Title: Sans titre
Author:

CHAPTER 6 Modeling Languages at a Glance

Modeling languages are conceptual tools that let designers formalize their thoughts and visualize reality explicitly, whether textually or graphically. This chapter describes the main features of modeling languages, considering general-purpose languages (GPLs), domain-specific languages (DSLs), and the intermediate solutions that customize GPLs for specific purposes.

Résumé :

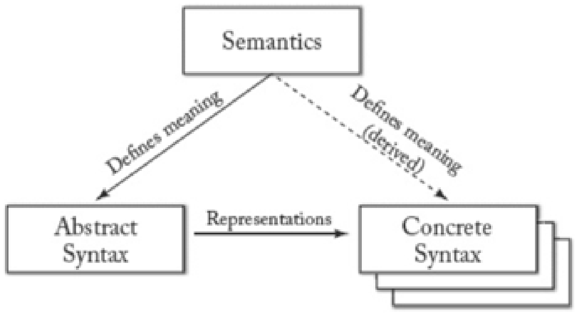
- Modeling languages formalize thoughts and visualize reality, covering general-purpose languages (GPLs), domain-specific languages (DSLs), and the intermediate solutions that customize GPLs.

6.1 ANATOMY OF MODELING LANGUAGES

A modeling language is defined by three core ingredients:

- *Abstract syntax*: Describes the language's structure and how primitives can be combined, regardless of representation.
- *Concrete syntax*: Describes specific representations, including encoding and visual appearance. It can be textual or graphical and is a reference for designers during modeling.
- *Semantics*: Describes the meaning of language elements and combinations.

Modeling languages require three mandatory ingredients: semantics, abstract syntax, and concrete syntax. Figure 6.1 illustrates their relationships. Semantics defines abstract syntax, which indirectly defines concrete syntax; the concrete syntax represents the abstract syntax.



This applies to both general-purpose languages (GPLs) and domain-specific ones (DSLs). Unfortunately, language designers and users often neglect aspects, especially semantics. This is likely due to the concrete syntax's visibility, representing the actual notation used in everyday applications.

However, defining a language without fully specifying its conceptual elements and meanings is illogical. A partial or incorrect semantics leads to incorrect language usage, misunderstandings, and differing interpretations of concepts and models. This can cause confusion and errors.

Modeling languages describe the real world at a specific level of abstraction and from a particular perspective. Their semantics, which aims to describe this in detail, enables correct language usage. Semantics can be defined as:

- *Denotational*: Defining the meaning of concepts, properties, relationships, and constraints through mathematical expressions.
- *Operational*: Defining the language's meaning by implementing an interpreter that directly defines model behavior.
- *Translational*: Mapping language concepts to another language with clearly defined semantics.

Résumé :

- A modeling language consists of abstract syntax, concrete syntax, and semantics.
- Modeling languages require semantics, abstract syntax, and concrete syntax. Semantics defines abstract syntax, which indirectly defines concrete syntax; the concrete syntax represents the abstract syntax.
- Language designers and users often neglect semantics, focusing instead on concrete syntax.
- Defining a language without specifying semantics leads to incorrect usage and

misunderstandings.
- Modeling languages describe the real world at a specific abstraction level. Their semantics details this, enabling correct language usage, which can be denotational (using mathematical expressions), operational (through an interpreter), or translational (mapping to another language).

6.2 MULTI-VIEW MODELING AND LANGUAGE EXTENSIBILITY

In software engineering, you typically model orthogonal system aspects using different modeling languages or a multi-viewpoint modeling language with various diagram types.

System aspects are classified into *static* (or *structural)* and dynamic aspects. Static aspects describe the main ingredients and their relations, while *dynamic* aspects describe their behavior in terms of actions, events, and interactions. Separating static and dynamic aspects is usually a good practice.

Furthermore, languages often include extensibility mechanisms that allow designers to expand coverage by defining new modeling elements. This is common in GPLs, which often need specialization for specific domains or problems. This can lead to a well-known domain-specific language from general-purpose ones.

UML, a general-purpose language proposed by OMG within the MDA framework, comprises various diagrams for system description and extension mechanisms. Several extensions have become standard languages like SysML and SoaML. Thus, UML can be considered a family of languages in three senses: it allows enough variability to be considered a group of languages, pairs with correlated languages like OCL to enhance its expressive power, and is accompanied by domain-specific profiles.

Résumé :

- Software engineering models orthogonal system aspects using different modeling languages or multi-viewpoint modeling languages.
- System aspects are classified into static (or structural) and dynamic aspects. Static aspects describe ingredients and relations, while dynamic aspects describe behavior. Separating these aspects is beneficial.

- Languages often include extensibility mechanisms for defining new modeling elements, especially in GPLs for domain-specific specialization, can lead to the creation of domain-specific languages.
- UML, a general-purpose language, is a family of languages due to its variability, correlated languages like OCL, and domain-specific profiles.

6.3 GENERAL-PURPOSE VS. DOMAIN-SPECIFIC MODELING LANGUAGES

As mentioned in Chapter 1, modeling languages are classified as domain-specific or general-purpose.

Domain-Specific Modeling Languages (DSMLs, or DSLs for short) are languages designed for specific domains, contexts, or companies to support people describing things in those domains.

General-Purpose Modeling Languages (GPMLs, or GMLs/GPLs) are modeling notations applicable to any sector or domain.

However, this distinction is not so deterministic and well-defined. Deciding whether a language is a DSL or GPL is not a binary choice. For instance, UML and similar languages seem to belong to the latter group, since they can model any domain. However, if we consider modeling in general, UML is a DSL tailored to the specification of (mainly object-oriented) software systems.

The decision on whether to use UML (or any other GPL) or a DSL for a modeling project is significant. While UML may not be the best option for certain domains (like user interaction design), a domain-specific language tailored to that domain can produce better results. However, discussing DSL vs. UML without acknowledging UML's many-domains language (MDL) nature is misleading. UML may not be suitable for all domains, but it can be easily applied to model many of them successfully.

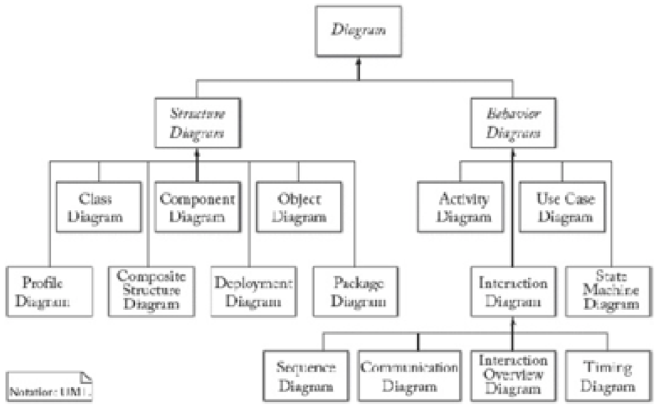
UML may have flaws, but we're aware of them. Creating an unnecessary DSL may repeat UML's mistakes. Avoid reinventing the wheel. If your DSL resembles UML, consider using a subset of UML and avoiding new "almost-UML" notations.

