**brief contents**

**PART 1 INTRODUCING DOMAIN-SPECIFIC LANGUAGES ..................1**

1 ■ Learning to speak the language of the domain 3
2 ■ The DSL in the wild 25
3 ■ DSL-driven application development 54

**PART 2 IMPLEMENTING DSLS ..................................................85**

4 ■ Internal DSL implementation patterns 87
5 ■ Internal DSL design in Ruby, Groovy, and Clojure 128
6 ■ Internal DSL design in Scala 166
7 ■ External DSL implementation artifacts 211
8 ■ Designing external DSLs using Scala parser combinators 241

**PART 3 FUTURE TRENDS IN DSL DEVELOPMENT .......................275**

9 ■ DSL design: looking forward 277
Every morning on your way to the office, you pull your car up to your favorite coffee shop for a Grande Skinny Cinnamon Dolce Latte with whip. The barista always serves you exactly what you order. She can do this because you placed your order using precise language that she understands. You don’t have to explain the meaning of every term that you utter, though to others what you say might be incomprehensible. In this chapter, you’ll look at how to express a problem in the vocabulary of a particular domain and subsequently model it in the solution domain. The implementation model of this concept is the essence of what is called a domain-specific language (DSL). If you had a software implementation of the coffee shop example where a user could place an order in the language that they use every day, you would have a DSL right there.
Every application that you design maps a problem domain to the implementation model of a solution domain. A DSL is an artifact that forms an important part of this mapping process. You’ll look more closely at the definition of what a DSL is a bit later. First, you need to understand the process that makes this mapping possible. For this mapping to work, you need a common vocabulary that the two domains share. This vocabulary forms one of the core inputs that lead to the evolution of a DSL.

A good abstraction is essential to a well-designed DSL implementation. If you want to dig deep into the subject of well-designed abstractions, appendix A has a detailed discussion about the qualities to look for. A good plan of attack is to skim the appendix now, then continue reading this chapter. Section 1.7 contains basic information about abstractions, but appendix A is much more detailed.

1.1 The problem domain and the solution domain

Domain modeling is an exercise that helps you analyze, understand, and identify the participants involved in a specific area of activity. You start with the problem domain and identify how the entities collaborate with each other meaningfully within the domain. In the earlier example of the coffee shop, you placed your order in the most natural language of the domain, using terminology that mapped closely to what the barista understands. Terminology forms the core entity of the problem domain. The barista could easily figure out exactly what she needed to serve you to fulfill your request because you’re both familiar with the required terminology.

1.1.1 The problem domain

In a domain modeling activity, the problem domain is the processes, entities, and constraints that are part of the business that you’re analyzing. Domain modeling, also known as domain analysis (see [1] in section 1.9), involves the identification of all the major components of the domain and how they collaborate. In the example you began with, the barista knew all the entities like coffee, whipped cream, cinnamon, and nonfat milk that formed her problem domain model. When you analyze a more complex domain like a trading and settlement system for financial brokers, securities, stocks, bonds, trade, and settlement are some of the components that belong to the problem domain. Along with these components, you’ll also study how securities are issued, how they’re traded in stock exchanges, settled between various parties, and updated in books and accounts. You identify these collaborations and analyze and document them as artifacts of your analysis model.

1.1.2 The solution domain

You implement a problem domain analysis model in terms of the tools and techniques offered by the solution domain. The barista could map your order to the procedure that she needed to follow to serve your Grande Skinny Cinnamon Dolce
The problem domain and the solution domain

Latte. The process she followed and the tools she used formed parts of her solution domain. When you’re dealing with a larger domain, you might need more support from your solution domain in terms of the tools, methodologies, and techniques that it needs to offer. You need to map the problem domain components into appropriate solution domain techniques. If you use an object-oriented methodology as the underlying solution platform, then classes, objects, and methods form the primary artifacts of your solution domain. You can compose these artifacts to form larger ones, which might serve as better representations of higher-level components in your problem domain. Figure 1.1 illustrates this first step in domain modeling. As you move along, you’ll flesh out the process of how to get to the solution domain by using techniques that domain experts can understand throughout the lifecycle of transformation.

The primary exercise involved in domain modeling is mapping the problem domain to artifacts of the solution domain, so that all components, interactions, and collaborations are represented correctly and meaningfully. To do this, you first need to classify domain objects at the proper level of granularity. When you correctly classify domain objects, each object of the problem domain is visible in the solution domain, with its proper structure and semantics. But your map can be only as good as the language of interaction between the domains. A solid interaction requires that the problem domain and the solution domain share a common vocabulary.

![Figure 1.1](image-url)

Figure 1.1  Entities and collaborations from the problem domain must map to appropriate artifacts in a solution domain. The entities shown on the left (security, trade, and so on) need corresponding representations on the right.
1.2 Domain modeling: establishing a common vocabulary

When you start an exercise in domain modeling, you start with the problem domain that you’re going to model. You need to understand how the various entities of the domain interact among themselves and fulfill their responsibilities. While you’re figuring all this out, you collaborate with domain experts and with other modelers. Domain experts know the domain. They communicate using the domain vocabulary, and use the same terminology when they explain domain concepts to the outside world. The modelers know how to represent an understanding of the model in a form that can be documented, shared, and implemented by software. The modelers must also understand the same terminology and reflect the same understanding in the domain model that they’re designing.

