"Jos Warmer's work has contributed greatly to the semantics of the UML. From that perspective, and in this book, he offers insight on how one can and can't use the UML to move to the next level of abstraction in building systems."
—Grady Booch

Experienced application developers often invest more time in building models than they do in actually writing code. Why? Well-constructed models make it easier to deliver large, complex enterprise systems on time and within budget. Now, a new framework advanced by the Object Management Group (OMG) allows developers to build systems according to their core business logic and data—indeed, independently of any particular hardware, operating system, or middleware.

Model Driven Architecture (MDA) is a framework based on the Unified Modeling Language (UML) and other industry standards for visualizing, storing, and exchanging software designs and models. However, unlike UML, MDA promotes the creation of machine-readable, highly abstract models that are developed independently of the implementation technology and stored in standardized repositories. There, they can be accessed repeatedly and automatically transformed by tools into schemas, code skeletons, test harnesses, integration code, and deployment scripts for various platforms.

Written by three members of OMG's MDA standardization committee, MDA Explained gives readers an inside look at the advantages of MDA and how they can be realized. This book begins with practical examples that illustrate the application of different types of models. It then shifts to a discussion at the meta-level, where developers will gain the knowledge necessary to define MDA tools.

Highlights of this book include:
The MDA framework, including the Platform Independent Model (PIM) and Platform Specific Model (PSM)

OMG standards and the use of UML

MDA and Agile, Extreme Programming, and Rational Unified Process (RUP) development

How to apply MDA, including PIM-to-PSM and PSM-to-code transformations for Relational, Enterprise JavaBean (EJB), and Web models

Transformations, including controlling and tuning, traceability, incremental consistency, and their implications

Metamodeling

Relationships between different standards, including Meta Object Facility (MOF), UML, and Object Constraint Language (OCL)

The advent of MDA offers concrete ways to improve productivity, portability, interoperability, maintenance, and documentation dramatically. With this groundbreaking book, IT professionals can learn to tap this new framework to deliver enterprise systems most efficiently.
MDA Explained: The Model Driven Architecture: Practice and Promise

By Anneke Kleppe, Jos Warmer, Wim Bast

Publisher: Addison Wesley
Pub Date: April 21, 2003
Pages: 192

Copyright
The Addison-Wesley Object Technology Series
The Component Software Series
Foreword
Preface
Introduction
Who Should Read This Book
How This Book Should Be Used
Typeface Conventions
Information on Related Subjects
Book Support and Example Implementation
Acknowledgments

Chapter 1. The MDA Development Process
Section 1.1. Traditional Software Development
Section 1.2. The Model Driven Architecture
Section 1.3. MDA Benefits
Section 1.4. MDA Building Blocks
Section 1.5. Summary

Chapter 2. The MDA Framework
Section 2.1. What Is a Model?
Section 2.2. Types of Models
Section 2.3. What is a Transformation?
Section 2.4. The Basic MDA Framework
Section 2.5. Examples
Section 2.6. Summary

Chapter 3. MDA Today
Section 3.1. OMG Standards
Section 3.2. UML as PIM Language
Section 3.3. Tools
Section 3.4. Development Processes
Section 3.5. Summary

Chapter 4. Rosa's Application of MDA
Section 4.1. Rosa's Breakfast Service
Section 4.2. Applying the MDA Framework
Section 4.3. The PIM in Detail
Section 4.4. Summary

Chapter 5. Rosa's PIM to Three PSMs
Section 5.1. The PIM to Relational Transformation
Section 5.2. The PIM to EJB Transformation
Section 5.3. The PIM to Web Transformation
Section 5.4. The Communication Bridges
Section 5.5. Summary

Chapter 6. Rosa's PSMs to Code
Section 6.1. Relational Model to Code Transformation
Section 6.2. EJB Model to Code Transformation
Section 6.3. The Web Model to Code Transformation
Section 6.4. Summary

Chapter 7. More on Transformations
Section 7.1. Desired Features of Transformations
Section 7.2. Controlling and Tuning Transformations
Section 7.3. Traceability
Section 7.4. Incremental Consistency
Section 7.5. Bidirectionality
Section 7.6. Implications on Transformations
Section 7.7. Summary

Chapter 8. Metamodeling
Section 8.1. Introduction to Metamodeling
Section 8.2. The Four Modeling Layers of the OMG
Section 8.3. The Use of Metamodeling in the MDA
Section 8.4. Summary

Chapter 9. Defining Your Own Transformations
Section 9.1. Transformations Definitions Revisited
Section 9.2. The Transformation Definition Language
Section 9.3. Example Transformation Definitions
Section 9.4. The Complete MDA Framework
Section 9.5. Summary

Chapter 10. Rosa's Transformation Definitions
Section 10.1. The UML to Relational Mapping
Section 10.2. The UML to EJB Mapping
Section 10.3. The UML to Web Mapping
Section 10.4. Summary

Chapter 11. OMG Standards and Additional Technologies
Section 11.1. Introduction
Section 11.2. The MOF
Section 11.3. Query, Views, and Transformations
Section 11.4. UML
Section 11.5. OCL
Section 11.6. The UML Action Semantics
Section 11.7. CWM
Section 11.8. UML Profiles
Section 11.9. Summary

Chapter 12. The MDA Promise
Section 12.1. The MDA Paradigm Shift
Copyright

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

MDA, UML, and MOF are either registered trademarks or trademarks of Object Management Group, Inc. in the United States and/or other countries.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales

(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales

(317) 581-3793
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Kleppe, Anneke G.


p. cm.

Includes bibliographical references and index.


QA76.76.D47K5635 2003

005.1--dc21 2003043725

Copyright 2003 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

Text printed on recycled paper

1 2 3 4 5 6 7 8 9 10—CRS—0706050403

First printing, April 2003
The Addison-Wesley Object Technology Series

Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors

For more information, check out the series Web site [http://www.awprofessional.com/otseries/].

Ahmed/Umrysh, Developing Enterprise Java Applications with J2EE and UML

Arlow/Neustadt, UML and the Unified Process: Practical Object-Oriented Analysis and Design

Armour/Miller, Advanced Use Case Modeling: Software Systems

Bellin/Simone, The CRC Card Book

Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools

Bittner/Spence, Use Case Modeling

Blakley, CORBA Security: An Introduction to Safe Computing with Objects

Booch, Object Solutions: Managing the Object-Oriented Project

Booch, Object-Oriented Analysis and Design with Applications, 2E

Booch/Bryan, Software Engineering with ADA, 3E


Box/Brown/Ewald/Sells, Effective COM: 50 Ways to Improve Your COM and MTS-based Applications

Carlson, Modeling XML Applications with UML: Practical e-Business Applications

Cockburn, Surviving Object-Oriented Projects: A Manager's Guide

Collins, Designing Object-Oriented User Interfaces
The Component Software Series

Clemens Szyperski, Series Editor

For more information, check out the series Web site [http://www.awprofessional.com/cseries/].

Allen, Realizing eBusiness with Components

Apperly et al., Service-and Component-based Development: Using the Select Perspective and UML

Atkinson et al., Component-Based Product Line Engineering with UML

Cheesman/Daniels, UML Components: A Simple Process for Specifying Component-Based Software

Szyperski, Component Software, 2E: Beyond Object-Oriented Programming

Whitehead, Component-Based Development: Principles and Planning for Business Systems
"The sooner you start, the longer it takes." This paradoxical slogan was coined by Fred Brooks in 1975 in his seminal software engineering text, The Mythical Man Month, to emphasize that in software projects preparation is never wasted. If you skimp on requirements capture, you'll waste effort designing things the customer doesn't want. If you skimp on design, you'll write a lot of pointless code that doesn't solve his problem . . . and so on.

Dr. Brooks' dictum reminds us that Good Design Matters. It's just as true now as it was twenty-five years ago, but much else has changed. His book was based on his experience with OS/360, at that time one of the largest software projects ever undertaken. Since then, software size and complexity have grown beyond recognition, as has the bewildering choice of implementation technologies. Should we be using C++, Java, Visual Basic, or C#? CORBA? .NET? Web services? ebXML? EJB? JavaScript? ASP? JSP? SQL? ODBC? JDBC? Today, a single software project typically uses several of these, placing each where its particular strengths are best used. We have also become increasingly aware of the importance of software maintenance. Mission-critical software can continue in use for decades, suffering constant upgrades to cope with shifting requirements and changing technologies, so that in time maintenance costs exceed all others combined.

This book describes the Model Driven Architecture (MDA) approach to creating good designs that cope with multiple-implementation technologies and extended software lifetimes. MDA is based on widely-used industry standards for visualizing, storing, and exchanging software designs and models. The best-known of these standards is the Unified Modeling Language (UML). The Object Management Group's (OMG) creation of UML has promoted good design by providing a common, widely-understood visual language for engineers to exchange and document their ideas, and this has led to a dramatic increase in the use of visual modeling. However, visual modeling has too often been seen merely as a way to draw pictures of software, pictures that must later be laboriously translated into runnable code. One of the traditional excuses for skimping on design is that comprehensive models are "just paper," and effort spent creating them could better be spent on writing real code. Partly as a result of this, there's a trend today towards development techniques that emphasize creating executable code instead of "mere" designs.

In contrast, MDA heavily emphasizes creating designs—not paper designs, but machine-readable models stored in standardized repositories. The intellectual effort invested in these models doesn't just sit passively on the page waiting to be laboriously recast as code. Instead, MDA models are understood by automatic tools from multiple vendors that generate schemas, code skeletons, test harnesses, integration code, and deployment scripts for the multiple platforms used in a typical project. Design effort invested in MDA models is repeatedly reused to generate multiple components, and by being updated over the life of an application, provides accurate documentation of how much-maintained software really works, rather than a frozen image of how things looked at the end of the design phase. In short, MDA is an architecture for creating good designs in today's multiplatform IT environment.

The authors of this book are well qualified to describe and document the MDA vision. They have contributed to OMG's creation of MDA, worked on commercial products that implement it, and used it in practice. The book is founded on solid theory, but it is nevertheless a practitioners' handbook, built around real-world examples, and offering guidance to IT professionals facing the ever-present problems of bringing in next year's project on time and on budget. Time spent reading it will not be time wasted. Remember: the sooner you start, the longer it takes.

Andrew Watson
OMG Vice President & Technical Director
35,000 ft. over Greenland
November 17th, 2002
Preface

For many years, the three of us have been developing software using object-oriented techniques. We started with object-oriented programming languages like C++, Smalltalk, and Eiffel. Soon we felt the need to describe our software at a higher level of abstraction. Even before the first object-oriented analysis and design methods like Coad/Yourdon and Object Modeling Technique (OMT) were published, we used our own invented bubble and arrow diagrams. This naturally led to questions like, "What does this arrow mean?" and "What is the difference between this circle and that rectangle?" We therefore rapidly decided to use the newly emerging methods to design and describe our software.

Over the years we found that we were spending more time on designing our models than on writing code. The models helped us to cope with larger and more complex systems. Having a good model of the software available made the process of writing code easier, and in many cases, even straightforward.

In 1997 some of us got involved in defining the first standard for object-oriented modeling called Unified Modeling Language (UML). This was a major milestone that stimulated the use of modeling in the software industry. When the OMG launched its initiative on Model Driven Architecture (MDA), we felt that this was logically the next step to take. People try to get more and more value from their high-level models, and the MDA approach supports these efforts.

At that moment we realized that all these years we had been naturally walking the path toward model-driven development. Every bit of wisdom we acquired during our struggle with the systems we had to build fitted in with this new idea of how to build software. It caused a feeling similar to an AHA-experience: "Yes, this is it!"—the same feeling we had years before when we first encountered the object-oriented way of thinking, and again when we first read the GOF book on design patterns. We feel that MDA could very well be the next major step forward in the way software is being developed. MDA brings the focus of software development to a higher level of abstraction, thereby raising the level of maturity of the IT industry.

We are aware of the fact that the grand vision of MDA, which Richard Soley, the president of the OMG, presents so eloquently, is not yet a reality. However, some parts of MDA can already be used today, while others are under development. With this book we want to give you insight into what MDA means and what you can achieve, both today and in the future.

Anneke Kleppe
Jos Warmer
Wim Bast
Soest, the Netherlands, March 2003
Introduction

Model Driven Architecture (MDA) is a fairly new asset in the field of computer science. This book explains the fundamentals of MDA in a broad perspective. Answers to questions like "What are models and how do they relate to code?" are given. The advantages of MDA and how these advantages can be realized, are discussed.

The concept of transformation of models is central to the realization of the benefits of MDA. We will explain what a transformation is, which types of transformations exist, and the way in which transformations can be defined. Yet the book remains critical towards the types and forms of transformations that will and will not lead to substantial benefits for the industry. Finally, the book contains a number of examples of transformations that are interesting in themselves.
Who Should Read This Book

The book is meant to be read by a wide audience. It is meant for anyone who wants to know what MDA is about and who wants to know the role of the different Object Management Group (OMG) standards and the tools that claim to support them. Technical managers will get an understanding of MDA that helps them judge when and how it can best be applied in their projects. It is also a book for the more experienced software developer who is interested in modeling and programming on a higher level of abstraction. Knowledge of the Unified Modeling Language (UML) and the Object Constraint Language (OCL) is presumed. Knowledge of Java, Enterprise Java Beans (EJB), SQL, and Java Server Pages (JSP) is helpful, especially for the examples, but not necessary.
How This Book Should Be Used

Surely a book like this is meant to be read, and to be interesting, from the start to the end, so we would like to encourage you to read it as a whole. However, we realize that not everyone has either the time or the technical background to read everything. Therefore, we would like to suggest three major tracks:

• Managers track: Managers who do not want to go into all details of the technology behind MDA should read Chapter 1 through Chapter 3, Chapter 11, and Chapter 12. Optionally, they might want to read Chapter 4.

• Developers track: People who are more interested in the application of MDA for developing software will want to read Chapter 1 through Chapter 7, Chapter 11, and Chapter 12.

• Specialists track: To get a thorough understanding of the technical background of MDA, you should read the entire book.

The details of the examples in Chapter 5, Chapter 6, and Chapter 10 can be skimmed through. This will not hinder understanding of the remainder of the book. However, it is important to realize that in real-life applications of MDA all of the details of these example chapters do exist, although they might sometimes be encapsulated into a black box.
Typeface Conventions

This book uses the following typeface conventions:

- All code examples are printed in a monospaced font.
- At the first introduction or definition of a term, the term is shown in italics.
- All references to classes, attributes, and other elements of a model are shown in italics.
- General emphasis is shown in italics.

