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.

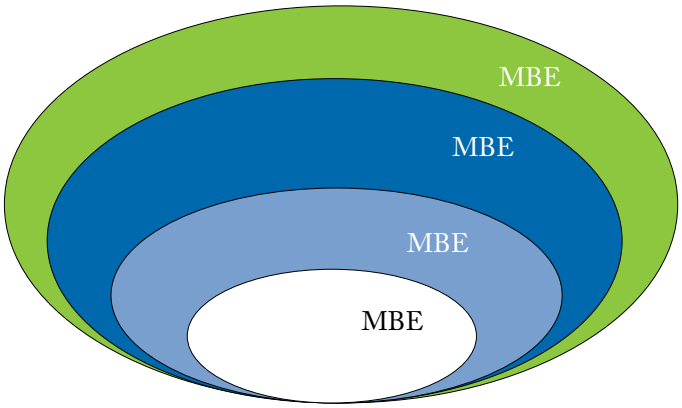

Figure 2.1: Relationship between the different MD* acronyms.

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.

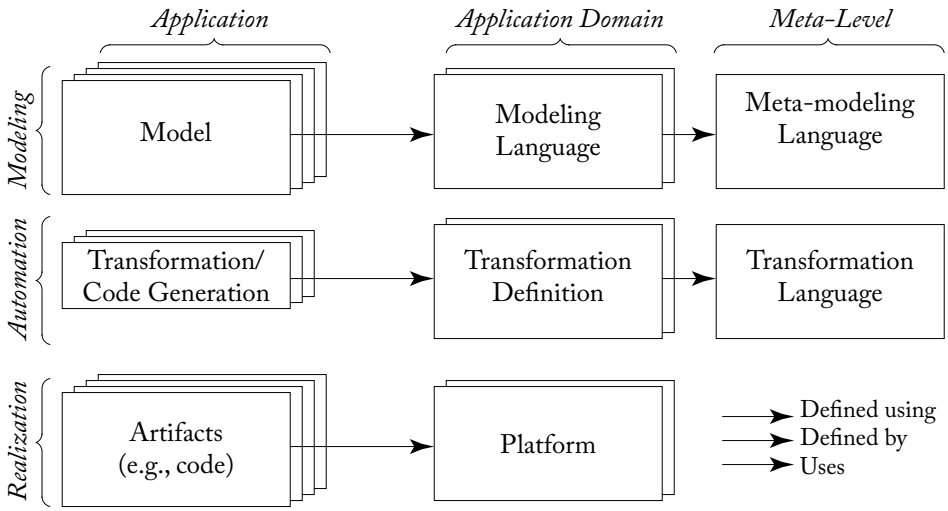

Figure 2.2: Overview of the MDSE methodology (top-down process).

