1 Introduction

1.1 Why Modeling?

Using models to design complex systems is common in many engineering disciplines, including architecture (buildings), civil engineering (roads and bridges), automotive engineering (cars), and avionics (airplanes). Models have an ever-growing list of applications in these areas. Engineers build them to assess system properties early, or to steer construction, production, and servicing processes. For one system, usually different kinds of models have to be built, each of which providing a different perspective. For instance, three-dimensional models are used when designing the chassis of a car, while analog-circuit models describe its electrical system. Blueprint models are used in production, while yet different ones, such as maintenance and service models, are used later when servicing systems. All these examples of models describe structural and functional properties of real-world systems. However, models can also be used to describe and assess rather intangible properties that are neither structural nor functional, such as system reliability, power consumption, efficiency or production cost. We say that models are *purpose-specific* and *domain-specific*: they are tailored for a given purpose and carry the main characteristic aspects of the domain. For example, telephone-network switching models are very different from railway-track switching models.

Definition 1.1. *A model is an abstraction of reality made with a given purpose in mind.*

The main purpose of engineering models is to combat complexity: complexity of the problem, complexity of the solution design, and complexity of the system production. The understanding of complex problems, solutions, designs, and processes is possible thanks to *abstraction* (Selic, 2003). Abstraction is a simplification and elimination of information with respect to a given purpose. A model does not contain all information, but it preserves the information necessary to perform the intended application of the model. We can say that *"all models are wrong, but some are useful"* (Box and Draper, 1987). For instance, aesthetic information is typically not necessary to assess performance.

Models can not only *abstract* (or hide) information, but they can also *approximate* it. For instance, the Newtonian gravity model is sufficiently precise for applications in mechanical engineering. It is also widely applied, although we know that it is not precise. It is crucial that both abstraction and approximation are not adverse to the purpose of a given model. Abstraction should not hide relevant information, and approximation should only lead to acceptably small errors.

Models are increasingly electronic thanks to a rapid growth of computing technology. In fact, most of the engineering models are computer models today, even if they describe physical artifacts, such as buildings or diesel engines. Building computerized models is cheaper than

building physical models. Specifically, it allows animation, simulation, and computation of non-structural and non-functional properties, such as weight and force of gravity on various elements.

Even though computerized models have overtaken other engineering disciplines, models in software engineering are not as popular as elsewhere. Yet, the use of models in software engineering is growing. Many software engineers use purposeful abstractions of design and computation without thinking of them as models. For instance, relational-database queries are models, and so are HTML web-pages and their style sheets. Reactive algorithms are often described using automata models. Efficiency of algorithms is approximated using asymptotic complexity models. In this book, we shall look at multiple opportunities of using models in computing and of introducing purposeful domain-specific modeling languages (DSMLs, or DSLs for short) into the development of software systems.

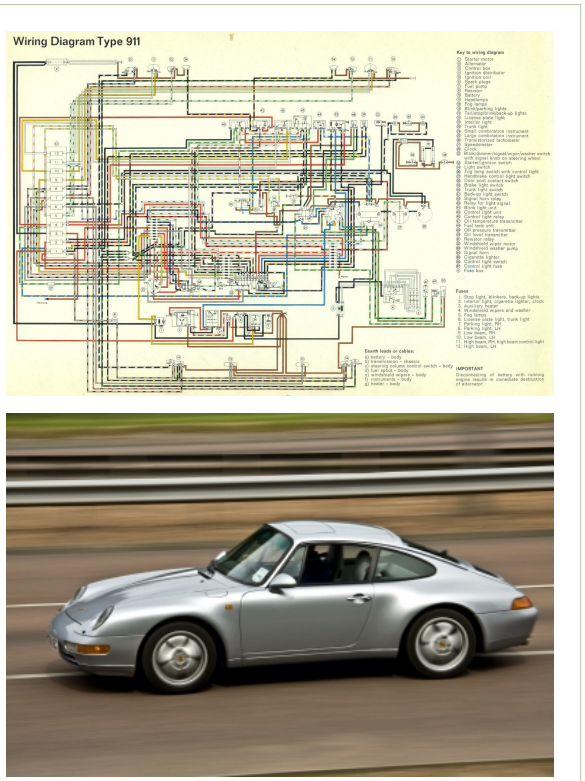

Figure 1.1
The electrical scheme of Porsche Carrera (above) and the actual system (below). Note the abstraction of information

