*2*

## Introduction to DSLs

*Domain-Specific Languages (DSLs) are becoming more and more important in software engineering. Tools are becoming better as well, so DSLs can be developed with relatively little effort. This chapter starts with a definition of important terminology. It then explains the difference between DSLs and general-purpose languages, as well as the relationship between them. I then look at the relationship to model-driven development and develop a vision for modular programming languages which I consider the pinnacle of DSLs. I discuss the benefits of DSLs, some of the challenges for adopting DSLs and describe a few application areas. Finally, I provide some differentiation of the approach discussed in this book to alternative approaches.*

### 2.1 Very Brief Introduction to the Terminology

While we explain many of the important terms in the book as we go along, here are a few essential ones. You should at least roughly understand those right from the beginning.

I use the term *programming language* to refer to general-purpose languages (GPLs) such as Java, C++, Lisp or Haskell. While DSLs could be called programming languages as well (although they are not *general purpose* programming languages) I don’t do this in this book: I just call them DSLs.

I use the terms *model*, *program* and *code* interchangeably because I think that any distinction is artificial: code can be written in a GPL or in a DSL. Sometimes DSL code and program code are mixed, so separating the two makes no sense. If the

|  |  |
| --- | --- |
| distinction is important, I say "DSL program" or "GPL code". |  |
| If I use model and program or code in the same sentence, the model usually refers to the more abstract representation. An example would be: "The program generated from the model is  . . . ".  If you know about DSLs, you will know that there are two main schools: *internal* and *external* DSLs. In this book I only address external DSLs. See Section 2.8 for details.  I distinguish between the execution engine and the target platform. The *target platform* is what your DSL program has to run on in the end and is assumed to be something we cannot change (significantly) during the DSL development process. The *execution engine* can be changed, and bridges the gap between the DSL and the platform. It may be an interpreter or a generator. An *interpreter* is a program running on the target platform that loads a DSL program and then acts on it. A *generator* (aka compiler) takes the DSL program and transforms it into an artifact (often GPL source code) that can run directly |  |
| on the target platform.1. |  |
| A language, domain-specific or not, consist of the following main ingredients. The *concrete syntax* defines the notation with which users can express programs. It may be textual, graphical, tabular or a mix of these. The *abstract syntax* is a data structure that can hold the semantically relevant information expressed by a program. It is typically a tree or a graph. It does not contain any details about the notation – for example, in textual languages, it does not contain keywords, symbols or whitespace. The *static semantics* of a language are the set of constraints and/or type system rules to which programs have to conform, in addition to being structurally correct (with regards to the concrete and abstract syntax). *Execution semantics* refers to the meaning of a program once it is executed. It is realized using the *execution engine*. If I use the term *semantics* without any qualification, I refer to the execution semantics, not the static semantics.  Sometimes it is useful to distinguish between what I call |  |
| *technical* DSLs and *application domain* DSLs2. The distinction |  |
| is not always clear and not always necessary, but generally I consider technical DSLs to be used by programmers and application domain DSLs to be used by non-programmers. This can have significant consequences for the design of the DSL.  There is often a confusion around meta-ness (as in meta |  |
| model) and abstraction. I think these terms are clearly different and I try to explain my understanding here.  The meta model of a model (or program) is a model that defines (the abstract syntax of) a language used to describe a model. For example, the meta model of UML is a model that defines all those language concepts that make up the UML, such as classifier, association or property. So the prefix *meta* can be understood as *the definition of*. The reverse direction of the relationship is typically called *instance of* or *conforms to*. It |  |
| also becomes clear that every meta model is a model3. A model |  |
| *m* can *play the role* of a meta model with regards to a set of other models *O*, where *m* defines the language used to express the models in *O*.  The notion of *abstraction* is different, even though it also characterizes the relationship between two artifacts (programs or models). An artifact *a*1 is more abstract than an artifact *a*2 if it leaves out some of the details of *a*2, while preserving those characteristics of *a*2 that are important for whatever *a*1 is used for – the purpose of *a*1 informs the the abstractions we use to approximate *a*2 with *a*1. Note that according to this definition, *abstraction* and *model* are synonyms: a simplification of reality for a given purpose. In this sense, the term *model* can also be understood as characterizing the relationship between two | . |
| artifacts. *a*1 is a model of *a*2. |  |
| *2.2 From General Purpose Languages to DSLs* |  |

General Purpose Programming Languages (GPLs) are a means for programmers to instruct computers. All of them are Turing complete, which means that they can be used to implement anything that is computable with a Turing machine. It also means that anything expressible with one Turing complete programming language can also be expressed with any other Turing complete programming language. In that sense, all programming languages are interchangeable.

So why is there more than one? Why don’t we program everything in Java or Pascal or Ruby or Python? Why doesn’t an embedded systems developer use Ruby, and why doesn’t a Web developer use C?

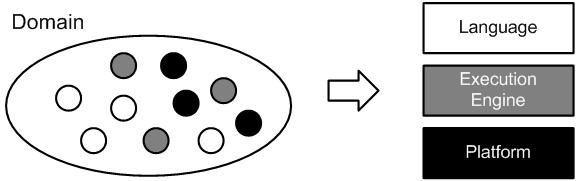
Of course there is the execution strategy. C code is compiled to efficient native code, whereas Ruby is run by a virtual machine (a mix between an interpreter and a compiler). But in principle, you could compile (a subset of) Ruby to native code, and you could interpret C.

The real reason why these languages are used for what they are used for is that the features they offer are optimized for the tasks that are relevant in the respective domains. In C you can directly influence memory layout (which is important when communicating with low-level, memory-mapped devices), you can use pointers (resulting in potentially very efficient data structures) and the preprocessor can be used as a (very limited) way of expressing abstractions with zero runtime overhead. In Ruby, closures can be used to implement "postponed" behavior (very useful for asynchronous web applications); Ruby also provides powerful string manipulation features (to handle input received from a website), and the meta programming facility supports the definition of internal DSLs that are quite suitable for Web applications (the Rails framework is *the* example for that).