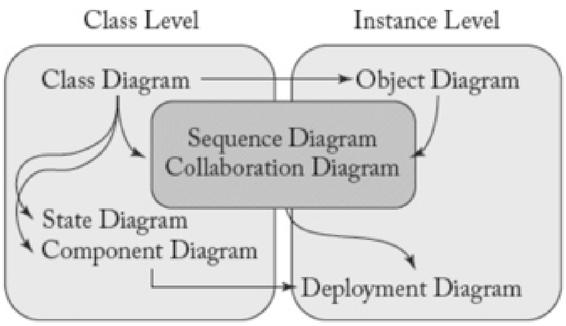
On the theoretical side, concerning GPLs, one may wonder if a language to be a GPL must also be Turing-complete. If not, it means it’s not truly "general," since it can’t solve every problem. However, this interpretation is rarely considered, and the definition of a language as GPL or DSL is more a matter of practice or subjectivity. Martin Fowler clarifies that even deciding whether a set of concepts (or operations) is a language or just a set of operations within another language is a matter of perspective.

> **Résumé** :
>
> - Modeling languages are classified as domain-specific or general-purpose.
> - Domain-Specific Modeling Languages (DSMLs, or DSLs for short) are designed for specific domains to support people describing things in those domains.
> - General-Purpose Modeling Languages (GPMLs, or GMLs/GPLs) are applicable to any sector or domain.
> - The distinction between DSL and GPL is not binary, as seen with UML, which can model any domain but is tailored to software systems.
> - UML, while not suitable for all domains, can be effectively applied to model many domains successfully.
> - Avoid reinventing the wheel by using a subset of UML instead of creating a new DSL.
> - The definition of a language as GPL or DSL is subjective, with the distinction often based on practical considerations rather than strict theoretical criteria.

6.4 GENERAL-PURPOSE MODELING: THE CASE OF UML

This section provides an overview of the *Unified Modeling Language (UML)*. UML is widely known and adopted, making it an interesting example for discussing general characteristics of modeling languages. It’s a full-fledged language suite with various diagrams for describing a system from different perspectives. Figure 6.2 shows the taxonomy of UML diagrams, with 7 for static aspects and 7 for dynamic aspects. As shown in Figure 6.3, some diagrams describe class characteristics, while others describe instance features and behavior. Some diagrams can describe both levels.





UML is a modeling language that doesn't enforce a specific development method. It's often mentioned with the Unified Process (UP), particularly its specialization, Rational Unified Process (RUP). RUP is an iterative software development process framework created by Rational Software Corporation (now IBM). While RUP and UML work well together, adopting RUP isn't necessary for using UML.

UML, a unifying language for software modeling, has a long history of merges and restructuring. Some pieces, like Harel's statecharts and Booch's notations, date back to the mid-1980s when various notations existed. The first version of UML emerged in 1996 from the fusion of Booch, Rumbaugh, and Jacobson's approaches. Discussed at OOPSLA, it was submitted to OMG in August 1997 and released as UML 1.1 in November 1997. Despite criticisms, UML remains a complex and inconsistent language, yet remains a crucial tool for software modeling.

Résumé :

- UML is a widely adopted modeling language suite with various diagrams for describing systems. It includes 14 diagrams, seven for static and seven for dynamic aspects, covering class characteristics, instance features, and behavior.
- UML is a modeling language that can be used independently of the Unified Process

> (UP) or its specialization, Rational Unified Process (RUP).
> - UML, a software modeling language, emerged in 1996 from the fusion of Booch, Rumbaugh, and Jacobson's approaches. Despite criticisms, it remains a crucial tool for software modeling.

6.4.1 DESIGN PRACTICES

UML facilitates system design and promotes good design practices. It offers several features:

- Integrated and orthogonal models: UML's diagrams share symbols and allow cross-referencing between modeling artifacts.
- Modeling at different levels of detail: UML allows designers to choose the right amount of information to include in diagrams based on the purpose and phase of development.
- Extensibility: UML provides features for designing customized modeling languages.
- Pattern-based design: UML includes well-known design patterns defined by the Gang of Four.

The following sections describe a few details about UML modeling to provide an overview of a typical GPL.

> **Résumé** :
>
> - UML facilitates system design with integrated and orthogonal models, modeling at different levels of detail, extensibility (customized modeling languages), and pattern-based design.
> - UML modeling details are provided to overview a typical GPL.

6.4.2 STRUCTURE DIAGRAMS (OR STATIC DIAGRAMS)

Structure diagrams describe the elements in a system being modeled. They're used extensively in documenting software systems at two main levels.

- The system's *conceptual items* of interest. This level of design describes the domain and system in terms of concepts and their associations. Typical diagrams describe this part.
    i. *Class diagram*: Describes a system's structure using classes, attributes, and relationships. Classes can be described at different levels of detail.
    ii. *Composite structure diagram*: Describes the internal structure of a class and its collaborations. They use a syntax similar to class diagrams but exploit containment

and connections between items. The core concepts are *parts* (i.e., roles played at runtime by instances), *ports* (i.e., interaction points connecting classifiers), and *connectors* (undirected edges connecting entities through ports).

iii. *Object diagram*: A view of the structure of example instances of modeled concepts at a specific point in time, possibly including property values. Similar to class diagrams, objects are identified by underlined names. They can also be identified by the tuple object name and class name, the simple object name, or the class they're instantiated from. The respective notations are as follows: objectName:ClassName or objectName or :ClassName

- The *architectural representation* of the system. This level of design describes the system's architectural organization and structure. It typically consists of reuse and deployment-oriented information, based on the conceptual modeling done in the previous step. The diagrams in this part of the design include:
  i. *Component diagram*: Describes how a software system is divided into components and their dependencies. In UML, a component is a distributable piece of implementation, including software code and other information, that implements a subsystem.
  ii. *Package diagram*: Describes how a system is divided into logical groupings called packages in UML, along with their dependencies.
  iii. *Deployment diagram*: Describes how software artifacts are deployed on hardware in systems' implementations.

Résumé :

- Structure diagrams describe system elements, used extensively in software documentation.
- The system's conceptual items are described using class, composite structure, and object diagrams. These diagrams illustrate the system's structure, internal class composition, and example instances of modeled concepts.
- The architectural design level describes the system's organization and structure, including component, package, and deployment diagrams.