There is no doubt that software engineers face the very same complexity problems that engineers in other disciplines have seen. In many ways, software systems are just as complex—commonly even more complex—than other achievements of engineering. Many commercial software systems would have more lines of code than the mechanical parts of a Boeing 747. This Jumbo Jet has 'only' six million parts, half of them being super-simple fasteners, many of them identical.[1] In December 2014, the Linux kernel had about 15 million lines of code. This complexity is (partly) controlled using a configuration model and an automated build build process (Berger et al., 2013). The Open Office productivity package has reached nine milion lines of code in 2012.[2] Microsoft Windows is reported to have exceeded 50 million lines in 2003. *"Therefore, it seems obvious that software systems, which are often among the most complex engineering systems, can benefit greatly from using models and modeling techniques"* (Selic, 2003), in order to combat the complexity.

1.2 Model-Driven Software Engineering

Software development is particularly suitable for the use of models. When building a car, there is a notable abstraction gap between a model and a real physical construction (recall Fig. 1.1). In computing, both models and systems are virtual digital objects, so this gap is much smaller. In fact, in software, everything is a model (Bézivin, 2005). It is possible

[1] http://www.boeing.com/commercial/747family/pf/pf_facts.html, retrieved 2015/02/10
[2] http://www.openoffice.org/FAQs/build_faq.html, retrieved 2015/02/10

Terminology of MDSE

The terminology established in the field of **MDSE** is relatively diverse. Beginners often struggle with the many synonyms that are used for one concept, and also the different usages of terms. In this book, we will use the terms **MDSE** (Model-Driven Software Engineering) and **DSL** (Domain-Specific Language).

MDSE is an umbrella term for the whole field of using models to *engineer* software. This *engineering* does not only comprise the development of software assets—as referred to with **MDSD** (Model-Driven Software Development), but also other activities in the lifecycle of a software, such as evolution and quality assurance. **MDE** (Model-Driven Engineering) and **MDD** (Model-Driven Development) in principle correspond to **MDSE** and **MDD**, but are supersets. They are broader approaches, not restricted to software, and comprise the engineering or development of further assets, such as hardware. **MDA** (Model-Driven Architecture) is often used synonymously to all these MDSE-related terms. However, it refers to a specific standard established by the Object Management Group (Group, 2014) describing a software-design process starting with domain modeling in order to obtain platform-specific models that can ultimately be executed (Frankel, 2003; Mellor, 2004).

Models in MDSE are defined in a language, which is often a **DSL** (Domain-Specific Language). A **DSL**, as opposed to a **GPL** (General-Purpose Language, such as Java, C# or Scala), focuses on expressing concepts in a specific domain. **DSL**s should be understandable by a domain expert; their strength is the reduced expressiveness compared to **GPL**s. **DSML** (Domain-Specific Modeling Language) refers to a subset of **DSL**—languages used specifically for domain-specific modeling. Further subsets of **DSL**s are domain-specific markup languages (e.g., XHTML, MathML) and domain-specific programming languages (e.g., Perl, shell-script languages). Although the majority of examples in this book concerns **DSML**s—since we use and develop languages to *model* software—we will use the term **DSL**, since we also consider markup languages and will implement so-called internal **DSL**s, which are embedded into a host **GPL**.

to refine models into a system in a continuous and often automatic manner, with much less effort than when designing cars or buildings. A computer program, more precisely, its source code in a programming language, is also a virtual object. The code is just yet another model that abstracts many aspects of a physical computation, but perhaps preserves more details than other models (enough to run the computation). This proximity of models and programs allows to to make modeling the central paradigm in development—the main idea of Model-Driven Software Engineering.