- 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

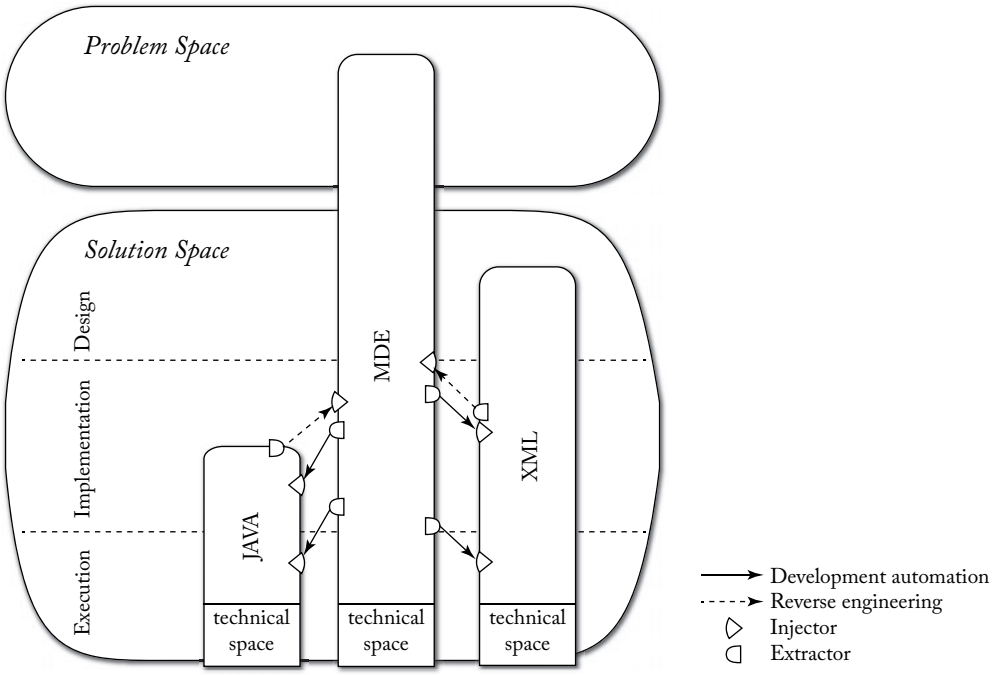

Figure 2.3: Technical spaces examples and coverage.

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.

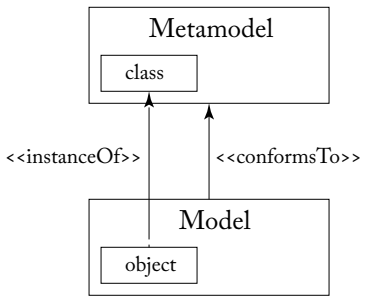

Figure 2.4: *conformsTo* and *instanceOf* relationships.

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.

It's clear that there is no need for further metamodeling levels beyond M3, as they would always include only the Class concept.

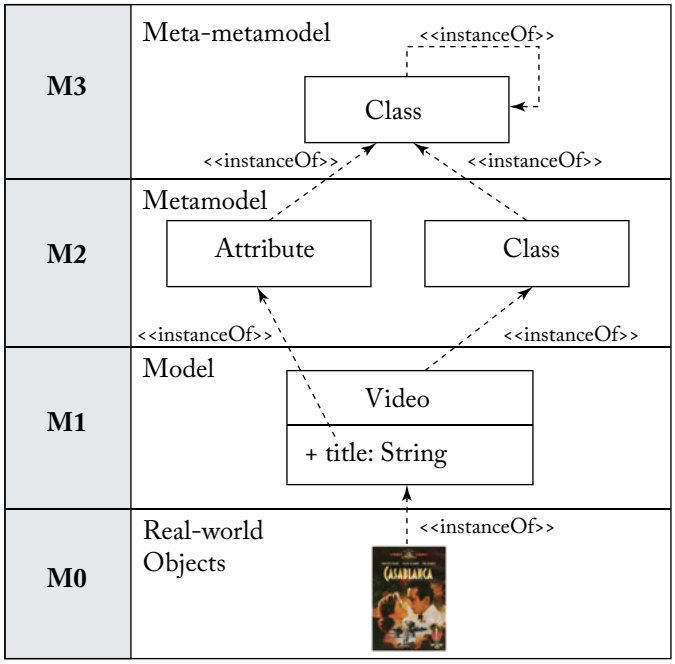

Figure 2.5: Models, metamodels, and meta-metamodels.

Although, one could think that metamodels represent a baroque conceptualization, metamodeling is extremely useful in practice (even when it is not recognized as such). Metamodels can be used proficiently for:

- defining new languages for modeling or programming;
- defining new modeling languages for exchanging and storing information; and
- defining new properties or features to be associated with existing information (metadata).

Notably, the term *metadata* is defined with the same approach and aims at describing data about existing data. Therefore, metamodeling perfectly fits with the description of such information.

Notice that the MDSE paradigm, also known as *modelware,* is not so different from grammarware (i.e., the technical space where languages are defined in terms of grammars) in terms of basic definition and infrastructure. This is highlighted in Figure 2.6: grammarware (shown on the right-hand side of the picture) can be seen as a broad technical space for defining languages through a domain-specific syntax. One of the most common ways for defining a language is the

EBNF form, which is a textual representation of the grammar rules for the language (an alternative graphical representation also exists, known as syntax graph). A grammar definition language can be recursively defined based on itself (as represented in the cyclic arrow at the topmost level in the grammarware stack). A specific language conforms to its specification and a model (or program or text) can be defined according to the language. Analogously, in the MDSE stack a language definition facility is available, recursively defined based on itself, and all the language definitions can be specified according to it. Finally, all the models will conform to the specified language. The image describes this stack within the standardized MDA framework, as specified by OMG (we will explore this case in detail in Chapter 4).

