Model-Driven Software Engineering in Practice

Second Edition

Synthesis Lectures on Software Engineering

Editor
Luciano Baresi, ***Politecnico di Milano***

The Synthesis Lectures on Software Engineering series publishes short books (75-125 pages) on conceiving, specifying, architecting, designing, implementing, managing, measuring, analyzing, validating, and verifying complex software systems. The goal is to provide both focused monographs on the different phases of the software process and detailed presentations of frontier topics. Premier software engineering conferences, such as ICSE, ESEC/FSE, and ASE will help shape the purview of the series and make it evolve.

Model-Driven Software Engineering in Practice: Second Edition
Marco Brambilla, Jordi Cabot, and Manuel Wimmer
2017

Testing iOS Apps with HadoopUnit: Rapid Distributed GUI Testing
Scott Tilley and Krissada Dechokul
2014

Hard Problems in Software Testing: Solutions Using Testing as a Service (TaaS)
Scott Tilley and Brianna Floss
2014

Model-Driven Software Engineering in Practice
Marco Brambilla, Jordi Cabot, and Manuel Wimmer
2012

Model-Driven Software Engineering in Practice: Second Edition

Marco Brambilla, Jordi Cabot, and Manuel Wimmer

www.morganclaypool.com

ISBN: 9781627057080 paperback
ISBN: 9781627059886 ebook
ISBN: 9781627056953 epub

DOI 10.2200/S00751ED2V01Y201701SWE004

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON SOFTWARE ENGINEERING

Lecture #4
Series Editor: Luciano Baresi, *Politecnico di Milano*
Series ISSN
Print 2328-3319 Electronic 2328-3327

Model-Driven Software Engineering in Practice

Second Edition

Marco Brambilla
Politecnico di Milano, Italy

Jordi Cabot
ICREA and Open University of Catalonia (UOC), Spain

Manuel Wimmer
TU Wien, Austria

SYNTHESIS LECTURES ON SOFTWARE ENGINEERING #4

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This book discusses how model-based approaches can improve the daily practice of software professionals. This is known as Model-Driven Software Engineering (MDSE) or, simply, Model-Driven Engineering (MDE).

MDSE practices have proved to increase efficiency and effectiveness in software development, as demonstrated by various quantitative and qualitative studies. MDSE adoption in the software industry is foreseen to grow exponentially in the near future, e.g., due to the convergence of software development and business analysis.

The aim of this book is to provide you with an agile and flexible tool to introduce you to the MDSE world, thus allowing you to quickly understand its basic principles and techniques and to choose the right set of MDSE instruments for your needs so that you can start to benefit from MDSE right away.

The book is organized into two main parts.

- The first part discusses the *foundations of MDSE* in terms of basic concepts (i.e., models and transformations), driving principles, application scenarios, and current standards, like the well-known MDA initiative proposed by OMG (Object Management Group) as well as the practices on how to integrate MDSE in existing development processes.
- The second part deals with the *technical aspects of MDSE*, spanning from the basics on when and how to build a domain-specific modeling language, to the description of Model-to-Text and Model-to-Model transformations, and the tools that support the management of MDSE projects.

The second edition of the book features:

- a set of completely new topics, including: full example of the creation of a new modeling language (IFML), discussion of modeling issues and approaches in specific domains, like business process modeling, user interaction modeling, and enterprise architecture
- complete revision of examples, figures, and text, for improving readability, understandability, and coherence
- better formulation of definitions, dependencies between concepts and ideas
- addition of a complete index of book content

KEYWORDS

modeling, software engineering, UML, domain-specific language, model-driven engineering, code generation, reverse engineering, model transformation, MDD, MDA, MDE, MDSE, OMG, DSL, EMF, Eclipse

Contents

Foreword

Technology takes forever to transition from academia to industry. At least it seems like forever. I had the honor to work with some of the original Multics operating system development team in the 1970s (some of them had been at it since the early 1960s). It seems almost comical to point out that Honeywell only ever sold a few dozen Multics mainframes, but they were advanced, really advanced—many of Multics' innovations (segmented memory, hardware security and privacy, multi-level security, etc.) took literally decades to find their way into other commercial products. I have a very distinct memory of looking at the original Intel 386 chip, impressed that the engineers had finally put Multics-style ring security right in the hardware, and less impressed when I discovered that they had done it exactly backward, with highly secure users unable to access low-security areas, but low-security users able to access the kernel. Technology transfer is a difficult and delicate task!

When I had the opportunity to help introduce a new technology and manage hype around that technology, I took it. At loose ends in 1989, I agreed to join the founding team of the Object Management Group (OMG), to help define commercial uptake for Object Technology (called object-oriented systems in the academic world, at least since Simula in 1967), and equally to help control the hype around the Object Technology marketplace. Having participated in the Artificial Intelligence (AI, or expert systems) world in the 1980s, I really didn't want to see another market meltdown as we'd experienced in AI: from the cover of *Time* magazine to a dead market in only five years!

That worked. OMG named, and helped define, the middleware marketplace that flourished in the 1990s, and continues today. Middleware ranges from: TCP socket-based, hand-defined protocols (generally an awful idea); to object-based, request-broker style stacks with automatically defined protocols from interface specifications (like OMG's own CORBA); to similarly automatically-defined, but publish-and-subscribe based protocols (like OMG's own DDS); to semantic integrate middleware with high-end built-in inference engines; to commercial everything-but-the-kitchen-sink "enterprise service bus" collections of request brokers, publish-and-subscribe, expert-system based, automatic-routing, voice-and-audio streaming lollapaloozas. Middleware abounds, and although there's still innovation, it's a very mature marketplace.

