A Tour of Language Customization Concepts

COLIN ATKINSON

University of Mannheim
68161 Mannheim
Germany

THOMAS KÜHNE

Darmstadt University of Technology
64289 Darmstadt
Germany

Abstract

Although the UML is often perceived as a universal modeling language it was not designed to fulfill this goal. Rather, the primary goal of the designers was to provide a unified modeling language. Because the applicability of the core language is limited, the UML has always offered mechanisms to enable it to be adapted to the needs of different users and applications. Over the years, a wide variety of different language customization mechanisms have been proposed, ranging from stereotypes, profiles, and metamodeling to domain-specific languages. However, the relationships between these different approaches and the different capabilities and options that they provide to users have never been clearly elaborated. In this chapter we provide a tour of the most important language customization mechanisms and by means of a unified case study we compare and contrast their pros and cons. We also review the current “state of the art” and present our view of how the “domain-customized language” approach to software engineering can best be supported in the future.
1. Introduction

Software engineering involves the creation of many artifacts which may be described using a variety of languages. However, choosing the optimal language for each artifact is not easy. One of the most important issues is the tension between the goal of widespread communication and the goal of using the most appropriate language for each specific task. With respect to modeling, a standard like the UML (Unified Modeling Language) [16,17] which has unified and replaced a large number of similar object-oriented modeling notations, is an ideal facilitator of world-wide
communication. It has, however, been criticized as promoting a “one size fits all” approach that provides suboptimal abstractions for the majority of applications. This is not surprising since the “U” in “UML” was never meant to mean “universal”, i.e., the UML is useful in a broad range of applications, but certainly not all. Domain-Specific Languages (DSLs), on the other hand, provide the best possible fit for a particular task but create communication chasms since their concepts and notations are often only known to a small team of language designers and users. In the worst case, domain-specific languages can be conceptually very similar, but vary considerably in terms of their notation, thus creating an unnecessary Tower of Babel and undoing the consolidation achievement of the UML.

Fortunately, there is some useful middle-ground between a fixed language standard and an uncompromisingly adapted, one-off language design which we refer to as a domain-customized language (DCL). A domain-customized language is a derived language whose definition is based on some existing base language. It customizes that language for some specific purpose using such operations as reduction, modification, and extension. Domain-customized languages are therefore a subclass of domain-specific languages whose members have carefully defined relationships to their respective base languages. Typically, the base language will be a widely-known standard and the amount of customization will be minimized in the sense that as much as possible of the base language will be left unaltered. The use of a DCL thus allows maximum communication while still being tailored to a specific task. The OMG’s (Object Management Group) approach of regarding the “UML as a family of languages” [5] and supporting the customization of the UML with so-called profiles may be regarded as promoting the DCL concept.

However, a DCL approach will only be economically viable if the cost of creating a DCL—which involves the adaptation of the syntax and semantics of a language—does not exceed the return on investment, i.e., the productivity gains. A DCL approach therefore needs to make the definition and derivation of languages as simple and intuitive as possible. As well as providing a chronological overview of the main concepts and technologies involved in defining and using DCLs, this chapter will therefore have a particular focus on the relative ease with which DCLs may be derived from base languages. Only language customization approaches and technologies that manage to minimize the cost of creating a DCL will have a long term future in software engineering.

2. Languages, Abstraction and Domain-Specificity

To set the scene for the tour of language customization technologies that follows, we use this section to explain some basic concepts and to introduce the case study that we use in the rest of the chapter.
2.1 Languages

In the following we do not distinguish between the traditional categories of modeling languages and programming languages, whose distinction has been considerably blurred by model-driven development approaches. Even though our examples are exclusively from the area of modeling, all observations and conclusions also apply to programming languages.

The task of a modeling language is to provide a set of concepts and an associated notation that allows the description of subjects of interest. In the case of software engineering, models are frequently used as construction plans, i.e., as descriptive models. In this case the subject is typically a software system to be built and sometimes includes the environment with which it interacts. The OMG’s MDA Guide consequently defines the term “model” as follows:

“A model of a system is a description or specification of that system and its environment for some certain purpose” [11].

It is, however, important to realize that “description” may refer to either of two modes of description: either one describes singular aspects of one particular incarnation of a system kind, i.e., one creates system snapshots, or one describes the universal aspects of all systems of a particular kind. Models using the singular description mode are token models [9] and represent the original subject. They are often expressed using UML object diagrams when referring to structural aspects of a system. Models using the universal description mode are type models and classify the original subject. They are often expressed using UML class diagrams when referring to structural aspects of the system. Although the description mode of a model is subject to interpretation—i.e., a class diagram may play the role of a type model (with respect to the described domain) but may at the same time also play the role of a token model (with respect to the implementation classes to be generated from it)—it can be useful to have a dedicated notation for each purpose. Some modeling languages, in particular domain-specific ones, only support one description mode explicitly. The UML supports dedicated notations for both token-level modeling (→ user instances) and type-level modeling (→ user types).

In order to be able to discuss the customization of modeling languages, we need to briefly introduce the fundamental ingredients of a language.

2.1.1 Abstract Syntax

The abstract syntax defines the basic set of concepts that can be used to make statements in the language together with rules for using them correctly. One particular useful way to specify the abstract syntax of a modeling language is to create a
model of the abstract syntax using the notation of the modeling language to be defined. This approach has been coined metamodeling since an (abstract syntax-) model is used to define the shape of other (user-) models. Metamodeling in this sense\(^1\) is useful to minimize the intellectual burden on language designers, since it does not require knowledge of a dedicated formalism such as (E)BNF, but lets language designers work with a familiar notation.

Figure 1 shows a simple model of the abstract syntax of the core parts of a MOF-like [13] modeling language expressed in the notation of that same language. The UML and the MOF share a common core which can be used to describe the abstract syntax of other languages, including UML and MOF themselves.

### 2.1.2 Well-formedness Rules

By analogy to programming languages, modeling languages typically involve a number of rules which cannot be enforced by the abstract syntax definition. Also known as static semantics in the area of programming language definition, well-formedness rules thus complement the restrictions introduced by the abstract syntax

\(^1\) This is, however, just one way to understand the term “metamodeling” [9]. We will discuss another interpretation in Section 6.
with further constraints which cannot be expressed using the metamodeling notation. For instance, the fact that identifiers are required to be unique within a namespace is a constraint that needs to be defined on top of the abstract syntax. Constraint languages such as OCL [18] can be used to formulate such constraints and attach them to the abstract syntax definition. Note that in contrast to grammars, metamodels may capture some well-formedness rules even without resorting to constraints, e.g., by using multiplicities.

2.1.3 Concrete Syntax

The concrete syntax defines the notation to be used to present structures which are represented using the abstract syntax. Textual languages also use the concrete syntax as a basis to generate import tools (i.e., parsers), but most modeling languages have a visual notation and only use the concrete syntax for displaying models.

2.1.4 Semantics

The final element of a language is its semantics which defines the interpretation given to sentences in that language, i.e., to expressions represented with the abstract syntax. Well-known ways of defining this interpretation include:

- **informal semantics**: in terms of natural language descriptions;
- **operational semantics**: in terms of relating abstract syntax concepts to execution machinery;
- **denotational semantics**: in terms of mathematical mappings to known semantic domains;
- **translational semantics**: in terms of mapping the language to a target language.

In all cases, the semantics are defined in terms of relationships to existing, accepted targets. In the context of tool-supported semantic definitions the translational semantics approach is the most common as it, in essence, corresponds to providing a compiler for a language. An operational semantics may be used to define an interpreter for a modeling language but then requires the additional specification of the execution machinery.

The economics of defining a DCL therefore depends on the ease with which the abstract syntax of a language, its well-formedness rules, concrete syntax, and its semantics can be defined and/or derived. Due to space constraints we will focus on the abstract syntax (see Sections 3–8) and only briefly touch upon the semantics (see Section 9). Before doing this, however, we need to clarify exactly what the term “domain-specific” entails.
2.2 Abstraction and Domain-specificity

Domain-specific approaches are often advertised as “raising the level of abstraction” and indeed many of their typical usage scenarios enable the specification of software at higher levels of abstraction. However, on closer inspection one observes that domain-specificity and abstraction level are in principle uncoupled and may be freely combined. We will briefly investigate the relationship between these two dimensions since it is instructive to see what choices are available in the two-dimensional design space and what role they play in typical development scenarios.

2.2.1 Abstraction

In the context of this chapter we use the term “abstraction” to be a measure of the extent to which a model or modeling language is geared toward solution technology. Thus, a model or language with a low degree of abstraction can be characterized as solution-oriented while a model or language with a high degree of abstraction can be regarded as problem-oriented. Notice that we are assuming that a model is a specification of a software system to be built and that the software is going to be executable. The ultimate “solution” technology from a software engineer’s point of view is therefore binary machine code. However, several levels of abstraction exist on top of this, such as assembly language, C, operating systems, libraries, virtual machines, object-oriented languages, middleware platforms, etc. The more “problem-oriented” a model or modeling language is, the more it will address the question of “what” is done by the system and the more it will abstract away from the properties of the ultimate solution technology. The more solution-oriented it is, the more it will address the question of “how” it is done in terms of the ultimate solution technology.

A solution-oriented language thus requires or allows—depending on whether one regards that as a necessity or a feature—a high degree of control over the realization aspects of a system. In contrast, a problem-oriented language focuses on the pure functional specification of a system without concern for realization aspects. The ambitious goal of domain-specific modeling—and of course model-driven development in general—is to allow the creation of models whose degree of abstraction is as close to the problem as possible, while still having a defined and automatic translation chain to the ultimate solution technology.

