

DSL Engineering

*Designing, Implementing and Using
Domain-Specific Languages*



Markus Voelter

with Sebastian Benz
Christian Dietrich
Birgit Engelmann
Mats Helander
Lennart Kats
Eelco Visser
Guido Wachsmuth

www.dslengbook.org

Part I

Introduction

1

About this Book

This book is about creating domain-specific languages. It covers three main aspects: DSL design, DSL implementation and software engineering with DSLs. The book looks only at external DSLs and focuses mainly on textual syntax. The book emphasizes the use of modern language workbenches. It is not a tutorial for any specific tool, but of course it provides examples and some level of detail for some of them: Xtext, MPS and Spoofax. The goal of the book is to provide a thorough overview of modern DSL engineering. The book tries to be objective, but it is based on my own experience and opinions.



1.1 Thank you!

Before I do anything else, I want to thank my reviewers. This book has profited tremendously from the feedback you sent me. It is a lot of work to read a book like this concentrated enough to give meaningful feedback. All of you did that. So, thank you very much! Here is the list, in alphabetical order: Alexander Shatalin, Bernd Kolb, Bran Selic, Christa Schwanninger, Dan Ratiu, Domenik Pavletic, Iris Groher, Jean Bezivin, Jos Warmer, Laurence Tratt, Mats Helander, Nora Ludewig, Sebastian Zarnekow, and Vaclav Pech.

A special thank you goes to my girlfriend Nora Ludewig. She didn't just volunteer to provide feedback on the book, she also had to endure all kinds of other discussions around the topic all the time. Thanks Nora!

I also want to thank itemis, for whom I have worked as an independent consultant over the last couple of years. They let me spend some of the contracted time working on the book. It would probably

have come out a year later, if that had not been the case. Thank you!

1.2 *Why this book*

First of all, there is currently no book available that explicitly covers DSLs in the context of modern language workbenches, with an emphasis on textual languages. Based on my experience, I think that this way of developing DSLs is very productive, so I think there needs to be a book that fills this gap. I wanted to make sure the book contains a lot of detail on how to design and build good DSLs, so it can act as a primer for DSL language engineering, for students as well as practitioners. However, I also want the book to clearly show the benefits of DSLs — not by pointing out general truths about the approach, but instead by providing a couple of good examples of where and how DSLs are used successfully. This is why the book has the three parts ~~mentioned above~~: DSL Design, DSL Implementation and Software Engineering with DSLs.

Even though I had written a book on Model-Driven Software Development (MDSD) before ¹, I feel that it is time for a complete rewrite. So if you are among the people who have read the "old" MDSD book, you really should continue reading. This one is very different, but in many ways a natural evolution of the old one. It may gloss over certain details present in the older book, but it will expand greatly on others.

I have learned a lot in the meantime, my viewpoints have evolved and the tools that are available today have evolved significantly as well. The latter is a reflection of the fact that the whole MDSD community has evolved: ten years ago, UML was the mainstay for MDSD, and the relationship to DSLs was not clear. Today, DSLs are the basis for most interesting and innovative developments in MDSD.

¹ T. Stahl and M. Voelter. *Model-Driven Software Development*. Wiley, New York, 2005

1.3 *What you will learn*

The purpose of this book is to give you a solid overview of the state of the art of today's DSLs. This includes DSL design, DSL implementation, and the use of DSLs in software engineering². After reading this book you should have a solid understanding of how to design, build and use DSLs. A few myths (good and bad) about DSLs should also be dispelled in the process.

Part II of the book, the one on DSL implementation, contains a lot of example code. However, this part is *not* intended as a full tutorial for any of the three tools. However, You should get a solid understanding of what these tools — and the classes of tools they stand for — can do for you.

² Each of these topics has a part of the book dedicated to it.

1.4 *Why no Publisher?*

This book is available as a PDF³ or as a printed version from Amazon: . I decided not to go with a publisher and sell the book at a very low price (almost free for the PDF, and as cheap as possible for the printed version).

Here is some background on book writing and making money: unless you are really famous and/or write a book on a mainstream topic, you will get one or two euros for each copy sold. That is, if you go through a "real" publisher. So, if you don't sell books in the tens or hundreds of thousands of copies, the money you can get out of a book directly is really not relevant, considering the amount of work you put into the book.

Going through a publisher will also make the book more expensive for you as the reader, so fewer people will read it. I decided that it is more important for me to reach as many readers as possible. Publishers may help get the book advertised, but in a niche community like DSLs I figure that word of mouth/blog/twitter is likely more useful. So I hope that you (personally :-)) will help me spread word about the book.

Finally, if you think the book is too cheap, you can donate some money via Flattr or Paypal (links for both options are at <http://voelter.de/dslbook>), or buy me a beer/dinner in case we meet one day :-).

³ There is no Kindle version of the book because the layout/figures/code do not translate very well into the Kindle format. However, you can of course read PDFs on a Kindle. I tried using my Nexus 7 tablet to read the book: if you use landscape format, it works reasonably well.

1.5 *About the Cover*

The cover layout resembles Addison-Wesley's classic cover design. I always found this design one of the most elegant book cover designs I have ever seen. The picture of a glider has been chosen to represent the connection to the cover of the original MDSD book, whose English edition also featured a glider⁴.

⁴ The MDSD book featured a Schleicher ASW-27, a 15m class racing glider. This book features a Schleicher ASH-26E, a 18m class self-launching glider.

1.6 *Feedback and Bugs*

Writing a book like this is a lot of work. At some point I ran out of energy and gave up: I just went ahead publishing it. I am pretty confident that there are no major problems left, but I am sure there are many small bugs and problems in the book. Sorry for that! If you find any, please let me know at voelter@acm.org. One of the advantages of an electronic book is that it is feasible to publish new editions relatively frequently. While I will certainly do other stuff in the

near future (remember: I ran out of energy :-)), I will try to publish an updated and bugfixed version relatively soon.

1.7 *Who should Read This Book*

Everybody who has read my original book on Model-Driven Software Development should read this book. This book can be seen as an update to the old one, even though it is a complete rewrite.

On a more serious note, this book is intended for developers and architects who want to implement their own DSLs. I expect solid experience in object oriented programming as well as basic knowledge about functional programming and (classical) modeling. It also helps if readers have heard the word *grammar* and *parser* before, although I don't expect any significant experience with these techniques.

