Model Driven Development: Implementing a New Software Development Paradigm in Computer Science and Engineering Courses

John B. Bowles[1] and Caroline M Eastman[2]

Abstract

Model Driven Development (MDD) is the first truly new paradigm for software development in 50 years. Since the advent of block-structured and procedural languages in the 1950s software has been developed using if-then-else control constructs, do-while and for loops for iteration, and subroutine calls. With MDD software is developed by specifying a high-level, abstract model of the application, typically in a language such as UML (Unified Modeling Language) and then generating the code through a series of automated, pattern-driven transformations. Ideally, all the application code could be generated from the model; in practice, with today's technology, only about 70% of the code can be produced in this manner; the rest must be hand coded within the framework produced by the model. Just as programming in a high-level language requires such tools as a compiler to translate the program into executable code, the MDD paradigm requires tools to translate the UML model into code that can then be compiled and executed.

During the Fall 2003 term we applied the MDD paradigm using Compuware's OptimalJ™[*] MDD development tool to projects in our Capstone Software Engineering Project course. In the course the students developed requirements for a J2EE type application (typically an on-line store). Working in teams, they constructed a UML model for the application, primarily the class diagram, but also including the use case models and sequence diagrams. OptimalJ was used to build a first prototype and then the prototype was improved through several iterations by refining the business rules and improving the web-page presentation. Testing was integrated with the development throughout the development process. An interesting challenge was how to effectively use teams in developing the software. The MDD paradigm and supporting software will be used in two Spring 2004 courses, the Capstone Software Engineering Project course and a Database System Design course.

1. Introduction

Since the advent of block-structured and procedural languages in the 1950s software has been developed using if-then-else and switch control constructs, do-while and for loops for iteration, and subroutine calls. The recent development of a Model Driven Development (MDD) process has begun to change this paradigm [Kleppe, 6]. The MDD development process begins by specifying the system requirements in a high level modeling language, most often UML (Unified Modeling Language) [Fowler, 4], to form a Platform Independent Model (PIM) of the system. The PIM specifies the system at a high level of abstraction. Ideally, it is based entirely on the end functionality that the system must provide and is independent of the technology through which it will ultimately be implemented. The MDD approach, and the tools needed to support it, have most effectively been used on web applications, particularly J2EE (Java 2 Enterprise Edition) applications of which the prototypical example is an on-line store, but it is expanding into other areas as well.

In both a traditional development process and in an MDD process, the high level PIM must be transformed into a Platform Specific Model (PSM) that specifies the system requirements in terms of the specific implementation

[1] Computer Science & Engineering, University of South Carolina, Columbia, SC 29208

[2] Computer Science & Engineering, University of South Carolina, Columbia, SC 29208

[*] OptimalJ, QARun, and QADirector are trademarks of Compuware Corporation.

technology that will be used. For example, if a J2EE application is being developed the PIM will be implemented using technologies such as web servers, EJB (Entity Java Beans) structures, and contain terms such as "session bean" and "entity bean". If a main frame application with a relational database is being developed, the PSM will include data structures in the form of tables and specify requirements in terms such as "row", "column", and "foreign key". Other terminology and structures would be specified for a .net application.

In a traditional development process the transformation from the PIM to the PSM is done manually through requirements and system level design specifications, usually in the form of text on paper. In an MDD development process, the transformation is done automatically using technology design patterns. A pattern is a solution to a recurring problem in a fixed context. Patterns are discovered—not created—by recognizing the common elements of the solution to a problem; but a pattern can be applied in a multitude of different ways [Shalloway, 8]. For example, in the field of architecture, where the importance of patterns was first recognized, a door solves the problem of controlling entrance and egress from a room—but the variety of doors seems almost limitless. Thus, for a certain class of applications (e.g., on-line stores) and a specific underlying technology (e.g., J2EE) by recognizing common problems and the patterns that define their solution it is possible to automatically translate the formal UML specifications into the requirements in terms of the technology being used.

Once the specifications for the particular technology have been developed it is necessary to produce the code that actually implements the system. This is where system architects traditionally define class skeletons and code the method variables, and methods to implement the functionality required of the various system components. In an MDD process, implementation patterns are used to translate from the PSM to the actual code that implements the functionality. Because the PSM fits its technology fairly closely, this translation process is relatively straightforward.

Figure 1 shows the progression from the business problem to be solved to the deployed, functional system that solves the problem.

