisions and tradeoffs a developer makes when building an enterprise application. Its advantage over other possible applications is due to the fact that a standardized and complete implementation is already provided. This implementation is used as a comparison point between the source code generated by each MDA tool. In addition, the Pet Store is built according the Model-View-Controller (MVC) design pattern, which most MDA tools support. To illustrate further the advantage of the Pet Store as a case study, the following subsections present the MVC design pattern (4.1.1) and how this pattern is used to segment the application (4.1.2).

4.1.1 Model-View-Controller Design Pattern

The Model-View-Controller (MVC) design pattern decouples an application into data access, business logic, and user interaction. MVC maps seamlessly into the domain of multi-tier enterprise applications. Models represent enterprise data and the business rules that govern access to this data. Views render contents of a model by accessing enterprise data through the model and specifying how that data is to be presented. Controllers translate interactions with the view into actions to be performed by the model. Based on user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

In the J2EE architecture, the Model-View-Controller is broken down into JavaServer Pages (JSP), which renders the view, Servlets, which act as controllers, and Enterprise JavaBeans (EJB) components, which are the models. The view layer consists of Screen Views, Composite Views, Screen Flow Managers, and View Helpers. Three additional components handle issues that arise in a distributed architecture namely, Service Locators, Value Objects, and Business Delegates. The controller layer components consist of Request Intercepting Filters, Event Controllers, Event Factories, Events, EJB Tier Controllers, and Command Factories. The model layer contains the components that handle business logic, namely, Session Facades, Business Objects, and Data Access Objects (DAO) which extract and formulate the data required to handle a client request. Figure 4.1 gives a breakdown of the MVC architecture.

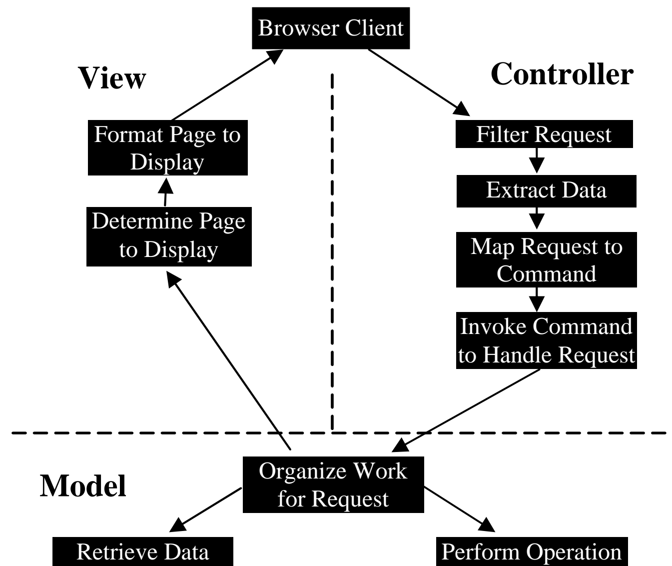

FIGURE 4.1 - Model-View-Controller Architecture in the Pet Store Application

For a detailed description of how the model, view, and controller components of the Pet Store web site handle requests and how different objects or components are designed to handle the application's functionality, the author refers the reader to [Singh].

4.1.2 Pet Store Evaluation

The evaluation of the Pet Store was based on use-case functionality. Since the Pet Store can be divided into MVC components, each function was divided by its Model, View, and Controller components. This provided a natural common delimiter between various functional points. Figure 4.2 is a use-case diagram depicting various pet store functionalities from the perspective of a customer. Each of these functionalities has an associated Model, View, and Controller.