6.4.3 BEHAVIOR DIAGRAMS (OR DYNAMIC DIAGRAMS)

Behavior diagrams describe system events and interactions. Different diagrams describe dynamic aspects, some equivalent in information. Designers choose graphical notation based on prominent aspects. Behavioral models with these diagrams usually describe a single or few system features and their dynamic interactions. The behavioral diagrams are the following:

- *Use case diagram*: Describes a system's functionality in terms of external actors, their goals, and dependencies. They help understand the system's boundaries and clarify usage scenarios. They're especially relevant in the requirements and early design phases.
- *Activity diagram*: Describes the step-by-step workflows of activities to reach a goal. It shows the overall flow of data and control through an oriented graph, with nodes representing the *activities*.
- *State machine diagram* (or *statechart*): Describes the states and state transitions of a system, subsystem, or object. They're suitable for describing event-driven, discrete behavior, but not continuous behavior. In statecharts, nodes represent states (not actions), unlike in activity diagrams.
- *Interaction diagrams*: A subset of behavior diagrams focus on the flow of control and data among system elements. They include the following diagrams.
    i. *Sequence diagram*: Shows how objects communicate through a temporal sequence of messages. Messages are sequenced vertically on a timeline, and the lifespan of associated objects is reported. Interaction diagrams describe system execution scenarios.
    ii. *Communication or collaboration diagram*: Shows the interactions between objects or classes using solid, undirected lines connecting elements that can interact. *Messages* flow through these *links* (solid, undirected lines connecting interacting elements). Sequencing information is obtained by numbering messages. They describe both some static structure (links and nodes) and dynamic behavior (messages) of the system, combining information from class, sequence, and use case diagrams.
    iii. *Interaction overview diagram*: Shows interaction diagrams represented by nodes.
    iv. *Timing diagrams*: Interaction diagrams that focus on timing constraints.

Résumé :

- Behavior diagrams describe system events and interactions, with different diagrams focusing on dynamic aspects. Use case, activity, state machine, and interaction diagrams are used to describe system functionality, workflows, states, and interactions, respectively.

6.4.4 UML TOOLS

Given UML’s popularity, many UML modeling tools are available. They’re easily distinguishable from other MDD tools because they focus solely on UML design, with limited support for various MDE scenarios. For instance, they differ from metamodeling tools, which focus on designing new modeling languages.

UML tools create UML models, import/export them in XMI format, and sometimes provide partial code generation. The market offers various UML tool licenses, from open source to freeware to commercial. Wikipedia has a comprehensive list of UML tools with their main features.

> **Résumé** :
>
> - UML modeling tools, popular due to UML’s prevalence, differ from metamodeling tools by focusing solely on UML design with limited support for other MDE scenarios.
> - UML tools create models, import/export them in XMI format, and sometimes generate code. Wikipedia lists UML tools with features.

6.4.5 CRITICISMS AND EVOLUTION OF UML

UML has been criticized for being verbose, cumbersome, incoherent, and unusable in domain-specific scenarios. These criticisms, as reported in Bell’s 2004 article "Death by UML Fever," are prevalent online. While some criticisms may be partially true, UML remains the reference design language for software engineers. OMG has taken these criticisms seriously and is simplifying the UML specification.

> **Résumé** :
>
> - UML, despite criticisms of verbosity and complexity, remains the reference design language for software engineers. OMG is simplifying the UML specification in response to these criticisms.

6.5 UML EXTENSIBILITY: THE MIDDLE WAY BETWEEN GPL AND DSL

If the development requires unusual modeling, consider using a domain-specific language. Instead of a new DSL, extend the existing GPL as an intermediate solution to fit the needs. For example, we show UML's extensibility features.

UML offers various extension features: stereotypes, constraints, tagged values, and profiles. The *Profile diagram*, focusing on language extensibility, shows stereotypes as classes and profiles as packages at the metamodel level. The extension relationship indicates the metamodel element a stereotype extends. This section provides an overview of these extensibility options.

> **Résumé** :
>
> - Consider using a domain-specific language for unusual modeling, or extending an existing GPL.
> - UML offers extensibility through stereotypes, constraints, tagged values, and profiles. The Profile diagram shows stereotypes as classes and profiles as packages.

6.5.1 STEREOTYPES

Stereotypes extend meta-classes by defining additional semantics for their concept. A model element can be stereotyped multiple ways. To define a stereotype, specify the following properties:

- *Base metaclasses*, defining the elements to be extended.
- *Constraints*, defining the special rules and semantics that apply to this specialization, characterizing the stereotype in terms of semantics.
- *Tagging values*, defining zero or more values the stereotype may need for proper functioning.
- *Icon*, defining the visual appearance of stereotyped elements in the diagrams.

One doubt may arise: why not use standard sub-classing instead of stereotyping? Subclassing defines a special relation between items in the same model, while stereotypes define new modeling concepts used to create models. Stereotypes are typically used to define (i) additional semantic constraints, (ii) additional semantics outside UML's scope (e.g., metadata for code generators), or (iii) new modeling concepts with specific behavior or properties that will often be reused.

> **Résumé** :
>
> - Stereotypes extend metaclasses by defining additional semantics for their concept. A model element can be stereotyped multiple ways, specifying base metaclasses, constraints, tagging values, and an icon.
> - Stereotypes define new modeling concepts, while subclassing defines a special relation between items in the same model. Stereotypes are used for additional semantic constraints, additional semantics outside UML's scope, and reusable new modeling concepts.

6.5.2 PREDICATES

Another way to vary UML semantics is to use *restriction predicates* (e.g., OCL expressions) that reduce semantic variation of modeling elements. Predicates can be attached to any UML meta-class or stereotype and can be formal or informal expressions. To be coherent with UML symbols, expressions must not contradict the inherited base semantics of elements.

> **Résumé** :
>
> - Restriction predicates, such as OCL expressions, can be used to vary UML semantics by reducing the semantic variation of modeling elements. These predicates can be attached to any UML meta-class or stereotype and can be formal or informal expressions.

6.5.3 TAGGED VALUES

Tagged values consist of a tag-value pair that can be attached as a typed property to a stereotype. They allow designers to specify additional information useful for implementing, transforming, or executing a model, such as defining project management data. For example, a tagged value "status = unit_tested" can declare that a component has gone through the unit testing phase.

> **Résumé** :
>
> - Tagged values are typed properties attached to stereotypes, providing additional

information for model implementation, transformation, or execution.

6.5.4 UML PROFILING

UML profiles are packages of related and coherent extensibility elements defined with the above techniques. They capture domain-specific variations and usage patterns for the language. In essence, profiles are domain-specific interpretations of UML, akin to domain-specific languages defined by extending or restricting UML.

Designers can define their profiles, but several UML profiles are standardized and widely used. Examples include well-known profiles that became standards in OMG (mentioned in Chapter 4 without highlighting they were UML profiles).

- UML Testing Profile (UTP): A UML profile that extends UML to support designing, visualizing, specifying, analyzing, constructing, and documenting testing artifacts.
- OMG Systems Modeling Language (SysML).
- Service-oriented architecture Modeling Language (SoaML).
- UML profile for a System on a Chip (SoCP).
- UML Profile for Schedulability, Performance, and Time.
- UML Profile for Modeling and Analyzing Real-time and Embedded Systems (MARTE).

