

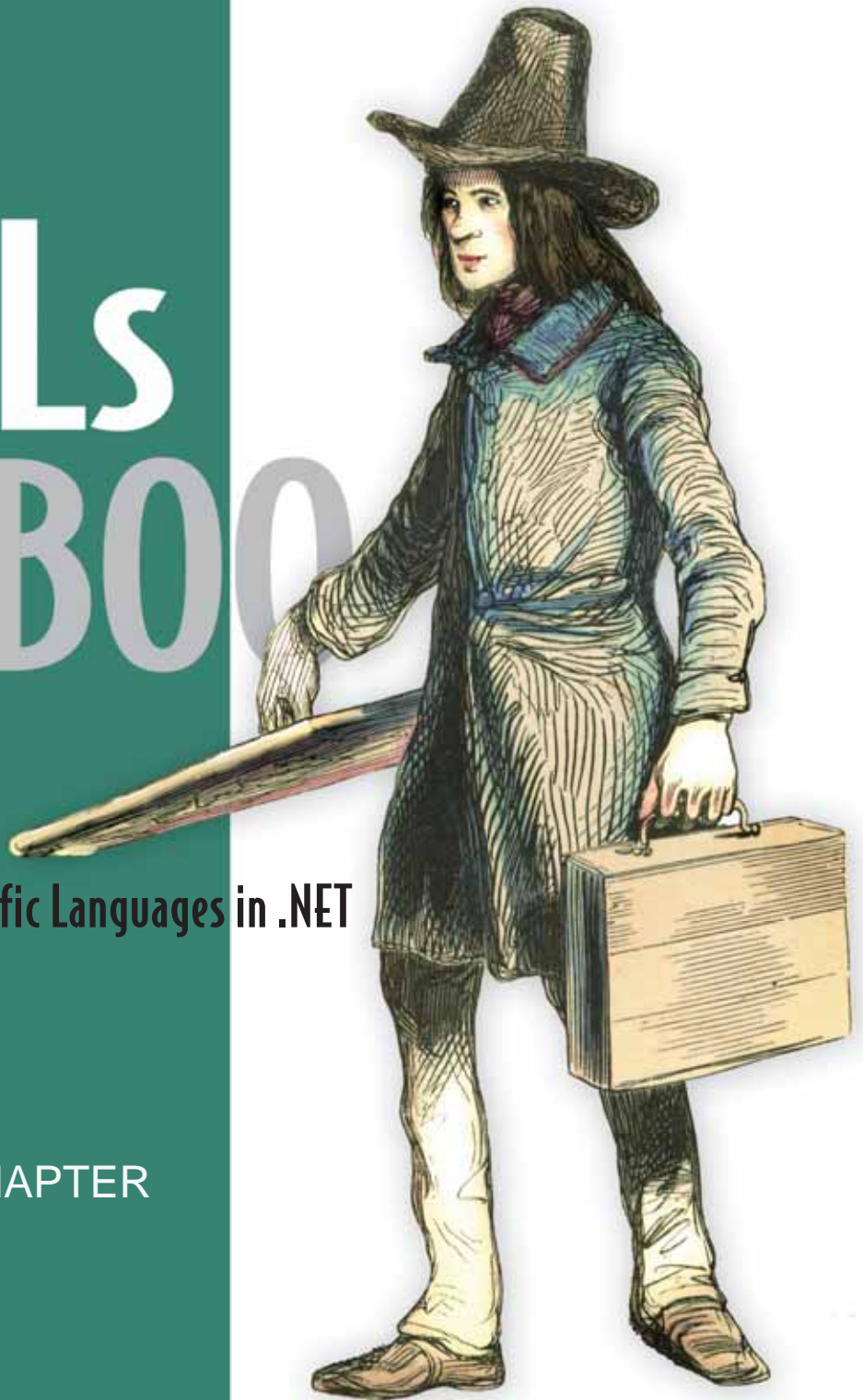
Ayende Rahien

DSLs in BOO

Domain-Specific Languages in .NET

SAMPLE CHAPTER

 MANNING





DSLs in Boo

Oren Eini

writing as Ayende Rahien

Chapter 1

brief contents

- 1 ■ What are domain-specific languages? 1
- 2 ■ An overview of the Boo language 22
- 3 ■ The drive toward DSLs 39
- 4 ■ Building DSLs 63
- 5 ■ Integrating DSLs into your applications 86
- 6 ■ Advanced compiler extensibility approaches 108
- 7 ■ DSL infrastructure with Rhino DSL 134
- 8 ■ Testing DSLs 150
- 9 ■ Versioning DSLs 173
- 10 ■ Creating a professional UI for a DSL 194
- 11 ■ DSLs and documentation 221
- 12 ■ DSL implementation challenges 239
- 13 ■ A real-world DSL implementation 263

What are domain-specific languages?



In this chapter

- Understanding domain-specific languages
- Distinguishing between domain-specific language types
- Why write a domain-specific language?
- Why use Boo?
- Examining domain-specific language examples

In the beginning, there was the bit. And the bit shifted left, and the bit shifted right, and there was the byte. The byte grew into a word, and then into a double word. And the developer saw the work, and it was good. And the evening and the morning were the first day. And on the next day, the developer came back to the work and spent the whole day trying to figure out what he had been thinking the day before.

If this story rings any bells, you're familiar with one of the most fundamental problems in computer science. The computer does what it is told, not what the programmer meant to tell it. Often enough, what the programmer tells it to do is in direct contradiction to what the programmer *meant* it to do. And that's a problem.

I've experienced this myself many times, and I'm not particularly incompetent. How, then, did I reach that point?

