

BUILDING Domain Specific Languages IN BOO

Ayende Rahien



 MANNING



Unedited Draft



**MEAP Edition
Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Introduction

Chapter 1 What are Domain Specific Languages?

Chapter 2 The drive toward Domain Specific Languages

Chapter 3 A short overview of the Boo language

Chapter 4 Digging into advance compiler extensibility approaches

Chapter 5 Building an application with DSL

Chapter 6 Structuring DSL

Chapter 7 Using Rhino DSL

Chapter 8 Testing our DSL

Chapter 9 Versioning DSL

Chapter 10 Creating professional DSL

Chapter 11 Documentation and user guides for the DSL

Chapter 12 Performance and other gotcha

Chapter 13 Best practices for DSL

Appendix A Creating an external DSL

Appendix B Creating a graphical DSL

Appendix C Frequently asked questions

1

What are Domain Specific Languages?

In this chapter:

- Understanding Domain Specific Languages
- Distinguishing between domain specific languages types
- Why do we want to write a Domain Specific Language?
- Examining Domain Specific Languages Examples

At first, 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 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 couldn't figure out what he was thinking last night, and spent the whole day figuring that out.

If this story rings any bells, then you understand the most fundamental problem in computing science. The computer does *what it was told*; it does not do *what I meant*. I have tried for a very long time to make the computer do what I mean, but no matter what flags I defined, or what threats were made, it keep doing precisely what it was told to do..

Often enough, what I told it to do was in direct contradiction to what I *meant* it to do. And that is a problem, a big one.

Now, I am not *that* much of an incompetent, at least I hope so.

How then, did I reach that point?

Before I can answer that, take a look at this piece of code:

```
push ebp
mov ebp, esp
pop ebp
lea eax, [edx+edx]
add eax, edx
shl eax, 2
sub cl, al
shr cl, 1
mov ea, al
ret
```

You are among a decided minority if you can take a single glance at this piece of code and deduct immediately what it is doing. Most developers would have to decipher that, at best.

How does this connect to my inability to tell the computer to do what I mean?

The problem is with the level in which I instruct the computer to do what I want. If I am working on the assembly language level, I have to very carefully instruct what to do, in *excruciating* detail.

I can turn up the level in which I talk to the computer, in which case I can instruct it to do things in terms that are higher level, and are able to capture what I want better.

We want to achieve the simplest, clearest way to talk to the computer. This is what this book is all about. At some point, we have to leave traditional programming languages in order to get the desired level of clarity. We will use this chapter to discuss approaches for achieving this goal, starting with what simple means.

1.1 Striving for simplicity

At all times, I want to produce code that is readable, maintainable and simple. Simple doesn't mean that it isn't doing its job, simple is what you get when you took a piece of code and removed everything that isn't related to the job that it is supposed to do.

Simple code is much harder to write than complex code.

The amount of effort that I need to dedicate to decipher what a piece of code does is a good indication to how simple the code is.

A simple example would be getting the date in two weeks. What is more readable?

C# Code to manipulate dates	C code to manipulate dates
<code>DateTime.Now.AddDays(14);</code>	<code>time() * 1209600;</code>

I don't think that there is a question between the readability of both solutions. In fact, an ever better solution would be:

`DateTime.Now.AddWeeks(2);`

This is not part of the CLR DateTime API, however.

Using higher level languages mean that I am concentrating less on how I want things to be done, in the machine level, and more on what I want to be done. Using .Net or Java, for instance, I rarely need to concern myself with memory allocation.

Nevertheless, there is still a problem. More often than not, I need to do more interesting things than merely calculate the date in two weeks. I need to express concepts and algorithms in ways that makes sense, and I usually need to build those with systems with significant size and complexity.

And, just to make it interesting, it is considered polite to do this in a manner that would make sense to the next developer that would have to touch this code. Especially since that poor person may very well be you. A good suggestion that I take to heart is to assume that the next developer to touch your code is an axe murderer that knows where you live, and has a *very* short fuse.

I was at the receiving end of code with... questionable quality several times, and there were times when the idea of getting an axe and demonstrating what Separation of Concerns meant had held an amazing amount of appeal; although I had a different idea about the concerns that needed to be separated at this point.