So, even within the field of general-purpose programming, there are different languages, each providing different features tailored to the specific tasks at hand. The more specific the tasks get, the more reason there is for specialized languages4.

|  |  |
| --- | --- |
| Consider relational algebra: relational databases use tables, rows, columns and joins as their core abstractions. A specialized language, SQL, which takes these features into account has been created. Or consider reactive, distributed, concurrent systems: Erlang is specifically made for this environment.  So, if we want to "program" for even more specialized environments, it is obvious that even more specialized languages are useful. A Domain-Specific Language is simply a language that is optimized for a given class of problems, called a *domain*. It is based on abstractions that are closely aligned with |  |
| the domain for which the language is built5. Specialized lan- |  |
| guages also come with a syntax suitable for expressing these abstractions concisely. In many cases these are textual notations, but tables, symbols (as in mathematics) or graphics can also be useful. Assuming the semantics of these abstractions is well defined, this makes a good starting point for expressing programs for a specialized domain effectively. |  |

*Executing the Language* Engineering a DSL (or any language) is not just about syntax, it also has to be "brought to life" – DSL programs have to be executed somehow. It is important to understand the separation of domain contents into DSL, execution engine and platform (see Fig. 2.1):

).

* Some concerns are different for each program in the domain

(white circles). The DSL provides tailored abstractions to express this variability concisely.

* Some concerns are the same for each program in the domain (black circles). These typically end up in the platform.
* Some concerns can be *derived by fixed rules* from the program written in the DSL (gray circles). While these concerns are not identical in each program in the domain, they are always the same *for a given DSL program structure*. These concerns are handled by the execution engine (or, in some cases, in frameworks or libraries that are part of the platform).

There are two main approaches to building execution engines: *translation* (aka *generation* or *compilation*) and *interpretation*. The former translates a DSL program into a language for which an execution engine on a given target platform already exists. Often, this is GPL source code. In the latter case, you build a new execution engine (on top of your desired target platforms) which loads the program and executes it directly.

If there is a big semantic gap between the language abstractions and the relevant concepts of the target platform (i.e. the platform the interpreter or generated code runs on), execution may become inefficient. For example, if you try to store and query graph data in a relational database, this will be very inefficient, because many joins will be necessary to reassemble the graph from the normalized tabular structure. As another example, consider running Erlang on a system which only provides heavyweight processes: having thousands of processes (as typical Erlang programs require) is not going to be efficient. So, when defining a language for a given domain, you should be aware of the intricacies of the target platform and the interplay between execution and language design[[1]](#footnote-1).

*Languages versus Libraries and Frameworks* At this point you should to some extent believe that specific problems can be more efficiently solved by using the right abstractions. But why do we need full-blown languages? Aren’t objects, functions, APIs and frameworks good enough? What does creating a *language* add to the picture?

* Languages (and the programs you write with them), are the cleanest form of abstraction – essentially, you add a notation to a conceptual model of the domain. You get rid of all the unnecessary clutter that an API – or anything else embedded in or expressed with a general-purpose language – requires. You can define a notation that expresses the abstractions concisely and makes interacting with programs easy and efficient.
* DSLs sacrifice some of the flexibility to express *any* program (as in GPLs) for productivity and conciseness of *relevant* programs in a particular domain. In that sense, DSLs are limited, or restricted. DSLs may be so restricted that they only allow the creation of correct programs (*correct-byconstruction*).
* You can provide non-trivial static analyses and checks, and an IDE that offers services such as code completion, syntax highlighting, error markers, refactoring and debugging. This goes far beyond what can be done with the facilities provided by general-purpose languages.

*Differences between GPLs and DSLs* I said above that DSLs sacrifice some of the flexibility to express *any* program in favor of productivity and conciseness of *relevant* programs in a particular domain. But beyond that, how are DSLs different from GPLs, and what do they have in common?

The boundary isn’t as clear as it could be. Domain-specificity is not black-and-white, but instead gradual: a language is *more* or *less* domain specific. The following table lists a set of language characteristics. While DSLs and GPLs can have characteristics from both the second and the third columns, DSLs are more likely to have characteristics from the third column.

Considering that DSLs pick more characteristics from the third rather than the second column, this makes designing DSLs a more manageable problem than designing general-pur-

pose languages. DSLs are typically just much smaller and simpler7 than GPLs (although there are some pretty sophisticated DSLs).

There are some who maintain that DSLs are always *declarative* (it is not completely clear what "declarative" means anyway), or that they may never be Turing complete. I disagree. They may well be. However, if your DSL becomes as big and

|  |  |
| --- | --- |
| general as, say, Java, you might want to consider just using |  |
| Java8. DSLs often start simple, based on an initially limited |  |
| understanding of the domain, but then grow more and more sophisticated over time, a phenomenon Hudak notes in his ’96 |  |
| |  |  |  | | --- | --- | --- | |  | **GPLs** | **DSLs** | | **Domain** | large and complex | smaller and well-defined | | **Language size** | large | small | | **Turing completeness** | always | often not | | **User-defined abstractions** | sophisticated | limited | | **Execution** | via intermediate GPL | native | | **Lifespan** | years to decades | months to years (driven by context) | | **Designed by** | guru or committee | a few engineers and domain experts | | **User community** | large, anonymous and widespread | small, accessible and local | | **Evolution** | slow, often standardized | fast-paced | | **Deprecation/incompatible changes** | almost impossible | feasible |   paper9. |  |
| So, then, are Mathematica, SQL, State Charts or HTML actually DSLs? In a technical sense they are. They are clearly optimized for (and limited to) a special domain or problem. However, these are examples of DSLs that pick more characteristics from the GPL column, and therefore aren’t necessarily good examples for the kinds of languages we cover in this book.  *2.3 Modeling and Model-Driven Development*  There are two ways in which the term *modeling* can be understood: descriptive and prescriptive. A *descriptive* model represents an existing system. It abstracts away some aspects and emphasizes others. It is usually used for discussion, communication and analysis. A *prescriptive* model is one that can be |  |
| used to (automatically) construct the target system. It must be much more rigorous, formal, complete and consistent. In the context of this chapter, and of the book in general, we always mean prescriptive models when we use the term model10. Us- |  |

7 Small and simple can mean that the language has fewer concepts, that the type system is less sophisticated or that the expressive power is limited.

ing models in a prescriptive way is the essence of model-driven (software) development (MDSD).