By the late 1990s, it was clear that the rapid rise of standardization focused on vertical markets (like healthcare IT, telecommunications, manufacturing, and financial services, OMG's initial range of so-called "domain" standards) would need something stronger than interface definition languages; to be more useful, standards in vertical markets (and arguably, all standards) should be defined using high-level, precise but abstract "modeling" languages. This class of languages should be much closer to user requirements, more readable by non-technical people, more

focused on capturing process and semantics; in general, they should be more expressive. The natural choice for OMG and its members was of course OMG's own Unified Modeling Language (UML), standardized in 1997 as an experimental use of OMG's standards process that had heretofore focused on middleware. Even better, the UML standardization effort had produced a little-known but critical modeling language called the Meta-Object Facility (MOF) for defining modeling languages. This core of MOF plus extensible, profileable UML would easily be the foundation for a revolution in software development—and beyond.

As the millennium approached, OMG's senior staff met to consider how we could nudge the OMG membership in this valuable new direction. We came up with a name (Model-Driven Architecture, or MDA); a picture (which hasn't been universally adopted, but still helped make the transition); and a well-received white paper that explained why MDA would be the next logical step in the evolution of software engineering (and by extension, how it matches modeling in other engineering disciplines, though generally with other names, like "blueprints.") OMG's senior staff then spent a few months pitching this idea to our leading members, to a very mixed review. Some had been thinking this way for years and welcomed the approach; while some thought that it would be seen as an abandonment of our traditional middleware space (which by the way, we have never abandoned; the latest OMG middleware standards are weeks old at this writing and many more are to come). The CEO of one of our key member companies found the concept laughable, and in a memorable phrase, asked "Where's the sparkle?"

I truly believe, however, that organizations which resist change are the least stable. OMG therefore carried on and in 2001 introduced Model-Driven Architecture to a waiting world with a series of one-day events around the world. David Frankel's eponymous book, written and edited as he and I flew around the world to introduce the MDA concept, came out shortly thereafter; key influencers joined us in the campaign to add major new OMG standardization efforts in the modeling space. We would continue to create, extend, and support existing standards and new standards in the middleware and vertical-market spaces, but we would add significant new activities. It occurred to me that we actually had already been in the modeling space from the beginning; one can think of the Interface Definition Language of CORBA and DDS as simply a poor-man's modeling language, with limited expression of semantics.

For a while, the "sparkle" our members looked for was in academia primarily. As an avid participant in several academic conferences a year, I can tell you that uptake of MDA concepts (and terminology, like "platform-specific model" and "platform-independent model") took off like a rocket in universities. It took time, but the next step was a technology "pull" from engineering organizations that needed to perform better than the past (many of whom had already been using MDA techniques, and now had a name to point to); the creation of the Eclipse Foundation, starting in 2002, and its early embrace of modeling technology, also helped greatly. By 2010, modeling was firmly embedded in the world's software engineering psyche, and Gartner and Forrester were reporting that more than 71 UML tools were available on the market and adopted at some level. That's some serious "sparkle," and OMG members reveled in the success.

An interesting parallel world began to appear around MOF and UML, recognizing that modeling languages didn't have to be limited to modeling software systems (or "software intensive systems," as many called them); that, in fact, most complex software systems have to interact with other complex engineered systems, from building architecture to complex devices like mobile phones and aircraft carriers. We decided to roll out an entire fleet of MOF-defined languages to address the needs of many different modelers and marketplaces:

- UML System on a Chip: for microchip hardware/firmware/software definition;
- SoaML: for service-oriented architectures;
- BPMN: for business process modelers;
- BMM: for modeling the motivations and structure of a business;
- SysML: for modeling large, complex systems of software, hardware, facilities, people, and processes;
- UPDM: for modeling enterprise architectures;
- CWM: for data warehouses.

Each of these have been successful in a well-defined marketplace, often replacing a mix of many other languages and techniques that have fragmented a market and market opportunity. Along the way, our terminology morphed, changed, and extended, with arguments about the difference between "model-driven" and "model-based;" one of my favorite memories is of a keynote speech I gave just a couple of years ago in Oslo, after which an attendee came up to argue with me about my definition of the phrase "model-driven architecture." He wasn't particularly impressed that I had made up the term; it reminded me of a classic (and possibly apocryphal) story about the brilliant pianist Glenn Gould, who when accosted by a composer for his interpretation of the composer's work, yelled, "You don't understand your own composition!"

Over the past decade many new phrases have appeared around MDA, and one of the ones I consider most apt is Model-Driven Software Engineering (MDSE). This history lesson brings us to the work of this book, to help the neophyte understand and succeed with the technologies that make up MDSE. What are these mystical "modeling languages," how do we transform (compile) from one to another, and most importantly, how does this approach bring down the cost of development, maintenance, and integration of these systems? These aren't mysteries at all, and this book does a great job enlightening us on the techniques to get the most from a model-driven approach.

I'd like to leave you, dear reader, with one more thought. Recently, I had the opportunity to create, with dear friends Ivar Jacobson and Bertrand Meyer, an international community dedicated to better formalizing the software development process, and moving software development

out of the fragmented "stone age" of insufficient theory chasing overwhelming need, to the ordered, structured engineering world on which other engineering disciplines depend. The Software Engineering Method and Theory (Semat) project brings together like-minded people worldwide to help bring software development into the 21st century, much as building architecture was driven into modernism by growing size and failures a millennium ago, and the shipbuilding industry had to formalize into ship blueprints some four centuries ago. My dream is that software engineering becomes engineering, and the huge stack of should-be-integrated engineering disciplines (civil, materials, software, hardware, etc.) be integrated into Model-Driven Engineering.