Of course, even without an automatic transformation chain, software engineering has been making use of different abstraction levels in the development process for a long time. A traditional waterfall process starts with highly abstract models in the analysis phase, progressing to a design phase with more solution-oriented models, and eventually ending up with very concrete and solution-specific descriptions of the system.
2.2.2 Domain-specificity

In the context of this chapter we use the term domain-specificity as a measure of the extent to which a modeling language is tailored to a certain application domain. A modeling language tailored to an application domain will directly support concepts which are naturally found in this application domain. A language with a low degree of domain-specificity contains general concepts which span multiple domains and is widely applicable (i.e., general-purpose), while a language with a high degree of domain-specificity contains concepts that are natural and well-suited for a particular domain and has a relatively narrow application domain in comparison. Domain-specific modeling languages therefore trade wide applicability for a minimum impedance mismatch which results in very intuitive and concise models.

Obviously, domain-specific approaches—such as Csound [4], a language dedicated to the creation of sounds and music, or Simulink [1], a language dedicated to simulation—have existed for a long time. However, the supporting tools and environment for such language were essentially developed “by hand” by the vendors that sell these systems, and users had to use the languages as they were delivered “out of the box”. The new interest in DSL’s stems from the fact that the definition of DSLs and their supporting tools is now a much less laborious task. As a result, they are now more frequently considered as a serious alternative or complement to regular software engineering practices.

2.2.3 Relationship between Abstraction and Domain-specificity

As illustrated in Fig. 2, abstraction and domain-specificity can be regarded as two orthogonal dimensions. The many ellipses in Fig. 2 indicate choices within the two-dimensional spectrum and therefore define the relative positions of the four displayed samples.

Figure 2 illustrates the fact that a modeling language has two properties—abstraction level and domain-specificity—which may occur in arbitrary combinations. For instance, the language in the bottom left-hand side corner of Fig. 2 is a highly general language, applicable to a wide range of application domains and a highly problem-oriented language, having very little concern for the realization of a solution. Such a language would typically be used in the “analysis” phase of a traditional software development project. Figure 3 shows GRL (Goal-Oriented Requirement Language) as a concrete example.

The bottom right-hand corner of our two-dimensional space features languages which are also general-purpose with respect to the application domain, but have a high-degree of solution-orientation. Java is an example of a widely-applicable language that requires/allows the specification of many implementation details.
The top right-hand corner features languages which are both highly customized to a certain application domain, such as e-commerce or gaming, and are designed to require/allow the specification of many realization details. It is fair to call these DSLs, however, the term DSM (Domain-Specific Modeling) is typically reserved for modeling languages in the top left hand corner of our two-dimensional spectrum. Such languages also focus on specific application domains, but they try to abstract away as much as possible from any realization concerns. An example might be a language for the specification of traffic light control systems offering concepts from that domain, such as cars, traffic lights, crossing etc. In contrast to a language with the same level of domain-specificity but a higher degree of solution-orientation, many realization aspects will be implicit and be either fixed or deferred to interpretation choices, such as various transformations employing different strategies for realizing control algorithms.

Figure 3, showing samples in the two-dimensional design space, points out that becoming more application domain-specific does not necessarily imply a higher level of abstraction. Likewise, a level of abstraction similar to that addressed by a dedicated “Pet Store” language may also be achieved with a general-purpose requirements language. The virtue of domain-specificity is the focus on a partic-
ular application area, allowing the models to be expressed using natural domain terms.

Although the two dimensions are theoretically independent, in practice there are correlations between them. Since typically the underlying goal of domain-specific languages is to provide a way for domain experts (but not computer scientists) to describe what they want, as opposed to how a system is going to be realized, most DSLs tend to be problem-oriented as well as domain-specific. In other words, most practical DSLs occupy the top left hand corner of the customization space. This frees developers from the need to work with solution-oriented concepts and allows them to write their high-level solutions using concepts natural to the domain in hand.

A common misconception with DSLs is that they always raise the level of abstraction at which applications are written beyond that of general purpose language such as the UML. This is because, like general purpose languages, they reduce the need for the human engineer to worry about the idiosyncrasies and properties of the underlying execution technology. However, this should not be confused with “raising the level of abstraction” relating to language concepts. What DSLs in fact do is to

---

**Fig. 3.** Some samples in the language design space.
provide the “right”, or the “best” abstractions for the problem in hand rather than the most abstract. Using concepts that are too abstract for a particular topic can be just as problematic as using concepts that are too concrete. In both cases the existing abstractions have to be constrained or generalized to the required concepts which can involve just as much effort. DSLs avoid this problem by providing exactly the right abstractions for the problem in hand.

2.2.4 **Abstraction and Domain-Specificity in the Development Process**

Even though Figs. 2 and 3 suggest an orthogonal and free combination of abstraction and domain-specificity there are of course combinations which are more useful and popular. Figure 4 shows a number of models expressed using different modeling languages and highlights the typical location of primary artifacts within a given development style.

In Fig. 4 the bottom row corresponds to typical mainstream development practices, using general-purpose languages. The shaded area in the bottom row indicates how the effort of human engineers is typically distributed in most mainstream projects.
today. Generally speaking, the amount of effort spent on developing artifacts in tradi-
tional development is much greater at the solution-oriented end of the abstraction
spectrum. This reflects the fact that the primary artifact is executable code (or some-
thing very close to it), with more high-level models playing only a supporting, but
definitely secondary, role.

In contrast, the top row of the diagram represents the opposite end of the domain-
specificity spectrum in which highly application domain-specific languages are used
to develop software for very narrow domains. As shown by the shaded area in the
top row, this has the very distinct effect of shifting the position of the primary ar-
tifacts. Two observations explain this phenomenon: First, truly domain-specific and
solution-oriented languages—of the kind needed to operate in the top right hand
side of the diagram—are today few and far between. Thus, in most domains it is
simply not economically viable to develop software in a highly domain-specific but
solution-oriented way. Second, even if such languages were more widely available
there would often be little incentive to use them. That is to say, it is much easier to
define automated transformations from problem-oriented to solution-oriented mod-
els in very specialized domains than it is in general purpose domains. For instance,
it is much easier to define a transformation for a tailored version of “Class” (e.g.,
“ShoppingCart”) compared to the general, underconstrained case where all is known
about the source element that it is a “Class”. Therefore, there is rarely a need to use
a solution-oriented, domain-specific language since the desired executable system
may be generated from a problem-oriented model just as well, with any realization
decisions deferred to the transformation.

Finally, the middle row shown in Fig. 4 is characteristic of “model-driven devel-
opment”. The focus on solution-oriented modeling (i.e., coding) is much less than
in the general purpose case, but higher than in the highly application specific case.
The middle row also shows that the UML is not only “general-purpose” with respect
to the application domain, it is also widely applicable with respect to the abstrac-
tion level targeted by corresponding models. Hence, in a transformation chain from
problem-oriented to solution-oriented models using relatively small transformation
steps between models, UML (often customized using profiles) may be used multiple
times.

2.2.5 Orphan DSLs versus DCLs

As illustrated in Fig. 2, domain-specificity is a relative concept rather than an
absolute concept. Certainly, there is no absolute threshold beyond which a language
deserves the label “domain-specific” and below which it does not. However, although
we do not provide any objective criteria here—and these may be hard to define—
judging whether one language is more or less domain-specific than another is fairly
intuitive.
In practice the label “domain-specific language” is used in a loose way to refer to any language which was designed with the deliberate intention of being more domain-specific than general, mainstream languages widely used for software development. The term “domain-specific language” is thus in practice often used to designate a language as being much more domain-specific than UML or Java, which are thought of as general-purpose languages.

We retain and use this loose meaning for the notion of a DSL. However, there is an important dichotomy which has a fundamental bearing on the topics we will discuss in the rest of the chapter: It is the difference between orphan DSLs and customized DSLs:

An orphan language is a language that is defined without any formal or explicit reference to an existing language. The language therefore exists independently in the customization space (as illustrated in Fig. 2) and is not regarded as being part of a family of languages. An orphan language might be the result of reusing an existing language in its development, but as long as the reuse was done in an ad-hoc way, the result remains an orphan language. An orphan language which is domain-specific is known as an orphan DSL.

A customized language is a language that is explicitly derived from and related to another language with the goal of supporting a different level of domain-specificity and/or abstraction. Such a language always exists in relation to another language. A customized language which is also a domain-specific language is referred to as a “domain-customized language” or DCL.

The notions of “orphan DSL” and “DCL” are thus disjoint and partition the set of DSLs. All DCLs are DSLs, but not all DSLs are DCLs. Formally, \( S_{DCL} \subseteq S_{DSL} \) and \( S_{ODSL} = S_{DSL} \setminus S_{DCL} \). The basic premise of DCL-based development is that the effort involved in creating a DCL (and supporting environment) from a general purpose language like UML is lower than that involved in creating an orphan DSL. Furthermore, DCL-based development aims to draw on the familiarity of developers with the base language, requiring them to learn only the new, customized concepts.

2.3 Case Study

An important goal of this chapter is to present the ideas and technologies of DCL-based software engineering in terms of a single unified case study. We chose to use an electronic Pet Store system because even though it relatively simple, it is rich enough to exercise all the key issues in DCL-based development.

To help illustrate how DCL based development works and to provide various reference points along the customization and abstraction axes we will start by showing
three different kinds of models which occupy three different locations in the 2D customization space.

### 2.3.1 UML Analysis Model

Figure 5 shows a model that corresponds to the bottom-middle entry in Fig. 4. Due to space constraints we only show a rather reduced model, focusing on a few of the key abstractions in the Pet Store problem space. A comprehensive model for the Pet Store would be much larger, but including all this additional information would serve no didactic purpose. On the contrary, important phenomena would be more difficult to spot.

Note that all the elements in Fig. 5 are expressed using standard UML modeling element types, such as classes, attributes, associations etc. As long as the semantics of these match the modeled domain, the use of UML works well, without any impedance mismatch. Since the UML is not a universal modeling language this will not always be the case. Even when it is a good fit, though, it is important to note that there is more freedom in editing the UML diagram of Fig. 5 then one may desire. Let us assume the job of an application developer is to adapt the design to include two kinds of shopping carts, e.g., using different payment methods: He may choose to include a “contains” aggregation relationship between the new “ShoppingCart” element and “User”, even though this does not make sense in the Pet Store application.