Sometime back I started working on a project that involved modeling the back-office operations of a large financial brokerage organization. I wasn’t a domain expert, and I didn’t know much about the details and complexities involved in the practices of the securities industry practices. Now, after working in that domain for quite a while, I think it’s similar enough to other domains that you might deal with to model most of my examples and annotations in this book on that domain. The sidebar in this section gives a brief introduction to the domain of securities trading and financial brokerage, which you’ll use as running examples for implementing DSLs. As you progress, I’ll define new concepts wherever applicable and focus on the relevant details only when necessary. If you’re not familiar with what goes on in a stock exchange, don’t panic. I’ll give you enough background in the sidebars to help you understand the basic concepts of what you model.

On the first day of our requirements analysis meeting, the domain specialists of the financial industry started talking about coupon bonds, discount bonds, mortgages, and corporate actions. These terms were part of the usual terminology that a

---

Financial brokerage systems: a background

The business of financial brokerage starts with a trading process. This process involves the exchange of securities and cash between two or more parties, referred to as the counterparties of the trade. On a certain date, the counterparties promise to make the trade (this date is referred to as the trade date) at a place known as the stock exchange, based on an agreed upon price, known as the unit price. The securities, which form one leg of the exchange process (the other being cash), can be of several types, such as stocks, bonds, mutual funds, and a host of other types that can have a hierarchy of their own. There are, for example, several types of bonds, like coupon bonds and discount bonds.

Within a certain number of days of the promise to trade, the exchange is made by transferring the ownership of funds and securities between the counterparties; this exchange is known as the settlement process. Each security type has its own lifecycle of trade, execution, and finalization, and passes through a series of state changes in the course of the trading and settlement process.
brokerage specialist uses to communicate, but I didn’t know what they meant. Also, lots of terms were being used synonymously. The terms discount bond and zero coupon bond are synonymous, and they were being used interchangeably by different domain experts in different contexts. But because these terms were unknown to me, confusion reigned. Not all of us were specialists in the financial industry, and we soon realized that we needed to share a common vocabulary to make the knowledge-sharing sessions more meaningful. Not only did we collaborate in terms of the common domain vocabulary, we also made sure that the model we designed and developed spoke the same language—the natural language of the domain.

1.2.1 Benefits of a common vocabulary
A common vocabulary, shared between the stakeholders of the model, serves as the binding force that unifies all artifacts that are part of the implementation. More importantly, with the common vocabulary in place, you can easily follow the path of features, functions, and objects across all phases of the project delivery cycle. The same terms that the modeler uses for documenting use-cases appear as module names in programs, entity names in data models, and object names in test cases. In this way, a common vocabulary bridges the gap between the problem domain and the solution domain. Creating a common vocabulary might take more time up-front than you’re initially willing to spend, but I can almost guarantee that you’ll save yourself a lot of redoing in the long run. Let’s look at some of the tangible benefits that a common vocabulary offers.

Shared vocabulary as the glue
During the requirements analysis phase, a shared vocabulary serves as the common bridge of understanding between the modelers and the domain experts. All your discussions are more succinct and effective. When Bob (who’s a trader) talks about interest accrual for bonds, Joe (who’s a modeler) knows that Bob is referring specifically to coupon bonds.

Common terminology in test cases
The common vocabulary can also serve as the basis for developing test cases. Then, the domain expert group can verify these test cases. A sample test case from my earlier project on brokerage system implementation reads: For a zero coupon bond issued by Trampoline Securities with a face value of USD 10,000 and a primary value date of 15th May 2001 at a price of 40%, the investor will have to pay USD 4,000 at issue launch. The test case makes perfect sense to the modeler, the tester, and the domain specialist who’s reviewing it, because it uses terminology that forms the most natural representation of the domain language.

Common vocabulary during development
If the development team is using the same vocabulary to represent program modules, the resulting code is also going to speak the same domain language. For example, if you talk about modules like bond trading and settlement of securities, when you write code, you’ll use the same vocabulary to name domain entities.
CHAPTER 1  Learning to speak the language of the domain

Developing and sharing a common vocabulary between the problem and solution domains is the first step in our march toward the solution domain. Let’s update figure 1.1 with this common glue that binds the domains together to come up with figure 1.2.

You know that the developers and the domain experts need to share a common vocabulary, but how will the language be mapped? How does the domain expert understand the model that the developers are generating? This communication problem is a common one in any software development ecosystem.

Looking at figure 1.2, you’ll realize that the domain experts are in no way equipped to understand the technical artifacts that currently populate the solution-domain model. As systems increase in complexity, the models get bloated and the communication gap keeps on widening. The domain experts don’t need to understand the complexities that surround an implementation model; they need to verify whether the business rules being implemented are correct. Ideally, the experts themselves would write test scripts to verify the correctness and comprehensiveness of the domain rules’ implementation, but that’s not a practical solution.

