MARKUS VOELTER

DSL ENGINEERING

Part I

Introduction

1 Setting the Stage

Domain-Specific Languages are becoming more and more important in software engineering. Tools are becoming better as well, so DSLs can be developed with less effort. In this chapter I first explain 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 end goal of DSLs. I discuss the benefits of DSLs, some of the challenges for adopting DSLs and describe a few application areas. Finally, I define a couple of important terms that will be used throughout the book.

1.1 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 exchangeable.

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 ran by a virtual machine (a mix between an interpreter and a compiler). But in principle, you could compile Ruby to native code, and you could interpret C. Or could you?

The real reason why these languages are used for what they are used for is that their features are tailored to the tasks that are relevant in the respective usage areas. 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), you have powerful string manipulation features (to handle input received from a website) and the meta programming facility supports the definition of abstractions that are very suitable for Web applications (Rails is*the* example for that).

By now it should be clear that even within "general-purpose programming" we use different languages that provide different features tailored to the specific tasks at hand. 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 relatively obvious that 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 conveniently express these abstractions. In many cases these are textual notations, but tables, symbols (as in mathematics) or graphics might also be useful. Assuming the semantics of the abstractions is well defined, good abstractions and suitable notations are a good starting point for effectively expressing programs for a specialized domain.

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

Engineering a language doesn't stop there, because the language has to be "brought to life" as well — programs written in the language have to be executed somehow. There are two main approaches for that: translation/generation/compilation and interpretation. The former translates a DSL program to a language for which an execution engine on a given target platform already exists. 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 gap between the language abstractions and the target platform (i.e., the platform the interpreter or generated code runs on), execution can become inefficient. E.g., if you try to store and query graph data in a relational database, this will be very inefficient because many joins are necessary. Another example is trying to run Erlang on a system which only provides heavy-weight processes — having thousands of processes as Erlang requires, is not going to be efficient. So, when defining a language for a given domain, you should be aware of the intricacies of the target platform and the interplay between execution and language design.

BY NOW YOU SHOULD BELIEVE to a certain extent that specific problems require specific abstractions. But why do we need full blown languages? Aren't objects, functions, APIs and frameworks good enough? What does the *language* add to the picture?

Languages (and the programs you write with them), are the clearest form of abstraction. You can get rid of all the unneeded clutter that an API — or anything else embedded in or expressed with a general-purpose language — requires. You can define a notation that fits the abstractions well and makes interacting with programs easy and efficient. 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.

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 whose abstractions and notations are aligned with the domain they work in.

So then how are DSLs different from "real" programming languages, and what do they have in common? Obviously the core abstractions are aligned with whatever domain the DSL is built for, and the notations are suitable as well. It may have custom Also, in many cases, DSLs are much smaller and simpler than GPLs (although there are some pretty sophisticated DSLs). I don't go as far as saying that DSLs are always declarative (it is not completely clear what this means anyway), or that they may never be Turing complete. But if your DSL approaches what Java can do, you might consider just using Java.

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

So, I guess we agree that Java is not a DSL. But what about Mathematica, SQL, State Charts or HTML? Are these DSLs? This is not so easy to answer. They are, of course, DSLs in the sense that they are languages that are optimized for (and limited to) a special domain or problem. However:

- These languages are really really big, complicated and involved. While I don't claim that "our" DSLs are always simple, it is really rare that you'll build something comparable to SQL in sophistication or size.

- An underlying assumption of many (process) recommendations in this book is the closed-world assumption: as the developer of a DSL, you know the users and can communicate with them (e.g. to make them remove deprecated language constructs from their models). This is not true for languages that are used by (more or less) the whole world, such as the ones mentioned above.

You may know the saying in 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 (like the ones mentioned above), we don't call them DSLs anymore. So maybe we can agree that those well-known, widely used and big languages are in fact DSLs, but because they are so big and widespread, not all of the advice I give in this book may apply.

TO BUILD GOOD DSLS, HOWEVER, it is not enough to just "underline all the nouns and verbs in the domain" and make them language features. This may be a good starting point, but it is way too simplistic. All the interesting DSLs I've built start out simple, with straightforward concepts, but then grow more sophisticated. For example, the way I have used to compose variants of appliances in one DSL resemble Scala's way of dealing with classes and traits. So it is a good idea to learn from other (programming) languages, especially regarding concepts such as modularity, information hiding, specialization and the ability to define your own abstractions in programs.

Remember the times when this was said to be a way to find the classes and methods in OO programming?

As we will see, domain-specificity is not black-and-white, but rather a gradual: a language is *more* or *less* domain specific. So I have to define how this book is not generally about programming language design. The following table lists a set of language characteristics. While DSLs and general purpose languages (GPLs) can have characteristics from both the second and the third columns, DSLs are more likely to pick characteristics from the right column. This makes designing DSLs a more tractable problem than designing general purpose languages

There are many connections to general purpose languages. DSL programs may be translated into GPL code. DSL code may be embedded in DSL programs. And DSL designers can learn a lot from GPLs.

Domain Size	large and complex	smaller and well-defined
Designed by	guru or committee	a few engineers and domain experts
Language Size	large	small
Turing-completeness	almost always	often not
User Community	large, anonymous and widespread	small, accessible and local
In-language abstraction	sophisticated	limited
Lifespan	years to decades	months to years (driven by context)
Evolution	slow, often standardized	fast-paced
Deprecation/Incompatible Changes	almost impossible	feasible

Figure 1.1: Domain-specific languages versus programming languages. DSLs tend to pick more characteristics from the third column, GPLs tend to pick more from the second.

1.2 Modeling and Model-Driven Development

The above approach of defining and using DSLs is a form of model-driven development: we create formal, tool- processable representations of certain aspects of software systems. We then use interpretation or code generation to transform those representations into executable code expressed in programming languages and the associated XML/HTML/whatever files. With today's tools it is easily possible to define arbitrary abstractions to represent any aspect of a software system in a meaningful way. It is also straightforward to built code generators that generate the executable artifacts. Within limits (depending on the particular tool used), it is also possible to define suitable notations that make the abstractions accessible to non-programmers (for example opticians or thermodynamics engineers).

However, there are also limitations to the approach. The biggest one is that 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 MDD 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 models with a common API. However, since every programming language and IDE has its own API for accessing the program's abstract syntax tree (AST), interoperability with source code is challenging — you can't treats source code the same way as models. 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. This results in generating skeletons into which source code is inserted, or the arcane practice of pasting C snippets into 5-in 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 modeling today? The table contains some (general and broad) statements:

Aspect	**Modeling**	**Programming**
Define your own notation and language	Easy	Possible
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	Standard
Debugger	Rarely	Almost always
Versioning/Diff/Merge	Depends on tooling	Standard

Figure 1.2: Comparing Modeling and Programming

So one can and should ask: why is there a difference?[1] I guess the primary reason is history, the two worlds have different origins and have evolved in different directions.

[1] We use modeling in its *prescriptive* version here. Of course there is a reason for *descriptive* modeling which is not intended to be executable and used primarily for communication and understanding among humans.

Programming languages have traditionally been using textual syntax. Modeling languages traditionally used graphical notations, because of the ill-conceived idea that what is represented as pictures is always easy to understand. Of course there are textual domain specific languages (and failed graphical general purpose programming languages), but the use of textual syntax for domain specific modeling has only recently become more prominent, see my discussion above. Programming languages have traditionally used storage based on concrete syntax, together with parsers that transform a character stream to an abstract syntax tree for further processing. Modeling languages have traditionally used editors that directly manipulate the abstract syntax, using projection to render the concrete syntax in the form of diagrams. Modeling tools have also provided the ability to define views, i.e. 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.