Under those circumstances, plain code may be clear in how it is doing things, but not really a good example of clarity in what it is doing or why. And, since we assume that the next developer is going to be a vicious killer with a nasty temper, we really want to make it easy to figure out what we are doing and what we meant.

We can make it work using Intention Revealing Programming and concepts taken from Domain Driven Design, and make sure that the API and the resulting code express intent, rather than mechanics.

Even then, we quickly reach a point where our ability to express intent is hampered by the syntax of the language that we are using. Programming languages are very good in expressing what the computer should do; they are less effective in expressing the developer intent when the code was written.

For that matter, they are far less suited to a host of other tasks. Let us take text processing for example. Assume that I want to validate a phone number like this: 01-23456789. Let us see it with code, shall we (as shown in listing 1.2)?

Listing 1.2 – 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 == 3 && input[i] != '-')
            return false;
        else if (char.IsDigit(input[i]) == false)
            return false;
    }
    return true;
}
```

Can you look at the code and understand what input it will accept without actually *deciphering* it?

If you haven't noticed by now, I consider the requirement to decipher code a bad idea. It is much safer all around, considering all the axe murderers that call themselves maintenance programmers.

Now let us look at a tool that is dedicated toward text processing, Regular Expressions. Validating the phone number using a regular expression is as simple as this one liner in Listing 1.3:

Listing 1.3 – Validating a phone number using regular expressions