What if you could offer the experts a communication model that builds on the common vocabulary and rolls off everyone’s tongue with the same fluidity that a domain person uses in his everyday business practice? You can. This is the moment when the DSL enters the picture!

1.3  Introducing DSLs

Joe, the IT head for the hypothetical company Trampoline Securities, had no idea what Bob, the trader, was up to as he leaned over Bob’s shoulders and took a sneak
peek at his console. To his amazement, Joe discovered that Bob was busy typing commands and statements in a programming environment that he thought belonged exclusively to the members of his development team. Here’s the fly-on-the-wall record of their conversation:

- Joe: Hey Bob, can you write programs?
- Bob: Yeah, sort of, in our new TrampolineEasyTrade system.
- Joe: But, but, you’re a trader, right?
- Bob: So? We use this software for that, too.
- Joe: You’re supposed to be *using* the software, not *programming* in it! The product isn’t even out of the development labs.
- Bob: But I thought it’d be great if I could write some tests for the software that I’ll be using later. That way, I can pass on my inputs to the development team way early in the sprint. Being part of this exercise makes me feel like I’m contributing more. I have a much better feel for what’s being developed. And I can check to see if my use cases are working, too.
- Joe: But that’s the responsibility of the development team! I sit with them every day. I’ve got tools in place to check code coverage, test coverage, and a bunch of other metrics that’ll guarantee that what we deliver is the best it can be.
- Bob: As far as knowing about financial brokerage systems is concerned, who do you think understands the domain better? Me? Or your set of tools?

Ultimately Joe had to admit that Bob, who’s an expert in the domain of financial brokerage systems, was better equipped to verify whether their new offering of the trading platform covered the functional specs adequately and correctly. What Joe couldn’t understand is how Bob, who isn’t a programmer, could write tests using their testing framework.

As a reader, you must also be wondering. Look at the following listing, which shows what Bob had up on his console.

### Listing 1.1 Order-processing DSL

```plaintext
place orders { 
  new Order to buy(100 sharesOf "IBM") 
    limitPrice 300
    allOrNone
    using premiumPricing,
  new Order to buy(200 sharesOf "CISCO") 
    limitOnClosePrice 300 
    using premiumPricing,
  new Order to buy(200 sharesOf "GOOGLE") 
    limitOnOpenPrice 300 
    using defaultPricing,
  new Order to sell(200 bondsOf "SUN") 
    limitPrice 300
    allOrNone
```
Looks like a code snippet, right? It is, but it also contains language that Bob usually speaks when he’s at his trading desk. Bob’s preparing a list of sample order-creation scripts that place orders on securities using various pricing strategies. He can even define a custom pricing strategy on his own when he places the order.

What’s the language that Bob’s programming in? It doesn’t matter to him, as long as he gets his work done. To him, it’s the same language that he speaks at his trading desk. But let’s determine how what Bob is doing differs from the run-of-the-mill coding that we do every day in our programming jobs:

- The vocabulary of the language that Bob is using seems to correspond closely with the domain that he belongs to. In his day job at his trading desk, he places orders for his clients using the same terminology that he’s writing directly into his test scripts.
- The language that he’s using, or the subset of the language that you see on his console, doesn’t seem to apply outside the domain of financial brokerage business.
- The language is expressive, in the sense that Bob can clearly articulate what he wants to do as he steps through the process of creating a new order for his client.
- The language syntax looks succinct. The syntactic complexities of the high-level languages you usually program in have magically disappeared.

Bob is using a domain-specific language, tailor-made for financial brokerage systems. It’s immaterial at this point what the underlying language of implementation is. The fact that the underlying language isn’t obvious from the code in listing 1.1 indicates that the designer successfully created an expressive language for a specific domain.

### 1.3.1 What’s a DSL?

A DSL is a programming language that’s targeted at a specific problem; other programming languages that you use are more general purpose. It contains the syntax and semantics that model concepts at the same level of abstraction that the problem domain offers. For example, when you order your Cinnamon Latte, you use the domain language that the barista readily understands.

**DEFINITION** Abstraction is a cognitive process of the human brain that enables us to focus on the core aspects of a subject, ignoring the unnecessary details. You’ll talk more about abstractions and DSL design in section 1.7. Appendix A is all about abstractions.

Programs that you write using a DSL must have all the qualities that you expect to find in a program that you write in any other computer language. A DSL needs to give you
the ability to design abstractions that form part of the domain. In the same way that you can build a larger entity out of many smaller ones in the problem domain, a well-designed DSL gives you that flexibility of composition in the solution domain. You should be able to compose DSL abstractions just like you compose your functionalities in the problem domain.

Now you know what a DSL is. Let’s talk about how it differs from other programming languages you’ve been using.