In the MDSD book, there was a chapter of process and organizational aspects. Except for maybe ten pages of process-related topics, this book does not address process and organization aspects at all. There are two reasons for this: one, these things haven't changed much since the old book, and you can read them there. Second, I feel these aspects were the weakest part of the old book, because it is very hard to discuss process and organizational aspects in a general way, independent of a particular context. Any working software development process will work with DSLs. Any strategy to introduce promising new techniques into an organization applies to introducing DSLs. The few *specific* aspects are covered in the aforementioned ten pages at the end of the design chapter.

1.8 *The Structure of this Book*

The rest of this first part is a brief introduction to DSLs. It defines terminology, looks at the benefits and challenges of developing and using DSLs, and introduces the notion of modular languages, **which plays** an important role throughout the book. This first part is written in a relatively personal, some may say, biased style. It presents DSLs based on my experience. It is not a scientific treatment.

Part II is about DSL design. It is a systematic exploration of seven design dimensions relevant to DSL design: expressivity, coverage, semantics, separation of concerns, completeness, language modularization and syntax. It also discusses fundamental language paradigms that might be useful in DSLs and looks at a number of process-related topics. It uses five case studies to illustrate the concepts. It does not at

all deal with implementation issues — we address this in part III.

Part III covers DSL implementation concerns. It looks at syntax definition, constraints and type systems, scoping, transformation and interpretation, debugging and IDE support. It uses examples implemented with three different tools (Xtext, MPS, Spoofax). Part III is not intended as a tutorial for any one of them, but should provide a solid foundation for understanding the technical challenges when implementing DSLs.

Part IV looks at using DSLs in for various tasks in software engineering, among them requirements engineering, architecture, implementation and, a specifically relevant topic, product line engineering. Part IV consists of a set of fairly independent chapters, each illustrating one of the software engineering challenges.

1.9 *How to Read this Book*

I have had a lot of trouble deciding whether DSL design or DSL implementation should go first. The two parts are relatively independent. As a consequence of the fact that the design part comes first, there are some references back to design concerns from within the implementation part. But still, I guess the two parts can be read in any order, depending on what you are more interested in. If you are new to DSLs, I suggest you start with Part III on DSL implementation. Part II about DSL Design is perhaps a bit too abstract and dense if you don't have any hands-on experience with DSLs.

Some of the examples in Part III are quite detailed. We wanted to make sure we didn't skim about relevant details. However, if some parts become too detailed, just skip ahead — usually the details are not important to understand subsequent subsections.

The chapters in Part IV are independent from each other and can be read in any sequence.

Finally, I think you should at least skim the rest of Part I. If you are already versed in DSLs, you may want to skip some sections or just skim over them. But I think it is important to understand where I am coming from to be able to make sense of some of the later chapters.

1.10 *Example Tools*

You could argue that this whole business about DSLs is nothing new. You could always build custom languages using parser generators such as lex/yacc, ANTLR or JavaCC. And of course you are right.

Martin Fowler’s DSL book⁵ emphasizes this aspect.

However, I feel that language workbenches, which are tools to efficiently create, integrate and use sets of DSLs in powerful IDEs, make a qualitative difference. DSL Developers, as well as the people who use the DSLs, are used to powerful, feature-rich IDEs and tools in general. If you want to establish the use of DSLs and you suggest to your users to use `vi` or `notepad.exe`, you won’t get very far in most organizations.

Also, the effort to develop (sets of) DSLs and their IDEs has been significantly reduced by the maturation of language workbenches. This is why in this book I focus on DSL engineering with language workbenches: I focus on IDE development just as much as I focus on language development.

This is not a tool-tutorial book. However, I will of course show how to work with different tools, but this should be understood more as representative examples of different tooling approaches⁶. I tried to use diverse tools for the examples, but for the most part I stuck to tools I happen to know (well) and that have serious traction in the real-world or the potential to do so: Eclipse Modeling + Xtext, JetBrains MPS, SDF/Stratego/Spoofax, and, to some extent, the Intentional Domain Workbench. All except the last one are open source. Here is a brief overview over the tools:

1.10.1 *Eclipse Modeling + Xtext*

The Eclipse Modeling project is an ecosystem or frameworks and tools for modeling, DSLs and all that’s needed or useful around it. It would easily merit its own book (or set of books), so I won’t cover it extensively. I restrict myself to Xtext, the framework for building textual DSLs, Xtend, a Java-like language optimized for code generation, as well as EMF/Ecore, the underlying meta meta model used to represent model data. Xtext may not be as advanced as SDF/Stratego or MPS, but the tooling is very mature and has a huge user community. Also, the surrounding ecosystem provides a huge number of add-ons that support the construction of sophisticated DSL environments. I will briefly look at some of these tools, among them graphical editing frameworks.

1.10.2 *JetBrains MPS*

The Meta Programming System (MPS) is a projectional language workbench, which means that there is no grammar and parser involved. Instead, editor gestures directly change the underlying AST which is projected in a way that looks like text. As a consequence, MPS supports mixed notations (textual, symbolic, tabular, graphical) and a wide range of language composition features. MPS is open source under the Apache 2.0 license, and developed by JetBrains. It is not as

⁵ M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2010

⁶ Consequently I suggest you read the examples for all tools so you appreciate the different approaches of how to solve a common challenge in language design. If you want to learn about one tool specifically, there are likely better tutorials for each of them.

<http://eclipse.org/Xtext>

<http://jetbrains.com/mps>

widely used as Xtext, but supports many advanced features.

1.10.3 *SDF/Stratego/Spoofax*

These tools are developed at the university of Delft in Eelco Visser's group. SDF is a formalism for defining parsers for context free grammars. Stratego is a term rewriting system used for AST transformations and code generation. Spoofax is an Eclipse-based IDE that provides a nice environment for working with SDF and Stratego. It is also not as widely used as Xtext, but it has a number of advanced features regarding language modularization and composition.

<http://strategoxt.org/Spoofax>

1.10.4 *Intentional Domain Workbench*

A few examples will be based on the Intentional Domain Workbench (IDW). This is a commercial product that, like MPS, uses the projectional approach to editing. The IDW has been used to build a couple of very interesting systems that can serve well to illustrate the power of DSLs. The tool is a commercial offering of Intentional Software.

<http://intentsoft.com>

Many more tools exist. If you are interested, I suggest you take a look at the Language Workbench Competition where a number of language workbenches (13 at the time of writing of this book) are illustrated by all implementing the same example DSLs. This provides a good way to compare the various tools.