![Diagram of Pet store analysis model](image-url)
domain. Generally speaking, whatever domain-specific rules may exist, they cannot be enforced by a plain UML diagram.

Figure 5 makes use of a so-called powertype in order to support the assignment of an individual “taxRate” to subclasses of “Product”, and to make sure the latter have a “price” attribute, which they must inherit from “Product” by virtue of being an instance of powertype “ProductType” [14]. We will see how this design detail is treated differently in each of the following variants.

2.3.2 Domain-Specific Model

Figure 6 shows the model that corresponds to the top left-hand entry of Fig. 4, using a language specialized for the design of Pet Stores, a subclass of electronic retailing applications. As is typical for such a DSL, a dedicated, intuitive notation is used to depict the concepts from the application domain.

Note that the model using domain-specific notation needs less labels, since each of the connectors has a dedicated meaning and can be distinguished from others by shape, color, or what element types it connects. Also note that there is no explicit mentioning of a “ProductType”. The Pet Store DSL may simply specify that all instances of product types (such as “Animal” or “Food”) will automatically have a “taxRate” slot and a “price” attribute.

![Fig. 6. Domain-specific pet store model.](image)
2.3.3 UML Solution-Oriented Model

Figure 7 shows the model that corresponds to the entry at the center of Fig. 4. It features concepts similar to those of Fig. 5, but these are now more solution-oriented because they refer to J2EE and EJB (Entity Java Bean) middleware technology. The model of Fig. 7 actually uses UML’s profiling mechanism (see also Section 3) to label the model elements in order to convey their mapping to different kinds of implementation concepts. The combination of UML + EJB profile can be considered to be a DCL.

In comparison to the model of Fig. 5 one may observe that the model of Fig. 7 not only adds mapping information, but also documents further realization choices. The “ProductType” concept has been realized as a reference from products to an object that both indicates the type of a product and contains a respective tax rate value. This reflects the fact that the intended solution technology for the EJB design only assumes an object level at runtime without the ability to attach values to classes, inquire about the type of objects and/or add new product types at runtime.

3. Derivation Types

In this section we consider the different kinds of derivations of a general-purpose modeling language that might be sensible from a language user’s point of view. In other words, we consider what is the available range of options for creating one language by derivation from another base language and what their associated properties are. We are not so much interested in how the base languages or derivations are defined, but in the logical effect of the derivation in terms of the modeling capabilities it
offers to users and the extent to which it enables backward and upward-compatibility of models.

In general, all derived languages are created from a base language by applying some derivation specification to it. We can represent this formally as follows:

\[ L_d = L_b \Phi D, \]

where \( L_d \) is the new language, \( L_b \) the base language, \( \Phi \) the derivation operator, and \( D \) the derivation description. A derivation description, \( D \), is composed of one or more primitive transformation operations, such as removing a language element, adding a language element, etc.

If a language is derived from a base language with some customization goal, i.e., the intention to place it at a different location in the customization space (see Fig. 3) we refer to it as a \textit{customized language}. Customization therefore may involve a change to the domain-specificity and/or the targeted abstraction level of the language. There are a number of derivation descriptions which do not change a language’s position in the customization space, e.g., renaming of concepts or replacing one paradigm with another without changing the domain-specificity or abstraction level. Such derivation descriptions then lead to purely derived languages as they have no customization goal in the sense of our coordinate system.

Obviously, one point in our customization coordinate system can be occupied by many languages, i.e., by languages addressing the same customization goal but using different means, such as different terminology, basic paradigm, etc. While they are equivalent in terms of reaching a customization goal, they will differ in terms of compatibility to the base language. A language that manages to attain a certain customization goal while maintaining a maximum level of compatibility (see below) to the base language we refer to as an \textit{essentially-customized language} (ECL).

Essentially-customized languages are very useful because they define the optimal compromise between being widely understood as represented by modeling standards (by maintaining maximum base language compatibility) and being the best possible fit for an application domain as represented by DSLs (by being customized to the desired level).

When a new language has been derived from a base language it is important to determine which of the following relationships hold between the two languages, if \( L_d = L_b \Phi D \):

\begin{align*}
\text{non-conformant derivation} & : \neg \exists m_b \in L_b: m_b \in L_d, \quad (1) \\
\text{partially-conformant derivation} & : \exists m_b \in L_b: m_b \in L_d, \quad (2) \\
\text{fully-conformant derivation} & : \forall m_b \in L_b: m_b \in L_d \quad (\equiv L_b \subseteq L_d). \quad (3)
\end{align*}
If a derivation yields property (1) then no model that can be created with the base language is a valid model of the new language. This implies rather radical reductions and modifications to the base language. In practice, this will rarely occur, as it strongly questions the choice of the base language to start with. If property (2) applies, at least one or more of the models created with the base language are also valid instances of the new language. However, unless the property (3) also applies, it means some models are not. This has important consequences for the upward-compatibility of an organization’s model base. If property (3) does not apply, then steps may have to be taken to ensure that an organization’s existing model base does not fall out of date. Note that a derived language with property (3) is known to be an essentially-customized language, since it achieves an optimal result regarding the upward-compatibility of models in the base language.

In the following subsections we discuss three different kinds of language derivations that may occur in practice. Note that we will often use the term “language” to refer to the set of all instances that may be created with a given language definition, i.e., to the set of all conformant models of a language definition.

### 3.1 Reduction

Reduction is the simplest kind of derivation. A reduction does not make any new modeling constructs available to modelers—on the contrary, it takes them away. The reduced language thus offers a subset of the features of the base language. Two ways of creating a language subset in this way may be distinguished.

#### 3.1.1 Destructive

A destructive reduction may make any change to the base language other than add new features. The result will be a language that is less expressive than the base language. However, because it is possible to remove features which were regarded as mandatory in the base language there can be cases in which a model of the reduced language will not be an instance of the base language. The right-hand side of Fig. 8

![Fig. 8. Destructive reduction.](image)
A TOUR OF LANGUAGE CUSTOMIZATION CONCEPTS

shows a change to the metamodel of Fig. 1 that illustrates this situation. If one removes concepts from a language that other concepts refer to in a mandatory manner, then models of the new language will not be conformant to the base language.

One strategy to gradually remove features without immediately invalidating models that use them is “deprecation” as known from Java. At first sight there might seem little use for reducing a language, but it can be quite useful to enforce company wide standards and modeling/programming practices. For example, a company might want to make sure that its modelers do not use “goto”-statements when modeling C programs or refrain from using powertypes in modeling.

3.1.2 Conformant

As illustrated in Fig. 9, a conformant reduction creates a new language (“language” here in the sense of the set of all conformant models) which is a proper subset of the base language, implying that only optional parts of the base language are removed. The right-hand side of Fig. 9 shows how a conformant reduction of the simplified metamodel in Fig. 1 could be achieved by removing the element “GeneralizableElement” from the metamodel.

On a technical level, the constraints of the new language must be within (i.e., stronger than) the constraints of the base language. This does not ensure that property (3) will apply after a conformant reduction, since a base language model may use optional features, but it ensures that every model of the derived language is a model of the base language.

3.2 Extension

Extension is the opposite of reduction. When a base language is extended, new modeling features are added and none are removed. However, there are two basic ways in which this can be done, resulting in two basic subcategories of extension.
3.2.1 Generalization

In generalization extensions, the new language is a superset of a number of base languages (see Fig. 10). Generalization extensions are a popular means to join hitherto independent base languages together in a unified extended language. As a consequence, a generalization extension satisfies the property (3) above—every model expressed in one of the base languages is also an instance of the new language. Apart from extending the notion of derivation to more than just a single base language, though, generalization extensions have the same subcategories as specialization extensions (see below). We will therefore treat them as a special case of disjoint specialization extensions that just feature more than one base language.

3.2.2 Specialization

In specialization extensions, the new language is a superset of the base language. As a result, property (3) applies and the new language is known to be a customized language. However, three different subcategories of specialization can be identified depending on whether or not the new language constructs can be understood in terms of the base language. Intuitively, we introduce a base language view on models created in the new language. If a new modeling concept can be seen using such a base language view then a new language construct is either a copy or a refinement of a base language construct. Technically, a parser for the base language will be able to read the new language construct, but will lose any additional information carried by the new construct. Intuitively, this is the case when a model expressed using the derived language can be regarded as being a direct instance of the derived language and an indirect instance of the base language. This is analogous to looking at an object through the interface of one of its class’s superclasses.

3.2.2.1 Disjoint. A disjoint extension creates new modeling constructs which are entirely unrelated to the modeling constructs of the base language (Fig. 11). Hence the only modeling constructs which can be understood in terms
of the base language are the base language constructs themselves:

$$\neg \exists m_d \in L_d \setminus L_b: m_d \in_p L_b.$$ 

Since such an extension results in a new language that is essentially the union of the base language plus an additional extension language, one may regard generalization extension as a special case of disjoint specialization, which happens to involve multiple base languages.

### 3.2.2.2 Conservative

A conservative extension does not introduce any entirely new modeling constructs but just refinements of existing ones (Fig. 12). Assuming that conservative extensions adhere to the Liskov Substitution Principle [10], all models expressed in the new language are therefore also indirect instances of the base language, i.e., the constructs of the new language can be understood in

```
2 "\text{in}_p" means "element of, with respect to a base language view", i.e., a member test involving a projection of the element with respect to the base language.
```
3.2.2.3 Additive. An additive extension lies between the two extremes represented by the previous two alternatives (Fig. 13). Some new language constructs from the extension are refinements of the base language; others are unrelated to the base language,

\[ \exists m_d \in L_d \backslash L_b : m_d \in \mathcal{P}L_b. \]

This results in a new language that may instantiate some models featuring new language constructs which will be indirect instances of the base language, i.e., can be entirely understood using a base language view, whereas the rest of the models featuring new language constructs cannot be understood using a base language view. Property (3) still applies, of course.