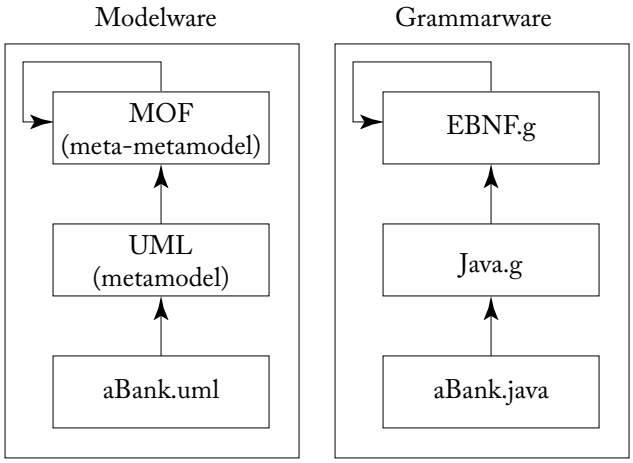

Figure 2.6: Modelware vs. Grammarware.

2.3.5 TRANSFORMATIONS

Besides models, model transformations represent the other crucial ingredient of MDSE and allow the definition of mappings between different models. Transformations are actually defined at the metamodel level, and then applied at the model level, upon models that conform to those metamodels. The transformation is performed between a source and a target model, but it is actually defined upon the respective metamodels (see also Figure 2.7).

MDSE provides appropriate languages for defining model transformations in order to provide designers with optimized solutions for specifying transformation rules. These languages can be used for defining model transformations in terms of transformation templates that are typically applied upon models according to some matching rules checked upon model elements.

Such transformation rules can be defined following different approaches: the transformation can be written manually from scratch by a developer, or can be defined as a refined specifica-

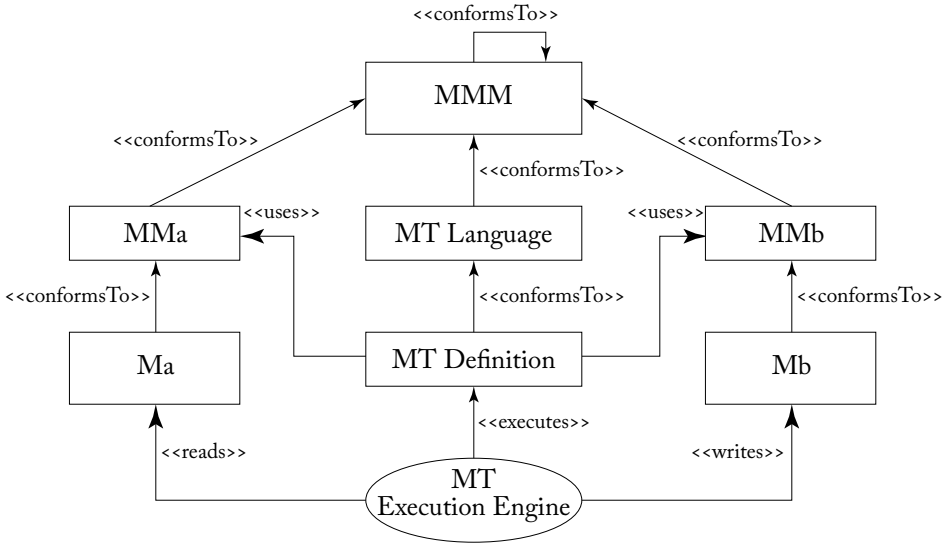

Figure 2.7: Role and definition of transformations between models.

tion of an existing one. Alternatively, transformations themselves can be produced automatically out of some higher level mapping rules between models. This technique is based on two phases:

1. defining a mapping between elements of a model to elements of another one (*model mapping* or *model weaving*); and

2. automating the generation of the actual transformation rules through a system that receives as input the two model definitions and the mapping between them and produces the transformations.