<http://languageworkbenches.net>

1.11 *Case Studies and Examples*

I strove to make this book as accessible and practically relevant as possible, so I provide a lot of examples. I decided against a single big, running example because (a) it becomes increasingly complex to follow along and (b) fails to illustrate different approaches to solving the same problem. However, we use a set of case studies to illustrate many issues, especially in Part II, DSL design. These examples are introduced below. These are taken from real-world projects.

1.11.1 *Component Architecture*

This language is an architecture DSL used to define the software architecture of a complex, distributed, component-based system in the transportation domain. Among other architectural abstractions, the DSL supports the definition of components and interfaces as well as the definition of systems, which are connected instances of components. The code below shows interfaces and components. An interface

is a collection of methods (not shown) or collections of messages. Components then provide and require ports, where each port has a name, an interface an optionally, a cardinality.

```
namespace com.mycompany {
  namespace datacenter {
    component DelayCalculator {
      provides aircraft: IAircraftStatus
      provides console: IManagementConsole
      requires screens[0..n]: IInfoScreen
    }
    component Manager {
      requires backend[1]: IManagementConsole
    }
    interface IInfoScreen {
      message expectedAircraftArrivalUpdate( id: ID, time: Time )
      message flightCancelled( id: ID )
    }
    interface IAircraftStatus ...
    interface IManagementConsole ...
  }
}
```

The next piece of code shows how these components can be instantiated and connected with each other.

```
namespace com.mycompany.test {
  system testSystem {
    instance dc: DelayCalculator
    instance screen1: InfoScreen
    instance screen2: InfoScreen
    connect dc.screens to (screen1.default, screen2.default)
  }
}
```

Code generators generate code that acts as the basis for the implementation of the system, as well as all the code necessary to work with the distributed communication middleware. It is used by software developers and architects and implemented with Eclipse Xtext.

1.11.2 Refrigerator Configuration

This case study describes a set of DSLs for developing cooling algorithms in refrigerators. Three languages are used. The first one describes the logical hardware structure of refrigerators. The second one describes cooling algorithms in the refrigerators using a state-based, asynchronous language. Cooling programs refer to hardware features and they can access the properties of hardware elements from expressions and commands. The third one is used to test cooling programs. These DSLs are used by thermodynamicists and are implemented with Eclipse Xtext.

The code below shows the hardware structure definition in the refrigerator case study. An appliance represents the refrigerator. It consists mainly of colling compartments and compressor compartments. A cooling compartment contains various building blocks that are important to the cooling process in a refrigerator. A compressor compartment contains the cooling infrastructure itself, e.g., a compressor and a fan.

```

appliance KIR {

  compressor compartment cc {
    static compressor c1
    fan ccfan
  }

  ambient tempensor at

  cooling compartment RC {
    light rclight
    superCoolingMode
    door rcdoor
    fan rcfan
    evaporator tempensor rceva
  }
}

```

The code below shows a simple cooling algorithm. Cooling algorithms are state based programs. States can have entry actions and exit actions. Insider a state, we check whether certain conditions are true, and then change the status of various hardware building blocks or change the state. It is also possible to express deferred behavior with the **perform ...after** keyword.

```

program Standardcooling for KIR {

  start:
  entry { state noCooling }

  state noCooling:
    check ( RC->needsCooling && cc.c1->standstillPeriod > 333 ) {
      state rcCooling
    }
    on isDown ( RC.rcdoor->open ) {
      set RC.rcfan->active = true
      set RC.rclight->active = false
      perform rcFanStopTask after 10 {
        set RC.rcfan->active = false
      }
    }

  state rcCooling:
    ...
}

```

Finally, the following code is a test script to test cooling programs. It essentially **stimulates** a cooling algorithm by changing hardware properties and then asserting that the algorithm reacts in a certain way.

```


cooling test for Standardcooling {
  prolog {
    set cc.c1->standstillPeriod = 0
  }
  // initially we are not cooling
  assert-currentstate-is noCooling
  // then we say that RC needs cooling, but
  // the standstillPeriod is still too low.
  mock: set RC->needsCooling = true
  step
  assert-currentstate-is noCooling
  // now we increase standstillPeriod and check
  // if it now goes to rcCooling
  mock: set cc.c1->stehzeit = 400
  step
  assert-currentstate-is rcCooling
}

```

1.11.3 *mbeddr C*

This case study (also covered in its own chapter in Part IV of the book) covers a set of extensions to the C programming language tailored to embedded programming, developed as part of mbeddr.com⁷. Extensions include state machines, physical quantities, tasks, interfaces and components. Higher level DSLs are added for specific purposes. An example used in a showcase application is the control of a Lego Mindstorms robot. Plain C code is generated and subsequently compiled with GCC or other, target device specific compilers. The DSL is intended to be used by embedded software developers and it is implemented with MPS.

⁷ <http://mbeddr.com>




```
module DecisionTableExample from cdesignpaper.gswitch imports nothing {
  enum mode { MANUAL; AUTO; FAIL; }

  mode nextMode(mode mode, int8_t speed) {
    return mode, FAIL
    

|            |                |              |
|------------|----------------|--------------|
|            | mode == MANUAL | mode == AUTO |
| speed < 30 | MANUAL         | AUTO         |
| speed > 30 | MANUAL         | MANUAL       |



  } nextMode (function)
}
```

Figure 1.1: A simple C module with an embedded decision table. This is a nice example of MPS' capability to use non-textual notations thanks to its projectional editor (which we describe in detail in the Part III of the book).



```
module Units from cdesignpaper.units imports nothing {
  unit kg for int8_t
  unit lb for int8_t
  kg/int8_t addWeights(kg/int8_t w1, kg/int8_t w2) {
    return w1 + w2;
  } addWeights (function)
  void addWeights (function) {
    addWeights(10kg, 20lb);
    addWeights(10kg, 20kg);
  } someClientCode (function)
  exported test case simpleUnits {
    kg/int8_t m1 = 10kg;
    lb/int8_t m2 = 10lb;
    assert(0) m1 + 10kg == 20kg;
  } simpleUnits(test case)
}
```

Figure 1.2: This extension to C supports working with physical units (such as kg and lb). The type system has been extended to include type checks for units. The example also shows the unit testing extension.



```

module StateMachine from cdesignpaper.statemachine imports nothing {

  statemachine Counter {
    in events
      start() <no binding>
      step(int[0..10] size) <no binding>
    out events
      << ... >>
    local variables
      int[0..10] currentVal = 0
      int[0..10] LIMIT = 10
    states ( initial = start )
      state start {
        on start [ ] -> countState { }
      }
      state countState {
        on step [currentVal + size > LIMIT] -> start { }
        on step [currentVal + size <= LIMIT]
          -> countState { currentVal = currentVal + size; }
        on start [ ] -> start { }
      }
  }
}

```

```

var Counter c1;

void aFunction() {
  trigger(c1, start);
  trigger(c1, step(1));
} aFunction (function)

```

Figure 1.3: This extension shows a state machine. Notice how regular C expressions are used in the guard conditions of the transitions. The inset code shows how the state machine can be triggered from regular C code.