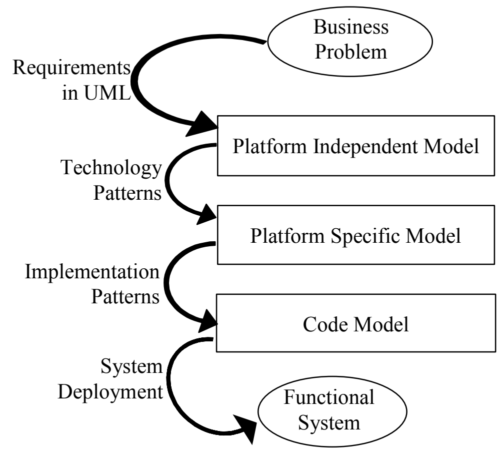


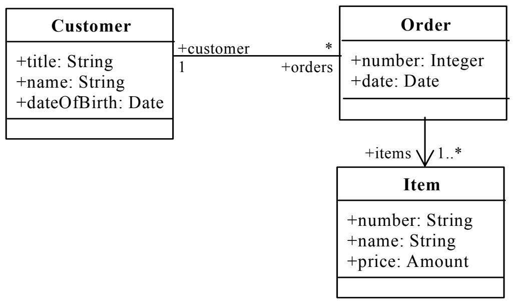
Figure 1: MDD development process showing the transformation of the business problem to a functional system.

1.1. AUTOMATION OF THE TRANSFORMATION

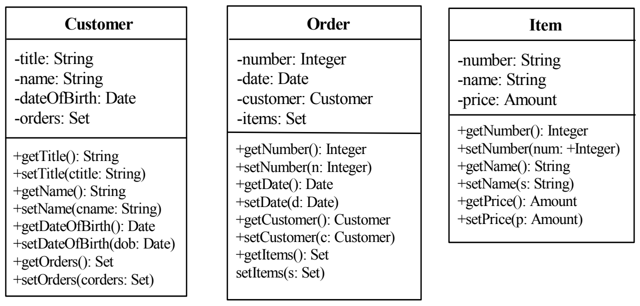
The transformations from model to model or from model to code are required in the traditional development process as well as in the MDD approach—but the translations are done by hand in the traditional approach as the requirements are specified and as the code is developed. Many tools have been developed to help with this process but usually they do little more than generate templates or "code skeletons" and most of the work of filling in the code and developing the system still has to be done by hand. In an MDD approach those transformations are performed by tools using "design patterns" appropriate for the given models.

To illustrate how the transformations are done, consider the example, adapted from [Kleppe, 6], showing sales order and customer classes in Figure 2. In Figure 2a a portion of a platform independent model in UML is shown. The model shows three classes: *Customer*, *Order*, and *Item*. A Customer is described by its attributes: *title*, *name*, and *dateOfBirth*; similarly, an Order has the attributes: *number* and *date*; and an Item has the attributes: *number*, *name*, and *price*. The three classes are associated. One Customer can have multiple Orders and each Order can include several Items. The corresponding part of the Java-specific PSM, still in UML, is shown in Figure 2b. Observe that the transformation from PIM to PSM generates *get*- and *set*- operations for each attribute in each class. In addition, instance variables and *get*- and *set*- operations are generated in each class for the opposite class of each association. Observe that where the multiplicity of the association is one, a single variable of the type of the opposite class is returned by the *get*- operation or required for the *set*- operation. Where the multiplicity is greater than one the variable is a *set*. Since the order-item association is one directional, corresponding *get*- and *set*- operations are not needed for the Item-PSM-class. These transformations from the PIM to the Java-specific PSM can all be done by employing transformation patterns in the form of rules such as:

- for each attribute (named *attributeName*) generate an operation called *get* concatenated with the name of the attribute and with the same return type as the attribute: *getAttributeName()*: *attributeType*.
- for each attribute (named *attributeName*) generate an operation called *set* concatenated with the name of the attribute and with a parameter having the same type as the attribute and no return value: *setAttributeName*(parameter: *attributeType*): *void*.
- for each association generate an attribute of the same name in the opposite class. If the multiplicity is 0 or 1 the type of this attribute is the opposite class; if the multiplicity is greater than 1 the type is *set*.



(a)



(b)

```
public class Customer
{
    String title;
    String name:
    Date dateOfBirth;
    Set orders;

   public String getName()
   {
      return name;
   }

   public void setName(String cname)
   {
      name = cname;
   }
      ...
```

(c)

Figure 2. Transformation from a Platform Independent Model to Code. (a) PIM; (b) Java-PSM; and (c) Java code for a customer order.