No other typeface conventions are being used.
Information on Related Subjects

More information on MDA can be found in books mentioned in the bibliography and on the following Web site from the OMG:

- [http://www.omg.org/mda](http://www.omg.org/mda)
Book Support and Example Implementation

The Rosa's Breakfast example from Chapter 4 through Chapter 6 and Chapter 10 can be executed using the OptimalJ tool. A free trial version of the tool, including the complete example, can be downloaded from the following Web page:

- [http://www.klasse.nl/mdaexplained](http://www.klasse.nl/mdaexplained)
Acknowledgments

We would like to thank our reviewers for the time they have spent reading different versions of this book. Without their constructive feedback, this book would not be what it is today. We have benefited greatly from the insights that they shared with us. We would like to thank Frank Baerveld, Jeroen Bruijning, Aldo Eisma, Ghica van Emde-Boas, Peter van Emde-Boas, Martin Gogolla, Martin Matula, Hans van Oosten, and Andrew Watson. Special thanks goes to Heinrich Hußmann who made the effort of reading the first draft even though much of his attention was directed to trying to keep his feet and equipment dry during the flooding of Dresden in the summer of 2002. Another person that deserves our special thanks is John Daniels, whose thorough and positive criticism has led us to rewrite large parts of the first draft of the book.

We also want to thank the team from Addison-Wesley for their help, especially Mary O'Brien, who became our editor at a very late stage, but arranged every detail of our collaboration smoothly.
Chapter 1. The MDA Development Process

This chapter describes some of the major problems in software development. It explains the concepts of the Model Driven Architecture (MDA), and discusses how MDA can help to solve these problems.
1.1 Traditional Software Development

Software development is often compared with hardware development in terms of maturity. While in hardware development there has been much progress, e.g., processor speed has grown exponentially in twenty years, the progress made in software development seems to be minimal. To some extent this is a matter of appearances. The progress made in software development cannot be measured in terms of development speed or costs.

Progress in software development is evident from the fact that it is feasible to build much more complex and larger systems. Just think how quickly and efficiently we would be able to build a monolithic mainframe application that has no graphical user interface and no connections to other systems. We never do this anymore, and that is why we do not have solid figures to support the idea that progress has been made.

Still, software development is an area in which we are struggling with a number of major problems. Writing software is labor intensive. With each new technology, much work needs to be done again and again. Systems are never built using only one technology and systems always need to communicate with other systems. There is also the problem of continuously changing requirements.

To show how MDA addresses these problems, we will analyze some of the most important problems with software development and discover the cause of these problems.

1.1.1 The Productivity Problem

The software development process as we know it today is often driven by low-level design and coding. A typical process, as illustrated in Figure 1-1, includes a number of phases:

1. Conceptualization and requirements gathering
2. Analysis and functional description
3. Design
4. Coding
5. Testing
6. Deployment
1.2 The Model Driven Architecture

The Model Driven Architecture (MDA) is a framework for software development defined by the Object Management Group (OMG). Key to MDA is the importance of models in the software development process. Within MDA the software development process is driven by the activity of modeling your software system.

In this section we first explain the basic MDA development life cycle, and next illustrate how MDA can help to solve (at least part of) the problems mentioned in the previous sections.

1.2.1 The MDA Development Life Cycle

The MDA development life cycle, which is shown in Figure 1-2, does not look very different from the traditional life cycle. The same phases are identified. One of the major differences lies in the nature of the artifacts that are created during the development process. The artifacts are formal models, i.e., models that can be understood by computers. The following three models are at the core of the MDA.

Figure 1-2. MDA software development life cycle

The PIM, PSM, and code are shown as artifacts of different steps in the development life cycle. More importantly, they represent different abstraction levels in the system specification. The ability to transform a high level PIM into a PSM raises the level of abstraction at which a developer can work. This allows a developer to cope with more complex systems with less effort.

1.2.2 Automation of the Transformation Steps

The MDA process may look suspiciously much like traditional development. However, there is a crucial difference. Traditionally, the transformations from model to model, or from model to code, are done mainly by hand. Many tools can generate some code from a model, but that usually goes no further than the generation of some template code, where most of the work still has to be filled in by hand.

In contrast, MDA transformations are always executed by tools as shown in Figure 1-3. Many tools are able to transform a PSM into code; there is nothing new to that. Given the fact that the PSM is already very close to the code, this transformation isn't that exciting. What's new in MDA is that the transformation from PIM to PSM is automated as well. This is where the obvious benefits of MDA come in. How much effort has been spent in your projects with the painstaking task of building a database model from a high-level design? How much (precious) time was used by building a COM component model, or an EJB component model from that same design? It is indeed about time that the burden of IT-workers is eased by automating this part of their job.

At the time of writing, the MDA approach is very new. As a result of this, current tools are not sophisticated enough to provide the transformations from PIM to PSM and from PSM to code for one hundred percent. The developer needs to manually enhance the transformed PSM and/or code models. However, current tools are able to generate a running application from a PIM that provides basic functionality, like creating and changing objects in the system. This does allow a developer to have immediate feedback on the PIM that is under development, because a basic prototype of the resulting system can be generated on the fly.
1.3 MDA Benefits

Let us now take a closer look at what application of MDA brings us in terms of improvement of the software development process.

1.3.1 Productivity

In MDA the focus for a developer shifts to the development of a PIM. The PSMs that are needed are generated by a transformation from PIM to PSM. Of course, someone still needs to define the exact transformation, which is a difficult and specialized task. But such a transformation only needs to be defined once and can then be applied in the development of many systems. The payback for the effort to define a transformation is large, but it can only be done by highly skilled people.

The majority of developers will focus on the development of PIMs. Since they can work independently of details and specifics of the target platforms, there is a lot of technical detail that they do not need to bother with. These technical details will be automatically added by the PIM to PSM transformation. This improves the productivity in two ways.

In the first place, the PIM developers have less work to do because platform-specific details need not be designed and written down; they are already addressed in the transformation definition. At the PSM and code level, there is much less code to be written, because a large amount of the code is already generated from the PIM.

The second improvement comes from the fact that the developers can shift focus from code to PIM, thus paying more attention to solving the business problem at hand. This results in a system that fits much better with the needs of the end users. The end users get better functionality in less time.

Such a productivity gain can only be reached by the use of tools that fully automate the generation of a PSM from a PIM. Note that this implies that much of the information about the application must be incorporated in the PIM and/or the generation tool. Because the high-level model is no longer "just paper," but directly related to the generated code, the demands on the completeness and consistency of the high-level model (PIM) are higher than in traditional development. A human reading a paper model may be forgiving—an automated transformation tool is not.

1.3.2 Portability

Within the MDA, portability is achieved by focusing on the development of PIMs, which are by definition platform independent. The same PIM can be automatically transformed into multiple PSMs for different platforms. Everything you specify at the PIM level is therefore completely portable.

The extent to which portability can be achieved depends on the automated transformation tools that are available. For popular platforms, a large number of tools will undoubtedly be (or become) available. For less popular platforms, you may have to use a tool that supports plug-in transformation definitions, and write the transformation definition yourself.

For new technologies and platforms that will arrive in the future, the software industry needs to deliver the corresponding transformations in time. This enables us to quickly deploy new systems with the new technology, based on our old and existing PIMs.
1.4 MDA Building Blocks

Now what do we need to implement the MDA process? The following are the building blocks of the MDA framework:

- High-level models, written in a standard, well-defined language, that are consistent, precise, and contain enough information on the system.

- One or more standard, well-defined languages to write high-level models.

- Definitions of how a PIM is transformed to a specific PSM that can be automatically executed. Some of these definitions will be "home-made," that is, made by the project that works according to the MDA process itself. Preferably, transformation definitions would be in the public domain, perhaps even standardized, and tunable to the individual needs of its users.

- A language in which to write these definitions. This language must be interpreted by the transformation tools, therefore it must be a formal language.

- Tools that implement the execution of the transformation definitions. Preferably these tools offer the users the flexibility to tune the transformation step to their specific needs.

- Tools that implement the execution of the transformation of a PSM to code.

At the time of writing, many of the above building blocks are still under development. Chapter 3 provides an overview of where we stand today.

In the following chapters each of the building blocks is further examined and we show how it fits into the overall MDA framework.
1.5 Summary

The Model Driven Architecture is a framework for software development, defined by the OMG. Key to MDA is the importance of models in the software development process. Within MDA the software development process is driven by the activity of modeling your software system.

The MDA development life cycle is not very different from the traditional life cycle. The artifacts of the MDA are formal models, i.e., models that can be understood by computers. The following three models are at the core of the MDA:

- Platform Independent Model (PIM), a model with a high level of abstraction, that is independent of any implementation technology.

- Platform Specific Model (PSM), a model tailored to specify your system in terms of the implementation constructs that are available in one specific implementation technology. A PIM is transformed into one or more PSMs.

- Code, a description (specification) of the system in source code. Each PSM is transformed into code.

Traditionally the transformations from model to model, or from model to code, are done mainly by hand. In contrast, MDA transformations are always executed by tools. Many tools have been able to transform a PSM to code; there is nothing new to that. What's new in MDA is that the transformation from PIM to PSM is automated as well.
Chapter 2. The MDA Framework

This chapter introduces the MDA framework that lies behind the process that we described in Chapter 1. The MDA framework consists of a number of elements that fit together in a specific way. A small and simple example is given to clarify the different elements.
2.1 What Is a Model?

The name MDA stresses the fact that models are the focal point of MDA. The models we take into account are models that are relevant to developing software. Note that this includes more than just models of software. When a piece of software is meant to support a business, the business model is relevant as well.

But what exactly do we mean when we use the word model? To come up with a definition that is both general enough to fit many different types of models is difficult. The definition also needs to be specific enough to help us specify automatic transformation of one model into another. In the English dictionary we can find various meanings of model:

- The type of an appliance or of a commodity
- The example used by an artist
- A person posing for an artist
- A replica of an item built on a smaller scale, i.e., a miniature
- An example of a method of performing work
- An ideal used as an example
- The form of a piece of clothing or of a hairdo, and so on

What all of the above definitions have in common is that:

- A model is always an abstraction of something that exists in reality.
- A model is different from the thing it models, e.g., details are left out or its size is different.
- A model can be used as an example to produce something that exists in reality.
2.2 Types of Models

The definition of model given in section 2.1, What Is a Model?, is a very broad one that includes many different kinds of models, so we will take a closer look at models. There are many ways to distinguish between types of models, each based on the answer to a question about the model:

- In what part of the software development process is the model used? Is it an analysis or design model?
- Does the model contain much detail? Is it abstract or detailed?
- What is the system that the model describes? Is it a business model or software model?
- What aspect of the system does the model describe? Is it a structural or dynamic model?
- Is the model targeted at a specific technology? Is it platform independent or platform specific?
- At which platform is the model targeted? Is it an EJB, ER, C++, XML, or other model?

What we need to establish is whether these distinctions are relevant in the context of model transformations. The answer to some of the above questions varies according to the circumstances. The distinction made is not a feature of the model itself. Whether a model is considered to be an analysis or design model depends not on the model itself, but on the interpretation of the analysis and design phases in a certain project. Whether a model is considered to be abstract or detailed depends on what is considered to be detail.

When the distinguishing feature is not a feature of the model itself, this feature is not a good indication for characterizing different types of models. So, answering the first two questions in the list above does not clearly distinguish different types of models. The answers to the other questions in the above list do indicate different types of models. We further investigate these distinctions in the following sections.

2.2.1 Business and Software Models

The system described by a business model is a business or a company (or part thereof). Languages that are used for business modeling contain a vocabulary that allows the modeler to specify business processes, stakeholders, departments, dependencies between processes, and so on.

A business model does not necessarily say anything about the software systems used within a company; therefore, it is also called a Computational Independent Model (CIM). Whenever a part of the business is supported by a software system, a specific software model for that system is written. This software model is a description of the
2.3 What is a Transformation?

The MDA process, as described in section 1.2.1, shows the role that the various models, PIM, PSM, and code play within the MDA framework. A transformation tool takes a PIM and transforms it into a PSM. A second (or the same) transformation tool transforms the PSM to code. These transformations are essential in the MDA development process. In Figure 1-3 we have shown the transformation tool as a black box. It takes one model as input and produces a second model as its output.

When we open up the transformation tool and take a look inside, we can see what elements are involved in performing the transformation. Somewhere inside the tool there is a definition that describes how a model should be transformed. We call this definition the transformation definition. Figure 2-5 shows the structure of the opened up transformation tool.

![Figure 2-5. Transformation definitions inside transformation tools](image)

Note that there is a distinction between the transformation itself, which is the process of generating a new model from another model, and the transformation definition. The transformation tool uses the same transformation definition for each transformation of any input model.

In order for the transformation specification to be applied over and over again, independent of the source model it is applied to, the transformation specification relates constructs from the source language to constructs in the target language. We can, for example, define a transformation definition from UML to C#, which describes which C# should be generated for a (or any!) UML model. This situation is depicted in Figure 2-6.

![Figure 2-6. Transformation definitions are defined between languages](image)

In general, we can say that a transformation definition consists of a collection of transformation rules, which are unambiguous specifications of the way that (a part of) one model can be used to create (a part of) another model. Based on these observations, we can now define transformation, transformation rule, and transformation definition.

A transformation is the automatic generation of a target model from a source model, according to a transformation definition.

A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.
2.4 The Basic MDA Framework

In the previous sections, we have seen the major elements that participate in the MDA framework: models, PIMs, PSMs, languages, transformations, transformation definitions, and tools that perform transformations. All of these elements fit together in the basic MDA framework, as depicted in Figure 2-7. Although most of the terms have been defined in the previous sections, we summarize of the elements and their role below:

- A model is a description of a system.
  - A PIM is a Platform Independent Model, which describes a system without any knowledge of the final implementation platform.
  - A PSM is a Platform Specific Model, which describes a system with full knowledge of the final implementation platform.

- A model is written in a well-defined language.

- A transformation definition describes how a model in a source language can be transformed into a model in a target language.

- A **transformation tool** performs a transformation for a specific source model according to a transformation definition.

From the developer's point of view, the PSM and PIM are the most important elements. A developer puts his focus on developing a PIM, describing the software system at a high level of abstraction. In the next stage, he chooses one or more tools that are able to perform the transformation on the PIM that has been developed according to a certain transformation definition. This results in a PSM, which can then be transformed into code.
2.5 Examples

In this section, we show two small examples of applying MDA. The examples themselves are not very complex, even rather trivial. These examples do not show the advantages of MDA. The purpose is to show how the MDA framework is applied in some concrete examples. In both examples we take a look at a high-level PIM modeled in UML and a lower-level PSM for Java, also written in UML. Chapter 4 gives a more complex and realistic example.

2.5.1 Public and Private Attributes

In the first example we define a transformation between two UML models. The source model is a platform independent model, which is transformed into a lower level, more platform specific model for use with Java. The focus is on transforming public attributes into their respective get- and set-operations. One of the classes in the PIM is shown in Figure 2-8. The class Customer contains three attributes: title, name, and dateOfBirth. All attributes are declared public. In a high-level PIM it is normal to use public attributes. The meaning of a public attribute in a PIM is that the object has the specified property, and that this property can change value over time.