1.11.4 Pension Plans

This DSL is used to efficiently describe families of pension plans for a large insurance company. The DSL supports mathematical abstractions and notations to allow insurance mathematicians to express their domain knowledge directly (Fig. 1.4), as well as higher level pension rules and unit tests using a table notation (Fig. ??). A complete Java implementation of the calculation engine is generated. It is intended to be used by insurance mathematicians and pension experts. It has been built by Capgemini with the Intentional Domain Workbench.

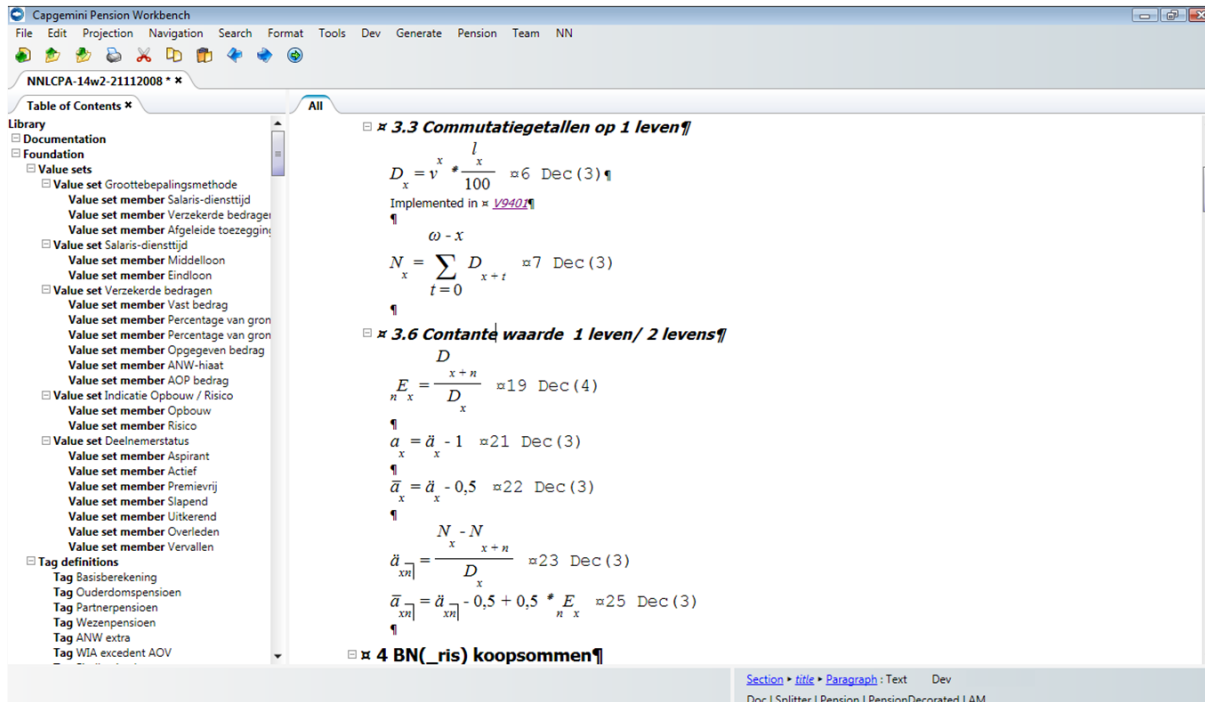


Figure 1.4: Example Code written using the Pension Plans language. Notice the mathematical symbols used to express insurance mathematics.

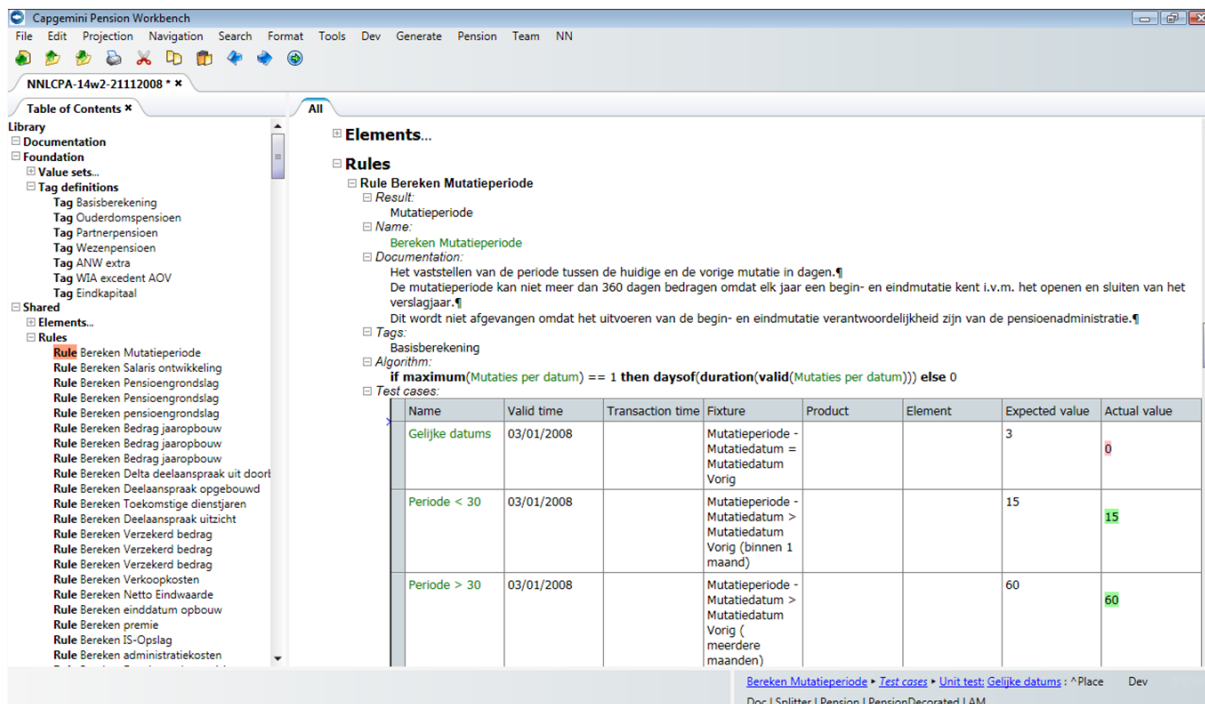



Figure 1.5: This example shows high-level business rules, together with a tabular notation for unit tests. The prose text is in Dutch, but it is not important to be able to understand it in the context of this book.