This allows the developers to concentrate on the conceptual aspects of the relations between models and then delegate the production of the transformation rules (possibly implemented within different technical spaces).

Another interesting aspect related to the vision of "everything is a model" is the fact that *transformations themselves can be seen as models*, and managed as such, including their metamodeling. The lower part of the Figure 2.7 shows two models (M_a and M_b), and a transformation M_t that transforms M_a into M_b. In the level above, the corresponding metamodels are defined (MM_a, MM_b, and MM_t), to which the three models (M_a, M_b, and M_t) conform, respectively. In turn, they all conform to the same meta-metamodel.

2.4 TOOL SUPPORT

Besides the conceptual tools provided by MDSE in terms of modeling and transformation languages, developers are provided with a large and diverse set of modeling tools that support the development of models, their transformation, and their integration within the software development process. Developers can nowadays choose among a complete spectrum of tools for MDSE, from open source to free to commercial ones, from full frameworks to individual tools, from desktop ones to cloud-based and SaaS (Software-as-a-service) solutions. Some of them support the whole MDSE process, in the sense that they provide facilities for defining new languages, models, transformations, and implementation platforms. Some others are design tools that support domain-specific notations or general-purpose ones.

2.4.1 DRAWING TOOLS VS. MODELING TOOLS

Many people assume that drawing tools and modeling tools are two interchangeable concepts but this is far from true. In fact, only some tools are drawing and modeling tools at the same time.

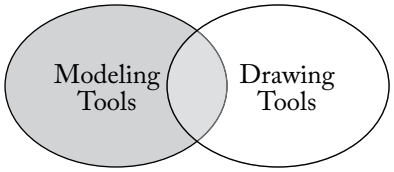

Figure 2.8: Modeling tools vs. drawing tools.

Some modeling tools use a concrete *textual syntax* for specifying models (not all models need to be graphical models), and therefore, there is no support for drawing them, even if the tools may be able to render the textual definition into some kind of graphical export format for visualization purposes. For instance, one can think about the textual modeling tools for UML[2] or any of the DSLs you may create with tools like XText[3] and EMFText.[4]

Furthermore, many drawing tools are not appropriate modeling tools. A drawing tool can be only considered as a modeling tool only if the tool "understands" the drawings, i.e., the tool does not deal with just shapes, lines, and arrows, but understands that they represent, e.g., classes, associations, or other modeling concepts. This should at least be enough to validate a model, i.e., check that the model is a correct instance of its metamodel.

For instance, a drawing tool may allow the designer to draw something funny like the model in Figure 2.9. The icons and shapes may be the ones defined in a given language (in the example, the UML notation) but the model may have no meaning at all, because the way elements are used

[2] http://modeling-languages.com/uml-tools/#textual
[3] http://www.eclipse.org/Xtext
[4] http://www.emftext.org

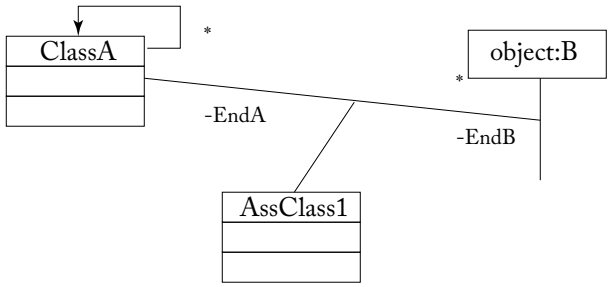

Figure 2.9: Wrong model defined with a drawing tool.

breaks the language rules. MDSE recommends always using a modeling tool to define models because of at least three main reasons.

- A modeling tool is able to export or manipulate the models using (external) access APIs provided by the tools. The API of a drawing tool may offer methods like *getAllRectangularShapes* but not a method like *getAllClasses*. The latter *is* the kind of API method exposed in a modeling tool, which is needed to easily manipulate models. The same discussion applies to the export formats. Drawing tools hardly offer any kind of semantic-aware export format that can be easily imported into other (modeling) tools.

- A modeling tool guarantees a minimum level of semantic meaning and model quality, because it grants alignment to some kind of metamodel. For the same drawing effort, having a model as an outcome (and not just a drawing) is much more valuable, even if the model has been defined just for communication purposes. In fact, it is very likely that this model could become useful for prototyping or code generation too, and for those purposes you must be able to use the model as a part of an MDSE chain.