Definition 1.2. Model-Driven Software Engineering (MDSE) *is a software-engineering methodology that focuses on creating and exploiting models to produce software. The focus is on models, modeling, and model analysis as opposed to programs and programming. MDSE relies heavily on automation to produce code, to analyze system properties, and to support development activities.*[3]

Example 1.3. Yahoo Pipes[4] is a web application for building data mash-ups that aggregate web feeds, web pages, and other services into a single stream of information. Technically, Yahoo Pipes is also a domain-specific modeling language, an associated modeling environment, and an execution platform for models. Fig. 1.2 (left) depicts an example model. Yahoo Pipes is an example of a visual modeling language, which uses a graphical as opposed to a textual syntax. The model editor is implemented as a web application running inside a browser. The

[3] Compare with the definitions by Selic (2003) and by Wikipedia editors [Wikipedia, MDE, 2011/08/27]
[4] http://pipes.yahoo.com/pipes/ retrieved January 2015

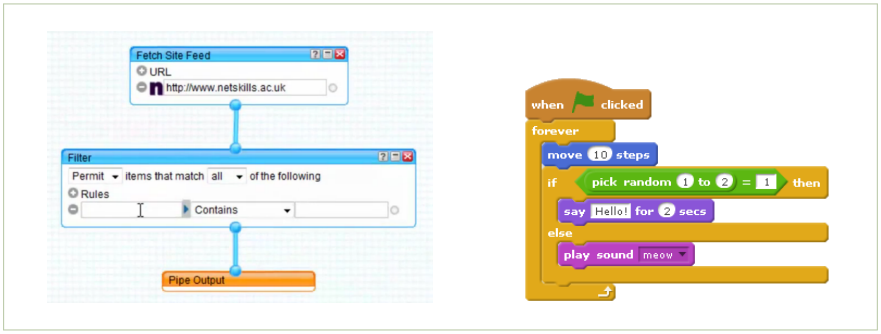

Figure 1.2
Facilitating end-user development with a DSL: Yahoo Pipes (left) and Scratch (right)

```
1   message Person {
2
3       enum PhType { MOBILE = 0; WORK = 1; }
4
5       message PhoneNo {
6           required string no = 1;
7           optional PhType type = 2 [default = MOBILE];
8       }
9
10      required string name = 1;
11      repeated PhoneNo phone = 2;
12  }
```

Figure 1.3
An example model in the Google Protocol Buffers language

model's syntax is similar to flow diagrams, so it is a kind-of *data-flow* model. The top node shows the source feed from which data will be extracted. The center node applies a filter to the data source. The last node is a rendering node that will display the output from a pipe. A Pipes model is formulated at a very high level of abstraction. Significant information is hidden. The user does not have to know about XML, RSS, parsing files, using web protocols, communication, and any imperative algorithm that executes the program and renders the pages. The semantics is provided by the underlying execution platform, presumably an interpreter. Interestingly, offline compilers also exist for this language, developed independently of Yahoo, so that Pipes programs can be used outside of the Pipes service. Thousands of Pipes models are hosted at the Yahoo Pipes website.

Example 1.4. Scratch[5] is another end-user oriented modeling language. Unlike Yahoo Pipes, Scratch is an imperative language, composed of control blocks akin to programming languages. Its semantics (not the syntax) is reminiscent of control-flow graphs. The syntax is designed to meet the expectations of the target user group, primary-school children. Scratch programs resemble jigsaw puzzles (see Fig. 1.2, right). Using a domain-specific syntax that matches the user expectations and the problem at hand well is one of the key success factors in designing DSMLs. Scratch boasted over a million users in 2014. The models are hosted on the scratch website and are freely accessible to the public.

