SAMPLE CHAPTER

DSLs IN ACTION

Debasish Ghosh

FOREWORD BY Jonas Bonér

MANNING

DSLs in Action

by Debasish Ghosh

Chapter 4

brief contents

4 Internal DSL implementation patterns

This chapter covers

- Embedded DSL patterns with metaprogramming
- Embedded DSL patterns with typed abstractions
- Generative DSL patterns with runtime metaprogramming
- Generative DSL patterns with compile-time metaprogramming

In part 1, you were inducted into the DSL-driven development paradigm. You saw DSLs in all their glory and the issues that you need to address to use DSLs in a real-world application. This chapter marks the beginning of the discussion about the implementation-level issues of DSLs.

Every architect has their own toolbox that contains tools for designing beautiful artifacts. In this chapter, you'll build your own toolbox to hold architectural patterns that you can use to implement DSLs. Figure 4.1 depicts the broad topics that I plan to cover.

As a DSL designer, you need to be aware of the idioms and best practices in DSL implementation. We'll start with how to build a collection of patterns that you can

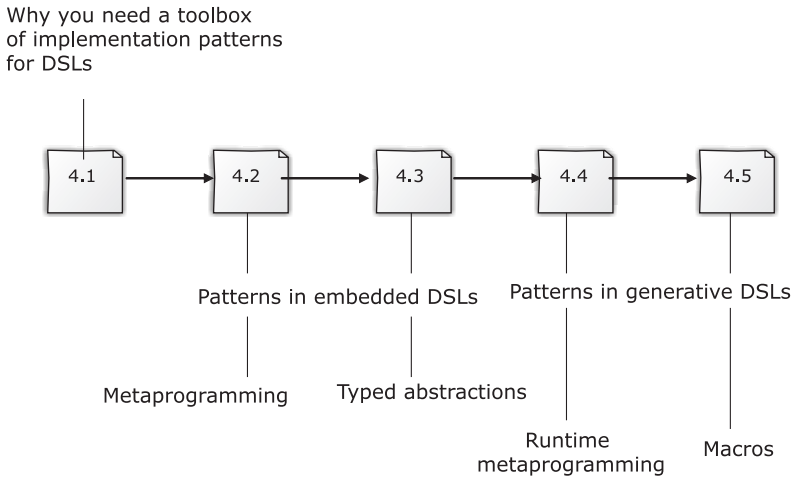


Figure 4.1 Roadmap of the chapter

use in real-life application development. Internal DSLs are most often embedded in a host language. Many of these languages support a meta-object protocol that you can use to implement dynamic behaviors onto your DSL. Most of these languages are dynamically typed, like Ruby and Groovy; we'll discuss several patterns that use the metaprogramming power of these languages in section 4.2. Statically typed languages offer abstraction capabilities to model your DSL with embedded types, which we'll look at in section 4.3 in the context of using Scala as the implementation language. In sections 4.4 and 4.5, we'll look at code-generation capabilities of languages you can use to implement concise internal DSLs. These are called generative DSLs because they have a concise syntax on the surface, but implement domain behaviors by generating code either during compile time or runtime. At the end of the chapter, you'll feel good knowing that now you have a bag full of tricks, patterns, and best practices that you can use to model real-life problem domains.

4.1 Filling your DSL toolbox

A master craftsman always has a packed toolbox. He starts filling up his toolbox with an initial seed that he inherited from his master, then enriches it over the years of practicing his art. We're talking about DSLs in this book, specifically about filling up your own toolbox for implementing internal DSLs.

Let's start with the common patterns of internal DSLs we discussed in section 2.3.1. These are patterns of implementation that you can apply in various scenarios when you're doing DSL-based design. The discussion in section 2.3.1 also contained code snippets that illustrated how the patterns manifest themselves in certain commonly used languages. In this chapter, we'll build on that discussion, provide examples from our problem domain of financial brokerage systems, and map the examples onto

implementations that realize their solution domains. As you read, make sure you collect all the things you need to enrich your toolbox. Sometimes I'll show you multiple implementation techniques for the same pattern, often in different languages, and highlight the trade-offs that each involves.

Before we move on to the details, look at figure 4.2, which picks up a few of the patterns that we identified in chapter 2 and annotates each of them with example artifacts that we'll discuss in this chapter. To clarify one of the annotations in the figure: look at the box "Reflective metaprogramming", which is a pattern for embedded DSL. In the examples that I'll discuss in this chapter, you'll see how to use this pattern to implement *implicit context* in Ruby and Groovy and *dynamic decorators* in Ruby. Both of these implementation artifacts make DSLs that help users express the intent clearly without any unnecessary complexity.

As figure 4.2 indicates, internal DSLs fall into two categories:

- *Embedded*—The DSL is inside the host language, which implies that you, the programmer, write the entire DSL explicitly
- *Generative*—Part of the DSL code (mostly repetitive stuff) is generated by the compile-time or runtime machinery of the language

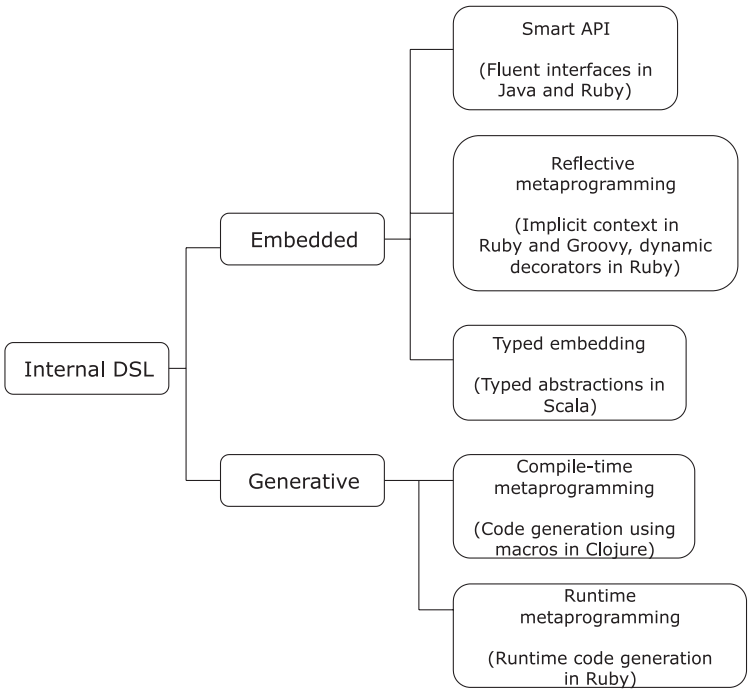


Figure 4.2 Internal DSL implementation patterns, along with example artifacts. I'll discuss each of these artifacts in this chapter, and provide sample implementations in the languages specified in the figure.

In any real-world application, patterns don't occur in isolation. Patterns manifest themselves in the form of cooperating forces and solutions in every use case that you'll be working with. The effects that one pattern generates are handled by another pattern; the entire system evolves as a coherent pattern language. In the following sections, I've chosen to follow the style that you'll encounter in your role as a DSL designer. Instead of providing a geometrically structured description of each of the patterns in isolation, I'll take sample DSL fragments from our financial brokerage systems domain and deconstruct each of them to explore these pattern structures. This method will not only give you an idea of how each of the patterns can be implemented in real-life use cases, but you'll also appreciate how these structures can be synergistically stitched together to create a greater whole.

Let's start with the pattern structures that you'll use to implement embedded DSLs.

4.2 Embedded DSLs: patterns in metaprogramming

Metaprogramming is writing programs that write programs. OK, that's what you find in textbooks, and more often than not you're misled into thinking that metaprogramming is yet another code generator with an embellished name. Practically speaking, metaprogramming is programming with the meta-objects that the compile-time or the runtime infrastructure of your environment offers. I won't go into the details of what that means; I'm sure you know all about it by now, after having gone through the detailed discussion that we had in section 2.5.

In this section, we'll look through examples in our domain that can be modeled using metaprogramming techniques. In some cases, I'll start with implementations in a language that doesn't offer the power of metaprogramming. Then you'll see how the implementation transforms to a more succinct one when we use a language that lets you program at a higher level of abstraction using metaprogramming.

> **CODE ASSISTANCE** In all of the following sections that have rich code snippets, I include a sidebar that contains the prerequisites of the language features that you need to know in order to appreciate the implementation details. These sidebars are just the feelers for the language features used in the code listing that follow. Feel free to read the appropriate language cheat-sheet that's in the appendixes for more information about a particular language.

Every subsection that follows contains a theme that manifests itself as an instance of a metaprogramming pattern implementation, all of which are shown in figure 4.2. We'll start with a sample use case, look at the DSL from the user's point of view, and deconstruct it to unravel the implementation structures. It's not that each use case implements only one pattern instance. In fact, all the following themes comprise multiple pattern instances working toward the fulfillment of realizing the solution domain.

4.2.1 Implicit context and Smart APIs

Let's begin with a brief refresher from the earlier chapters. Clients register themselves with stock trader firms; these firms trade the clients' holdings for them and keep them safe. For more information about client accounts, refer to the callout *Financial brokerage systems: client account* in section 3.2.2.

With that out of the way, let's talk about designing a DSL that registers client accounts with the trader. You can judge for yourself how applying metaprogramming techniques under the hood can make your APIs expressive without users knowing anything about the implementation.

JUDGING THE EXPRESSIVITY OF A DSL

Look at the following DSL script. This DSL creates client accounts to be registered with the stock broker firm.

> **Ruby tidbits you need to know**
>
> - *How classes and objects are defined in Ruby.* Ruby is OO and defines a class in the same way that any other OO language does. Even so, Ruby has its own object model that offers functionalities to the users to change, inspect, and extend objects during runtime through metaprogramming.
> - *How Ruby uses blocks to implement closures.* A closure is a function and the environment where the function will be evaluated. Ruby uses block syntax to implement closures.
> - *Basics of Ruby metaprogramming.* The Ruby object model has lots of artifacts like classes, objects, instance methods, class methods, singleton methods, and so on that allow reflective and generative metaprogramming. You can dig into the Ruby object model at runtime and change behaviors or generate code dynamically.

Listing 4.1 DSL that creates a client account

```
Account.create do

  number      "CL-BXT-23765"
  holders     "John Doe", "Phil McCay"
  address     "San Francisco"
  type        "client"
  email       "client@example.com"

end.save.and_then do |a|

  Registry.register(a)
  Mailer.new
        .to(a.email_address)
        .cc(a.email_address)
        .subject("New Account Creation")
        .body("Client account created for #{a.no}")
        .send
end
```

1 Create account

2 Save account

3 Send mail after open

This code listing shows how the client *uses* the DSL. Note how it hides the implementation details and makes the account creation process expressive enough for the client. It not only creates the account, it does other stuff as well. Can you identify what else it does by looking at listing 4.1, without looking at the underlying implementation? If you can figure out all the actions that the code does you have a smart DSL implementation.