Figure 2-8. Platform independent model

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>+title : String</td>
</tr>
<tr>
<td>+name : String</td>
</tr>
<tr>
<td>+dateOfBirth : Date</td>
</tr>
</tbody>
</table>

In the PSM, where we model the source code instead of the business concepts, the use of public attributes is considered to be bad design. It is better to apply information hiding techniques and encapsulate the public attributes as shown in Figure 2-9. All attributes are private and all access to the Customer is directed through well-defined operations. This allows the customer object to have control over its use and over its change of attributes.

Figure 2-9. Platform specific model targeted at Java

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-title : String</td>
</tr>
<tr>
<td>-name : String</td>
</tr>
<tr>
<td>-dateOfBirth : Date</td>
</tr>
<tr>
<td>+getTitle() : String</td>
</tr>
<tr>
<td>+setTitle(title : String)</td>
</tr>
<tr>
<td>+getName() : String</td>
</tr>
<tr>
<td>+setName(name : String)</td>
</tr>
<tr>
<td>+getDateOfBirth() : Date</td>
</tr>
<tr>
<td>+setDateOfBirth(d : Date)</td>
</tr>
</tbody>
</table>

Both PIM and PSM are useful, as they provide the right level of information to different types of developers and other stakeholders in the software development process. However, there is a clear relationship between the models.
2.6 Summary

In this chapter we have seen the MDA framework and the role that the different elements play within it:

- A model is a description of a system
  - A PIM is a Platform Independent Model, which describes a system without any knowledge of the final implementation platform.
  - A PSM is a Platform Specific Model, which describes a system with full knowledge of the final implementation platform.

- A model is written in a well-defined language.

- A transformation definition describes how a model in a source language can be transformed into a model in a target language.

- A transformation tool performs a transformation for a specific source model according to a transformation definition.

The framework is neutral with respect to the languages that are used to model or code a system. In practice, UML is the most widely used modeling language and probably will be for a while.
Chapter 3. MDA Today

This chapter describes what has been achieved within the MDA at the time of writing. Hopefully, by the time you read this, the technology will have advanced somewhat further. This chapter gives an overview of the various OMG standards that can be used within the MDA, the tool support that you can expect, and development processes that you can use.
3.1 OMG Standards

The MDA is defined and trademarked by the OMG. We therefore first take a look at the OMG standards that play a role within the MDA framework.

3.1.1 OMG Languages

The OMG defines a number of modeling languages that are suitable to write either PIMs or PSMs. The most well-known language is UML. This is the most widely used modeling language.

The Object Constraint Language (OCL) is a query and expression language for UML, which is an integral part of the UML standard. The term "Constraint" in the name is an unfortunate leftover from the time when OCL was used to specify only constraints to UML models. Currently, OCL is a full query language, comparable to SQL in its expressive power.

The Action Semantics (AS) for UML defines the semantics of behavioral models in UML. Unfortunately, it defines the behavior at a low-level foundation. Therefore, it is not directly suitable for writing PIMs. It lacks the higher level of abstraction that is necessary. The AS is not a language that can be used directly by a modeler, because it does not define a concrete syntax; you cannot write down anything at all in a standardized way.

UML includes a profile mechanism that enables us to define languages derived from the UML language. The language defined in the profile is a subset of UML with additional constraints and suitable for a specific use. It uses the UML diagrammatic notation and OCL textual queries, and looks like UML. Many such profiles are standardized by the OMG; others are not standardized, but publicly available. Official OMG profiles include the CORBA Profile, the Enterprise Distributed Object Computing (EDOC) Profile, the Enterprise Application Integration (EAI) Profile, and the Scheduling, Performance, and Time Profile. More profiles are being developed and will be standardized in the coming years. Profiles are usually suitable for writing PSMs.

The UML/EJB Mapping Specification (EJB01) is an example of a profile that is standardized through the Java Community Process. Several profiles for other programming languages, like Java, C#, and so on, are defined by individual organizations and tool vendors.

Another language that is defined by the OMG is the Common Warehouse Metamodel (CWM). This is a language specifically designed to model data mining and related systems.

Chapter 11 describes the various OMG languages and their role in MDA in more detail.

3.1.2 OMG Language and Transformation Definitions

Languages used within the MDA need to have formal definitions so that tools will be able to automatically transform the models written in those languages. All of the languages standardized by the OMG have such a formal definition. The OMG has a special language called the Meta Object Facility (MOF), which is used to define all other languages. This ensures that tools are able to read and write all languages standardized by the OMG.

The transformation definitions used in the MDA framework are currently defined in a completely nonstandardized way. To allow standardization of these transformation definitions, the OMG is currently working on a standard language to write transformation definitions. This standard is called QVT, which stands for Query, Views, and Transformations. At the time of writing, the Request for Proposals (RfP) for QVT has been published. QVT is still being worked on by OMG members so we don't yet know exactly how the finished specification will look.
3.2 UML as PIM Language

As seen in section 1.3.1 the level of completeness, consistency, and unambiguity of a PIM must be very high. Otherwise, it is not possible to generate a PSM from a PIM. Let's investigate to what extent UML is a good language for building PIMs.

3.2.1 Plain UML

The strongest point in UML is the modeling of the structural aspects of a system. This is mainly done through the use of class models, which enables us to generate a PSM with all structural features in place. The example in Chapter 4 shows how this is done.

UML has some weak points that stop us from generating a complete PSM from a PIM. The weak area in UML is in the behavioral or dynamic part. UML includes many different diagrams to model dynamics, but their definition is not formal and complete enough to enable the generation of a PSM. For example, what code (for any platform) would you generate from an interaction diagram, or from a use case?

Plain UML is suitable to build PIMs in which the structural aspects are important. When a PSM is generated, a lot of the work still remains to be done on the resulting model, to define the dynamic aspects of the system.

3.2.2 Executable UML

Executable UML (Mellor 2002) is defined as plain UML combined with the dynamic behavior of the Action Semantics (AS). The concrete syntax used in Executable UML has not been standardized.

The strength of plain UML, modeling the structural aspect, is present in Executable UML as well. Executable UML to some extent mends the weak point in plain UML, the modeling of behavior. In Executable UML the state machine becomes the anchor point for defining behavior. Each state is enhanced with a procedure written in the AS.

In principle, Executable UML is capable of specifying a PIM and generating a complete PSM, but there are a few problems:

- Relying on state machines to specify complete behavior is only useful in specific domains, especially embedded software development. In other, more administrative, domains the use of state machines to define all behavior is too cumbersome to be used in practice.

- The AS language is not a very high-level language. In fact, the concepts used are at the same abstraction level as a PSM. Therefore, using Executable UML has little advantage over writing the dynamics of the system in the PSM directly. You will have to write the same amount of code, at the same level of abstraction.

- The AS language does not have a standardized concrete syntax or notation; therefore, you cannot write anything in a standard way.

3.2.3 UML–OCL Combination

Using the combination of UML with OCL to build a PIM allows for PIMs that have a high quality; that is, they are consistent, full of information, and precise. The strong structural aspect of UML can be utilized and made fully complete and consistent. Query operations can be defined completely by writing the body of the operation as an OCL expression. Business rules can be specified using OCL, including dynamic triggers.

The dynamics of the system can be expressed by pre- and post-conditions on operations. For relatively simple operations the body of the corresponding operation might be generated from the post-condition, but most of the time the body of the operation must be written in the PSM. In that case, generating code for the pre- and post-condition ensures that the code written in the PSM conforms to the required specification in the PIM.

Although the dynamics of the systems still cannot be fully specified in the UML–OCL combination, the combination of UML class models with OCL allows for a much more complete generation of PSMs and code than does plain UML. The use of the combination of UML and OCL is at the moment of this writing probably the best way to develop a high quality and high level PIM, because this results in precise, unambiguous, and consistent models that contain much information about the system to be implemented.
3.3 Tools

Ever since MDA became a popular buzzword, vendors have claimed that their tools support MDA. Tools that were on the market for many years, even before the name MDA was invented, make these claims. Most of these claims are true, in the sense that they support some aspect of MDA. We will use the MDA framework as shown in Figure 2-7 to analyze what level of support a tool really offers.

3.3.1 Support for Transformations

Support for MDA comes in many different varieties. Simple code generation from a model has been done for more than a decade, and lies well within the boundaries of MDA. The demands that MDA places on models and transformations of models in the ideal situation, however, are very high. In this section we will focus on the support for transformations that tools can offer.

PIM to PSM Transformation Tools

This type of tool transforms a high level PIM into one or more PSMs. This type of tool is barely available at the time of writing, although some tools offer minimal functionality in this area.

PSM to Code Transformation Tools

The most well-known support for MDA is given by tools that act as black-box PSM to code transformation tools. They have a built-in transformation definition and take one predefined type of model as source and produce another predefined type as target. The source model is a PSM, while the target is the code model. In fact, code generation from traditional CASE tools follows this pattern.

Several tools persist the relationship between the PSM and code, and enable you to see changes in either of the models reflected in the other model immediately after the change. This is possible because of the fact that the PSM and code are relatively close to each other, and have almost the same level of abstraction.

PIM to Code Transformation Tools

Another type of tool supports both the PIM to PSM and the PSM to code transformation. Sometimes the user will only see a direct PIM to code transformation and the PSM is left implicit. With this type of tool, the source and target language and the transformation definition are built into the tool that acts as a black box.

UML is usually used as the PIM language. Dynamic functionality that cannot be expressed in UML needs to be added manually in the generated code.

Tunable Transformation Tools

Tools should allow for some tuning or parameterization of a transformation. Access to the transformation definition to adjust it to your own requirements is usually not available. The best one can get today is a transformation definition written in an internal tool-specific scripting language. It is a time-consuming task to make changes to such a script. Because there is no standard language to write transformation definitions yet (see QVT in section 3.1.2), transformation definitions are by definition tool-specific.
3.4 Development Processes

The MDA does not require a specific process to be used for software development. Processes are not standardized by the OMG, so you are free to choose your own. Of course, the central position of models and transformations in MDA should be reflected in such a process. We will take a short look at some of the more popular processes and show how they can be used for MDA development.

3.4.1 Agile Software Development

A current trend in the software development process is to minimize the amount of effort and time spent in building models that will only serve as documentation, even though they do capture some of the interesting aspects of the software to be built. The purpose is to ensure that software is delivered that works for the users. Since requirements continuously change, the software that is being developed must change accordingly. The ability for a software project to accommodate changes in a flexible and immediate way is the core aspect of Agile Software Development.

According to Cockburn (2002), "Working software is valued over comprehensive documentation." Well, we couldn't agree more. At the Web site http://agilemanifesto.org you can find the Agile Manifesto describing the agile principles. On this website you may post a quote why you like this approach. The quote from one of the authors of this book is:

As co-author of the UML standard, people usually think I love large and detailed models. The contrary is true, a model is only worth building if it directly helps to achieve the final goal: building a working system. With the emergence of MDA tools, it becomes possible to directly move from model to code. This "promotes" models from being merely documentation to becoming part of the delivered software, just like the source code.

Because changing a model means changing the software, the MDA approach helps support agile software development.

3.4.2 Extreme Programming

The XP approach is a very popular way of working, where the focus lies on writing code in small increments, such that you have a working system all the time. Each new requirement must be accompanied by an explicit test case, which is used to test the software. When adding new functionality, all previous tests are run in addition to the new tests, to ensure that existing functionality is not broken.

As we explained in section 1.1.1, the focus on code only is too limited. Code must be augmented with so-called "markers" that document the code at a higher level. In extreme programming this is often seen as overhead. When we realize that these markers may take the form of MDA models that directly transform into code, creating these markers is not overhead anymore. On the contrary, the high-level models help to develop the software faster.

This means that we can bring XP to a higher abstraction level, and we might want to talk about "Extreme Modeling."

3.4.3 Rational Unified Process (RUP)

The RUP is a process that is much more elaborate, or much heavier, than the agile or extreme processes. Project managers often like these larger and more structured processes because they give a better feeling of control over the project. Especially for large projects, it is clear that a more elaborate process than extreme programming is needed. On the other hand, many people consider the RUP process as being too large and unwieldy, favoring bureaucratic development of large stacks of paper over "real" software development, or in a more positive view, a balanced approach between models and code.
3.5 Summary

In this chapter we have seen that the MDA framework can be populated by a number of different OMG standards. At the same time standards like the MOF and UML profiles allow non-OMG organizations to develop their own standards that will fit seamlessly in the MDA framework.

Tools are of great importance to the success of MDA. A wide spectrum of functionality is needed in an MDA environment, including traditional tools. Each tool may provide some or more of the functionality needed, and there are standardized ways that tools can communicate with each other.

Note that MDA is an emerging technology that is still in its infancy. Neither the languages nor the tools are developed enough to achieve the hundred percent code generation that is promised by MDA. There is always a need for manual change within the generated code. However, tools can provide enough functionality to make a significant impact on your software development process.

MDA can be used in existing software development processes. What is needed is more focus on the models that are developed to ensure that they are complete and consistent enough to be used in MDA transformations.

In Chapter 12 we take a look at the promise that MDA brings us. When the languages and tools become more mature, MDA has the ability to become a major paradigm shift in the software development community.
Chapter 4. Rosa's Application of MDA

Together with the next two chapters, this chapter gives a concrete example of the MDA process. In order to show the power of the MDA approach, the system described is not trivial. It is not a toy system, but a real-life example. We will demonstrate how a fairly simple PIM is transformed automatically into rather complex PSMs and code. The complexity of the complete example is considerable. However, the example is not completely detailed out in all parts of the system in order to limit the size of this book. In this chapter the requirements of the example system are stated and an overview is given of the models and transformations involved.

If you are confident that you completely understand the concepts from the earlier chapters, then you may skip the following three chapters, and go directly to Chapter 7.
4.1 Rosa's Breakfast Service

The example we will be exploring in this book is the ordering system for Rosa's Breakfast Service. The example system described in this and subsequent chapters is implemented using the OptimalJ tool. You can download OptimalJ and the example at the Web site http://www.klasse.nl/mdaexplained.

4.1.1 The Business

Rosa has founded a company that supplies a complete breakfast delivered to the homes of its customers. Customers can order from one of the breakfast menus on Rosa's Web site, indicate the hour and place of delivery, give their credit card number, and the ordered breakfast will be delivered. Rosa's slogan is: "Surprise your husband, wife, lover, valentine, mother, father, or friend on their special day while still enjoying the comfort of your bed."

Rosa has composed a series of breakfasts, each of which comes with special decorations. For instance, the Valentine breakfast is served on plates decorated with small hearts and cupids, with matching napkins. The items served are set by the choice of breakfast. For instance, if you choose the French breakfast, you will get one cup of coffee, one glass of orange juice, two croissants and one roll, butter, and jam. But if you choose the English breakfast, you will get two fried eggs and bacon, three pieces of toast, marmalade, and a pot of tea. The Champagne Feast breakfast, which always serves two persons, includes a bottle of champagne, four baguettes, a choice of french cheese and pâtés, and a thermos bottle filled with coffee (to sober up afterwards). Orders can be filled for any number of people, where any in the party may order a different breakfast.