Defining and using DSLs is a flavor of MDSD: we create formal, tool-processable representations of specific aspects of software systems11. We then use interpretation or code gener-

models and code, where code generators either create skeletons into which source code is inserted (directly or via the generation gap pattern), or the arcane practice of pasting C snippets into 300 by 300 pixel sized text boxes in graphical state machine tools (and getting errors reported only when the resulting in-

tegrated C code is compiled). So what really is the difference between programming and (prescriptive) modeling today? The table in Fig. 2.3 contains some (general and broad) statements:

|  |  |  |
| --- | --- | --- |
|  | **Modeling** | **Programming** |
| **Define your own notation/language** | Easy | Sometimes possible to some extent |
| **Syntactically integrate several langs** | Possible, depends on tool | Hard |
| **Graphical notations** | Possible, depends on tool | Usually only visualizations |
| **Customize generator/compiler** | Easy | Sometimes possible based on open compilers |
| **Navigate/query** | Easy | Sometimes possible, depends on IDE and APIs |
| **View Support** | Typical | Almost Never |
| **Constraints** | Easy | Sometimes possible with Findbugs etc. |
| **Sophisticated mature IDE** | Sometimes, effort-dependent | Standard |
| **Debugger** | Rarely | Almost always |
| **Versioning, diff/merge** | Depends on syntax and tools | Standard |

|  |  |
| --- | --- |
| *Why the Difference?* So one can and should ask: why is there a difference in the first place? I suspect that the primary reason is history: the two worlds have different origins and have evolved in different directions.  Programming languages have traditionally used textual concrete syntax, i.e. the program is represented as a stream of characters. Modeling languages traditionally have used graphical notations. Of course there are textual domain-specific languages (and mostly failed graphical general-purpose languages), but the use of textual syntax for domain-specific modeling has only recently become more prominent. Programming languages have traditionally stored programs in their textual, concrete syntax form, and used scanners and parsers to transform this character stream into an abstract syntax tree for further processing. Modeling languages have traditionally used editors that directly manipulate the abstract syntax, and used projec- |  |
| tion to render the concrete syntax in the form of diagrams14. |  |
| |  |  | | --- | --- | | ation to transform those representations into executable code expressed in general-purpose programming languages and the associated XML/HTML/whatever files. With today’s tools it is technically relatively simple to define arbitrary abstractions that represent some aspect of a software system in a meaning- |  | | ful way12. It is also relatively simple to build code generators |  | | that generate the executable artifacts (as long as you don’t need sophisticated optimizations, which can be challenging). Depending on the particular DSL tool used, it is also possible to define suitable notations that make the abstractions easily understandable by non-programmers (for example opticians or thermodynamics engineers).  However, there are also limitations to the classical MDSD approach. The biggest one is that modeling and programming often do not go together very well: modeling languages, environments and tools are distinct from programming languages, environments and tools. The level of distinctness varies, but in many cases it is big enough to cause integration issues that can make adoption of MDSD challenging.  Let me provide some specific examples. Industry has settled on a limited number of meta meta models, EMF/EMOF being the most widespread. Consequently, it is possible to navigate, query and constrain arbitrary models with a common API. However, programming language IDEs are typically *not* built on top of EMF, but come with their own API for representing and accessing the syntax tree. Thus, interoperability between models and source code is challenging – you cannot treat source code in the same way as models in terms of how you access the AST programmatically.  A similar problem exists regarding IDE support for modelcode integrated systems: you cannot mix (DSL) models and (GPL) programs while retaining reasonable IDE support. Again, this is because the technology stacks used by the two are differ- |  | | ent13. These problems often result in an artificial separation of |  |   This approach makes it easy for modeling tools to define *views*, the ability to show the same model elements in different contexts, often using different notations. This has never really been a priority for programming languages beyond outline views, | |  |
| inheritance trees or call graphs.  Here is one of the underlying premises of this book: there should be no difference15! Programming and (prescriptive) |  |

modeling should be based on the same conceptual approach and tool suite, enabling meaningful integration[[2]](#footnote-2). In my experience, most software developers don’t want to model. They

of this idea. want to program, but:

*at different levels of abstraction:* some things may have to be described in detail, low level, algorithmically (a sorting algorithm); other aspects may be described in more high-level terms (declarative UIs)

*from different viewpoints:* separate aspects of the system should be described with languages suitable to these aspects (data structures, persitence mapping, process, UI)

*with different degrees of domain-specificity:* some aspects of systems are generic enough to be described with reusable, generic languages (components, database mapping). Other aspects require their own dedicated, maybe even project-specific DSLs (pension calculation rules).

*with suitable notations,* so all stakeholders can contribute directly

to "their" aspects of the overall system (a tabular notation for testing pension rules)

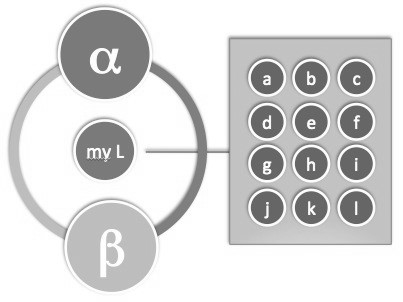
*with suitable degrees of expressiveness:* aspects may be described imperatively, with functions, or other Turing complete formalisms (a routing algorithm), and other aspects may be described in a declarative way (UI structures)

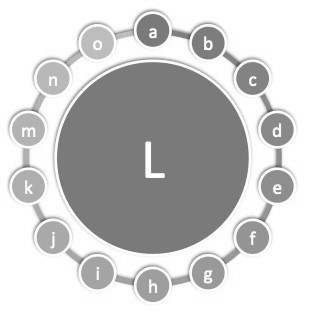
*always integrated and tool processable,* so all aspects *directly* lead to executable code through a number of transformations or other means of execution.

This vision, or goal, leads to the idea of modular languages, as explained in the next section.

### 2.4 Modular Languages

I distinguish between the size of a language and its scope. Language size simply refers to the number of language concepts in that language. Language scope describes the area of applicability for the language, i.e. the size of the domain. The same domain can be covered with big and small languages. A big language makes use of linguistic abstraction, whereas a small language allows the user to define their own in-language abstractions. We discuss the tradeoffs between big and small languages in detail as part of the chapter on Expressivity (Section 4.1), but here is a short overview, based on examples from GPLs.