1.11.5 WebDSL

WebDSL is a language for web programming⁸ that integrates languages the different concerns of web programming, including persistent data modeling (entity), user interface templates (define), access control⁹, data validation¹⁰, search, and more. The language enforces inter-concern consistency checking, providing early detection of failures¹¹. The fragments in Fig. 1.6 and Fig. 1.7 show a data model, user interface templates, and access control rules for posts in a blogging application. WebDSL is implemented with Spoofax and is used in the researchr digital library (<http://researchr.org>).



```
entity Post {
  key      :: String (id)
  blog     → Blog
  urlTitle :: String
  title    :: String (searchable)
  content  :: WikiText (searchable)
  public   :: Bool (default=false)
  authors  → Set<User>
  function isAuthor(): Bool {
    return principal() in authors;
  }
  function mayEdit(): Bool {
    return isAuthor();
  }
  function mayView(): Bool {
    return public || mayEdit();
  }
}
```

```
access control rules
rule page post(p: Post, title: String) {
  p.mayView()
}
rule template newPost(b: Blog) {
  b.isAuthor()
}
section posts
define page post(p: Post, title: String) {
  title{ output(p.title) }
  bloglayout(p.blog){
    placeholder view { postView(p) }
    postComments(p)
  }
}
define permalink(p: Post) {
  navigate post(p, p.urlTitle) { elements }
}
```

⁸ E. Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007

⁹ D. M. Groenewegen and E. Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In *ICWE*, pages 175–188, 2008

¹⁰ D. Groenewegen and E. Visser. Integration of data validation and user interface concerns in a dsl for web applications. *SoSyM*, 2011

¹¹ Z. Hemel, D. M. Groenewegen, L. C. L. Kats, and E. Visser. Static consistency checking of web applications with WebDSL. *JSC*, 46(2):150–182, 2011

Figure 1.6: Example Code written in WebDSL. The code shows data structures and utility functions.

Figure 1.7: More WebDSL example code. This one shows access control rules as well as a page definition.

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 the 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 more or less 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) **as part of** 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, and then acts on** a DSL program. 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

I use the term *processor* to refer to any program that processes programs expressed with a DSL. It may be the a constraint checker, a type inference engine, a code generator, a model transformation, an interpreter, or a sophisticated analysis tool.

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 those. 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 **a 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* DSLs (sometimes also called business DSLs, vertical DSLs, or "fachliche DSLs" in German). 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.

¹ In an example from enterprise systems, the platform could be JEE and the execution engine could be an enterprise bean that runs an interpreter for a DSL. In embedded software, the platform could be a real-time operating system, and the execution engine could be a code generator that maps a DSL to the APIs provided by the RTOS.

2.2 From General Purpose Languages to DSLs

General Purpose Programming Languages (GPLs) are 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 then, 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 down 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 (important to communicate 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), it provides powerful string manipulation features (to handle input received from a website) and the meta programming facility supports the definition of internal DSLs that can be *very suitable* for Web applications (the Rails framework is *the* example for that).

So, even within the field of general-purpose programming there are different ones, each providing different features tailored to the specific tasks at hand². The more specific the tasks get, the more reason there is for specialized languages. 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 built. Specialized languages also come with a syntax suitable to con-

² We do this is real live as well! I am sure you have heard about the eskimos who have 13 different words for snow, because this is relevant in their "domain". They have tailored their language for their environment.

SQL has tables, rows and columns, Erlang has lightweight tasks, message passing and pattern matching.

cisely express these abstractions. 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 effectively expressing programs for a specialized domain.

■ **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. A domain typically consists of three different kinds of **concerns** (see Fig. 2.1):

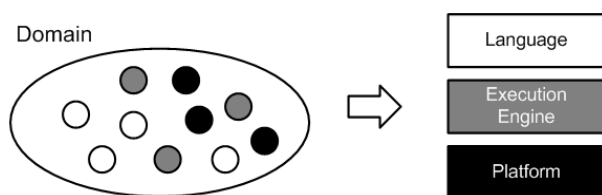


Figure 2.1: Fixed domain concerns end up in the platform, variable concerns end up in the DSL. Those concerns that can be derived by rules from the DSL program end up in the execution engine.

- 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 (grey 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.



There are two main approaches for building execution engines: translation (aka **generation or compilation**) and interpretation. The former translates a DSL program to 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 heavy-weight 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 design³.

■ *Languages vs. Libraries and Frameworks* By now you should believe to a certain extent 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 concisely expresses the abstractions 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-by-construction**).
- You can provide non-trivial static analyses and checks and provide an IDE that provides 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 for 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 rather 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

³ This may sound counter-intuitive. Isn't a DSL supposed to abstract away from just these details of execution? Yes, but: it has to be possible to implement a reasonably efficient execution engine. DSL design is a compromise between appropriate domain abstractions and the ability to get to an efficient execution. A good DSL allows the DSL *user* to ignore execution concerns, but allows the DSL *implementor* to implement a reasonable execution engine

In the end, this is what allows DSLs to be used by non-programmers, one of the value propositions of DSLs: they get a clean, custom, productive environment that allows them to work with languages that are closely aligned with the domain they work in.

and the third columns, DSLs are more likely to have characteristics from the third column.



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

Consider that DSLs pick more characteristics from the third rather than the second column, this makes designing DSLs a more manageable problem than designing general purpose languages. DSLs are typically just much smaller and simpler⁴ 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 Java. 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 paper⁵.

So, then, are Mathematica, SQL, State Charts or HTML 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.

Figure 2.2: Domain-specific languages versus programming languages. DSLs tend to pick more characteristics from the third column, GPLs tend to pick more from the second.

⁴Small and simple can mean that the language has fewer concepts, that the type system is less sophisticated or that the expressive power is limited

Alternatively, if your tooling allows it, extending Java with domain-specific concepts.

⁵P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996

You may know the saying in artificial intelligence (AI) that "everything that works in practice isn't called AI anymore". Ira Baxter suggests that this is also true for DSLs: as soon as a DSL is really successful, we don't call them DSLs anymore.