The items served are set, but customers can indicate the style in which the breakfast is served. They can choose between simple, grand, and deluxe. A simple breakfast is served on a plastic tray with carton plates and paper napkins. The orange juice glass, when included, is made of plastic, too. A grand breakfast is served on a wooden tray with pottery plates and cups, and simple white cotton napkins, and the glass is made of real glass. A deluxe breakfast is served on a silver tray with a small vase with some flowers. The plates are fine pottery plates and the napkins are decorated, made from the finest linen. Obviously, the price of the breakfast is higher when the serving style is better. Some breakfast types, like the Champagne Feast, can be ordered in the grand or deluxe style only.

Rosa has ten employees that come in her kitchen at half past five in the morning and work until noon. Five of them take care of deliveries, and five do the cooking and preparation of the breakfasts. Rosa's kitchen is located next to a bakery. The first thing Rosa does in the morning is get fresh bread from the bakery. All other ingredients are kept in supply. Twice a week their inventory is resupplied.

Rosa wants to give her customers a bit of flexibility. Customers may, after choosing a standard breakfast as basis, decide to put in extra comestibles, alter the amount of certain parts, and even remove parts from the breakfast. So, if you like the Champagne Feast breakfast, you may order two bottles of champagne instead of one, add another baguette, and leave out the smelly cheese and the coffee (coffee won't help after two whole bottles of champagne anyhow).

4.1.2 The Software System

In this example we are not very interested in the delicious breakfasts Rosa makes; instead, we look at the system needed to support Rosa's business. The ordering system is a standard Web-based, three-tier application. There will be two different Web interfaces, one for the customers, and one for Rosa's employees to indicate which breakfasts they need to make and deliver. If the customer agrees, his or her name and address will be kept in the system. This will enable Rosa to give a discount to regular customers.
4.2 Applying the MDA Framework

Rosa will be interested only in the final system. But, because this is an example of how the MDA framework can be applied, we are interested in the process of building Rosa's Breakfast System. We will dissect this process into parts that have meaning within the MDA framework. We must identify which PSMs and code models should be delivered and which transformation definitions should be used to generate the PSMs and code models. All elements of the MDA framework used in this example are shown in Figure 4-1. The models are shown as rectangles and the transformations are shown as arrows.

Figure 4-1. Three levels of models for Rosa's Breakfast Service

4.2.1 The PIM and PSMs

To start the MDA process we need to build a platform-independent model that comprises the whole of Rosa's business. For the sake of simplicity, our PIM will be written in plain UML. This is the only model that the developer will create completely "by hand." The other models are mostly generated.

Because each tier is implemented using a different technology, we need three PSMs, one for each tier. The first PSM specifies the database, and is described by a relational model depicted in an Entity-Relationship diagram.

The PSM for the middle tier, which we call the EJB model, is written in a language that is a UML variant. It uses classes, associations, and so on, as in UML, but there are a number of stereotypes defined explicitly for the EJB platform.

The PSM for the Web interface is also written in a UML variant. This language uses different stereotypes than the UML variant used for the EJB model. Neither UML variant is standardized as a profile. They are small and simple, so we will not give an explanation of these UML variants.

4.2.2 The PIM to PSM Transformations

Because three PSMs need to be generated, we need three PIM to PSM transformations:
4.3 The PIM in Detail

The PIM for Rosa's Breakfast System is depicted in Figure 4-2. The PIM is the only model that must be created by humans in a creative process. To find out how to build such a model you can read a large number of books on UML and modeling. Here we assume that the creative process has been successfully completed with the PIM in Figure 4-2 as the result.

![Figure 4-2. PIM of Rosa's Breakfast Service](image)

In the PIM every standard breakfast contains a number of parts; each part indicates the amount in which a certain comestible is present in the breakfast. Every order consists of a number of breakfasts. The price of each breakfast is determined based on the chosen style and the price of the chosen standard breakfast. The price of the order is simply the addition of the prices of all breakfasts plus a small delivery fee.

The model in Figure 4-2 defines the breakfast services independently from any specific technology, so indeed, it is a PIM. But Rosa does not want a model, she wants a running system. Therefore, we need to transform the PIM into a number of PSMs, taking into account the relationships between these PSMs.
4.4 Summary

In order to show the power of the MDA approach, an example system is described that is not trivial. The example is a system that supports Rosa's Breakfast Service, a company that supplies a complete breakfast delivered to the homes of its customers.

A fairly simple PIM specifying the system is described. This PIM will be automatically transformed into a Relational PSM, a Web PSM, and an EJB PSM. The three PSMs will then be transformed into three separate code models. The next two chapters describe the transformations to be performed.
Chapter 5. Rosa's PIM to Three PSMs

This chapter gives insight into the three PIM to PSM transformations that are needed to implement the system for Rosa's Breakfast Service.
5.1 The PIM to Relational Transformation

The transformation rules for generating relational database models typically take care of a consistent object-relational mapping. Although most of these rules are rather straightforward and well-known (Blaha 1998), it can be a hard job to execute them manually. Small changes in the PIM can have a large effect on the relational model. For instance, changing the type of an attribute in the PIM from a simple data type to a class means introducing a foreign key in the corresponding table. The simple data type can be mapped directly to a column in a table. But if the data type is a class, this class will be mapped to a table itself. The column will now have to hold a reference (foreign key) to a key value in that other table.

What rules should be used to generate a relational model? Note that we want to formulate rules that will apply to any UML model, not only to Rosa's PIM. First, we must decide how the basic data types are being mapped. This is a fairly simple task. All we need to do is find the right corresponding data type in the relational model. Data types can be mapped according to the following rules. Note that we define an arbitrary length for each of the relational data types.

- A UML string will be mapped onto a SQL VARCHAR(40).
- A UML integer will be mapped onto an INTEGER.
- A UML date will be mapped onto a DATE.

But what do we with the Address? In the PIM the address is not a class, but a data type, a struct containing only attributes, and no operations. We have two options: either make a separate table for every data type, or inline the data type into the table that holds the attribute. Here we choose the latter option, because it will simplify the alignment with the EJB model. So for struct data types we have the following rule:

- A UML data type that has no operations will be mapped onto a number of columns, each representing a field in the data type.

Second, every class should be transformed into a table, where all attributes are fields in the table (rules ClassToTable and AttrToCol). When the type of the attribute is not a data type but a class, the field in the table should hold a foreign key to the table representing that class (rule AttrToFkey). Note that we do not yet take into account the possibility that the multiplicity of the attribute is more than one.

The third step is more complicated. Associations in the UML model need to be transformed into a foreign key relation in the database model, possibly introducing a new table. Note that we have several possibilities for the multiplicities of an association from class A to class B in a UML model:

- The multiplicity at A is zero-or-one.
5.2 The PIM to EJB Transformation

To complete the system for Rosa's Breakfast Service we need to generate an EJB PSM. We will make a number of architectural choices in the way that we use the EJB framework. These choices are specific for Rosa's Breakfast Service. Depending on the project requirements and the possibilities offered by the available tools, you will need to make your own choices in your own situation. We start out by explaining some aspects of the choices we have made regarding the EJB PSM.

5.2.1 A Coarse Grained EJB Model

The EJB model for Rosa's Breakfast Service is structured in a rather different manner than the PIM. We could have built a component model for Rosa's Breakfast Service by simply generating a component for each class. However, it is crucial for the performance of EJB components in a distributed environment with remote access that the communication between components remains minimal.

The attributes of an object could be exchanged one by one, where each get- or set-operation on an attribute value causes a remote method invocation. This is a straightforward, but rather naive approach. To minimize the frequency of interaction between components, wherever possible, it is better to exchange all attributes of an object in one remote call. This results in messages that are more complex, and include a relatively high amount of data, and in component interfaces that have a relatively low number of operations, relieving the burden on the communication network.

Furthermore, it is better to keep the number of components relatively small, because intercomponent communication will not burden the network, whereas intracomponent communication will. To minimize the number of components, we can combine closely related classes into one component, instead of having a component for each of them separately. We call a component model that adheres to these principles a coarse grained component model.

A coarse grained component model is a model of components where the components are large and have infrequent interaction with a relatively high amount of data in each interaction.

In contrast to the coarse grained component model there is the fine grained component model.

A fine grained component model is a model of components where the components are small and have frequent communication with a low amount of data in each interaction.

For Rosa's Breakfast Service we have chosen to use a coarse grained EJB component model. There are a number of books where you can find out more about component models (for example, Cheeseman 2001, Szyperski 1998, and Allen 1998). This book does not address that subject.

The interfaces of the coarse grained components have methods to exchange complete sets of associated objects, and they do not have methods for accessing the individual attributes and associations between these objects. Therefore, a number of classes in the source model must be clustered into so-called EJB data schemas.

An EJB data schema is a set of classes, attributes, and associations that is served by an EJB component as a whole.
5.3 The PIM to Web Transformation

The Web model specifies the definitions of the Web components. The Web components serve HTML content to the user. Each component serves a subset of the classes and associations of the system. Extra details are added to the Web model to define the layout and the user interactions of the HTML pages. In this example, the Web components serve the same classes as in the data schemas of the entity beans.

Web components are defined similarly to EJB components. The served classes and associations are defined in Web data schemas similar to the EJB data schemas. The most important differences are:

- The data types in the Web model define user presentation and interaction details.
- In the Web application model there are no key classes; instead, the key classes of the EJB model are referenced.
- Web actions are added that define actions that can be triggered by the end-user.

The Web data schemas define which information is shown that can be altered by the user. One Web data schema is typically presented to the user in more than one HTML page. A user may create, query, alter, and delete objects from the domain. Which changes the user may execute is defined in properties of the elements of the Web data schemas.

5.3.1 The Transformation Rules

The rules for generating the Web application model from the UML model are almost equal to the ones for generating the EJB application model. Again, we will leave out the words UML and Web whenever the source and target model are clear.

1. Each class that is not part of another class is transformed into a component and a data schema. The component is set to serve the data schema.

2. Each class is transformed into a data class residing in a data schema that is generated from the class that is the outer-most composition of the transformed class.

3. Each association is transformed into an association within a data schema that is generated from the class that is the outer-most composition of the transformed association.
5.4 The Communication Bridges

The MDA process has not been completely described if we do not reference the generation of the communication bridges between the relational model and the EJB model and between the Web model and the EJB model.

In Figure 4-1 on page 46, the communication bridges are shown by arrows. This means that there is a direction in the relation between the two models. The Web model uses (and knows) the EJB model, and the EJB model uses (and knows) the relational model.

Both bridges are simple and have been explained. The data storage for the EJB components is provided by a relational database, structured according to the generated relational model in Figure 5-1. The EJB-Relational bridge is constituted by the relation between the table generated for a UML class and the EJB data class that is generated from the same UML class. The relationship between the generated relational model and the generated EJB component is shown in Figure 5-5. The figure depicts the EJB component model for Rosa's Breakfast Service without showing the EJB data schemas, but with all dependencies on the tables of the relational model of Figure 5-1.

The bridge between the Web model and the EJB model is constituted by the links to the EJB key classes and EJB components as shown in Figure 5-4. Note that both communication bridges are relationships between the PSMs. This relationship will need to be preserved when the PSMs are transformed into code.
5.5 Summary

As pointed out earlier, the MDA especially pays off when the transformation process is not trivial, or when the transformation is well-known but involves much work. In the example given in this chapter, both situations are present. The UML-Relational mapping is worthwhile because it speeds up the process. The UML-EJB mapping is clearly a non-trivial mapping, where the transformation rules supply much knowledge of the platform.

In general, we can say that when the structures of the source and target language differ greatly, the transformation becomes more complex. Small changes in the source model may have a huge impact on the structure of the target models and thus on the implementation code.

The process of implementing a PIM is greatly improved by the application of MDA in terms of quality, because we are forced to specify transformation definitions that are generally applicable.
Chapter 6. Rosa's PSMs to Code

To really implement Rosa's Breakfast Service we have to generate compileable and executable code. This chapter describes the code models and the transformations from the PSMs to the code models. Each of the three PSMs is transformed into a separate code model that is applicable for the specific technology.
6.1 Relational Model to Code Transformation

The result of the transformation from UML to relational model that is described in this section is a PSM, which is still a model. It is not the code to create the database with. Luckily, the PSM is closely linked to the relational database platform, and uses platform-specific concepts and constructs. From this model, it is easy to generate code.

From the relational model, a pair of SQL scripts is generated. One script is for creating the tables and the other script for dropping the tables. We will examine the creation script only, because the drop script is very simple and does not add much value to the example. The transformation is rather straightforward because the structure and amount of detail of the relational model is already similar to the SQL language. The following rules are used to generate the SQL creation script from the Relation model.

1. For each table, generate a "CREATE TABLE" text, followed by the name of the table, and a "{ ", then execute rule 2, followed by rule 3, and end with "};".

2. For each column in the table, generate the name of the column, followed by the name of the type, and (optional) size of the column, then generate "NOT" if the column may not have the NULL value and end with "NULL,".

3. Generate a "PRIMARY KEY (" text, followed by a comma-separated list of the names of the columns of the primary key, and end with ")".

Fragments of the generated code can be found in Appendix B.
6.2 EJB Model to Code Transformation

To explain the EJB model to code transformation in detail, we must zoom in on the overall picture of the relations between all models used in the MDA process. Figure 6-1 shows a refinement of Figure 4-1. In it a distinction is made between the EJB PSM that represents the coarse grained component model, as described in section 5.2, and an EJB PSM that is much closer to the generated code. This PSM contains class definitions, and is called the EJB Class PSM. The EJB Class PSM consists of class diagrams where the classes relate one-to-one to the actual classes in the Java code. You could say that this model is on the same abstraction level as the EJB source code, but in a diagrammatic presentation.

![Figure 6-1. The models in the MDA process in detail](image)

In this section, we describe the transformation from the EJB Component PSM to the EJB Class PSM. The EJB Class PSM is written in the language defined by another UML variant, the standardized EJB Profile (Java Community Process document JSR26, 2001). Because the generation of code from the EJB Class PSM is very simple, we do not describe it. Fragments of the generated code can be found in Appendix B.

Before explaining the transformation rules, a small introduction to the specific aspects of entity beans in EJB is given.

### 6.2.1 Some Remarks on EJB Code

According to the EJB specification (Sun Microsystems, Enterprise JavaBeans Specification, Version 2.1, 2002), a person (or persons) who develops entity beans must provide the following:

- Entity bean class and all classes it depends on.
6.3 The Web Model to Code Transformation

6.3.1 The Structure of the Web Code

To implement the Web components, we have to generate code that handles user requests from the Internet and that produces HTML as a response. In general, the complexity of this code can be high. A good MDA tool provides its user with transformations that are highly complex, thus creating a fully functional and working application, using well-established coding patterns. In this book, however, we generate Web code with simple functionality using a simple coding pattern. Although this is clearly not a sufficient MDA solution, we avoid long discussions about complex Web coding solutions.