1.1 *Striving for simplicity*

Take a look at this piece of code:

```
for (p = freelist, oldp = 0;
     p && p != (struct chunk *)brkval;
     oldp = p, p = p->next) {
    if (p->len > nelems) {
        p->len -= nelems;
        q = p + p->len;
        q->next = 0;
        q->len = nelems;
        q++;
        return (void *)q;
    }
    if (p->len == nelems) {
        if (oldp == 0)
            freelist = p->next;
        else
            oldp->next = p->next;
        p->next = 0;
        p++;
        return (void *)p;
    }
}
```

You're among a decided minority if you can take a single glance at this code and deduce immediately what it's doing. Most developers would have to *decipher* this piece of code.

How does this connect to my difficulty in telling the computer what I want it to do? The problem is the level at which I instruct the computer what to do. If I am working down at the assembly level (or near assembly), I have to instruct the machine what to do in excruciating detail. The preceding piece of code was taken from the FreeBSD boot loader's `malloc` method, and there are good reasons it looks the way it does, but writing at this level has a big cost in productivity and flexibility.

Alternatively, I can instruct the computer to do things in higher-level terms, where it can better interpret what I want it to do.

As developers, we always want to achieve the simplest, clearest way to talk to the computer, regardless of the task at hand. Different tasks (low-level memory manipulation, for example) require us to work at different levels, but we always strive for readable, easily maintainable code. Within a given context, we may need to sacrifice those goals for other, more important goals (usually performance), but that should only be done very cautiously.

And when we are not working on low-level code, we will, at some point, have to leave general-purpose programming languages behind to get the desired level of clarity. Building our own languages, each focused specifically on a single task, is a great way to achieve this simplicity and clarity.

What we'd like to find are clear, concise, and simple ways to instruct the machine what we want it to do, rather than to laboriously micromanage it.

1.1.1 Creating simple code

Producing code that's readable, maintainable, and simple is a great goal. But simple code is much harder to write than complex code. It's easy to throw code at a problem until it goes away. Simple code, on the other hand, is what you get when you remove all the complexity from the code. That isn't to say that it's complicated to write simple code; it's just that writing *complex* code is easy. The amount of effort it takes to decipher what a piece of code does is a good indication of how simple the code is.

Consider these two examples of getting the date in two weeks' time. Which is more readable?

- C# code: `DateTime.Now.AddDays(14);`
- C code: `time() + 1209600;`

I don't think there's any question about which is more readable. In fact, an even better solution would be this:

```
DateTime.Now.AddWeeks(2);
```

But this isn't part of .NET's Base Class Library (BCL) `DateTime` API.

Using higher-level concepts means you can concentrate more on what you want to be done, and less on how it should be done at the machine level. When using .NET or Java, for instance, I rarely need to concern myself with memory allocation.

That's helpful, but more often than not, you'll need to do more interesting things than merely calculate the date two weeks from now. You'll need to express concepts and algorithms in ways that make sense, and you'll need to be able to use them in projects of significant size and complexity. Having clearer ways to express those concepts translates directly into a more maintainable code base, which means reduced maintenance costs and an easier time changing and growing the system.

NOTE It's considered polite to express intent in code in a manner that will make sense to the next developer who works with your code, particularly because that poor person may be you. A good suggestion that I take to heart is to assume that the next developer to touch your code will be an axe murderer who knows where you live and has a short fuse.

1.1.2 Creating clear code

Code may be clear about how it's doing things, but it might not be clear about what it's doing or why. Because we're assuming that the next developer will be a vicious killer with a nasty temper, we should make it easy to figure out what we've done and what we meant.

We can make our code easier to understand by using intention-revealing programming and concepts taken from domain-driven design, a design approach that says that your API, code structure, and the code itself should express intent, be expressed in the

language of the domain, and generally have a high correlation with the problem domain that the application is trying to solve.

Even then, we quickly reach a point where our ability to express intent is hampered by the syntax of the language that we're using.

1.1.3 **Creating intention-revealing code**

Programming languages make it easy to tell the computer what it should do, but they can be less effective at expressing developer intent. For that matter, most general-purpose languages (such as C# or Java) are far less suited for a host of other tasks.

Let's consider text processing, for example. Suppose you want to validate an Israeli phone number like this: 03-9876543. You might do this with the code shown in listing 1.1.

Listing 1.1 Validating a phone number

```
public bool ValidatePhoneNumber(string input)
{
    if (input.Length != 10)
        return false;
    for (int i = 0; i < input.Length; i++)
    {
        if (i == 2 && input[i] != '-')
            return false;
        else if (char.IsDigit(input[i]) == false)
            return false;
    }
    return true;
}
```

Can you look at this code and understand what input it will accept without *deciphering* it? If you haven't noticed by now, I consider the need to decipher code bad.

Now let's look at a tool that's dedicated to text processing: regular expressions. Validating the phone number using a regular expression is as simple as the one-liner in listing 1.2.

Listing 1.2 Validating a phone number using regular expressions

```
public bool ValidatePhoneNumber(string input)
{
    return Regex.IsMatch(input, @"^\d{2}-\d{7}$");
}
```

In this case, the use of a specialized tool for text processing has made the intent much easier to understand, but you need to understand the tool. Anyone who knows regular expressions can glance at the code and figure out what input it will accept.

Another approach is to use masked input to define a mask for certain input, which would result in code like this:

```
Mask.Validate(input, "##-#####");
```

The challenges of specialized tools

Regular expressions are notorious for being write-only tools because the results can be difficult to read, particularly if you don't write them carefully.

Using special tools to handle specialized tasks requires that you understand how to use the tools. If you don't understand regular expressions, and I hand you listing 1.2, how will you deal with it?

We'll touch on this topic later in the book; most of chapter 11 is dedicated to techniques that can help people come to grips with custom languages.

Assuming you know that # is the character for matching a numeral, this is even easier to understand than the regular expression approach. (The .NET framework doesn't have any masked-input validation facilities beyond WinForms' `MaskedTextBox`.)