- A modeling tool typically provides appropriate features for model transformations. Obviously, since at the end models are encoded as files, one can think to use usual imperative programming languages for defining model transformations. However, this lowers the level of abstraction of the entire modeling framework and typically ends up in producing cumbersome and unmaintainable pieces of software. That's why MDSE pushes for adopting declarative model transformation languages that feature a set of primitives explicitly targeted to defining the transformation rules.

2.4.2 MODEL-BASED VS. PROGRAMMING-BASED MDSE TOOLS

One initial distinction that can be made between MDSE tools is their classification in terms of the development paradigm used for their implementation. Indeed, MDSE tools themselves can be developed using traditional coding techniques or by applying the same MDSE principles that they aim to promote and support. Although both approaches are valid, in this book we

will put more emphasis on model-based MDSE tools, due to their better integration, conceptual simplicity, and also intellectual coherency and honesty (in a sense, one should apply the principles one advocates).

2.4.3 ECLIPSE AND EMF

Besides the various proprietary modeling tools that are available for model-based enterprise development, one tooling platform that has become prominent in the MDSE world is the Eclipse development environment. A set of interesting tools for MDSE have been made available under Eclipse, thus enabling a fertile flourishing of initiatives upon this platform. In this book we will mainly refer to the Eclipse framework since it's open source (but allows commercial extensions) and comprises popular components for all the modeling tasks we will describe.

The Eclipse Modeling Framework (EMF) is the core technology in Eclipse for model-driven engineering. EMF is a good representative of model-based MDSE tools for various reasons. First, EMF allows the definition of metamodels based on the metamodeling language called *Ecore*. Second, EMF provides generator components for producing from metamodels (i) a specific Java-based API for manipulating models programmatically and (ii) modeling editors to build models in tree-based editors. Third, EMF comes with a powerful API covering different aspects such as serializing and deserializing models to/from XMI as well as powerful reflection techniques. Fourth, based on EMF, several other projects are defined which provide further functionalities for building model-based development support within Eclipse.

2.5 ADOPTION AND CRITICISMS OF MDSE

Modeling is frequently prescribed and advocated by software engineering practitioners and consultants. However, its adoption in industry is less frequently undertaken [6]. Some studies done in the past years show that modeling is not extensively adopted among practitioners [20] or it is not a core part of the development process [24], although more recent studies have a more positive view [34].

It is true that modeling has yet to cross the chasm and become mainstream, mainly due to inflated expectations raised in the past by various MDSE initiatives, including MDA and before that plain UML-based proposals. However, new developments in the last few years in terms of new technologies and methods (like domain-specific languages), the widening of the scope of the field (modeling is not only used for code generation but also for reverse engineering, interoperability, and software maintenance), a more pragmatic modeling approach (no need to model every detail for every project), and the release of *de facto* standard tools (in particular the EMF open-source modeling framework on top of the Eclipse platform) have changed the landscape and suggest that mainstream industrial adoption may happen sooner than later. As Steve Mellor likes to say, "modeling and model-driven engineering will be commonplace in three years time."[5]

[5]With the caveat that, as he said, he has been giving the same answer since 1985!

As with basically any other new technology/paradigm, MDSE is following the typical technology hype cycle (see Figure 2.10), where after an initial peak of inflated expectations (mostly caused by the popularization of UML, sold by some vendors and consultancy companies as the silver bullet for software development), MDSE fell down to the trough of disillusionment. We are now at the slope of enlightenment where our better knowledge of what MDSE is for and how it must be used helps companies to choose better the projects where MDSE can make a difference so that we can soon achieve the plateau of productivity where companies will be really able to take profit from the benefits MDSE brings to the table. We hope this book contributes to this evolution.