```
public bool ValidatePhoneNumber(string input)
{
    return Regex.IsMatch(input, @"^\d{2}-\d{8}$");
}
```

In this case, the usage of a specialized tool for text processing has made the intent much easier to understand. Now I can glance at the code and, assuming that I know Regular Expressions, figure out immediately what input it would like to accept.

The other side of regular expressions

Regular Expressions are notorious for being a write only tool, which is somewhat true, and certainly a subject that we will spend some time discussing further along in the book.

This also expose another issue, the use of a special tool to handle specialized task requires that you'll understand the usage of that tool. If you have no idea about regular expressions and I hand you the listing 1.3, how are you going to deal with it? We will touch this topic as well further on in the book.

Let us take a look at querying and filtering, as another place where just code is no longer sufficient. Trying to get all the customers that live in London is not a query that you want to handle in code. It is quite complex issue to handle, especially if you want to handle this efficiently and in a transaction-safe manner.

It is far easier to send an SQL statement to the database and let that sort out how it wants to handle it on its own.

So far, I have been consciously avoiding the use of the term Domain Specific Languages, but considering the title of the book, I feel that we should start discussing it at some point. Actually, it looks like there is no avoiding it, actually.

1.2 Understanding Domain Specific languages

What commonality can we find between Regular Expressions and SQL?

Both are languages designed to handle a specific task, both are focused on letting you express what you meant, not the actual implementation details of how the execution should work.

That is left to some magic engine in the background that can take that meaning and turn it into something that is a computer can run. Both are languages for a very narrow domain, text processing and database querying, respectively.

The reason that they are so successful is that they are incredibly useful. They reduce the complexity that you need to handle and they are very flexible in terms of what you can make them do.

Martin Fowler¹ defines Domain Specific Languages as:

Domain specific language (DSL) is 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.

Domain Specific Languages (which I will abbreviate to DSL from now on) are not a new idea by any means. They have been around long before the start of computing.

People have always developed specialized vocabularies as a way to express more than the common language can express. That is why on sea, sailors use terms like port and starboard and shiver my timbers as well as a whole host of other terms to be able to communicate clearly even in bad conditions².

That is why Eskimos has plenty of words for snow, and why it is important to make a distinctions between accounts receivable and accounts payable if you are building an accounting package.

The situation has been much the same in computer science as well. From the first computer program by Ada Lovelace to the mini languages so common in Unix³, from UML to BNF, etc.

From the beginning, it was recognized that trying to express what you mean in plain code is not really a workable approach. You needed to have a clearer, much more focused, way to express your intent.

¹ Martin Fowler – Domain Specific Language - <http://martinfowler.com/bliki/DomainSpecificLanguage.html>

² Well, arguably shiver my timbers is not really useful for clear communication.

³ Awk, Sed and Grep come to mind immediately, for instance.

Some people believe that drawing diagrams and detailed documentation is the way to go. You can't express yourself in code in a sufficiently clear manner, they say, so you must express intent and purpose elsewhere.

I obviously disagree with this statement. I think that expressing yourself in code is not only possible, but desirable, if only because the code is the one true representation of the system, and the only one that is guaranteed to be in sync with what is actually going on.

However, as we have seen, code isn't always the clearest of methods to express intent. This is where intention revealing programming comes into play. One of the tools in that particular box is creating DSL to allow us to express clearly and efficiently express intent and meaning in code.