The complete list is available online on the OMG website.

Résumé :

- UML profiles are domain-specific interpretations of UML, capturing domain-specific variations and usage patterns for the language.
- Several standardized UML profiles, including UTP, SysML, SoaML, SoCP, and MARTE, extend UML to support various modeling needs.
- The complete list is available online.

6.6 OVERVIEW ON DSLS

Domain-Specific Modeling Languages (DSMLs), also known as Domain-Specific Languages (DSLs), are designed to address the needs of a specific application domain. They are tailored to the domain's requirements, both in terms of expressive power and notation, making them particularly useful.

DSLs don’t force users to learn more general-purpose languages with irrelevant concepts. Instead, they provide appropriate modeling abstractions and primitives closer to the domain.

Résumé :

- Domain-Specific Modeling Languages (DSMLs), also known as Domain-Specific Languages (DSLs), are tailored to specific application domains, enhancing their usefulness.
- DSLs simplify domain-specific modeling by providing relevant abstractions and primitives.

6.6.1 PRINCIPLES OF DSLS

DSLs are essential for many application scenarios, beyond software development. Designing a DSL requires deep domain knowledge and language engineering practices. Finding the right abstractions for defining a DSL is challenging and time-consuming.

To be useful, the DSL should follow these principles:

- The language should provide good abstractions, be intuitive, and simplify development, not complicate it.
- The language should be adopted and used by everyone, and its definition should be agreed upon after some evaluation, not just by one person.
- The language must evolve and be updated based on user and context needs to avoid extinction.
- Domain experts prioritize maximizing productivity in their domains, but they’re unwilling to invest significant time in defining methods and tools. Therefore, the language must be combined with supporting tools and methods.
- A good DSL should be open for extensions but closed for modifications, following the open-close principle that "software entities should be open for extension, but closed for modification."

Good DSLs don’t appear out of nowhere. MDSE aims to have good language designers who build suitable DSLs for the right audience. Refining a good language takes years, involving industrial experience and thousands of users. While DSL development is encouraged, careful design is essential. Defining a new DSL is not trivial and may lead to a language with weaknesses or drawbacks for future users. Tools, design interfaces, and runtime architectures should be tailored to the domain to avoid user rejection.

> **Résumé** :
>
> - DSLs require domain knowledge and language engineering practices for design.
> - A useful DSL should provide good abstractions, be intuitive, and evolve based on user needs. It should be adopted by all, combined with supporting tools, and follow the open-close principle.
> - Good DSLs require careful design, industrial experience, and thousands of users over years. Defining a new DSL is not trivial and should be tailored to the domain.

6.6.2 CLASSIFICATION OF DSLS

DSLs can be classified based on focus, style, notation, internality, and execution.

Focus

DSLs can be vertical or horizontal. Vertical DSLs target specific industries or fields, like configuration languages for home automation, modeling languages for biological experiments, and analysis languages for financial applications. Horizontal DSLs have broader applicability and can apply to many applications. Examples include SQL, Flex, IFML, WebML, and more.

Style

DSLs can be declarative or imperative. Declarative DSLs express computation logic without describing control flow. They define what a program should accomplish, not how. Service choreography is a typical declarative definition, defining Web service coupling rules. Imperative DSLs require defining an executable algorithm with steps and control flow to complete a job. Service orchestration is a typical imperative definition, defining a start-to-end flow of execution between Web services.

Notation

DSLs can be graphical or textual. Graphical DSLs use visual models and graphical development primitives like blocks, arrows, edges, containers, and symbols. Textual DSLs include XML-based notations, structured text notations, and textual configuration files.

Internality

External DSLs, as defined by Martin Fowler, have their own custom syntax. You can write a full parser for them and use them to create self-standing, independent models and programs.

Internal DSLs extend a host language to mimic a specific domain or objective. This can be achieved by embedding DSL pieces in the host language or providing abstractions, structures, or functions. Embedding allows reuse of host language tools and facilities.

Execution

Executability can be implemented through model interpretation (reading and executing DSL scripts at runtime) or code generation (applying a complete model-to-text (M2T) transformation at deployment time).

> **Résumé** :
>
> - DSLs are classified based on focus, style, notation, internality, and execution.
>   - **Focus** : DSLs can be vertical, targeting specific industries, or horizontal, with broader applicability. Examples include SQL, Flex, IFML, and WebML.
>   - **Style**: DSLs can be declarative, defining what a program should accomplish, or imperative, defining an executable algorithm with steps and control flow.
>   - **Notation**: DSLs can be graphical or textual, using visual models or structured text.
>   - **Internality**: Internal DSLs extend a host language to mimic a domain, while external DSLs have their own custom syntax and can be used independently.
>   - **Execution**: Executability can be implemented through model interpretation or code generation.

6.7 DEFINING MODELING CONSTRAINTS (OCL)

Modeling and metamodeling languages are usually small, with limited constructs for defining new languages.

This facilitates their practical use (learning curve is not steep since language designers don’t need to learn many concepts), but it has a price. Metamodeling languages can only express a limited subset of relevant information for defining a modeling language. They only support defining basic modeling constraints, such as cardinality constraints that restrict possible links between modeling elements.

Object Constraint Language (OCL) is a general-purpose, textual formal language adopted as a standard by the OMG. It's a *typed*, *declarative*, and *side effect-free specification* language. Each OCL expression has a type, evaluates to a value of that type, and must conform to its rules and operations. OCL expressions can query or constrain the system's state but not modify it. It doesn't include imperative constructs and doesn't include implementation details or guidelines.

OCL complements metamodels with textual rules that models conforming to them must follow. These rules, known as well-formedness rules when applied to a metamodel, define the set of well-formed models that can be specified with that specific modeling language.

OCL constraints are invariants defined in a specific type's context. Their condition must be satisfied by all instances of the *context type*. The standard OCL library defines primitive and collection types, along with their operations. Quantifiers (like for *all* and *exists*) and other iterators (*select*, *reject*, *closure*, etc.) are also available. Access to object properties and navigation to related objects is done using the dot notation. An introductory OCL tutorial is at `http://modeling-languages.com/object-constraint-language-ocl-a-definitive-guide`.

Résumé :

- Modeling and metamodeling languages are limited in their constructs for defining new languages.
- Metamodeling languages, while easy to use, have limitations in expressing modeling language information. They only support basic constraints, like cardinality restrictions.
- Object Constraint Language (OCL) is a typed, declarative, and side-effect-free specification language adopted by the OMG. It evaluates to a value of its type and conforms to its rules and operations.
- OCL complements metamodels with textual rules defining well-formed models.
- OCL constraints define invariants for a specific type, ensuring all instances satisfy the condition. The standard OCL library includes primitive and collection types, operations, quantifiers, and iterators.