Examples of big languages include Cobol (a relatively old language intended for use by business users) or ABAP (SAP’s language for programming the R/3 system). Big languages (Fig. 2.4) have a relatively large set of very specific language concepts. Proponents of these languages say that they are easy to learn, since "There’s a keyword for everything". Constraint checks, meaningful error messages and IDE support are relatively simple to implement because of the large set of language concepts. However, expressing more sophisticated algorithms can be clumsy, because it is hard to write compact, dense code.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | the code is necessary to reverse engineer its domain semantics.  There is a third option: modular languages (Fig. 2.6). They are, in some sense, the synthesis of the previous two. A modular language is made up of a minimal language core, plus a library of language modules that can be imported for use in |  |   Let us now take a look at small languages (Fig. 2.5). Lisp or Smalltalk are examples of small GPLs. They have few, but very powerful language concepts that are highly orthogonal, and hence, composable. Users can define their own abstractions. Proponents of this kind of language also say that those are easy to learn, because "You only have to learn three concepts". But it requires experience to build more complex systems from these basic building blocks, and code can be challenging to read because of its high density. Tool support is harder to build because much more sophisticated analysis of a given program. The core is typically a small language (in the way defined above) and can be used to solve any problem at a low level, just as in Smalltalk or Lisp. The extensions then add first class support for concepts that are interesting the target domain. Because the extensions are linguistic in nature, interesting analyses can be performed and writing generators (transforming to the minimal core) is relatively straight- |

|  |
| --- |
| and IDE tooling. Once a language module is imported, it behaves as an integral part of the composed language, i.e. it is integrated with other modules by referencing symbols or by being syntactically embedded in code expressed with another module. Integration on the level of the type system, the semantics and the IDE is also provided. An extension module may even be embeddable in *different* core languages[[3]](#footnote-3). |

forward. New, customized language modules can be built and used at any time. A language module is like a framework or library, but it comes with its own syntax, editor, type system .

* *Language designers define a DSL in three main parts: schema, editor(s) and generator(s).* I agree that ideally a language should be defined "meta model first", i.e. you first define a schema (aka the meta model or AST), and then the concrete syntax (editor or grammar), based on the schema: MPS does it this way. However, I think it is also ok to start with the grammar, and have the meta model derived. This is the typical workflow with Xtext, although it can do both. From the language user’s point of view, it does not make a big difference in most cases.
* *A language workbench can persist incomplete or contradictory information.* I agree. This is trivial if the models are stored in a concrete textual syntax, but it is not so trivial if a persistent representation based on the abstract syntax is used.

|  |  |
| --- | --- |
| This idea isn’t new. Charles Simonyi18 and Sergey Dmitriev19 |  |
| have written about it, so has Guy Steele in the context of Lisp20. The idea of modular and extensible languages also relates very much to the notion of language workbenches as defined by Martin Fowler21. He defines language workbenches as tools where:  • *Users can freely define languages that are fully integrated with each other.* This is the central idea for language workbenches, but also for modular languages, since you can easily argue that each language module is what Martin Fowler calls a language. "Full integration" can refer to referencing as well as embedding, and includes type systems and semantics. |  |
| • *The primary source of information is a persistent abstract representation* and *language users manipulate a DSL through a projectional editor.* This implies that projectional editing must be used22. I don’t agree. Storing programs in their abstract |  |
| representation and then using projection to arrive at an editable representation is very useful, and maybe even the best |  |
| approach to achieve modular languages23. However, in the |  |
| end I don’t think this is important, as long as languages are modular. If this is possible with a different approach, such |  |
| as scannerless parsers24, that is fine with me. |  |

Let me add two additional requirements. For all the languages built with the workbench, tool support must be available: syntax highlighting, code completion, any number of static analyses (including type checking in the case of a statically typed ing the structure of the tokens, thereby

|  |
| --- |
| language) and ideally also a debugger25. A final requirement - |

is that I want to be able to program complete systems within the language workbench. Since in most interesting systems you

will still write parts in a GPL, GPLs must also be available in the environment based on the same language definition/editing/processing infrastructure. Depending on the target domains, this language could be Java, Scala or C#, but it could also be C/C++ for the embedded community. Starting with an existing general-purpose language also makes the adoption of the approach simpler: incremental language extensions can be developed as the need arises.

*Concrete Syntax* By default, I expect the concrete syntax of DSLs to be textual. Decades of experience show that textual syntax, together with good tool support, is adequate for large and complex software systems26. This becomes even more true if you consider that programmers will have to write less code in a DSL – compared to expressing the same functionality in a GPL – because the abstractions available in the languages will be much more closely aligned with the domain. And programmers can always define an additional language module that fits a domain.

Using text as the default does not mean that it should stop there. There are worthwhile additions. For example, symbolic (as in mathematical) notations and tables should be supported. Finally, graphical editing is useful for certain cases. Examples include data structure relationships, state machine diagrams or data flow systems. The textual and graphical notations must be integrated, though: for example, you will want to embed the expression language module into the state machine diagram to be able to express guard conditions.

The need to *see* graphical notations to gain an overview over complex structures does not necessarily mean that the program has to be *edited* in a graphical form: custom visualizations are important as well. Visualizations are graphical representations of some interesting aspect of the program that is read-only, automatically laid out and supports drill-down back to the program (you can double-click on, say, a state in the diagram, and the text editor selects that particular state in the program text).

*Language Libraries* The importance of being able to build your own languages varies depending on the concern at hand. Assume that you work for an insurance company and you want

to build a DSL that supports your company’s specific way of defining insurance contracts. In this case it is essential that the language is aligned exactly with your business, so you have to define the language yourself27. There are other similar ex-

amples: building DSLs to describe radio astronomy observations for a given telescope, our case study language to describe cooling algorithms for refrigerators, or a language for describing telecom billing rules (all of these are actual projects I have worked on).

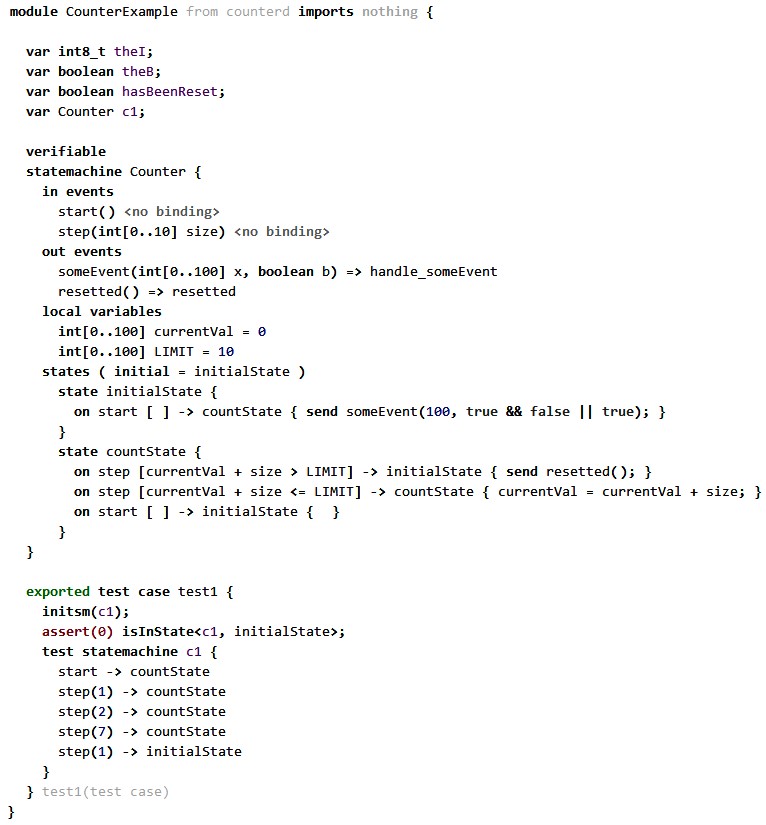
However, for a large range of concerns relating to software engineering or software architecture, the relevant abstractions are well known. They could be made available for reuse (and adaptation) in a library of language modules. Examples include:

* Hierarchical components, ports, component instances and connectors.
* Tabular data structure definition (as in the relational model) or hierarchical data structure definition (as in XML and XML schema), including specifications for persisting that data.
* Definition of rich contracts, including interfaces, pre- and post conditions, protocol state machines and the like.
* Various communication paradigms, such as message passing, synchronous and asynchronous remote procedure calls and service invocations.
* Abstractions for concurrency based on transactional memory or actors.

This sounds like a lot to put into a programming language. But remember: it will not all be in one language. Each of those concerns will be a separate language module that will be used in a program only if needed.

It is certainly not possible to define all these language modules in isolation. Modules have to be designed to work with each other, and a clear dependency structure has to be established. Interfaces on language level support "plugging in" new language constructs. A minimal core language, supporting primitive types, expression, functions and maybe OO, will act as the focal point around which additional language modules are organized.

Many of these architectural concerns interact with frameworks, platforms and middleware. It is crucial that the abstractions in the language remain independent of specific technology solutions. In addition, when interfacing with a specific technology, additional (hopefully declarative) specifications might be necessary: such a technology mapping should be a separate model that references the core program that expresses the application logic. The language modules define a language for specifying persistence, distribution or contract definition. Technology suppliers can support customized generators that map programs to the APIs defined by their technology, taking into account possible additional specifications that configure the mapping28.



*A Vision of Programming* For me, this is the vision of programming I am working towards. The distinction between modeling and programming vanishes. People can develop code using a language directly suitable to the task at hand and aligned with their role in the overall development project. They can

also build their own languages or language extensions, if that makes their work easier. Most of these languages will be relatively small, since they only address one aspect of a system, and typically extend existing languages (Fig. 3.13 shows an example of extensions to C). They are not general-purpose: they are DSLs.

|  |  |
| --- | --- |
| they can become even better, for example in the development of debuggers or integration of graphical and textual languages, but we are clearly getting there.  *2.5 Benefits of using DSLs*  Using DSLs can reap a multitude of benefits. There are also some challenges you have to master: I outline these in the next section. Let’s look at the upside first.  *2.5.1 Productivity*  Once you’ve got a language and its execution engine for a particular aspect of your development task, work becomes much more efficient, simply because you don’t have to do the grunt |  |
| work manually29. This is the most obviously useful if you can |  |
| replace a lot of GPL code with a few lines of DSL code. There are many studies that show that the mere amount of code one has to write (and read!) introduces complexity, independent of what the code expresses, and how. The ability to reduce that amount while retaining the same semantic content is a huge advantage.  You could argue that a good library or framework will do the job as well. True, libraries, frameworks and DSLs all encapsulate knowledge and functionality, making it easily reusable. However, DSLs provide a number of additional benefits, such as a suitable syntax, static error checking or static optimizations and meaningful IDE support.  *2.5.2 Quality* |  |
| Using DSLs can increase the quality of the created product: fewer bugs, better architectural conformance, increased maintainability. This is the result of the removal of (unnecessary) degrees of freedom for programmers, the avoidance of duplication of code (if the DSL is engineered in the right way) and the consistent automation of repetitive work by the execution |  |

Tools for this implementing this approach exist. Of course

|  |  |
| --- | --- |
| engine30. As the next item shows, more meaningful valida- |  |
| tion and verification can be performed on the level of DSL programs, increasing the quality further.  *2.5.3 Validation and Verification*  Since DSLs capture their respective concern in a way that is not cluttered with implementation details, DSL programs are more semantically rich than GPL programs. Analyses are much easier to implement, and error messages can use more meaningful wording, since they can use domain concepts. As mentioned above, some DSLs are built *specifically* to enable non-trivial, formal (mathematical) analyses. Manual review and validation also becomes more efficient, because the domain-specific aspects are uncluttered, and domain experts can be involved more directly.  *2.5.4 Data Longevity*  If done right, models are independent of specific implementation techniques. They are expressed at a level of abstraction that is meaningful to the domain – this is why we can analyze and generate based on these models. This also means that models can be transformed into other representations if the |  |
| need arises, for example, because you are migrating to a new DSL technology. While the investments in a DSL implementation are specific to a particular tool (and lost if you change it), the models should largely be migratable31.  *2.5.5 A Thinking and Communication Tool*  If you have a way of expressing domain concerns in a language that is closely aligned with the domain, your thinking becomes clearer, because the code you write is not cluttered with implementation details. In other words, using DSLs allows you to separate essential from accidental complexity, moving the latter to the execution engine. This also makes team communication simpler.  But not only is using the DSL useful; also, the act of *building* the language can help you improve your understanding of the |  |
| domain for which you build the DSL. It also helps straighten |  |
| out differences in the understanding of the domain that arise from different people solving the same problem in different ways. In some senses, a language definition is an "executable |  |
| analysis model"32. I have had several occasions on which cus- |  |
| tomers said, after a three-day DSL prototyping workshop, that |  |

|  |  |
| --- | --- |
| *2.5.6 Domain Expert Involvement*  DSLs whose domain, abstractions and notations are closely aligned with how domain experts (i.e. non-programmers) express themselves, allow for very good integration between developers and domain experts: domain experts can easily read, and often write program code, since it is not cluttered with implementation details irrelevant to them. And even when do- |  |
| main experts aren’t willing to write DSL code, developers can at least pair with them when writing code, or use the DSL code to get domain experts involved in meaningful validation and reviews (Fowler uses the term "business-readable DSLs" in this case). At the very least you can generate visualizations, reports or even interactive simulators that are suitable for use by domain experts.  *2.5.7 Productive Tooling* |  |
| In contrast to libraries, frameworks, and internal DSLs (those embedded into a host language and implemented with host language abstractions), external DSLs can come with tools, i.e. IDEs that are aware of the language. This can result in a much improved user experience. Static analyses, code completion, visualizations, debuggers, simulators and all kinds of other niceties can be provided. These features improve the productivity of the users and also make it easier for new team members to become productive34. |  |

they had learned a lot about their own domain, and that even if they never used the DSL, this alone would be worth the effort spent on building it. In effect, a DSL is a formalization of the Ubiquitous Language in the sense of Eric Evans’ Domain Driven Design33.