Figure 2c shows part of the Java code produced for the Customer class. The transformation from the Java-PSM model to the code model is straightforward. The code-class includes each of the attributes in the PSM for that class as a private instance variable of the corresponding type. Each get-variable operation returns a value of the corresponding type and contains code to return the value of the corresponding-instance-variable. Each set-variable operation assigns the parameter to the corresponding instance variable and is of type *void*.

Ideally, the tools used to generate the PSM from the PIM and the code model from the PSM would generate working code for exactly what the customer had in mind in the original system specifications. Unfortunately, that is not yet the case and, in fact, it might never be. While a working model can be developed much fine tuning is needed to change the look and feel of the user interface and to incorporate specific business rules (that cannot yet be specified in the UML model) into the application. With today's technology, only about 70% of the code can be produced in this manner; the rest must be hand coded within the framework produced by the model.

Recently introduced tools, such as OptimalJ from Compuware, aid in this process. All are based on using UML to model the system requirements. OptimalJ is currently the most mature of these tools and offers the most complete transformation process. These tools were originally intended to enhance productivity (a single analyst can replace many software coders), shorten the development cycle, and make a functioning prototype system available to the end user earlier in the development process. In a recent study of development productivity, a 3-person development team using OptimalJ (the team was experienced in J2EE development but they had not used OptimalJ previously) showed a 35 % productivity gain compared to an equally experienced team using traditional methods and tools to develop the same application. (333 development hours compared to 507.5 development hours) [5]. In addition, since the PIM is at a higher level of abstraction a developer can handle more complex applications with less effort.

2. Senior Capstone Software Engineering Course

Our senior Capstone Software Engineering course has traditionally followed a conventional software development process such as that described by Stiller and LeBlanc [Stiller, 8]. Students are assigned to work in teams of about 4 students each. Each team has to select a project and prepare a short, typically one-paragraph, description defining the project. They then develop the requirements, design the system, implement, and test the product. The sequence of deliverables is shown in Figure 3.

Using MDD the sequence of deliverables was changed as shown in Figure 4. The scenarios and use cases showed how users interacted with the system. The list of "nouns" identifying the main components of the system and an initial class diagram showing the major classes and their associations were constructed from these models. Sequence diagrams were developed to better understand the interactions and the exchange of information between the classes. Once these steps were finished the refined class diagram, including the class attributes, was constructed; the first prototype could then be generated automatically from the class diagram; thus no class skeletons were produced and no Java code was written.

The initial prototype left much to be desired. There was no attractive introductory web page. Instead the user was greeted by a sales order page where amounts could be entered in dialog boxes; the default colors (green and purple) were not particularly attractive; there were no logos, pictures, or other graphics; there were no informational statements or instructions on what to do. There were active links to other web pages, but even these were based on the attribute and association names (which were not necessarily very informative) and in some cases buttons or pull down menus would have been preferred. On the other hand, the prototype had been produced very quickly and a user could begin to experiment with it and get some "feel" for how the system worked.

Over the next few weeks, the initial prototype was improved. The students could change the default color scheme to one that was more appropriate for the application and that better suited their tastes. They could add "business rules" to compute values such as the total cost of a purchase or to compute a quantity discount if certain constraints were satisfied. They could also institute some limited error checking such as requiring that entries in certain fields satisfy simple regular expressions; for example, a postal code might have to be either 5 digits or 4 digits and two letters:

([1–9] [1–9] [1–9] [1–9] [1–9]) | ([1–9] [1–9] [1–9] [1–9] [A–Z] [A–Z])

Improving on the initial prototype was a slow and tedious process but as we gain experience and expertise in using OptimalJ it should go better and faster.

General requirements and Project plan
Refined requirements specification
Scenarios
Primary class list
Class diagrams
Use case diagrams
Structured walk-through (in class)
Object diagrams
Refined class diagrams
User interface mock-ups
State machines
Collaboration diagrams
Sequence diagrams
Object diagrams
Refined class diagrams
Class skeletons
Informal walk-through (in-class)
Implementation plan
Source code
Test plan
Test analysis report
System integration
System delivery and demonstration

Figure 3. Deliverables for traditional project development schedule [Stiller, 9].