CHAPTER 7 Developing your Own Modeling Language

Modeling is often misunderstood as just drawing pretty pictures. However, models follow a clearly defined structure, similar to program code, conforming to the associated *metamodel* representing the modeling language's *abstract syntax*. This well-defined structure enables various operations, such as loading/storing models, querying, transforming, and checking well-formedness. In some modeling environments and tools, metamodels are hidden behind the user interface, making them invisible to modelers. Designers may not need a clear understanding of these aspects in simple scenarios, but it's crucial for complex MDE design, especially when building tailored environments. Developing and using a metamodel is crucial because other language aspects, like visual notation, are heavily based on them.

> **Résumé** :
>
> - Modeling follows a structured approach, similar to program code, and is based on metamodels representing the modeling language's abstract syntax. This structure enables various operations and is crucial for complex MDE design.

7.1 METAMODEL-CENTRIC LANGUAGE DESIGN

This chapter develops a new modeling language. We'll define the most important syntactical ingredients, namely the *abstract syntax* and the *concrete syntax* introduced in Chapter 6.

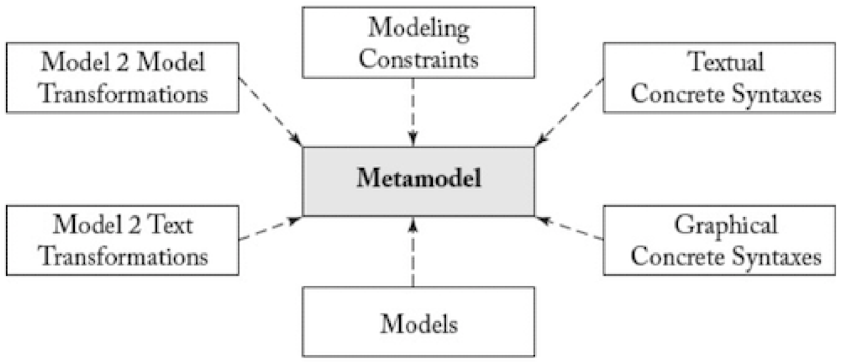
The first ingredient is defining modeling concepts and their properties by defining metamodels, which play a role corresponding to grammars for textual languages. A grammar defines *all valid sentences* of a language, while a metamodel defines valid models of a modeling language. Metamodels are defined with *metamodeling languages*, which are heavily based on the core concepts of structural object-oriented modeling languages, such as *classes*, *attributes*, and *associations*—so to speak, the core concepts of UML class diagrams. The term "metamodeling languages" originates from the fact that these modeling languages are applied for *modeling* modeling languages.

The prefix *meta* in this context implies defining the technique itself. *Metalearning*, for example, means *learning how to learn*, and metamodeling means modeling how to model. The prefix meta can be applied multiple times. *Meta-metamodeling*, for instance, means *modeling how to metamodel*. The prefix *meta* may also be applied several times. Consider the term *meta-metamodeling*, which stands for modeling how to metamodel. The language for defining how to build metamodels is called *meta-metamodel*, a model that represents *all valid metamodels*. However, as models are abstractions, metamodels and meta-metamodels only define the abstract syntaxes of the languages they represent. Concrete syntaxes and semantics are currently not covered by these models and must be specified with additional artifacts.

Metamodels define modeling concepts and their properties using classes, attributes, and associations. However, modeling constraints are only partially described (see Chapter 6). UML class diagrams can define multiplicity constraints for attributes, association ends, and types. A *constraint language* can define more complex validation rules based on these metamodel elements. OCL is the language of choice for defining constraints beyond simple multiplicity and type constraints, and it can also be used for metamodels. Current metamodeling languages based solely on class diagrams cannot express constraints like "*A model element must have a unique name." However, this constraint is easily specifiable in OCL.

Metamodeling frameworks allow specifying metamodels using dedicated editors and generating modeling editors from them for *defining* and *validating* models. Metamodels are used (i) *constructively* as *a set of production rules* for building models and (ii) *analytically* as *a set of constraints* a model must fulfill to conform to its metamodel.

All language aspects beyond abstract syntax are defined on the basis of metamodel, as illustrated in Figure 7.1. This chapter first defines the abstract syntax of a modeling language, including constraints, and then defines graphical and textual concrete syntaxes based on it. We also discuss current technologies available on top of Eclipse for developing these syntactical aspects.


Model 2 Model Transformations
Modeling Constraints
Textual Concrete Syntaxes
Metamodel
Model 2 Text Transformations
Graphical Concrete Syntaxes
Models

Transformation engineering, based on modeling language metamodels, is the subject of the next two chapters (Chapter 8 for model-to-model transformations and Chapter 9 for model-to-text transformations). Transformations formalize the semantics of a modeling language using different approaches, as introduced in Chapter 6: (i) *denotational semantics* by mapping the language to a formal language; (ii) *operational semantics* by defining a model simulator; or (iii) *translational semantics* by defining a code generator for executable code. The first two approaches use model-to-model transformations, while the latter is often implemented as model-to-text transformations.

Résumé :

- This chapter focuses on developing a new modeling language, specifically defining its abstract and concrete syntax.
- Metamodels define valid models of a modeling language, similar to how grammars define valid sentences. Metamodeling languages, based on object-oriented modeling concepts, are used to model modeling languages.
- The prefix "meta" implies defining the technique itself, as seen in metalearning (learning how to learn) and metamodeling (modeling how to model). The prefix can be applied multiple times, as in meta-metamodeling, which models how to metamodel, representing abstract syntaxes but not concrete syntaxes or semantics.
- OCL is the preferred language for defining complex modeling constraints beyond multiplicity and type constraints, as it can express constraints like "A model element must have a unique name."
- Metamodeling frameworks enable model specification and editor generation for defining and validating models. Metamodels are used constructively for building models and analytically for enforcing constraints.
- This chapter defines the abstract syntax of a modeling language, including constraints, and then defines graphical and textual concrete syntaxes based on it.
- Transformation engineering, using modeling language metamodels, formalizes modeling language semantics through denotational, operational, and translational approaches.

7.2 ABSTRACT SYNTAX DEVELOPMENT

In this section, we explain how metamodeling languages are developed using metamodeling as the central technique. The OMG has established the *Meta Object Facility* (MOF) as the standard metamodeling language. Several other languages and tools have been proposed, including the *Eclipse Modeling Framework* (EMF) and its metamodeling language *Ecore*. Both MOF and Ecore are based on a subset of UML class diagrams for describing structural aspects, but Ecore is tailored to Java for implementation purposes. We first use simplified UML class diagrams corresponding to MOF to explain metamodel definition, and then discuss Ecore's peculiarities in Section 7.2.2.

Why not reuse UML for metamodeling as it is? MOF is similar to UML class diagrams but focuses on defining other modeling languages. It removes concepts like n-ary associations, association classes, interfaces, and dependencies available in UML. The main differences lie in their domains. UML is a comprehensive object-oriented modeling language covering structural, behavioral, conceptual, and implementation aspects. In contrast, MOF is a specialized DSML for metamodeling that reuses a subset of UML's core.