I want to make the point that there should be no difference. Programming and modeling should be based on the same fundamental approach, enabling meaningful integration. No programmer really wants to model, they want to program, but:

at different levels of abstraction: some things may have to be described in detail, low level, algorithmically, other aspects may be described in more high-level terms

from different viewpoints: seaprate aspects of the system should be described with languages suitable to these aspects

with different degrees of domain-specificity: some aspects of systems are generic enough so they can be described with reusable, generic languages. Other aspects require their own dedicated, maybe even project-specific DSLs.

with suitable notations, so all stakeholders can contribute directly to "their" aspects of the overall system

with suitable degrees of expressiveness: aspects may be described imperatively, with functions, or other turing-complete formalisms, and other aspects may be described in a declarative way

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 need for modular languages.

1.3 Modular Languages

Looking at today's languages, there are two major kinds of languages. Big ones and small ones. Big languages Fig. 1.3 have a relatively large set of very specific language concepts. These languages are, the proponents say, easy to learn, since "there's a keyword for everything". Examples include Cobol or ABAP.

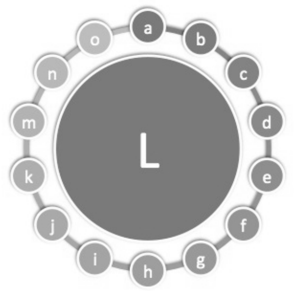


Figure 1.3: A Big Language: Many specific language concepts

The other kind of language is the small language. It has very few, but very powerful language concepts that are highly composable. Users can build their own abstractions. Proponents of this kind of language also say that those are easy to learn, because "you only have to learn three concepts". Lisp is the prominent example of this kind of language.

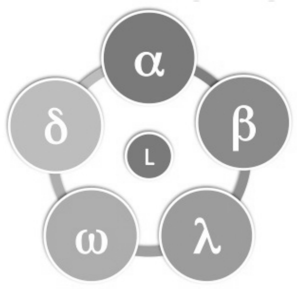


Figure 1.4: A Small Language: Few, but powerful language concepts

Here is a third option: modular languages. 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. New, custom modules can be built and used at any time. Each module addresses a specific concern of the system to be built. 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.

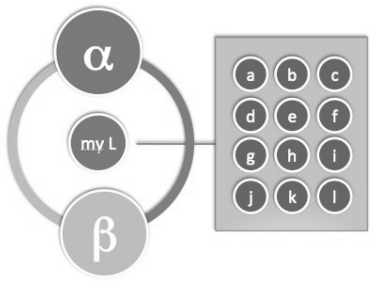


Figure 1.5: A Modular Language: A small core, and a library of reusable language modules

This idea isn't new. Charles Simonyi[2] and Sergey Dmitriev[3] have written about it, so has Guy Steele in the context of Lisp[4]. The idea also relates very much to defines language workbenches as tools where:

[2] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOPSLA*, pages 451–464, 2006

[3] S. Dmitriev. Language oriented programming: The next programming paradigm, 2004

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

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 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. These two state that projectional editing be used. 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, as we will see below. However, in the end I don't care, as long as languages are modular. If this is possible with a different approach, such as scannerless parsers, that is fine with me.

Language designers define a DSL in three main parts: schema, ed-

itor(s), and generator(s). I agree that ideally a language should be defined "meta model first", i.e., you first define a schema, then the editor or grammar, and then the generator to map your constructs to existing languages. However, it is also okay for me to start with the grammar, and have the meta model derived. From the user's point of view, it does not make a big difference.

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, I want to get tool support: syntax highlighting, code completion, static type checking, 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. This means that together with DSL, general-purpose languages 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 or C#, but it could also be 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.

GENERALLY I EXPECT THE SYNTAX to be textual. Decades of experience show that textual syntax, together with good tool support, is perfectly adequate for large and complex software systems. This becomes even more true if you consider that programmers will have to write less code, since the abstractions available in the languages will be much more closely aligned with the domain than is the case for traditional languages. Programmers can always define a language module that fits a domain. However, there are worthwhile additions. For example, semi-graphical syntax would be useful, i.e. the ability to use graphical symbols as part of a fundamentally text-oriented editor: mathematical symbols, fraction bars, subscript/superscript, or even tables, can make programs represent certain domains much more clearly.

Custom visualizations are important as well. Visualizations are read-only, automatically layouted and provide drill-down back to the program. They are used to illustrate certain global properties of the program or to answer specific questions, typically related to interesting metrics.

Finally, actual graphical editing is useful for certain cases. Examples include data structure relationships, state machine diagrams or data flow systems. The textual and graphical notations must be integrated, though: you will want to embed the expression language module into the state machine diagram to be able to express guard conditions.

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 domain specific language that supports your company's specific way of defining insurance contracts. It is essential that the language is exactly aligned with your business, so you have to define the language yourself. There are other similar examples: building DSLs to describe radio astronomy observations for a given telescope, a 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).

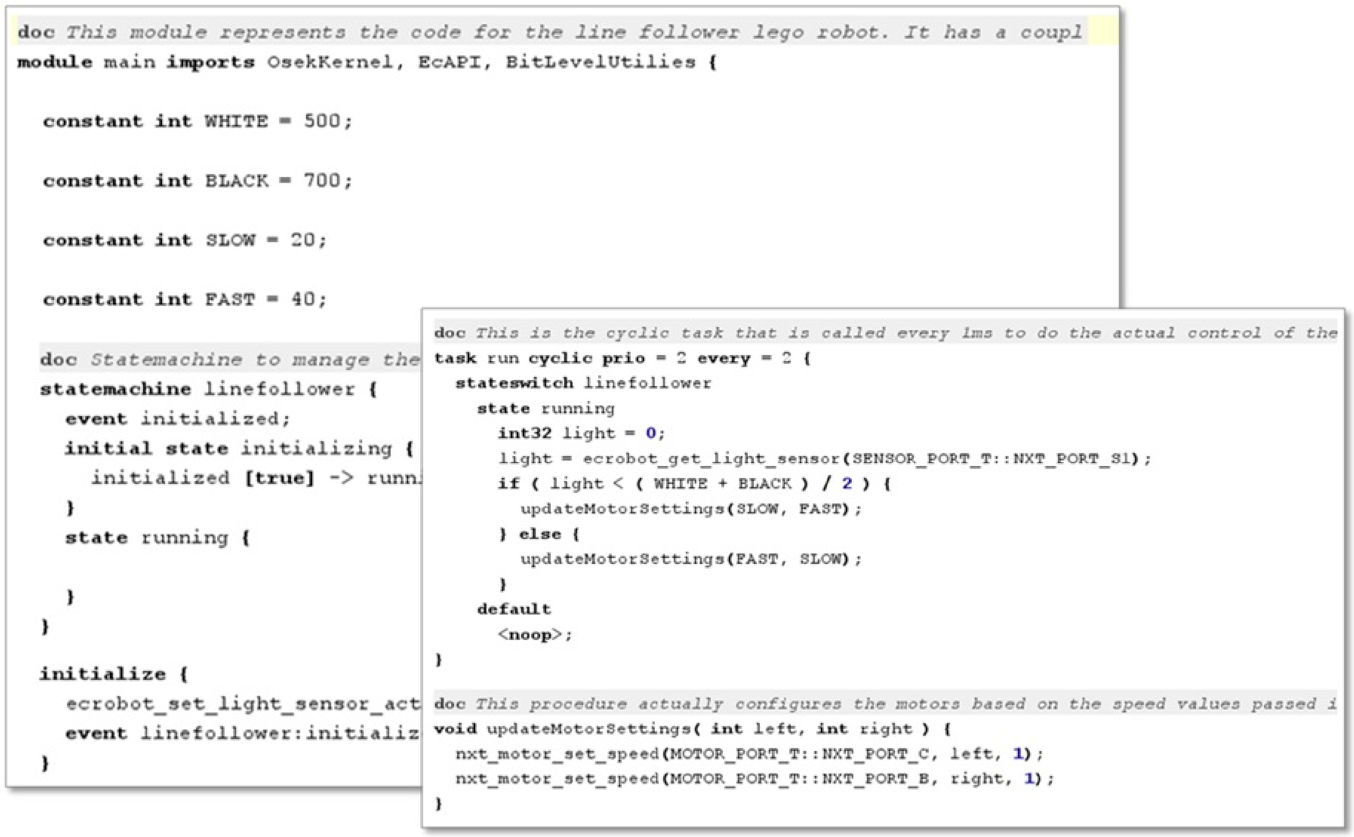
However, for a large range of technical or architectural concerns, the abstractions are well known. They could be made available for reuse (and adaptation) in a library of language modules. Examples include

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

This sounds like a lot 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 likely act as the focal point around which additional language modules are organized. Some of the tools outlined in the next section support this approach.