In your hands is part of the first step.

Richard Mark Soley, Ph.D.
Chairman and Chief Executive Officer
Object Management Group, Inc.
June 2012

10,000 meters over the central United States

Acknowledgments

This book wouldn't be the same without all the enriching discussions we have had with many other MDSE fans (and detractors!) during the last years—in person or within online forums. It would be almost impossible to list all of them here and therefore we wish to thank them all and to acknowledge their direct or indirect contribution to this book and to the MDE field at large, especially our current and previous colleagues.

An explicit mention must go to the ones who concretely helped us in the writing of this book. First of all, thanks to Diane Cerra, our Managing Editor at Morgan & Claypool, who believed in our project since the beginning and followed us with infinite patience throughout the whole book production process.

Secondly, thanks to Richard Soley, Chairman and CEO of OMG, who graciously agreed to introduce our work with his authoritative foreword.

And finally, last but not least, thanks to all the people that helped review the book: Ed Seidewitz (Model Driven Solutions), Davide di Ruscio (L'Aquila University), Juan Carlos Molina (Integranova), Vicente Pelechano (Polytechnic University of Valencia), and a bunch of our own colleagues and friends who carefully read and commented on what we were writing.

Marco Brambilla, Jordi Cabot, and Manuel Wimmer
January 2017

CHAPTER 1

Introduction

The human mind inadvertently and continuously re-works reality by applying cognitive processes that alter the subjective perception of it. Among the various cognitive processes that are applied, abstraction is one of the most prominent ones. In simple words, abstraction consists of the capability of finding the commonality in many different observations and thus generating a mental representation of the reality which is at the same time able to:

- generalize specific features of real objects (generalization);
- classify the objects into coherent clusters (classification); and
- aggregate objects into more complex ones (aggregation).

Actually, generalization, classification, and aggregation represent natural behaviors that the human mind is natively able to perform (babies start performing them since they are a few months old) and that are performed by people in their everyday life. Abstraction is also widely applied in science and technology, where it is often referred to as *modeling*. We can informally define a model as a simplified or partial representation of reality, defined in order to accomplish a task or to reach an agreement on a topic. Therefore, by definition, a model will never describe reality in its entirety.

1.1 PURPOSE AND USE OF MODELS

Models have been, and are, of central importance in many scientific contexts. Just think about physics or chemistry: the billiard ball model of a gas or the Bohr model of the atom are probably unacceptable simplifications of reality from many points of view, but at the same time have been paramount for understanding the basics of these fields; the uniform motion model in physics is something that will never be accomplished in the real world, but is extremely useful for teaching purposes and as a basis for subsequent, more complex theories. Mathematics and other formal descriptions have been extremely useful in all fields for modeling and building upon models. This has been proven very effective at description and powerful at prediction.

A huge branch of philosophy of science itself is based on models. Thinking about models at the abstract and philosophical level raises questions in semantics (i.e., the representational function performed by models), ontology (i.e., the kind of things that models are), epistemology (i.e., how to learn through or from models), and philosophy.

Models are recognized to implement at least two roles by applying abstraction:

- *reduction* feature: models only reflect a (relevant) selection of the original's properties, so as to focus on the aspects of interest; and
- *mapping* feature: models are based on an original individual, which is taken as a prototype of a category of individual and is abstracted and generalized to a model.

The *purpose* of models can be different too: they can be used for *descriptive purposes* (i.e., for describing the reality of a system or a context), *prescriptive purposes* (i.e., for determining the scope and details at which to study a problem), or for defining how a system shall be implemented.

In many senses, also considering that it is recognized that observer and observations alter the reality itself, at a philosophical level one can agree that "everything is a model," since nothing can be processed by the human mind without being "modeled." Therefore, it's not surprising that models have become crucial also in technical fields such as mechanics, civil engineering, and ultimately in computer science and computer engineering. Within production processes, modeling allows us to investigate, verify, document, and discuss properties of products before they are actually produced. In many cases, models are even used for directly automating the production of goods.

The discussion about whether modeling is good or bad is not really appropriate. We all always create a mental model of reality. In this sense, one can say that you cannot avoid modeling. This is even more appropriate when dealing with objects or systems that need to be developed: in this case, the developer must have in mind a model for her/his objective. The model always exists, the only option designers have is about its form: it may be mental (existing only in the designers' heads) or explicit [53]. In other words, the designer can decide whether to dedicate effort to realizing an explicit representation of the model or keeping it within her/his own mind.

1.2 MODELING FOR SOFTWARE DEVELOPMENT

The scope of this book is to discuss approaches based on modeling for the development of software artifacts. In this context, this is known as *Model-Driven Software Engineering* (MDSE). MDSE practices proved to increase efficiency and effectiveness in software development, as demonstrated by various quantitative and qualitative studies [1].

The need for relying on models for software development is based on four main facts.

1. Software artifacts are becoming more and more complex and therefore they need to be discussed at different abstraction levels depending on the profile of the involved stakeholders, phase of the development process, and objectives of the work.
2. Software is more and more pervasive in people's lives, and the expectation is that the need for new pieces of software or the evolution of existing ones will be continuously increasing.
3. The job market experiences a continuous shortage of software development skills with respect to job requests.