Up until recently, creating DSL was a fairly complex matter. You needed to understand compiler theory and BNF, for a start. Then you had to build an abstract syntax tree and create an interpreter or output the executable binary.

It isn't that hard, when you get down to it, but it is not something that you would want to do in a rainy afternoon or over a long weekend. The experience is out there, but the initial cost was nontrivial.

Building your own compiler

I stated that building your own compiler / interpreter isn't hard. This is true, to some extent. The main difficulties in going with 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 especially true if you want to write a fully fledged language.

Writing a single purpose language is actually not that hard, it is when you want to scale this language up that the complexity arrives. When I am talking about scaling the language I am talking about adding such things as the if statement, operators precedence, loops and the like.

And those are the very basics, mind you.

Appendix [TODO: appendix number] contains a simple implementation of a language that can be used to specified language rules. It has its own "parser" and "interpreter", if they can be called that.

It demonstrates the ease in which you can start working, and the complexity threshold that you need to pass to make this into a more useful approach.

Recently I was consulting for a company that had built its own DSL for defining business rules. They have over 100,000 lines of C++ code that they need to maintain, and apparently performance is a big concern at the moment.

After looking at it for a while, it became apparent that they could have switched the whole thing to an internal DSL and save quite a bit of time, effort, and pain. They would also have gotten much better performance from it, as an aside.

Recently, however, the situation has changed. The first popular example that I can think about is probably Ruby on Ruby, which is, in essence, a DSL for building web applications.

That has sparked quite a bit of interest in the field, but things have been moving steadily toward more accessible language oriented programming for quite a while. The tools that we have for building languages have been improving, but it was the introduction a wildly popular DSL (that was recognized as such) that really started to get things rolling.

1.3 Distinguishing between domain specific languages types

When we talk about DSL, we have several distinct types that come to mind.

- External DSL
- Graphical DSL
- Fluent Interfaces
- Internal / Embedded DSL

We will discuss those types in turn, and learn what properties and uses they have.

1.3.1 External DSL

When we talk about external DSL, we are discussing a DSL that exists outside the confines of an existing language. SQL and Regular Expressions are two examples of external DSL.

When building an external DSL, we are starting from scratch. We define your syntax and the required capabilities, and start working from there. This means that you have a lot of power in your hands, but on the other hand, it also means that you need to handle everything yourself.

By everything, I do mean everything, from defining operator precedence semantics to specifying how if statement works.

Common tools for building external DSL include Lex and Yacc, ANTLR, Coco/R, and many others.

Basically, you are starting by defining the language syntax. Usually using BNF or a derivative, afterward you use the tool to generate a parser. You can now run the parser over a code string. The end result would be an Abstract Syntax Tree (AST), which would be the representation of the original string, in the language AST.

Perhaps an example would make it clearer. Given the following code, written in a fictional language:

Listing 1.4 – An if statement in a fictional language

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

We can see the abstract syntax tree that was generated from this piece of code in figure 1.1.

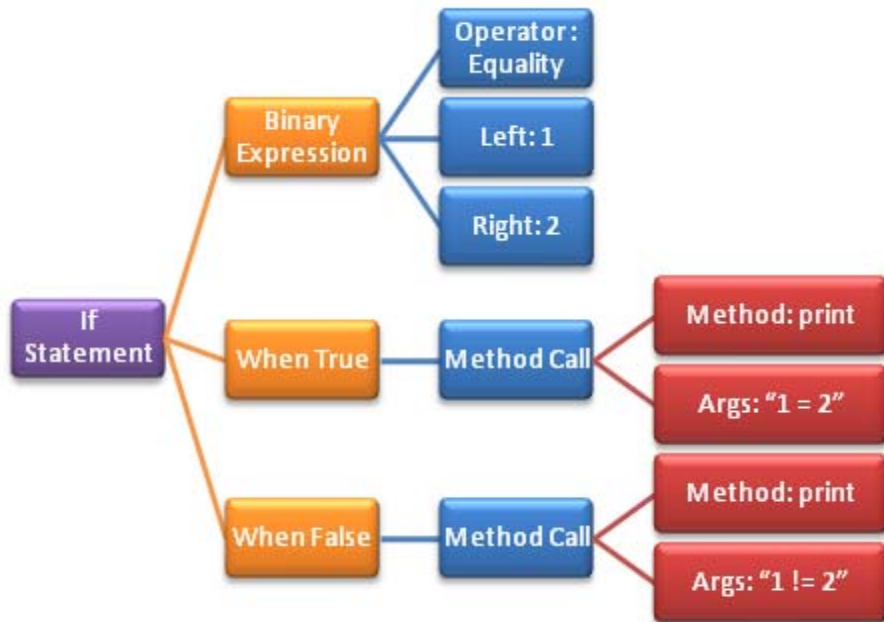


Figure 1.1 – A hierarchical representation of the AST generated from a simple if statement.

You can then either build an interpreter that understand this AST and can execute it, or output some sort of an executable from this AST. Another common approach is to

transform this AST into another form of AST that is easier to work with, but has less correlation to the original text.

External DSL are extremely powerful, but they also carry with them a not insignificant cost. In general, I prefer to avoid going with the external DSL route, mainly because of the cost issue, but also because internal DSL serves me quite well in most cases.