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. 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.



```
doc This module represents the code for the line follower lego robot. It has a coupl
module main imports OsekKernel, EcAPI, BitLevelUtilies {

  constant int WHITE = 500;

  constant int BLACK = 700;

  constant int SLOW = 20;

  constant int FAST = 40;

  doc Statemachine to manage the
  statemachine linefollower {
    event initialized;
    initial state initializing {
      initialized [true] -> runn
    }
    state running {

    }
  }

  initialize {
    ecrobot_set_light_sensor_act
    event linefollower:initializ
  }
```

```
doc This is the cyclic task that is called every 1ms to do the actual control of the
task run cyclic prio = 2 every = 2 {
  stateswitch linefollower
    state running
      int32 light = 0;
      light = ecrobot_get_light_sensor(SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
    default
      <noop>;
}

doc This procedure actually configures the motors based on the speed values passed i
void updateMotorSettings( int left, int right ) {
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, left, 1);
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, right, 1);
}
```

Figure 1.6: A program written in a modularly extended C

For me, this is the vision of programming I am working toward. The distinction between modeling and programming is gone. People can develop code using a language directly suitable to the task. They can also build languages, if that makes sense. These languages will be relatively small, since they only address one aspect of a system. They are not general-purpose. They are DSLs.

Tools for this approach exit. 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

1.4 *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 let's look at the upside first.

1.4.1 *Automation*

The most obvious benefit of using DSLs is that - once you've got a language and an execution engine - your work in the particular aspect of software development covered by the DSL becomes much more efficient, simply because you don't have to do the grunt work manually[5]. This is most obvious if you generate a whole truck load of code from a relatively small DSL program. There are many studies that show that the pure amount of code one has to write (and read!) can become a problem. The ability to reduce that amount while retaining the same semantic content, is a huge advantage.

[5] Presumably, the amount of DSL code you have to write is much less than what you'd have to write if you'd use the target platform directly. If that were not the case, using DSLs in that context makes no sense.

You could argue that a good library or framework will do the job as well. And that is certainly true in some cases, but not in all. However, DSLs provide a number of additional benefits, as we will see in the next couple of items.

1.4.2 *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. 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).

1.4.3 *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 by implementation details.

In other words: using DSLs allows you to separate essential from incidental complexity, moving the latter to the execution engine. This also makes team communication and reviews simpler.

But not just using the DSL is useful; also, the act of building the language can help you sharpen your understanding of whatever you build the language for. In some sense, a language definition is an "executable analysis model" - remember the days when "analysts" created "analysis models"? 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' DDD.

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.

TODO: cite

1.4.4 *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 the techies and the domain people. 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.

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 where you maybe didn't think this is true.

1.4.5 *Platform isolation*

Using DSLs and an execution engine makes the application logic expressed in the DSL code independent of the target platform[6]. 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 specific.

[6] 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.

1.4.6 *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, the avoidance of duplication in code (if the DSL is engineered in the right way) and the automation of repetitive work (by the execution engine). As the next item shows, more meaningful validation can be performed on the level of DSL programs, increasing the quality further.

1.4.7 *Analysis and Checks*

Because DSLs capture their respective concern in a way that is uncluttered from the implementation, it is more semantically rich. Analyses are much easier to implement and error messages can use more meaningful wording, because it happens on the domain level. As mentioned above, some DSLs are built specifically to enable non-trivial, formal analyses.

1.4.8 *Productive Tooling*

In contrast to libraries and frameworks, DSLs can come with tools, i.e. IDEs, that are aware of the language. This can result in a much improved user experience. Code completion, visualizations, debuggers, simulators and all kinds of other niceties can be provided.

It is fair to say that libraries and frameworks *can* expose themselves as internal DSLs in a way that the host language's IDE is language-aware to at least some extent. However, it's quite difficult - if not impossible - to attain the same quality user experience and then the disadvantages of using internal DSLs remain. Once again, I focus exclusively on external DSLs for this book.

OFTEN, NO SINGLE ONE OF THE ADVANTAGES would drive you to use a DSL. But in many cases you can benefit in multiple ways, so the sum of the benefits is worthwhile the (undoubtably necessary) investment into the approach.

1.5 *Challenges*

TANSTAAFL[7]. This is also true for DSLs. Let us look at the price you have to pay to get all these nice benefits mentioned above.

[7] There ain't no such thing as a free lunch

1.5.1 *Effort of Building the DSLs*

Since DSLs are specific to a domain, DSLs often have to be built before they can be used, as part of a project. For technical DSLs, 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., mortgage contracts 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 and 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.

I hope that this book helps you get over the learning curve with some of these tools. But in the end, you (or somebody else) have to make the decision when and how to use DSLs. Experience helps here.

1.5.2 *Language Engineering Skills*

DSLs are not rocket science. But it is a skill that not everybody has, because it is not practiced that often by most people. Also, the whole language/compiler thing has a bad rep 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 in itself - independent of tooling and technicalities - is a skill that needs to be learned and trained. How do you find out which abstractions need to go into the languages? How do you create "elegant" languages? I will elaborate on this later in the book, but there is a significant element of experience and practice you need to build up over time.

1.5.3 Process Issues

Using DSLs inevitably leads to work share: 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 challenge[8]

[8] 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.

1.5.4 Maintenance

A related issue is language evolution and maintenance. Again, just like any other asset you develop for use in related contexts, you have to plan ahead (people, cost, time, skills) for the maintenance phase. A language that is not actively maintained might get out of date - and hence less useful - over time. 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 approach[9].

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

1.5.5 Political 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 will always remain an element of politics, selling and convincing that is relatively independent of the actual technical arguments. These problems will 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 this if you're the person who is driving the introduction of DSLs into your organization[10].

[10]

So, 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 an informed decision. In the end, you have to decide for yourself or maybe ask people for help who have done it before.

A good way of defining the applicability of something is to point out clearly when something is not applicable. So, let's 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[11] Another sign of problems is this: if you build your DSL iteratively and over time, 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 and it is maybe time for an about-face.

[11] 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.

Another problem is an unknown target platform. If you don't know how to implement something on the target platform in the first place, you'll have a hard time to implement an execution engine (generator or interpreter). You might want to consider writing (or inspecting) a couple of representative example applications manually to understand the patterns that 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 experience is a prerequisite. 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.

1.6 Applications of DSLs

In this book, I am not just going to explain how to build DSLs. I also want to explore why you might want to do that in the first place. Instead of elaborating on the advantages one by one, I am going to provide real-life, mature examples of how DSLs are used. The whole second part of the book is concerned with this. Below, you'll find a couple of brief examples, some of which I will get back to in the second part. I will list the promised benefits of using DSLs briefly in the next section, hoping that the rest of this book will really convince or help you in convincing others.