4. Software development is not a self-standing activity: it often imposes interactions with non-developers (e.g., customers, managers, business stakeholders, etc.) which need some mediation in the description of the technical aspects of development.

Modeling is a handy tool for addressing all these needs. That's why we strongly believe that MDSE techniques will see more and more adoption. This vision is supported by all the major players in this field (i.e., tool vendors, researchers, and enterprise software developers), and also by business analysts. For instance, Gartner foresees a broad adoption of model-driven techniques thanks to the convergence of software development and business analysis. This is particularly true in scenarios where the complexity of new, service-oriented architectures (SOAs) and cloud-based architectures, jointly applied with business process management (BPM), demand more abstract approaches than mere coding. Also from the standardization and tooling point of view, business and IT modeling and technologies are converging. This brings huge benefits to organizations, which struggle to bridge the gap between business requirements and IT implementation. Companies try to exploit this convergence and to ensure collaboration among the IT organization, business process architects, and analysts who are using and sharing models. Obviously, this also implies organizational changes that should move toward more agile approaches, combined with fostering modeling efforts and reusable design patterns and frameworks to improve productivity, while ensuring quality and performance [11].

A different discussion is the use people make of models (based on Martin Fowler's classification[1]):

- models as sketches: models are used for communication purposes, only partial views of the system are specified;
- models as blueprints: models are used to provide a complete and detailed specification of the system; and
- models as programs: models, instead of code, are used to develop the system.

Of course, during the course of a development process, a team can use the models in several different ways. For instance, while discussing design decisions models could be used as sketches as an aid for the discussion; and after, complete models could be defined as part of the blueprint of the system. Finally, these blueprint models may be further refined to create the system using code generation techniques to minimize the coding tasks.

1.3 HOW TO READ THIS BOOK

This is an introductory book to model-driven practices and aims at satisfying the needs of a diverse audience.

[1] http://martinfowler.com/bliki/UmlMode.html

Our aim is to provide you with an agile and flexible book to introduce you to the MDSE world, thus allowing you to understand the basic principles and techniques, and to choose the right set of instruments based on your needs. A word of caution: this is neither a programming manual nor a user manual of a specific technology, platform, or toolsuite. Through this book you will be able to delve into the engineering methodologies, select the techniques more fitting to your needs, and start using state of the art modeling solutions in your everyday software development activities.

We assume the reader is familiar with some basic knowledge of software engineering practices, processes, and notations, and we also expect some software programming capabilities.

The book is organized into two main parts.

- The first part (from Chapter 2 to Chapter 6) discusses the *foundations of MDSE* in terms of principles (Chapter 2), typical ways of applying it (Chapter 3), a standard instantiation of MDSE (Chapter 4), the practices on how to integrate MDSE in existing development processes (Chapter 5), and an overview of modeling languages (Chapter 6).
- The second part (from Chapter 7 to Chapter 10) deals with the *technical aspects of MDSE*, spanning from the basics on how to build a domain-specific modeling language (Chapter 7), to the description of Model-to-Model and Model-to-Text transformations (Chapters 8 and 9, respectively), and the tools that support the management of MDSE artifacts (Chapter 10).

We may classify the attitude of a reader of this book into three main categories: the curious, the user, and the developer—plus, the atypical one of the B.Sc., M.Sc, or Ph.D. student or who needs to address some academic course on MDSE. It's up to you to decide what are your expectations from this book. Obviously, you may want to read the book from the first to the last page anyway, but here we try to collect some suggestions on what not to miss, based on your profile.

MDSE Curious

If you are interested in general in MDSE practices and you would like to learn more about them, without necessarily expecting to do the technical work of using model-driven approaches or developing modeling languages, you are probably *MDSE curious*.

This attitude is extremely useful for CTOs, CIOs, enterprise architects, business and team managers who want to have a bird's eye view on the matter, so as to make the appropriate decisions when it comes to choosing the best development techniques for their company or team.

In this case, our suggestion is to focus on the first part of the book, namely the: MDSE principles (Chapter 2) and usage (Chapter 3); MDA proposal by OMG (Chapter 4); and overview on the modeling languages (Chapter 6). We also suggest reading how to integrate MDSE in existing development processes (Chapter 5) and possibly the tools that support the management of MDSE artifacts (Chapter 10).

MDSE User

If you are a technical person (e.g., software analyst, developer, or designer) and you expect to use in your development activities some MDSE approach or modeling notation, with the purpose of improving your productivity and learning a more conceptual way for dealing with the software development problem, you are propably an *MDSE user*. In this case, you are most likely not going to develop new modeling languages or methods, but you will aim at using the existing ones at your best.

If you think you fall into this category, we recommend that you read at least the basics on MDSE principles and usage in Chapters 2 and 3; overview on OMG's MDA approach (Chapter 4) and on modeling languages (Chapter 6); description of Model-to-Model and Model-to-Text transformations (Chapters 8 and 9, respectively); and tools that support the management of MDSE projects (Chapter 10).

MDSE Developer

Finally, if you already have some basic knowledge of MDSE but you want to move forward and become a hardcore MDSE adopter, by delving into the problems related to defining new DSLs, applying end-to-end MDSE practices in your software factory, and so on, then we classify your profile as *MDSE developer*.

As such, you are probably interested mostly in the second part of the book, which describes all the details on how to technically apply MDSE. Therefore, we recommend that you read at least the following chapters: the overview on modeling languages (Chapter 6), the basis on domain-specific modeling languages (Chapter 7); the description of Model-to-Model and Model-to-Text transformations (Chapters 8 and 9, respectively), and the tools that support the management of MDSE artifacts (Chapter 10).