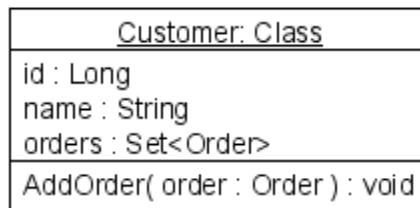
I would use an external DSL for specifying languages that are too far afield from programming languages. SQL is a good example for that. You cannot build that with an internal DSL. You can build something that is similar, but you cannot get it quite right.

1.3.2 Graphical DSL

Another form of DSL is the graphical DSL. What do I mean by that? I am talking about a DSL that is not textual, but rather uses shapes and lines in order to express intent.

A good example of that would be UML, which you are most likley already familiar with. Figure 1.2 display a small part of a typical UML diagram. UML is a DSL for describing software systems. In fact, 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 – UML diagram displaying the customer object



There has been also a lot of effort invested in making it possible to write your own graphical DSL. 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 off of them.

There are quite a few examples of Graphical DSL that you have most probably heard about:

- UML
- BizTalk
- Sql Server Integration Services
- Workflow Foundation

I don't like graphical DSL. To be rather more exact, I simply adore graphical DSL for *documentation*, but I find that they make a rather poor job when it comes to actual development when it comes to real world scenarios.

I have had some experience with all of the above, and I can say with a great degree of confidence that they all share common problems, and that those are inherit to the graphical DSL model.

For a start, any problem of any real complexity would be very hard to build using a graphical DSL. Just arranging all the shapes in a way that you can browse them easily takes a significant amount of time, and then you get to the issue of actually understanding what this thing does.

The whole point of a graphical DSL is to hide information that you don't want to see, to let you see "big picture". All of which means that you can't really see the whole at the same time, and this leads to a lot of time spent jumping between various elements on the DSL surface, trying to gather all the required data. Graphical DSL are visually verbose, they need a lot of screen estate to express notion that takes just a few lines using a textual DSL.

And then there are UI issues that are important. How do I do a search & replace operation in a graphical DSL? How can I just look for something? And, of course, there is a reason why mouse driven development is not a good idea, if only for your wrists.

Beyond that, there are serious issues with how do you make a reasonable diff between two versions. No, reading XML files diff (assuming they are even diffable) don't work, that defeats the whole purpose of using a graphical DSL in the first place. You need some way to express a diff in a graphical way, and so far I haven't seen any good way to do that.

This makes typical Graphical DSL a problem in terms of source control. This is a huge problem as far as I am concern, and I have run into exactly this issue in more than one case, with no easy or good solutions in sight.

From an implementation perspective, there are other issues that you need to consider. In a graphical DSL, the need to express things visually is obviously pretty important, so we need to have some conventions about what shapes and connections we have. The actual what and how will depend on the actual language that you are trying to implement.

If you haven't guessed so far, I am not a fan of graphical DSL. Not for programming, at least, it is a valuable tool for documentation and very high level design, but quite a few people are pushing it too hard and attempt to make it the be-all-end-all programming tool.

Personally, I think that this is resurgence of the CASE Tools and Executable UML from a few years ago. It didn't really work that well then, and I don't have a lot of expectation that it would end up significantly different this time around.

1.3.3 Fluent Interfaces

I have had some doubts about including fluent interfaces in the list of possible DSL. This is because I think of fluent interfaces as a degenerate internal DSL for languages with little syntactic flexibility.

Fluent interfaces are a way to structure your API in such a fashion that operations flow in a natural manner. It is easier to demonstrate than explain, so check listing 1.5:

Listing 1.5 – 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 very simple, the `Rotate()` method appears on listing 1.6:

Listing 1.6 – 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, this is just method chaining, but this has big implications on the readability of the operations, as well as the instructive nature that can result. With some careful thought about the nature of the return values, we can get a good language, high readability *and* the support of intellisense, to aid in the writing of the language statements.

But the above fluent interface is not really impressive, but listing 1.7 should be. This is *valid C#* code, and I am going to assume that even if you never used it in the past, it is probably going to be instantly readable:

Listing 1.7 – Using a fluent interface for querying

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

And this gives you all the users from London that registered in the last 3 months. It is also one of the easiest ways to perform queries with support from intellisense. Unfortunately, this fluent interface is based on code generation, operator overloading, and

generics abuse; it was *very* hard to create and is not something that I would want to do again.

Listing 1.8 is another example would be the fluent interface to configure StructureMap, an IoC container:

Listing 1.8 – Configuring the StructureMap container using Fluent Interface

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

And the last example in listing 1.9 is for specifying regular expressions in a way that would hopefully prevent them from being a write only:

Listing 1.9 – Using fluent interface to create a regular expression that match strings similar to <div class="game">

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

Fluent interfaces are very useful, and with C# 3 and VB.Net 9, there are even more options. Obviously the query syntax above can be replaced by a Linq query, but the other examples are still very much relevant, and with the new capabilities like extensions methods and lambdas, you can take them quite a long way.

Unfortunately, you usually can't take them far enough.

Why do I say that? Mainly because the mainstream languages are simply too limited in their syntax flexibility to allow you enough freedom to express yourself as appropriate. This is especially true when you want a DSL that you can show a business user or give to an IT person, where neither one is a programmer by trade.

And this leads us directly toward the last item on our list, Internal DSL.

1.3.4 Internal / Embedded DSL

Internal DSL 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 would make sense to both the author and the reader, not to the compiler.

Obviously, that limits an internal DSL expressiveness to the limits allowed by the 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* can build a good DSL with C++, but it

would probably include macros galore, and I am not placing bets on how maintainable it can be.

Creating Internal DSL with C# 3.0

By the way, I think that this is true for C# 3 as well. The changes to the language would certainly enable richer expressiveness, but the syntax baggage is still *quite* big, and having to make the compiler happy is a burden that I would rather avoid in a DSL.

Extension methods and lambda expression will certainly help, but they will not change the fundamental syntax too much. For all intents and purposes, C# isn't a good language to create internal DSL in.

The popular choices for building internal DSL are dynamic languages, Lisp and Smalltalk were probably the first where this was a common place occurrence. Today, people mostly use Ruby, Python, and Boo.

Why do people turn to those languages for building DSL? The main reason is that those languages have quite a bit of syntactic flexibility. This means that listing 1.10 is valid Ruby code, for example:

Listing 1.10 – A rake build script, specifying tasks to run at build time

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

It is part of a build script written using Rake⁴. Rake is a good example of using a DSL to express intent in a manner that is more immediately understood. Consider the amount of XML that you need to write using NAnt to do the same thing, and now consider how readable that would be.

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

The major advantage of an internal DSL is that it takes all the power of the language that it is written for. You *don't* have to write the semantics of the if statement, or redefine

⁴ Rake – Ruby Make, a build tool that uses a Ruby based DSL to specify actions to take during the build process
<http://rake.rubyforge.org/>