You can easily identify the actions that the DSL performs from the code. It creates the account ❶ and saves it (possibly to the database) ❷. Then it, among other things, registers the account and sends a mail to the account holder ❸.

As far as expressivity is concerned, the DSL does a fairly good job of offering intuitive APIs to the user. Show this DSL snippet to a domain expert and they'll also be able to identify the sequence of actions that it does because the DSL honors the vocabulary of the domain. Now let's look at the underlying implementation.

DEFINING THE IMPLICIT CONTEXT

The `Account` object is an abstraction that implements the methods called while an instance is created. This is the plain old Ruby way to define the instance methods of a class. The script is shown in the following listing.

Listing 4.2 Expressive domain vocabulary in implementation of `Account`

```
class Account
  attr_reader :no, :names, :addr, :type, :email_address

  def number(number)
    @no = number
  end

  def holders(*names)
    @names = names
  end

  def address(addr)
    @addr = addr
  end

  def type(t)
    @type = t
  end

  def email(e)
    @email_address = e
  end

  def to_s()
    "No: " + @no.to_s +
    " / Names: (" + @names.join(',').to_s +
    ") / Address: " + @addr.to_s
  end
end
```

Now let's look beyond the obvious method definitions of the abstraction and gaze into some of the subtle aspects that you can identify as the implementation patterns that we talked about earlier.

With any code snippet, you need to define the context under which that code will run. The context can be an object that you defined earlier or an execution environment that you set up explicitly or declared implicitly. Some languages mandate that the context be explicitly wired with every invocation of the corresponding methods. Consider the following Java code:

```
Account acc = new Account(number);
acc.addHolder(hName1);
acc.addHolder(hName2);
acc.addAddress(addr);
//..
```

All invocations of methods that operate on the `Account` object need to have the context object passed explicitly as the dispatcher. Explicit context specification makes verbose code that doesn't help make DSLs readable and succinct. It's always a plus if a language allows implicit context declaration; it results in terse syntax and a concise surface area for the API. In listing 4.1, we invoke methods like `number`, `holders`, `type`, `email`, and others on an *implicit context* of the `Account` that gets created.

How do you make the context implicit? Here's the relevant code, implemented in Ruby as part of the `Account` class that does exactly that through a clever bit of metaprogramming:

```
class Account
  attr_reader :no, :names, :addr, :type, :email_address

  ## rest as in listing 4.1

  def self.create(&block)
    account = Account.new
    account.instance_eval(&block)
    account
  end
end
```

❶ create takes a block

❷ eval block in Account context

Look at ❷, where `instance_eval` is a Ruby metaprogramming construct that evaluates the block that's passed to it in the context of the `Account` object on which it's invoked. It's as if the newly constructed `account` were implicitly passed to every invocation of the method that you pass around in the Ruby block ❶. This code is an example of *reflective metaprogramming*. The context of evaluation is determined through reflection during runtime by the Ruby execution environment.

You can perform the same trick in Groovy as well, which is another language with strong metaprogramming capabilities.

The previous Ruby code snippet becomes the code shown in the following listing in Groovy.

Groovy tidbits you need to know

- How to *create a closure in Groovy* and set up a context for method dispatch.

Listing 4.3 Implicit context set up for method dispatch in Groovy

```
class Account {

  // method definitions

  static create(closure) {
    def account = new Account()
    account.with closure
    account
  }
}

Account.create {
  number      'CL-BXT-23765'
  holders     'John Doe', 'Phil McCay'
  address     'San Francisco'
  type        'client'
  email       'client@example.com'
}
```

Note how the implementations differ, yet we have APIs of similar expressivity in both the languages.

USING SMART APIS TO IMPROVE EXPRESSIVENESS

Readability is an inevitable consequence of expressiveness in a DSL. Implementing *fluent interfaces* is one way to improve readability and make Smart APIs. You can implement method chaining so that the output from one method flows naturally as the input of another. This technique makes your series of API invocations feel more natural, and it's closer to the sequence of actions that you would perform in the problem domain. This makes the APIs smart in the sense that you don't need to include any boilerplate code in their invocation.

Consider the sequence of method calls that causes mail to be sent in ❸ of listing 4.1. The API invocation flows in the same sequence of actions that you would do when you're working with a mail client.

Watch out for fluency issues when you design DSLs on your own. The next listing shows the snippet of code that implements the `Mailer` class that we used in the DSL in listing 4.1.

Listing 4.4 Mailer class with fluent interfaces

```
class Mailer
  attr_reader :mail_to, :mail_cc, :mail_subject, :mail_body

  def to(*to_recipients)
    @mail_to = to_recipients
    self
  end

  def cc(*cc_recipients)
    @mail_cc = cc_recipients
    self
  end
```

❶ Return self for chaining

```
  def subject(subj)
    @mail_subject = subj
    self
  end

  def body(b)
    @mail_body = b
    self
  end

  def send
    # actual send
    puts "sending mail to (#{@mail_to.join(",")})"
  end
end
```

The `Mailer` instance is returned to the caller ❶ to be used as the context for the next invocation. The `send` method is the final method of the chain that finishes the entire sequence and sends the email.

In listing 4.1, which shows how to use the account-creation DSL, the three steps involved in creating the account are quite explicit. Let's look at an even more explicit view in figure 4.3, which shows the steps that apply the patterns that give the DSL its shape.

The steps for applying a pattern are: create an instance of the account, save it to the database, and do other actions as part of the follow up. The third step in the sequence has been explicitly modeled as a closure, or a Ruby block. Note that the block takes the

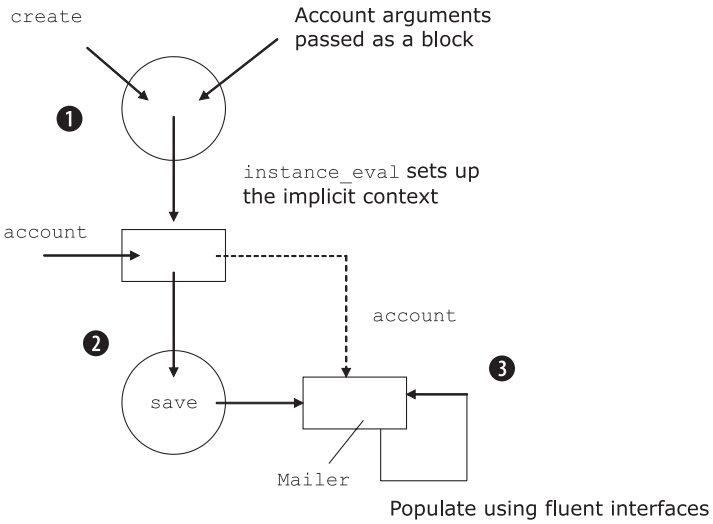


Figure 4.3 The steps of pattern application: ❶ The account is created through the implicit context that is set up using `instance_eval`. ❷ The account is saved. ❸ The Mailer is set up using fluent interfaces and gets the account from a block.

created `account` instance as an input and uses it to perform the other actions. The instance remains unchanged; this is called a side-effecting action.

Managing side effects is an extremely subtle issue in program design. You need to decouple them to make abstractions pure. Abstractions that are free from side effects make your world a better place. Always shoot for explicit separation of side-effecting actions when you're designing DSLs. In listing 4.1, all the side-effecting code is decoupled in a separate block. Doing this is not specific to internal DSL design; you should this in mind when you're designing any abstractions.

In this section, we discussed two of the implementation patterns that are used extensively in DSL-based design paradigms. The key takeaways from this section are listed in the sidebar.

Key takeaways from this section

Implement Smart APIs with fluent interfaces using method chaining (`Mailer` class in listing 4.4).

Implicit context makes DSLs less verbose and incurs less surface area for the APIs, leading to better expressivity (the create class method in the Ruby snippet and the static create method in the Groovy code in listing 4.3).

Isolate side effects from pure abstractions (the Ruby block in listing 4.1 that registers the account and sends mail to the account holder).

Next we'll look at other implementation structures that use reflective metaprogramming to implement dynamic behaviors in your DSL.

4.2.2 Reflective metaprogramming with dynamic decorators

During our explorations of metaprogramming techniques, you saw in section 4.2.1 how you can use them to make your DSL expressive and succinct. In this section, we'll explore yet another aspect of runtime metaprogramming: manipulating class objects dynamically to decorate other objects.

The Decorator design pattern is used to add functionalities to objects dynamically during runtime. (In appendix A, I talk about the role of decorators as a design pattern for enabling composability between abstractions.) In this section, we're going to look at this topic more from an implementation perspective and see how the power of metaprogramming can make more dynamic decorators.

DECORATORS IN JAVA

Let's start with the abstraction for `Trade`, a domain entity that models the most fundamental entity involved in the trading process. For the purpose of this example, we'll consider the contract from the point of view of composing a `Trade` object with decorators that affect its *net value*. For basic information about how a trade's net cash value is computed, take a look at the accompanying sidebar.

Financial brokerage systems: the cash value of a trade

Every trade has a cash value that the counterparty receiving the securities needs to pay to the counterparty delivering the securities. This final value is known as the *net settlement value* (NSV). The NSV has two main components: the gross cash value and the tax and fees. The gross cash value depends on the unit price of the security that was traded, the type of the security, and additional components like the yield price for bonds. The additional tax and fee amounts include the taxes, duties, levies, commissions, and accrued interest involved in the trading process.

The gross cash value calculation depends on the type of the security (equity or fixed income), but is fundamentally a function of the unit price and the quantity traded.

The additional tax and fee amounts vary with the country of trade, the exchange where the trading takes place, and the security that's traded. In Hong Kong, for example, a stamp duty of 0.125% and a transaction levy of 0.007% are payable on equity purchases and sales.

Consider the Java code in the following listing.

Listing 4.5 `Trade` and its decorators in Java

```
public class Trade {
    public float value() {
        // ..
    }
}

public class TaxFeeDecorator extends Trade {
    private Trade trade;

    public TaxFeeDecorator(Trade trade) {
        this.trade = trade;
    }
    @Override
    public float value() {
        return trade.value() + //..;
    }
}

public class CommissionDecorator extends Trade {
    private Trade trade;

    public CommissionDecorator(Trade trade) {
        this.trade = trade;
    }
    @Override
    public float value() {
        return trade.value() + //..;
    }
}
```

1 Trade abstraction

2 Compute and return the trade value

Decorators 3

4 Details of tax and fee computation

5 Details of commission computation

The code in this listing implements the contract for the `Trade` abstraction ❶ and two of the decorators ❸ that can be composed with `Trade` to impact the net value of the execution ❷, ❹, ❺. The usage of this decorator pattern is as follows:

```
Trade t =
    new CommissionDecorator(
        new TaxFeeDecorator(new Trade()));
System.out.println(t.value());
```

You could go on adding additional decorators from the outside over the basic abstraction of `Trade`. The final value that's computed will be the net effect of applying all the decorators on the `Trade` object.