Optionally, if you need to refresh your mind on the basic MDSE principles, you can read Chapters 2 and 3. As a final read, if you are also involved in management activities, you can read the chapter on agile development processes (Chapter 5).

Student

If you are a *student* of an academic or professional course, our suggestion is to go through the whole book so as to get at least the flavor of the objectives and implications of the MDSE approach. In this case, this book possibly will not be enough to cover in detail all the topics of the course you are following. Throughout the text we offer a good set of references to books, articles, and online resources that will help you investigate the topics you need to study in detail.

Instructor

If you are an *instructor* of an academic or professional course, you will be able to target your specific teaching needs by selecting some chapters of the book depending on the time span available in your course and the level of technical depth you want to provide your students with. The teaching

materials covering the whole book are available as slidesets, linked from the book web site (http://www.mdse-book.com).

In addition to the contents of the book, more resources are provided on the book's website (http://www.mdse-book.com), including the examples presented in the book.

CHAPTER 2

MDSE Principles

Models are paramount for understanding and sharing knowledge about complex software. MDSE is conceived as a tool for making this assumption a concrete way of working and thinking, by transforming models into first-class citizens in software engineering. Obviously, the purpose of models can span from communication between people to executability of the designed software: the way in which models are defined and managed will be based on the actual needs that they will address. Due to the various possible needs that MDSE addresses, its role becomes that of defining sound engineering approaches to the definition of models, transformations, and their combinations within a software development process.

This chapter introduces the basic concepts of MDSE and discusses its adoption and perspectives.

2.1 MDSE BASICS

MDSE can be defined as a methodology[1] for applying the advantages of modeling to software engineering activities. Generally speaking, a methodology comprises the following aspects.

- *Concepts*: The components that build up the methodology, spanning from language artifacts to actors, and so on.
- *Notations*: The way in which concepts are represented, i.e., the languages used in the methodology.
- *Process and rules*: The activities that lead to the production of the final product, the rules for their coordination and control, and the assertions on desired properties (correctness, consistency, etc.) of the products or of the process.
- *Tools*: Applications that ease the execution of activities or their coordination by covering the production process and supporting the developer in using the notations.

All of these aspects are extremely important in MDSE and will be addressed in this book.

In the context of MDSE, the *core concepts* are: models and transformations (i.e., manipulation operations on models). Let's see how all these concepts relate together, revisiting the famous equation from Niklaus Wirth:

[1]We recognized that methodology is an overloaded term, which may have several interpretations. In this context, we are not using the term to refer to a formal development process, but instead as a set of instruments and guidelines, as defined in the text above.

Algorithms + Data Structures = Programs

In our new MDSE context, the simplest form of this equation would read as follows:

Models + Transformations = Software

Obviously, both models and transformations need to be expressed in some *notation*, which in MDSE we call a modeling language (in the same way that in the Wirth equation, algorithms and data structures need to be defined in some programming language). Nevertheless, this is not yet enough. The equation does not tell us what kinds of models (and in which order, at what abstraction level, etc.) need to be defined depending on the kind of software we want to produce. That's where the model-driven *process* of choice comes to play. Finally, we need an appropriate set of *tools* to make MDSE feasible in practice. As for programming, we need IDEs that let us define the models and transformations as well as compilers or interpreters to execute them and produce the final software artifacts.

Given this, it goes without saying that MDSE takes the statement "everything is a model" very seriously. In fact, in this context one can immediately think of all the ingredients described above as something that can be modeled. In particular, one can see transformations as particular models of operations upon models. The definition of a modeling language itself can be seen as a model: MDSE refers to this procedure as *metamodeling* (i.e., modeling a model, or better: modeling a modeling language; or, modeling all the possible models that can be represented with the language). And this can be recursive: modeling a metamodel, or describing all metamodels, means to define a meta-metamodel. In the same way one can define as models also the processes, development tools, and resulting programs.

MDSE tries to make the point that the statement "everything is a model" has a strong unifying and driving role in helping adoption and coherence of model-driven techniques, in the same way as the basic principle "everything is an object" was helpful in driving the technology in the direction of simplicity, generality, and power of integration for object-oriented languages and technologies in the 1980s [10].

Before delving into the details of the terminology, one crucial point that needs to be understood is that MDSE addresses design of software with a *modeling* approach, as opposed to a *drawing* one. In practical terms, we distinguish these two approaches because drawing is just about creating nice pictures, possibly conforming to some syntactical rules, for describing some aspects of the design. On the other side, modeling is a much more complex activity that possibly implies graphical design (but that could be replaced by some textual notations), but it's not limited to depicting generic ideas: in modeling the drawings (or textual descriptions) have implicit but unequivocally defined semantics which allow for precise information exchange and many additional usages. Modeling, as opposed to simply drawing, grants a huge set of additional advantages, including: syntactical validation, model checking, model simulation, model transformations, model execution (either through code generation or model interpretation), and model debugging.

2.2 LOST IN ACRONYMS: THE MD* JUNGLE

The first challenge that a practitioner faces when addressing the model-driven universe is to cope with the plethora of different acronyms which could appear as obscure and basic synonyms. This section is a short guide on how to get out of the acronym jungle without too much burden. Figure 2.1 shows a visual overview of the relations between the acronyms describing the modeling approaches.