Querying and filtering are other situations where code is no longer sufficient. Let's say we want to retrieve data for all the customers in London. This isn't a query that you'll want to handle by yourself. Building an optimized query plan, instructing the data store which section of the data should be scanned, building manual filters for each individual query ... all of that can be quite tedious. It's quite a complex task, particularly if you want to handle it efficiently and in a transaction-safe manner.

It's far easier to send a SQL statement to the database and let it sort out how it wants to handle the request on its own. This allows us to speak at a much higher level of abstraction and ignore the details of how the data is retrieved.

So far, I have been consciously avoiding the use of the term *domain-specific languages*, but it's time we started discussing it.

1.2 Understanding domain-specific languages

Martin Fowler defines a domain-specific language (DSL) as “a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem” (<http://martinfowler.com/bliki/DomainSpecificLanguage.html>).

Domain-specific languages aren't a new idea by any means. DSLs have been around since long before the start of computing. People have always developed specialized vocabularies for specialized tasks. That's why sailors use terms like *port* and *starboard* and are not particularly afraid of *gallows*. Doctors similarly have a vocabulary that is baffling to the uninitiated, and weather forecasters have specific terms for various types of clouds, winds, and storms.

Regular expressions and SQL are similarly specialized languages:

- Both are languages designed for a narrow domain—text processing and database querying, respectively.
- Both are focused on letting you express what you mean, not how the implementation should work—that's left to some magic engine in the background.

The reason these languages are so successful is that the focus they offer is incredibly useful. They reduce the complexity that you need to handle, and they're flexible in terms of what you can make them do.

1.2.1 **Expressing intent**

From the beginning of computer programming, it was recognized that trying to express what you mean in natural language isn't a viable approach. A clearer, much more focused, way to express intent was needed—that's why we have code, which is unambiguous (most of the time) and easy for the computer to understand.

But while code may be unambiguous to a computer, it can certainly be incomprehensible to people. Understanding code can be a big problem. You tend to write the code once, and read it many more times. Clarity is much more important than brevity. By ensuring that our code is readable, clear, and concise, we make an investment that will benefit us both in the immediate future (producing software that is simpler and easier to change) and in the long term (providing easier maintainability and a clearer path for extensibility and growth).

But, as we've seen, code isn't always the clearest way to express intent. This is where intention-revealing programming comes into play, and one of the tools in that category is creating a DSL to clearly and efficiently express intent and meaning in code.

1.2.2 **Creating your own languages**

Most people assume that creating your own computer language is a fairly complex matter. This is because most of the literature out there assumes that you want to build a full-blown general language. This puts a lot of burden on you, as the language author.

It isn't simple to create a general language, but it's certainly possible. It just isn't something you'd want to do on a rainy afternoon or over a long weekend. The experience is out there, but the initial cost remains nontrivial.

But you don't always have to write your own language from scratch. You can utilize an existing language (called the *host language* or *base language*) to provide built-in language and runtime facilities, and then add more syntax and behavior on top

Building your own compiler

I stated that building your own compiler or interpreter isn't hard. This is true, to some extent. The main difficulties in going that route are the scope of the work and the fact that most of the work is arcane at worst and tedious at best. This is particularly true if you want to write a full-fledged language.

Writing a general-purpose language is a big task. You need to deal with the details of the syntax and worry about creating an execution engine (for interpreted languages) or generating IL (Intermediate Language) or machine code (for compiled languages). I don't consider it to be a complex task, but it is a big one.

Building a single-purpose language is a far easier (and smaller) task, because the scope is much reduced. A good example of that can be seen in rSpec, a Ruby library for creating behavior-driven specifications. One of its capabilities is a story runner that accepts specifications written in English (<http://blog.davidhelimsky.net/articles/2007/10/21/story-runner-in-plain-english>). I suggest looking at how it works. It's quite ingenious in its simplicity.

The problem with that approach for natural language processing is that you hit its limits quickly. It works only when the statements follow a rigid format, so although it may look like natural language, it is, in fact, nothing of the sort. If you want to make the language more intelligent, you have to accept the additional complexity of building a more full-featured language.

I once consulted for a company that had built a DSL for defining business rules. They had over 100,000 lines of C++ code that they needed to maintain, and performance was a big concern. It became apparent that they could have switched the whole thing to an internal DSL (a DSL that's hosted in an existing language, which we'll talk about shortly) and saved quite a bit of time, effort, and pain.

of it. A popular example is Ruby on Rails, which is, in essence, a DSL for building web applications.

The tools for language-oriented programming had been improving for quite a while, but it was the introduction of Ruby on Rails—a wildly popular DSL that was recognized as such—that really started to get things rolling.

1.3 Distinguishing between DSL types

In the world of DSLs, we often distinguish between several types:

- External DSLs
- Graphical DSLs
- Fluent interfaces
- Internal or embedded DSLs

We'll discuss those types in turn, and look at their properties and uses.

1.3.1 External DSLs

When we talk about external DSLs, we're discussing DSLs that exist outside the confines of an existing language. SQL and regular expressions are two examples of external DSLs.

Building an external DSL means starting work from a blank slate. You need to define the syntax and required capabilities, and start working from there. This means that you have a lot of power in your hands, but you also need to handle everything yourself. And by "everything," I do mean *everything*, from defining operator precedence semantics to specifying how an `if` statement works.

Common tools for building external DSLs include Lex, Yacc, ANTLR, GOLD Parser, and Coco/R, among others. Those tools handle the first stage, translating text in a known syntax to a format that a computer program can consume to produce executable output. The part about “producing executable output” is usually left as an exercise for the reader. There are few tools to help you with that.

NOTE One tool that comes to mind for producing executable output is the Dynamic Language Runtime (DLR), a Microsoft project that aims to give us dynamic languages in .NET. One basic underpinning of this project is a set of classes that specify the behavior of a program (the abstract syntax tree, or AST) that the DLR can turn into an executable. There are other such tools, for sure, but the DLR is the only one I know of in the .NET space.