If Java is your language of implementation, listing 4.5 is your DSL for computing the net value of a given trade. Now it's obvious that the code is almost the best that we can do with Java as the implementation language. It makes perfect sense to a programmer, and if they are familiar with the design patterns that the Gang of Four has taught us (see [1] in section 4.7), they must be satisfied with a domain-specific realization of the sermon. But, can we do better?

IMPROVING THE JAVA IMPLEMENTATION

With the reflective metaprogramming capabilities of Ruby or Groovy, we can make the DSL more expressive and dynamic. But before we look into the corresponding implementation, let's identify some of the areas that are potential candidates for improvement. See table 4.1.

Dynamically typed languages like Ruby and Groovy are more concise in syntax than Java. Both offer *duck typing*, which can make more reusable abstractions at the expense of static type safety that languages like Java and Scala give you out of the box. (You can also implement duck typing in Scala. We'll talk about that in chapter 6.) Let's explore more of the dynamic nature of Ruby to get the knowledge you need to improve on the points mentioned in table 4.1.

Table 4.1 Possible improvement areas for decorators in the Java DSL

Can we improve?	How?
Expressivity and domain friendliness.	Be less verbose, but the sky's the limit here.
Hardwired relationship between `Trade` and the decorators.	Get rid of static inheritance relationships, which will make the decorators more reusable.
Readability. The Java implementation reads outside in, from the decorators to the core `Trade` abstraction, which isn't intuitive.	Put `Trade` first, and then the decorators.

DYNAMIC DECORATORS IN RUBY

The following listing is a similar `Trade` abstraction in Ruby, slightly more fleshed out than the Java implementation in terms of real-life contents.

Ruby tidbits you need to know

- *How modules in Ruby can help implement mixins* that you can tag on to other classes or modules.
- *Basics of Ruby metaprogramming* that generate runtime code through reflection.

Listing 4.6 `Trade` abstraction in Ruby

```
class Trade
  attr_accessor :ref_no, :account, :instrument, :principal

  def initialize(ref, acc, ins, prin)
    @ref_no = ref
    @account = acc
    @instrument = ins
    @principal = prin
  end

  def with(*args)
    args.inject(self) { |memo, val| memo.extend val }
  end

  def value
    @principal
  end
end
```

❶ Dynamic module extension

Apart from the `with` method ❶, there's not much to talk about in terms of implementation differences with this code's Java counterpart. I'll get back to the `with` method shortly. First, let's look at the decorators. I've designed the decorators as Ruby modules that can be used as *mixins* in our implementation (for more information about mixins, see appendix A, section A.3):

```
module TaxFee
  def value
    super + principal * 0.2
  end
end

module Commission
  def value
    super - principal * 0.1
  end
end
```

Even without examining the details too closely, you can see that the decorators in the snippet aren't statically coupled to the base `Trade` class. Wow, we've already successfully improved one of the issues that were listed in table 4.1.

Remember we mentioned duck typing? You can mix in the modules of the above snippet with *any* Ruby class that implements a `value` method. And it so happens that our `Trade` class also has a *value* method. But how exactly does the `super` call work in

```
tr = Trade.new('r-123', 'a-123', 'i-123', 200).with TaxFee, Commission
tr.value
```

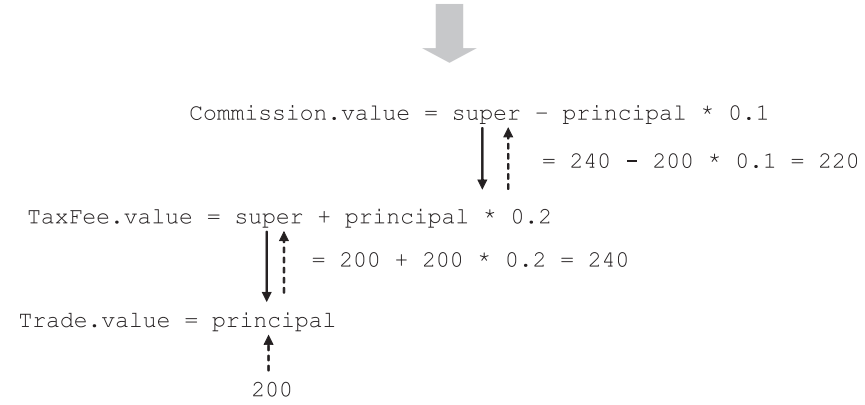


Figure 4.4 How the `super` call wires up the `value()` method. The call starts with `Commission.value()`, `Commission` being the last module in the chain, and propagates downward until it reaches the `Trade` class. Follow the solid arrow for the chain. Evaluation follows the dotted arrows, which ultimately results in 220, the final value.

these module definitions? In Ruby, if you specify a `super` call without any arguments, Ruby sends a message to the parent of the current object to invoke a method of the same name. This is where reflective metaprogramming comes in as the sweet spot of implementation. In this case, it happens to be the `value` method. Notice how we're invoking a `super` class method without statically wiring up any specific super class. Figure 4.4 illustrates how the `super` calls of the `Trade` class and the decorators chain together at runtime to give us the desired effect.

But where is the metaprogramming magic going on? And how does it make our DSL more expressive? For the answers, let's go back to the `with` method in listing 4.6. What it does is take all the decorators that are passed as arguments to the method and creates an abstraction, *dynamically* extending the `Trade` object with all of them. Look at figure 4.5, which shows how the decorators get wired dynamically with the subject class.

The dynamic extension in the figure is equivalent to the effect that we could have had with static extension of Ruby classes. But the fact that we can make it happen during runtime makes things much more reusable and decoupled. Here's how you apply decorators to the `Trade` object using the implementation in listing 4.6.

```
tr = Trade.new('r-123', 'a-123', 'i-123', 20000).with TaxFee, Commission
puts tr.value
```

Using metaprogramming techniques, we wired objects during runtime and implemented the DSL in the previous snippet. Now it reads well from the inside out, as you would expect with the natural domain syntax. The syntax has less accidental complexity than the Java version and is certainly more expressive to the domain person. (See appendix A, section A.3.2 for a discussion of accidental complexity.) There you have

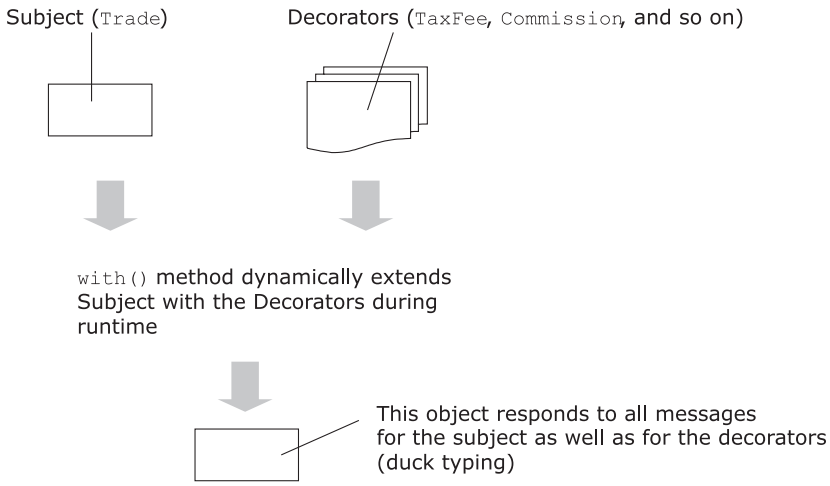


Figure 4.5 The subject (`Trade` class) gets all the decorators (`TaxFee` and `Commission`) and extends them dynamically using the `with()` method.

Key takeaway from this section

The Decorator design pattern helps you attach additional responsibilities to objects. If you can make decorators dynamic like we did with Ruby modules in this section, you get the added advantage of better readability for your DSL.

it. We've successfully addressed all three items for improvement from table 4.1. Wasn't that easy?

It's not all coming up roses though. Any pattern based on dynamic metaprogramming has pitfalls that you need to be aware of. Read the accompanying sidebar for that mild note of exception that might come back to bite you in the end.

When you use runtime metaprogramming in dynamically typed languages, the conciseness of your syntax is improved, the expressivity of the domain language is greatly enhanced, and you can dynamically manipulate the abilities of class structures. All these pluses come at the price of diminished type safety and slower execution speed. I'm not trying to discourage you from using these techniques in designing DSLs for your next project. But, as always, designing abstractions is an exercise in managing trade-offs. In the course of using DSL-based development, you'll encounter situations where static type safety might be more important than offering the best possible expressiveness to your DSL users. And as you'll see in course of this chapter, even with static type checking there are options with languages like Scala for making your DSL as expressive as its Ruby counterpart. Weigh all the options, then decide whether to take the plunge.

In this section, you saw how you can use metaprogramming to help you implement dynamic decorators. It's so different and much more flexible than what you would do in a statically typed language like Java or C#. Most importantly, you saw how you can use dynamic decorators to implement a real-world DSL snippet for our domain. In the next section, we'll continue our journey through the world of reflective metaprogramming techniques and implement yet another variant of one of the most popular design patterns in Java.

4.2.3 Reflective metaprogramming with builders

Remember our order-processing DSL that we built as part of a motivating example in chapter 2? We started with an example in Java where we used the Builder design pattern (see [1] in section 4.7) to make the order-processing DSL expressive to the user. As a quick recap, here's how the client uses the Java implementation of the DSL, replicated from section 2.1.2:

```
Order o =
  new Order.Builder()
    .buy(100, "IBM")
    .atLimitPrice(300)
    .allOrNone()
    .valueAs(new OrderValuerImpl())
    .build();
```

This code uses the fluent interfaces idiom to build a complete `Order`. But with Java, the entire building process is static; all the methods that the builder supports need to be statically invoked. Using patterns of dynamic metaprogramming in Groovy, we can make builders much more minimalistic, but still expressive (see [2] in section 4.7; I also discuss the minimalism property of abstraction design in appendix A, section A.2). The user has to write less boilerplate code, and that makes the final DSL more precise and easier to manage. The language runtime uses reflection (that's why it's called reflective metaprogramming) to do what would otherwise have to be done statically, using lots of boilerplate code.

Groovy tidbits you need to know

- *How classes and objects* are defined in Groovy.
- *Groovy builders* let you create hierarchical structures using reflection. The syntax that it offers is concise and ideal for a DSL.