3.3 Modification

Modifications neither purely remove nor purely add modeling constructs, they may change existing ones and/or involve a combination of reduction and extension (Fig. 14). Since modifications involve reduction and extension (otherwise they would be plain extensions or reductions respectively), the corresponding number and kinds of subcategories exist (conformant/destructive × disjoint/conservative/additive), i.e., six in total. The resulting properties (1)–(3) can be calculated for modifications by simply combining the individual reduction and extension properties, with the weakest (strength increasing in the order of (1)–(3)) always taking precedence.
Note that a destructive reduction may be complete, so if all of the base language is removed prior to adding different, unrelated language concepts (disjoint extension) then property (1) applies, strongly questioning the choice of the base language which is not reused at all.

4. Lightweight Language Customization

Of all the customization types discussed in the previous section, “Conservative Specialization Extension” has the nicest properties. Not only does it ensure that all models expressed in the base language are expressible with the customized language (→ upward-compatibility of models) but also that all models expressed using the customized language are understandable (albeit with some information loss) through a base language view (→ backward-compatibility of models). Obviously, restricting oneself to just “Conservative Specialization Extension” is quite limiting, since then it is neither possible to remove concepts from the base language nor to introduce completely new ones that go beyond specializations of existing ones. Nevertheless, as it is an extension type that ensures maximum compatibility in both upward- and backward directions, it is an attractive option when customizing a language.

The so-called “Lightweight Extension” mechanism associated with the UML directly supports the “Conservative Specialization Extension” form of customization provided that one refrains from using constraints to remove features from the base language. The primary reason why this form was chosen for the UML, however, was the fact that the first generation of modeling tools that existed at the time had the UML language definition hardwired into their code. In other words, it was not possible to change the language supported by a tool without changing its source code. To get around this problem a so-called “Lightweight Extension” mechanism was introduced which allowed changes to the UML metamodel to be simulated within what
is otherwise a normal model. In this section we give an overview of the UML 2.1 lightweight extension mechanism and illustrate how it can be used.

### 4.1 Extension Concepts

Lightweight extensions are defined using a combination of three distinct concepts: stereotypes, stereotype properties (also known as tag definitions) and constraints. These are illustrated in Fig. 15.

#### 4.1.1 Stereotypes

Stereotypes are the core of the extension mechanism. A stereotype essentially has the effect of defining a specialization of an element in the metamodel of the base language. Every stereotype must therefore be defined with respect to at least one base element. To reflect the conceptual similarity to the specialization relationship, the extension relationship between a stereotypes and its base class has a very similar form. The only difference is that the triangle at the head of the extension relationship is solid (black) while the triangle at the head of the specialization relationship is empty (white). The left-hand side of Fig. 15 shows an example of a stereotype definition. Here “EJB session” is defined as an extension to the base abstraction “class” from the core UML metamodel in Fig. 1. This has the logical effect of defining a new virtual metaclass which represents a concept in a specific domain.

The right-hand side of Fig. 15 is similar but illustrates the use of a required stereotype. It introduces the abstraction “OCL Constraint” as a mandatory specialization of the class “Constraint” from the core UML metamodel, that is, models using a profile (see below) containing this stereotype definition must only use “OCL Constraints”.

From the language user’s point of view the effect of attaching a stereotype to a base class in the metamodel is to allow instances of that class to be “branded”
In this figure, “ShoppingCart” is a class that has been branded with the stereotype “EJB Session”. This means that it is no ordinary class, but a class representing the notion of “EJB Session” (-bean) which is of importance in the domain of building EJB applications. To a layman such a branding adds little value, but to an expert in the domain branding a class with a domain concept such as “EJB Session” adds a lot of extra information to the model. Model element branding is therefore a good example of “Customized Modeling”—the idea that base language concepts are adapted to make the language more specific and expressive for a certain purpose.

4.1.2 Stereotype Properties

Stereotypes can be used to effectively add specializations of the classes in the metamodel of the base language. However, on their own they cannot add new attributes to the specializations. To do this stereotype properties, also known as tag definitions, must be used. A stereotype property is essentially a new attribute attached to the new conceptual metamodel element represented by the stereotype. The definition of the stereotype “EJB Session” in Fig. 15 includes the definition of two properties “state” and “maxSessionTimeout”. These can be defined using all the normal types for attributes, including the values of an enumeration type as illustrated in Fig. 15. “StateKind” is an enumeration class which defines the values that the property “state” can assume.

The values of stereotype properties at the stereotype usage level are also known as tagged values. They effectively correspond to slots in normal class instances. The difference, as shown in Fig. 16, is that tagged values are defined in a separate model element. Fig. 16 shows the stereotype class “ShoppingCart” with tagged values “stateful” and “600” for its “state” and “maxSessionTimeout” properties respectively. The new UML 2 terminology—"stereotype property" instead of "tag definition"—is a sign of the growing recognition that tag definitions and tagged values are effectively nothing more than attributes and slots, but at the level of the metamodel and model respectively. However, semantically they are still a special extension concept that cannot be fully explained using the attribute/slot analogy. For example, instances of stereotyped classes can not refer to stereotype properties.
4.1.3 Constraints

When used in a lightweight extension, constraints typically define well-formedness rules for model elements branded with stereotypes. The stereotype definition “EJB Session” on the LHS of Fig. 15 features such a constraint definition. It serves to indicate that a well-formed model element branded with this stereotype must not have a tagged value for its “maxSessionTimeOut” tag that exceeds “3599”.

4.2 Profiles

Although all parts of the lightweight extension mechanism can be used individually the intention is that they be used collectively to define a coherent customization for a specific purpose. Such a coherent set of extensions is known as a profile and as with all logically related sets of model elements in the UML they are usually collected into a common package. Figure 17 shows an example of how some of the extensions from Fig. 15 can be combined together into a profile package.

As illustrated in Fig. 17, a profile is always related to a base metamodel which it gains access to by means of the “imports” dependency. As can been seen from Fig. 17, profiles possess both of the key elements of a customized language: they
In order to make the extension available for use in a new model it is necessary to “apply” that profile to that model. Figure 18 illustrates how the extensions in a profile are applied to another model using the “apply” dependency. This effectively makes the customization available for use in the creation of a model, in this case the EJB-specific implementation of the Pet Store example illustrated in Fig. 7.

4.3 Lightweight Extension: Pros and Cons

The lightweight extension mechanism offered by the UML has some advantages and disadvantages compared to a heavyweight approach allowing liberal manipulation of the UML language definition.

4.3.1 Advantages

First, unless a profile selects a subset of the base language or constraints are used to effectively limit or remove the applicability of existing modeling elements, the defined customization will always be a “Conservative Specialization Extension” ensuring maximum upward- and backward-compatibility.

Second, stereotypes can be added and removed from model elements dynamically. Since stereotype applications are treated like annotations of regular model elements
it is possible to change the branding (i.e., stereotypes) of model elements just as easily as it is to change their links and their attribute values. The stereotyped element is unaware whether a particular stereotype is applied to it. In other words, stereotypes represent an implementation strategy for dynamically reclassifying model elements, including the addition/removal of properties.

Third, the profile mechanism allows a model element to be branded by multiple stereotypes, which is equivalent to it having multiple types. The corresponding heavyweight equivalent would be an explicitly defined metaclass which is a specialization of all of the base element classes which the branded model element is intended to be an instance of. While such an explicit combination could be used to resolve any conflicts arising from the combination of various stereotypes, one would have to create one explicit combination class for all possible stereotype combinations and for all base classes to which the stereotype is applicable. In this basic form, this would not be a viable approach because of the combinatorial explosion of cases.

### 4.3.2 Disadvantages

First, even if used to their maximum effect profiles can only support a combination of destructive reduction (through selecting a subset of the metamodel to modify and constrain) and conservative specialization extension (through stereotypes and stereotype properties). The important need to extend the base language with completely new concepts that cannot be explained as specializations of existing ones is not supported. This limits the range of customized languages which may be defined through profiles.

Second, stereotypes and stereotype properties are additional concepts which could often be replaced with metaclasses and their attributes with an additional degree of expressiveness. Metaclasses may inherit from each other and may have associations. Attributes of metaclasses could be specified just like slot values for objects. Apart from the above mentioned dynamic properties and multiple applicability of stereotypes they just represent a lightweight way of metamodeling. The latter could be offered in various forms, allowing various kinds of modifications that yield the same guarantees for extensions as featured by stereotypes.

Third, since stereotype-based branding offers such a handy means of assigning a special status to a model element there is a great deal of confusion in the general modeling community over how exactly they should be used. In fact, as reported in [2], three different usage patterns can be observed in practice. Figure 19(a) illustrates the “official” usage mode in which the stereotype is used to classify the class it stereotypes. Here the concept “taxed” is correctly being used to brand a model element “Animal” as being conceptually of the type “taxed”, i.e., this particular pet
store product kind is associated with a tax rate, not any individual product such as “polly”.

However, an equally frequent usage in practice is to use the guillemot notation to name a generalization (supertype) of the branded class (see Fig. 19(b)). Here the intent of applying the stereotype “priced” to “Animal” is not to define additional properties for “Animal” but to define additional properties for instances of “Animal” such as “polly”, e.g., that they have a “price” slot. Even though stereotypes are not designed for doing this, they are nevertheless often (mis-)used for exactly this purpose. A shorthand notation for specifying supertypes [3], e.g., “⟨priced⟩Animal” would probably reduce the amount of improper stereotype uses of this form.