In this example, the Web part of the application is implemented according to the J2EE standard for Web tiers (Sun Microsystems, Java 2 Platform, Enterprise Edition Specification, Version 1.3, 2001). We generate code that generates simple HTML pages that hold query results only. The components are implemented using JSP (Sun Microsystems, JavaServer Pages Specification, Version 1.2, 2001). Each request from a user instantiates one JSP and the resulting HTML is sent back to the browser. We generate exactly one JSP file for each Web component. At run-time, the JSP produces an HTML page containing a table with rows that correspond to all objects of one type (e.g., all customers) and columns corresponding to the attributes of that type (e.g., address and account number).

The JSPs access the data from the EJB components by iterating over a set of EJB Data Objects and getting the values of the attributes provided by them. The iteration is implemented in embedded Java code. The JSP code uses the EJB data object manager for the retrieval of the data objects. JSP supports the access of get methods for the attributes by simply stating the names of the attributes.

6.3.2 The Transformation Rules

The following rules are used to generate the JSP code from Web Components and the Web Data Classes:

- For each Web Component one "query" JSP file is generated with the same name as the Web component.
- Within each query JSP one header is generated containing one name per attribute.
- Within each query JSP one "useBean" element is generated to get access to the remote interface of the corresponding EJB entity bean.
- Within each query JSP one iteration is written in embedded Java using the EJB data object manager to get the collection and iterate over the set of EJB data objects.
- Within each iteration one HTML row is generated with a "getProperty" element for each attribute of the served data class.

In Appendix B, you can find some fragments of the generated code.
6.4 Summary

In this chapter, we have seen that the transformation of the relational model to code is simple, because the relational model contains structures that can be mapped one to one to the structures used in the SQL language.

To transform the EJB model from section 5.2 to code, we introduced an intermediate model, called the EJB Class PSM. The EJB Class PSM was written in the language defined by another UML variant, the standardized EJB profile (Java Community Process document JSR26, 2001). Because the generation of code from the EJB Class PSM is very simple, we did not describe it.

In order to avoid the complexity that Web code can bring, we have defined a simple transformation from Web model to JSP and HTML code.
Chapter 7. More on Transformations

This chapter takes a closer look at the anatomy of a transformation. We will see that a transformation needs to be more than the process of generating a target model if we want to be able to maintain consistency between source and target models.
7.1 Desired Features of Transformations

In section 2.3 we defined a transformation as the generation of a target model from a source model. This means that transformations are purely processes. The process is described by a transformation definition, which consists of a number of transformation rules, and is executed by a transformation tool. In an MDA approach there are a number of features of the transformation process that are very desirable. We name them in order of importance:

1. Tunability, which means that although the general rule has been given in the transformation definition, an application of that rule can be tuned; for example, when transforming a UML String to a VARCHAR in an entity-relationship model, you might want the length of the VARCHAR to differ for each occurrence of a UML String.

2. Traceability, which means that one can trace an element in the target model back to the element(s) in the source model from which it is generated.

3. Incremental consistency, which means that when target-specific information has been added to the target model and it is regenerated, the extra information persists.

4. Bidirectionality, which means that a transformation can be applied not only from source to target, but also back from target to source.

Each of these features puts demands on the transformations. In the next sections, we further investigate the desired features. Section 7.6 describes the implications of the features on transformations.
7.2 Controlling and Tuning Transformations

A user of a transformation tool will most likely want to have some control over the transformation process. A response of an experienced software developer to the example in section 5.1 was: "But how do I tell the tool that I want the name of Comestible to be transformed to VARCHAR(20), and the transportForm to VARCHAR(50)?" This developer wants to influence some details of the transformation process, which, in our view, is a legitimate request.

There are several ways in which a user may control the transformation process. In the following sections we describe them all. When choosing your transformation tool, be aware that the options the tool offers to exercise control can be very different from tool to tool.

7.2.1 Manual Control

The most direct control a user can have over a transformation is to be able to manually define which model element is transformed by which transformation rule. This is by definition the most flexible solution, but it is error-prone and much work for the user. Imagine taking these decisions for even a small class model with ten classes, 40 attributes, 12 associations, and 100 operations. You'll be asked to make at least 162 choices. Most users will end up hitting the "OK" button blindly. For larger models, the situation becomes impossible to handle. Therefore, we need to take a look at solutions that scale up in practice.

7.2.2 Conditions on Transformations

Another way of giving the user the power of control is by attaching a condition to each transformation rule. This condition describes when the rule should be applied. A transformation rule will not simply look like: "Every class is transformed into ...," but something like, "Every class with stereotype <<persistent>> is transformed into ..." In principle, all properties of the model elements in the source model can be used in this condition. Even very detailed rules like, "Every class with string 'THING' in its name is transformed into ...," or "Every class associated with a class named 'XYZ' is transformed into ..." should be expressible.

Preferably we should make the conditions mutually exclusive. This allows full automation of the transformation process. This approach can be combined with the manual control for those cases where more than one condition holds.

7.2.3 Transformation Parameters

The transformation process can also be tuned by using parameters. Transformation definitions can be parameterized to make them conform to a certain style. For example, when transforming a public attribute to a private attribute with getter and setter operations the exact prefix strings (usually get and set) can be defined as parameters of the transformation. When a transformation is performed, the user needs to set these parameters. Providing default values can help to avoid mandatory lengthy parameter lists. Other typical parameters can be the length of fixed-length data types, and so on.

In Rosa's example in section 5.1, the transformation definition from PIM to the Relational PSM transformed each "String" in the PIM into a VARCHAR(40) in the PSM. The choice of the length 40 is rather arbitrary and could better be defined as a parameter of the transformation. This allows the developer to influence selected details that are made during the transformation process.
7.3 Traceability

In a number of situations, traceability is a helpful asset in the application of MDA. In Chapter 3, we saw that a PIM usually does not contain all information necessary to implement the complete system. A user must fill in the gaps in the PSM manually. When the user has the possibility of changing the PSM, he can potentially also change parts of the PSM that are generated. Obviously, this is a source of trouble.

The least a tool should do is warn its user that the changed part is generated from a PIM. It would be better when the tool can suggest further changes in either PSM or PIM. When, for instance, the user changes the name of an operation in the PSM, the tool might suggest that the corresponding operation in the PIM be renamed as well. An even better tool could perform the renaming. To be able to offer this type of support to its user, the tool needs to trace back the operation in the PSM to an operation in the PIM.

Another situation where traceability is useful is when the project is well underway. The PIM is developed, the PSM is generated and its gaps are filled, code is generated, and then some requirements change. Often it is easier to indicate what part of the PIM is affected by the changed requirements than to tell which part of the code must be adapted. When parts of the code and parts of the PSM can be traced back to elements in the PIM, an impact analysis of the requested changes is far easier to make.

In the case where the system has been delivered and bug reports come in, the parts of the code that are erroneous can be found by looking at the elements of the PIM that represent the faulty functionality. Even when bugs are fixed "quick and dirty" in the code, traceability is an asset, because the necessary changes to the PSM and PIM can be listed by the tool or even automatically executed.
7.4 Incremental Consistency

When a target model has been generated, it usually needs some extra work, such as filling in the code of an operation, or fine tuning a user interface for optimal use. When you regenerate the target model (because of changes in the source model), you want the extra work done on the original target model to remain. This is what we call incremental consistency.

When a change in a source model takes place, the transformation process knows which elements in the target model need to be changed as well. An incremental transformation process can replace the old elements with the newly generated ones, while keeping the extra information in the target model in place. This means that changes in the source model have minimal impact on the target model.

We can see why minimizing these changes is important in Rosa's Breakfast Service, where one of the target models is a relational model of a database. Rosa might already have a fully populated database in use. If we simply regenerate the complete target model and lose the explicit relationship with the existing target model, we have created a problem. We can create a new database corresponding with the new relational model, but we need to do a migration from all data in the existing database to the new one. If we use the incremental approach, we can keep all the data of unchanged parts of the relational target model without any migration. We only need to migrate the parts whose model has changed.

Note that both traceability and incremental consistency are most relevant as long as the PIM is not a complete description of the system and the transformation tools are not as good as current day compilers. Both are less relevant in the transformation from PSM to code. The implications of these features on PIM to PSM transformations are addressed in section 7.6.
7.5 Bidirectionality

The last of the desired features of transformations, bidirectionality, or transformations that can work in both directions, has the lowest priority. Bidirectional transformations can be achieved in two ways:

- Both transformations are performed according to one transformation definition.
- Two transformation definitions are specified such that they are each other's inverse.

The first way is shown in Figure 7-1. Because of the difference in source and target language, it is very difficult to build a transformation definition that works two ways. An example is where we transform a statechart in a business model to a plain Java programming model. In our transformation, we might transform a state to a boolean typed attribute. It is generally not possible to regenerate the same statechart from a Java code model. There is no way to find from the Java code which attributes should be states in the business model. Even though the two models might be semantically equivalent, the abstractions from the business model are lost in the code model.

![Figure 7-1. Bidirectional transformation](image)

In the second way, it is very difficult to be sure that both transformation definitions are each other's inverse. Take, for example, the transformation of public to private attributes from section 2.5.1. We cannot prove that the second set of rules is the inverse of the first set of rules. We simply believe that they are. For more complex transformation definitions, it is more difficult to prove, and our beliefs are tested severely.

As we can see, it is very difficult to define bidirectional transformations. But there is a second reason for giving bidirectionality a low priority. If additional information is added to the target language, or if there is information in the source language that is not mapped to the target, bidirectionality is impossible to achieve. For example, when we transform a business model to a relational model, we only transform the structural information from the source model. All dynamic information in the source model is ignored. It is impossible to regenerate the complete business model from the relational model.

A third reason for giving bidirectionality a low priority is that complete bidirectional transformations between models is only possible if the expressive power of the source and target modeling language is identical. This means that the abstraction levels of both source and target model are equivalent. The fact that a PIM is at a higher abstraction level than a PSM is one of the essential characteristics for getting added value out of the usage of MDA. If bidirectional transformations imply that the source and target models, i.e., the PIM and PSM, are at the same abstraction level,
7.6 Implications on Transformations

What are the implications of the desired features on the transformations? In short, there are two things we require of a transformation:

- It should have parameters with which you can tune it.
- It should maintain a persistent source-target relationship.

7.6.1 Transformation Parameters

In order to achieve the first requirement, we add parameters to the transformation rules in the transformation definition. For each parameter we can set a default value. For instance, the rule from section 5.1 for transforming Strings to VARCHARs becomes:

- A UML string is mapped onto a SQL VARCHAR (parameter i: Integer [default=20] )

Every application of this rule in the transformation process needs to have its own value for parameter i. In most cases the default value will be used, but for the tool to be able to support the requested user control, variation of the default must be possible. The tool might, for instance, produce a list of UML Strings in the source model that are transformed according to the rule above. The user will indicate for which of the UML Strings a different parameter value should be used. In Rosa's example the following table could be shown, in which the user has already changed the value of the parameter for Comestible.transportForm.

<table>
<thead>
<tr>
<th>Name of source element</th>
<th>Transformed to</th>
<th>Parameter value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comestible.name</td>
<td>VARCHAR</td>
<td>20</td>
</tr>
<tr>
<td>Comestible.transportForm</td>
<td>VARCHAR</td>
<td>40</td>
</tr>
<tr>
<td>StandardBreakfast.name</td>
<td>VARCHAR</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 7-1. Using Transformation Parameters for the Length of a VARCHAR

Of course, the values indicated by the user should persist when the target needs to be regenerated. No user wants to enter those data over and over again. To realize this, there are three options:
7.7 Summary

In this chapter we saw that transformations can be viewed as more than just processes. Because transformations are used in situations where we cannot generate hundred percent complete target models, we need to be able to support changes in both the source and target models. This can only be achieved if we know the relationship between the changing models.

A transformation can be viewed as an instance and thus be able to keep information about transformations between specific models. This allows tools to support changes in both models, while still keeping them consistent. Transformation instances can also hold parameters and/or user choices that were given during the transformation process and make sure that these are stored and reapplied in future transformations as well.
Chapter 8. Metamodeling

This chapter explains what metamodeling is, and why it is relevant within the context of MDA. The concept of metamodeling is explained using the four modeling layers of the OMG architecture.
8.1 Introduction to Metamodeling

In Chapter 2 we defined a model as a description of (part of) a system written in a well-defined language. A well-defined language was defined as a language which is suitable for automated interpretation by a computer. The question we will answer in this chapter is: "How do we define such a well-defined language?"

In the past, languages were often defined using a grammar in Backus Naur Form (BNF), which describes what series of tokens is a correct expression in a language. This method is suitable and heavily used for text-based languages, like programming languages. We could use a BNF grammar to define modeling languages. It does fulfill the requirement that it is suitable for automated interpretation. However, BNF restricts us to languages that are purely text based. Because modeling languages do not have to be text based, and often aren't (they can, for example, have a graphical syntax, like UML), we will need a different mechanism for defining languages in the MDA context. This mechanism is called metamodeling.

A model defines what elements can exist in a system. If we define the class Cat in a model, we can have instances of Cat, (like "our neighbor's cat") in the system. A language also defines what elements can exist. It defines the elements that can be used in a model. For example, the UML language defines that we can use the concepts "Class," "State," "package," and so on, in a UML model. Looking at this similarity, we can describe a language by a model: the model of the language describes the elements that can be used in the language.

Every kind of element that a modeler can use in his or her model is defined by the metamodel of the language the modeler uses. In UML you can use classes, attributes, associations, states, actions, and so on, because in the metamodel of UML there are elements that define what is a class, attribute, association, and so on. If the metaclass Interface was not included in the UML metamodel, a modeler could not define an interface in a UML model.

Because a metamodel is also a model, a metamodel itself must be written in a well-defined language. This language is called a metalanguage. So, BNF is a metalanguage. **Figure 8-1** shows this approach. But there are few comments that must be made.

**Figure 8-1. Models, languages, metamodels, and metalanguages**

First, a metalanguage plays a different role than a modeling language in the MDA framework, because it is a
8.2 The Four Modeling Layers of the OMG

To understand the relationships between the various OMG standards that play a role within the MDA framework, you must be aware of the layers of modeling that are defined. The OMG uses a four-layered architecture for its standards. In OMG terminology these layers are called M0, M1, M2, and M3.

8.2.1 Layer M0: The Instances

At the M0 layer there is the running system in which the actual ("real") instances exist. These instances are, for example, the customer named "Joe Nobody" living at "Universal Road 154" in "London, UK" and the customer named "Mark Everyman" living at "South Avenue 665B" in "Denver, USA." Usually there are many customer instances, all with their own data. These instances may exist in various incarnations, such as data in a database, or as an active object running in a computer.

Note that when you are modeling a business and not software, the instances at the M0 layer are the items in the business itself, for example, the actual people, the invoices, and the products. When you are modeling software, the instances are the software representations of the real world items, for example, the computerized version of the invoices or the orders, the product information, and the personnel data.