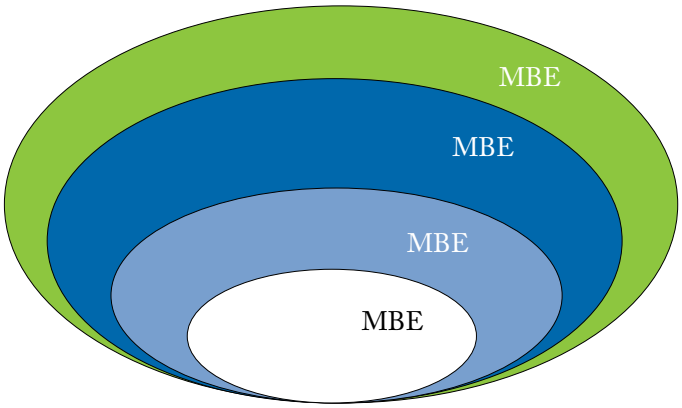

Figure 2.1: Relationship between the different MD* acronyms.

Model-Driven Development (MDD) is a development paradigm that uses models as the primary artifact of the development process. Usually, in MDD the implementation is (semi)automatically generated from the models.

Model-Driven Architecture (MDA) is the particular vision of MDD proposed by the Object Management Group (OMG) and thus relies on the use of OMG standards. Therefore, MDA can be regarded as a subset of MDD, where the modeling and transformation languages are standardized by OMG.

On the other hand, MDE would be a superset of MDD because, as the E in MDE suggests, MDE goes beyond the pure development activities and encompasses other model-based tasks of a complete software engineering process (e.g., the model-based evolution of the system or the model-driven reverse engineering of a legacy system).

Finally, we use "model-based engineering" (or "model-based development") to refer to a softer version of MDE. That is, the MBE process is a process in which software models play an important role although they are not necessarily the key artifacts of the development (i.e., they do NOT "drive" the process as in MDE). An example would be a development process where, in the analysis phase, designers specify the domain models of the system but subsequently these models are directly handed out to the programmers as blueprints to manually write the code (no automatic code generation involved and no explicit definition of any platform-specific

model). In this process, models still play an important role but are not the central artifacts of the development process and may be less complete (i.e., they can be used more as blueprints or sketches of the system) than those in an MDD approach. MBE is a superset of MDE. All model-driven processes are model-based, but not the other way round.

All the variants of "model-driven whatever" are often referred to with the acronym *MD* (Model-Driven star)*. Notice that a huge set of variants of all these acronyms can be found in literature too. For instance, MDSE (Model-Driven Software Engineering), MDPE (Model-Driven Product Engineering), and many others exist. MDE can be seen as the superset of all these variants, as any MD*E approaches could fall under the MDE umbrella. The focus of this book is on MDSE.

2.3 OVERVIEW OF THE MDSE METHODOLOGY

In this section we delve into the details of the MDSE ingredients and philosophy. In particular, we will clarify that modeling can be applied at different levels of abstractions, and that a full-fledged modeling approach even leads to modeling the models themselves. Finally, we will describe the role and nature of model transformations.

2.3.1 OVERALL VISION

MDSE provides a comprehensive vision for system development. Figure 2.2 shows an overview of the main aspects considered in MDSE and summarizes how the different issues are addressed. MDSE seeks for solutions according to orthogonal dimensions: conceptualization (columns in the figure) and implementation (rows in the figure).

The *implementation* issue deals with the mapping of the models to some existing or future running systems. Therefore, it consists of defining three core aspects.

- The modeling level: where the models are defined.
- The realization level: where the solutions are implemented through artifacts that are actually in use within the running systems (this consists of code in case of software).
- The automation level: where the mappings from the modeling to the realization levels are put in place.

The *conceptualization* issue is oriented to defining conceptual models for describing reality. This can be applied at three main levels.

- The application level: where models of the applications are defined, transformation rules are performed, and actual running components are generated.
- The application domain level: where the definition of the modeling language, transformations, and implementation platforms for a specific domain are defined.

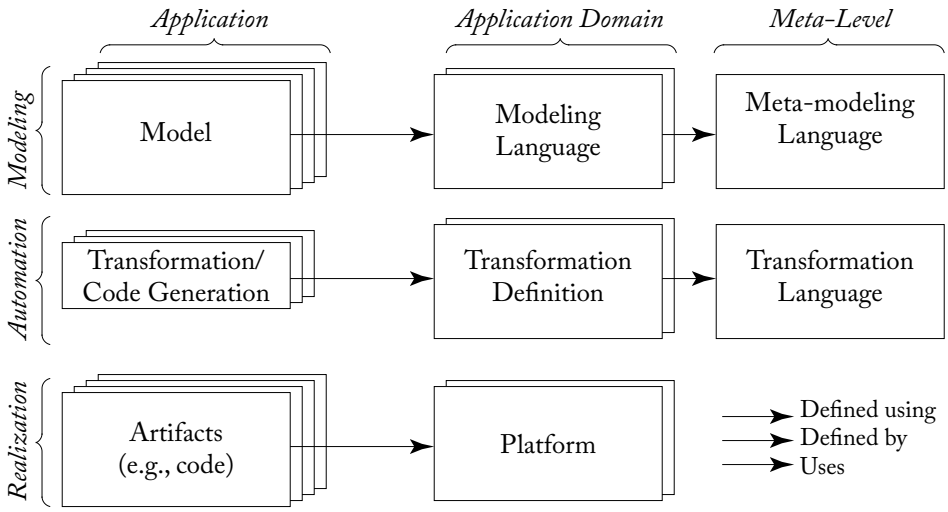

Figure 2.2: Overview of the MDSE methodology (top-down process).