THE MAGIC OF GROOVY BUILDERS

Consider the skeleton components in listing 4.7 for modeling a `Trade` object in Groovy. Once again, as a side note, the `Trade` abstraction we're developing here is much simpler than what you would use to develop a production-quality trading system and is only for the purpose of demonstrating the concept that we're addressing in this section.

Listing 4.7 `Trade` abstraction in Groovy

```
package domain.trade

class Trade {
  String refNo
  Account account
  Instrument instrument
  List<Taxfee> taxfees = []
}

class Account {
  String no
  String name
  String type
}

class Instrument {
  String isin
  String type
  String name
}

class Taxfee {
  String taxId
  BigDecimal value
}
```

Here we have a plain old abstraction for `Trade` that contains an `Account`, an `Instrument`, and a list of `Taxfee` objects. First let me introduce the `builder` script that will magically introspect into the guts of the classes and create the correct objects with the values that you supply for them.

Listing 4.8 Dynamic builders for `Trade` objects in Groovy

```
def builder =
  new ObjectGraphBuilder()

builder.classNameResolver = "domain.trade"
builder.classLoader = getClass().classLoader

def trd = builder.trade( refNo: 'TRD-123') {
  account(no: 'ACC-123', name: 'Joe Doe', type: 'TRADING')
  instrument(isin: 'INS-123', type: 'EQUITY', name: 'IBM Stock')
  3.times {
      taxfee(taxId: 'Tax ${it}', value: BigDecimal.valueOf(100))
  }
}

assert trd != null
assert trd.account.name == 'Joe Doe'
assert trd.instrument.isin == 'INS-123'
assert trd.taxfees.size == 3
```

❶ **Building the builder**

❷ **Dynamically created methods**

If you're from a Java background, the code in this listing looks magical indeed. The DSL user writes methods, like `trd` ❶, that construct the builder that creates trade objects. Within the `trd` method, the user calls methods like `account` and `instrument`

❷, which we don't have as part of the `Trade` class. Yet somehow the code runs, as if these methods were created magically by the language runtime. It's the power of Groovy metaprogramming that does the trick.

INSIDE GROOVY BUILDERS

In fact, it's the combination of metaprogramming techniques, named parameters, and closures that make the DSL snippet in listing 4.8 work so wonderfully. You've seen quite a few examples in earlier chapters of how closures work in Groovy. Let's dig into some of the details of how the runtime discovers the class names, builds the correct instances, and populates with the data that you've supplied to the builder. The salient points are listed in table 4.2.

Table 4.2 Builders and metaprogramming

Runtime discovery	How it works
Matching the method name	For any method invoked on the `ObjectGraphBuilder`, Groovy matches the method name with a `Class` using a `ClassNameResolver` strategy that gives it the `Class` to instantiate.
Customizing `ClassNameResolver`	You can customize the `ClassNameResolver` strategy with your own implementation.
Creating the instance	When Groovy has the `Class`, it uses another strategy, `NewInstanceResolver`, that calls a no-argument constructor to create a default instance of the class.
Working with hierarchical structures	The builder gets more sophisticated when you have references within your class that set up a parent/child relationship (like we have in `Trade` and `Account` in listing 4.8). In these cases, it uses other strategies like `RelationNameResolver` and `ChildPropertySetter` to locate the property classes and create instances.

For more details about how Groovy builders work, refer to [2] in section 4.7.

You've seen enough of metaprogramming techniques and how you can use them to design expressive DSLs. It's now time to take stock of what we did in this whole section and review the patterns from figure 4.2 that we've implemented so far. After all, you want to use them as the tools in your repertoire to build a world of DSLs.

Key takeaway from this section

You can use builders to construct an object incrementally within your *DSL*. When you make the builders dynamic, you cut down on the boilerplate code that you have to write. Dynamic builders like the ones in Groovy or Ruby smooth out the implementation aspect of your DSL by constructing methods dynamically through the meta-object protocol of the language runtime.

4.2.4 Lessons learned: metaprogramming patterns

Don't think about the patterns we've covered as isolated entities, shut away by themselves. When you work on a domain, you'll find that each results in *forces* that you need to *resolve* by applying other patterns (see [5] in section 4.7). Figure 4.6 is a reminder of what we've implemented so far. In the figure, the DSL patterns from figure 4.2 are listed on the left and the implementation instances of each of them that have been used in the code examples in this chapter are listed on the right.

We discussed a few important patterns that you would frequently use in implementing internal DSLs. These are primarily targeted for dynamically typed languages that offer strong metaprogramming capabilities.

From reflective metaprogramming we're going to move on to another category of patterns that you'll use when you're implementing internal DSLs in statically typed

Key takeaway from this section

The patterns we discussed in this section help you make your DSLs less verbose and more dynamic. We used the metaprogramming capabilities of the language to do stuff during runtime that would you would otherwise have to do statically, using boilerplate code.

It's not only the specific implementation in Ruby or Groovy that matters. You need to think through the context that leads to these implementations. Indeed, there are lots of other ways to make your DSL dynamic when you're using powerful implementation languages.

When you get a feel for the problem that this technique solves in real-life domain modeling, you'll be able to identify many such instances and come up with solutions of your own.

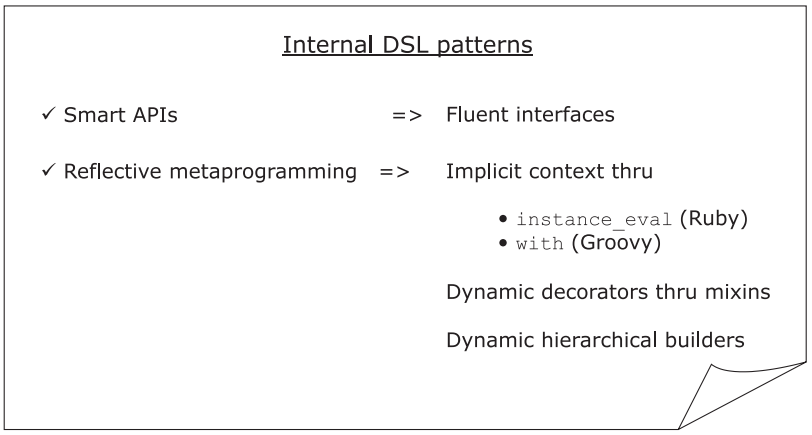


Figure 4.6 Internal DSL patterns checklist up to this point. In this chapter, you've seen implementations of these patterns in Ruby and Groovy.

languages like Scala. When you model your DSL components as typed abstractions, some of your business rules are implemented for free as part of the language's type system. This is yet another way to make your DSL concise, yet expressive.

4.3 Embedded DSLs: patterns with typed abstractions

In the patterns we've discussed so far, you've learned how to get concise code structures in your DSL, not only in how they're used, but also as part of the implementation.

In this section, we move away from dynamic languages and try to find out if we can make DSLs expressive using the power of a type system. For these examples, we're going to be using Scala. (All Scala code that I demonstrate in this section is based on Scala 2.8.) We'll focus on how types offer an additional level of consistency to our DSL, even before the program can run. At the same time, types can make a DSL as concise as some of the dynamic languages that we've already discussed. Refer to figure 4.2 frequently, which is our frame of reference for all the patterns that we'll be discussing in this chapter.

4.3.1 Higher-order functions as generic abstractions

So far in our discussions about domains, we've concentrated on the operations of the financial brokerage system, like maintaining client accounts, processing trades and executions, and placing orders on behalf of its clients. In this section, let's look at a client document that determines all trading activities that the broker does during the work day. A daily transaction activity report for a client account is something that the trading organization generates for some of its clients and dispatches to their respective mailing addresses.

GENERATING A GROUPED REPORT

Figure 4.7 shows a sample client activity report that contains the instruments traded, the quantity, the time of the trade, and the amount of money involved in the trade.

A/C Name: _____ Address: _____

Trading Activity for 12/12/2009

Instrument	Quantity	Time	Amount
Google	2000	08:20	
IBM	1200	11:30	
Google	350	11:45	.....
Verizon	350	12:10	
IBM	2100	12:20	
Google	1200	12:50	
....	....	....	

Figure 4.7
A simplified view of a sample client activity report statement

A/C Name: _____ Address: _____

Trading Activity for 12/12/2009

Instrument	Quantity	Time	Amount
Google	2000 1200 350	08:20 11:45 12:50	
IBM	1200 2100	11:30 12:20	.....
Verizon	350	12:10	
....	....	....	

Figure 4.8
Sample view of the account activity report sorted and grouped by the instruments traded during the day. Note how the instruments are sorted and the quantities grouped together under each instrument.

Many organizations offer their clients a flexible way to view daily transactions. They can view it sorted on specific elements, or grouped. If I have a trading account, I might want to see all my transactions sorted and grouped by the individual instruments traded during the day. Figure 4.8 shows that view.

Maybe I also want a view that groups all my transactions based on the quantities traded in each of them, as in figure 4.9.

In reality, the report can contain lots of other information. We're going to focus only on what is useful for the context we're going to implement. In this example, we'll construct a DSL that lets a client view his trading activity report based on a custom grouping function. We'll start with a DSL that implements separate functions for each

A/C Name: _____ Address: _____

Trading Activity for 12/12/2009

Quantity	Instrument	Time	Amount
350	Google Verizon	12:50 12:10	
1200	Google IBM	11:45 11:30	.....
2000	Google	08:20	
2100	IBM	12:20	
....	....	.....	

Figure 4.9
Sample view of the account activity report sorted and grouped by the quantity of instruments traded during the day.

grouping operation. Then, we'll improve the verbosity of our implementation by designing a generic `groupBy` combinator that accepts the grouping criterion as a higher-order function.

> **DEFINITION** As a reminder, a combinator is a higher-order function that takes another function as input. Combinators can be combined to implement DSL structures as we'll see here and also in chapter 6. Appendix A also contains details about combinators.

You'll see the power of Scala's type system that makes the operation statically type safe, combined with its ability to handle higher-order functions. To that end, let's jump right in with code examples.

> **Scala tidbits you need to know**
>
> - *Case classes define immutable value objects.* A case class is a concise way to design an abstraction where the compiler gives you a lot of goodies out of the box.
> - *Implicit type conversions* allow you to extend an existing abstraction in a completely noninvasive way.
> - *For-comprehensions* offer a functional abstraction of an iterator over a collection.
> - *Higher-order functions* let you design and compose powerful functional abstractions.