**How’s a DSL different from a general-purpose programming language?**

The answer to the difference is in the definition itself. The two most important qualities of a DSL that you need to remember are:

- A DSL is targeted at a **specific problem area**
- A DSL contains syntax and semantics that model concepts at the **same level of abstraction as the problem domain**

When you program using a DSL, you deal only with the complexity of the problem domain. You don’t have to worry about the implementation details or other non-essential elements of the solution domain. (For more discussion about nonessential complexity, see appendix A.) More often than not, people who aren’t expert programmers can use DSLs—if the DSL has the appropriate level of abstraction. Mathematicians can easily learn and work with Mathematica, UI designers feel comfortable writing HTML, hardware designers use VHDL (very-high-speed integrated circuit hardware description language; a DSL used in electronic design automation) to name a few such use cases. Because nonprogrammers need to be able to use them, DSLs must be more intuitive to users than general-purpose programming languages need to be.

You write a program only once, but you manage its evolution for many years. For a program to evolve, it needs to be nurtured by people, many of whom may not have been involved in designing the initial version. The key issue is communication, the ability for your program to communicate with its intended audience. In the case of a DSL, the direct audience is neither the compiler nor the CPU, but the human minds that need to understand its behavior. The language needs to be communicative to its audience and allow code snippets that are expressive enough to map to the thought process of the domain modeler. For this to happen, the DSL that you design has to offer the correct level of **syntactic and semantic** abstractions to the user.

**What’s in a DSL for business users?**

As you’ve learned from the discussion so far, DSLs stand out from normal high-level programming languages in two ways:

- DSLs offer a higher level of abstraction to the user. This implies that you don’t have to be concerned about the nuances of identifying specific data structures or other low-level details. You can focus on solving the problem at hand.
- DSLs offer a limited vocabulary that’s specific to the domain it addresses. The fact that it contains nothing extra helps you focus on the problem that you’re
CHAPTER 1 Learning to speak the language of the domain

modeling. A DSL doesn’t have the horizontal, spread-out focus of a general-purpose programming language.

Both these qualities make DSLs a friendlier tool for the nonprogramming domain expert. Your business analysts understand the domain, which is what a DSL abstracts.

With more and more programming languages offering higher levels of abstraction design, DSLs are poised to be a major component in today’s application development ecosystem. Nonprogramming domain analysts will surely have a major role to play here. With a DSL implementation in place, they’ll be able to write test scripts correctly from day one. The idea isn’t to run the scripts immediately, but to ensure that you’ve adequately covered the possible business scenarios in your implementation. When the DSL is designed at an effective level of abstraction, it’s not unusual for domain experts to browse through source code that defines the business logic. They’ll be able to verify the business rules, and provide immediate feedback to developers based on their observations.

Now that you’ve seen some of the values that a DSL offers to you as a developer and as a domain user, let’s take a look at some of the commonly used DSLs in the industry today.

1.3.2 Popular DSLs in use

DSLs are everywhere. Whether or not you brand them as DSLs, I’m sure you’re using a lot of them in every application that you develop. Table 1.1 lists a few of the most commonly used DSLs.

Table 1.1 Commonly used DSLs

<table>
<thead>
<tr>
<th>DSL</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
<td>Relational database language used to query and manipulate data</td>
</tr>
<tr>
<td>Ant, Rake, Make</td>
<td>Languages for building software systems</td>
</tr>
<tr>
<td>CSS</td>
<td>Stylesheet description language</td>
</tr>
<tr>
<td>YACC, Bison, ANTLR</td>
<td>Parser-generator languages</td>
</tr>
<tr>
<td>RSpec, Cucumber</td>
<td>Behavior-driven testing language in Ruby</td>
</tr>
<tr>
<td>HTML</td>
<td>Markup language for the web</td>
</tr>
</tbody>
</table>

There are a lot more DSLs that you use on a regular basis. Can you identify some of the common characteristics that these languages have? Here are a few:

- All DSLs are specific to the domain. Each language is of limited expressivity; you can use a DSL to solve the problem of that particular domain only. You can’t build cargo management systems using only HTML.

**DEFINITION** Martin Fowler used the term limited expressivity to describe the most important characteristic of a DSL. In his 2009 DSL Developer’s
Conference keynote talk ([3] in section 1.9), Martin mentioned that it’s this *limited expressivity* that differentiates a DSL from a general-purpose programming language. You can model anything and everything with a general-purpose programming language. With a DSL, you can model only one specific domain, but in a more expressive way.

- For each of the languages listed in table 1.1 (and the other popular ones being used), you usually need to use the *abstractions* that they publish. Barring specific exceptions, you don’t even need to know the underlying *implementations* of these languages. Every DSL offers a set of contracts that you can use to build your solution domain model. You can compose multiple contracts to build more complex models. But you don’t need to step out of the offered contracts and get down to the implementation level of the DSL.

- Every DSL is expressive enough to make its intentions clear to the nonprogramming user. The DSL isn’t merely a collection of APIs that you use; every API is concise and speaks the vocabulary of the domain.