1.6.1 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 lives[12].

[12] 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.

Examples include the generation of array-based implementations for state machines, any number of interface/contract definitions out of 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).

1.6.2 Architecture DSLs

This is one of the most interesting use cases of DSLs and I have seen (and contributed to the creation of) many instances. This is why a later section focusses on architecture DSLs extensively

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. Architecture DSLs are usually developed during the architecture exploration phase of a project and make sure that the system architecture is consistently implemented by a potentially large development team. 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.

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 SCA components 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.

1.6.3 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 the proverbial wrist watch (you know what I am talking about if you know Metacase :-)).

1.6.4 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, and usually more effort in building the language, since a "messy" application domain first has to be understood, structured and possibly "re-taught" to the domain experts. In contrast, technical DSLs are often much easier to define, since they are guided very much by existing formal artifacts (architectures, frameworks, middleware infrastructures).

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

1.6.5 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*.

1.6.6 DSLs used for Analysis

Another category of DSL use is as the basis for analysis, checking and proofs. Of course, checking plays a role in all use cases for DSLs - you want to make sure that the models you release for downstream use are "correct" in a sense that goes beyond what the language syntax already commands. But in some cases, DSLs are used to express concerns in a formalism that lends itself to formal analysis or model checking (concurrency, resource allocation, etc.). 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.

Examples include systems to find counterexamples, allocate resources or optimize resource consumption, or to come up with optimal scheduling algorithms.

1.6.7 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.

Product Line Engineering (PLE) is a huge topic which I cannot do justice in this book, let alone this paragraph.

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.

NOTICE THAT IN ALL OF THIS BOOK I only cover use cases for DSLs in which the model is a first-class citizen (of the software development process) and/or the authoritative source. Obviously, you can use modeling and languages and specialized notations just to illustrate or visualize things in order to facilitate communication among humans. As we all know, this is very useful, but to this end you usually don't go the extra mile of formalizing your language - in most cases, intuitive understanding or a little legend is enough. I consider this use of models and (informal) languages as outside the scope of our book.

2
About this Book

I HAVE DECIDED TO WRITE A BOOK on Domain-Specific Languages (DSLs). The book will cover three main aspects: DSL design, DSL implementation and software engineering with DSLs, focussing mainly on textual languages. The book will makes use of modern language workbenches. It is not a tutorial for any specific tool, but of course I will provide examples and some level of detail for mainly MPS and Xtext, and to a lesser degree, some others. The goal of the book is to provide a thorough overview of modern DSL engineering. The book tries to be objective, but it is majorly based on my own experience and opinions.

2.1 Why this book

A lot has been written about language design, language development, modeling, code generation and the like. I have already coauthored a book on this topic before, so why do we need another one?

First of all, there is currently no book available that explicitly covers DSLs developed with modern language workbenches, with an emphasis on textual languages. I feel that this way of developing DSLs is very productive and very useful, so I decided I'd need to provide a book to fill 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.

Even though I have written a book on Model-Driven Software Development (MDSD) before, I feel that it is time for a complete rewrite[1]. 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

[1] I have learned a lot in the meantime, my viewpoints have evolved and the tools that are available today have evolved significantly. Merely creating a new edition of the old MDSD book does not do justice to these developments. This is why I have decided to be part of a book project that starts over. Also, many people specifically asked me to do that

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.

2.2 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 book[2] emphasizes this aspect.

[2]

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. Developers, as well as the domain experts that use the DSL are used to powerful, feature-rich IDEs and tools in general. If you want to establish the use of DSLs and you propose to your users to use `notepad.exe`, you won't get very far.

Also, the effort to develop (groups 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.e. 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 approaches. I tried to use diverse tools for the examples, but for the most part I sticked to tools I happen to know (well) and that have serious traction in the real-world or the potential to do so: SDF/Stratego/Spoofax, Eclipse Modeling + Xtext, JetBrains MPS and, to some extent, the Intentional Domain Workbench. Here is a brief overview over the tools:

TODO: Do we really keep that one in? Maybe ask the guys in Amsterdam?

2.2.1 SDF/Stratego/Spoofax

http://strategoxt.org/Spoofax

These tools are developed at the university of Delft in Eelco's group. SDF is a formalism for defining grammars and 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. This tooling has a number of advanced features in the space of language modularization and composition.

2.2.2 Eclipse Modeling + Xtext

http://eclipse.org/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, Xpand, a code generation engine, as well as EMF/Ecore, the underlying meta meta model used to represent model data. Xtext may not be as advanced as SDF/Stratego, 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, graphical editing frameworks among them.

2.2.3 *JetBrains MPS*

http://jetbrains.com/mps

The Meta Programming System (MPS) is a projectional language workbench, which means that there is no grammar and parser involved, rather, 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.

2.2.4 *Intentional Domain Workbench*

http://intentsoft.com

A few examples will be based on the Intentional Domain Workbench. This is a commercial product that also 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.

Way 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) all implement the same examples. This provides a good way to compare the various tools.

http://languageworkbenches.net

2.3 *Case Studies and Examples*

I strive to make this book as accessible and practically relevant as possible, so I provide a lot of examples. Throughout the book I use relatively simple examples, described in detail, to introduce principles. I use more sophisticated and elaborate examples, described in less detail, to showcase the power of DSLs. These examples are introduced below. These are mostly taken from real projects, some of them anonymized or stripped of all too project-specific idiosyncrasies.

I decided against a single big, running example because (a) it is a lot of work, (b) becomes increasingly complex to follow along and (c) fails to illustrate different approaches to the same problem, because the running example can contain only one solution. So, our examples are mostly isolated.

2.3.1 Component Architecture

This is an architecture DSL used to define the software architecture of a complex, ditributed, component-based system in the transportation domain. Among other architectural abstractions, the DSL DSL supports the definition of components and interfaces (Fig. 2.1) as well as the definition of systems, which are connected instances of components (Fig. 2.1). 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 distribution middleware. It is used by software developers and architects and implemented with Eclipse Xtext.

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

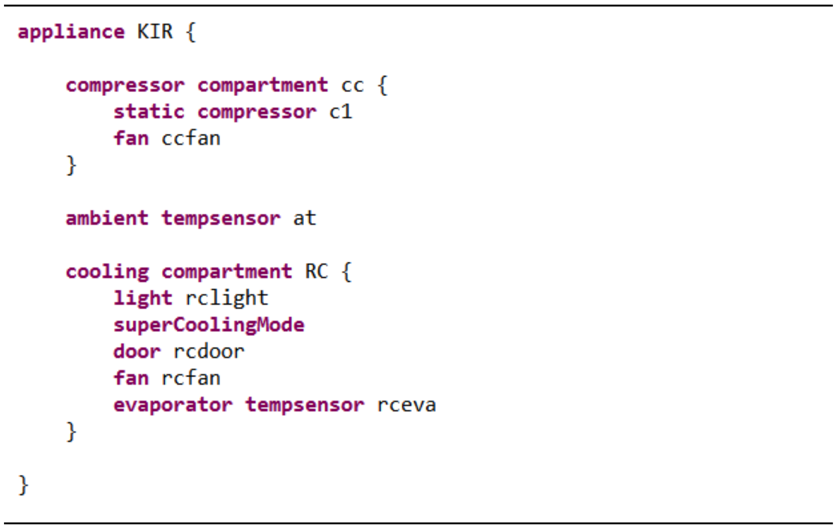
Figure 2.1: Components and interfaces

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

Figure 2.2: Component instances and connectors

2.3.2 Refrigerator Configuration

This case study describes a set of DSLs for cooling algorithms in refrigerators. Three languages are used. The first one describes the logical hardware structure of refrigerators (Fig. 2.3). The second one describes cooling algorithms in the refrigerators using a state-based, asynchronous language (Fig. 2.4). 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 (Fig. 2.5). These DSLs are used by thermodynamists and are implemented with Eclipse Xtext.



```
appliance KIR {

    compressor compartment cc {
        static compressor c1
        fan ccfan
    }

    ambient tempsensor at

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

}
```