SETTING UP THE BASE ABSTRACTIONS

Let's start bottom up with the final view of how your DSL will look. Then we'll work toward implementing the same thing using Scala. Here's how users will use your DSL:

```
activityReport groupBy(_.instrument)
activityReport groupBy(_.quantity)
```

In the code snippet, the first invocation generates the activity report grouped by instrument, and the second one generates it grouped by the quantity traded. The following snippet implements the basic abstraction of a client activity report. Let's take a more detailed look at some of the features that the abstraction offers.

```
type Instrument = String

case class TradedQuantity(instrument: Instrument,
  quantity: Int)

implicit def tuple2ToLineItem(t: (Instrument, Int)) =
  TradedQuantity(t._1, t._2)

case class ActivityReport(account: String,
  quantities: List[TradedQuantity]) {
  //..
}
```

1. Concrete type definition
2. Value object
3. Implicit Tuple2 to LineItem
4. The main abstraction

This isn't a Scala book. Even so, for your understanding, I'm going to highlight some of the features that this code fragment offers. It'll help you compare it with an equivalent Java fragment, and demonstrate some of the expressive power that it has.

Keeping in mind that we need to be expressive in our DSL, we start with a type definition that models a domain artifact ❶. This keeps the code self-documented by avoiding opaque native data types. It also makes the code more meaningful to a domain user and keeps the type of `Instrument` flexible enough for future changes.

The case class `TradedQuantity` ❷ models a value object. Value objects are supposed to be immutable and Scala *case classes* are a succinct way to represent them. A *case class* offers automatically immutable data members, syntactic convenience for a built-in constructor, and out-of-the-box implementations of `equals`, `hashCode`, and `toString` methods. (Case classes are a great way to model value objects in Scala. For more details, see [4] in section 4.7.)

The implicit declaration ❸ is the Scala way to provide automatic conversion between data types. Implicits in Scala are *lexically scoped*, which means that this type conversion will be functional only in the module where you explicitly import the implicit definition. In this example, the tuple `(Instrument, Int)` can be implicitly converted to a `TradedQuantity` object through this declaration. Note that `(Instrument, Int)` is the Scala literal representation for a tuple of two elements. A more verbose representation is `Tuple2[Instrument, Int]`. (In section 3.2.2, I discussed how Scala `implicits` work. If you need to, make a quick visit back to refresh your memory.)

Finally, we come to the main abstraction of the client activity report. `ActivityReport` ❹ contains the account information and the list of all tuples that represent the quantities and instruments traded during the day.

Now we're going to step through the following iterative modeling process and implement the grouping function that'll give the client the custom views they like to have for their daily transaction report. Table 4.3 shows how we'll improve our model using an iterative process.

Table 4.3 Iterative improvement of the DSL

Step	Description
Create a DSL for the client to view the trading activity report. Support grouping operations by `Instrument` and `Quantity`.	Implement specialized grouping functions: `groupByInstrument` and `groupByQuantity` Reduce repetitive boilerplates by implementing a generic grouping function: `groupBy{T <% Ordered[T]]`

First we'll implement the `groupBy` functions.

First attempt: a specialized implementation

If we build specialized implementations of the `groupBy` function for the `ActivityReport` abstraction, the DSL user is going to get quite an expressive API. But we're discussing *implementations* here; expressive *usage* can't be the only yardstick for judging the

completeness of our DSL. The following listing shows the specialized implementation for grouping by `Instrument` and `Quantity`. Notice how we need to define specialized functions for each kind of grouping that we want to give to the user.

Listing 4.9 Activity report with specialized implementations of `groupBy`

```
type Instrument = String

case class TradedQuantity(instrument: Instrument,
  quantity: Int)

implicit def tuple2ToLineItem(t: (Instrument, Int)) =
  TradedQuantity(t._1, t._2)

case class ActivityReport(account: String,
  quantities: List[TradedQuantity]) {
  import scala.collection.mutable._

  def groupByInstrument = {
    val m =
      new HashMap[Instrument, Set[TradedQuantity]]
        with MultiMap[Instrument, TradedQuantity]

    for(q <- quantities)
      m addBinding (q.instrument, q)

    m.keys.toList
          .sortWith(_ < _)
          .map(m.andThen(_.toList))
  }

  def groupByQuantity = {
    val m =
      new HashMap[Int, Set[TradedQuantity]]
        with MultiMap[Int, TradedQuantity]

    for(q <- quantities)
      m addBinding (q.quantity, q)

    m.keys.toList
          .sortWith(_ < _)
          .map(m.andThen(_.toList))
    }
  }
```

❶ MultiMap with mixin

❷ For comprehension

❸ Grouping by Instrument

Can you identify the drawbacks of this implementation? Before we look into them, let me briefly explain some of the Scala idioms that we're using in this listing.

In the implementation of `ActivityReport` in listing 4.9, `quantities` can contain multiple entries for the same `Instrument`, so we define a multimap in ❶ using Scala's mixin syntax. With `HashMap`, we mix in the *trait* `MultiMap` to get a concrete instance of the `MultiMap`. For more details about Scala traits and mixins, refer to [4] in section 4.7.

We iterate over `quantities` and populate the `HashMap` using *for comprehensions* of Scala ❷. This is quite different from a `for` loop that we have in imperative languages. (I'll discuss `for` comprehensions in detail when we talk about monadic structures in

Scala in section 6.9.) In ❸ we sort the keys of the `MultiMap` and forms a `List` grouped by `Instrument`. Each member of the `List` is a `Set` containing the quantities that correspond to a single `Instrument`. The underscores have their usual meaning that we discussed in section 3.2.2.

The main drawback of the implementation in listing 4.9 is the number of boilerplate repetitions that the code contains. `groupByInstrument` and `groupByQuantity` have the same overall structure; only the attribute based on which the grouping is to be done is different. Do you hear a familiar voice from the past, crying out against a violation of well-designed abstraction principles? If not, turn to appendix A, where I discuss how the process of *distillation* keeps your abstractions free from all accidental complexities. The problem with this code is that the specialized `groupBy` implementation encourages boilerplate code. Not only that, if we add more grouping criteria later to the `ActivityReport` class, we'll need to write more boilerplate code to implement customized grouping. What can we do about that? What we need is a more generic implementation.

THE GENERIC IMPLEMENTATION

Let's make the implementation more generic and subsume the specialized methods.

Listing 4.10 Generic implementation of `groupBy`

```
type Instrument = String

case class TradedQuantity(instrument: Instrument,
  quantity: Int)

implicit def tuple2ToLineItem(t: (Instrument, Int)) =
  TradedQuantity(t._1, t._2)

case class ActivityReport(account: String,
  quantities: List[TradedQuantity]) {
  import scala.collection.mutable._

  def groupBy[T <% Ordered[T]](f: TradedQuantity => T) = {
    val m =
      new HashMap[T, Set[TradedQuantity]]
        with MultiMap[T, TradedQuantity]
    for(q <- quantities)
      m addBinding (f(q), q)
    m.keys.toList.sort(_ < _).map(m.andThen(_.toList))
  }
}
```

❶ **Parameterized by what to group**

Now the implementation is reduced in size. Did you see how the density increased when we implemented the generic `groupBy` ❶ to create more powerful abstractions? Table 4.4 provides a summary of how you implement the generic `groupBy`.

Now let's look at the invocation of `groupBy` by the DSL user and follow the sequence of implementation steps that's executed. This exercise will help you understand how the type system of Scala works behind the scenes to create an expressive DSL structure.

Table 4.4 Implementing a generic `groupBy`

Step	Description
Implement a generic `groupBy`	Needs to be parameterized with the type that we'll use for grouping the activity report. It will accept a function `f` as its argument, which models the criteria of grouping. `groupBy` is an example of Scala's support for *higher-order functions*. You can pass around functions like any other data type as parameters and return types. You can use this ability to abstract the criteria and replace specialized implementations of grouping as in listing 4.9. Look at figure 4.10 to understand how such a generic function works under the hoods

Examining what goes on behind the scenes is an important step in DSL design and if you're an implementer, you should be extremely sure to understand every bit of it. Read this section several times if you need to, until you have a clear understanding of how the method dispatch works in Scala. Quite a few idioms are hidden in the 15 lines of implementation shown in listing 4.10 that need careful consideration. When you're confident you see how the various idioms are wired to fulfill the contract that the API publishes, you'll be able to make it a part of your toolbox.

```
val activityReport =
  ActivityReport("john doe",
    List(("IBM", 1200), ("GOOGLE ", 2000), ("GOOGLE", 350),
      ("VERIZON", 350), ("IBM", 2100), ("GOOGLE", 1200)))

println(activityReport groupBy(_.instrument))
println(activityReport groupBy(_.quantity))
```

Instead of trying to explain that code textually, let me explain the sequence of actions that take place for the invocation `activityReport groupBy(_.instrument)` in figure 4.10.

Higher-order functions aren't specific only to typed abstractions. All modern languages offer higher-order functions and closures, irrespective of whether or not they're statically typed. You can use the pattern implementations I discuss here in different ways and languages. Keep an eye on the context in which the patterns are being used, and use your implementation language to get things done.

Remember, we're trying to explore all internal DSL implementation patterns across the languages that you use on the JVM. You can use a statically typed language or a dynamically typed one; either way, your goal should be to use the appropriate tool for the power you need in modeling your DSL.

In the next section, we're going to discuss how to use explicitly typed constraints to express domain logic and behavior. This is something that dynamically typed languages can't support. But given an expressive and rich type system, using explicitly typed constraints can be a potent tool in your toolbox. They can make your DSLs unbelievably succinct.

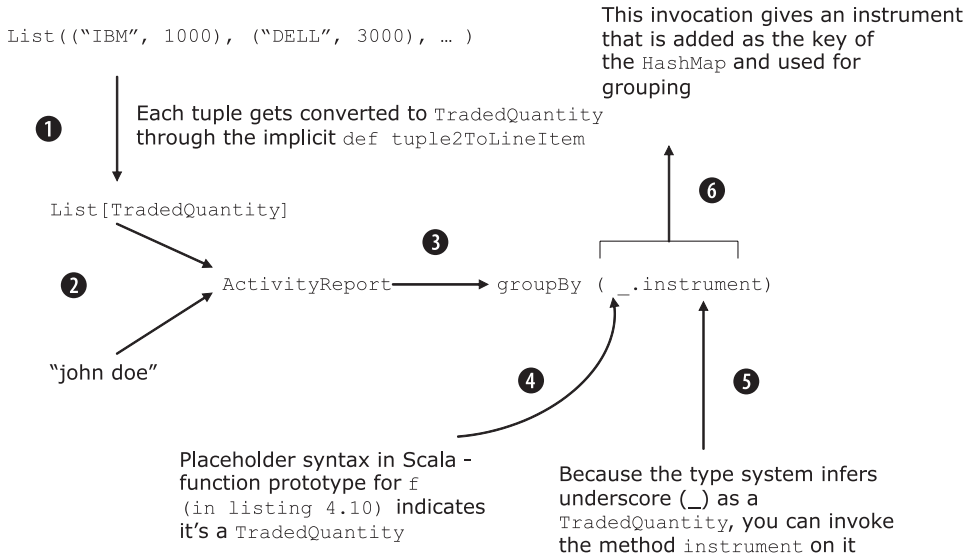


Figure 4.10 Activity report computation grouped by instrument (`groupBy(_.instrument)`). Follow the steps in the figure and correlate them with listing 4.10 and the snippet that follows it, which uses the DSL to compute the `ActivityReport` for "john doe".