- For every DSL, you can go back to your source file months after you wrote them and immediately be able to figure out what you meant.

It’s a fact that DSL-based development encourages better communication between developers and domain experts. This is its greatest virtue. By using a DSL, a nonprogramming domain expert won’t transform himself into a regular programmer. But with the expressiveness and explicitly communicative APIs that DSLs offer, the domain expert will be able to understand which business rules the abstraction implements and whether it adequately covers all possible domain scenarios.

Let’s look at one motivating example of a DSL snippet selected from the list in table 1.1. Consider the following snippet from a Rakefile, which is mainly used to build Ruby-based systems:

```ruby
desc "Default Task"
task :default => [:test]
Rake::TestTask.new { |t|
  t.libs << "test"
  t.pattern = 'test/*_test.rb'
  t.verbose = true
  t.warning = false
}
```

This code snippet creates a number of unit tests that can be run as the default task. Even if you don’t know Ruby, this snippet means the same thing to you; it’s just as expressive to you. How can that be? The snippet has explicit hotspots that match vocabulary you’re familiar with and provides an easy-to-use interface to the user of the DSL. In this case, Rake will be used by the developer. The language of the code uses semantics that match the level of abstraction that a developer expects and understands. Similarly, if you develop a DSL for the trader community, you need to keep in mind the level of expressiveness that suits the expectations and experiences of a trader.
CHAPTER 1 Learning to speak the language of the domain

at the dealing desk. This section contains a sidebar that has a short introduction to some of the basic terminology of the trading system. Have a look at the definitions because you’ll be using many of them in the example DSLs that you’ll develop over the course of the book.

When you design a DSL, keep your target users in mind. A DSL needs to be as expressive and granular as necessary for the user to understand it. In the following chapters, you’ll learn how to design DSLs at the level of abstraction that feels most natural to users. Meanwhile, let’s fill in some of the missing links in figure 1.2 so you’ll have a more complete picture of how DSLs enable a better mapping between the problem and the solution domain.

1.3.3 Structure of a DSL

Look at figure 1.3, which shows how a DSL script binds the common vocabulary to the underlying implementation model of the solution domain.

The following describes the three principles that a well-designed DSL embodies to make your software more communicative to domain users:

- A DSL provides a direct mapping to the artifacts of the problem domain. If the problem domain has an entity named Trade, the DSL script must contain the same abstraction that plays the same role.
- The DSL script must use the common vocabulary of the problem domain. The vocabulary becomes the catalyst for better communication between developers and business users. When business users interact with the software domain model, the DSL script is their interface, as shown in figure 1.3.
The DSL script must abstract the underlying implementation. This principle is an important part of good abstraction design, and it applies to DSLs as well. The DSL script cannot contain accidental complexities that deal with implementation details.

In figure 1.3, the relationships shown between the node labeled DSL script and the other nodes illustrate these three principles. If you keep these principles in mind as you design your DSL, your software will communicate effectively to domain users. In the next section, you’ll look at the execution model of a DSL—how the DSL script and its implementation model is realized when you run your application.

### 1.4 Execution model of a DSL

Domain experts use the DSL script to understand the domain model and business rules. You, as a developer, need to implement the DSL in terms of an underlying technology platform. In most cases, a DSL is nothing but a layer of abstraction over the host language that presents a domain-friendly interface to the business users. (It’s not always the host language. See section 1.5 for details about DSL classification.) You’re kind of extending the host language to implement another language on top of it. This concept is sometimes referred to as a *metalinguistic abstraction*. You’ll also come across DSLs that don’t use an embedded language for implementation. Maybe it uses a custom language that the team designed specifically for implementing the DSL. In
section 1.5, you’ll look more closely at how DSL implementations are classified. For now, let’s talk about how you execute a DSL script.

Figure 1.4 shows the three most common ways to execute a DSL script.

1. The script can directly execute the underlying model without any more code generation or manipulation. There might be an interpreter that directly interprets the script and runs it. The UNIX little programming languages awk and sed are examples of DSLs that execute directly.

2. A DSL script that’s developed on a virtual machine follows the second model. The semantic model underlying any Java DSL script generates bytecodes that are executed on the JVM.

3. Some languages offer compile-time metaprogramming. When you’re developing a DSL using this kind of language, you build metastructures as part of your source code, which get translated to the normal forms of the language before it runs. Lisp supports this technique through *macros* that get expanded to normal Lisp forms during the macro expansion phase (I discuss this in more detail in appendix B). For these languages, there’s an intermediate stage where you have source code translation before the byte code is generated for the virtual machine.
Now that you’re comfortable with the three common models of execution for a DSL script, revisit the DSL in listing 1.1 that Bob was playing with. Irrespective of the language of implementation, you’ll discover that it also needs a semantic model as its underlying implementation. That model might be a host language like Ruby or Scala, or it might be a custom language that the developers at Trampoline Securities designed to implement the trading DSL.