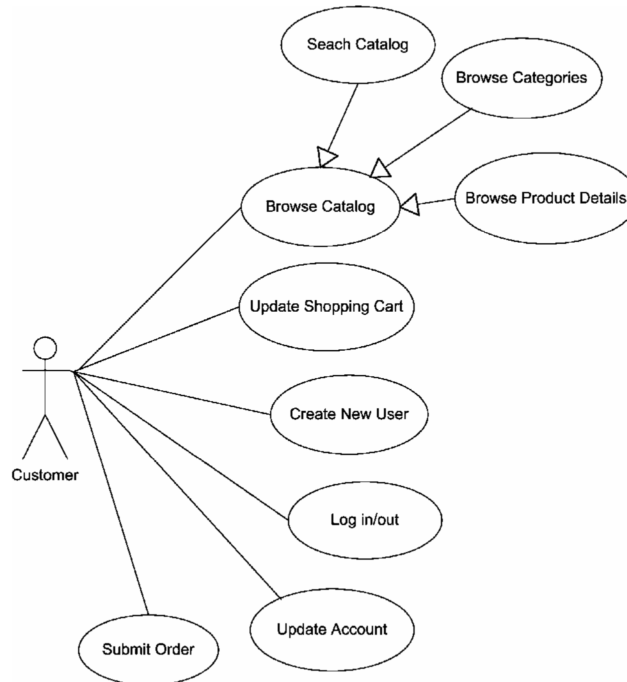

FIGURE 4.2 – Pet Store Use-Case Diagram

4.2 The Tools

Three tools were chosen per MDA approach and were used to generate the aforementioned Pet Store functionalities. For the elaboration approach, this included Rational XDE [IBM], OptimalJ [Compuware], and ArcStyler [Interactive Objects]. Current to the time of this writing, these tools represent the mainstream of MDA tools. For the translation approach, the three tools chosen are also central to their methodology. This included iUML [Kennedy-Carter], Wonder Machine Enterprise Edition [Sygel], and Gorilla Execution Engine [Gorilla Logic]. The reader should note that the full capability of these tools could not be utilized in this project since the evaluation or demo versions of the tools were used.

4.3 Evaluation Method

A three valued tuple $<M,V,C>$ was assigned to each functionality f of the Pet Store application, where M, V, and C represents the score for Model, View, and Controller components respectively. Thus, each tool t was assigned a set of tuples, such that, $t_n = \{<M_1,V_1,C_1>, \ldots , <M_k,V_k,C_k>\}$ where n is the number of tools (i.e. 6) and k is the number of functionalities (i.e. 8). The values ranged between 1 through 5, where a score of 1 represented simple skeleton code generation, and where a score of 5 represented full code generation. Scores of 2, 3, or 4 represented some, half, or most code generation respectively.

To further elaborate on this scoring scheme consider again the function `addItem(item)` in the `ShoppingCartBean` class which creates a new `CartItem` for a given `CatalogItem` and adds it to the `Cart`. Figure 4.3 shows the `addItem` fully generated by a tool which received a score of 5.

```
public void addItem(petstore.catalog.CatalogItemLocal item)
{
  String key = item.getItemId();
  if (!cart.containsKey(key)) {
    CartItem newItem = new CartItem(item);
    cart.put(key,newItem);
  }
}
```

FIGURE 4.3 – addItem generation with a score of 5

Figure 4.4 shows a mostly generated `addItem` function by a tool which received a score of 4. Notice the absence of a `newItem` and the improper use of the `put` function.

```
public void addItem(petstore.catalog.CatalogItemLocal item)
{
  String key = item.getItemId();
  if (!cart.containsKey(key)) {
    cart.put(key,item);
  }
}
```
FIGURE 4.4 – addItem generation with a score of 4

Figure 4.5 shows a half generated `addItem` function by a tool which received a score of 3. This code generator lacked the logic to generate the if-statement.

```
public void addItem(petstore.catalog.CatalogItemLocal item)
{
  String key = item.getItemId();
  cart.put(key,item);
}
```
FIGURE 4.5 – addItem generation with a score of 3

Figure 4.6 shows some generation of the `addItem` function by a tool which received a score of 2. This generation demonstrates a lack of a broader understanding of the system.

```
public void addItem(petstore.catalog.CatalogItemLocal item)
{
  cart.put(item);
}
```
FIGURE 4.6 – addItem generation with a score of 2

Lastly, figure 4.7 shows a simple skeleton code generation of the `addItem` function by a tool which received a score of 1. This generation demonstrates the inability to reason about a function besides its signature. This type of skeleton code generation has been the norm of model-to-code generation until recently.

```
public void addItem(petstore.catalog.CatalogItemLocal item)
{
    //Insert Code Here
}
```
FIGURE 4.7 – addItem generation with a score of 1

Once the tuples had been assigned values, the interquartile mean was taken of each Model, View, and Controller score. Recall that an interquartile mean is a truncated mean which discards the lowest 25% and the highest 25% of the scores so to insure insensitivity to outliers that may have been created by subjective scoring. Formally, $t_{n(IQM)} = < \frac{2}{n}\sum_{i=(n/4)+1}^{3n/4} M_i, \frac{2}{n}\sum_{i=(n/4)+1}^{3n/4} V_i, \frac{2}{n}\sum_{i=(n/4)+1}^{3n/4} C_i >$ where *M, V, C* are sorted lists from lowest-to-highest.

In order to further quantify the code-generation completeness of each tool and to insure corrected balance, a weighted average was taken of the combined *M, V, C* components. The standard Pet Store implementation consists of 14,273 lines of code of which of 5,891 lines of code are focused on Views (approximately 40%), 6,088 lines of code are focused on Models (approximately 40%), and 2,566 lines of code are focused on Controllers (approximately 20%). Thus, a weight of 2, 2, and 1 where given to the values of *M, V* and *C* respectively. Therefore, for each tool $t \in T$, $t = \frac{M_{IQM} \cdot 2 + V_{IQM} \cdot 2 + C_{IQM} \cdot 1}{5}$ where IQM signifies interquartile mean of a given set.