4.3.2 Using explicit type constraints to model domain logic

When you design a domain model, you implement abstraction behaviors that must honor the rules and constraints that the domain imposes on you. Languages like Ruby and Groovy are dynamically typed, so all such domain rules need to be encoded as runtime constraints. In section 4.2, you saw how reflective metaprogramming works toward implementing DSL structures that model these domain rules in Ruby and Groovy. In this section, I'll start with an example of runtime validation implementation in Ruby. Then I'll demonstrate how you can implement similar constraints more succinctly using the static type system of Scala.

RUNTIME VALIDATION IN RUBY

Consider the simple example from our domain of a `Trade` abstraction in Ruby, which we partially modeled in listing 4.7. A `Trade` object needs an `Account` object, which we call the *trading account* of the client. In listing 4.7, the account object is well represented with the `attr_accessor` class methods. In the domain of trading systems, there can be multiple types of accounts (discussed in the sidebar titled *Financial brokerage systems: client account* in section 3.2.2). But the account that we specify in a `Trade` abstraction is constrained to be a *trading account* only; it can't be a *settlement-only account*. This domain rule needs to be validated every time we build a `Trade` object with an `Account` object. How do you do this in Ruby? You can insert the usual validation check as in the following snippet:

```
class Trade
  attr_accessor :ref_no, :account, :instrument, :principal

  def initialize(ref, acc, ins, prin)
    @ref_no = ref
    raise ArgumentError.new("Has to be a trading account")
      unless trading?(acc)
    @account = acc
    ## ..
```

Wherever you expect to have a trading account passed in your domain model, you need to do this same validation over and over *during runtime*. (You can make the validation more declarative using class methods, as used in Rails; but still, it remains a runtime validation.) You also need to write explicit unit tests for each of these cases to check whether the particular domain behavior fails as it should when supplied with a nontrading account. All this requires additional code, which you can avoid if your language supports explicit specification of typed constraints.

In statically typed languages, you can specify constraints over specific types that'll be checked during compile time. When you have a program that compiles successfully, at least one level of consistency of domain behaviors is already enforced within your model.

EXPLICITLY TYPED CONSTRAINTS IN SCALA

Let's try to model a `Trade` object in Scala that has some domain constraints over accounts and instruments. At the end of this exercise, you'll realize how explicit type constraints can make your DSL abstractions promise an extra level of consistency that the dynamically typed ones can't, even before you execute it. You'll definitely want this trick in your toolkit when you're using a statically typed language in your application.

> **Scala tidbits you need to know**
>
> - The *power of type-based programming*. You can use types to express many constraints the domain in your DSL. Generic type parameters and abstract types are your friends.
> - *Abstract vals* let you keep the abstraction open until the last stage of instantiation.

Every trade object needs a `Trading` account. Here's how we model this behavior in Scala. The following listing shows only this one aspect of a `Trade` object and is meant only as an example.

Listing 4.11 `Trade` object with typed constraints in Scala

```
trait Account
trait Trading extends Account
trait Settlement extends Account
```

(1) Two Account types

```
trait Trade {
  type A <: Trading

  val account: A
  def valueOf: Unit
}
```

❷ Account subtype of Trading

❸ Account instance

This listing is an example of how types can enforce implicit business rules. In the listing, we've modeled `Account` and `Trade` objects using Scala traits (see [4] in section 4.6). We've used separate types for `Trading` and `Settlement` accounts ❶. As a programmer, you can't pass a `Settlement` account to a method that expects a `Trading` account. The compiler enforces this rule; a business rule that expects a `Trading` account doesn't have to explicitly check to determine whether you passed a valid account type.

We've also defined some of the business rules explicitly. We've abstracted type `A` with constraints (`<: Trading`) in `Trade` ❷. You can't instantiate a `Trade` object with any account type other than `Trading` ❸. You don't have to write any extra code to enforce the validation; again, the compiler does it for you.

A trade is a contract between two parties that involves an exchange of instruments. If you need a little brush up on some of the attributes of a trade, see the sidebar that's in section 1.4. Depending on the instruments that are traded, the behavior, lifecycle, and calculations of the trade vary. An *equity trade* is one that involves an exchange of equities with currencies. When a fixed income is the instrument type that's being exchanged in a trade, we call it a *fixed income trade*. For more details about equities, fixed incomes, and other instrument types, see the sidebar in this section.

Financial brokerage systems: instrument types

Instruments that are traded can be of various types designed to meet the needs of the investors and issuers. Depending on the type, every instrument follows a different lifecycle in the trading and settlement process.

The two main classifications are equity and fixed income.

Equities can again be classified as common stock, preferred stock, cumulative stock, equity warrants, or depository receipts. The types of fixed income securities (also known as bonds) include straight bonds, zero coupon bonds, and floating rate notes. For the purpose of our discussion, it's not essential to be familiar with all these details. What is important is that the `Trade` abstractions will vary, depending on the type of instrument that's being traded.

Let's specialize our definition of `Trade` to model an `EquityTrade` and a `FixedIncomeTrade` in the following listing.

Listing 4.12 `EquityTrade` and `FixedIncomeTrade` model

```
trait Instrument
trait Stock extends Instrument
```

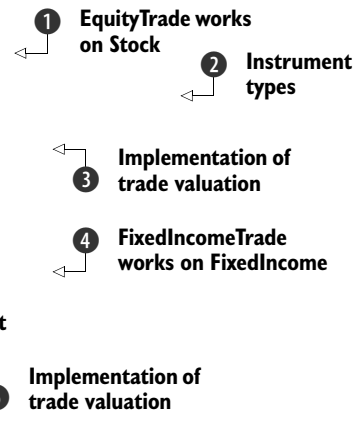
```
trait FixedIncome extends Instrument

trait EquityTrade extends Trade {
  type S <: Stock

  val equity: S
  def valueOf {
    //..
  }
}

trait FixedIncomeTrade extends Trade {
  type FI <: FixedIncome

  val fi: FI
  def valueOf {
    //..
  }
}
```



Like the explicit constraints on `Account` that we used in listing 4.11, in this listing we're constraining the traded instrument type. Again, business rules are enforced implicitly by the compiler.

We've specified separate types for `EquityTrade` and `FixedIncomeTrade` ❶ and ❹. As a programmer, you can't pass a `FixedIncomeTrade` to a method that expects an `EquityTrade`. The compiler enforces this rule; a business rule that expects a particular type of `Trade` doesn't have to explicitly check to determine whether you passed a valid trade type.

An `EquityTrade` trades a `Stock` ❶ and a `FixedIncomeTrade` trades a `FixedIncome` ❹. The basic business rule is completely enforced at the compiler level without a single line of validation from the programmer. Accordingly, you constrain the abstract vals `equity` ❷ and `fi` ❺.

The `valueOf` method is polymorphic and typed. You can provide separate implementations of the `valueOf` method ❸ and ❻, assuming that your `Trade` abstraction gets an appropriate type, either `Account` or `Instrument`.

Using typed abstractions and explicit constraints on the values and types, we implemented quite a bit of domain behavior without a single line of procedural logic. Not only is our main code base smaller, the number of unit tests that you need to write and maintain has also been reduced. When you need to maintain a code base, don't you feel more comfortable when you have a declarative type annotation that expresses some critical semantics of the domain being modeled?

This discussion was on a different path from what we've been doing with dynamic language-based DSL implementations in the earlier sections. Now let's look back at what you've learned about the *statically typed way of thinking* and how it differs from the Ruby or Groovy way.

4.3.3 Lessons learned: thinking in types

In this section, you've learned how types can play a significant role in designing expressive domain abstractions. The main difference from the earlier Groovy and Ruby examples is that with the safety net of static type checking, you already have one level of correctness built into your implementation. A typed code that compiles correctly guarantees that you'll satisfy many of the constraints of the domain. We'll explore this further in chapter 6 when we design more DSLs using Scala. Figure 4.11 is an updated checklist of internal DSL patterns that you learned about in this section.

We've discussed a few important patterns that you'll use frequently when you're implementing internal DSLs with statically typed languages. Although metaprogramming is the secret sauce for dynamic languages, typed abstractions offer concise DSL development mechanisms when you're using statically typed languages.

Key takeaways from this section

The main purpose of this section was to make you *think in types.* For each abstraction that's in your domain model, make it a typed one and organize the related business rules around that type. Many of the business rules will be automatically enforced by the compiler, which means you won't have to write explicit code for them. If your implementation language has a proper type system, your DSL will be as concise as ones written using dynamic languages.

So far you've seen implementation patterns that make concise internal DSLs, either by abstracting domain rules within a powerful type system or through reflection using the metaprogramming power of the host language. In the next section, we'll look at patterns that will make the language runtime write code for you. You're going to make concise DSLs by using generated code.

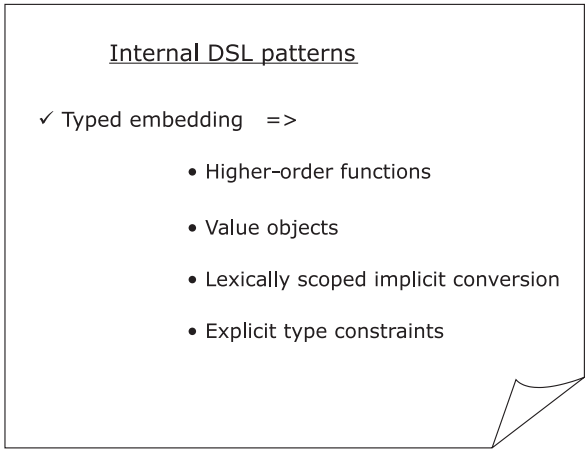


Figure 4.11
Program structures for *typed embedding* of internal DSLs. These patterns teach you how to think with types in a programming language.

4.4 Generative DSLs: boilerplates for runtime generation

Metaprogramming has many facets. In the previous sections, you saw numerous examples of reflective metaprogramming. The VM introspects on meta-objects during runtime, discovers objects that can be applied to the current context, and magically invokes it. But you can look at metaprogramming in a different way. In fact, the classic definition of metaprogramming is *writing code that writes code.*

This definition has specific semantics when we talk about it in the context of different languages. Languages like Lisp offer compile-time metaprogramming, as we saw in detail in section 2.5.2. Languages like Ruby and Groovy offer runtime metaprogramming and can generate code during runtime using `eval` and dynamic dispatch methods. In this section, we'll examine a specific example of how you can reduce the surface area of your DSL abstractions by writing less *explicit* code, instead relying on the language runtime to generate the rest for you. You might be asking yourself, why is this important?