The third form, which is less frequent, is a mixture of the first two and uses stereotypes for both type and instance classification. Currently, the UML has just one official interpretation for stereotype applications and consequently offers only one notation. In practice, modelers find the need to cover all three cases shown in Fig. 19 and some notational means to distinguish them [2] would greatly reduce the confusion experienced by modelers.

### 5. Customization Support Environments

In Section 3 we described the different forms of language customizations that make sense from a theoretical point of view, and in Section 4 we discussed a way of supporting one of them—namely, the “lightweight extension” approach. How-

**FIG. 19.** Stereotype usage patterns.
ever, we have not yet discussed the issues involved in supporting the others using a “heavyweight” approach. The goal of this section is to address the architecture of tools supporting “heavyweight” modeling and language customizations.

5.1 Multiple Modeling Levels

The first generation of modeling tools was only capable of supporting modeling in one fixed language. An environment intended to support modifications to the modeling language and/or support modeling in completely different modeling languages has to deal with at least three modeling levels, shown in Fig. 20.

The middle level shown in Fig. 20 contains the language definition which can be used to create user models at the bottom level. To support its modification or replacement this level must be user modifiable. As long as modeling environments treated the language as being fixed there was no need to represent the language definition in a modifiable way; the tool builders just chose an arbitrary approach and hardcoded it into the tool. A flexible tool, however, has to support a further level (labeled “meta-language” in Fig. 20) to determine how language definitions may be specified. We have already mentioned that it is beneficial to define a language by using a metamodel. The level labeled “meta-language” contains a model that defines the concepts that can be used to define a language, the MOF being a concrete example. The UML superstructure [17] in turn is an example of content that may reside at the “language” level.

Note that there are two ways of looking at the middle “language” level, depending on whether one regards its contents as instances of the level above or as types for the level below. Figure 21 depicts both viewpoints within the middle level.

The left-hand side of Fig. 21 shows a language definition as an instance of the meta-language and the right-hand side shows the same language definition using the usual notation, i.e., as a number of types describing the contents of the level below. While both views are just different interpretations of the same set of concepts (the language definition) they flag an important issue: can the language definition data (as

![Fig. 20. Three-level language definition stack.](image)
seen through the left-hand side view) be directly used to control (model-)data at the level below or is it necessary to promote the instances to types before they can fulfill this function?

5.2 Two-Level Architectures

The above mentioned “promotion step” is often used in today’s tools since they are often implemented using only two levels (i.e., a type level and an instance level). Consider implementing the three levels shown in Fig. 21 using a typical object-oriented language like Java. The classic approach is to define Java classes for each of the elements in the top level, and to use Java objects to represent the language definition data in the middle level (see Fig. 22(a)).

However, these objects may not be directly used in the style just described to represent and control the data at the “user models” level. This is possible only if the language definition is available as types, i.e., classes (see Fig. 22(b)). To achieve this, the objects need to be “promoted” to types. Figure 23 explicitly illustrates the “promotion step” from objects to types.

We refer to an approach of the kind depicted in Fig. 23 as a “cascading architecture” since two-level implementation “windows” are used repeatedly to cover multiple modeling levels. Each of the tool’s partial architectures covers only two levels, but in combination a hierarchy of multiple levels may be constructed (see
The advantage of this cascading architecture is that one never needs to support more than two levels at the same time and access to or manipulation of data is always very efficient, since the access types (the top level of the two) are known at compile time. The disadvantage, however, is the need to perform the promotion step which represents a similar development overhead to compilation in the context of programming languages (in comparison to direct interpretation). This approach is sometimes referred to as a generative approach since each tool is usually generated automatically from the objects at the immediate level above. A significant number of today’s metamodeling (language customization) tools are based on this architecture, relying on the generation of modeling tools.

5.3 Flattened Architectures

The alternative to the generative, cascading approach is to dispense with the idea of always associating metamodels with the type level (e.g., Java classes) of the sup-
porting implementation technology. This requires that elements of user models will not be instances of metamodel elements anymore in the usual sense (e.g., using the Java object-class relationship). Instead, the “instance of” relationship is just modeled as data. As a result, in our example the user models, the language definition and the relationship between them are just data at the same level.

Figure 24 shows how what we call a “flattened architecture” accommodates two of the three levels of Fig. 20 in just one level. Figuratively speaking, it squeezes two of the levels into one. One of the advantages of this architecture is the ability to change content at the language level and immediately use the new definition for the creation of user models. Furthermore, even when user models may no longer conform to a changed language definition, they are still represented using the common (meta-language) format. In comparison, models represented using a two-level cascading style depend on their language definition to stay fixed. A change to the language implies the need to migrate models represented in the old language definition to the new language definition. Note that in Fig. 24 the role of the “meta-language” level has turned into providing a common repository format, rather than defining the primary concepts with which to build language definitions. It can, however, be used for both roles—the MOF being a prominent example.

A disadvantage of the flattened architecture is reduced performance related to the accessing and manipulation of model data since its definition has to be interpreted. Moreover, access to the data has to occur in a reflective style, using very common access methods, parameterized with (elements of) the language definition. This is less straightforward and cannot be type-checked using the implementation technology (e.g., Java type checking).

Note that the “user models” level may in general be internally structured. Users can model at the type level just as easily as at the instance level. Figure 25 shows how the “user models” level may be subdivided into two such levels.

We did not arrange the “language”, “user types” and “user instance” levels in a linear fashion because the “language” level describes the contents of both “user types” and “user instances” levels. Figure 25 hence describes the OMG’s so-called four-layer architecture, with “meta-language” corresponding to the MOF (M3), “language” to “UML metamodel” (M2) and “user types”/“user instances” to “user mod-
6. Ontological Metalevels

In the previous section we focused on a relationship between models and their (meta-)models which we refer to as linguistic classification. This reflects the widely accepted approach of characterizing metamodels as being language definitions and “user models” as being statements expressed in that language. The “meta-” prefix is motivated by the location of language definitions in the original, stack-oriented visualization of description levels that was put forward by CDIF [15] and then became the foundation of the UML infrastructure [16]. This hierarchy (of which the three top levels are shown in Fig. 20) assumes that “user models” are type models (of user data) and uses the same type-classification to describe “user models” with a (language definition) model. Hence, the language definition apparently represents second order classification with respect to user data, and it is therefore natural to think of it as being “meta” with respect to “user data”. Note that this line of reasoning works well in the CDIF context, since user models are exclusively type models. However, the case of the UML is more involved as already pointed out in the previous section. In the next section we investigate why the term “metamodel” is not as appropriate in the case of the UML [9].

One should always qualify the term “metamodel” since although linguistic metamodeling is important there is an important alternative: ontological metamodeling. Consider Fig. 25, where we have drawn a level boundary between “user types” and “user instances”. Although we have drawn this boundary one would normally not think of a user type model as defining a language of which user instance models are
statements expressed in that language. The underlying meaning of user types is of
course to create domain abstractions in terms of classification. We therefore refer to
the “instance of” relationship between a user instance and a user type as ontological
classification (as opposed to linguistic classification).

Note there is no specific reason why the number of user modeling levels should
be limited to two. It is perfectly possible to envisage three or more ontological
classification levels. Elements at the third level (i.e., the types of the types of the
instances) would warrant the label of metatypes, due to their second order classifi-
cation property. To distinguish such types from “language definition”-types we refer
to the former as ontological metatypes or domain metatypes and the latter as linguis-
tic (meta-)types, since the former are concerned with describing domain knowledge
rather than language features.

Discussing the nature of, and potential support for, ontological metatypes is impor-
tant in the context of this chapter, since language customization is often performed
not to enrich one’s vocabulary of modeling primitives, but to capture ontological
information due to a lack of a better alternative. After the following subsections
which clarify the notion of ontological metatypes and their support we will further
discuss the relationship between ontological (meta-) modeling and language cus-
tomization.

6.1 Ontological Metatypes

A good example of a domain concept which naturally corresponds to the notion
of a metatype is the concept of “ProductType”. This is an important concept in the
Pet Store example because it classifies the different kinds of products that are sold
in the store. Different kinds of products, such as “Animal”, are naturally thought of
as instances of “ProductType”. However, specific product objects, such as individual
animals or concrete food items, could not naturally be thought of as instances of
“ProductType”. For example, it does not make sense to say that “parrotInstance2947”
(alias “Polly”) is a “ProductType”. The most natural relationship between these con-
cepts is illustrated as Fig. 26. Object “parrotInstance2947” is an instance of “Animal”
and “Animal” is an instance of ProductType. Since Fig. 26 shows concepts occupying
three distinct ontological levels, “ProductType” is naturally thought of as an (onto-
logical) metatype of “parrotInstance2947”.

Even though ontological classification hierarchies will naturally be much less deep
than generalization hierarchies, it is easily possible to think of requiring four or more
classification levels. For example, the Pet Store owner may receive frequent requests
for a “parrot just like Polly” or children saying “I want a Lassie”, intending to refer to
a Collie that looks like the famous movie character. The Pet Store design may there-
fore be changed to no longer interpret “Polly” and “Lassie” as referring to individual
animals but to a class of animals that looks like the prototypical instance. In this case, element “Polly” becomes a type for all parrots looking like “Polly”, creating a four-level classification hierarchy.

At this point it is important to address a potential source of confusion—the mixing up of ontological metatypes and supertypes. They are often mixed up because they are both more abstract than the notion that they classify or generalize and in natural language they can both be related to the more concrete notion using the phrase “is-a”. In other words, if in a particular domain it is natural to say “X is a Y”, it is possible that Y may be a metatype or a supertype. However, the right choice can easily be determined with a simple test: If instances of “X” can be considered to be instances of “Y” as well then “Y” is a supertype not a metatype. Consider the situation in Fig. 27.

