Mashups have emerged as an innovative software trend that re-interprets existing Web building blocks and leverages the composition of individual components in novel, value-adding ways. Additional appeal also derives from their potential to turn non-programmers into developers.

Daniel and Matera have written the first comprehensive reference work for mashups. They systematically cover the main concepts and techniques underlying mashup design and development, the synergies among the models involved at different levels of abstraction, and the way models materialize into composition paradigms and architectures of corresponding development tools.

The book deliberately takes a balanced approach, combining a scientific perspective on the topic with an in-depth view on relevant technologies. To this end, the first part of the book introduces the theoretical and technological foundations for designing and developing mashups, as well as for designing tools that can aid mashup development.

The second part then focuses more specifically on various aspects of mashups. It discusses a set of core component technologies, core approaches, and architectural patterns, with a particular emphasis on tool-aided mashup development exploiting model-driven architectures. Development processes for mashups are also discussed, and special attention is paid to composition paradigms for the end-user development of mashups and quality issues.

Overall, the book is of interest to a wide range of readers. Students, lecturers, and researchers will find a comprehensive overview of core concepts and technological foundations for mashup implementation and composition. Even without low-level coding details, practitioners like software architects will find guidance on key implementation concepts, architectural patterns, and development tools and approaches. A related website provides additional teaching material which can be used either as part of a course or for self study.

This book is timely, provides a thorough scientific investigation and also has practical relevance in the general area of composition and mashups. It is of particular interest to researchers and professionals wishing to learn about relevant concepts and techniques in service mashups, composition, and end-user programming.

From the Preface by Boualem Benatallah, University of New South Wales, Sydney
4.2 Model-Driven Design

The context of model-based developed, i.e., conceptual modeling. We feel it is important to introduce the respective ideas and concepts, in order to clarify its differences and commonalities with MDSD and the respective terminology.

4.2.1 Conceptual modeling

Conceptual modeling has its roots in databases [27]. Before implementing a database, it is common practice today to first design one or more graphical schemas (a synonym of "model" in the database community) of the database structure to be implemented. As illustrated in Figure 4.1, there are typically three different levels of abstraction in database design, in addition to the final code that implements the models in the chosen database management system:

- **Conceptual schema**: This schema expresses the concepts, i.e., the entities, that the database will describe. The conceptual schema relates entities via relationships, generalizations/specializations, and attributes. It is typically expressed as Entity-Relationship (ER) diagram.

- **Logical schema**: This schema expresses the concepts of the conceptual model in terms of concepts that are compatible with the target database technology. For instance, a relational database does not allow the use of m:n relationships, a construct of the conceptual schema; such relationships must therefore be translated into two 1:n relationships and a bridge table, in order for the model to be compatible with the relational data model. The logical schema is typically expressed as ER diagram with only a subset of constructs (e.g., without generalizations or m:n relationships).

- **Physical schema**: This schema eventually expresses the database structure in terms of the specific database technology and system chosen. For instance, the physical schema tells how to actually store data into files, how to index them, and similar. The physical model can, for example, be expressed by drawing concrete tables, keys, and similar.

**Fig. 4.1** The different schemas in the conceptual modeling stack for database design.
4.2 Model-Driven Design

- XMI: All MOF-compliant models can be serialized and stored in the XML Metadata Interchange (XMI) format, an XML mapping for the MOF. XMI enables interoperability among MDA tools like editors and code generators.

- Models at different levels of abstraction: Similarly to the conceptual modeling approach, also MDA is based on three/four core models (see Figure 4.2), which in the case of MDA are all MOF-based and typically specified via dedicated UML profiles:
  - Computation-independent model (CIM): This model captures the domain or business knowledge to be managed by the application under development, typically in the form of a UML class diagram (similar in expressive power to the ER diagram of a conceptual schema). It represents the static knowledge about the domain in a technology- and computation-independent fashion. The CIM is optional in MDSD.
  - Platform-independent model (PIM): This model describes the application in terms of architectural styles, software components, relationships, and similar. It describes a solution in a technology-independent but computation-specific way, so that it is still portable among different platforms with similar architectural and computing assumptions.
  - Platform description model (PDM): This model describes the details of the target platform that enables the implementation of the modeled application. This model may come in a variety of different forms (from formal models to informal documentation and manuals); the final vision is to describe also the PDM in a MOF-compliant fashion.
  - Platform-specific model (PSM): Given a PDM, it is possible to map the PIM into a PSM, i.e., a model that expresses the PIM in terms of the concrete platform chosen for the implementation of the application. The level of abstraction of the PSM is not standardized and may vary from concrete implementations that can already be executed to lower-level PIMs that need further transformation.