Figure 2.3: Hardware structure definition in the refrigerator case study

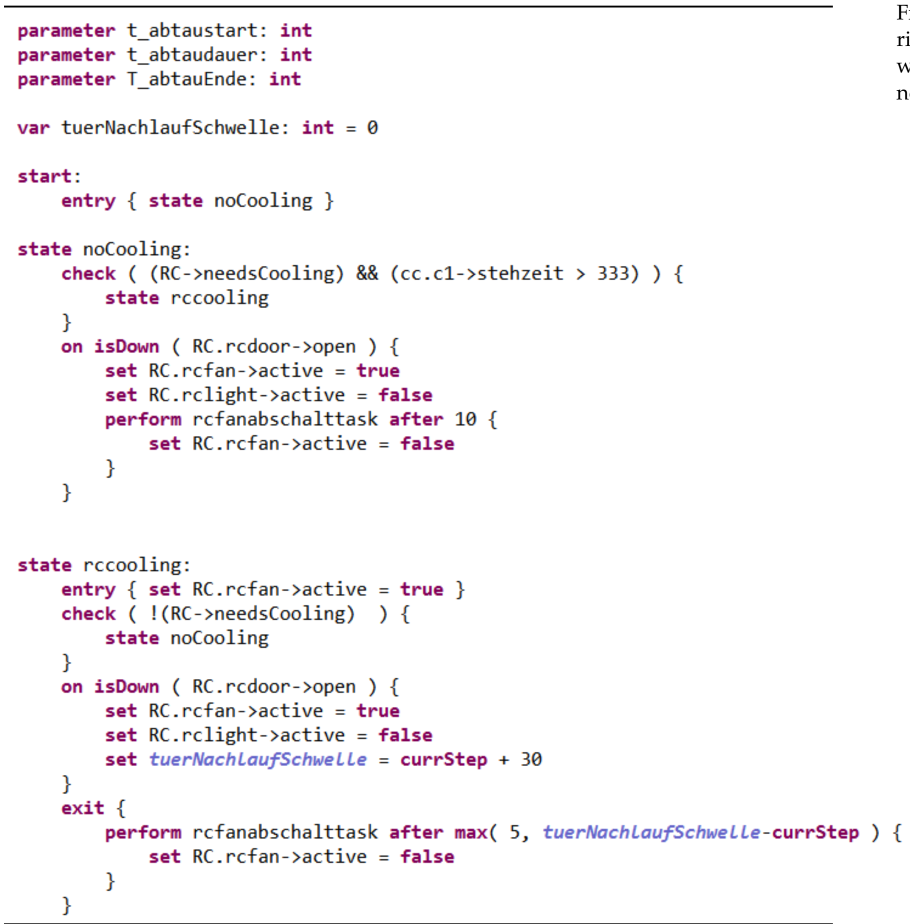
2.3.3 *Extended C*

This case study (detailed in[3]) covers a set of extensions to the C programming language tailored to embedded programming. These include state machines, physical quantities, tasks, interfaces and components. Higher level DSLs are added for specific purposes such as controlling a Lego Mindstorms robot. ANSI C is generated and subsequently compiled with GCC. This DSL is intended to be used by embedded software developers and it is implemented with MPS

[3] M. Voelter. Embedded software development with projectional language workbenches. In *MoDELS*, pages 32–46, 2010

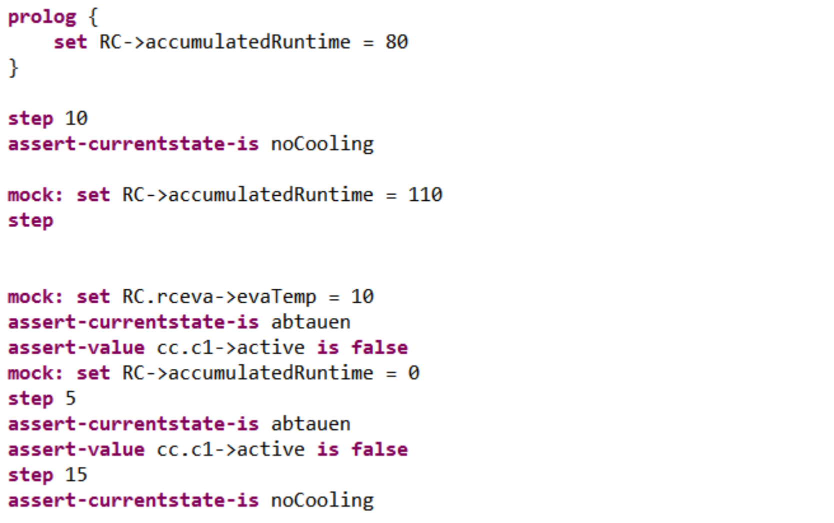
2.3.4 *Pension Plans*

The pension DSL supports mathematical abstractions and notations to allow insurance mathematicians to express their domain knowledge directly (Fig. 2.7) as well as higher level pension rules and unit tests in the table notation (Fig. **??**). This DSL is used to efficiently describe families of pension plans for a large insurance company. 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.



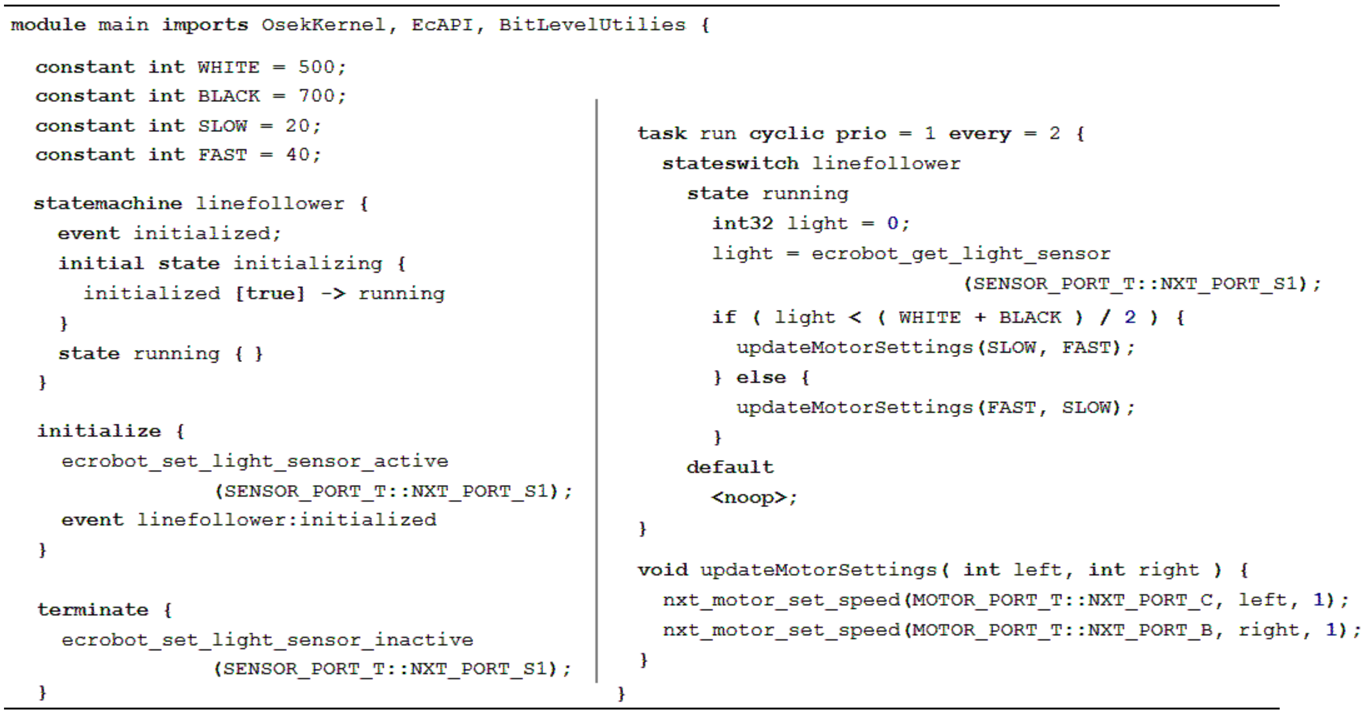
```
parameter t_abtaustart: int
parameter t_abtaudauer: int
parameter T_abtauEnde: int

var tuerNachlaufSchwelle: int = 0

start:
    entry { state noCooling }

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


state rccooling:
    entry { set RC.rcfan->active = true }
    check ( !(RC->needsCooling)  ) {
        state noCooling
    }
    on isDown ( RC.rcdoor->open ) {
        set RC.rcfan->active = true
        set RC.rclight->active = false
        set tuerNachlaufSchwelle = currStep + 30
    }
    exit {
        perform rcfanabschalttask after max( 5, tuerNachlaufSchwelle-currStep ) {
            set RC.rcfan->active = false
        }
    }
```