In this example, the class “Animal” has both a superclass “Product” and a metaclass “ProductType”. As already mentioned above, when discussing the relationships between the concepts in this model in natural language it would be natural to state that “Animal is-a ProductType”. However, it would also be natural to state that “an Animal is-a Product”. So why is one cast as a classifier of “Animal” and the other as a supertype? The answer is hinted at by the slightly different forms of the above statements. In the second we prefixed the statement with the article “an”, while in the first there was no such article. This reflects the fact that in the first case, as already mentioned above, it would not be natural to state that “parrotInstance2947 is-a ProductType” whereas it would be natural to state that “parrotInstance2947 is-a Product”. This indicates that “parrotInstance2947” can be regarded an indirect instance of “Product” as well, as shown in Fig. 27. This therefore makes the set of animals a subset of the set of products and naturally places them in a specialization relationship.
A natural example of a domain with multiple ontological levels and multiple specialization levels is the Linnaean classification system used in Biology. Figure 28 shows how generalization and ontological classification can be used together to place our sample parrot within the kingdom of animals. Of course, this is overkill for the Pet Store application. A cut down version of this scheme, however, could in principle be used to organize the products sold by the store.

Consequently, the architecture shown in Fig. 25 should be extended to look like the architecture in Fig. 29, i.e., it should feature an unbounded number of ontological modeling levels.
6.2 The Dual Facet Property

As soon as more than two ontological levels are used, e.g., as in Fig. 26, an interesting question arises: what exactly is an element (e.g., “Animal”) that is both the instance of a type (“ProductType”) and a type for its instances (“parrotInstance2947”)? The top part of Fig. 30 highlights the two ways one can look at “Animal”.

In comparison to Fig. 26 we have added some sample properties to show that in general an element (here “Animal”) has both slots (“taxRate”) and attributes (“price” and “name”). The topmost occurrence of “Animal” depicts the “instance” role of “Animal” in which it is seen as an instance of its classifier “ProductType”. We have therefore used the UML notation for objects to reinforce this perspective in which “Animal” has a “taxRate” slot with a value of “16” by virtue of the fact that its classifier, “ProductType” has an attribute “taxRate”. This is a normal UML class/object relationship in which “Animal” plays the role of an object.

Just below in Fig. 30 we show the “class” perspective on “Animal” in which it is seen as a classifier for its instances at the level below. We therefore use the UML notion for classes, declaring two attributes for “Animal”, one of type Float and the other of type String. Instances of “Animal” such as “parrotInstance2947” thus have corresponding slots with appropriate values for these types, “2502” and “Polly”, re-
respectively. Again this is a normal UML class/object relationship but in this case “Animal” plays the role of a class.

The bottom part of Fig. 30 shows how both “class” and “object” perspectives on “Animal” can be integrated into a single model element. Note that this is not official UML syntax. The UML’s support for more than two ontological levels is rather limited as described in the following subsections.

Summarizing, when there are more than two ontological classification levels, it is necessary to explicitly recognize that model elements—with the exception of those at the top and bottom levels—have two distinct facets, both of which are a real and meaningful part of the element. This is highlighted in Fig. 31 which shows a clabject, i.e., a combined “class” and “object”. The instance facet of a clabject represents its object-like properties and the type facet its class-like properties.

Note that although we have used the term “perspective” before, the facets of a clabjects are not to be confused with the “instance” and “type” viewpoints shown in Fig. 21. These viewpoints provided views of the same set of data either as instances or as types. The instance and type facet of a clabject are two different entities and a perspective, as mentioned above, may only mask one of them, but does not interpret the same facet either as an instance or a type property. Elements that populate a multi-level modeling environment are, hence, neither classes nor objects as understood in classic two-level modeling environments.

The dual facet property of clabjects is not an issue that occurs in isolation with respect to architectures, such as the one shown in Fig. 29. The dual facet phenomenon arises whenever multiple ontological levels are modeled. The next subsections discuss ways of providing at least three ontological levels in the absence of a supporting architecture such as the one in Fig. 29.

![Fig. 31. Clabjects.](image-url)
6.3 The Item-Descriptor Pattern

Since ontological metatypes such as “ProductType” in Fig. 26 frequently occur in practice modellers have always found ad-hoc ways of modeling them. A well-known workaround that is used in two-level environments is the “Item Descriptor” pattern [6] which is often employed when domain types need to be available as runtime data [7]—that is, when domain types are required to be treated as instances. Sometimes the motivation is to simply have a way to store and change type related data (such as tax rates) [6], sometimes one actually needs to support the creation of new types at runtime [8].

The basic idea of the Item-Descriptor pattern is to represent both application instances and types as objects. Figure 32 shows this idea in the context of our case study.

The “instance of” relationship between “Animal” and “parrotInstance2947” is part of the runtime data and is described by the “instance of” association between “ProductType” and “Product”. Ignoring the type “Product” in Fig. 32—which is just required in order to have a way to produce instances like “parrotInstance2947”, since the object “Animal” cannot do it—the hierarchy between “ProductType”, “Animal”, and “parrotInstance2947” is exactly the same as in Fig. 26, except that the “instance of” relationship between “Animal” and “parrotInstance2947” is user modeled instead of being supported by a multi-level modeling environment. This observation highlights the “workaround” character of the Item-Descriptor pattern which represents a poor man’s way of modeling at three ontological levels. Note how the instance and type facets of “Animal” are split between “Animal” and “Product”. The disadvantages of applying this pattern include the lack of any supported type checking between “Animal” and “parrotInstance2947”, the lack of support for inheritance between modeled types, and the addition of accidental complexity for scenarios that

![Diagram](image)

Fig. 32. Item descriptor pattern.
could be more concisely described using proper support for ontological metamodeling.

6.4 Powertypes

The above subsection described a way to introduce more ontological levels by using a two-level technology. The UML, in fact, has some limited support for dealing with more than two-levels. The best approximation of ontological metamodeling support in the UML is the powertype concept. An example use of a powertype in our Pet Store example is shown in Fig. 33 (which is an excerpt of Fig. 5).

The basic idea behind the powertype concept is to represent the separate facets of a clabject as separate model elements and to establish a constraint between the separate model elements. In Fig. 33, the abstractions that we wish to think of as clabjects (with two facets) are “Animal” and “Food”. The type facet of these abstractions is defined in their common supertype, “Product”, since any attributes and associations defined on “Product” influence the form of their instances. The instance facet of these abstractions, in contrast, is defined in their common classifier, “ProductType”, since any attributes and associations defined in “ProductType” will be possessed by “Animal” and “Food” in the form of slots and links respectively. The classifier of the clabject abstraction(s), in this case “ProductType”, is referred to as the powertype of the supertype (“Product”).

The aforementioned constraint between the elements defining the instance facet (“ProductType”) and the type facet (“Product”) for clabjects is the requirement that every instance of a class designated to be a powertype (“ProductType”) must also be a subclass of another designated class (“ProductType”). This ensures that all instances

![Fig. 33. Powertypes.](image-url)
of a powertype also obtain the desired type facet, which cannot otherwise be achieved in a direct manner.

Note that the desire to influence the type facet of an element arises from the fact that a particular concept (here “ProductType”) not only wants to specify properties of its instances (e.g., “Animal”) but also properties of the instances of its instances, e.g., the fact that “parrotInstance2947” has a “price” slot. One would like to associate this property with the “ProductType” abstraction, since whenever a new instance of “ProductType” is created, this instance needs to be guaranteed to have a “price” attribute. We refer to such requirements as the need for **deep characterization**.

Obviously, when dealing with only two-level scenarios such a need does not arise as the one type level can always control the one instance level. In a multi-level classification hierarchy, however, the limitation of a type to only influence its instances at the intermediate level below, without influence on its instance’s instances, may

![Diagram](image-url)

**Fig. 34.** Deep characterization using powertypes.
become an issue. “Deep characterization” then needs to be addressed, independently of whether workarounds for representing the multiple levels are used or not.

As illustrated in Fig. 34, powertypes support “deep characterization” by requiring that all instances of an ontological metaclass also be subclasses of a particular class. It is this latter superclass which defines the attributes that ensure that all instances of the instance of the powertype abstraction have a particular slot.

6.5 Deep Instantiation

Although the powertype mechanism is able to address “deep characterization” it does not do so in a very elegant way. The basic problem is that in order to define the type facet of a powertype’s instance an extra superclass has to be added. In essence, therefore, two model elements (“ProductType” and “Product”) are being used to capture a single clabject abstraction.

In order to support a more elegant and direct way of representing deep characterization two ingredients are needed. One is a notation which combines the type and instance facets of a clabject in a single model element (see the bottom of Fig. 30). The other is a concept that allows the intended characterization depth of a clabject’s properties to be specified. Figure 35 illustrates our example using the above mentioned ingredients.

The key idea behind the concept of characterization depth is to unify the concepts of “attribute” and “slot” into the notion of a field and to indicate whether a field should be thought of as a slot or an attribute by assigning it a potency value. The potency value of a field indicates how many times it can be instantiated (with respect to instantiation depth) with the understanding that instantiation reduces the potency of a field by one. A field with potency 2 (such as the “price” attribute of “ProductType”) needs to be instantiated twice in succession before it becomes a slot, while a field with potency 1 (such as the “price” attribute of “Animal”) becomes a slot after one instantiation. A field with potency 0 (such as the “price” slot of “parrotInstance2947”) cannot be instantiated at all. A field of potency 1 thus corresponds to a regular attribute and field of potency 0 corresponds to a regular slot.

Potency values may not only be assigned to fields, but also methods, and more importantly here to clabjects. The same principles apply as for fields, so for example “parrotInstance2947” can be instantiated from “Animal” since the latter is an instance of a potency two clabject.

Modeling scenarios such as the one depicted in Fig. 35 using clabjects and potency reduces the number of modeling elements needed to address “deep characterization” thus reducing the accidental complexity in a model.
7. Orthogonal Classification Architecture

In the previous section we introduced the notion of ontological metatypes and explained how they can help model domains with multiple classifications levels. In this section we show how ontological and linguistic metamodeling can be accommodated within a single unified modeling framework.