2.3 Modeling and Model-Driven Development

There are two ways the term **modeling** can be understood: descriptive and prescriptive. A *descriptive* model represents an existing system. It abstracts away certain aspect, 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 model⁶. Using models in a prescriptive way is the essence of model-driven (software) development (MDSD).

⁶Some people say that models are always descriptive, and once you become prescriptive, you enter the realm of programming. That's fine with me. As we will see, I don't distinguish between programming and modeling, just between more or less abstract languages and models.

Defining and using DSLs is a flavor of MDSD: we create formal, tool-processable representations of certain aspects of software systems⁷. We then use interpretation or code generation 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 to represent some aspect of a software system in a meaningful way⁸. 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 does 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 one. 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 the same way as models with respect to how you access the AST programmatically.

A similar problem exists regarding IDE support for model-code 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 different⁹. These problems often results in an artificial separation of 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 300x300 pixels sized text boxes in graphical state machine tools (and getting errors reported only once the resulting, integrated 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:

■ *Why the Difference?* So one can and should ask: why is there a difference in the first place? I guess the primary reason is history,

⁷ One can also do MDSD without DSLs by, for example, generating code from general purpose modeling languages such as UML.

⁸ Designing a *good* language is another matter — Part II DSL Design provides some input on designing DSLs

⁹ Of course, an integration can be created as Xtext/Xtend/Java shows. However, this is a *special* integration with Java. Interoperability with, say, C code, would require a new and different integration infrastructure.

	Modeling	Programming
Define your own notation and language	Easy	Sometimes Possible to some Extent
Syntactically integrate several languages	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
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

the two worlds have different origins and have evolved in different directions.

Programming languages have traditionally been using textual concrete syntax, i.e. the program is represented as a stream of characters. Modeling languages traditionally have been using graphical notations. Of course there are textual domain specific languages (and mostly failed graphical general purpose programming 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 to an abstract syntax tree for further processing. Modeling languages have traditionally used editors that directly manipulate the abstract syntax, and use projection to render the concrete syntax in the form of diagrams¹⁰. 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 difference!¹¹. Programming and (prescriptive) modeling should be based on the same conceptual approach and tool suite, enabling meaningful integration. In my experience, most software developers don't want to model. They 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, persistence mapping, process, UI)

with different degrees of domain-specificity: some aspects of systems are generic enough so they can be described with reusable, generic languages (components, database mapping). Other aspects require

Figure 2.3: Comparing Modeling and Programming

¹⁰ This is not something we think about much. To most of us this is obvious. If it were different, we'd have to define grammars that can parse two-dimensional graphical structures. While this is possible, it has never caught on in practice.

¹¹ This is my personal opinion. While I know enough people who share it, I also know people who disagree

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)

and 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 [DSL design chapter on Expressivity \(Section ??\)](#), 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.

Let us now take a look at small languages. Lisp or Smalltalk are examples of small GPLs. They have few, but very powerful language concepts that are highly composable. Users can define their own abstractions. Proponents of this kind of language also say that those are

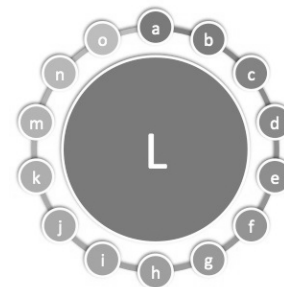


Figure 2.4: A Big Language has many very specific language concepts

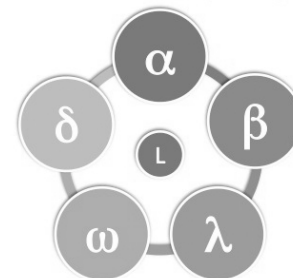


Figure 2.5: A Small Language: Few, but powerful language concepts

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 the code is necessary to reverse engineer its domain semantics.

Let us look at a third option, modular languages. They are, in some sense, the synthesis of the previous two. A modular language is made of a minimal language core, plus a library of language modules that can be imported for use in 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 first class concepts, interesting analyses can be performed and writing generators **(transforming to the minimal core)** is relatively straight 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, 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 languages¹².

This idea isn't new. Charles Simonyi¹³ and Sergey Dmitriev¹⁴ have written about it, so has Guy Steele in the context of Lisp¹⁵. The idea of modular and extensible languages also relates very much to the notion of language workbenches as defined by Martin Fowler¹⁶. 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 used¹⁷. 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 languages. However, **in the end I don't care**, as long as languages are

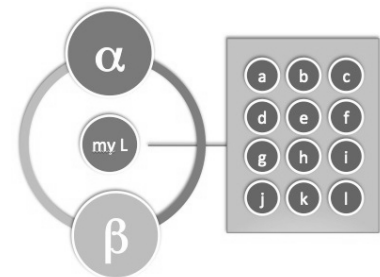


Figure 2.6: A Modular Language: A small core, and a library of reusable language modules

¹² This may sound like internal DSLs. However, static error checking, static optimizations and IDE support is what differentiates this approach from internal DSLs.

¹³ C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOP-SLA*, pages 451–464, 2006

¹⁴ S. Dmitriev. Language oriented programming: The next programming paradigm, 2004

¹⁵ G. L. S. Jr. Growing a language. *lisp*, 12(3):221–236, 1999

¹⁶ M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2010

¹⁷ Projectional editing means that users don't write text that is subsequently parsed. Instead, user interactions with the concrete syntax directly change the underlying abstract syntax. We'll discuss this technology much more extensively in Part III of the book.

modular. If this is possible with a different approach, such as scannerless parsers¹⁸, that is fine with me.

- **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, **it is also okay for me** to start with the grammar, and have the meta model derived (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, it is not so trivial if a persistent representation based on the abstract syntax is used.

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 case of a statically typed language) and ideally also a debugger. A central idea of language workbenches is that language definition always includes IDE definition. The two should be integrated.

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 systems¹⁹. 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 test 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

¹⁸ Scannerless parsers do not distinguish between recognizing tokens and parsing the structure of the tokens, thereby avoiding certain problems with grammar composability. We'll discuss this further in the book as well

¹⁹ As I said before, this is not true for all formalisms. Expressing hierarchical state charts textually can be a challenge. However, textual syntax is a good *default* that can be used unless otherwise indicated

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 **layouted** 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 exactly aligned with your business, so you have to define the language yourself²⁰. There are other similar examples: building DSLs to describe radio astronomy observations for a given telescope, the 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

²⁰ Examples concepts in the insurance domain include native types for dates, times and time periods, currencies, support for temporal data and the supporting operators, as well as business rules that are "polymorphic" regarding their period of applicability

This sounds like a lot of stuff 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 defined. **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 mapping. This is a little bit like service provider interfaces (SPIs) in Java enterprise technology.