General Requirements
Project Plan
Detailed Requirements
Scenarios
Use Cases
Requirements Verification
Class List
Class Diagrams – 1st OptimalJ prototype
Structured Walk-Through (in class)
Test Scenarios
Sequence Diagrams
Test Cases
2nd prototype with refined User Interface and business rules
Deployment diagrams
Test Evaluations & Defect Report
Test Summary Report
System delivery and demonstration

Figure 4. MDD project development schedule.

2.1. Team Development

Being able to work productively as a member of a team is an important attribute for an employee in industry. The scope of even a small project, including such activities as generating the initial requirements, developing the application, developing and carrying out a marketing plan, producing user and technical documentation, providing customer support and system maintenance, securing needed resources, and tracking resources used, to name only a few, is almost always more than a single person can do. In addition, no company can afford to risk either significant resources or its future existence on a single person. Thus, learning to work as a member of a team is an important part of most engineering curricula.

Team projects in academia are much more problematic. Usually, the projects are relatively small—something that can be done in a semester, so the need for the team isn't obvious. Students have widely differing interests, goals, and expectations—some just want to "get by" while others are driven to do their best work and most can only devote a small part of their effort to the team. If teams are assigned randomly the best students on the team often have to carry the weaker ones; if the teams are self-selected weak students often find comfort in working together but may be unable to finish the project.

MDD tends to exacerbate the team problem. Developing the architecture for the system is inherently a one or two person task. Much of the work where a larger team is needed, developing the detailed design and implementing the code, is automated with MDD. Consequently, how best to utilize a team in an MDD environment is still an open question. One suggested approach consists of partitioning the project into relatively independent subprojects that can then be dispersed among the team members [Compuware, 2]. The approach we took was to assign different threads of execution, defined by the use cases, to different members of a team. Each member then had to develop the class diagram appropriate for that thread; the final class diagram was a composite, consisting of the logical OR of these thread diagrams. In principle, this allowed the project to be developed in a piecemeal manner—the class diagram for one use case could be produced and the code for that use case generated; then the class diagram could be expanded to include the next use case, until the entire project was included.

2.2. Testing

Testing is an increasingly more important part of project development. In today's environment, where more and more software development is being farmed out to third party development teams, testing the final product to ensure that it conforms to specifications is becoming more and more important. In the traditional development process testing is done at the end of the development period, where, since the delivery date remains fixed, it is often short-changed if the project schedule slips. Many companies have begun to better integrate testing with the development process. They have found that this finds many errors much earlier in the development process when they are easier and less costly to fix and results in a higher quality product. With the MDD approach, the shortened development schedule enabled us to follow this strategy for testing.

We began by making the project teams larger than normal—five or six students instead of the usual three or four. Each team was split into two group: "developers" and "testers". The testers were responsible for integrating testing of the product throughout the development cycle. The entire project team worked to produce the general requirements defining the project. Then the developers built the use case models and defined the detailed requirements. Once these were available, the testers "verified" the requirements using the criteria in Figure 5 [Kit 5]. While the developers were building and debugging the class diagrams, the testers were developing "test scenarios"—looking at the usage scenarios and determining what needed to be tested. They then developed the project test plan and constructed specific test cases. Since this was a class in which the goal was for the students to learn about project development and not a commercial endeavor where the goal is to ensure that the product is as high quality as possible, we did not do exhaustive testing, but only required that the students define one positive and one negative test case for each use case. Beginning with the initial prototype the students could begin executing their test cases. The test group then prepared a "test execution report" and a "defect report" for each test run.

Complete. All items needed to specify the system are included

Correct. There are no obvious errors.

Clear. The items are stated precisely, and unambiguously. There is only one interpretation of a requirement.

Consistent. One item must not conflict with another.

Relevant. Each item is pertinent to the problem.

Testable. It must be possible to determine during development and acceptance that the item has or has not been satisfied.

Traceable. It must be possible to trace each requirement to its origin in the problem environment.

Feasible. It must be possible to implement each item with the available tools, techniques, and resources, personnel and within the project cost and schedule constraints.

Free of unwarranted design detail. Requirements should not encumbered with proposed solutions to the problem. I.e., they describe what must be done rather than how it will be done.

Manageable. It should be possible to change one item without unduly impacting on other items.

Figure 5. Check list for verifying requirements [Kit, 5].