Benefits of metamodeling. By using a metamodeling language, the language designer can define the abstract syntax of their own modeling language, whether general or domain-specific. Regardless of the metamodeled language, an explicit metamodel conforming to a standardized meta-metamodel offers several benefits.

- **Precise language definition**: There's a formal definition of the language's syntax that machines can process. For instance, metamodels can check if models are valid instances.
- **Accessible language definition**: UML class diagrams, a well-known modeling language, are used to define metamodels. Knowledge of UML class diagrams is sufficient to read and understand these definitions.
- **Evolvable language definition**: Metamodels can be modified like any other model. For instance, new modeling concepts can be added to the language by creating new subclasses for existing metamodel classes. An accessible language definition simplifies adapting modeling languages based on metamodels.

Generic tools, metamodel agnostic, may be developed at the metametamodel level. Current metamodeling frameworks offer sophisticated reflection techniques to develop programs applicable to all instances of metamodels conforming to a specific meta-metamodel. Examples include.

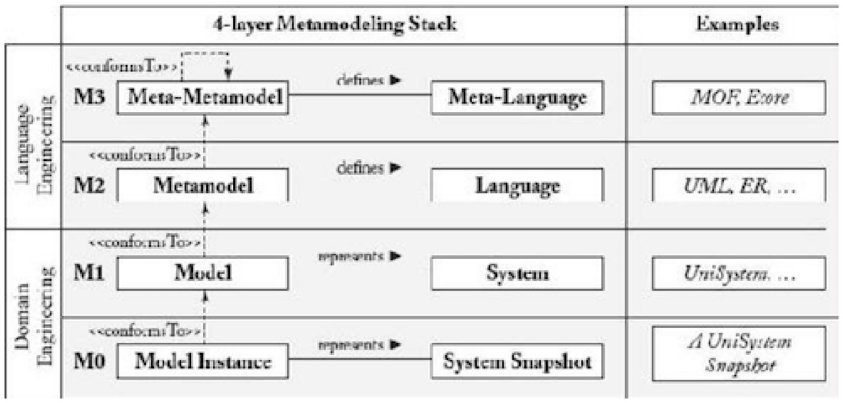
- **Exchange formats**: The meta-metamodel supports serializing/deserializing models into XML documents for exchange between tools supporting the same meta-metamodel.
- **Model repositories**: Models can be stored and retrieved from a model repository using

generic storage/loading functionalities, similar to model exchange.

- **Model editors**: Generic editors applicable to all models, regardless of the modeling language, may be provided for model modification.

The list of generic support is extensive. Simple programs can compute model metrics, such as the number of model elements. Sophisticated tool support can be developed at the meta-metamodel level, e.g., model comparison and versioning support, as discussed in Chapter 10.

Four-layer metamodeling stack. Current metamodeling tools typically use a four-layer metamodeling stack, as introduced in Chapter 2. Figure 7.2 shows an overview of this architecture. The upper half focuses on language engineering, building models for defining modeling languages, while the lower half focuses on domain engineering, building models for specific domains. The top layer, named M3 in OMG terms, contains metamodeling languages that specify the metamodeling concepts used to define metamodels. These languages are usually focused and offer a minimal set of concepts. To have a finite metamodeling architecture, these languages are often reflexively defined, meaning they can describe themselves. No additional language on top of M3 is needed to define M3, but having the same language on M4 as on M3 seems to be practical and offers similar expressibility. MOF and Ecore are designed as reflexive languages typically used on the M3 layer.



On the M2 layer, metamodels define modeling languages (e.g., UML, ER) by defining their concepts. These metamodels can be instantiated to build models on the M1 layer. M1 models represent systems like a university management system (UniSystem in Figure 7.2) and define domain concepts using language concepts from M2. On the M0 layer, domain concept instances represent real-world entities, e.g., a snapshot of the university management system at a specific time.

The four-layer metamodeling stack assumes that a model on layer M conforms to a model on layer $M+1$. This relationship, called *conformsTo*, means that a model is a graph with nodes and edges that fulfill the constraints of the next higher level. If each element on layer M fulfills the constraints of its type element on layer $M+1$, the model on layer M conforms to its *type model* on layer $M+1$. In case of M0 and M1, the domain model instances conform to the domain model. In case of M1 and M2, the model conforms to its metamodel. In case of M2 and M3, the metamodel conforms to its meta-metamodel. Finally, in case of M3, the meta-metamodel conforms to itself, as the *conformsTo* relationship is reflexive.

> **Résumé** :
>
> - Metamodeling languages, such as MOF and Ecore, are developed using metamodeling techniques. MOF is the OMG standard, while Ecore, tailored to Java, is part of the Eclipse Modeling Framework.
> - MOF, a specialized DSML for metamodeling, reuses a subset of UML's core, focusing on defining other modeling languages while omitting certain UML concepts. The main differences lie in their domains.
> - Metamodeling allows language designers to define abstract syntax for modeling languages, offering benefits regardless of the metamodeled language.
>   - **Precise language definition**: Machines can process formal language syntax definitions, such as metamodels checking model validity.
>   - **Accessible language definition**: UML class diagrams define metamodels, requiring UML knowledge for understanding.
>   - **Evolvable language definition**: Metamodels allow for the addition of new modeling concepts by creating subclasses for existing metamodel classes, simplifying the adaptation of modeling languages.
> - Generic tools can be developed at the metametamodel level using sophisticated reflection techniques.
>   - **Exchange formats**: The meta-metamodel supports XML serialization/deserialization for model exchange.
>   - **Model repositories**: Model repositories store and retrieve models using generic storage/loading functionalities.
>   - **Model editors**: Generic model editors are available for all models, regardless of modeling language.
> - Generic support for models includes computing model metrics and developing sophisticated tool support at the meta-metamodel level.
> - A four-layer metamodeling stack is commonly used, with the upper half focusing on

language engineering and the lower half on domain engineering. The top layer, M3, contains metamodeling languages for defining metamodels, often reflexively defined for a finite architecture.

- Metamodels define modeling languages (M2), which are instantiated to build models (M1) representing systems. Domain concept instances (M0) represent real-world entities.
- The four-layer metamodeling stack assumes models conform to higher-level models, with each layer's elements fulfilling the constraints of the next layer. This relationship is reflexive for the meta-metamodel.

7.2.1 METAMODEL DEVELOPMENT PROCESS

One way to conceptualize a metamodel is to view it as the schema that stores its models within a repository. Consequently, developing a metamodel is akin to creating a class diagram for a concrete domain. The initial step involves identifying the concepts that should be supported by the language. Subsequently, these concepts must be concisely specified using a metamodeling language. Once the first metamodel version is ready, generic editors can be employed to test the modeling language. These editors facilitate the interpretation of the metamodel, enabling the construction of example models. By utilizing these editors to model reference examples, a preliminary assessment can be made to determine if all concepts have been adequately defined or if any modifications to the metamodel are necessary. In essence, the metamodeling process, in its most basic form, comprises a three-step, iterative, and incremental process.