Consider Ant, the popular build tool, and the XML-based DSL that it presents to the user. As a developer, when you look at the following XML snippet in Ant, you’ll find that it expresses familiar concepts. The code clearly spells out that it’ll build a jar as the target and that this task has a dependency on the task compile.

```xml
<target name="jar" depends="compile">
  <mkdir dir="${build.dist}"/>
  <jar jarfile="${build.dist}/${name}-${version}.jar">
    <fileset dir="${build.classes}" includes="***"/>
    <fileset dir="${src.dir}"
      <include name="*"/>
    </fileset>
  </jar>
</target>
```

This DSL script has an underlying semantic model; the implementation is in the form of Java classes, methods, and packages that create interfaces for tasks and dependencies. The developer doesn’t have to cross the boundaries of the DSL interface and dig down into the implementation in order to use Ant. Of course, there might be an exceptional situation when the developer might need to do so, because Ant is an extensible framework. But that’s only the exception.

So far, we’ve mostly been talking about DSL scripts that are designed as extensions of a host language, but that’s not the only kind of DSL script there is. You can also classify DSLs based on the way you implement them. The next section lays down a taxonomy of DSLs.

### 1.5 Classifying DSLs

A DSL speaks the language of the domain. The richer the domain, the more expressive the DSL needs to be. To the domain user, a DSL makes him understand the story of the domain that the developers have implemented as the underlying model. It doesn’t matter to him how the underlying model has been implemented, so long as he has coherent access to the domain abstractions through the DSL script.

The most popular way to classify DSLs is related to the way you implement them. Martin Fowler made this broad classification some time back and it’s recognized and followed by almost all practitioners in the industry today. He classifies a DSL as internal or external, depending on whether it’s been implemented on top of an existing host language. Internal DSLs are also known as embedded DSLs because they’re implemented as an embedding within a host language. (Internal DSLs will be discussed further in chapters 5 and 6 where you’ll implement DSLs using JVM languages like Ruby, Groovy,
18  
CHAPTER 1  Learning to speak the language of the domain

Scala, and Clojure.) External DSLs are also called standalone DSLs because they’re developed ground-up as an independent language, without using the infrastructure of an existing host language. Chapters 7 and 8 deal more with external DSLs.

Besides these two broad classifications, you’re also looking at newer paradigms of DSL development. Companies like Intentional Software (http://www.intentsoft.com/) have come out with tools you can use to create nontextual DSLs. Such developments and growing trends are subjects in chapter 9. For now, you will focus on the two main classifications and use examples to discuss some of their characteristics.

1.5.1 Internal DSLs

An internal DSL is one that uses the infrastructure of an existing programming language (also called the host language of the DSL) to build domain-specific semantics on top of it. One of the most popular internal DSLs used today is Rails, which is implemented on top of the Ruby programming language. When you write Rails code, you’re programming in Ruby, based on the semantics that Rails implements for developing web applications. In most cases, an internal DSL is implemented as a library on top of the existing host language. In section 2.1, you’ll develop an order-processing DSL as an example of an internal DSL, based on Java and Groovy as the host language. Figure 1.5 illustrates the structure of an internal DSL.

As you see in figure 1.5, the internal DSL script is a thin veneer over the abstractions of an underlying host language. Now let’s see what an external DSL looks like.

1.5.2 External DSLs

An external DSL is one that’s developed ground-up and has separate infrastructure for lexical analysis, parsing techniques, interpretation, compilation, and code generation. Developing an external DSL is similar to implementing a new language from scratch with its own syntax and semantics. Build tools like make, parser generators like YACC, and lexical analysis tools like LEX are examples of popular external DSLs. Of course, the complexity of an external DSL implementation depends on how rich you

![Figure 1.5 You implement an internal DSL using an existing host language and the infrastructure that it offers.](image-url)
Classifying DSLs

want it to be. In most cases, you’ll find that the external DSL doesn’t need to have all the complexities of a full-blown language. You’ll see many examples in chapters 7 and 8. Figure 1.6 shows how an external DSL is structured on top of a custom language infrastructure.

Figure 1.6 shows the generic components of an external DSL. In real-life examples, you might not need all of them or you might decide to combine components, depending on the complexity of your language.

Do you need to create a DSL in the form of a textual representation? Not always; a graphical representation can often be more self-explanatory. Let’s see how.

1.5.3 Nontextual DSLs

Besides internal and external DSLs, there’s a growing trend in the industry toward developing richer ways of modeling the domain. A DSL needs to be a representation of the domain but the definition doesn’t mandate that this representation or language needs to be a textual one. In fact, many claim that software code is too narrow a medium to adequately express domain knowledge. Some of the reasons that are often cited are:

- Text allows only limited notational freedom to express a domain problem.
- Many domain problems are better visualized by the domain user in the form of rich artifacts like spreadsheets or graphical models.
- In a text-based script, domain logic is often scattered within the maze of syntactic structures that are accidentally too complex.
- A domain expert is always more comfortable manipulating visual models than source code.