7.1 Strict Metamodeling

The first and most important issue to consider when defining the architecture of a multi-level modeling environment is how the notion of “level” is actually defined. So far we have appealed to intuition and have also used the “instance of” relationship in a liberal way without specifying whether we mean the “linguistic” or the “ontological” case. Without becoming more precise in this respect and, in particular, giving the term “level” a meaning, it is not possible to unambiguously decide where modeling elements, even fundamental ones such as “Class” or “Object”, reside in a multi-level classification hierarchy.
There are only two basic ways of approaching this issue. One way is to de-emphasize the idea of levels altogether and simply regard all model elements anywhere in the modeling framework as inhabiting some kind of model "soup". With this unstructured approach, there are no rigidly defined levels and model elements are just placed anywhere. The other way is to rigidly enforce the idea of levels and to adopt the notion that every "instance of" relationship crosses a level boundary and that the "instance of" relationships thus implicitly define the hierarchies of levels. In other words, every classifier must be at a different level from its instances—namely at the level immediately above them. This notion, traditionally referred to as "strict metamodeling", is defined as follows:

In an n-level modeling architecture, $M_0, M_1, \ldots, M_{n-1}$, every element of an $M_m$-level model must be an instance of exactly one element of an $M_{m+1}$-level model, for all $0 \leq m < n - 1$, and any relationship other than the "instance of" relationship between two elements $X$ and $Y$ implies that level($X$) = level($Y$).

### 7.2 Ontological versus Linguistic Dimensions

Because of its value in helping to organize the various languages and models involved in model driven development, strict metamodeling has for some time been the underlying organizational principle used in most modeling environments including the UML. However, it has traditionally only been used (at least explicitly) to define and relate the models within a single dimension. In the original four-level modeling hierarchy there was no notion of different kinds of "instance of" relationships, and every new level of classifiers was viewed as being stacked on top of just one other level. There was no notion of levels existing within a single level as implied by the idea of ontological versus linguistic classification. As a result, the interpretation of the four-layer architecture in which user instances (i.e., user model elements representing individuals of the universe of discourse) reside at a level $M_0$, below users types (classifying individuals of the universe of discourse) at level $M_1$, and the latter below a level $M_2$, containing the UML language definition, contradicted the strict metamodeling tenet that all types must be directly at the next level above their instances. User objects such as "polly" at level $M_0$ need to be classified by "Object" ("Instance Specification" since UML 2.0) at level $M_2$. The corresponding "instance of" relationships crossed two levels, violating strictness and questioning the design and meaning of "levels".

However, once one realizes that the "two-level crossing" "instance of" relationships are linguistic "instance of" operations, an arrangement of levels that preserves strictness is easily achieved. Figure 36 shows how one can think of the top language definition layer as spanning all user modeling levels.
One may now argue that the “instance of” relationships within the bottom row of Fig. 36, the ones within the language usage level, break the rules of strictness because they do not cross level boundaries. However, being ontological “instance of” relationships, they form their own separate, orthogonal classification hierarchy. In this way, the notion of strictness is applicable to each (linguistic and ontological) classification hierarchy separately. Figure 36 therefore shows strict metamodeling applied to two dimensions. We refer to an environment like this that retains the strictness doctrine separately in orthogonal modeling dimensions as an Orthogonal Classification Architecture.

In principle the number of levels in either direction is unlimited. However, the number of levels usually needed in practice is small. As discussed in Section 7 above, in the ontological dimension the number of levels rarely rises above four. We will consider how many levels are useful in the linguistic level in the following section.

8. Languages versus Libraries

In this section we consider how the orthogonal classification architecture described above may support “language customization”, the central theme of this chapter.

8.1 Class Libraries

In Section 6, where we introduced the distinction between ontological and linguistic classification, we observed that in general a model element can have one
(or more) superclass(es) as well as an ontological classifier, and that these impart properties to different facets of the model element. Up to now we have regarded the linguistic metamodel (i.e., the language definition) as defining the conceptual building blocks with which a user creates a model of the domain of interest, however, it is also possible to make concepts available to users in terms of predefined classes in the ontological hierarchy.

In fact this is not just a “possibility” it is actually an approach that has been used with great success in mainstream object-oriented environments such as Smalltalk and Java for some time. Most of the abstractions used by a Java programmer when developing a new application actually come from existing class “libraries” rather than from the language itself. The root of the class inheritance hierarchy in every Java program is the class “Object” defined in the core package. Not all of the properties of a Java class are therefore defined in the Java language definition; a number of them are defined in the class “Object” which every class inherits from. However, in order to take advantage of even more predefined functionality or structure, new Java classes are often defined as subclasses of more specialized classes. The standard Java environment consists of a large number of classes in a collection of different packages providing functionality ranging from I/O to database access and GUI’s. In addition, there is a much larger resource of third party class libraries that provide predefined functionality for all imaginable purposes. Moreover, these class libraries can range from being highly general, such as those in the Java standard, to very domain-specific, such as libraries providing financial services or capabilities for different platforms. For examples, at the level of Java code, the EJB platform used in the case study is made available to users in the form of predefined classes not only to use in a client-server fashion but also to inherit from.

In object-oriented programming technology, therefore, libraries play a big role in making domain-customized functionality available in the development of new applications. For some reason, this approach has never been used in the definition of modeling environments. In particular, in the case of the UML, all the predefined modeling abstractions are defined within the metamodel, and there is no notion of predefined classes at the user model level that users can specialize when defining their own classes.

However, there is no good reason why this should not be done. On the contrary, there are many good reasons why it would be advantageous. Figure 37 shows for example, how a (new) class “Animal” can be defined by inheriting from a preexisting class in a predefined library of model elements. Some classes, like “Product”, are specific to the retailing domain, whereas others, like “Object”, are general purpose.

The only limitation of straightforward class libraries of the kind just introduced is that they can only influence the type facet of user model elements as explained in Section 6. This means that they can define properties that instances of the model
elements must have, but they can not specify properties (i.e., slots) for the model elements themselves. However, specifying such properties is easily achieved by introducing a predefined library of model elements at the ontological metalevel as well. In Fig. 38, the model element, “Animal”, receives features from its supertypes (“Product” and “Object”) as well as from its types (“ProductType” and “Class”). The former influence the type facet of “Animal” and the latter the instance facet (and possibly the type facet through the use of potency values higher than “one”).

8.2 Library Customization

In the previous sections the issue of customizing a software engineering environment for a particular domain was always discussed in terms of language customization. However, as we have just seen, libraries can have as big an influence on the building blocks used to describe software applications as a language can. Thus, we also need to consider the role that library customization can play in the overall customization process.
Figure 39 shows the “big picture” with the complete range of elements that provide the building blocks for, and influence the properties of, the elements modeled by the user (in this case “Animal” and “collieInstance6452”). The elements at the top language definition level (here called “repository”, see below) represent the language, per se, while those in the ontological levels below represent the predefined elements at the language user level. Note that predefined elements may appear at any ontological level, including the bottom-most, for cases such as the Boolean values “true” and “false”. As shown in Fig. 39, the predefined modeling elements can be organized into different packages (libraries) reflecting different levels of abstraction and domain-specificity. For example, the web store library contains elements that are specific to the domain of web stores.

In general, therefore the customization of a modeling environment based on an Orthogonal Classification Architecture can be done in one of two places: the language level (top) and the user level (bottom). Within the user level, customization may be achieved at any of the present levels. Traditionally, in the context of modeling languages the emphasis has been on language level customization. However, there are advantages to performing as much of the customization as possible at the user level in terms of the ontological levels. This, of course, requires the core concepts (such as “Class”, “Object”, etc.) to be available as library elements at ontological levels. As these concepts then are no longer needed in the language definition, the latter can be made very small, in fact, it becomes a minimal “repository format” that simply needs to provide a medium for all modeling content in the ontological user levels. As a result, the “language” can become very small and stable, whereas the core modeling concepts become customizable.
Note that if you turn the diagram of Fig. 39 by 90 degrees to the right, and relabel "Repository" with "MOF" then the hierarchy looks a lot like the original four-layer-architecture stack, except that the MOF is not used in its role as a meta-metalanguage at the top, but in its role as a level spanning repository format. However, in the architecture of Fig. 39 no elements in the second (leftmost) ontological level need to classify elements in the bottom (rightmost) ontological level, as was the case in the original four-layer architecture interpretation. There is also no level-spanning language definition, as in the latest four-layer architecture interpretation where $M_2$ spans both user types and instances at level $M_1$. The orthogonal classification architecture in combination with a "library metaphor" for predefining modeling concepts therefore completely abandons the idea of a (fixed) language in the linguistic sense and represents everything (flexibly) as user modeled libraries and models.

9. Transformations

In previous sections we have discussed how to best support the derivation of new languages from existing base languages, but only in terms of the definition of the abstract syntax of the new languages. This is justifiable since the abstract syntax forms the basis for the definition of all other language elements. However, it will not be economically viable to create domain-customized languages if the definition of their semantics is difficult and laborious. Since a practically useful semantics in a tool context must be a translational semantics, i.e., a semantics that is based on mapping a new language onto a target language with known semantics, the ease of creating a semantics boils down to the ease of creating a so called "exogenous" transformation [12]. A comprehensive discussion of the various existing transformation approaches is out of the scope of this chapter but we briefly want to illuminate the implications that the various alternatives to language definition have on the effort required to query the source model for the pieces of information required to perform transformations.