4.4.1 How generative DSLs work

When you design a generative DSL, you'll write less boilerplate code. Instead, the language generates that code for you through metaprogramming. Figure 4.12 offers a visual explanation.

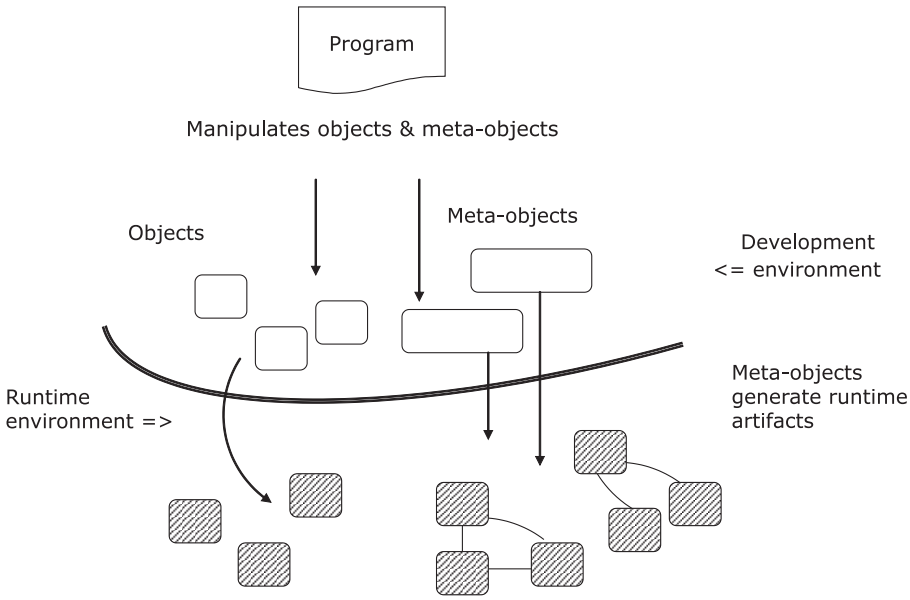


Figure 4.12 Runtime metaprogramming generates code from meta-objects during runtime. The meta-objects generate more objects, which reduces the amount of boilerplate code that you need to write.

Besides the objects that you develop, you also manipulate meta-objects that generate more artifacts for you when the program runs. These additional artifacts represent the code that the language runtime writes for you. It's as if you're giving subtle instructions to your assistant so you can concentrate on the more important aspects of your job. The assistant takes care of all the routine operations that your instructions advised him to perform. But what are these meta-objects? Where do they live and how do they do what you ask them to do? Join me as we explore designing generative DSLs in Ruby.

4.4.2 Ruby metaprogramming for concise DSL design

In our domain of securities trading and settlement, we've been talking about trades in this chapter. In a real-world application, a `Trade` module is a complex one with lots of domain objects and associated business rules, constraints, and validations. Some of these validations are generic and apply to all similar attributes in the same context, but others are specific to the context and need to be explicitly coded within the class definition. Nonetheless, the general flow of validation checks is the same and can be centralized or generated using appropriate techniques.

Let's see how Ruby metaprogramming can make things simpler for you.

USING CLASS METHODS TO ABSTRACT VALIDATION LOGIC

Assume that you're developing your trading application in Rails using `ActiveRecord` for persistence handling. Here's an idiomatic code snippet for the `Trade` model.

```
class Trade < ActiveRecord::Base
  has_one                   :ref_no
  has_one                   :account
  has_one                   :instrument
  has_one                   :currency
  has_many                  :tax_fees

  ## ..

  validates_presence_of     :account, :instrument, :currency
  validates_uniqueness_of   :ref_no

  ## ..
end
```

If you have experience using Rails in a project, you know what the last two lines in the class definition do. The two Ruby *class methods*, `validates_presence_of` and `validates_uniqueness_of`, encapsulate the validation logic for the attributes that we supply as arguments. Note how the domain constraints for these attributes are nicely abstracted away from the surface area of the exposed API (a good example of distilled model design). I discuss the principles of distillation in abstraction design in section A.3. During runtime, these methods generate the appropriate code snippets that validate these attributes.

Ruby tidbits you need to know

- Basics of Ruby *metaprogramming*. The Ruby object model has many artifacts like classes, objects, instance methods, class methods, singleton methods, and so on, that let you use reflective and generative metaprogramming. You can dig into the Ruby object model at runtime and change behaviors or generate code dynamically.
- *Modules* and how they let you implement mixins for extending your existing abstractions.

MIXINS FOR DYNAMIC METHOD GENERATION

Let's do something similar for the `Trade` abstraction that we developed in Ruby in listing 4.6. What we'll do is put inline validation logic into the class definition of `Trade`, but we'll hide the details of invocation and the exception-reporting machinery of the validation behind the scenes; all that boilerplate code is going to be generated during runtime. Here's how we want the abstraction to look:

```
class Trade
  include ...

  attr_accessor :ref_no, :account, :instrument
  trd_validate :principal do |val|
    val > 100
  end

  ## ..
end
```

❶ What to include?

❷ Validation logic as block

In this snippet, something appears to be missing in ❶ (I'll clarify this shortly). Also, `trd_validate` is the validation machinery that can generate runtime code for invoking the validation logic that we pass as the block in ❷.

But, where does `trd_validate` come from? It must be something that we've defined elsewhere and needs to be linked to the code that defines the main class. Possibly the elided portion in ❶ is the place to look at. Let's unravel the code a bit more. Before we go into the details of how this `Trade` model gets the `trd_validate` method, here's a Ruby `module TradeClassMethods` that defines our class method `trd_validate`:

```
module TradeClassMethods
  def trd_validate(attribute, &check)
    define_method "#{attribute}=" do |val|
      raise 'Validation failed' unless check.call(val)
      instance_variable_set("@#{attribute}", val)
    end

    define_method attribute do
      instance_variable_get "@#{attribute}"
    end
  end
end
```

❶ Generate setter for attribute

❷ Generate getter for attribute

What does this snippet do? It generates `setter` ❶ and `getter` ❷ methods during runtime for the attribute that's passed to the method `trd_validate`, using Ruby's dynamic method definition capabilities. In the process of defining them, this code also generates more code for invoking the validation logic that the user passed to it in the form of a block. Nice! Just imagine the amount of boilerplate code that we didn't have to write by using this metaprogramming feature. This savings is multiplied for every attribute that you use to invoke `trd_validate`.

THE FINAL GLUE

Now it's time to assemble things. Let's define another module that glues `TradeClassMethods` with the `Trade` class and makes `trd_validate` available within `Trade`. The following listing is the final version of the code that glues everything together and makes all the above magic happen.

Listing 4.13 `Trade` with domain validation

```
## enable_trade_validation.rb
require 'trade_class_methods'
module EnableTradeValidation
  def self.included(base)
    base.extend TradeClassMethods
  end
end

## trade.rb
require 'trade_class_methods'
require 'enable_trade_validation'

class Trade
  include EnableTradeValidation

  attr_accessor :ref_no, :account, :instrument
  trd_validate :principal do |val|
    val > 100
  end

  ## ..
end
```

You just saw how to avoid writing boilerplate validation logic explicitly by using metaprogramming techniques to generate it. This code is generated *during runtime* when the Ruby VM executes your program.

Key takeaways from this section

Most of the patterns we discussed in this chapter *focus on making your DSL less verbose* and yet more expressive. Ruby and Groovy offer strong support for runtime metaprogramming that generates code for you. Whenever you feel that the code you're writing for implementing a DSL looks repetitive, think metaprogramming. Instead of writing the code yourself, let the language runtime write it for you.

What a long, exciting chapter this is. We've been discussing lots of tricks that you need up your sleeve when you're writing DSLs. Don't worry if everything isn't sinking in the first time you read it. After you get an overall idea of these techniques, you'll be able to think through your problem and carve out your solution domain in the best way possible. If you need a light refreshment to recharge your programming batteries, go get it now. What we'll be discussing next relates to a new development on the JVM of an age-old paradigm of program development. It's Lisp metaprogramming, packaged on the JVM, and it's called Clojure. Ruby and Groovy metaprogramming is primarily based on runtime code generation, but Clojure does it during compile time using macros. We're going look at how macros shape the way you think about designing internal DSLs.

4.5 Generative DSLs: macros for compile-time code generation

Finish up your snack, because now we're going to talk about how generative metaprogramming in Ruby and Groovy generates code during runtime. Your DSL surface syntax will still be concise; you don't have to write the boilerplate stuff that the language runtime generates for you. With Clojure (the Lisp on the JVM), you get all the benefits of code generation, minus the runtime overhead of it that Groovy and Ruby incur. (For information about Clojure, go to http://clojure.org.) Invented by Rich Hickey, Clojure offers the syntax and semantics of Lisp, along with seamless integration with the object system that Java offers. For a more detailed introduction to the language and the runtime, refer to [3] in section 4.7.

4.5.1 Metaprogramming with Clojure

Clojure is a dynamically typed language that implements duck typing and offers powerful functional programming capabilities. In this section, we'll focus more on the generative capabilities of Clojure that are available through its system of macros.

Code generation using Clojure macros is a form of *compile-time metaprogramming*, which we discussed briefly in section 2.3.1. If you're not familiar with the basic concepts of how compile-time macros work in Lisp or Clojure, now's a good time to review the concepts that I discuss in appendix B. As a brief refresher, look at figure 4.13, which outlines the basic flow of events in a compile-time metaprogramming system. The program that you write defines higher-order abstractions as macros that get expanded to valid Clojure forms during the compilation phase.