- **Step 1: Modeling domain analysis**: Three aspects must be considered in developing a modeling language: *purpose*, *realization*, and *content*. The most challenging aspect is identifying modeling concepts and their properties. To do this, analyze the modeling domain and find reference examples that should be expressible in the language. These examples define the language's requirements.
- **Step 2: Modeling language design**: A metamodeling language formalizes identified modeling concepts by modeling its abstract syntax and constraints using OCL. This step yields a metamodel for developing the language.
- **Step 3: Modeling language validation**: The metamodel is instantiated by modeling reference examples to validate its completeness and correctness. Other language design principles like simplicity, consistency, scalability, and readability must also be considered. This step provides valuable feedback for the next metamodel development iteration.

This process currently focuses on developing the abstract syntax of the language. To get domain expert feedback, a concrete syntax is also needed for an accessible modeling language. In practice, the complete process should involve developing both the abstract and concrete syntax. However, we discuss them sequentially in this book for didactic reasons, but strongly encourage developing both syntactical concerns together in practice.

To make the metamodel development process more concrete, it’s now being used to develop a metamodel for a DSML.

> **Résumé** :
>
> - The metamodel development process, in its most basic form, comprises a three-step, iterative, and incremental process, which involves identifying concepts, specifying them using a metamodeling language, and testing the modeling language with generic editors to build example models.
>   - **Step 1: Modeling domain analysis**: Developing a modeling language involves considering purpose, realization, and content, with identifying modeling concepts and their properties being the most challenging aspect.
>   - **Step 2: Modeling language design**: A metamodeling language formalizes modeling concepts using OCL, yielding a metamodel for language development.
>   - **Step 3: Modeling language validation**: The metamodel is validated through modeling reference examples, also considering other language design principles for feedback.
> - Developing both abstract and concrete syntax is crucial for a modeling language, though discussed sequentially for didactic reasons.
> - Metamodel development process is being used to develop a metamodel for a DSML.

Step 1: Modeling Domain Analysis

Several sources of information can be exploited to find appropriate modeling concepts. The choice of sources depends on the purpose of the modeling language. For abstracting from low-level program code, existing programs can be a major source of information. Recurring patterns in the code can be abstracted to modeling concepts. For a specific domain, document analysis or expert interviews may be appropriate. These activities should lead to concrete reference examples for communication with domain experts, testing the language and code generators, etc.

Essence of the abstract syntax. The first version of the language content description includes various information. It primarily introduces modeling concepts by defining their properties, which is crucial for developing the abstract syntax. Sometimes, other aspects, like notation and semantics, are also discussed. To build the first version of the metamodel, the essence of the abstract syntax must be filtered out, resulting in a list of concepts and their properties.

> **Résumé** :
>
> - Existing programs, document analysis, and expert interviews can inform modeling concepts, providing reference examples for communication, testing, and code generation.
> - The first version of the language content description introduces modeling concepts and their properties, which are essential for developing the abstract syntax.

Step 2: Modeling Language Design

First, concepts are transformed into classes, intrinsic properties into attributes, and extrinsic properties are defined as associations between classes. This initial structure allows reasoning about further modeling constraints. Attributes require types like String, Integer, and Boolean. If there's a range of possible values, enumerations may be defined. For the association ends, upper and lower bounds of multiplicities must be set properly.

Furthermore, one must reason about the containment structure of the metamodel, as elements in models are often nested. For instance, the content layer contains classes, which contain attributes. This structure must be defined in the metamodel by declaring association ends as compositions. The most significant consequence is that deleting the container element deletes all of its contained elements.

To improve readability and extendability, inheritance relationships between classes reuse attributes and associations (e.g., abstract classes for pages and links). Refactorings on metamodels can improve class diagram structure. Shifting attributes from subclasses to superclasses, extracting superclasses from existing classes, or substituting associations with explicit classes to define additional properties are recurring operations. Metamodels should fulfill object-oriented modeling quality attributes.

Discourse. For defining a modeling concept, metamodeling languages offer three options: classes, attributes, or associations. When choosing a representation, consider the advantages and disadvantages of each. This decision significantly impacts the modeling possibilities and user experience. User feedback can help improve the language by switching between patterns.

If a more powerful content modeling language is needed, import existing metamodels, like the class diagram part of the UML metamodel, instead of remodeling the language from scratch.

Modeling constraints. Several constraints cannot be defined by current metamodeling languages only in terms of graphical elements. Therefore, OCL defines additional constraints as well-formedness rules. These rules are implemented as additional invariants for metamodel classes and must hold for every model. Thus, constraints are defined on the metamodel and validated on the model level.

Introducing OCL invariants for metamodel classes precisely defines a modeling language, leading to higher-quality models, especially important for code generation. Otherwise, problems may go undetected until the final implementation, manifesting as compile-time or runtime errors, or remain undetected in the worst case.

OCL invariants can define well-formedness rules, modeling guidelines, and best practices, including naming conventions.

> **Résumé** :
>
> - Concepts are transformed into classes, intrinsic properties into attributes, and extrinsic properties into associations. Attributes require types and enumerations, while association ends require multiplicity bounds.
> - The metamodel’s containment structure, defining nested elements like classes and attributes, necessitates composition associations, ensuring container deletion deletes contained elements.
> - Inheritance relationships between classes reuse attributes and associations, improving readability and extendability. Refactoring metamodels, such as shifting attributes and extracting superclasses, can enhance class diagram structure.
> - Metamodeling languages offer classes, attributes, and associations for defining modeling concepts, each with advantages and disadvantages impacting modeling possibilities and user experience.

> - Import existing metamodels for more powerful content modeling.
> - OCL defines additional constraints as well-formedness rules, implemented as invariants for metamodel classes, ensuring model validity.
> - OCL invariants for metamodel classes precisely define a modeling language, improving model quality and aiding code generation.
> - OCL invariants define well-formedness rules, modeling guidelines, and best practices.

Step 3: Modeling Language Validation

Early validation of the abstract syntax can be achieved by manually instantiating the metamodel. This allows immediate assessment of whether the designed metamodel covers the modeling domain. Since object-oriented metamodeling languages like MOF define metamodels, models consist of objects. UML object diagrams represent models by instantiating class diagrams using:

- *objects* for *Classes*;
- *values* for *Attributes*; and
- *links* for *Associations*.

Object diagram notation. Objects are notated like classes using a rectangle. The first compartment contains the object's identifier and type, which must correspond to a class name in the metamodel. The second compartment defines attribute values in attribute slots. Links are notated as associations between objects.

For instantiating models from metamodels, the same rules as instantiating object diagrams from class diagrams apply. First, only concrete classes can be instantiated. Second, data types like enumerations constrain attribute values but cannot be instantiated. Third, meta-features of attributes and references (multiplicities and uniqueness constraints) constrain object diagrams. Finally, containment references are represented as links in object diagrams. Deleting a container object automatically deletes all directly and indirectly contained elements. To enhance readability, containment links are shown with the black diamond notation.