Example 1.5. Google Protocol Buffers is a data modeling language aimed at flexible and efficient serialization and persistence of structured data across multiple programming languages and platforms. An example is shown in Fig. 1.3. Since the language was initially developed for passing messages between different machines in a request/response protocol, it uses 'message' as a metaphor for a data structure. It can be classified as a kind of textual

[5] http://scratch.mit.edu

```
1  class Client < ActiveRecord::Base
2      has_one :address
3      has_many :orders
4      has_and_belongs_to_many :roles
5  end
```

Figure 1.4
A simple data model in Ruby on Rails, using an internal DSL embedded in Ruby syntax

language (syntax is expressed as a stream of characters), an *interface definition language* and a *structural modeling language*, a competitor of, for instance, XML. However Protocol Buffers are small and clean, use very little bandwidth for transmitting the data and have a human readable syntax for schema, unlike XML. The example model describes a `Person` structure, with two properties `name`, and `phone`, where the latter is a message itself (a substructure) of type `PhoneNo`. The data elements described by this model are assigned ordinal numbers representing their placement in the serialization, which allows reordering and renaming the fields, without changing the message format. The format also allows specifying optional elements with default values. It is an important design criterion for the message serialization domain to allow as much backwards compatibility as possible, when the message format changes or its definition is refactored. Implementing a proper message format serialization infrastructure with these properties is actually cumbersome, even if it is needed in many software projects. Protocol buffers have a standalone implementation for at least python, Java, and C++. They demonstrate an important motivation for DSMLs and modeling: *code reuse*. The protocol buffers libraries have been implemented once for all and are maintained in only one place, saving a lot of effort duplication. Since they are used by many, the libraries are of considerably higher quality than if they would be reimplemented repetitively in different projects. In February 2014, there were 48,162 message types defined in 12,183 Protocol Buffers models across the Google code tree.

Example 1.6. Ruby is a dynamically typed, interpreted, object-oriented language, often used to implement DSMLs due to its flexibility. Ruby on Rails is the most well-known framework implemented in Ruby. It is a web-application framework that gathers information from the web server and the database and uses it to render web pages and interact with the users. Like in most web frameworks, the key element of a Ruby on Rails application is a data model expressed in an internal DSML. See Fig. 1.4 for an example. These models are used to scaffold the application using powerful code generators, as well as to access the database while it is operated. In Ruby on Rails we find examples of relational modeling, UI modeling, use of specialized editors for domain-specific models, and modeling of user interfaces.

Observe that similarly to Google Protocol Buffers, in Rails, models are mixed with code. Using modeling in software development does not exclude programming. Modeling is simply a more efficient way to program aspects of the systems that are well understood. For this reason, in practice, MDSE is often introduced into systems and domains that are already mature.

A common misunderstanding is that abstract models cannot be used effectively in software production, as they contain little information, not enough to generate systems from them. In all the above examples, automatic infrastructures complete the abstract models with concrete information, effectively turning them into programs. Once we add enough information to models, they effectively become programs, loosing all the advantages of modeling. This

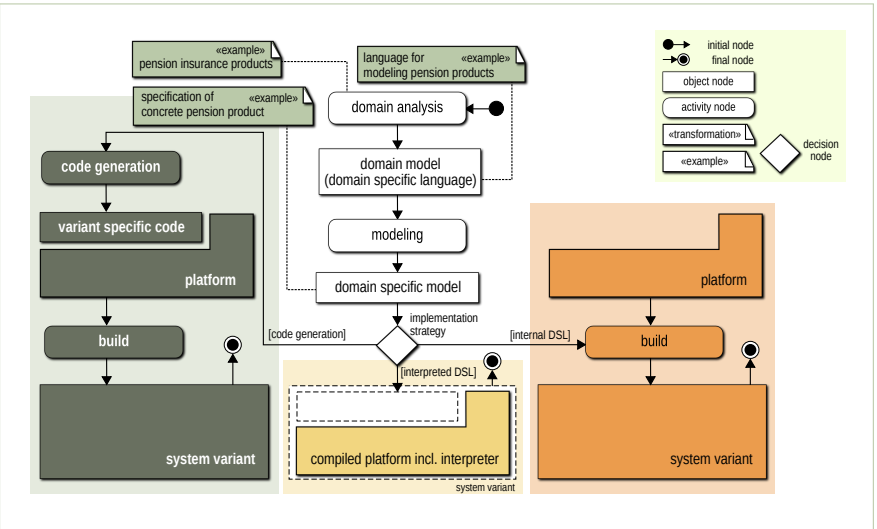

Figure 1.5
Basic processes and architectures used in MDSE with domain specific modeling languages (presented using UML Activity Diagrams syntax)