33 E. Evans. *Domain-driven design: tackling complexity in the heart of software*.

#### 2.5.8 No Overhead

If you are generating source code from your DSL program (as opposed to interpreting it) you can use domain-specific abstractions without paying any runtime overhead, because the generator, just like a compiler, can remove the abstractions and generate efficient code. And it generates the same lowoverhead code, every time, automatically. This is very useful in cases where performance, throughput or resource efficiency is a concern (i.e. in embedded systems, but also in the cloud, where you run many, many processes in server farms; energy consumption is an issue these days).

#### 2.5.9 Platform Independent/Isolation

|  |  |
| --- | --- |
|  |  |
| pendent of the target platform36. It is absolutely feasible to change the execution engine and the target platform "underneath" a DSL to execute the code on a new platform. Portability is enhanced, as is maintainability, because DSLs support separation of concerns – the concerns expressed in the DSL (e.g. the application logic) is separated from implementation details and target platform specifics.  Often no single one of the advantages would drive you to using a DSL. But in many cases you can benefit in multiple ways, so the sum of the benefits is often worth the (undoubtedly necessary) investment in the approach.  *2.6 Challenges*  There is no such thing as a free lunch. This is also true for DSLs. Let’s look at the price you have to pay to get all the benefits described above.  *2.6.1 Effort of Building the DSLs* |  |
| Before a DSL can be used, it has to be built37. If the DSL |  |
| has to be developed as part of a project, the effort of building it has to be factored into the overall cost-benefit analysis. |  |
| For technical DSLs38, there is a huge potential for reuse (e.g. a |  |
| large class of web or mobile applications can be described with the same DSL), so here the investment is easily justified. On the other hand, application domain-specific DSLs (e.g. pension plan specifications) are often very narrow in focus, so the in- |  |

In some cases, using DSLs can abstract from the underlying technology platform35. Using DSLs and an execution engine makes the application logic expressed in the DSL code inde-

vestment in building them is harder to justify at first glance. But these DSLs are often tied to the core know-how of a business and provide a way to describe this knowledge in a formal, uncluttered, portable and maintainable way. That should be a priority for any business that wants to remain relevant! In both cases, modern tools reduce the effort of building DSLs considerably, making it a feasible approach in more and more projects.

#### 2.6.2 Language Engineering Skills

Building DSLs is not rocket science. But to do it well requires experience and skill: it is likely that your first DSL will not be great. Also, the whole language/compiler thing has a bad reputation that mainly stems from "ancient times" when tools like lex/yacc, ANTLR, C and Java were the only ingredients you could use for language engineering. Modern language workbenches have changed this situation radically, but of course there is still a learning curve. In addition, the definition of good languages – independent of tooling and technicalities – is not made simpler by better tools: how do you find out which abstractions need to go into the languages? How do you create "elegant" languages? The book provides some guidance in Part II, DSL Design, but it nevertheless requires a significant element of experience and practice that can only be build up over time.

#### 2.6.3 Process Issues

Using DSLs usually leads to work split: some people build the languages, others use them. Sometimes the languages have been built already when you start a development project; sometimes they are built as part of the project. In the latter case especially, it is important that you establish some kind of process for how language users interact with language developers and with domain experts39. Just like any other situation in which 39 For some DSLs, the users of the DSL are the same people who build the DSL (often true for utility DSLs). This is great because there is no communication overhead or knowledge gap between the domain expert (you) and the DSL developer (you). It is a good set of half-baked DSLs, each covering related domains, possibly with overlap, but still incompatible. The same problem can arise with libraries, frameworks or tools. They can all be addressed by governance and effective communication in the team42.

|  |
| --- |
| texts, you have to plan ahead (people, cost, time, skills) for the maintenance phase. A language that is not actively maintained and evolved will become outdated over time and will become a liability. During the phase where you introduce DSLs into an organization especially, rapid evolution based on the requirements of users is critical to build trust in the approach[[4]](#footnote-4).  *2.6.5 DSL Hell*  Once development of DSLs becomes technically easy, there is a danger that developers create new DSLs instead of searching for and learning existing DSLs. This may end up as a large |

|  |  |
| --- | --- |
| one group of people creates something that another group of people relies on, this can be a challenge40. |  |
| *2.6.4 Evolution and Maintenance*  A related issue is language evolution and maintenance. Again, just like any other asset you develop for use in multiple con- |  |

#### 2.6.6 Investment Prison

|  |  |
| --- | --- |
| The more you invest in reusable artifacts, the more productive you become. However, you may also get locked into a particular way of doing things. Radically changing your business may seem unattractive once you’ve become very efficient at the current one. It becomes expensive to "move outside the box". To |  |
| avoid this, keep an open mind and be willing to throw things away and come up with more appropriate solutions.  *2.6.7 Tool Lock-in*  Many of the DSL tools are open source, so you don’t get locked into a *vendor*. But you will still get locked into a *tool*. While it is feasible to exchange model data between tools, there is essentially no interoperability between DSL tools themselves, so the investments in DSL implementation are specific to a single tool.  *2.6.8 Cultural Challenges*  Statements like "Language Engineering is complicated", "Developers want to program, not model", "Domain experts aren’t programmers" and "If we model, we use the UML standard" are often-overheard prejudices that hinder the adoption of DSLs. I hope to provide the factual and technical arguments for fighting these in this book. But an element of cultural bias may still remain. You may have to do some selling and convincing that is relatively independent of the actual technical arguments. Problems like this always arise if you want to introduce something new into an organization, especially if it changes significantly what people do, how they do it or how they interact. A lot has been written about introducing new ideas into orga- |  |
| nizations, and I recommend reading *Fearless Change* by Rising and Manns43 if you’re the person who is driving the introduc- |  |
| tion of DSLs into your organization. |  |

Of course there are other things that can go wrong: your DSL or generator might be buggy, resulting in buggy systems. You might have the DSL developed by external parties, giving away core domain knowhow. The person who built the DSL may leave the company. However, these things are not specific to DSLs: they can happen with anything, so we don’t address them as challenges in the context of DSLs specifically.