Feedback for metamodel improvements. When testing metamodels, several changes may be needed to represent and formalize the language properly. Common modifications include marking classes as concrete/abstract, setting references as containment/non-containment, restricting/enlarging feature multiplicities, specializing/generalizing feature types, or deleting/ introducing new elements. More complex changes, such as refactoring for switching between metamodeling patterns or introducing design patterns like the composition pattern, may also be required to improve the language definition.

Changing the metamodel when instances exist may cause trouble if the changes break the metamodel's conformance relationships with the models. For instance, renaming or deleting metamodel elements may prevent the models from being loadable in modeling editors. Chapter 10 discusses how to handle such metamodel/model co-evolution problems.

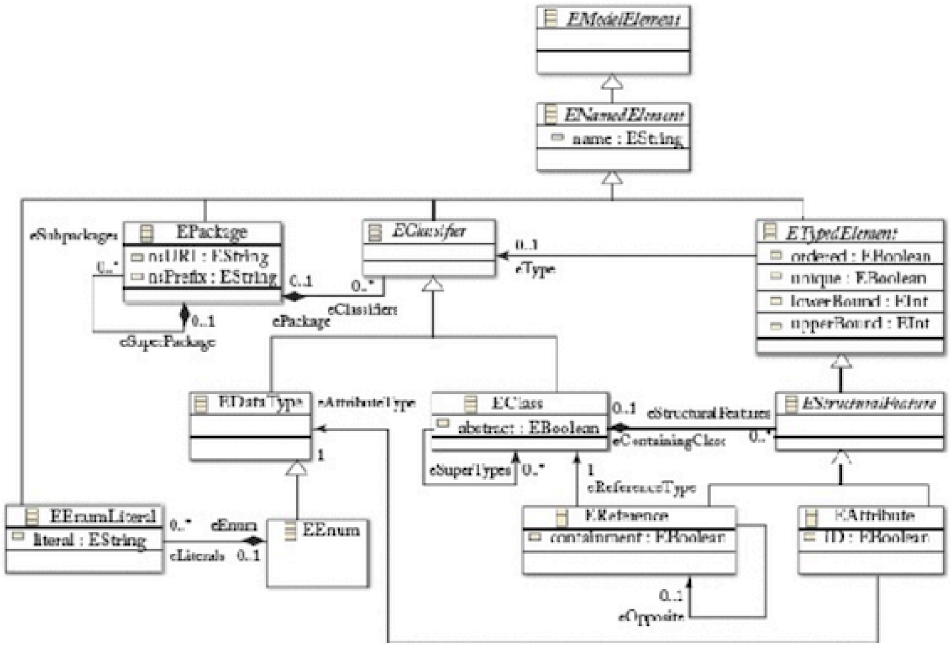
> **Résumé** :
>
> - Early validation of abstract syntax can be achieved by manually instantiating the metamodel. UML object diagrams represent models by instantiating class diagrams using objects, values, and links.
> - Objects are notated like classes using rectangles with identifiers, types, and attribute values. Links are associations between objects.
> - Instantiating models from metamodels follows rules similar to instantiating object diagrams from class diagrams. Concrete classes can be instantiated, while data types and meta-features constrain object diagrams.
> - Testing metamodels may require changes to represent and formalize the language, including marking classes, setting references, restricting feature multiplicities, specializing feature types, and introducing new elements.
> - Changing the metamodel when instances exist can cause trouble, potentially preventing models from being loadable in modeling editors. Chapter 10 discusses how to handle metamodel/model co-evolution problems.

7.3.2 METAMODELING IN ECLIPSE

After defining metamodels using UML class and object diagrams, we discuss how EMF supports these tasks. EMF has its own metamodeling language, Ecore, and tool support for defining and instantiating Ecore-based metamodels. This subsection provides a brief overview of EMF and its core functionalities. For more information, refer to dedicated EMF resources on our book website.

Ecore's modeling concepts are depicted in Figure 7.3. Ecore comprises `EClassifiers` (`EClasses, EDataTypes, and EEnums`) and `EStructuralFeatures` (`EAttributes` and

`EReferences`). `EClasses` are first-class citizens in Ecore-based metamodels. They can have multiple `EReferences` and `EAttributes`, and multiple superclasses. `EAttributes` are part of a specific `EClass`, have lower and upper bounds, and can be ordered and unique. `EAttribute` types are either simple `EDataTypes` or `EEnums` with restricted values. `EString`, `EBoolean`, `EInt`, and `EFloat` are part of Ecore's default data type set. `EReferences` are part of a specific `EClass` and have lower and upper bounds. They refer to `EClasses` and can be bi-directional with opposite `EReferences`. `EReferences` can be ordered, unique, and containment references, meaning all contained objects are deleted if the container is deleted. `EPackages` group related `EClasses`, `EEnums`, `EDataTypes`, and other `EPackages`.



For specifying Ecore-based metamodels in Eclipse, several concrete syntaxes are available. EMF has a tree-based editor for modeling metamodels in their abstract syntax, similar to object diagrams, but using the containment hierarchy to form an explicit tree structure. However, working with the abstract syntax doesn't scale, so editors supporting concrete syntaxes for Ecore are recommended. Several graphical editors for Ecore, such as the Ecore tools project, allow modeling Ecore metamodels using a similar syntax as UML class diagrams. Other approaches define Ecore metamodels textually, like KM3 and Emfatic.

OCL support for Ecore.

Several plugins in Eclipse define OCL constraints for Ecore-based metamodels. Eclipse OCL, a project implementing the OCL standard, includes OCLinEcore, which defines invariants for

Ecore-based metamodels and evaluates constraints within modeling editors. The Dresden OCL Toolkit provides further support for defining and evaluating invariants for EMF-based models. The Epsilon Validation Language of the Epsilon project, inspired by OCL, has a slightly different syntax compared to Java and syntax extensions for practical aspects of model validation, such as customizable error messages.

> **Résumé** :
>
> - EMF supports metamodeling using Ecore and tool support for defining and instantiating Ecore-based metamodels.
> - Ecore comprises `EClassifiers` (`EClasses` , `EDataTypes` , and `EEnums`) and `EStructuralFeatures` (`EAttributes` and `EReferences`). `EClasses` can have multiple `EReferences` and `EAttributes` , and multiple superclasses. `EAttributes` have lower and upper bounds, and `EReference` types can be bi-directional and containment references.
> - EMF provides a tree-based editor for modeling metamodels in their abstract syntax, but concrete syntaxes are recommended for scalability. Graphical editors like the Ecore tools project and textual approaches like KM3 and Emfatic are available.
> - Eclipse OCL, Dresden OCL Toolkit, and Epsilon Validation Language support defining and evaluating OCL constraints for Ecore-based metamodels.