argument misses the fact that in MDSE there are two sources of information—the models are just one of them. The other one is the language definition (and then also the language implementation). A good modeling language captures the commonality of the domain in the language semantics and implementation, and let the aspects that vary across uses of the framework to be specified in models. Yahoo Pipes, Scratch and Protocol Buffers are extremely simple languages, yet one can derive complex systems out of their models. This is not unlike their programming languages, which also abstract details of computation that are common to all programs, in the compiler, in the execution platform, and in the hardware.

This book discusses methods and techniques for designing and implementing domain-specific modeling languages. We cover domain analysis, design of the syntax and semantics, as well as implementation of code generators and interpreters. Our primary goal is to learn how to design high-quality modeling languages at relatively little cost, so that they can be used in much smaller projects, or with a much smaller user base, than the above examples. We are focusing primarily on discussing technology and implementation, as we deeply believe that automation is key to the successful use of models in software projects.

Figure 1.5 summarizes the main steps and elements of an implementation of a modeling language. The white part is concerned with modeling. It begins with domain analysis and creation of the domain model (so called meta-modeling); then with creation of the actual models. We will cover this material in Chapter 2. In this diagram we show the process very abstractly. The completion of the DSL design will also require deciding on the concrete syntax and implementing it. Then a decision on how to implement the DSL is taken. The simplest way is to implement a direct interpreter (yellow). Two alternatives are sketched: one to the left (green) and one to the right (orange). The right choice (an *internal* DSL) is to embed the language syntax in a programing language. This way the models do not need to be directly interpreted. They just need to be linked with the right infrastructure (the platform) and then can be directly executed. The left choice, leading to a so called *external* DSL, uses a code generator to translate the model to the target language. Its output is then linked with the rest of the application platform to create a system variant. Google Protocol Buffers are an example of an external DSL, while the data modeling language in Rails is an internal DSL of Ruby (although Rails uses code generation for many purposes, so the picture there is somewhat

complex, as the framework is rather large).

1.3 Model-Driven Software Engineering in Practice

What can we gain by using modeling in software engineering?

Proximity of the domain and the implementation. We can express software design using concepts that are closer to the problem domain than to the implementation technology. We can stop talking about classes and loops, and instead consider business entities, cash-flow processes, and customers. In extreme cases, software can be tailored by domain experts or more technical users, when the modeling language used is designed with end-users in mind. A good example here are customizable enterprise systems which are implemented by software engineers who are highly skilled in software architectures and programming, but customized by business-domain consultants who know more about enterprise architectures and business processes. Also many computer games allow end-user extensions through various "mod" packages implemented as domain-specific programs (models). However, for most systems the benefit is reachable more easily by raising the abstraction level at which the developers work—like in the Protocol Buffers and Rails examples.

Improved maintainability. Software defined in domain terms is easier to maintain. Models are easier to understand than low-level code, and they can serve the role of documentation at times. It is easier to introduce new developers to tailor the systems using an abstract DSL instead of changing the low-level code, since most DSLs ensure that the changes stay within assumed design invariants.

Automatic code generation. Software or significant parts of it can be automatically generated from models. In this book, we will experience it when designing and implementing DSLs using MDSE methods. We will generate implementations of models, (de)serializers of models, instance generators for languages, tests and editors for models, and well-formedness checkers for models. Besides, in practice, *interpretation* is a very good (and very often better) alternative to code generation. Interpretation is often easier to implement and test, while maintaining the same benefits.

Simulation. Early simulation of models is one of the most powerful techniques used by hardware engineers and by engineers of embedded systems. Construction of an executable model allows simulating the behavior of the system before the system is actually constructed, and finding design mistakes early. However, software can be executed very quickly if automatic generation is used, so there is no acute need to simulate. Since software is virtual and does not require physical production processes, it is available as soon as it is designed.