In response to these reasons, one other type of DSL is fast becoming the next-generation way to model and harvest domain knowledge. The domain user gets to see and process a representation of the domain knowledge through an editor called
the Projection Editor. The Projection Editor can project the appropriate view of the
domain to the user, which he can then manipulate without writing a single line of
code. At the back end, the Projection Editor can generate code that models the users’
intentions. Intentional’s DSL Workbench (http://www.intentsoft.com) and JetBrains’
Meta Programming System (MPS) (http://www.jetbrains.com/mps) are two examples
of rich DSL modeling tools. In chapter 9, you’ll see more such examples and the fea-
tures that they offer in the discussion of future trends of DSL-based development.

Classifying DSLs as internal, external, and nontextual is only one broad way of
looking at the types of implementations that DSLs can have. For all practical purposes,
you can consider the nontextual DSLs as external DSLs only, because the underlying
infrastructure that you use to develop DSL APIs isn’t a host language.

Now that you have a pretty good idea of what DSLs are and how you can use them
to improve communication between developers and domain users, what do you think
are some of the valid use cases for writing a DSL? Do you need to write a DSL for every
piece of code that you develop? Or are there specific circumstances that make a more
compelling case for DSL-based development?

1.6 When do you need a DSL?

Every application has business rules that need to be explicit, readable, and declarative. A DSL is an ideal way to model these kinds of rules. It doesn’t take a lot of effort
to develop a DSL that expresses a time period as 2.weeks.ago instead of time() –
1209600. But the impact that it has on users can be huge.

Should you use DSL-based development in your next project? Before you decide,
you need to weigh the pros and cons. As with any other technology, DSLs can have pit-
falls. As a developer, you’re the best person to judge whether you need a DSL for modeling the current problem. For that, you need to be aware of some of the common
advantages and disadvantages that DSLs offer.

1.6.1 The advantages

DSL-based development gets you more return on your investment when the complex-
ity of the domain is high. As I mentioned before, you’re going to use small DSL
gines in almost every project that you implement. When you’re planning for a com-
plex modeling project, you need to make a conscious decision and weigh your options
before making the final call. Following are some points that will help you weigh in on
your decision toward DSL-based development.

**DSLs are expressive**

They tend to provide a small, focused surface area for the APIs and deal with abstrac-
tions that speak the precise semantics of the domain. Users love them.

**DSLs are concise**

Because they’re concise, DSLs are easy to look at, see, think about, and show. Dan Roam
(see [2] in section 1.9) calls these the four steps to visual thinking. It’s the conciseness
of a DSL that reduces the semantic distance between the program and the problem.
**DSLS are designed at a higher level of abstraction**

DSLs don’t have to deal with lower-level language constructs, optimizing data structures, and other implementation techniques. Instead, DSLs embody domain knowledge at a level where it can be conserved, validated, and reused more easily than an implementation that’s based on a general-purpose programming language. This makes DSLs suitable for many nonprogramming domain experts.

**DSLS can give higher payoff**

DSL-based development tends to produce a higher payoff in the long run of your development lifecycle.

**DSL-based development is scalable**

If the project team has an imbalance of expertise in a specific programming language, expert programmers can focus initially on the implementation of the DSL. The rest of the team can then use the DSL. The DSL, because it’s at a higher level of abstraction, becomes easier to learn and can be used as the vehicle to scale up the development team.

As is the case with any other technology paradigm, DSL-based development has its share of advantages when you use it in a development cycle. We’ll talk more about DSL-based development in chapter 3. Next are some of the common pitfalls of DSLs that might cause heartache for your development project.

### 1.6.2 The disadvantages

All the disadvantages of DSLs relate to implementation overheads that incur additional cost in the software development lifecycle.

**Language design is hard**

DSL implementation is language design, and language design is a complex task that doesn’t scale. Instead of starting anew with the complexities of the lexers and grammars of your language, most DSLs are implemented as an embedding within a higher-level language. Still, it’s complex enough and is definitely not an exercise to be undertaken by nonexpert programmers. Later chapters cover language features and their suitability for implementing embedded DSLs.

**DSLS have an upfront cost**

DSL-based development has an upfront cost that you’ll incur in your project. Accepting this cost makes sense only when the model is at least moderately complex. You’ll eventually benefit when the cost factors level off during the later stages of the development cycle.

**Using DSLs can lead to performance concerns**

DSLs sometimes can cause performance concerns for your application. After all, it’s yet another layer of indirection. As a project manager, you need to consider factors like scale of deployment and scope of reusability when you’re deciding whether to use DSL-based development.
**DSLs sometimes lack adequate tool support**

Any development methodology needs rich tool support to scale out to the community of programmers. Tool support includes availability of IDE integrations, unit testing support, language workbenches, and profiling support to name a few. If your DSL generates multiple target languages for execution, interoperability between all the languages can also be a potential concern.

**Yet-another-language-to-learn syndrome**

Any external DSL has to be learned separately by the developers. With internal DSLs, all you have to learn is the interface that it publishes on top of the existing host language. But developers are often disturbed to find that not only do they have to learn yet another new language, but it’s one that has limited applicability.