Building rich external DSLs is similar to building a general purpose language. You need to understand compiler theory before starting on that path. If you’re interested in that, I recommend reading *Compilers: Principles, Techniques, and Tools*, by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, which is a classic book on the subject.

This book focuses on building languages on top of existing languages, not starting from scratch and going the whole way. Nevertheless, some background in compiler theory is certainly helpful, even when building a DSL that uses an existing language, so let’s take a quick look at the process of building a language from scratch.

First, the grammar and syntax are often defined using a notation such as BNF (Backus-Naur Form) or a derivative, and then you use a tool to generate a parser. Once you’ve done that, you can run the parser over a code string, which will produce an abstract syntax tree (AST), which is the representation of the original string as an AST based on your definition of the language.

An example will make this clearer. Consider the code in listing 1.3, written in a fictional language.

Listing 1.3 An if statement in a fictional language

```
if 1 equals 2:  
    print "1 = 2"  
else:  
    print "1 != 2"
```

The AST that was generated from the code in listing 1.3 is shown in figure 1.1.

You can then either build an interpreter that understands this AST and can execute it, or output an executable from the AST. Another common approach is to transform the AST into a semantic model that’s easier to work with, but that has little correlation to the original text.

External DSLs are extremely powerful, but they also carry with them a significant cost. In general, I prefer to avoid going the external DSL route, mainly because of the cost, but also because internal DSLs serve well in most cases.

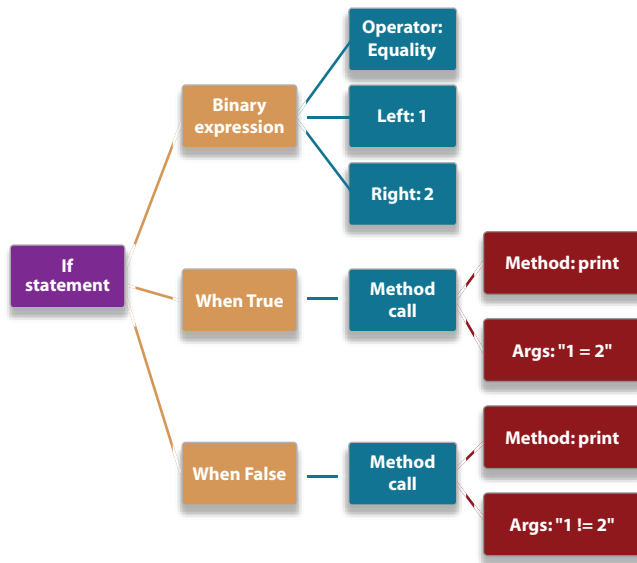


Figure 1.1 A hierarchical representation of the AST generated from a simple `if` statement

I would use an external DSL for specifying languages that are too far afield from existing programming languages. SQL is a good example of a DSL you couldn't build as an internal DSL. You could build something similar, but you couldn't get it quite right, so an external DSL would be the right approach for writing your own SQL dialect.

1.3.2 Graphical DSLs

Another form of DSL is the graphical DSL. This is a DSL that uses shapes and lines to express intent rather than using text.

UML is a good example of a graphical DSL. UML is a DSL for describing software systems, and quite a lot of money and effort has been devoted to making UML the one true model from which you can generate the rest of the application. Figure 1.2 displays a small part of a typical UML diagram.

Graphical DSLs are great for expressing a lot of information in a concise way. Often, it's much easier to understand a problem when you see it than when it's explained in words. This visualization approach also allows for communication at a high level because of the physical limitations of the image. It's much easier to understand what's going on because there is less information, and you can see the big picture.

A lot of effort has been invested in making it possible to write your own graphical DSLs. Microsoft has the Visual Studio DSL Tools, which is a framework that allows you to build tools similar to the class designer and generate code with them.

There are quite a few examples of graphical DSLs that you've probably heard about:

- UML
- BizTalk orchestrations and maps

Customer
Id : long
Name : string
Orders : Set<Order>
AddOrder(order : Order) : void

Figure 1.2 UML class diagram displaying the `Customer` object

- SQL Server Integration Services
- Windows Workflow Foundation

I've had some experience with all of these graphical DSLs, and they all share common problems inherent to the graphical DSL model.

The whole point of a graphical DSL is to hide information that you don't want to see, so you can see the big picture. This means that you can't see the whole at the same time, which leads to a lot of time spent jumping between various elements on the DSL surface, trying to gather all the required data. Graphical DSLs are visually verbose; they often need a lot of screen real estate to express notions that take a few lines with a textual DSL.

And then there are important UI issues. Searching is difficult in a graphical DSL, as are search and replace operations. And mouse-driven development isn't a good idea, if only in consideration of your wrists.

Beyond that, there are serious difficulties working with graphical DSLs in a team environment. It's easy to pull out a code file and compare two versions (called a *diff*, or *diffing*), but this breaks down for graphical DSLs, even those that persist their data into XML files. Even assuming the XML persistence format is human-readable (and I haven't seen an example that was), comparing the XML defeats the whole purpose of using a graphical DSL in the first place. You need some way to express a diff graphically, and I haven't seen any good way to do that.

This makes graphical DSLs a problem in terms of source control. This is a huge issue as far as I am concerned, and I have run into this issue more than once, with no easy or good solutions in sight.

If you haven't guessed so far, I'm not a fan of graphical DSLs for programming. Graphical DSLs are great for *documentation*, but I find that they aren't very good for development when it comes to real-world scenarios.

1.3.3 *Fluent interfaces*

I had some doubts about including fluent interfaces in this list of DSL types because I think of them as degenerate internal DSLs for languages with little syntactic flexibility (such as C# or Java), which means their options for language extensibility are limited. Fluent interfaces are ways to structure your API so that operations flow naturally and provide more readable code. They tend to be valid only when used by developers during actual development, which limits their scope compared to other DSLs.

It's easier to demonstrate than explain, so take a look at listing 1.4, which runs a set of transformations on an image using a fluent interface API.