9.1 Model Access

For the purpose of discussing different ways of accessing model elements in order to collect relevant information for transformations it is instructive to think of a model database which is queried for elements in the model. In each of the above applications one wants to identify a precise subset of the full set of elements (no less and no more). In transformations, such a set typically represents all elements that are mapped to the target language in the same way, i.e., by applying the same sub-transformation.
**FIG. 40. Transformation taxonomy.**

**Figure 40** shows a taxonomy of various ways of characterizing such model element subsets.

Most alternatives fit under a common “content-driven” characterization since they use model element values to ascertain whether they belong to a particular subset or not. Note that even an element’s type or supertype can be regarded as a value associated with this element. The taxonomy of **Fig. 40** generalizes the exploitation of these last two values of an element to “is-a” driven since it is possible to phrase, e.g., “Animal is-a ProductType” and “an Animal is-a Product”, and both are ways to reference a set of elements in relation to another element that shapes the elements in question in characteristic ways (through classification or generalization respectively).

In **Fig. 40** “property-driven” refers to the ability to use values from the instance facet of elements, such as slot values in UML objects or tagged values in UML classes. This allows a subset of elements (e.g., those branded with a certain stereotype) to be further subsetted depending on whether or not a certain tagged value is present.
The category “context-driven” refers to the consideration of values of the elements which may be reached from them (excluding types and supertypes). This is useful for identifying source patterns which will be mapped to target patterns.

The one alternative that represents its own top-level branch is “annotation driven”. Annotations, such as marks [11], do not belong to the elements themselves and should be kept independent from them since they will typically change as the target model or the transformation is changed. The underlying assumption here is that a given model will be transformed to many target models depending on choices made for implementation technologies, etc. Even if one manages to use the MDA metaphor of using annotations “thought of as being applied to a transparent layer placed over the model” [11], one will still have to update the bindings of the annotations to the model elements whenever the annotations change. Hence, although the “transparent layer” metaphor allows the source model to be unpolluted by target model details (in the form of annotations), the recurring need to change annotations and their binding to the source model vastly increases the cost of defining transformations. As a result, one should maximize “content driven” alternatives to defining transformations and avoid “annotation driven” ones. In the following, we discuss how the three different approaches to language design address this and other issues related to defining transformations.

9.2 DSL/DSM Paradigm

Domain-specific languages support “type driven” element selections well, since elements are typically classified with a rich domain vocabulary such as “Button, ShortCutButton, MenuButton”, etc. Even if the domain does not stipulate a very fine grained classification it is easy to retrofit it into the domain-specific language in order to support fine-grained guidance for transformations. In contrast, in a classic UML model, classes such as “ShoppingCart” and “Video” would just be known as instances of M2-level element “Class” and typically annotations (in the form of stereotypes or marks) would be used in order to guide the generation tool to e.g., produce a EJB session bean in the one and an EJB entity bean in the other case. In a DSL approach one already has different linguistic classifiers for “ShoppingCart” and “Video” respectively and thus a way to guide the transformation accordingly.

A drawback of the DSL approach regarding transformations is that it makes transformations difficult to reuse. Since any domain-specific language is allowed—and in fact, encouraged—to use a complete new set of modeling concepts, transformations will have to be adapted for the new vocabulary even if the differences are just in the names. This is further testimony to the fact that domain-customized languages should be preferred over orphan domain-specific languages.
9.3 OMG Architecture

The use of a standard modeling language such as UML addresses the above mentioned problem regarding the reuse of transformations by fixing a standardized metamodel. Hence, transformations may use a "linguistic-type driven" approach (see Fig. 40) independently of the current domain the user is working in.

This, however, creates another complication which results from the fact that transformations will typically have to consider both the linguistic and the ontological type for determining the correct target element type. For instance, one may want to guide the transformation of object "Animal" by considering it as an instance of M2 element "Class" but also exploit its ontological type, e.g. element "ProductType". Transformations therefore need to consider both the linguistic and ontological type of an element in order to determine the correct target element.

This example highlights another problem of the OMG approach to defining transformations. It cannot be assumed that useful information to guide transformations will be expressed with a certain mechanism. As we have seen in previous sections, classes may be classified through powertypes or stereotypes, making it difficult to keep transformations simple and reuse them. Moreover, the way to access information, such as type information changes with the level one is assuming. For example, objects have yet another different way of specifying their type. Access to the instance facet of objects (through slots) is different to that of classes (through tagged values). These are symptoms of lost opportunities to unify the ontological levels, ultimately causing unnecessary complications for transformations which cannot be expressed in a level-independent manner.

Furthermore, since ontological levels only implicitly exist within M1, often opportunities for a target independent way of annotating classes are lost. Instead of using stereotypes to assign natural metatypes, such as "ActivityType" or "ProductType", they are typically used to directly indicate the generation of "EJB Session" or "EJB Entity". In fact, the MDA guide[11] even assumes that marks will be used to directly indicate target elements. However, if this approach is followed, marks will have to change each time the target model changes, for instance, due to a change in target platform choice. We have already pointed out that this is to be avoided. It is much better to use domain-specific and target-independent types and, if necessary, keep information about which types need to be mapped to which target element types in an external dictionary.

Finally, although there is no immediate technical reason for it, UML users assume rich support for guiding transformations through stereotypes and little or no support for guiding them through supertypes. This may lead to curious attempts to classify class "Animal" as a "Product" (by using a stereotype) where a subtype relationship would have been correct as explained in Section 4.3.2.
9.4 Library Approach

The infrastructure approach presented in Sections 7 and 8 and its associated way of hosting predefined concepts (presented as the “library approach”) implies an approach to defining transformations that rectifies the problems we have just identified with the other approaches.

Assigning different transformations strategies to different model elements is easy, even though one will often not have the benefit of having a fine-grained linguistic classification as with DSL/DSM approaches. However, when faced with transforming “ShoppingCart” and “Animal” in different ways, for example, one just introduces the ontological metatypes “ActivityType” and “ProductType” respectively. This obviates the need to resort to the undesirable use of annotations.

On the other hand, transformations can always rely on the existence of fixed upper-ontologies (i.e., predefined libraries). Transformations may make use of more specific (meta-)types (such as “ProductType”) but they can always rely on the existence of general ones (such as “Class”). This is not only beneficial for creating reusable transformations but also means that it is not required to introduce domain-specific (meta-)types unless one finds a need for them.

In order to guide transformations there is, therefore, no need to consider both linguistic and ontological dimensions. Transformations using models based on the library approach never need to consult linguistic types, as these are in any case too general to provide any useful information. However, this does not imply a loss of information. For instance, model element “Animal” is characterized as both “Product” and “Object” in the ontological generalization hierarchy, since “Object” is a supertype of “Product”. This makes it easy to refine and thus also reuse existing mappings. A mapping that only exploited the fact that some elements are subtypes of “Object” can be refined to one that exploits the fact that some elements are subtypes of “Product” without implying a shift from linguistic classification to ontological classification. Such a seamless way of adapting transformations is of paramount importance for providing a cost effective way of defining semantics.

Furthermore, the uniform representation of elements in the ontological domain makes it easy to access information in a level independent manner. For instance, product types with a tax rate of 7% may be transformed into different target elements than those with a tax rate of 16%. Accessing these values, however, is no different to accessing the price of animal instances, for example.

Finally, the library approach assigns equal weight to the type- and supertype properties of an element and makes deliberate use of supertype relationships for defining the superstructure itself. The usefulness of using “supertype driven” model element selection is indeed indirectly acknowledged in the MDA guide by a “Model Transformation” example [11, Section 3.10.3].
10. Conclusion

In this chapter we have taken a grand tour of the underlying principles and the practical technological issues involved in supporting a domain-customized language approach to software engineering. We started by outlining the conceptual differences between the notions of abstraction and domain-specificity and showed that they conceptually map out a two dimensional language space. We also explained how the notion of domain-specific languages relates to the notion of domain-customized languages in that the latter are a special form of the former which are explicitly derived from an existing base language. With the notion of an essentially-customized language we have furthermore characterized the optimal compromise regarding the diverging goals of “wide intelligibility” (as achieved by modeling standards) and of “desiring the best possible fit regarding an application domain” (as achieved by DSLs). An essentially-domain-customized language is as domain-specific as necessary and as compatible to its base language as possible.

After laying out these basic principles and showing how we used them to develop models for our running case study—a Pet Store e-commerce system—we went on to outline the range of theoretically possible language derivation types and explained their pros and cons for the end user. We then described in detail the particular lightweight language customization mechanism offered by the UML. Known as the “profile” mechanism, this allows the effects of metamodel specialization to be achieved without actually making any changes directly to the metamodel. The section that followed discussed the range of different architectures that can be used in the construction of DCL support tools, ranging from simple “two-level” environments through multi-tool cascading environments to multi-level, flattened architectures in which one or more modeling levels are flattened into a single modeling level.

We then explored the distinction between ontological metamodeling and linguistic metamodeling and argued that both forms are valuable, despite the fact that ontological metamodeling has been neglected by current mainstream modeling approaches. We then introduced the notion of having multiple ontological modeling levels within a single linguistic level and showed how this naturally led to the notion of the Orthogonal Classification Architecture as a clean way of supporting ontological multi-level modeling. The next section built on this by explaining how customization can be characterized in terms of library specialization as well as language specialization, and explained why it makes sense to realize most of the customization capabilities needed by end users in the former way rather than the latter way. Finally, Section 9 finished with a discussion of how all the various notions discussed before relate to and support the definition of transformations, an essential element of the DCL paradigm.
As the universally accepted general purpose modeling language the UML offers the ideal foundation for the realization of a DCL paradigm of the kind explained in this chapter. However, as pointed out in several sections, there are several places in which the UML language and the UML infrastructure need to be improved. First, the UML infrastructure should be founded on the Orthogonal Classification Architecture and should define most of the predefined modeling constructs in the form of predefined libraries rather than as part of the linguistic metamodel. This will allow a large proportion of language customization to be achieved through the much simpler mechanism of library customization. Second, the language needs to provide clean notational support for the concepts of clabjects and fields and the related mechanism of potency.

We hope that the ideas and suggestions described in this chapter will help the reader derive more benefits from the DCL technologies which are going to become an important part of software engineering in the near future.

REFERENCES