**DSLs can lead to language cacophony**

Typically, when you develop an application, you need to use multiple DSLs. When you have multiple languages, there’s always the concern that when you combine them you won’t get a unified model for the domain. DSL composition isn’t easy, because individual DSLs tend to evolve independently of each other. Unless you manage it carefully, language multiplicity can lead to anarchy.

As you saw in figure 1.3, a DSL is a linguistic abstraction that’s on top of an underlying implementation model. The better you abstract your domain model, the easier it is to build a natural language on top of it. Let’s look at the qualities that the underlying model needs to have in order to be a strong foundation for an expressive DSL.

### 1.7 DSLs and abstraction design

In earlier sections of this chapter, I’ve used the term *abstraction* to loosely mean any artifact from the domain that exhibits a coherent set of behavior. *An abstraction focuses on the essential attributes of the subject, removing any unnecessary details from the user.* But what constitutes the essential parts depends on the perspective from which you view the abstraction. In this section, you’ll look at how abstraction is related to designing a DSL and what role it plays in making your DSL expressive.

As you’ll see in chapters 5 and 6, a well-designed abstraction is the foundation on which you build the linguistic layer of the DSL. But how do you make your abstractions well-designed?

From the criteria that make an abstraction optimal, I’ve identified four as the essential qualities that the design should support. Table 1.2 summarizes these qualities.

Designing good abstractions is a separate topic. In this chapter, I won’t digress into the details. Instead, I discuss abstraction design extensively in appendix A. There I discuss each of the qualities described in table 1.2 in much more detail and with lots of real-world examples. Go through the appendix before you dive into the next chapters. When you’re comfortable distinguishing well-designed abstractions from the poorly designed ones, you’ll better appreciate how they contribute to more effective DSL design techniques.
You’ve reached the end of a long introduction to the rationale behind DSLs. When you model a specific domain, your implementation needs to speak the vocabulary of the domain. When you have the common vocabulary in place, the DSL brings the domain syntax and semantics into your solution model.

Be sure your DSL is expressive enough by using well-designed abstractions that use the power of the host language. Designing abstractions is an iterative process, and so is designing a good DSL. You can’t achieve a well-designed DSL in the first iteration. It always evolves through a collaborative effort between the developer and the domain expert. Involve the team of domain experts early in the development process. If they can understand what your abstraction promises and verify the implementation of their business rules, that’s proof that your model is both correct and sufficiently expressive.

Laying the groundwork for an unfamiliar paradigm of development is always an arduous process. Kudos to you for successfully undertaking that task. Now you’ll start the journey into the real-world pragmatics of DSL design and implementation. In chapter 2, the focus is more on actual DSLs that have been implemented using modern languages on the JVM. The adventure starts with Java, then continues into the expressiveness of Groovy, Scala, and Ruby. You’ll notice how the expressiveness of our

### Key takeaways & best practices

- **A DSL is a communication medium between developers and business practitioners.** Always involve your domain expert while you’re designing a DSL.
- **A DSL might not be suitable for every occasion.** Weigh the pros and cons before you decide to design and invest in one.
- **DSL design is always iterative.** Give it the diligence and effort that it deserves.
- **Keep in mind that the syntax of the DSL needs to be expressive enough for the end user.** Don’t overengineer your DSL. Doing that only makes the syntax cluttered and increases the complexity of the implementation.
models increases as you use some of today’s state-of-the-art programming languages. Stay tuned!

1.9 References


Your success—and sanity—are closer at hand when you work at a higher level of abstraction, allowing your attention to be on the business problem rather than the details of the programming platform. Domain Specific Languages—“little languages” implemented on top of conventional programming languages—give you a way to do this because they model the domain of your business problem.

**DSLs in Action** introduces the concepts you’ll need to build high-quality domain-specific languages. It explores DSL implementation based on JVM languages like Java, Scala, Clojure, Ruby, and Groovy and contains fully explained code snippets that implement real-world DSL designs. For experienced developers, the book addresses the intricacies of DSL design without the pain of writing parsers by hand.

**What’s Inside**
- Tested, real-world examples
- How to find the right level of abstraction
- Using language features to build internal DSLs
- Designing parser/combinator-based little languages

This book is written for developers working with JVM languages. Others will find techniques that (generally) work for them too.

**Debasish Ghosh** is a senior member of the ACM and an influential blogger. He works with DSLs based on Java, Ruby, Clojure, and Scala.

For online access to the author and a free ebook for owners of this book, go to manning.com/DSLsinAction

“Covers a lot of ground...not only widely but deeply.”
—From the Foreword by Jonas Bonér, Scalable Solutions

“Thorough, well thought-out, carefully crafted.”
—Guillaume Laforge, SpringSource

“Covers the what, why, when, and how of DSLs.”
—David Dossot, Programmer & Author

“Exhaustive, competent, and compelling.”
—Federico Tomassetti, Politecnico di Torino

“Five languages, no trivial examples, all in one book.”
—John S. Griffin, Overstock.com