Figure 2.4: A simplified cooling algorithm. Note that I don't show a program with real-world complexity because I am not allowed to show it.



```
prolog {
    set RC->accumulatedRuntime = 80
}

step 10
assert-currentstate-is noCooling

mock: set RC->accumulatedRuntime = 110
step


mock: set RC.rceva->evaTemp = 10
assert-currentstate-is abtauen
assert-value cc.c1->active is false
mock: set RC->accumulatedRuntime = 0
step 5
assert-currentstate-is abtauen
assert-value cc.c1->active is false
step 15
assert-currentstate-is noCooling
```

Figure 2.5: A test script to test cooling programs



```
module main imports OsekKernel, EcAPI, BitLevelUtilies {

  constant int WHITE = 500;
  constant int BLACK = 700;
  constant int SLOW = 20;
  constant int FAST = 40;

  statemachine linefollower {
    event initialized;
    initial state initializing {
      initialized [true] -> running
    }
    state running { }
  }

  initialize {
    ecrobot_set_light_sensor_active
              (SENSOR_PORT_T::NXT_PORT_S1);
    event linefollower:initialized
  }

  terminate {
    ecrobot_set_light_sensor_inactive
              (SENSOR_PORT_T::NXT_PORT_S1);
  }
```

```
task run cyclic prio = 1 every = 2 {
  stateswitch linefollower
    state running
      int32 light = 0;
      light = ecrobot_get_light_sensor
                    (SENSOR_PORT_T::NXT_PORT_S1);
      if ( light < ( WHITE + BLACK ) / 2 ) {
        updateMotorSettings(SLOW, FAST);
      } else {
        updateMotorSettings(FAST, SLOW);
      }
    default
      <noop>;
}

void updateMotorSettings( int left, int right ) {
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, left, 1);
  nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, right, 1);
}
}
```

Figure 2.6: Example Code written using the Extended C language. It contains C functions, constants, as well as a task and a state machine.

2.3.5 *WebDSL*

WebDSL is a language for web programming[4] that integrates languages the different concerns of web programming, including persistent data modeling (entity), user interface templates (define), access control[5], data validation[6], search, and more. The language enforces inter-concern consistency checking, providing early detection of failures[7]. The fragment in Fig. 2.9 shqows a data model, user interface templates, and access control rules for posts in a blogging application. WebDSL is implemented with the Spoofax Language Workbench[8] and is used in the researchr digital library (`http://researchr.org`).

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

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

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

[7] 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

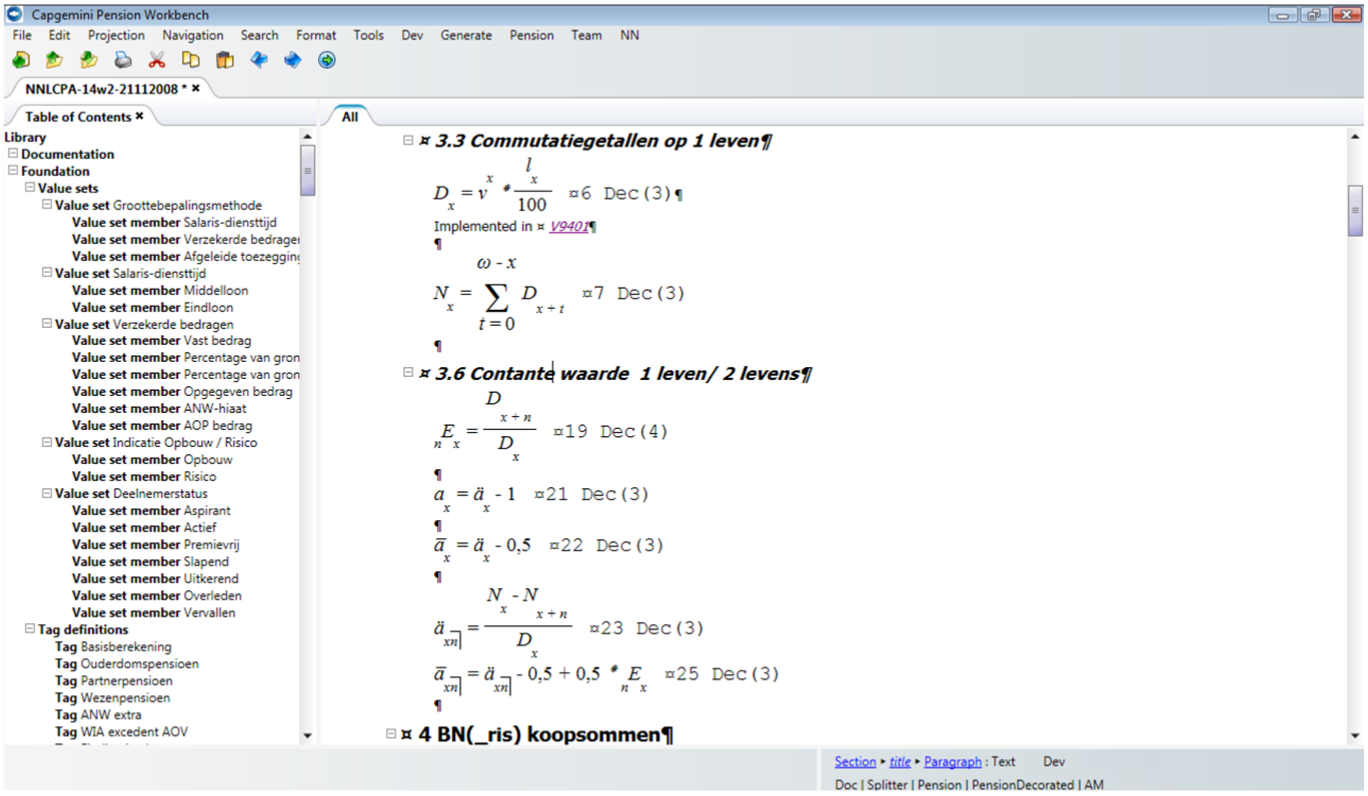
2.4 *Differentiation from other Works and Approaches*

2.4.1 *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). I distinguish internal DSLs in that sense from language embedding (as covered later in the book) because I feel that one main 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 book.