Listing 1.4 *Fluent interface for specifying graphical transforms*

```
new Pipeline("rhino.png")
    .Rotate(90)
    .Watermark("Mocks")
    .RoundCorners(100, Color.Bisque)
    .Save("fluent-rhino.png");
```

The implementation of a fluent interface is simple. The `Rotate()` method is shown in listing 1.5.

Listing 1.5 Implementation of a method in a fluent interface

```
public Pipeline Rotate(float degrees)
{
    RotateFilter filter = new RotateFilter();
    filter.RotateDegrees = degrees;
    image = filter.ExecuteFilter(image);
    return this;
}
```

On the surface, fluent interfaces are simply a type of method chaining, but they have implications for the readability of the operations, as well as for the instructive nature that can result from building good fluent interfaces. By carefully planning the return values, we can create a good language with high readability *and* we can gain the support of IntelliSense to aid in the writing of the language statements.

The fluent interface in listing 1.4 isn't particularly impressive, but listing 1.6 should be. This is valid C# 2.0 code, and even if you've never used it before, this code will probably be instantly readable.

Listing 1.6 Using a fluent interface for querying

```
User.FindAll(
    Where.User.City == "London" &&
    Where.User.RegisteredAt >= DateTime.Now.AddMonths(-3)
);
```

This code gives you all the users from London that registered in the last 3 months. Unfortunately, this fluent interface is based on code generation, operator overloading, and generics abuse; it was *very* hard to create and it isn't something I'd want to create again. (This code was written for C# 2.0; LINQ, in C# 3.0, makes this example look aged.)

Listing 1.7 is another example that shows the fluent interface to configure `StructureMap`.

Listing 1.7 Configuring the StructureMap container using a fluent interface

```
registry.AddInstanceOf<IWidget>()
    .WithName("DarkGreen")
    .UsingConcreteType<ColorWidget>()
    .WithProperty("Color").EqualTo("DarkGreen");
```

NOTE `StructureMap` (<http://structuremap.sourceforge.net/>) is an Inversion of Control (IoC) container, probably the oldest on .NET. An Inversion of Control container is a tool that helps you manage dependencies and lifetimes of objects in your application. You can think about them as very smart factories, although that doesn't do them justice. You can read more about IoC containers here: <http://www.martinfowler.com/articles/injection.html>.

And, finally, listing 1.8 shows an example for specifying regular expressions in a comparatively readable way.

Listing 1.8 Using a fluent interface to create a regular expression

```
SmartRegex.Create("<div",
    SmartRegex.Space >= 0,
    "class='game'",
    SmartRegex.Space >= 0,
    ">");
```

Fluent interfaces are useful, and with C# 3.0 and VB.NET 9, we have some interesting options for expressing ourselves. The query syntax in listing 1.6 could easily be replaced by a LINQ query, but the other examples are still relevant. With the new capabilities like extension methods and lambdas, you can take fluent interfaces quite a long way.

Unfortunately, you usually can't take them far enough. Mainstream languages are too inflexible in their syntax to allow you the freedom to express yourself appropriately. I have tried this approach, and I have bumped into the limits of the language several times. This approach breaks down for the interesting scenarios.

This is particularly true when you want to express business requirements in a way that would make sense even to non-programmers. It would be good indeed if you could show businesspeople what you're doing, in a way that made sense to them.

This leads us directly toward the last item on our list, internal DSLs.

1.3.4 *Internal or embedded DSLs*

Internal DSLs are built on top of an existing language, but they don't try to remain true to the original programming language syntax. They try to express things in a way that makes sense to both the author and the reader, not to the compiler.

Obviously, the expressiveness of an internal DSL is limited by whatever constraints are imposed by the underlying language syntax. You can't build a good DSL on top of C# or Java; they have too much rigidity in their syntax to allow it. You probably could build a good DSL on C++, but it would probably include preprocessor macros galore, and I wouldn't place any bets on how maintainable it would be.

The popular choices for building internal DSLs are dynamic languages; Lisp and Smalltalk were probably the first common choices. Today, people mostly use Ruby, Python, and Boo. People turn to those languages for building DSLs because they have quite a bit of syntactic flexibility. For example, listing 1.9 is valid Ruby code.

Listing 1.9 A Rake build script, specifying tasks to run at build time

```
task :default => [:test]
task :test do
  ruby "test/unittest.rb"
end
```

Listing 1.9 is part of a build script written using Rake, a build tool that uses a Ruby-based DSL to specify actions to take during the build process (<http://rake.rubyforge.org/>). Rake is a good example of using a DSL to express intent in an understandable manner. Consider the amount of XML you'd need to write using an XML-based build tool, such as NAnt or MSBuild, to do the same thing, and consider how readable that would be.

Other features that usually appear in dynamic languages are also useful when building DSLs: closures, macros, and duck typing.

The major advantage of an internal DSL is that it takes on all the power of the language it's written for. You don't have to write the semantics of an `if` statement, or redefine operator precedence, for instance. Sometimes that's useful, and in one of my DSL implementations I *did* redefine the `if` statement, but that's probably not a good thing to do in general, and it's rarely necessary.

A DSL built on top of an existing language can also be problematic, because you want to limit the options of the language to clarify what is going on. The DSL shouldn't be a full-fledged programming language; you already have that in the base language, after all.

The main purpose of an internal DSL is to reduce the amount of work required to make the compiler happy and increase the clarity of the code in question. That's the syntactic aspect of it, at least. The other purpose is to expose the domain. A DSL should be readable by someone who is familiar with the domain, not the programming language. That takes some work, and it's far more important than mere syntax; this is the core reason for building a DSL in the first place.

Or is it? Why do we need DSLs again?

1.4 Why write DSLs?

Why do you need a DSL? After all, you're reading this book, so you already know how to program. Can't you use "normal" programming languages to do the job, perhaps with a dash of fluent interfaces and domain-driven design to make the code easier to read?