While having one group assigned to "development" and another group assigned to "testing" mimicked what is sometimes done in industry, we felt that both groups needed to learn to use the development and testing tools we were using in the course. To accomplish this we planned to have the developers and testers switch "hats" after the initial prototype and test cases were constructed. This would give the testers an incentive to learn to use OptimalJ and the developers an incentive to learn to use the testing and test management tools, QARun™* and QADirector™* respectively. It also meant that the testers would have to "live with" what the developers had built and the developers would have to live with the test plans and cases that the testers had prepared.

Alas, the one project team/two-group approach worked no better in the class than it generally does in industry—and for many of the same reasons. First, all of the team members wanted to work on developing the product requirements; then everyone felt that they had to be included in the test plans and in running the test cases. So, in practice, the two groups were not separated well enough for the effort to be effective. The developers in essence verified their own requirements, defined their own test cases, and ran their own tests.

We plan to remedy this in the Spring term by mimicking another approach to testing used in industry. Each group will be charged with developing a set of requirements (including, a set of user scenarios, use cases, and user interfaces) for their project. These will then be given to another group to verify. Once they have been verified, the verifying group will be responsible for building the project. The group that originally developed the requirements will also have to develop a test plan and appropriate test cases based on those requirements. It will then have to test the completed project against those requirements. The challenge will be to do this in a one-semester course.

3. Conclusions

Model Driven Development (MDD) is the first truly new paradigm for software development in 50 years. During the Fall 2003 term we applied the MDD paradigm, supported with Compuware's MDD development tool, OptimalJ, to projects in our Capstone Software Engineering Project course. In the course, the students developed a J2EE type application such as an on-line store. They began by constructing a UML model of the application—primarily the application class diagram, but also including the use case models and sequence diagrams; then OptimalJ was used to generate a first prototype from the class diagram. This prototype was then improved through several iterations by refining the business rules and improving the web-page presentation. The MDD paradigm allowed us to integrate testing with the development throughout the development process. This significantly improved the testing of the application but more distance between the testers and the developers is needed. How to most effectively use teams to develop software using the MDD paradigm is still an open question. We partitioned the project based on threads from the use cases but it is not clear that this is the most effective way. The MDD paradigm and supporting software will be used in two Spring 2004 courses, the Capstone Software Engineering Project course and our Database System Design course.

References

1. Andersen, Sandra, *Data Structures in Java A Laboratory Course*, Jones and Bartlet, 2002.
2. Compuware Corp, "Team Development with OptimalJ", (2001), on-line at http://javacentral.compuware.com/members/downloads/whitepapers.htm
3. Deitel, H. M. and Deitel, P. J., (2003) *Java How to Program*, 5th Ed., Prentice Hall.
4. Fowler, Martin; Scott, Kendall; (2000) *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd Ed., Addison-Wesley, New York.
5. Kit, Edward, (1995) *Software Testing in the Real World: Improving the Process*, Addison-Wesley, New York.
6. Kleppe, Anneke; Warmer, Jos; and Bast, Wim; (2003) *MDA Explained: The Model Driven Architecture Practice and Promise*, Addison Wesley, New York.
7. "Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach Productivity Analysis", The Middleware Company, June 2003.
8. Shalloway, Alan; and Trott, J. R., (2002) *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, New York.
9. Stiller, Evelyn; and LeBlanc, Cathie (2002) *Project-Based Software Engineering: An Object-Oriented Approach*, Addison-Wesley, New York.

John B. Bowles

Computer Science and Engineering / University of South Carolina / Columbia, SC 29208 / e-mail: bowles@engr.sc.edu. John Bowles is an Associate Professor in the Computer Science and Engineering Department at the University of South Carolina where he teaches and does research in reliable system design. Previously he was employed by NCR Corporation and Bell Laboratories where he worked on several large system design projects. He has a BS in Engineering Science from the University of Virginia, an MS in Applied Mathematics from the University of Michigan, and a PhD in Computer Science from Rutgers University. Dr. Bowles is a senior member of IEEE and ASQ, and an ASQ Certified Reliability Engineer.

Caroline M. Eastman

Computer Science and Engineering / University of South Carolina / Columbia, SC 29208 / e-mail: eastman@engr.sc.edu. Caroline M. Eastman is Professor and Director of Undergraduate Studies in the Department of Computer Science and Engineering at the University of South Carolina. She has been a Program Director at the National Science Foundation and a faculty member at Florida State University and Southern Methodist University. She received her Ph.D. from the University of North Carolina at Chapel Hill. Her research interests include effective web searches, customizable information systems, and security in multimedia web databases.