*Is it worth it?* Should you use DSLs? The only realistic answer is: it depends. With this book I aim to give you as much help as possible. The better you understand the topic, the easier it is to make an informed decision. In the end, you have to decide for yourself, or maybe ask for the help of people who have done it before.

Let us look at when you should *not* use DSLs. If you don’t understand the domain you want to write a DSL for, or if you don’t have the means to learn about it (e.g. access to somebody who knows the domain), you’re in trouble. You will identify the wrong abstractions, miss the expectations of your future users and generally have to iterate a lot to get it right, making the development expensive44. Another sign of problems is

this: if you build your DSL iteratively and over time and the changes requested by the domain experts don’t become fewer and smaller, and concern more and more detailed points, then you know you are in trouble, because it seems there is no common understanding about the domain. It is hard to write a DSL for a set of stakeholders who can’t agree on what the domain is all about.

Another problem is an unknown target platform. If you don’t know how to implement a program in the domain on the target platform manually, you’ll have a hard time implementing an execution engine (generator or interpreter). You might want to consider writing (or inspecting) a couple of representative example applications to understand the patterns that should go into the execution engine.

DSLs and their tooling are sophisticated software programs themselves. They need to be designed, tested, deployed and documented. So a certain level of general software development proficiency is a prerequisite. If you are struggling with unit testing, software design or continuous builds, then you should probably master these challenges before you address DSLs. A related topic is the maturity of the development process. The fact that you introduce additional dependencies (in the form of a supplier-consumer relationship between DSL de-

velopers and DSL users) into your development team requires that you know how to track issues, handle version management, do testing and quality assurance and document things in a way accessible to the target audience. If your development team lacks this maturity, you might want to consider first introducing those aspects into the team before you start using DSLs in a strategic way – although the occasional utility DSL is the obvious exception.

### 2.7 Applications of DSLs

So far we have covered some of the basics of DSLs, as well as the benefits and challenges. This section addresses those aspects of software engineering in which DSLs have been used successfully. Part IV of the book provides extensive treatment of most of these.

#### 2.7.1 Utility DSLs

One use of DSLs is simply as utilities for developers. A developer, or a small team of developers, creates a small DSL that automates a specific, usually well-bounded aspect of software development. The overall development process is not based on DSLs, it’s a few developers being creative and simplifying their own lives45.

Examples include the generation of array-based implementations for state machines, any number of interface/contract definitions from which various derived artifacts (classes, WSDL, factories) are generated, or tools that set up project and code skeletons for given frameworks (as exemplified in Rails’ and Roo’s scaffolding). The Jnario language for behavior-driven development is discussed as an example of a utility DSL in Chapter 19.

#### 2.7.2 Architecture DSLs

A larger-scale use of DSLs is to use them to describe the architecture (components, interfaces, messages, dependencies, processes, shared resources) of a (larger) software system or platform. In contrast to using existing architecture modeling languages (such as UML or the various existing architecture description languages (ADLs)), the abstractions in an architecture DSL can be tailored specifically to the abstractions relevant to the particular platform or system architecture. Much

|  |  |
| --- | --- |
| more meaningful analyses and generators are possible in this way. From the architecture models expressed in the DSL, code skeletons are generated into which manually written application code is inserted. The generated code usually handles the integration with the runtime infrastructure. Often, these DSLs also capture non-functional constraints such as timing or resource consumption. Architecture DSLs are usually developed during the architecture exploration phase of a project. They can help to ensure that the system architecture is consistently implemented by a potentially large development team. |  |
| For example in AUTOSAR46, the architecture is specified in |  |
| models, then the complete communication middleware for a distributed component infrastructure is generated. Examples in embedded systems in general abound: I have used this approach for a component architecture in software-defined radio, as well as for factory automation systems, in which the distributed components had to "speak" an intricate protocol whose handlers could be generated from a concise specification. Finally, the approach can also be used well in enterprise systems that are based on a multi-tier, database-based distributed server architecture. Middleware integration, server configuration and build scripts can often be generated from relatively concise models.  We discuss an example architecture DSL for distributed, component-based systems as one of the case studies in Part II of the book, and also in Chapter 18. |  |

#### 2.7.3 Full Technical DSLs

For some domains, DSLs can be created that don’t just embody the architectural structure of the systems, but their complete application logic as well, so that 100% of the code can be generated. DSLs like these often consist of several language modules that play together to describe all aspects of the underlying system. I emphasize the word "technical", since these DSLs are used by developers, in contrast to application domain DSLs.

Examples include DSLs for some types of Web application, DSLs for mobile phone apps, as well as DSLs for developing state-based or dataflow-based embedded systems. As an example of this class of DSLs we discuss mbeddr, a set of extensions to C for embedded software development as an example in Part II and in Chapter 20.

#### 2.7.4 Application Domain DSLs

In this case the DSLs describe the core business logic of an application system independent of its technical implementation. These DSLs are intended to be used by domain experts, usually non-programmers. This leads to more stringent requirements regarding notation, ease of use and tool support. These also typically require more effort in building the language, since a "messy" application domain first has to be understood, structured and possibly "re-taught" to the domain experts47.

Examples include DSLs for describing pension plans, a DSL for describing the cooling algorithms in refrigerators, a DSL for configuring hearing aids or DSLs for insurance mathematics.

We discuss the pension plan example in Part II, and discuss a DSL for defining health monitoring applications in Chapter 22.

#### 2.7.5 DSLs in Requirements Engineering

A related topic to application domain DSLs is the use of DSLs in the context of requirements engineering. Here, the focus of the languages is not so much on automatic code generation, but rather on a precise and checkable complete description of requirements. Traceability to other artifacts is important. Often, the DSLs need to be embedded or otherwise connected to prose text, to integrate them with "classical" requirements approaches.

Examples include a DSL for helping with the trade analysis for satellite construction, or pseudo-structured natural language DSLs that assume some formal meaning for domain entities and terms such as *should* or *must*48. We discuss the connection of DSLs and requirements engineering in Chapter 17.