So far, I've focused entirely on the *how*, which is somewhat hypocritical of me, because this entire book is going to focus on abstracting the *how*. Let's look at the *why*—the different needs that lead the drive toward a DSL.

There are several reasons you might want a DSL, and they're mostly based on the problems you want to solve. These are the most common ones:

- Making a technical issue or task simpler
- Expressing rules and actions in a way that's close to the domain and understandable to businesspeople
- Automating tasks and actions, usually as part of adding scriptability and extensibility features to your applications

We'll look at each of those scenarios in detail and examine the forces that drive us toward using DSLs and the implications they have on the languages we build.

1.4.1 **Technical DSLs**

A technical DSL is supposed to be used by someone who understands the development environment. It's meant to express matters more concisely, but it's still very much a programming language at heart. The main difference is that a programming language is more general, whereas a technical DSL is focused on solving the specific problem at hand. As such, it has all the benefits (and drawbacks) of single-purpose languages. Rake, Binsor, Rhino ETL, and Watir are examples of technical DSLs.

NOTE As mentioned earlier, Rake is a build tool that uses a Ruby-based DSL to specify actions to be taken during the build process (<http://rake.rubyforge.org/>). Binsor is a DSL for defining dependencies for the Windsor IoC container (<http://www.ayende.com/Blog/category/451.aspx>). Rhino ETL is a DSL-based extract-transform-load (ETL) tool (<http://www.ayende.com/Blog/category/545.aspx>). And Watir is an automation DSL for driving Internet Explorer, mostly used for integration testing (<http://en.wikipedia.org/wiki/Watir>).

It makes sense to build a technical DSL if you need richer ways to specify what you want to happen. Technical DSLs are usually easier to write than other DSLs, because your target audience already understands programming—it takes less work to create a language that make sense to them.

In fact, the inclusion of programming features can make a sweet DSL indeed. We already saw a Rake sample (listing 1.9); listing 1.10 shows a Binsor example.

Listing 1.10 A Binsor script for registering all controllers in an assembly

```
for type in AllTypesBased of IController("MyApplication.Web"):  
  # Component is a keyword that would register the type in the container  
  component type
```

It takes two lines to register all the controllers in the application. That's quite expressive. It's also a sweet merge between the use of standard language operations (`for` loops) and the DSL syntax (`component`).

This works well if your target audience is developers. If not, you'll need to provide a far richer environment in your DSLs. We usually call this type of DSL a business DSL.

1.4.2 **Business DSLs**

A business DSL needs to be (at the very least) readable to a businessperson with no background in programming. This type of DSL is mainly expressive in terms of the domain, and it has a lot less emphasis on the programming features that may still exist. It also tends to be more declarative than technical DSLs. The emphasis is placed on the declarative nature of the DSL and on matching it to the way the businesspeople think about the tasks at hand, so the programming features are not necessary in most cases.

For example, you wouldn't generally encourage the use of `for` loops in your business DSL, or explicit error handling, null checking, calling base class libraries, or any of the things that you would normally do in a technical environment. A business

DSL should be a closed system that provides all the expected usages directly in the language.

I can't think of a good example of a non-proprietary business DSL. There are business rule engines, but I wouldn't call them DSLs. They're one stage before that; they have no association to the real domain that we work with.

A good example of a business DSL that I have seen was created by a mobile phone company that needed to handle the variety of different contracts and benefits it offered. It also needed a short time to market, to respond rapidly to market conditions.

The end result was a DSL in which you could specify the different conditions and their results. For instance, to specify that you get 300 minutes free if you speak over 300 minutes a month, you would write something similar to listing 1.11.

Listing 1.11 A DSL for specifying benefits in a mobile phone company

```
when call_minutes_in_current_month > 300 and
  ➤      has_benefit "300 Minutes Free!!!":
      give_free_call_minutes 300, "300 Minutes Free!!!"
```

This DSL consists of a small language that can describe most of the benefits the company wants to express. The rest is a matter of naming conventions and dropping files in a specified folder, to be picked up and processed at regular intervals. We'll discuss the structure of the engine that surrounds the DSL itself in chapter 5.

Listing 1.11 still looks like a programming language, yes. But although a businessperson may not always be able to write actions using a business DSL, they should be able to read and understand them. After all, it's their business and their domain that you're describing. We'll see more complex examples later in the book, but for now let's keep this simple.

Using a business DSL requires business knowledge

This is something that people often overlook. When we evaluate the readability of a DSL, we often make the mistake of determining how readable it is to the layperson.

A business DSL uses business language, which can be completely opaque to a layperson. I have no idea what it means to adjust a claim, but presumably it makes sense to someone in the insurance business, and it's certainly something I would expect to see in a DSL targeted at solving a problem in the insurance world.

Why wouldn't a businessperson be able to write actions using a business DSL? One of the main reasons is that even a trivial syntax error would likely stop most nonprogrammers in their tracks. Understanding and overcoming errors requires programming knowledge that few businesspeople have. Although a DSL is supposed to be readable for nonprogrammers, it's still a programming language with little tolerance for such things as omitting the condition in an `if` statement, and many businesspeople would be unable to go over the first hurdle they faced.

It's important to know your audience—don't assume anything about your audience's ability or inability to write code. Although you might not expect them to understand programming, they may have experience in automating small tasks using VBA and Excel macros.

Matching the business DSL's capabilities to that of the expected audience will prove a powerful combination. You can provide the businesspeople with the tools, and they can provide the knowledge and the required perspective.

Conversely, creating a DSL that misses the target audience is likely to result in problems. In a business DSL, expressing the domain and the concepts in the language is only half the work; the other half is matching the language's capabilities and requirements to the people who will use it.

I suggest making the decision about whether you're creating a business-readable or a business-writable DSL as early in the game as you can possibly can. This decision will greatly affect the design and implementation of the DSL, and getting it wrong is likely to be expensive.