TODO: Point at good books to read on this topic



Library
- Documentation
- Foundation
  - Value sets
    - **Value set** Grootebepalingsmethode
      - **Value set member** Salaris-diensttijd
      - **Value set member** Verzekerde bedrage
      - **Value set member** Afgeleide toezeggin
    - **Value set** Salaris-diensttijd
      - **Value set member** Middelloon
      - **Value set member** Eindloon
    - **Value set** Verzekerde bedragen
      - **Value set member** Vast bedrag
      - **Value set member** Percentage van gron
      - **Value set member** Percentage van gron
      - **Value set member** Opgegeven bedrag
      - **Value set member** ANW-hiaat
      - **Value set member** AOP bedrag
    - **Value set** Indicatie Opbouw / Risico
      - **Value set member** Opbouw
      - **Value set member** Risico
    - **Value set** Deelnemerstatus
      - **Value set member** Aspirant
      - **Value set member** Actief
      - **Value set member** Premievrij
      - **Value set member** Slapend
      - **Value set member** Uitkerend
      - **Value set member** Overleden
      - **Value set member** Vervallen
  - Tag definitions
    - **Tag** Basisberekening
    - **Tag** Ouderdomspensioen
    - **Tag** Partnerpensioen
    - **Tag** Wezenpensioen
    - **Tag** ANW extra
    - **Tag** WIA excedent AOV

¤ 3.3 Commutatiegetallen op 1 leven¶

$D_x = v^x * \frac{l_x}{100}$ ¤6 Dec(3)¶

Implemented in ¤ *V9401*¶

¶

$N_x = \sum_{t=0}^{\omega - x} D_{x+t}$ ¤7 Dec(3)

¶

¤ 3.6 Contante waarde 1 leven/ 2 levens¶

${}_nE_x = \frac{D_{x+n}}{D_x}$ ¤19 Dec(4)

¶

$a_x = \ddot{a}_x - 1$ ¤21 Dec(3)

¶

$\bar{a}_x = \ddot{a}_x - 0{,}5$ ¤22 Dec(3)

¶

$\ddot{a}_{x\overline{n|}} = \frac{N_x - N_{x+n}}{D_x}$ ¤23 Dec(3)

$\bar{a}_{x\overline{n|}} = \ddot{a}_{x\overline{n|}} - 0{,}5 + 0{,}5 * {}_nE_x$ ¤25 Dec(3)

¶

¤ 4 BN(_ris) koopsommen¶

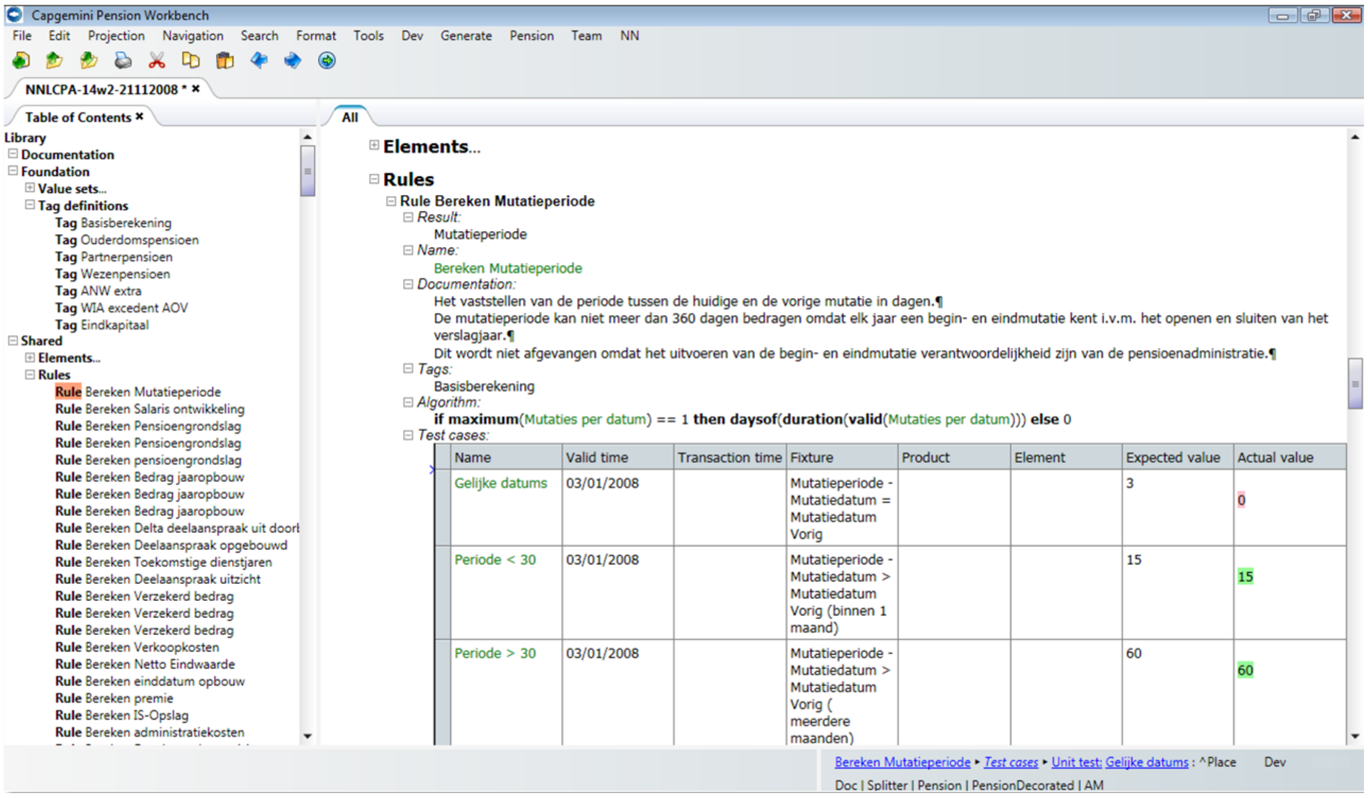
Figure 2.7: Example Code written using the Pension Plans language

2.4.2 *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.

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.

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 whose DSL discussion useless), so UML itself does not count as a DSL. However, UML provides profiles, which are a 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, only a very small part of the core UML is used (and you have to make sure your users don't use the other part) with the majority of concepts coming from the profiles. It is our 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.