■ *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 the role in the overall development effort. 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. They are not general-purpose. They are DSLs.

Tools for this implementing this approach exist. Of course they can become even better, for example, regarding development of debuggers or regarding integration of graphical and textual languages, but we are clearly getting there.

MPS is one of them, which is why I focus a lot on MPS in this book. Intentional's Domain Workbench is another one. Various Eclipse-based solutions are getting there as well

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. But


```

module CounterExample from counterd imports nothing {

  var int8_t theI;
  var boolean theB;
  var boolean hasBeenReset;
  var Counter c1;

  verifiable
  statemachine Counter {
    in events
      start() <no binding>
      step(int[0..10] size) <no binding>
    out events
      someEvent(int[0..100] x, boolean b) => handle_someEvent
      resetted() => resetted
    local variables
      int[0..100] currentVal = 0
      int[0..100] LIMIT = 10
    states ( initial = initialState )
      state initialState {
        on start [ ] -> countState { send someEvent(100, true && false || true); }
      }
      state countState {
        on step [currentVal + size > LIMIT] -> initialState { send resetted(); }
        on step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; }
        on start [ ] -> initialState { }
      }
  }

  exported test case test1 {
    initsm(c1);
    assert(0) isInState<c1, initialState>;
    test statemachine c1 {
      start -> countState
      step(1) -> countState
      step(2) -> countState
      step(7) -> countState
      step(1) -> initialState
    }
  } test1(test case)
}

```

Figure 2.7: A program written in a modularly extendible C (from the mbeddr.com project)

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 manually²¹. This is the most obviously useful if you can replace a lot of GPL code with a few statements in the DSL. There are many studies that show that just the 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 DSL all encapsulate knowledge and functionality, making it easily reusable. However, DSLs provide a number of additional benefits, such as a suitable syntax, a static error checking or static optimizations.

²¹ Presumably, the amount of DSL code you have to write is much less than what you'd have to write if you used the target platform directly.

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 engine)²². As the next item shows, more meaningful validation and verification can be performed on the level of DSL programs, increasing the quality further.

The approach can also yield better performance if the execution engine contains the necessary optimizations. However, implementing these is a lot of work, so most DSLs do not lead to significant gains in performance.

²² This is also known as correct-by-construction: the language only allows the construction of correct programs.

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**. 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 into a DSL implementation are specific to a particular tool (and lost if you change it), the models should largely be migratable²³.

²³ This leads to an interesting definition of legacy code: it is legacy, if you cannot access the domain semantics of a data structure, and hence, you cannot automatically migrate it to a different formalism.

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 just using the DSL is 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 sense, a language defini-

Building a language requires formalization and decision making: you can't create a DSL if you don't really know what you're talking about.

tion is an "executable analysis model"²⁴ I have had several occasions where customers said after a three-day DSL prototyping workshop that 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 the DSL. In effect, a DSL is a formalization of the Ubiquitous Language in the sense of Eric Evans' Domain Driven Design²⁵.

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 techies and domain people: domain people can easily read, and often write program code, since it is not cluttered with implementation details irrelevant to them. And even when domain 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 productive.

2.5.8 No Overhead

If you are generating source code from your DSL program (as opposed to interpreting it) you can use ~~nice~~, 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 low-overhead 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, and energy consumption is an issue these days).

²⁴ Remember the days when "analysts" created "analysis models"?

²⁵ E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley, 2004

Of course, the domain (and the people working in it) must be suitable for formalization, but once you start looking, it is amazing how many domains fall into this category. Insurance contracts, hearing aids and refrigerators are just some examples you maybe didn't expect. On the other hand, I once helped build a DSL to express enterprise governance and business policies. This effort failed, because the domain was much too vague and too "stomach driven" for it to be formalizable.

JetBrains once reported the following about their *webr* and *dnq* Java extensions for web applications and database persistence: "Experience shows that the language extensions are easier to learn than J2EE APIs. As an experiment, a student who had no experience in web development was tasked to create a simple accounting application. He was able to produce a web application with sophisticated Javascript UI in about 2 weeks!"

2.5.9 Platform isolation

In some cases, using DSLs can abstract from the underlying technology platform²⁶. Using DSLs and an execution engine makes the application logic expressed in the DSL code independent of the target platform²⁷. 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 worthwhile the (undoubtedly necessary) investment into the approach.

2.6 Challenges

There ain't no such thing as a free lunch. This is also true for DSLs. Let us look at the price you have to pay to get all these ~~nice~~ benefits described above.

2.6.1 Effort of Building the DSLs

Before a DSL can be used, it has to be built. If it has been built already, before a project, then using the DSL is obviously useful. If the DSL has to be developed as part of a project, the effort for building it has to be factored into the overall cost-benefit analysis. For technical DSLs²⁸, 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 side, application domain-specific DSLs (e.g., pension plan specifications) are often very narrow in focus, so the investment of 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 for describing this knowledge in a formal, uncluttered, portable and maintainable way. That should also 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.

²⁶ Remember OMG's MDA? They introduced the whole model-driven approach primarily as a means to abstract from platforms (probably a consequence of their historical focus on interoperability). There are cases where interop is the primary focus: cross-platform mobile development is an example. However, in my experience, platform independence is often just one driver in many, and it is typically not the most important one.

²⁷ this is not necessarily true for architecture DSLs and utility DSLs whose abstractions may be tied relatively closely to the concepts provided by the target platform.

²⁸ Technical DSLs are those that address aspects of software engineering such as components, state machines or persistence mappings, not application domain DSLs for a technical domain (such as automotive software or machine control)

There are three factors that make DSL creation cheaper: deep knowledge about a domain, experience of the DSL developer and productivity of the tools. This is why focussing on tools in the context of DSLs is important.

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" where tools like lex/yacc, ANTLR, C and Java were the only ingredients you would 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 there is a significant element of experience and practice you need to 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. Especially in the latter case it is important that you establish some kind of process of how language users interact with language developers and how they interact with domain experts (in case they are not the language users themselves). Just like any other situation where one group of people creates something which another group of people relies on, this can be a challenge²⁹

²⁹ This is not much different for languages than for any other shared artifact (frameworks, libraries, tools in general), but it also isn't any simpler and needs to be addressed.

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 contexts, you have to plan ahead (people, cost, time, skills) for the maintenance phase. A language that is not actively maintained and evolved will get outdated over time and become a liability. Especially during the phase where you introduce DSLs into an organization, a rapid evolution based on the requirements of users is critical to build trust in the approach³⁰.