- The meta-level: where conceptualization of models and of transformations are defined.

The core flow of MDSE is from the application models down to the running realization, through subsequent model transformations. This allows reuse of models and execution of systems on different platforms. Indeed, at the realization level the running software relies on a specific platform (defined for a specific application domain) for its execution.

To enable this, the models are specified according to a modeling language, in turn defined according to a metamodeling language. The transformation executions are defined based on a set of transformation rules, defined using a specific transformation language.

In this picture, the system *construction* is enabled by a *top-down process* from *prescriptive models* that define how the scope is limited and the target should be implemented. On the other side, *abstraction* is used *bottom-up* for producing *descriptive models* of the systems.

The next subsections discuss some details on modeling abstraction levels, conceptualization, and model transformations.

2.3.2 DOMAINS, PLATFORMS, AND TECHNICAL SPACES

The nature of MDE implies that there is some context to model and some target for the models to be transformed into. Furthermore, the software engineering practices recommend to distinguish between the problem and the solution spaces: the *problem space* is addressed by the *analysis* phase in the development process, while the *solution space* is addressed by the *requirements collection* phase first (defining *what* is the expected outcome), and subsequently by the *design* phase (specifying

how to reach the objective). The common terminology for defining such aspects in MDSE is as follows.

The *problem space* (also known as problem domain) is defined as the field or area of expertise that needs to be examined to understand and define a problem. The *domain model* is the conceptual model of the problem domain, which describes the various entities, their attributes, roles, and relationships, plus the constraints and interactions that describe and grant the integrity of the model elements comprising that problem domain. The purpose of domain models is to define a common understanding of a field of interest, through the definition of its vocabulary and key concepts. One crucial recommendation is that the domain model must not be defined with a look-ahead attitude toward design or implementation concerns, while it should only describe assets and issues in the problem space.

On the contrary, the *solution space* is the set of choices at the design, implementation, and execution level performed to obtain a software application that solves the stated problem within the problem domain.

Technical spaces represent working contexts for the design, implementation, and execution of such software applications. These working contexts typically imply a binding to specific implementation technologies (which can be combined together into a coherent *platform*) and languages.

The concept of technical space is crucial for MDSE because it enables the possibility of deciding the set of technical tools and storage formats for models, transformations, and implementations. Notice that a technical space can either span both the problem and solution domains or cover only one of these aspects.

Figure 2.3 shows some examples of technical spaces, which span different phases of the development: MDSE, spanning from the problem definition down to the design, implementation, and even execution of the application (e.g., through model interpretation techniques); XML and the Java framework, which are more oriented toward implementation.

During the software development process it is possible to move from one technical space to another (as represented by the arrows in the figure). This implies the availability of appropriate software artifacts (called *extractors*) that are able to extract knowledge from a technical space and of others (called *injectors*) that are able to inject such knowledge in another technical space.

Notice also that the way in which models are transformed and/or moved from one technical space to another depends on the business objective of the development activity: indeed, MDSE can be applied to a wide set of scenarios, spanning from software development automation, to system interoperability, reverse engineering, and system maintenance. These aspects will be addressed extensively in Chapter 3.

2.3.3 MODELING LANGUAGES

As we will see in detail later, modeling languages are one of the main ingredients of MDSE. A modeling language is a tool that lets designers specify the models for their systems, in terms of graphical or textual representations. In any case, languages are formally defined and ask the

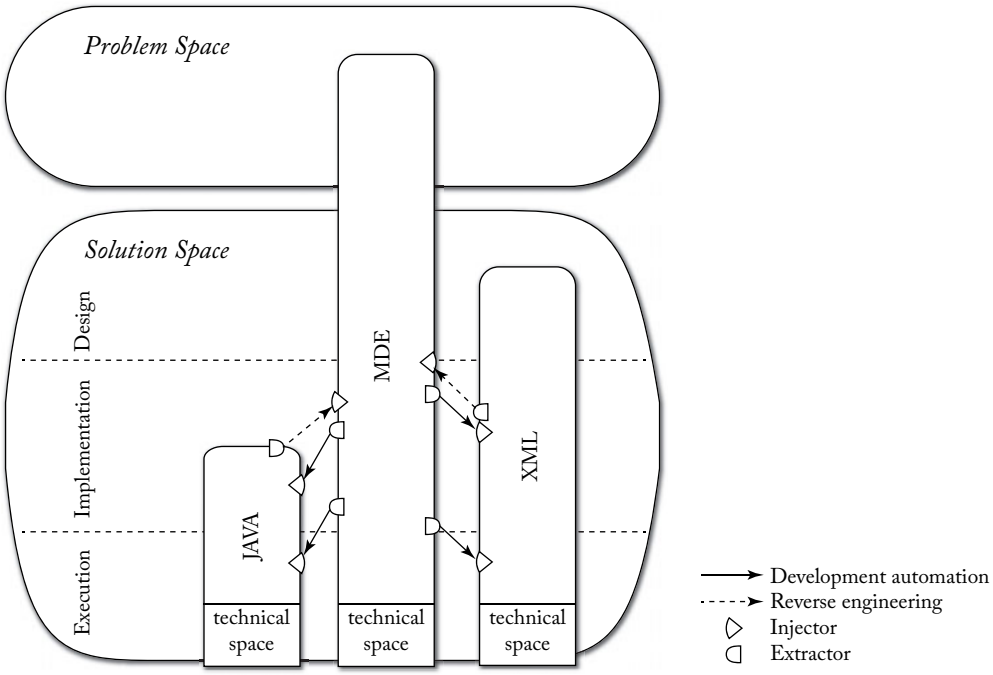

Figure 2.3: Technical spaces examples and coverage.

designers to comply with their syntax when modeling. Two big classes of languages can be identified.