1.4.3 Automatic or extensible DSLs

Automatic or extensible DSLs may also be called IT DSLs. This type of DSL is often used to expose the internals of an application to the outside world.

Modern games are usually engines configured with some sort of scripting language. Another use for this style of DSL would be to get into the internals of an application and manage it. With such a DSL, you could write a script that would reroute all traffic from a server, wait for all current work to complete, and then take the server down, update it, and bring it up again.

Right now, it's possible to do this with shell scripts of various kinds, but most enterprise applications have a rich internal state that could be made at least partially visible. A DSL that would allow you to inspect and modify the internal state would be welcome. Many administrators would appreciate having more options for managing their applications.

Another way to look at this is to consider all the VBA-enabled applications out there, from Office to AutoCAD to accounting packages and ERP systems. VBA's extensibility enables users to create scripts that access the state of the system. The same thing can be done for enterprise applications using automation DSLs (at far less cost in licensing alone).

1.5 Boo's DSL capabilities

I've mentioned Lisp, Smalltalk, Ruby, Python, and Boo as languages that are well suited for writing internal DSLs, so why does this book focus on Boo? And have you even heard of this language? Boo has yet to become a household name (but just you wait), so we probably need to discuss what kind of language it is.

Boo is an object-oriented, statically typed programming language for the Common Language Infrastructure (CLI) with a Python-inspired syntax and a special focus on language and compiler extensibility. It's this focus on extensibility that makes it ideally

Boo runs on Java as well—say hello to BooJay

Boo is not just a CLR language; it's also a JVM language. You can learn more about Boo on Java in the BooJay discussion group: <http://groups.google.com/group/boojay/>.

This screencast will introduce you to what BooJay is capable of: http://blogs.codehaus.org/people/bamboo/archives/001751_experience_boojay_with_monolipse.html.

suiting for building DSLs. That it runs natively on the Common Language Runtime (CLR; the technical name for the .NET platform) is a huge plus, because it means that your existing toolset is still relevant.

I dislike pigeonholing myself, but I'll readily admit that I mostly work on software based on the .NET Common Language Runtime. This is a common, stable platform¹ with a rich set of tools and practices. It makes sense to keep my DSL implementation within this platform because I already know most of its quirks and how to work around them. I can use my existing knowledge to troubleshoot most problems. Staying within the CLR also means that I'll have little problem when calling a DSL from my code, or vice versa—both the DSL and my code are CLR assemblies and interoperate cleanly.

Figure 1.3 shows an application that makes use of several DSLs. Those DSLs can access the application logic easily and natively, and the application can shell out to the DSL for decisions that require significant flexibility.

As much as I like the CLR, though, the common languages for it aren't well suited to language-oriented programming—they're too rigid. Rigid languages don't offer many options to express concepts. You have the default language syntax, and that's it.

Boo *is* a CLR language with a default syntax that's much like Python and with some interesting opinions about compiler architecture. Because it's a CLR language, it will compile down to IL (Intermediate Language—the CLR assembly language), and it will be able to access the entire base class library and any additional code you have lying around. It also will perform as fast as any other IL-based language. You don't sacrifice performance when you choose to use a DSL—at least not if you go with Boo.

In fact, you can debug your Boo DSL in Visual Studio, profile it with dotTrace (a really sweet .NET profiler from JetBrains: <http://www.jetbrains.com/profiler/>), and

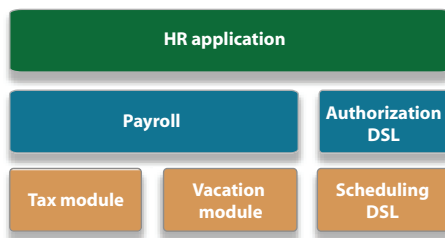


Figure 1.3 DSLs used as integral parts of an application

¹ Stable as long as you don't start playing with reflection emit and generics; there are dragons in that territory.

Using IronRuby or IronPython as host languages for DSLs

What about IronRuby and IronPython? They are CLR implementations of languages that have already proven to be suited for building DSLs.

I have two major issues with using these host languages for my DSLs. The first is that, compared to Boo, they don't offer enough control over the resulting language. The second is that both languages run on the Dynamic Language Runtime, which is a layer on top of the CLR, which is what Boo runs on.

This means that calling into IronRuby or IronPython code from C# code isn't as simple as adding a reference and making the call, which is all you need to do in Boo's case.

even reference it from any .NET language (such as C# or VB.NET). Similarly, your code (in any .NET language you care to name) will be able to make calls into the DSL code. This is the most common way to execute a DSL—by simply calling it.

What makes Boo special is that the compiler is open. And not open in the sense that you can look at the code, which isn't particularly useful unless you're writing compilers; what I mean is that you can easily change the compiler object model while it's compiling your code.

As you can imagine, this has some significant implications for your ability to use Boo for language-oriented programming. In effect, Boo will let you modify the language itself to fit your needs. I'll spend most of chapters 2 and 6 explaining this in detail, and you'll see what really made me choose Boo as the host language for my DSL efforts.

Boo also contains some interesting features for language-oriented programming, such as meta-methods, quasi-quotation, AST macros, and AST attributes.

As I mentioned, we'll explore Boo's language-oriented programming features in full in chapters 2 and 6. But for now, let's look at a few real-world DSLs written in Boo.

1.6 Examining DSL examples

Before we conclude this chapter, I want to give you a taste of the kind of DSLs you can create in Boo. This should give you some idea about the flexibility of the language.

1.6.1 Brail

Brail will probably remind you strongly of classic ASP or PHP. It's a text templating language, built by me, in which in you can mix code and text freely. Here's a sample:

```
<h1>My name is ${name}</h1>
<ul>
  <%   for element in list: %>
    <li>${element}</li>
  <% end %>
</ul>
```

This example will output a header and a list in HTML format based on the input variables passed to it.