Finally, each tool was segmented into two sets, *Translation* and *Elaboration*, based on the tool's MDA approach. The mean of each set was taken in order to remove vendor bias, which resulted in one value for each approach and thus made evaluation clear-cut.

5. Results

Results of the evaluation methods presented in section 4 are shown in figure 5.1. Note the scoring results are presented in the *<M,V,C>* format as discussed in section 4.

	Elaboration			Translation		
	XDE	Arcstyler	OptimalJ	iUML	WMEE	GXE
Search Catalog	<5,3,4>	<4,4,3>	<5,4,5>	<4,3,5>	<3,2,4>	<3,2,3>
Browse Categories	<4,3,5>	<3,2,3>	<3,3,4>	<4,3,4>	<3,2,3>	<3,2,4>
Browse Product Details	<3,2,4>	<3,3,3>	<4,3,3>	<3,2,4>	<3,2,3>	<3,2,3>
Update Shopping Cart	<4,3,3>	<3,4,3>	<4,3,4>	<5,3,3>	<3,3,4>	<3,2,3>
Create New User	<4,3,4>	<4,3,4>	<4,3,4>	<4,3,4>	<3,2,3>	<2,2,3>
Log In/Out	<4,3,4>	<5,3,5>	<4,4,4>	<4,3,4>	<3,2,3>	<2,3,4>
Update Account	<3,3,3>	<5,3,4>	<4,3,3>	<4,3,4>	<3,2,3>	<3,2,3>
Submit Order	<4,4,5>	<5,4,5>	<5,4,4>	<5,4,4>	<3,2,2>	<3,2,3>
AVG.	<4, 3.5, 4>	<4, 3.25, 3.5>	<4, 3.25, 4 >	<4,3,4>	<3,2,3>	<3,2,3>
TOTAL	(3.8 + 3.6 + 3.7) / 3 = **3.7**			(3.6 + 2.6 + 2.6) / 3 = **2.93**		

Figure 5.1 – Evaluation Results

The Elaboration approach resulted in a greater average score of *3.7* or approximately 74% code generation. The Translation approach averaged to a score of *2.93* which is roughly 58% code generation. Some of the functionalities proved to be undemanding for some tools, admittedly since some functions were less complex then others.

A noticeable observation is that views, or more specifically user-interfaces (UI), present a major weakness. Although MDA tools may allow the modeling of the structure of the classes that make up the UI and how these classes interact, it is mostly impossible

to model the physical layout and navigational characteristics of the UI. This seems to be a general observable weakness in UML. An enhancement to UML which allows for the modeling of page-flow may allow for better code-generation.

The Elaboration approach, although producing a weaker PSM at the first PIM-to-PSM mapping, presents a clear advantage in overall code generation since it provides the ability to continually modify the PSM. Nevertheless, PSMs at times were not easily readable and consequently difficult to edit correctly. Furthermore, synchronization between the PIM and PSM at times seemed incomplete and hard to follow. Elaboration at the code-level was not accounted for in the results since this would not represent any form of code-generation but simply coding.

6. Conclusion

From the time of Brooks' articles and before, the problems of software engineering have endured. Brooks accurately observed that software, by its very essence, is abstract, complex, and constantly subject to pressures of change. Thus, software engineering continues even today to be troubled with essential difficulties which prohibit its productivity, reliability, and simplicity. The vision of the future in the form of OMG's MDA illustrates some promising improvements.

Figure 6.1 illustrates the spectrum of modeling approaches [Brown]. Each category identifies a potential use of models in assisting software developers in creating code of running applications for a specific runtime platform. The progression through the model-code spectrum reveals the different approaches in Model Driven Architecture.

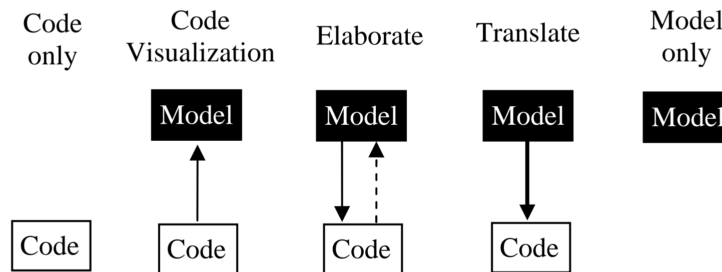

FIGURE 6.1 – Spectrum of relationships between models and code

A majority of software developers continue in the old code-only approach and perceive no superior need in expressing their system with models. These developers rely entirely on the expressiveness of a programming language. On the other end of the spectrum, in the model-only approach, developers use models purely as aids to understanding the System Under Study and are used as a basis for discussion, communication, and analysis