- Multi-stage transformations and action languages: MDA fosters the use of subsequent model-to-model transformations (e.g., from PIM to PSM), model-to-code transformations, model markings to guide the transformation process, and action languages that are similar to model-based processes.

- Figure 4.2 Models and transformations in the Model-Driven Architecture (MDA) [195].
Fig. 4.3 The ingredients of architecture-centric model-driven software development (AC-MDSD) according to Stahl and Völter [256].
4.3 Metamodeling

The centerpiece of MDSD are the models that are used to design applications in a graphical manner. Metamodeling is the activity that is concerned with the design of the modeling languages that actually enable the abstract development approach that characterizes MDSD. Good modeling languages contain fundamental conceptual, domain, and technological knowledge regarding the development of their target applications and represent the core value of MDSD. Without sensibly and purposefully designed modeling languages, MDSD would not be useful. It is therefore of utmost importance that developers put the necessary effort – and competence – into the design of their modeling languages, especially if we consider that modeling languages typically do not change fast over time and are designed to support the development of multiple applications on top of a same platform infrastructure.

4.3.1 The metalevels

A model, e.g., a UML object diagram, describes the structure and nature of instances, e.g., runtime objects for a given instant of time during the execution of an application. Similarly, a metamodel describes the structure and nature of model elements, i.e., model constructs. The prefix “meta” indicates that we are dealing with models about models. That is, the term is relative, i.e., referring to the model the metamodel is talking about, not absolute.

In Figure 4.4, we show the four metalevels introduced by the OMG, denoted M0, M1, M2, and M3. The metalevel M0 corresponds to concrete runtime instances of an application; the metalevel M1 to the model of the application; the metalevel M2 to the model of the model (the metamodel), i.e., to the meta-metamodel; and the metalevel M3 to the meta-metamodel itself.

Fig. 4.4 The four metalevels proposed in OMG’s Meta Object Facility [215].
Fig. 4.5 Meta levels vs. abstraction levels. Abstraction levels express different abstractions of a same artifact (the application) at metalevel M1; metalevels express different artifacts (instances, model constructs, metamodel constructs).
4.3.2 Metamodels, MOF and UML profiles

For the concrete specification of a metamodel, there are multiple techniques that can be used, depending on the desired interoperability and the expected importance of the metamodel to be developed: generic metamodels, UML inheritance, MOF-based metamodels, and UML profiles.

4.3.2.1 Generic metamodel

In Figure 4.6 we show a generic metamodel for a finite state machine (FSM) that reacts to the reception of simple characters in input, causing the FSM to transition from one state to another without producing any own output in response. We express the metamodel via a common UML class diagram at the metalevel M2, yet other formalisms could also be used to express the same information (e.g., the BNF). According to the metamodel, a FSM comprises one or more states and a set of transitions with a source and a target state and an input that triggers them. Two special states, a start state and an end state, further denote the starting and accepting states of the FSM.

Fig. 4.6 A platform-independent M2 metamodel for a finite state machine with start and end states. The FSM triggers its transitions upon the reception of an input character.
4.3 Metamodeling

That the resulting modeling language would in principle have the whole expressive power of UML, if the developer does not restrict himself (or the modeling tool does so) to using only those constructs of the extended UML metamodel that indeed refers to FMSs only. That is, specializing the UML metamodel means inheriting the whole complexity of UML. The benefit of doing so is a MOF-compliant metamodel and UML tool support.

Independently of which of the two options we choose, it is important to note that the metamodel contains both modeling constructs and constraints. In Figure 4.6 we simply use natural language to state that, in order for a FSM to be correct, it must contain at least one start state and at least one end state. In the context of UML, the more formal version of this statement would be expressed in the Object Constraint Language (OCL) [212], which can also be evaluated by advanced UML MDSD tools to check the expressed properties. The constructs of the metamodel together with its constraints define the so-called static semantics of the model.

4.3.2.2 MOF-based metamodel

Instead of specializing the UML metamodel, we can obtain a MOF-compliant metamodel for our modeling language also if we directly use the MOF to specify the metamodel. That is, we neglect the whole metamodel of UML and define our very own metamodel in terms of instances of MOF constructs.

Figure 4.7 depicts a simplified version (without attributes and detailed relationship descriptions) of the Essential MOF (EMOF) model [215], the core of the MOF and the starting point for MOF-compliant metamodel specifications.

---

**Fig. 4.7** A simplified version of the EMOF package, the core of the MOF [215].
Fig. 4.8 A metamodel for a FSM defined as instance of the Meta Object Facility (MOF) [215] with some <<instance of>> relationships highlighted: states and transitions are instances of MOF::Class, relationships of MOF::Association, and attributes of MOF::Property.
4.3 Metamodeling

Profiles are based on stereotypes, tagged values, and constraints:

• **Stereotypes**: Stereotypes are UML’s mechanism to extend the basic vocabulary of UML with elements with custom meaning and properties, starting from the core elements of UML itself (e.g., UML::Class). For instance, in Figure 4.9 we define stereotypes (graphically represented by the <<stereotype>> adornment) for states and transitions as extensions of the UML::Class concept. The figure uses the extension relationship introduced in UML 2.0 for the definition of stereotypes [217].