8.2.2 Layer M1: The Model of the System

The M1 layer contains models, for example, a UML model of a software system. In the M1 model, for instance, the concept Customer is defined, with the properties name, street, and city.

There is a definite relationship between the M0 and M1 layers. The concepts at the M1 layer are all categorizations or classifications of instances at the M0 layer. Likewise, each element at the M0 layer is always an instance of an element at the M1 layer. The customers named "Joe Nobody" and "Mark Everyman" are instances of the M1 element Customer.

M1 elements directly specify what instances in the M0 world look like. The UML Class Customer describes what customer instances at the M0 layer look like. Instances that do not adhere to their specification at the M1 layer are not feasible. For instance, an instance that has a name and city property, but not a street property is not an instance of the Customer class, and must be specified by another class.

Figure 8-3. Relationships between layers M0 and M1
8.3 The Use of Metamodelling in the MDA

The reason that metamodelling is important within the MDA framework is twofold. First, we need a mechanism to define modeling languages, such that they are unambiguously defined. A transformation tool can then read, write, and understand the models. Within MDA we define languages through metamodels.

Secondly, the transformation rules that constitute a transformation definition describe how a model in a source language can be transformed into a model in a target language. These rules use the metamodels of the source and target languages to define the transformations. This is further explained in section 9.1. For now it suffices to say that to be able to understand and make transformation definitions, we must understand the metamodels of the source and target language.

8.3.1 The Extended MDA Framework

Figure 8-8 shows how the MDA framework is completed with the metamodelling level. The lower half of the figure is identical to the basic MDA framework from Figure 2-7. This is what most developers will see eventually. At the upper half we introduce the metalanguage for defining languages.

Typical developers will see the basis framework only, without the additional metalevel. A smaller group of developers, usually the more experienced ones, will need to define languages and transformations between languages. For this group a thorough understanding of the metalevel in the extended MDA framework is essential.
8.4 Summary

In this chapter we saw how languages are defined by metamodels written in a metalanguage. The OMG defines a four-layer metalevel hierarchy, although any number of levels could potentially be used. In principle, a metamodel is just a model, but it is used at a different level in the metamodeling hierarchy.

The MDA framework introduced in Chapter 2 is extended with the appropriate metalevels, thus becoming the extended MDA framework.
Chapter 9. Defining Your Own Transformations

This chapter explains how transformations are defined. Not all users of MDA will define their own transformations; many will be incorporated in the MDA tools. Still, it is good to know how transformations are defined, especially because it enables you to make a more knowledgeable choice of tools.
9.1 Transformations Definitions Revisited

What information do you need to generate a model from another model? You have to relate every element in the source model with one or more elements in the target model. The most direct way of doing this is to relate the metaclass of the element in the source with the metaclass of the element(s) in the target. Because of the instance-type relationship between model element and metaclass, every occurrence of the metaclass in the source will conform to the rules laid down for that metaclass.

Actually, the first transformation rules explained in Chapter 2, The MDA Framework, already used the metalevel, although in an informal way. Here they are again:

- For each class named className in the PIM there is a class named className in the PSM.

- For each public attribute named attributeName : Type of class className in the PIM the following attributes and operations are part of class of class className in the target model.
  - A private attribute with the same name: attributeName : Type
  - A public operation named with the attribute name preceded with "get" and the attribute type as return type: getAttributeName() : Type
  - A public operation named with the attribute name preceded with "set" and with the attribute as parameter and no return value: setAttributeName(att : Type)

The first rule refers to Class, which is an element of both the PIM and the PSM languages. In other words, the metamodel of those modeling languages includes the metaclass Class. The first rule also refers to an attribute className of the metaclass Class.

The second rule refers to a metaclass Attribute in the PIM. From the rule we can also infer that Attribute has the following (meta)attributes: attributeName, Type, and a (meta)attribute that can take the value public. We will name the latter attribute visibility. The metaclasses Class and Attribute live at the M2 level as we have seen in Figure 8-4. The metamodel of the PIM language therefore includes the metaclasses as shown in Figure 9-1.

![Figure 9-1. The Class and Attribute metaclasses in the PIM language](image-url)
9.2 The Transformation Definition Language

In the previous section, we saw transformation rules written in plain English. This is understandable for a human reader, but not for an automated system. In the MDA framework, as shown in Figure 2-7 and Figure 8-8, we need a transformation definition that can be plugged into a transformation tool, which can then automatically execute the transformation. In this section we define a formal notation for writing transformation rules and transformation definitions. Because the rules are not in English but in a formal syntax, they can be interpreted and executed by a transformation tool.

At the time of writing this book, no standard language for writing transformation definitions existed (see section 11.3). Therefore we, the authors, had to define our own transformation definition language. Parts of this language make use of OCL. The purpose of this language is solely to explain the formalization of transformation rules in this book, it is not intended to be a proposal for a standard transformation language. This section explains the transformation definition language used in this book.

9.2.1 Requirements for a Transformation Rule

Any definition of a transformation rule should contain the following information:

- The source language reference.
- The target language reference.
- Optional transformation parameters, for example, constants used in the generation of the target.
- A set of named source language model elements (called S) from the source language metamodel.
- A set of named target language model elements (called T) from the target language metamodel.
- A bidirectional indicator: a boolean that states whether or not a source model may/can be (re)generated from the target.
- The source language condition: an invariant that states the conditions that must hold in the source model for this transformation rule to apply. The invariant may only be expressed on elements from set S.
- The target language condition: an invariant that states the conditions that must hold in the target model for this transformation rule to apply. The invariant may only be expressed on elements from set T.

Furthermore, we prefer to name each transformation rule for referential purposes, although this name could be
9.3 Example Transformation Definitions

This section shows the transformation rules for one simple example. The PIM and PSM languages are fixed. Note that it is easy to change the rules such that the PIM will map to any mainstream object-oriented language. The transformation below can easily be adapted to transform to, for example, Python, Java, or C++.

9.3.1 Public and Private Attributes

The second transformation rule explained in section 9.1 specifies a transformation from public attribute to private attribute and getter and setter operations. The rule can be seen as a relationship class between the metaclasses of both PIM and PSM language, as shown in Figure 9-3. To distinguish between the metaclass Attribute of the PIM and the PSM language, the role names used in the associations are sourceAttribute and targetAttribute, respectively. Remember that actually the two Attribute classes shown in the figure are one and the same, because both the PIM and PSM languages are UML.

Figure 9-3. The transformation rule, viewed as a class

The model specified by the diagram is, however, not yet precise enough to be a complete specification of the transformation rule. We have to augment it with parameters, conditions, and so on. The following specifies the transformation rule completely using the language defined in the previous section. In this case the <~> symbol represents equality, because both source and target language are the same.

Transformation PublicToPrivateAttributes (UML, UML) {
  params
    setterprefix: String = 'set';
    getterprefix: String = 'get';
  source
    sourceAttribute : UML::Attribute;
  target
    targetAttribute : UML::Attribute;
    getter : UML::Operation;
    setter : UML::Operation;
    targetAttribute.visibility = VisibilityKind::private;
  sourceAttribute : UML::Attribute;
  sourceAttribute.visibility = VisibilityKind::public;
  targetAttribute.visibility = VisibilityKind::private;
  sourceAttribute : UML::Attribute;
  sourceAttribute.visibility = VisibilityKind::public;
  targetAttribute.visibility = VisibilityKind::private;
}
9.4 The Complete MDA Framework

Of course, transformation definitions need to be written in a well-defined language to allow transformation tools to read and execute the transformations. A language in which these definitions are written is called a transformation definition language. In such a language you can define transformations based on the metamodels of the languages. Because it works on the metalevel, a transformation definition language is a metalanguage.

The MDA framework is obviously not complete without a transformation definition language. In Figure 9-5 the complete MDA framework is shown. This is identical to the extended MDA framework introduced in section 8.3.1, with the addition of the transformation definition language.

![Figure 9-5. The complete MDA framework]
9.5 Summary

This chapter has shown that transformations can be defined formally using a transformation definition language. To be able to do this, we need to have the metamodels of the source and target languages available.

The transformation definition language, as defined in section 9.2, is not the standard language as it will be defined by the OMG standardization process. At the time of writing, this standardization process is still under way and it seems likely that many different languages will be proposed. It will take some time before this process is finished. The purpose of the transformation definition language in this book is only to explain the formalization of transformation rules.

We have used the transformation definition language to formalize the simple transformations explained in Chapter 2. In the upcoming chapter, the language is used to define some of the example transformations from Chapter 5 and Chapter 6 in a formal manner.
Chapter 10. Rosa's Transformation Definitions

This chapter shows some examples of transformation definitions. All examples are based on the case from Chapter 4, Rosa's Breakfast Service. The transformations are defined using the transformation definition language defined in section 9.2, The Transformation Definition Language. There is no guarantee that the rules presented in this chapter are 100 percent correct. This is because at the time of writing this book no tools for checking the consistency were available. The intention of this chapter is to illustrate the detail level at which transformations have to be defined in a computational independent manner.
10.1 The UML to Relational Mapping

This section describes the transformation definition for transforming PIMs in UML to PSMs dependent on SQL. This is the formal definition of the same transformation that has been used in section 5.1 to transform Rosa's PIM into a relational model. In section 5.1, the transformation was described informally in plain English, which isn't likely to allow tool support. The definition in this section, on the other hand, is formal enough to enable tools to automatically perform the transformation.

As we have seen in section 9.2, we need the metamodels of both the source and target languages for a formal transformation definition. In the transformation definition, we refer to elements from those metamodels. We therefore need the UML metamodel and the SQL metamodel. The UML metamodel we use is a simplification of the UML 1.4 standard (OMG documents formal/2001-09-67), and is depicted in Figure 10-1. Note that the example UML metamodel in the previous chapter in Figure 9-4 is an even more simplified version of the UML metamodel. The metamodel in Figure 10-1 adds some generalizations like Classifier and Feature, and adds an AssociationClass.

![Figure 10-1. Simplified UML metamodel](image)

The SQL metamodel is shown in Figure 10-2. This metamodel is a simplified version of the SQL model from the CWM standard (OMG documents formal/2001-10-01 and formal/2001-10-27) and is consistent with the data definition part of SQL (ISO/IEC 9075:1992). Based on the metamodels for the source and target languages, we can now start to specify the transformation definition. In this case, the complete transformation definition has been broken down into two steps. First, we generate an incomplete relational model where the columns of the foreign keys are not fully specified. Next, we take care of the completion, that is, the generation of the columns of the foreign keys.
10.2 The UML to EJB Mapping

This section describes the transformation definition for transforming PIMs in UML to models of EJB components. The definition is identical to the rules used in section 5.2. As explained before, we need the metamodel for both the source and the target language. The source language is UML, for which we already introduced a metamodel in Figure 10-1. The target language is the coarse grained EJB model described in section 5.2.1. The metamodel for the coarse grained EJB components used is depicted in Figure 10-3. For the sake of simplicity, the transformations and metamodel elements for enumerations and data types without operations (that is, structs) are not defined. These are used for the Style and Address classes in Rosa's model.

![Figure 10-3. Simplified EJB metamodel](image)

Note that we use some additional queries on the UML metamodel called getAllContained() and getOuterMostContainer(). These queries are defined in OCL in section 10.2.1 as additional operations on the UML metamodel.

1. There is a transformation from each UML class to a key class in the EJB model. For each class in the UML model, a separate key class is generated. This is needed in the EJB platform. This rule formalizes rule 1 in section 5.2.2.
10.3 The UML to Web Mapping

This section describes the transformation definition for transforming PIMs in UML to models of Web components. The definition is identical to the rules used in section 5.3.1. The source language is the simplified UML, defined in the metamodel in Figure 10-1. The target language is the Web model described in section 5.3. The metamodel for Web components used is depicted in Figure 10-4. The rules depend on the UML to EJB transformation rules. This is because the generated Web components depend on the EJB components, similar to the fact that EJB Entity Components depend on the Tables of the relational model.

As already explained in section 5.3.1, the UML to Web transformation rules are very similar to the UML to EJB transformation rules. This is also the reason why both transformation definitions use the same query operations defined in section 10.2.1.

1. Each class that is not the part of a composite aggregation is mapped to a Web component. Classes that are composite parts of other classes do not conform to this rule and are transformed by the next rule. This rule formalizes rule 1 in section 5.3.1.

Transformation ClassToWebComponent (UML, Web) {

- Transformation UMLParameterToWebParameter (UML, Web) {
  - UML Parameters are transformed into Web Parameters. This rule formalizes another part of rule 6 in section 5.3.1.

- Transformation UMLOperationToBusinessMethod (UML, Web) {
  - An Operation in UML becomes a Web action in the Web model. This rule formalizes a part of rule 6 in section 5.3.1.

- Transformation UMLAssociationClassEndToWebAssociation (UML, Web) {
  - This rule also defines a transformation from a UML association end to a Web association end. The condition states that this transformation can only be applied when the UML class is part of a composite aggregation. This rule formalizes a part of rule 2 in section 5.3.1.

- Transformation UMLAssociationEndToWebAssociationEnd (UML, Web) {
  - In this case, the type of the Web association end is a Web data class. This rule formalizes a part of rule 3 in section 5.3.1.

- Transformation AssociationClassToDataClass (UML, Web) {
  - This is a straightforward transformation from UML aggregation. This rule formalizes rule 1 in section 5.3.1.

As shown in Figure 10-4, the UML to Web transformation is represented through a simplified metamodel showing the key entities and their relationships. The metamodel includes classes like WebDataSchema, WebClass, WebDataType, WebDataClass, WebComponent, WebAssociationEnd, WebAttribute, and WebAction, among others, with their respective properties and relationships defined.
10.4 Summary

This chapter has shown the formal definitions of the transformations for Rosa's system. The formal definition is quite long and far from trivial; every detail needs to be correct. It is a time-consuming task to write it, and a good tool for checking the consistency is required. However, we must realize that without applying MDA, these kinds of transformations have always been executed by hand and not automated nor well defined. After the formalization is completed, it can be used many times in many projects and the payback can be huge.
Chapter 11. OMG Standards and Additional Technologies

This chapter explains the various OMG standards that are relevant within the context of MDA.
11.1 Introduction

The concepts underpinning MDA can be applied without the use of standards. However, to enable productive use of MDA it is necessary to have a set of related standards on modeling. This allows the industry to develop tools and enables interoperability of MDA solutions and tools.

Some of the most important modeling standards are defined by the OMG. In this chapter we describe the relevant standards with a special focus on the way they fit together. Figure 11-1 gives examples of some elements of the MDA framework that have been defined by the OMG. Figure 11-2 gives some examples of elements that fit the MDA, although they have not been defined by the OMG.
11.2 The MOF

The MOF (OMG document number formal/2002-04-03) is an OMG standard that defines the language to define modeling languages. The MOF resides in the M3 layer, as explained in Figure 8-6. As there is no higher layer, the MOF is defined using the MOF itself. It is the language in which the UML and CWM (see section 11.7) definitions, that is, the UML and CWM metamodels, are written. A simplified version of the MOF model is shown in Figure 11-3.