If you're interested, you can read up on Brail at the Castle Project: <http://www.castleproject.org/monorail/documentation/trunk/viewengines/brail/index.html>.

1.6.2 Rhino ETL

I built this DSL after I'd had enough of using an ETL tool that wasn't top-notch, to say the least. That ETL tool also used a graphical DSL as the building block, and the pain of using it on a day-to-day basis is one of the main reasons I dislike graphical DSLs.

NOTE ETL stands for extract, transform, and load. It's a generic term for moving data around in a data warehouse. You *extract* the data from one location, *transform* it (probably with data from additional locations), and *load* it into its final destination. The classic example is moving data from a relational database to a decision-support system.

Rhino ETL employs the concept of steps, with data flowing from one step to the next. Usually the first step will extract the data from some source, and the last will load it to the final destination.

Here's an example of a full ETL process:

```
operation split_name:
  for row in rows:
    continue if row.Name is null
    row.FirstName, row.LastName = row.Name.Split()
    yield row

process UsersToPeople:
  input "source_db", Command = "SELECT id, name, email FROM Users"
  split_names()
  output "destination_db", Command = ""
    INSERT INTO People (UserId, FirstName, LastName, Email)
    VALUES (@UserId, @FirstName, @LastName, @Email)
    """:
    row.UserId = row.Id
```

This code gets the users list from the source database, splits the names, and then saves them to the destination database. This is a good example of a DSL that requires some knowledge of the domain before you can utilize it.

There's more information about Rhino ETL at <http://www.ayende.com/Blog/category/545.aspx>.

1.6.3 Bake (Boo Build System)

NAnt is an XML-based build system that works for simple scenarios, but it gets very complex very fast when you have a build script of any complexity. Bake, written by Georges Benatti, takes much the same conceptual approach (using tasks and actions), but it uses Boo to express the tasks and actions in the build script.

The resulting syntax tends to be easier to understand than the equivalent NAnt script at just about any level of complexity. XML has no natural way to express conditions and loops, and you often need those in a build script. It's much easier to read a

build script when you're using a programming language to natively express concepts such as control flow. Interestingly enough, the ease of readability holds not only for complex build scripts, but also for simple ones, if only because XML-based languages have so much ceremony attached to them.

Here's a simple example that will create the build directory and copy all DLLs to it:

```
Task "init build dir":
  if not Directory.Exists("build"):
    Mkdir "build"
  Cp FileSet("lib/*.dll").Files, "build", true
```

You can find out more about the Boo Build System at Google Code: <http://code.google.com/p/boo-build-system/>.

1.6.4 Specter

Specter is a behavior-driven development (BDD) testing framework, written by Andrew Davey and Cedric Vivier. It allows developers to build specifications for the object under test instead of asserting their behavior. You can read more about BDD here: <http://behaviour-driven.org/>.

Using Specter makes Boo behave in a way that's a better match for BDD. Here's an example for specifying how a stack (the data structure) should behave:

```
context "Empty stack":
  stack as Stack
  setup:
    stack = Stack()

  specify stack.Count.Must == 0

  specify "Stack must accept an item and count is then one":
    stack.Push(42)
    stack.Count.Must == 1
```

You can find out more about Specter here: <http://specter.sourceforge.net/>.

1.7 Summary

By now, you should understand what a DSL is. It used to be that the investment required to create a DSL was only justified for the big problems, but the tools have grown, and this book will help you understand how you can create a nontrivial language on a rainy afternoon.

Of all the DSL types presented in this chapter, I most strongly favor internal DSLs for their simplicity, extensibility, and low cost compared to the other approaches. Fluent interfaces are also good solutions, but they are frustrating when you bump into the limits of the (rigid) host language.

I have a strong bias against graphical DSLs for their visual verbosity, their complexity of use, and, most importantly, for the mess they often make out of source control. A graphical DSL doesn't lend itself to diffing and merging, which are critical in any scenario that involves more than a single developer.

Fortunately, this book isn't about graphical DSLs. It's about internal (or embedded) DSLs written in Boo. In chapter 2, we'll take a quick dive into Boo; just enough to get your feet wet and give you enough understanding of the subject to start building interesting languages with it.

Without further ado, let's get on with Boo.

DSLs in BOO

Ayende Rahien



A general purpose language like C# is designed to handle all programming tasks. By contrast, the structure and syntax of a Domain-Specific Language are designed to match a particular applications area. A DSL is designed for readability and easy programming of repeating problems. Using the innovative Boo language, it's a breeze to create a DSL for your application domain that works on .NET and does not sacrifice performance.

DSLs in Boo shows you how to design, extend, and evolve DSLs for .NET by focusing on approaches and patterns. You learn to define an app in terms that match the domain, and to use Boo to build DSLs that generate efficient executables. And you won't deal with the awkward XML-laden syntax many DSLs require. The book concentrates on writing internal (textual) DSLs that allow easy extensibility of the application and framework. And if you don't know Boo, don't worry—you'll learn right here all the techniques you need.

What's Inside

- Introduction to DSLs, including common patterns
- A fast-paced Boo tutorial
- Dozens of practical examples and tips
- An entertaining, easy-to-follow style

A leader in the .NET community, **Ayende Rahien**, whose real name is **Oren Eini**, contributes to numerous open-source projects including NHibernate, Castle, and Rhino Mocks. He blogs and speaks on architecture, data access, testing, and other topics.

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

“A great gateway into Boo and Domain-Specific Languages.”

—Justin Chase, Microsoft

“Useful, readable, and empowering—really captures the Boo spirit.”

—Avishay Lavie, Contributor to *The Boo Programming Language*

“Goes way beyond Boo particulars into universally applicable guidance.”

—Mark Seemann, Safewhere

“... will erase any doubts you may have about writing your own DSL.”

—Garabed “Garo” Yeriazarian Baker Hughes, Inc.