• **Tagged values**: A stereotype can have attributes to define additional properties for new vocabulary elements. Being conceptually expressed at metalevel M2, these attributes do not refer to attributes of runtime instances, but to attributes or properties of the modeling constructs at the M1 level, i.e., metadata from the point of view of the model. There are different ways of rendering tagged values in an M1 model instance, e.g., inside comments attached to the respective modeling element. The purpose of tagged values is typically that of providing the code generator with additional instructions on how to generate code or transform models or to provide means for configuration management.

• **Constraints**: As for all metamodels in general, also profiles allow the use of OCL for the specification of static constraints further refining the possible use of the modeling constructs at metalevel M1.

![Fig. 4.9 A metamodel for a FSM defined as UML 2.0 profile. Syntactically, UML profiles are at the metalevel M1, yet semantically they are at metalevel M2, as profiles specialize the UML metamodel (e.g., UML::Class).](image-url)
Fig. 4.10 A simplified excerpt of the Classes diagram of the Constructs package of the UML metamodel [217], the starting point for the definition of UML profiles.
regarding the serialization of models; however, XML-based formats typically facilitate the last step in the MDSD process, i.e., code generation, which we talk about in the next section.

4.3.3.1 Abstract syntax

The first approach to assign a syntax to a metamodel like the one of our FSM, is to use a UML object diagram, which allows the representation of instances of metamodel elements. For example, in Figure 4.11 we model a state machine with four states (S1-S4) and four transitions (T1-T4), respectively triggered by the inputs “a”, “b”, “c”, and “d”. S1 is a start state; S4 an end state, S2 and S3 are intermediate states.

Fig. 4.11 A model instance of the finite state machine defined in Figure 4.6 using a UML object diagram as modeling syntax.
Fig. 4.12 A model instance of the finite state machine defined in Figure 4.6 using an own, more intuitive modeling syntax.
Depending on the platform features and the modeled application, either configuration files or code or both are generated; individual code may be plugged in manually. As a consequence, generating code from a model is not as naive as to generate the complete code of the application specified in a model from scratch. A large part of its functionalities is already available as reusable components inside the target platform, and the code generation process only needs to provide these components with suitable configuration settings and/or to generate code that makes use of them.

As illustrated in Figure 4.13, there is generally however an additional kind of artifact that requires our attention: individual code, i.e., custom code and functionality, which can be programmed manually and plugged into the application to be generated. Especially in the context of AC-MDSD, where the focus of the modeling effort is on the careful selection and configuration of readily available architectural elements, it is not given for granted that all application capabilities expressed in the model are already supported by the underlying platform. In order to overcome possible lacks of functionality, both the platform and the model must support suitable extensibility mechanisms, which allow the developer to plug in custom application logic. The final vision of MDA with its support for executable UML, instead, would allow the developer to specify any kind of application logic in the model and to generate the respective code from it, practically turning the model into code. Yet, this vision is not yet reality, and we are not sure it will ever become.
We summarize the key concepts as illustrated in Figure 4.14. A model is an instance of a metamodel, which, in turn, is an instance of a meta-metamodel. That is, the meta-metamodel tells how to construct a metamodel; the metamodel tells how to construct a model. The metamodel assumes a key role in the MDSD process, in that it is the basis for the design of the modeling language used to develop models. It has an abstract syntax and static semantics. The abstract syntax specifies how the modeling language's structure looks like. The modeling language has a concrete syntax, which realizes the abstract syntax and provides it with concrete constructs (e.g., XML elements, graphical constructs, textual instructions, or similar). The concrete syntax is what the parser of a model transformer or interpreter reads, the abstract syntax expresses how they structure the parsed model internally (in memory). The static semantics of the metamodel defines the requirements for the well-formedness of models (e.g., that a content unit construct can only occur inside a page construct). The dynamic semantics of the modeling language defines how models are processed and eventually turned into applications, a logic the developer must intimately know in order to be able to correctly express a target application via a model.

We discussed MDSD in this chapter, since MDSD is the basis for many mashup tools and platforms and discussing them, first of all, requires the necessary understanding of MDSD principles. We have seen that a well-designed MDSD approach comes with the power of alleviating the developer from some development tasks (typically, the most repetitive and annoying ones) and speeding up development. However, MDSD does not come without problems.

Fig. 4.14 Summary of the key concepts of MDSD.