Capgemini Pension Workbench
File Edit Projection Navigation Search Format Tools Dev Generate Pension Team NN
NNLCPA-14w2-21112008 *
Table of Contents
All

```
Library
⊟ Documentation
⊟ Foundation
  ⊞ Value sets...
  ⊟ Tag definitions
      Tag Basisberekening
      Tag Ouderdomspensioen
      Tag Partnerpensioen
      Tag Wezenpensioen
      Tag ANW extra
      Tag WIA excedent AOV
      Tag Eindkapitaal
⊟ Shared
  ⊞ Elements...
  ⊟ Rules
      Rule Bereken Mutatieperiode
      Rule Bereken Salaris ontwikkeling
      Rule Bereken Pensioengrondslag
      Rule Bereken Pensioengrondslag
      Rule Bereken pensioengrondslag
      Rule Bereken Bedrag jaaropbouw
      Rule Bereken Bedrag jaaropbouw
      Rule Bereken Bedrag jaaropbouw
      Rule Bereken Delta deelaanspraak uit doorb
      Rule Bereken Deelaanspraak opgebouwd
      Rule Bereken Toekomstige dienstjaren
      Rule Bereken Deelaanspraak uitzicht
      Rule Bereken Verzekerd bedrag
      Rule Bereken Verzekerd bedrag
      Rule Bereken Verzekerd bedrag
      Rule Bereken Verkoopkosten
      Rule Bereken Netto Eindwaarde
      Rule Bereken einddatum opbouw
      Rule Bereken premie
      Rule Bereken IS-Opslag
      Rule Bereken administratiekosten
```

```
⊞ Elements...

⊟ Rules
  ⊟ Rule Bereken Mutatieperiode
    ⊟ Result:
        Mutatieperiode
    ⊟ Name:
        Bereken Mutatieperiode
    ⊟ Documentation:
        Het vaststellen van de periode tussen de huidige en de vorige mutatie in dagen.¶
        De mutatieperiode kan niet meer dan 360 dagen bedragen omdat elk jaar een begin- en eindmutatie kent i.v.m. het openen en sluiten van het
        verslagjaar.¶
        Dit wordt niet afgevangen omdat het uitvoeren van de begin- en eindmutatie verantwoordelijkheid zijn van de pensioenadministratie.¶
    ⊟ Tags:
        Basisberekening
    ⊟ Algorithm:
        if maximum(Mutaties per datum) == 1 then daysof(duration(valid(Mutaties per datum))) else 0
    ⊟ Test cases:
```

Name	Valid time	Transaction time	Fixture	Product	Element	Expected value	Actual value
Gelijke datums	03/01/2008		Mutatieperiode - Mutatiedatum = Mutatiedatum Vorig			3	0
Periode < 30	03/01/2008		Mutatieperiode - Mutatiedatum > Mutatiedatum Vorig (binnen 1 maand)			15	15
Periode > 30	03/01/2008		Mutatieperiode - Mutatiedatum > Mutatiedatum Vorig (meerdere maanden)			60	60

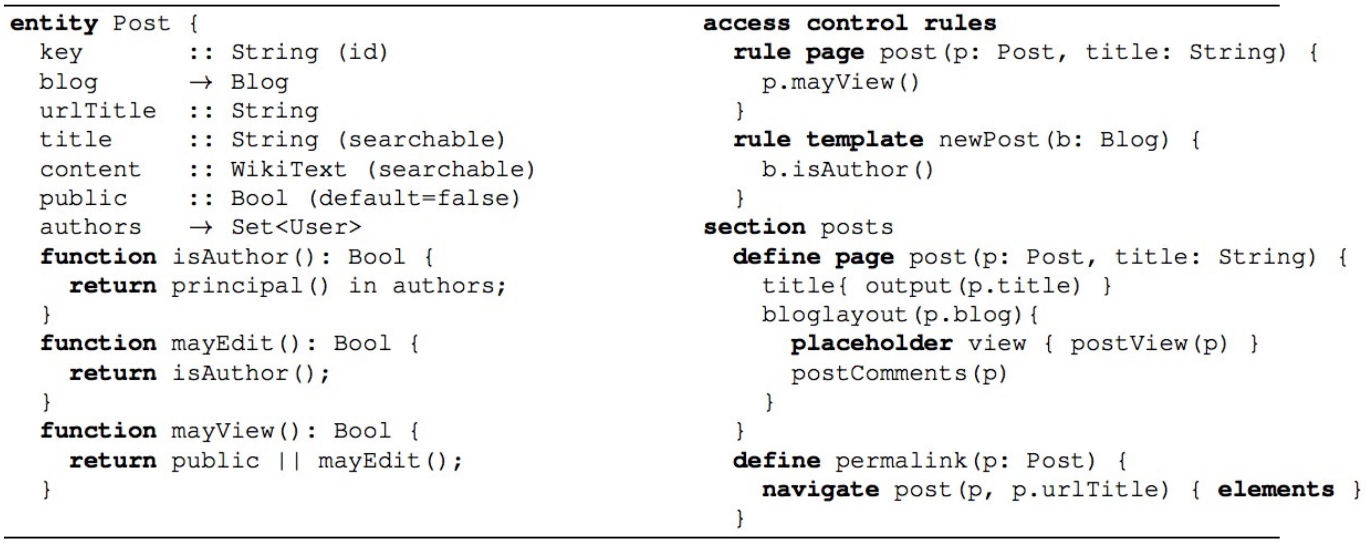
Bereken Mutatieperiode ▸ Test cases ▸ Unit test: Gelijke datums : ^Place Dev
Doc | Splitter | Pension | PensionDecorated | AM

Figure 2.8: Example Code written using the Pension Plans language

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 our decision to not cover it in any detail.

So is UML used in an MDD context? Sure. People build profiles and use UML-based DSLs, especially in large organizations where the perceived need for standardization is paramount[9]. As a cross-organizational example of an OMG-related standard: SysML is widely used in the aerospace/defense sector and has a lot of traction and applicability there. However, for "normal programmers", textual DSLs and the accompanying tools are just much more usable and pragmatic.

[9] It's interesting to see that even these sectors increasingly embrace DSLs. I did a couple of projects in the aerospace/defense sector where I replaced general-purpose, UML-based modeling with very specific and much more productive DSLs. It is also interesting to see how sectors define their own standard languages - I hesitate to call them DSLs, somehow. For example, the automotive industry is in the process of standardizing on AUTOSAR.



```
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 }
  }
```

Figure 2.9: Example Code written in WebDSL

2.4.3 *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 nature.

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.

The ideal tool environment 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 situation.

This book focusses mostly on textual languages; I discuss other forms of notations only peripherally.

TODO: Point at good books to read on this topic (DSM Book)

2.5 *Terminology*

I will introduce the detailed terminology around DSLs as I explain the concept in the book. However, there are a couple of important ones I want to define right away.

I use the terms model, program and code interchangeably because I think that any distinction is more or less artificial — see my discussion in the previous section. Code can be written in what is typically considered a programming language, or in a DSL. Sometimes DSL code and program code are mixed, so separating the two makes no sense.

As I elaborate on later, languages consist of a structure definition (called abstract syntax or meta model), one or more definitions of the notation (also called concrete syntaxes) and semantics. In semantics we distinguish between the static semantics (constraints, type system) and the operational semantics (what something means as it is executed). 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 (i.e. domain experts).

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 as part of the DSL development process. The execution engine can be changed, and bridges the gap between the DSL and the platform. So, when interpreting DSL models, the execution engine is the interpreter and is a program that runs natively on the target platform. In case of a generative approach, the execution engine is the code generator which produces code that can natively run on the target platform.

This process can be multi-stage. You can generate code that is then interpreted. Or you can generate code which is further transformed into something else, and so on. In this case each stage can be seen as a DSL-execution engine pair, cascading several of these combinations on top of each other.

I use the term *processor* to refer to the program that processes models expressed with a DSL. It may be a code generator, a model transformation, an interpreter, or even a sophisticated analysis tool.

The field discussed in this book has many names. These include Model-Driven Development, Domain-Specific Modeling, Generative Software Development, Language-Oriented Programming, Model-Integrated Computing, Model-Based Engineering, and, if you want to stretch it a bit, Model-Driven Architecture and End-User Programming. Now, I don't claim that all of these are the same and I am very aware of the differences. However, they all have a common core which is designing languages for a specialized/limited domain including their execution environments and IDEs. With this book, I aim to cover this core.

2.6 *How to read this Book*

The book covers three main aspects. Part I, DSL Design, deals with how to design good DSLs. Part II, DSL Implementation, looks at ingredients to DSL contruction, illustrated with a couple of different tools. Part III, Software Engineering with DSLs, looks at what you can do with DSLs. These three parts can be read more or less in any order, and some parts may be skipped. The parts are relatively self-contained and only rely on a common terminology introduced in this section.

Have fun reading the book!