among teams. Code visualization is a step up from a code-only approach. These developers use one of many tools available to parse their code and generate some graphical notation of the code's structure or behavior. The elaboration approach, as described earlier, offers roundtrip engineering via bi-directional exchange between the design and its code. However, this approach is clearly an interim to what the MDA vision necessitates. Modification of the PSM and code do not offer a true PIM centered approach and thus lacks the elementary foundation on which the MDA vision is built upon (e.g. an inclusive PIM). The results presented in section 5 indicate that tools using the translation approach have not fully matured. 100% code generation is not yet available. In fact, translation approaches still lags behind the dominate elaboration approach. Moreover, these tools frequently make use of standard application frameworks that ease the code generation task but also constrain the possible styles of generated applications. Therefore, tools using the translation approach typically specialize in the generation of limited styles of applications.

Despite these shortcomings, translation is at the most advantageous point in the spectrum. Simply put, this approach offers the most pure MDA methodology. It offers a model-centric approach wherein the technologically independent models have sufficient details to enable the generation of a full system implementation. Therefore, the translation approach presents a clear advantage to other approaches since it offers executability and thus immediate testability of models. Moreover, translation fulfills the 100% principle which emphasizes the need for a separation of the conceptual model from all implementation details. Amid increasing use of patterns and reuse, a model-centric

programming environment would constitute a paradigm shift in the software engineering process and fit well in a continuing trend of increasing levels of abstraction. Perhaps, if the translation approach reaches 100% code generation, it has the potential of becoming the silver bullet that will slay the software werewolf.

REFERENCES

Action Semantics Consortium. *Action Semantics for the UML.* OMG Document: ad/2001-08-04. <http://www.kabira.com/as/home.html>

Brooks, Fredrick. The Mythical Man-Month. Addison Wesley Longman, Inc. 1995.

Brown, Alan. *An introduction to Model Driven Architecture*. IBM Rational, 2005.

Eriksson, Hans-Erik, et al. UML 2 Toolkit. John Wiley & Sons. 2003.

Frankel, David. Model Driven Architecture. Wiley Publishing, Inc. 2003.

Gorilla Logic. *Gorilla Execution Engine*. <http://www.gorillalogic.com/>

Haywood, Dan. *MDA: Nice Idea, Shame About The…*www.theserverside.com. May 2004.

IBM Corporation. *Rational Rose XDE Developer for Java - Evaluation v6.12* <http://www-306.ibm.com/software/awdtools/developer/java/>

Interactive Objects. *Arcstyler 5.0*. < http://www.io-software.com/products/ >

Kennedy Carter. *iUML 2.2*. <http://www.kc.com/products/iuml/>

Kleppe, Anneke, et al. MDA Explained: The Model Driven Architecture--Practice and Promise. Addison-Wesley Professional, 2003.

Miller, Joaquin and Mukerji, Jishnu, eds. *The Model-Driven Architecture, Guide Version 1.0.1*, OMG Document: omg/2003-06-01.

Miller, Joaquin. *The several styles of Model Driven Architecture*. Lovelace Computing Company, 2003.

McNeile, Ashley. *MDA: The Vision with the Hole?* www.metamaxim.com. 2003.

Mellor, Stephen J. et al. MDA Distilled. Addison-Wesley Professional, 2004.

MOF. *MOF 2.0 Core Final Adopted Specification*, OMG Document ptc/03-10-04.

Compuware. *OptimalJ Developer Edition*. <http://www.compuware.com/>

QVT. *Revised submission for MOF 2.0 Query/Views/Transformations RFP*. OMG Document omg/2003/08/18.

Raistrick, Chris, et al. Model Driven Architecture with Executable UML. Cambridge University Press, 2004.

Schach, Stephen. Object Oriented and Classical Software Engineering. McGraw-Hill Higher Eduction, 2002.

Singh, Inderjeet, et al. Designing Enterprise Applications with the J2EE Platform, Second Edition. Addison-Wesley, 2002.

Seidewitz, Ed. *What Model Mean*. IEEE Software. September/October 2003. pg26-32.

Tolbert, Douglas M. *CS 260-001: Perspectives on Model Driven Architectures*. Spring 2004. Dept. of CSE, UC Riverside. <http://www.cs.ucr.edu/~dtolbert/>

- - - . *Conceptual Modeling*. Encyclopedia of Microcomputers. Ed. Allen Kent, et al. New York: Marcel Dekker, 2000.

Watson, Andrew. VP & Technical Director. *April 2004 Technology process update*. omg/04-04-01.