Still, simulation makes sense for establishing properties of systems, when obtaining them directly from a running instance would be expensive or slow. For instance, virtual simulation of network protocols is much more efficient, than setting up a physical network infrastructure, deploying the implementations and running tests. Simulation makes sense for complex performance properties of many systems, as performance simulations can often be run much faster than observing the system in real time.

Models provide useful oracles and visualizations for system monitoring and debugging. Simulation can be used to mock components that are not yet implemented and to mock users

behavior. Alternatively, specifications of system behavior (models) can be linked to a running system for runtime monitoring. Errors would be flagged whenever the actual execution diverges from the specification given in a model. When program state, or problematic data, are visualised as models, debugging any potential divergence from the specification becomes much easier.

Instance Generation. A related activity to simulation is *instance generation*. Instances of data models can be generated to serve as test-data. Many design mistakes in data models can be established quickly by analyzing examples of unexpected instances of data, cases which should be disallowed.

Model Checking and Verification System models can be verified (for example for safety properties). Since code generation is used, we are highly confident that properties of the model are also properties of the final system. Model checking and verification is predominantly used for established modeling languages (such as MATLAB Simulink), since developing model verification infrastructure is unfortunately quite expensive, and requires advanced expertise. For this reason verification tools rarely exist for project specific languages.

Model-based system integration. If systems rely primarily on models, then data exchange and integration of systems can be done via models. This is particularly convenient, since models can be translated to models in other languages by *model transformations*, which are small programs implemented in languages specialized for model transformation.

Basic infrastructure easily available. As we will see, the mechanisms for serialization and deserialization of models, well-formedness checkers, tests, instances and rich editors can be obtained easily for domain specific languages. This means that working with models, is more effective than working with code capturing similar information. The support mechanisms of a programming language know little about the information you are editing. For instance, serializing complex data structures using programming language libraries usually does not work well; either it is not portable across program runs, or across machines, or introduces unacceptable changes to the data, or includes unnecessary runtime information. Similarly a Java or C# editor, can provide relative little feedback about errors in descriptions of business entities.

Hutchinson et al. (2011) attempt to provide solid scientific evidence on whether model-driven software engineering is commercially beneficial. To understand this, they have asked 250 individuals in diverse organizations to respond to a survey, and interviewed in depth 22 professionals using MDSE in 17 different companies. Table 1.1 shows their summary of positive and negative influences of model-driven development on the development process. A rather balanced view emerges in the table. It is useful to study it, to appreciate what MDSE can improve, and what it can spoil (so what cost it bears).

Table 1.2 quantifies the above influences by looking, which MDSE application areas lead to increase in productivity and maintainability and to what extent. High number in the "Increased" column means that most respondents apply the practice, and that they find it beneficial (it increases productivity or maintainability). Clearly all practices (but testing) are beneficial if they are used, however for testing the beenfit is less clear cut. Use of models for communication, design, documenting, and code generation are most wide spread among

Impact Factor	Illustrative Examples of MDE Influences	
	Positive Influences	**Negative Influences**
Productivity		
• *Time to develop code*	*Reduced* by: automatic code generation.	*Increased* by: time to develop computer-readable models; implement model transformations, etc..
• *Time to test code*	*Reduced* by: fewer silly mistakes in generated code; model-based testing methods, etc.	*Increased* by: effort needed to test model transformations and validate models, etc.
• *ROI on modeling effort*	*Positive* influences of modeling: more creative solutions; developers see the "bigger picture".	*Negative* influences of modeling: "model paralysis"; distracting influence of models.
Portability		
• *Time to migrate to a new platform*	*Reduced* by: simply applying a new set of transformations.	*Increased* by: effort required to develop new transformations or customize existing ones.
Maintenance		
• *Time for stakeholders to understand each other*	*Reduced* since: easier for new staff to understand existing systems; code is "self-documenting".	*Increased* since: generated code may be difficult to understand.
• *Time needed to maintain software*	*Reduced* since: maintenance done at the modeling level; traceability links automatically generated.	*Increased* since: need to keep models/code in sync, etc.