Before we dive into the implementation details, let's look at a snippet of the problem domain that we're going to cover. When a client places an order to the broker for trading an instrument (either buy or sell), the following sequence of actions take place:

1. The broker places the order in the exchange
2. Street-side trades are done between brokers according to the placed order and results in execution
3. Executions get allocated to client accounts and results in generation of client trade

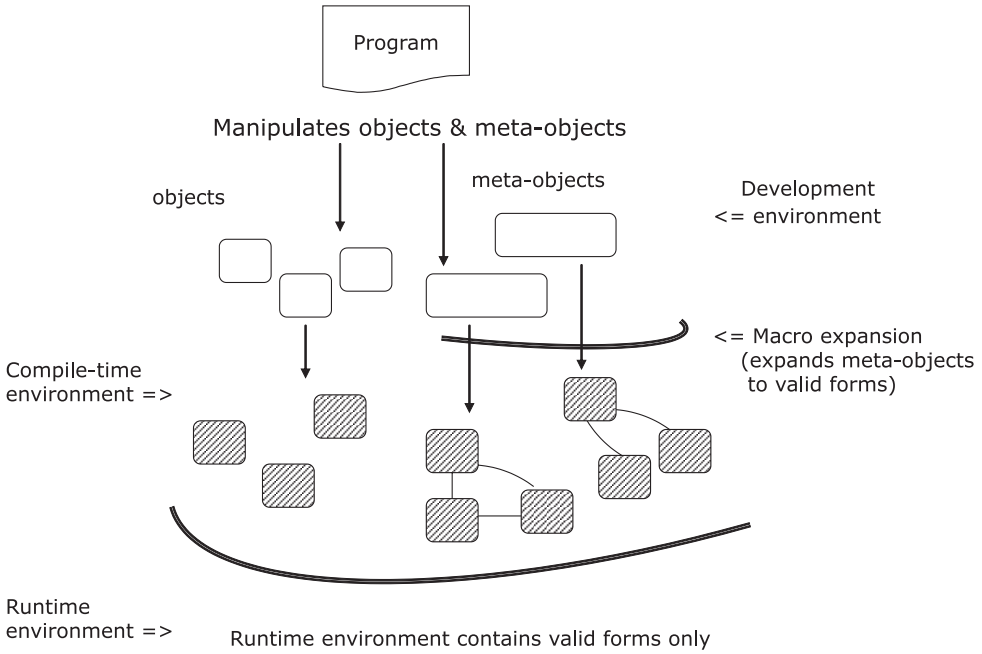


Figure 4.13 Compile-time metaprogramming generates code through macro expansion. Note that we're still in the compilation phase when the code is generated. This technique doesn't have any runtime overhead, unlike the earlier one in figure 4.12.

Let's try to implement the allocation process that generates the client trade from an execution. In reality there's a many-to-many relationship between orders, executions, and trades. For the purpose of this example, let's assume that there's a one-to-one relationship between execution and the client trade. We'll use the power of Clojure duck typing to model our use case.

> **Clojure tidbits you need to know**
>
> - *Prefix syntax and functional thinking.* Clojure uses prefix notation and is based on s-expressions. Clojure is uniform in syntax throughout, there's no order of operations, and it boasts absolute consistency. Clojure is a functional language in which you organize your modules as functions.
> - *Maps in Clojure* are used as a ubiquitous data structure to model objects.
> - *Macros* let you define syntactic extensions for your DSL. They make your DSLs concise through compile-time code generation.

4.5.2 Implementing the domain model

Here's how we define a trade and an execution in Clojure. Trade and execution essentially contain the same information. The difference is that an execution contains a

broker account, but a trade is done on a *client account*, based on the order that the client places. The following snippet defines a sample trade and a sample execution.

```
(def tr1
 {:ref-no "tr-123"
  :account {:no "cl-a1" :name "john doe" :type ::trading}
  :instrument "eq-123" :value 1000})

(def ex1
  {:ref-no "er-123"
   :account {:no "br-a1" :name "j p morgan" :type ::trading}
   :instrument "eq-123" :value 1000})
```

Clojure keyword ::trading ❶

An account is a separate structure embedded within the trade structure. An account contains an attribute `:type` that indicates whether it's a trading account or a settlement account. The type is modeled as a Clojure keyword that's nothing but a symbolic identifier that evaluates to itself ❶. Clojure keywords provide fast equality checks and are used as lightweight constant strings. For more information about client accounts and the various types that it supports, read the sidebar *Financial brokerage systems: client account* in section 3.2.2.

Let's define two functions in listing 4.14: one that checks whether an account is a trading account, and another that allocates an execution to a client account generating a client trade. The latter is the main domain problem use case that we'll focus on. We'll start with a function definition and see how you can make the implementation of the function more succinct when macros are generating repetitive boilerplates for it.

Listing 4.14 Allocation function for execution

```
(defn trading?
  "Returns true if the account is a trading account"
  [account]
  (= (:type account) ::trading))

(defn allocate
  "Allocate execution to client account and generate client trade"
  [acc exe]
  (cond
    (nil? acc) (throw (IllegalArgumentException.
                 "account cannot be nil"))
    (= (trading? acc) false) (throw (IllegalArgumentException.
                               "must be a trading account"))
    :else {:ref-no (generate-trade-ref-no)
           :account acc
           :instrument (:instrument exe) :value (:value exe)}))
```

❶ **Validation**

Look at what's going on in the `allocate` function in this listing. The core business logic that `allocate` handles is in the `:else` clause of the `cond` statement. The first two condition clauses ❶ perform two validations that we need to do for every operation that we carry out within the trading subsystem. For any method on trade, we need to validate that the trading account is a non-null entity and that it's indeed a trading

account and not a settlement account. That's what comprises the main explicit surface area in the `allocate` method. The `allocate` method contains accidental code complexity that needs to be factored away from the core API implementation.

4.5.3 The beauty of Clojure macros

Because the validations that we do in the `allocate` method are generally applicable to all trade functions, why not refactor them into a reusable entity? Then we'd have a validate function that can be invoked for all accounts like we did for `trd_validate` when we defined the Ruby module `TradeClassMethods` earlier in section 4.4.1. But macros provide distinct advantages, which I discuss in the callout.

Clojure supports macros you can use to generate code *during the compilation phase*; then the macro is expanded into the normal Clojure forms. The advantage is twofold:

1. You avoid the overhead of the function call through inline expansion of the macro during the compilation phase.
2. The code is more readable because you don't use the lambdas that would have been required if you used higher-order functions.

The next example will make all this clearer. In the following snippet, I define a macro that can be used as a control abstraction, much like the normal Clojure form, but that encapsulates all the validation logic within it.

```
(defmacro with-account
  [acc & body]
  `(cond
    (nil? ~acc) (throw (IllegalArgumentException.
                        "account cannot be nil"))
    (= (trading? ~acc) false) (throw (IllegalArgumentException.
                                      "must be a trading account"))
    :else ~@body))
```

Note how the `body` can consist of a variable number of forms that get inserted into the generated code via the splicing unquote `~@`. (For more details about how the splicing unquote works, refer to [3] in section 4.7.) If we implemented a function for the validation logic, we'd have to use lambdas instead. Using `with-account`, this is how our allocate function looks:

```
(defn allocate
 "Allocate execution to client account and generate client trade"
 [acc exe]
 (with-account acc
   {:ref-no (generate-trade-ref-no)
    :account acc
    :instrument (:instrument exe) :value (:value exe)}))
```

Now the implementation is more concise and succinct because it needs to focus only on the core domain logic. All exception handlings and accidental complexities are

Key takeaways from this section

The focus of this section was to *make your DSL implementation concise yet expressive.* This idea has been the recurring theme for all the sections in this chapter. The difference is in the implementation itself.

With Clojure macros, you make your DSL concise *without having any impact on the runtime performance.* You can bend the language itself to express the exact syntax and semantics that your DSL asks for. The Lisp family offers this awesome power as a natural way to model the real world.

being factored away to the macro, without incurring any overhead on the runtime performance. The macro `with-account` is not only coupled with the implementation of `allocate`; it's a general control structure, looks like a normal Clojure form, and is reusable across all the APIs that need trading accounts to be validated.

Generative DSLs write code for you. But as an observant reader, you must've already figured out the differences that language implementations offer with respect to when the code is generated. Section 4.4 discussed *runtime* code generation; in this section, you saw how Clojure implements *compile-time* code generation using the power of macros. Both strategies have their advantages and disadvantages, and you need to weigh your options before deciding on one specific implementation.

4.6 Summary

You've ridden with me for a long time in this chapter. You deserve kudos for being a part of this journey. We covered almost the entire range of internal DSL implementation patterns, with problem snippets from our domain of financial brokerage systems.

Key takeaways & best practices

When you design an internal DSL, follow the best practices for the language concerned. Using the language idiomatically always results in the optimal mix of expressiveness and performance.

A dynamic language like Ruby or Groovy gives you strong metaprogramming capabilities. Design your DSL abstractions and the underlying semantic model that rely on these capabilities. You'll end up with a beautifully concise syntax, leaving the boilerplates to the underlying language runtime.

In a language like Scala, static typing is your friend. Use type constraints to express a lot of business rules and use the compiler as the first-level verifier of your DSL syntax.

When you're using a language like Clojure that offers compile-time metaprogramming, use macros to define custom syntax structures. You'll get the conciseness of Ruby with no additional runtime performance penalty.

Dynamic languages like Ruby and Groovy offer powerful reflective metaprogramming paradigms that you can use to make your DSL implementations both concise and expressive. Such languages let you manipulate their meta-model during runtime, which makes them great tools for implementing more dynamic structures.

In this chapter, you saw how to make more dynamic builders and decorators that harness the power of metaprogramming. You also saw how you can use static typing to express domain constraints declaratively and how to implement generative DSLs that generate code during compile time or runtime.

Because you've read this chapter, you can now think in terms of mapping your DSL implementation to the idioms and best practices that your language offers. We talked about lots of patterns that you can easily map to appropriate contexts of your DSL model. In previous chapters, you heard about the virtues of DSL-based development, but you never got to see how a particular scenario of your problem domain maps to a specific implementation structure. The following list contains the main features that this chapter adds to your DSL landscape:

- You can now improve the conciseness of your DSL by using the power contained in your implementation language.
- If you're using a statically typed language, you can build your DSL models around typed abstractions.
- If your language offers metaprogramming, you can use it to reduce the surface syntax of the DSL and let the language runtime or the compile-time machinery generate the code for you.

Now it's time to look at more examples from the real world. In the next chapter, we'll discuss the dynamically typed family of languages and what they can do to implement expressive DSLs. So what are you waiting for? Grab your coffee mug, get a refill, and turn the page for more DSL delights.

4.7 References

1. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.
2. Konig, Dierk, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. 2009. *Groovy In Action,* Second Edition. Manning Early Access Program Edition. Manning Publications.
3. Halloway, Stuart. 2009. *Programming Clojure*. Pragmatic Bookshelf.
4. Odersky, Martin, Lex Spoon, and Bill Venners. 2008. *Programming in Scala: A Comprehensive Step-By-Step Guide*. Artima.
5. Coplien, James O. *Design Pattern Definition.* http://hillside.net/patterns/222-design-pattern-definition.

PROGRAMMING/SOFTWARE ENGINEERING

DSLs IN ACTION

Debasish Ghosh

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

Free eBook
SEE INSERT

"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

ISBN 13: 978-1-935182-45-0
ISBN 10: 1-935182-45-5
54499
9 781935 182450

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]