Domain-Specific Languages (DSLs) are languages that are designed specifically for a certain domain, context, or company to ease the task of people that need to describe things in that domain. If the language is aimed at modeling, it may be also referred to as *Domain-Specific Modeling Language (DSML)*. DSLs have been largely used in computer science even before the acronym existed: examples of domain-specific languages include the well-known HTML markup language for Web page development, Logo for pen-based simple drawing for children, VHDL for hardware description languages, Mathematica and MatLab for mathematics, SQL for database access, and so on.

General-Purpose Modeling Languages (*GPML*s, *GML*s, or *GPL*s) instead represent tools that can be applied to any sector or domain for modeling purposes. The typical example for these kinds of languages is the UML language suite, or languages like Petri-nets or state machines.

To avoid misunderstandings and different variants on the naming, we will use DSL and GPL as acronyms for these two classes in the rest of this book.

Within these classes, further distinctions and classifications can be defined. Given that modeling inherently implies abstraction, a very simple way of classifying the modeling languages

and the respective models is based on the *level of abstraction* at which the modeling is performed. Intuitively, it's easy to understand that some models are more abstract than others. In particular, when dealing with information systems design, one can think of alternative models that:

- describe requirements and needs at a very abstract level, without any reference to implementation aspects (e.g., description of user requirements or business objectives);
- define the behavior of the systems in terms of stored data and performed algorithms, without any technical or technological details;
- define all the technological aspects in detail.

Given the different modeling levels, appropriate transformations can be defined for mapping a model specified at one level to a model specified at another level.

Some methods, such as MDA, provide a fixed set of modeling levels, which make it easier to discuss the kinds of models that a specification is dealing with. In particular, MDA defines its abstraction hierarchy as from the list above.

Models are meant to describe two main dimensions of a system: the static (or structural) part and the dynamic (or behavioral) part. Thus, we can define the following:

- *Static models*: Focus on the static aspects of the system in terms of managed data and of structural shape and architecture of the system.
- *Dynamic models*: Emphasize the dynamic behavior of the system by showing the execution sequence of actions and algorithms, the collaborations among system components, and the changes to the internal state of components and applications.

This separation highlights the importance of having different views on the same system: indeed, a comprehensive view on the system should consider both static and dynamic aspects, preferably addressed separately but with the appropriate interconnections.

Without a doubt, multi-viewpoint modeling is one of the crucial principles of MDSE. Since modeling notations are focused on detailing one specific perspective, typically applying an MDSE approach to a problem may lead to building various models describing the same solution. Each model is focused on a different perspective and may use a different notation. The final purpose is to provide a comprehensive description of the system, while keeping the different concerns separated. That's why models can be interconnected through cross-referencing the respective artifacts.

While in principle, it's possible to define a design as composed of models based on several independent notations (possibly coming from different standardization bodies, or even including proprietary or third-party notations), it is convenient to exploit a suite of notations that, despite being different and addressing orthogonal aspects of the design, have a common foundation and are aware of each other. That's the reason why general-purpose languages typically do not include just one single notation, but instead include a number of coordinated notations that complement

each other. These languages are also known as *Modeling Language Suites*, or *Family of Languages*, as they are actually composed of several languages, not just one. The most known example of language suites is UML itself, which allows the designers to represent several different diagram types (class diagram, activity diagram, sequence diagram, and so on).

2.3.4 METAMODELING

As models play a pervasive role in MDSE, a natural subsequent step to the definition of models is to represent the models themselves as "instances" of some more abstract models. Hence, exactly in the same way we define a model as an abstraction of phenomena in the real world, we can define a *metamodel* as yet another abstraction, highlighting properties of the model itself. In a practical sense, metamodels basically constitute the definition of a modeling language, since they provide a way of describing the whole class of models that can be represented by that language.

Therefore, one can define models of the reality, and then models that describe models (called metamodels) and recursively models that describe metamodels (called meta-metamodels). While in theory one could define infinite levels of metamodeling, it has been shown, in practice, that meta-metamodels can be defined based on themselves, and therefore it usually does not make sense to go beyond this level of abstraction. At any level where we consider the metamodeling practice, we say that a model *conforms* to its metamodel in the way that a computer program conforms to the grammar of the programming language in which it is written. More specifically, we say that a model conforms to its metamodel when all its elements can be expressed as instances of the corresponding metamodel (meta)classes as seen in Figure 2.4.

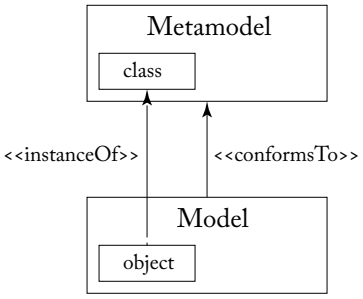

Figure 2.4: *conformsTo* and *instanceOf* relationships.

Figure 2.5 shows an example of metamodeling at work: real-world objects are shown at level M0 (in this example, a movie); their modeled representation is shown at level M1, where the model describes the concept of Video with its attributes (title in the example). The meta-model of this model is shown at level M2 and describes the concepts used at M1 for defining the model, namely: Class, Attribute, and Instance. Finally, level M3 features the meta-metamodel that defines the concepts used at M2: this set collapses in the sole Class concept in the example.