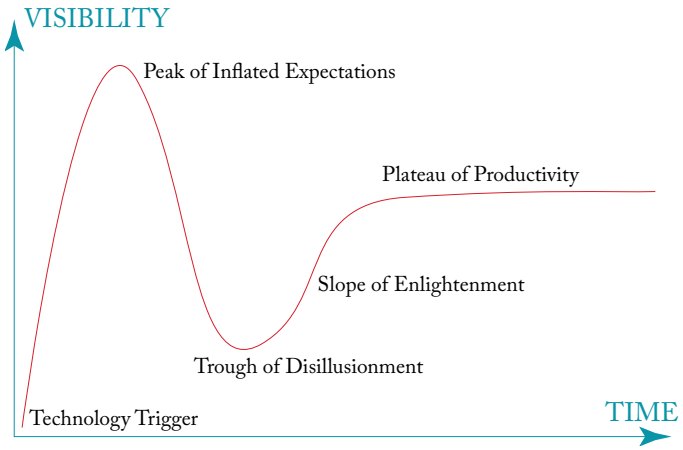

Figure 2.10: Technology hype cycle.

At this stage, adoption of MDSE still offers plenty of opportunities and a competitive advantage over more conservative organizations that will wait until the end before jumping on the bandwagon (Figure 2.11).

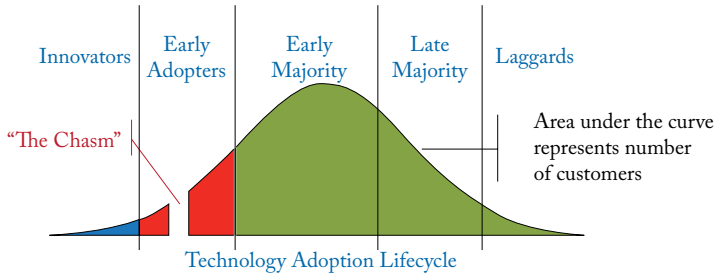

Figure 2.11: Technology adoption life cycle.

Despite the technical change that companies must face when embracing MDSE, experts often think that the biggest challenges faced by companies adopting model-driven approaches are not so much on the technical side, but much more on the human factor side [34]. Here are some responses by well-known experts in the field when asked about the biggest difficulty in embracing MDSE, BPM, SOA, and similar practices:

- "dealing with people's habits and resistance to change" (R. M. Soley, OMG);
- "accepting and adopting standards" (S. White, IBM);
- "who to ask for guidance and training, and tools availability; difficulty of communication between users, business lines, and other stakeholders" (A. Brown, The Open Group);
- "dealing and managing people" (S. Mellor); and
- "impatience of getting to the results" (T. Jensen, IBM).

These aspects sum up to the well-known fact that: "Learning a new tool or technique actually lowers programmer productivity and product quality initially. You achieve the eventual benefit only after overcoming this learning curve" [29].

Despite the advantages of MDSE, one should not regard it as the panacea for all the problems in the world. Some experts actually have some words of caution about the MDSE approach and what it implies:

- Beware of statements of pure principles: when it comes down to it, the real point of software engineering practice is to increase productivity, reduce errors, and cut code [25]. When MDSE is not perceived as delivering such properties, it is regarded with suspicion.
- Sometimes, the main message of MDSE is perceived as advocating modeling with respect to programming. However, that is not the right way to see it. The real question instead is to understand and define the right abstraction level for addressing each development activity. At the end, this applies also to programming itself.
- In the view of traditional programmers, diagrams are considered, after all, just pretty pictures. As such, they represent more of a burden than an advantage (as for documentation, they need to be kept aligned, discussed, and so on). This opinion is mainly due to unclear understanding of modeling (vs. drawing) and of its advantages, as we will discuss later in the book.
- Models may be perceived as an oversimplification of reality [63]. Indeed, models are an abstraction of reality, where unnecessary details are omitted. However, MDSE practices can be used to combine or refine models to cover all the interesting aspects.

- Models are perceived as useless artifacts that nobody will appreciate, including end users that just look for software that executes and performs well. This is a way to express that models and running applications are perceived as competitors, which is not true at all, especially if you consider the concept of executable models, which can become such through code generation or model-interpretation techniques.
- "To a Computer Scientist, everything looks like a language design problem. Languages and compilers are, in their opinion, the only way to drive an idea into practice" (David Parnas). This is a clear warning on how and when to apply MDSE techniques for language definition. As we will see later when addressing domain-specific languages, we also advise prudence in embarking in a language definition task.