#### 2.7.6 DSLs used for Analysis

Another category of DSL use is as the basis for analysis, checking and proofs. Of course, checking plays a role in all use cases for DSLs – you want to make sure that the models you release for downstream use are "correct" in a sense that goes beyond what the language syntax already enforces. But in some cases, DSLs are used to express concerns in a formalism that lends itself to formal verification (safety, scheduling, concurrency, resource allocation). While code generation is often a part of it, code generation is not the driver for the use of this type of DSL. This is especially relevant in complex technical systems, or in systems engineering, where we look beyond only

software and consider a system as a whole (including mechanical, electric/electronic or fluid-dynamic aspects). Sophisticated mathematical formalisms are used here – I will cover this aspect only briefly in this book, as part of the Semantics chapter (Section 4.1).

#### 2.7.7 DSLs used in Product Line Engineering

At its core, PLE is mainly about expressing, managing and then later binding variability between a set of related products. Depending on the kind of variability, DSLs are a very good way of capturing the variability, and later, in the DSL code, of describing a particular variant. Often, but not always, these DSLs are used more for configuration than for "creatively constructing" a solution to a problem.

Examples include the specification of refrigerator models as the composition of the functional layout of a refrigerator and a cooling algorithm, injected with specific parameter values. We look at the relationship of DSLs and PLE in Chapter 21.

### 2.8 Differentiation from other Works and Approaches

#### 2.8.1 Internal versus External DSLs

Internal DSLs are DSLs that are embedded into general-purpose languages. Usually, the host languages are dynamically typed and the implementation of the DSL is based on meta programming (Scala is an exception here, since it is a statically typed language with type inference). The difference between an API and an internal DSL is not always clear, and there is a middle ground called a *Fluent API*. Let’s look at the three:

* We all know what a regular object-oriented API looks like. We instantiate an object and then call a sequence of methods on the object. Each method call is packaged as a separate statement.
* A fluent API essentially chains method calls. Each method call returns an object on which subsequent calls are possible. This results in more concise code, and, more importantly, by returning suitable intermediate objects from method calls, a sequence of valid subsequent method calls can be enforced (almost like a grammar – this is why it could be considered a DSL). Here is a Java/Easymock example, taken from Wikipedia:
* Fluent APIs are chained sequences of method calls. The syntax makes this obvious, and there is typically no way to change this syntax, as a consequence of the inflexible syntax rules of the host language. Host languages with more flexible syntax can support internal DSL that look much more like actual, custom languages. Here is a Ruby on Rails example49, which defines a data structure (and, implicitly, a

|  |
| --- |
| Collection coll = EasyMock.createMock(Collection.**class**);  EasyMock.expect(coll.remove(**null**)).andThrow(**new** NullPointerException()).  atLeastOnce(); |

database table) for a blog post:

|  |
| --- |
| **class** Post < ActiveRecord::Base  validates :name, :presence => true validates :title, :presence => true,  :length => { :minimum => 5 }  **end** |
| While I recognize the benefits of fluent APIs and internal DSLs, I think they are fundamentally limited by the fact that an important ingredient is missing: IDE support50. In classical inter- | | |  |
| nal DSLs, the IDE is not aware of the grammar, constraints or other properties of the embedded DSL beyond what the type system can offer, which isn’t much in the case of dynamically typed languages. Since I consider IDE integration an important ingredient to DSL adoption, I decided not to cover internal | | |  |
| DSLs in this book51. | | |  |
| *2.8.2 Compiler Construction*  Language definition, program validation and transformation or interpretation are obviously closely related to compiler construction – even though I don’t make this connection explicit in the book all the time. And many of the techniques that are traditionally associated with compiler construction are applicable to DSLs. However, there are also significant differences. | | |  |
| The tools for building DSLs are more powerful and convenient and also include IDE definition52, a concern not typically as- | | |  |
| sociated with compiler construction. Compilers also typically | | |  |

generate machine code, whereas DSLs typically transform to source code in a general-purpose language. Finally, a big part of building compilers is the implementation of optimizations (in the code generator or interpreter), a topic that is not as prominent in the context of DSLs. I recommend reading the "Dragon Book"[[5]](#footnote-5) or Appel’s Modern Compiler Construction[[6]](#footnote-6).

#### 2.8.3 UML

So what about the Unified Modeling Language – UML? I decided not to cover UML in this book. I focus on mostly textual DSLs and related topics. UML does show up peripherally in a couple of places, but if you are interested in UML-based MDSD, then this book is not for you. For completeness, let us briefly put UML into the context of DSLs.

|  |  |
| --- | --- |
| UML is a general-purpose modeling language. Like Java or Ruby, it is not specific to any domain (unless you consider software development in general to be a domain, which renders the whole DSL discussion pointless), so UML itself does |  |
| not count as a DSL55. To change this, UML provides profiles, |  |
| which are a (limited and cumbersome) way to define variants of UML language concepts and to effectively add new ones. It depends on the tool you choose how well this actually works and how far you can adapt the UML syntax and the modeling |  |
| tool as part of profile definition. In practice, most people use only a very small part of UML, with the majority of concepts defined via profiles. It is my experience that because of that, it is much more productive, and often less work, to build DSLs with "real" language engineering environments, as opposed to using UML profiles.  So is UML used in MDSD? Sure. People build profiles and use UML-based DSLs, especially in large organizations where the (perceived) need for standardization is para- mount56.  *2.8.4 Graphical versus Textual*  This is something of a religious war, akin to the statically-typed versus dynamically-typed languages debate. Of course, there is a use for both flavors of notation, and in many cases, a mix is the best approach. In a number of cases, the distinction is even hard to make: tables or mathematical and chemical notations |  |
| are both textual and graphical in nature57. |  |
| However, this book does have a bias towards textual notations, for several reasons. I feel that the textual format is more |  |

generally useful, that it scales better and that the necessary tools take (far) less effort to build. In the vast majority of cases, starting with textual languages is a good idea – graphical visualizations or editors can be built on top of the meta model later, if a real need has been established. If you want to learn more about graphical DSLs, I suggest you read Kelly and Tolvanen’s book *Domain Specific Modeling* [[7]](#footnote-7).

1. e [↑](#footnote-ref-1)
2. s [↑](#footnote-ref-2)
3. . [↑](#footnote-ref-3)
4. . [↑](#footnote-ref-4)
5. 6 [↑](#footnote-ref-5)
6. 8 [↑](#footnote-ref-6)
7. 2008 [↑](#footnote-ref-7)