Figure 11-3. The MOF metamodel (simplified)

11.2.1 yMOF Tools

The MOF is not only used to define modeling languages, but also to enable the building of tools for defining modeling languages. The MOF therefore provides some additional functionality.

The MOF Repository Interface

The MOF definition includes a specification of the interface for a MOF repository. This interface allows us to get information about M1 models from a MOF-based repository. This interface is defined using CORBA-IDL and is therefore usable in many environments. Especially for Java, there is a native interface providing the same functionality. This is called the Java Metadata Interface (JMI) and it is described in Sun Microsystems, Java Metadata Interface Specification, Version 1.0 (2002). From Java applications, this interface is easier to use than the CORBA-to-Java mapping.

Model Interchange

The MOF is also used to define a stream- or file-based interchange format for M1 models. Whenever a modeling language is defined using a metamodel described in the MOF, the MOF defines a standard way to generate an interchange format for models in that language. This interchange format is based on XML, and it is called XMI (XML Metadata Interchange).
11.3 Query, Views, and Transformations

Currently a new standard is under development, called the Query, Views, and Transformations Standard (QVT). This standard addresses the way transformations are achieved between models whose languages are defined using the MOF. It will become part of the MOF, and will have the following parts:

- A language for creating views on model
- A language for querying model
- A language for writing transformation definitions

The latter part is the one that is the most relevant to MDA. It will in time be a standardized replacement of the transformation definition language we have defined in section 9.2.
11.4 UML

UML (OMG documents formal/2001-09-67) is the standard modeling language at the M2 level, defined using the MOF. A vast majority of the models that are being developed are UML models. To be able to use MDA in software development, it is therefore necessary to understand UML.

11.4.1 The UML Metamodel

UML is the most widely used modeling language defined in the MOF. Figure 11-4 shows an extension of the UML metamodel used in Chapter 9 and Chapter 10. It is still a small and simplified part of the standard metamodel, describing only part of the language. The UML metamodel describes exactly how a UML model is structured. From the metamodel, we can deduce the following:

- At the top we see that everything in UML is a ModelElement and has a "name."

- The abstract metaclass Classifier is a generalization of Class, Interface, and DataType. They have much in common, but some specifics as well.

- A Class may implement an Interface, but a DataType or an Interface may not.

- A Feature (Attribute, AssociationEnd or Operation) is part of exactly one Classifier; it can never be part of multiple Classifiers.

- An association has a minimum of two association-ends, but may have more.

- An operation may have zero or more parameters.

Figure 11-4. UML metamodel (simplified)
11.5 OCL

OCL (OMG document ad/02-05-09, 2002) is an expression language in which you can write expressions over models, for instance, derivation rules for attributes, the body of query operations, invariants, and pre- and postconditions. OCL can be used for both MOF and UML models. Using OCL extends the expressive power of UML/MOF, and allows the modeler to create more precise and more extensive models.

Any OCL expression evaluates to a value. OCL is a specification language and an OCL expression always describes what the value is, but never dictates how the expression should be calculated. Of course, OCL expressions can be translated to programming languages (for example, Java) to specify how the expression is executed. OCL expressions are "pure" expressions in the sense that they never have side effects.

11.5.1 Using OCL with UML

Traditionally, OCL has been used in UML to specify constraints. The most common types of constraints are invariants, preconditions, and postconditions. More recently the use of OCL for other types of expressions has become popular. This includes the use of OCL for the following purposes:

- Specifying initial attribute values
- Specifying the derivation rules for attributes or associations
- Specifying the body of query operations
- Specifying the targets for messages being sent
- Specifying guard conditions in statecharts
- Specifying end-user queries on a UML model

A UML model of a system becomes more precise and more complete by applying OCL. In the context of MDA, this means that the source model of a transformation becomes much richer, which makes it possible to generate a much more complete target model. The ability to specify a precise and complete source model allows an MDA transformation to generate more of the PSM or code. The value of the MDA approach to the developers is enhanced considerably. OCL is used for specifying expressions in UML models by any developer that uses UML. As such, all developers can benefit from the added precision and expressiveness.

11.5.2 Using OCL with the MOF
11.6 The UML Action Semantics

The UML Action Semantics, as described in OMG document ptc/2002-01-09, is an extension of UML providing the metamodel for a so-called action language. The purpose of this action language is to provide a foundation for dynamic semantics of UML. In principle, the action semantics can be used to write directly executable UML models (see also section 3.2.2) as is shown in Mellor and Balcer 2002 and in Kennedy Carter White Paper, CTN 80, v2.2.

There is no concrete syntax for the action semantics given in the standard; therefore, you cannot write any statement in the language in a standardized way. There are several vendor-specific languages that claim to map to the abstract syntax of the action semantics. The Action Semantics attaches all dynamics of a system to statemachines and is mainly used in the domain of embedded software.

The concepts in the Action Semantics are defined at a very low level. For instance, the concepts input and output pin are defined, which represent the fact that an action (function/operation) may have input and/or output parameters. In many respects the action semantics looks more like UML assembler than a language that is useful for a UML modeler. As such, it might be useful as a foundation, but leaves a definite need for a language at a higher level of abstraction.
11.7 CWM

CWM, as described in OMG documents formal/2001-10-01 and formal/2001-10-270, is a modeling language that is specifically meant to model data warehousing applications. The metamodel has a lot in common with the UML metamodel, but it has a number of special metaclasses, for example, for modeling relational databases. The developers of CWM have removed everything from UML that was not needed for their purpose, and added the specific data warehousing details. The behavioral parts of the UML metamodel (like statemachines or collaborations) are not in CWM.

Because data warehousing is a technology that combines information from many different sources, the CWM metamodel includes simple metamodels for a number of things:

- Relational databases
- Records or structures
- OLAP
- XML
- Transformations (not as in MDA!)
- Visualization of information
- Data Mining
- Multidimensional databases
- Business metadata
- Warehouse processes
11.8 UML Profiles

The profile concept is a specialization mechanism defined as part of UML. A profile defines a specific way of using UML. For example, the CORBA Profile for UML defines a specific way of using UML to model CORBA interfaces, and the Java Profile for UML defines a way to model Java source code in UML.

A profile is defined by a set of stereotypes, a set of related constraints, and a set of tagged values.

A stereotype definition has a name and is attached to elements in the UML metamodel. For example, the stereotype <<JavaClass>> is defined for the UML metaclass Class (see Figure 11-4) in the EJB Profile (Sun Microsystems, Enterprise JavaBeans Specification, Version 2.1, 2002). In a UML model one can apply this stereotype to each class in a model.

A constraint can be attached to a stereotype definition. This constraint is expressed in OCL in terms of the UML metamodel, and describes the restrictions on instances of the model elements to which the stereotype is applied. For example, for each class in a UML model labeled with the stereotype <<JavaClass>>, the following constraint should hold: "A Java class may have at most one superclass."

A tagged value is an additional meta-attribute that is attached to a UML metaclass in the UML metamodel. A tagged value has a name and a type and is attached to a specific stereotype. It can be given a value in a model, but only for elements that have the corresponding stereotype.

The effect of a profile is that it defines a specialized variant of UML, for a specific purpose. An alternative is to define a new metamodel instead of using a profile. This would result in separate metamodels for Java, CORBA, and so on.

11.8.1 The Role of Profiles in MDA

A profile defines a specialized metamodel, which is by definition a subset of the UML metamodel. In fact, a profile defines a new language by reusing the UML metamodel.

Most profiles that are currently used define languages specific for certain platforms, like CORBA, Java, or C++ profiles. A model with such a profile and its applied stereotypes can only be used as a PSM.
11.9 Summary

There are many, many standards that may have a relationship with MDA. The most important one is the MOF standard that allows us to define metamodels. In addition to the MOF, we need a transformation definition language to describe transformations between models. In the transformation definition language we may also use OCL to specify queries and conditions. Although there is no transformation language standardized yet, it is expected to have similar features to the one defined in section 9.2.

All the other languages and standards that we have seen only play the role of source and/or target language within MDA. Furthermore, whenever we define a new language through a MOF metamodel, it can be used within an MDA environment. The UML language and all of its derivatives defined by profiles are the modeling languages that will be used often, but are not essential to the MDA approach.
Chapter 12. The MDA Promise

This chapter describes future developments that are triggered by the MDA. It describes what is not yet realized, but what is achievable if we follow in the direction the MDA is going. Some of these developments may take place within a couple of years, others might take longer.
12.1 The MDA Paradigm Shift

In Chapter 1 we explained that the MDA development life cycle is not very different from the traditional life cycle. What is crucial is that the MDA, when fully implemented, will establish a shift of focus within the life cycle. To explain this vision on the MDA we will take a look at the history of programming.

12.1.1 A Historic Perspective

In short, we can characterize the history of programming along the time line given in Table 12-1. Each period is characterized by the dominant type of programming language. The programming languages used in each period are built on the assets of the previous period.

First there was raw machine code, which automated the wiring of computers. Next, symbolic representations of this raw machine language, called assembly languages, were made, and assembly language programs were converted to machine code by an automated tool called an assembler. Following, there was the time of the procedural programming languages. These were built on top of assembly languages. They were designed to be easier than an assembly language for a human to understand, including things like named variables, procedures, subroutines, and so on. Compilers or interpreters were used to convert the programs into assembly or raw machine code.

Object-oriented programming languages were built on the assets of procedural languages. They extend the procedural languages with much more powerful structuring mechanisms. Often compilers for object-oriented programming languages use a two-step method. First the program is translated into a low-level third-generation language, next this result is compiled into machine code.

<table>
<thead>
<tr>
<th>Years</th>
<th>Programming was done by...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950</td>
<td>Raw machine code</td>
</tr>
<tr>
<td>1950–1965</td>
<td>Assembly languages</td>
</tr>
<tr>
<td>1965–1985</td>
<td>Procedural programming languages</td>
</tr>
<tr>
<td>1985–today</td>
<td>Object-oriented programming languages</td>
</tr>
<tr>
<td>today–...</td>
<td>What is next?</td>
</tr>
</tbody>
</table>

12.1.2 A Shift of Focus
12.2 The Development Process

What are the consequences of applying the MDA on the software development process? Comparing the MDA process to the traditional process, much will remain the same. Requirements still need to be captured and systems still need to be tested and deployed. What will change are the activities of analysis, low-level design, and coding.

During analysis a PIM will need to be developed. There will probably be a special group of people who will perform this task. They will be aware of the functionality that needs to be implemented, and they will be driven by the needs of the business and by the business model.

Most likely there will be a different group of people responsible for the transformation of the PIM to one or more PSMs. They will have knowledge of different platforms, different system architectures, and of the transformation definitions that are available for the transformation tools they are using. They will decide with what parameter values the transformations are to be applied. They will choose between various target platforms and target system architectures. Does the system require a three-layer architecture, or is a simple fat client-server architecture sufficient? Do we use J2EE or the .net as target platform? These are the kind of questions the PSM creator will need to answer. It is the responsibility of the PSM creator to be aware of and act upon quality of service aspects.

To find the answer to these questions, the PSM creator will need some information from the PIM analyst other than the PIM itself. He will need to know about the non-functional business requirements. For instance, when a PIM analyst tells the PSM creator that the system will be used intensively by over five thousand persons, the PSM creator will wisely choose another target architecture, and use another transformation definition than if the system were to be used by a group of five.

Another task of the PSM creator is to respond to changes in both the PIM and the transformation definitions. The PIM and the transformation definitions may change independently of each other. When the business requirements change, only the PIM will be affected. When the target platform changes, for instance because a new version is installed, only the transformation definition will have to be renewed. The PSM creator must respond to these changes because any change must be reflected in the generated PSM. Meanwhile, the previous version of the system that has already been deployed, may contain lots of data that has to migrate to the new version of the system. In the future this task of the PSM creator will need to be supported by tools. These tools may or may not be integrated in the transformation tool.

Because the PSM creator will need transformation definitions, buying or writing transformation definitions will be a task new to the MDA development process. In our view, there will be a third group of people who will write transformation definitions. They will partly be employed by the companies that build software systems, but a large part of this group will be employed by transformation tool vendors. Their tools will only have customer value when transformation definitions are available. Figure 12-2 shows the different participants in the MDA process, the tools they are using, and the artifacts they are producing.

Figure 12-2. Participants, tools, and artifacts in the MDA process
12.3 The Tools

What are the consequences of applying the MDA on the software development tools? Again, looking at the difference between the traditional development process and the MDA development process, we find three groups of people who will need new or better tools, and one group who will not need any tools, because it will not be involved in the MDA process.

The last group is the group of code writers. When the MDA approach has matured, this group will not be needed anymore. Current code writing tools, like IDEs specific to a certain programming language, will cease to be relevant in the development process. The trend is already noticeable. Currently, many IDE vendors are progressing their environments to integrate a modeling tool as well. Figure 12-3 shows how in the future the functionality supporting activities concerning code will be less important. Instead, new functionality will be demanded on the high end of the software factory pipeline.

![Figure 12-3. Functionality in a future MDA development environment](image)

A new kind of IDE will be demanded by the transformation definition developers. They will need specialized environments to create, edit, and test their transformation definitions. Because the aim is to reuse transformation definitions, the group of transformation definition developers will not be very large. Therefore, it is likely that there will not be a large choice of transformation definition IDEs.

The group of PSM creators will work with transformation tools. The desired functionality for these tools will include the following:

- The ability to choose between different implementation platforms
- The flexibility to switch within one platform between different implementation strategies, application architectures, and coding patterns
- Openness to plug-in multiple modeling languages, transformation definitions

The group of PIM analysts are the group that is served already. There is a large number of modeling tools available, most of which are able to read and write the standard interchange format for UML models. Still, the PIM analysts will need better modeling tools; tools that provide more support on checking the model and on the integration of the model parts. For example, the flow of information between the various UML diagrams should be supported. But most of all the PIM analyst needs better modeling languages.
12.4 The Modeling Languages

What are the consequences of applying the MDA on the modeling languages? To be able to write PIMs that completely specify the system to be generated, including both static and dynamic aspects, we need a different set of modeling languages.

Today's modeling languages provide us with the means to specify the structural part of the functionality in a PIM. For the dynamic part, they depend on ordinary programming languages to fill in the gaps in the model. The action language used in Executable UML (Mellor and Balcer 2002) tries to fill this gap, but as explained in section 3.2.2, the available concepts are at the same abstraction level as the current procedural and object oriented languages. Unless the action language gets to a higher level of abstraction, it will not be able to support the MDA process fully.

A modeling language suited for MDA should offer the following:

- Expressive enough to specify systems completely. This includes both static and dynamic aspects of a system. There should be no need for the developers to fall back to ordinary programming languages.

- A general applicable, non application-specific, language. Application specific languages like 4GLs never really set off. Most programming is still done using generally applicable programming languages.

- Abstract often-used patterns of lower-level constructs into single higher-level constructs.