Table 1.1
Illustrative influences of MDSE as presented by Hutchinson et al. (2011)

practitioners.

Other findings of the survey include:

- 83% of respondents think that MDSE is a good thing and 5% that it is not.
- The majority of respondents considered the use of MDSE on their projects to be beneficial in terms of personal and team productivity, maintainability and portability (58-66%). However a significant number disagreed (17-22%). This suggests that there is some challenge in implementing MDSE successfully.
- MDSE users employ multiple modeling languages. 40% employ domain-specific languages (DSLs).
- Huge productivity gains are quoted in the qualitative study (at least two-fold, up to eight-fold), which are sometimes hidden or downplayed, to protect against cut downs.
- 50+ tools are used by the respondents, suggesting a lack of maturity—definitive market leaders are yet to emerge. Tools are immature, complaints about prices are common.

Let us remark that this study is a good example of an assessment in software engineering. For students it shows what can be asked about quality of your research project results (for example for the evaluation chapter of your thesis). For practitioners, it demonstrates what questions one should ask himself about the development process when one considers modifying it.

Further Reading

Besides the papers of Selic (2003) and of Hutchinson et al. (2011) discussed above, a good overview of what MDSE offers to software developers can be found in the first two chapters of the book by Stahl and Völter (2005). In his newer book, Voelter (2013) focuses on using and developing domain-specific languages. This book is more comprehensive than ours, but we focus on presenting the material in style and structure that is suitable for use in a university course, without assuming an extensive training in compiler theory. A somewhat more stan-

Activity	Productivity		Maintainability	
	Increased	Not Used	Increased	Not Used
Use of models for team communication	73.7%	7.0%	66.7%	6.7%
Use of models for understanding a problem at an abstract level	73.4%	4.8%	72.2%	6.1%
Use of models to capture and document designs	65.0%	9.3%	59.9%	10.7%
Use of domain-specific languages (DSLs)	47.5%	32.6%	44.0%	33.7%
Use of model-to-model transformations	50.8%	24.6%	42.6%	28.4%
Use of models in testing	37.8%	33.9%	35.2%	32.4%
Code generation	67.8%	12.0%	56.9%	12.6%
Model simulation/ Executable models	41.7%	38.3%	39.4%	35.9%

Table 1.2 The impact of model-driven software engineering activities on productivity and maintainability of software according to Hutchinson et al. (2011)

dard presentation of DSL design is given by Fowler and Parsons (2011), who thoroughly and excellently discuss the patterns and guidelines for implementing and using DSLs. However, they are less focused on obtaining the implementation with the modern tools at low cost, so their implementation of DSLs is not as much model-driven as presented in our book. Brambilla et al. (2012) present a very good overview of model-driven development architectures, processes, and benefits. In our opinion, their book is very suitable for experienced software developers, who appreciate the software engineering issues solved by MDSE, and who are trained in (programming-) language design and implementation.

Finally, Bettini (2013) gives a very pragmatic, even hands-on, course on development of textual DSLs with the Xtext framework. Since this is the same framework as used in this book, Bettini's volume is a very convenient companion. While we focus more on general aspects of language design, and present the methods as far as possible in a tool independent manner, Bettini explains how to realize the concrete design with the Xtext tooling.

Exercises

Exercise 1.1. Chef[6] is a deployment and configuration management language. It has started as an internal DSL implemented in Ruby and has grown out into a proper external DSL. Analyze the language from the perspective of concepts mentioned in this chapter. What is the domain described by models in this language? What information is present, what is abstracted away, hidden? What is the style of this language (syntax, semantics)? What tasks are automated thanks to Chef? From where the Chef infrastructure takes information to execute simplistic models? Browsing through these slides http://www.slideshare.net/chef-software/overview-of-chef-fundamentals-webinar-series-part-1 should suffice for this discussion.

[6] https://learn.chef.io/

This is an open question, with no perfect answer. It is meant to help you explore the concepts. If you feel so inclined, you can replace Chef with any other DSL, or you can try the same question on other DSLs additionally. Chef itself is of no particular importance for the rest of the book.