³⁰ While this is an important aspect, once again, it is not worse for DSLs than it is for any other shared, reused asset.

2.6.5 *DSL Hell*

Once development of DSLs becomes technically easy, there is a danger that developers create new DSLs instead searching for and learning existing DSLs. **This may end up in a large 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 are all addressed by governance and effective communication in the team³¹.

³¹ It also helps if DSLs are incrementally extensible, so an existing language can be extended instead of creating a completely new language.

2.6.6 Investment Prison

The more you invest into reusable artifacts, the more productive you become. However, you may also get locked into a particular way of doing things. Radically changing your business model 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 into 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 there may still remain an element of cultural bias. 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 organizations, and I recommend reading the Fearless Change book by Rising and Manns³² if you're the person who is driving the introduction 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 ~~guy~~ 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 these as challenges in the context of DSLs specifically.

■ *Is it worth it?* Should you use DSLs? Presumably 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

With the advent of the digital age, we all know many businesses that went bankrupt because they had stuck too long to a dying business model. Maybe they just couldn't see that things will change, but maybe it was because they were so efficient at what they were doing, they couldn't invest into new ~~stuff~~ for fear of cannibalizing their mainstream business.

³² M. Manns and L. Rising. *Fearless change patterns for introducing new ideas*. Addison-Wesley Professional, first edition, 2004

an informed decision. In the end, you have to decide for yourself or maybe ask people for help 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 expensive³³ 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 concerning 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 manually, on the the target platform, 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 the **respective** tooling are sophisticated software programs themselves. They need to be designed, tested, deployed, 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 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. So, if your development team lacks this maturity, you might want to consider to first introduce those concerns into the team before you start using DSLs in a strategic way — the occasional utility DSL is the obvious exception.

³³ If everyone is aware of this, then you might still want to try to build a language as a means of building the understanding about the domain. But this is risky, and should be handled with care.

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 in which aspects of software engineering DSLs have been used successfully. Part IV of the book provides extensive treatment for 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 lives³⁴.

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).

³⁴ Often, these DSL serve as a "nice front end" to an existing library or framework, or automates a particularly annoying or intricate aspect of software development in a given domain.

2.7.2 *Architecture DSLs*

A somewhat more large-scale use is to use DSLs 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 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 make sure that the system architecture is consistently implemented by a potentially large development team.

For example, In AUTOSAR, the architecture is specified in models and 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, where 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.

2.7.3 *Full Technical DSLs*

For certain 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 the next category.

Examples include DSLs for (certain kinds of) Web applications, DSLs for mobile phone apps as well as DSLs for developing state-based or dataflow-based embedded systems.

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 experts³⁵.

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.

³⁵ In contrast, technical DSLs are often much easier to define, since they are guided very much by existing formal artifacts (architectures, frameworks, middleware infrastructures).

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 into 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 **imply some formal meaning into** domain entities and terms such as *should* or *must*³⁶.

³⁶ The latter kind of DSLs, also called *Controlled Natural Language*, is quite different from the kinds of DSLs we cover in this book. I will not cover it any further.

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 al-

ready 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 why this type of DSLs is used. 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 ??).

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.

2.8 *Differentiation from other Works and Approaches*

2.8.1 *Internal vs. 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 how 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 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:

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

- Fluent APIs are chained sequences of method calls. The syntax makes this obvious, and there is no way how to change this syntax, typically, 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 example³⁷, which defines a data structure (and, implicitly, a database table) for a blog post:

```
class Post < ActiveRecord::Base
  validates :name, :presence => true
  validates :title, :presence => true,
              :length => { :minimum => 5 }
end
```

³⁷ taken from <http://rubyonrails.org>

While I recognize the benefits of internal DSLs, I think they are fundamentally limited by the fact that an important ingredient is missing: IDE support. In classical internal 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 book³⁸.

³⁸ In addition, I don't have enough real-world experience with internal DSLs to be able to talk about them in a book

Note that with modern language workbenches, you can also achieve language extension or embedding, resulting in the same (or even a somewhat cleaner) syntax. However, these extensions and embeddings are *real* language extensions (as opposed to meta programs) and do come with support for static constraint checking and IDE support. We cover this extensively in the book.

2.8.2 Compiler Construction

Languages definition, program validation and transformation or interpreters are obviously closely related to compiler construction. 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 definition³⁹, a concern not typically associated 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.

³⁹ There are universities who teach compiler construction based on language workbenches and DSLs.

2.8.3 UML

So what about the Unified Modeling Language - UML? I decided not to cover or use 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 DSL. UML can be seen as an integrated *collection* of DSLs that describe various aspects of software systems: class structure, state based behavior, or deployment. However, these DSLs still address the overall domain of *software*. 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 as part of profile definition. In practice, most people **only a** very small part of the UML with the majority of concepts defined via profiles. It is my experience that because of that, it is much more productive, and often even less work, to build DSLs with "real" language engineering environments as opposed to UML profiles.

This is one reason why I decided not to cover it. Also, the fact that this book has an emphasis on textual languages and UML is graphical, obviously also drove my decision to not cover it in any detail.

So is UML used in an MDSD context? Sure. People build profiles and use UML-based DSLs, especially in large organizations where the perceived need for standardization is paramount⁴⁰.

2.8.4 Graphical **vs.** Textual

This is something of a religious war, akin to the statically-typed versus dynamically-typed languages debate elsewhere. 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 nature⁴¹.

However, this book does have a bias towards textual notations, for several reasons. I feel that the textual format is more generally useful, 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

When I wrote my "old" book on MDSD, the UML played an important role. At the time, I really did use UML a lot for projects involving models and code generation. Over the years, the importance of UML has diminished significantly (in spite of the OMG's efforts in popularizing both UML and MDA) mainly because of the advent of modern language workbenches.

⁴⁰ It's interesting to see that even these sectors increasingly embrace DSLs. I know of several projects in the aerospace/defense sector where UML-based modeling approaches were replaced with very specific and much more productive DSLs. It is also interesting to see how sectors define their own standard languages. While I hesitate to call it a DSL, the automotive industry is in the process of standardizing on AUTOSAR.

⁴¹ The ideal tool will allow you to use and mix all of them and we will see in the book how close existing tools come to this ideal case.

to learn more about graphical DSLs, I suggest you read Kelly and Tolvanen's Domain Specific Modeling book⁴².

⁴² S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, March 2008