- Suitable for n-tier application development, including three-tier, two-tier, and single-tier applications. The actual number of tiers should be of no consequence in the model, but should be adjusted in the settings of the transformation tool.

- Suitable for distributed applications. Transformation tools should take care of building the bridges between the various nodes.

- Seamlessness between the model and its implementation.

- Support for managing large models, for instance by supporting an aspect-oriented manner of modeling (Kiczales 1997).

Note that when a modeling language is created that allows full specification of a system, including both static and dynamic aspects, it becomes more than a modeling language. In fact, the modeling languages of tomorrow will have the same status as the high-level programming languages of today.
12.5 Summary

In this chapter we have taken a look into the crystal ball and predicted what the future of software development might look like when the MDA is applied on a large scale.

Although the MDA is still in its infancy, it already shows the potential of changing software development radically. Nowadays the focus of the software development process is on writing code. In the future the focus will shift to writing the PSM, and from there to writing the PIM. People will forget the fact that the PSM needs to be transformed into code, because generating the code will be automated. This is such a major change in the development process that it can be called a paradigm shift.

The shift of focus from code to models will have consequences on the software development process, the languages used to write models, and the software development tools. In the software development process, three participants can be recognized:

- The PSM creator, who is responsible for transforming a PIM to one or more PSMs. The PSM creator will be the one that uses transformation tools. He will choose the right transformation for the job and parameterize it.

- The transformation definition developer, who is responsible for creating and maintaining transformation definitions. The transformation definition developer will need a specialized environment to create, edit, and test his transformation definitions.

- The PIM analyst, who is responsible for creating and maintaining PIMs. The PIM analyst will need better modeling tools. But most of all the PIM analyst needs better modeling languages.

New modeling languages suited for MDA will be defined. They will allow full specification of a system, including both static and dynamic aspects. Such a language will have the same status as the programming languages of today.
Appendix A. Glossary
Glossary

Abstraction

A broad and general term indicating (1) a less detailed model that conforms to (defines a subset of the properties of) another model, and (2) the process through which a less detailed but conforming model is made, that is, the process of removing details that are not relevant to the purpose of the model.

Abstraction Level

The inverse of the (relative) amount of details that are in a model.

See also [High] See also [Low Abstraction Level]

Communication Bridge

An implementation or model of communication between two parts of a system. Mostly used in a context where the parts of the system are realized using different technologies.

Coarse Grained Component

Components that have infrequent interaction with a relatively high amount of data in each interaction.

Diagram

A visible graphical rendering of (a part of) a model.

Fine Grained Component

Components that have frequent communication with a low amount of data in each interaction.

High Abstraction Level

A (relatively) low amount of details.

Language

In this book we use the word language as a synonym for Well-Defined Language.

Low Abstraction Level

A (relatively) high amount of details.

Mapping

The constraining relationship between the structure of the source and target language in a transformation definition.
Appendix B. The Code for Rosa's System

Section B.1. The SQL Code for Rosa's System

Section B.2. The EJB Code for Rosa's System

Section B.3. The JSP Code for Rosa's System
B.1 The SQL Code for Rosa's System

The following code snippet is the SQL script to create the tables for Rosa's Breakfast Service:

```sql
CREATE TABLE Comestible (  
  comestibleID INTEGER NOT NULL,  
  name VARCHAR (40) NULL,  
  price REAL NULL,  
  minimalQuantity INTEGER NULL,  
  transportForm VARCHAR (40) NULL,  
  PRIMARY KEY (comestibleID)
);

CREATE TABLE StandardBreakfast (  
  standardBreakfastID INTEGER NOT NULL,  
  name VARCHAR (40) NULL,  
  price REAL NULL,  
  style INTEGER NULL,  
  PRIMARY KEY (standardBreakfastID)
);

CREATE TABLE Part (  
  standardBreakfastID INTEGER NOT NULL,  
  comestibleID INTEGER NOT NULL,  
  quantity INTEGER NULL,  
  PRIMARY KEY (standardBreakfastID, comestibleID)
);

CREATE TABLE Customer (  
  customerID INTEGER NOT NULL,  
  accountNumber DECIMAL NULL,  
  addressStreet VARCHAR (40) NULL,  
  addressCity VARCHAR (40) NULL,  
  addressStreetNumber VARCHAR (40) NULL,  
  addressPostalCode VARCHAR (40) NULL,  
  addressTelephoneNumber VARCHAR (40) NULL,  
  PRIMARY KEY (customerID)
);

CREATE TABLE BreakfastOrder (  
  breakfastOrderID INTEGER NOT NULL,  
  customerId INTEGER NOT NULL,  
  orderDate DATE NULL,  
  deliveryAddressStreet VARCHAR (40) NULL,  
  deliveryAddressCity VARCHAR (40) NULL,  
  deliveryAddressStreetNumber VARCHAR (40) NULL,  
  deliveryAddressPostalCode VARCHAR (40) NULL,  
  deliveryAddressTelephoneNumber VARCHAR (40) NULL,  
  deliveryDate DATE NULL,  
  deliveryTime TIME NULL,  
  discount REAL NULL,  
  PRIMARY KEY (breakfastOrderID)
);

CREATE TABLE Breakfast (  
  breakfastID INTEGER NOT NULL,  
  breakfastOrderID INTEGER NOT NULL,  
  standardBreakfastID INTEGER NOT NULL,  
  number INTEGER NULL,  
  PRIMARY KEY (breakfastID)
);

CREATE TABLE Change (  
  breakfastID INTEGER NOT NULL,  
  comestibleID INTEGER NOT NULL,  
  quantity INTEGER NULL,  
  PRIMARY KEY (breakfastId, comestibleID)
);
```
B.2 The EJB Code for Rosa's System

The following code snippet describes the remote interface of the entity bean called BreakfastOrder:

```java
import java.rmi.*;
import javax.naming.*;
import javax.ejb.*;
import breakfast.ejb.breakfastorder.*;

public interface BreakfastOrder extends EJBObject {
    public BreakfastOrderDataObject getBreakfastOrder()
            throws RemoteException;

    public void setBreakfastOrder(BreakfastOrderDataObject update)
            throws NamingException,
                    FinderException,
                    CreateException,
                    RemoteException;

    public float calculatePrice()
            throws RemoteException;
}
```

The following code snippet shows the implementation of the data class for BreakfastOrder:

```java
import breakfast.ejb.*;
import java.util.*;

public class BreakfastOrderDataObject
        extends DataObject implements java.io.Serializable {

    private BreakfastOrderKey key;

    /**
     * Creates a new BreakfastOrderDataObject.
     * @param key initialize all fields necessary to uniquely identify
     * this object
     */
    public BreakfastOrderDataObject(BreakfastOrderKey key) {
        this.key = key;
        this.deliveryAddress = new Address();
    }

    public int getBreakfastOrderID() {
        return key.getBreakfastOrderID();
    }

    // References to associated single classes:
    private CustomerKey customercustomerKey;

    /**
     * Sets the foreign key CustomerKey of the singular referenced
     * Customer
     * @param CustomerKey
     */
    public void setCustomerCustomer(CustomerKey s) {
        this.customercustomerKey = s;
    }

    /**
     * Returns the foreign key CustomerKey of the singular referenced Customer
     */
    public CustomerKey getCustomerCustomer() {
        return customercustomerKey;
    }

    // References to by-value-treated collections of objects:
    private breakfast.ejb.breakfastorder.BreakfastDataObjectCollection
            breakfastbreakfast;

    public void addBreakfast(breakfast.ejb.breakfastorder.BreakfastDataObject added) {
        this.breakfastbreakfast.add(added);
    }

    public void removeBreakfast(breakfast.ejb.breakfastorder.BreakfastDataObject removed) {
        this.breakfastbreakfast.remove(removed);
    }

    /**
     * Returns the internal BreakfastDataObjectCollection of the multiple
     * contained Breakfast.
     * @return breakfast.ejb.breakfastorder.BreakfastDataObjectCollection
     */
    public breakfast.ejb.breakfastorder.BreakfastDataObjectCollection
            getBreakfast() {
        if (breakfast == null) {
            breakfast =
            new breakfast.ejb.breakfastorder.BreakfastDataObjectListImpl();
        }
        return breakfast;
    }

    // No associated collections of objects by reference

    // Attributes:
    private java.util.Date orderDate;

    /**
     * Returns attribute orderDate.
     * @return java.util.Date
     */
    public java.util.Date getOrderDate() {
        return orderDate;
    }

    public void setOrderDate(java.util.Date value) {
        this.orderDate = value;
    }

    private Address deliveryAddress;

    /**
     * Returns a copy of attribute deliveryAddress.
     * Since deliveryAddress is a StructType, we do not want that the
     * object returned by getDeliveryAddress() can implicitly change the
     * state of the owner BreakfastOrder.
     * Therefore a copy is returned.
     * @return Address
     */
    public Address getDeliveryAddress() {
        return deliveryAddress != null ? deliveryAddress.deepCopy() : null;
    }

    public void setDeliveryAddress(Address value) {
        this.deliveryAddress = value;
    }

    private java.util.Date deliveryDate;

    /**
     * Returns attribute deliveryDate.
     * @return java.util.Date
     */
    public java.util.Date getDeliveryDate() {
        return deliveryDate;
    }

    public void setDeliveryDate(java.util.Date value) {
        this.deliveryDate = value;
    }

    private java.util.Date deliveryTime;

    /**
     * Returns attribute deliveryTime.
     * @return java.util.Date
     */
    public java.util.Date getDeliveryTime() {
        return deliveryTime;
    }

    public void setDeliveryTime(java.util.Date value) {
        this.deliveryTime = value;
    }

    private float discount;

    /**
     * Returns attribute discount.
     * @return float
     */
    public float getDiscount() {
        return discount;
    }

    public void setDiscount(float value) {
        this.discount = value;
    }
```

This Page illustrates how a Retrieve is executed on the EJB tier.

It is written as pure Jsp, with a connection to basic OptimalJ data-structures.

// let Remote View Manager retrieve data
CustomerCollection = CustomerDom.retrieve();

// iterate collection and write data elements
while (it.hasNext()) {
    Customer = (CustomerDataObject) it.next();
    request.setAttribute("Customer", Customer);
%
    <tr align="left"><font size="3"><jsp:getProperty name="Customer" property="id"/></font></tr>
    <tr align="left"><font size="3"><jsp:getProperty name="Customer" property="address"/></font></tr>
    <tr align="left"><font size="3"><jsp:getProperty name="Customer" property="accountNumber"/></font></tr>
}
Bibliography


abstraction level 2nd 3rd 4th 5th 6th 7th 8th 9th
Action Semantics [See AS]
Agile Software Development 2nd
analysis 2nd
AS 2nd 3rd
assembler languages
association 2nd
multiplicity 2nd
navigability
attribute
Backus Naur Form  [See BNF]
bidirectionality  2nd
BNF
business
   metadata
   method
   model
   rules
CASE tool
CIM
coarse grained component model

code
  bridge
generation
model
coding

Common Warehouse Metamodel  [See CWM]

communication bridge, PSM bridge

composite aggregation

Computational Independent Model  [See CIM]

conceptualization
constraint

container managed persistency

controlling transformations

CORBA
  IDL

corba:profile

CWM
data mining
data object manager
deployment
descriptor
design
decision
documentation
2nd
dynamic model
dynamic view
Enterprise Application Integration: [See EAI]

EDOC:Enterprise Distributed Object Computing: [See EDOC]

Entity-Relationship model
Executable UML 2nd
eXtreme Programming [See XP]
fine grained component model  2nd
four-layered architecture
future developments
get operation  2nd  3rd
IDE  2nd  
incremental consistency  
ncremental development process  
information hiding  
instance-of relationship  
Interactive Development Environment  [See IDE]  
interoperability  2nd  3rd  
iterative development process
J2EE
Java
  collection classes
Java Database Connectivity  [See JDBC]
Java Metadata Interface    [See JMI]
Java:Profile
  profile
JavaServer Pages     [See JSP]
JDBC
JMI
JSP  2nd
language
definition of 2nd
well-defined 2nd 3rd
layers of modeling
local home interface
local interface
M0 layer
M1 layer 2nd
M2 layer 2nd 3rd 4th
M3 layer 2nd
maintenance 2nd 3rd
MDA
devolution life cycle 2nd 3rd
development process 2nd 3rd
tools 2nd 3rd 4th 5th 6th
MDA framework 2nd 3rd 4th 5th 6th
Meta Object Facility [See MOF]
metaclass 2nd 3rd
metalaugure
metametamodel
metamodel 2nd 3rd
metamodeling
model 2nd 3rd
definition
interchange mechanisms
purpose
relationships between models
repository
modeling languages
modeling of behavior
MOF 2nd 3rd 4th 5th
repository
multidimensional databases
Object Constraint Language  [See OCL]
Object Management Group  [See OMG]
Object-oriented programming languages
OCL  2nd  3rd  4th  5th  6th
OK to remove specific reference to QVT and just refer to section 3.1.2?
OMG  2nd  3rd
OptimalJ
outermost composition
parameterization of transformation
parameters to transformation
persistence management
PIM 2nd 3rd 4th analyst
PIM-to-PSM transformation 2nd
Platform Independent Model [See PIM]
Platform Specific Model [See PSM]
portability 2nd 3rd
preservation of characteristics
primitive data types
productivity 2nd
productivity gain
profile
UML Profile 2nd
used as PSM
prototype
PSM 2nd 3rd
bridge
creator
PSM-to-code transformation 2nd
Query, Views, and Transformations Standard  [See QVT]
QVT  2nd
Rational Unified Process  [See RUP]
raw machine code
relational databases
relationships between models
remote home interface
remote interface
remote method invocation
requirements
RUP
scripting language
second normal form
set operation 2nd 3rd
software development process 2nd
software model
source language
source model 2nd
tag
SQL 2nd
metamodel
state machine
stereotype
structural model
structural view
system
tagged value
target
  language
  model  2nd
  platform
testing
tree-tier architecture
traceability
transformation  2nd  3rd
  bidirectionality  2nd
  conditions on
  example of rule  2nd
  execution
  incremental consistency  2nd
  parameter  2nd
  parameters  2nd
  persistent
  public to private attributes
  rule  2nd  3rd  4th  5th  6th
  rule as object
  source language  2nd  3rd
  target language  2nd  3rd
  tool  2nd  3rd  4th  5th  6th  7th  8th  9th
  traceability  2nd
  tunability
  tuning
transformation definition  2nd  3rd  4th  5th  6th  7th  8th  9th  10th
  example  2nd
  from other transformation definitions
transformation definition language  2nd  3rd
  tuning of transformation
  transformation:tuning
UML metamodel
  2nd  3rd  4th  5th  6th  7th  8th
UML profile
  profile
UML Profile
  profile
Unified Modeling Language  [See UML]
warehouse operation
warehouse processes
waterfall process
Web components metamodel
well-defined language 2nd definition
XML Metadata Interchange  [See XMI]

XP  2nd