operator precedence, for instance. Of course, sometimes that *is* useful, and in one of my DSL implementation I *did* redefined the if statement, but that is probably not a good thing to do on general principal.

One thing we need to be aware of is that fluent interfaces and internal DSL are much the same thing, both of them use the capabilities of the language to try to define a more readable approach for a problem, the main difference is that fluent interfaces do not (cannot, in most cases) go far enough to earn the distinction of a DSL.

Figure 1.3 and figure 1.4 shows the main differences between the two.

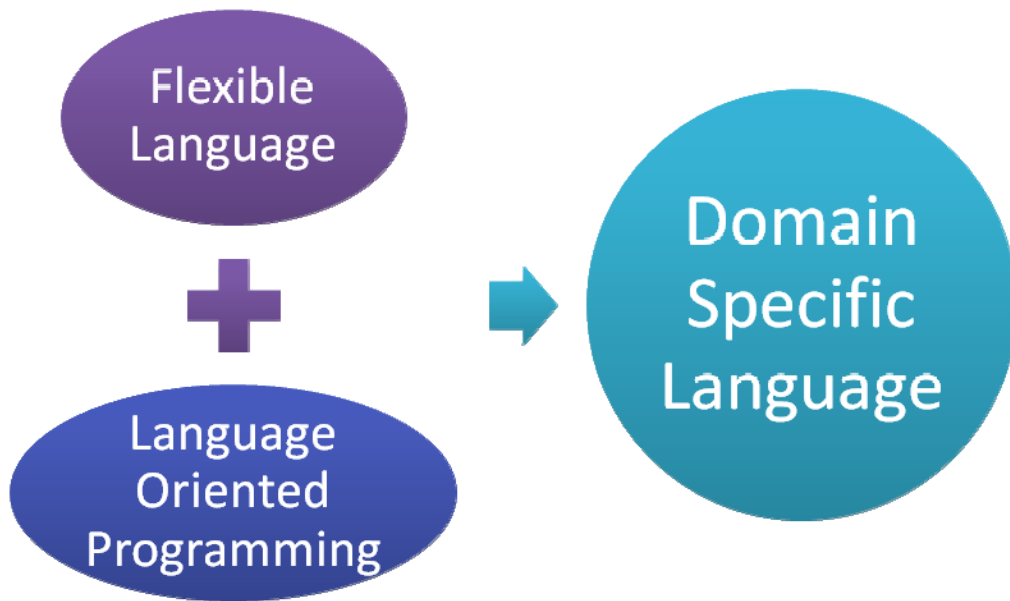


Figure 1.3 – Mixing flexibility with language orientation gives a domain specific language.

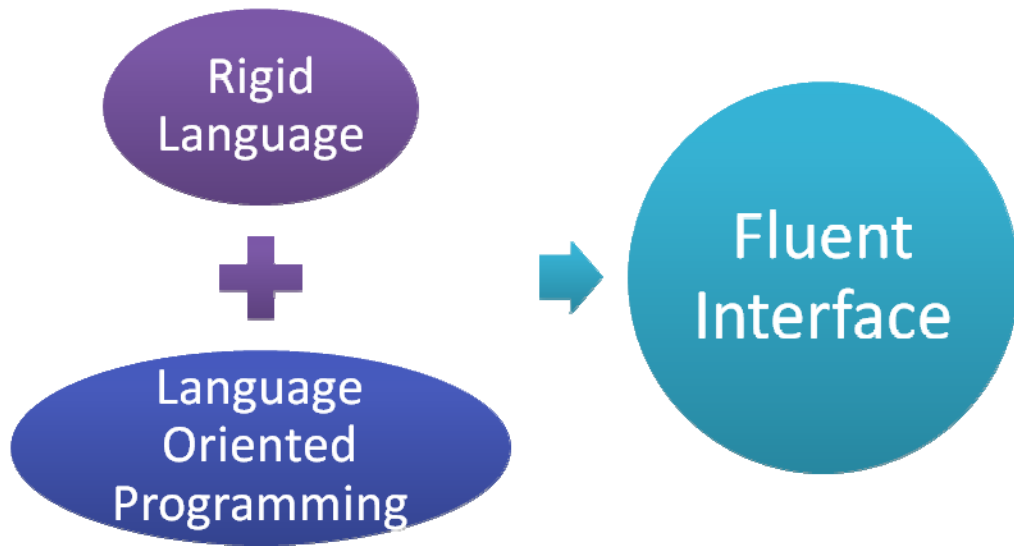


Figure 1.4 – Mixing rigid language with language orientation gives a fluent interface

A DSL should be readable for someone who is familiar with the *domain*, not the programming language. I am going to assume that you have, in the past, seen such programs that were perfectly in line with the stated code style policy, but where... not quite clear with regard to what the code actually does.

A DSL built on top of an existing language can also be problematic, since you *want* to limit the options of the language, in order to make it clearer in what is going on, rather than turn the DSL into a fully fledged programming language; we already have that in the base language, after all.

The main purpose of an internal DSL is to reduce the amount of stuff that you need there to make the compiler happy, and increase the clarity of the code in question. That is the syntactic aspect of it, at least. The other purpose, of course, it to expose the domain, which is another matter entirely, that task is going to take quite a bit of this book.

It is also a task that is far more important is mere syntax; this is the core reason why you have built the DSL in the first place. Or is it? Why do we need DSL again?

1.4 Why do we want to write a DSL

So far, I have talked about some examples of DSL, from SQL to Regular Expressions to UML, and made the (thus far) unsubstantiated claim that it is not really hard to build a DSL.

What I did not do was explain why you need a DSL at all. After all, since you are reading this book, you already know how to program. Can't we just use "normal" programming languages to do the same job? We can even do with a dash of fluent interfaces and Domain Driven Design to make the code easier to read.

In fact, I so far spoke entirely on the *how*, which is probably bad of me, since this entire book revolves around the idea that *how* shouldn't matter. We need to inspect the different needs that lead the drive toward a DSL.

There are several reasons that might lead you toward wanting a domain specific language. Those are usually driven by the need that you have. The most common ones are:

- Making a technical issue or task simpler.
- Expressing rules and actions in a way that is close to the domain and understandable to business people.
- Automating tasks and actions, scriptability and extensibility

We will go in detail over each of those scenarios and examine the forces that drive us toward wanting to utilize a DSL in this scenario and the implications that this have on the language that we will build.

1.4.1 Technical driven DSL

A technical DSL is supposed to be consumed by someone that understands development. It is meant to express matters in a more concise form, but it is still very much a programming language at heart. The main difference is that it is a language focused on solving the specific problem at hand.

As such, it has all the benefits (and drawbacks) of single purpose languages. Examples of technical DSL include Rake, Binsor⁵, Rhino ETL⁶, etc.

The driving force around building a technical DSL is that you want to have richer expressiveness in specifying what you want to happen. Technical DSL are usually easier to write, because your target audience already understands programming.

In fact, the use of programming features can make a DSL very sweet indeed. We have already seen a Rake sample, so let us see the Binsor sample in listing 1.11:

⁵ Binsor – A domain specific language for defining dependencies for the Windsor Inversion of Control Container.

<http://www.ayende.com/Blog/category/451.aspx>

⁶ Rhino ETL – A DSL based ETL tool

<http://www.ayende.com/Blog/category/545.aspx>

Listing 1.11 – A Binsor script for registering all controllers in an assembly.

```
for type in Assembly.Load("MyApplication.Web").GetTypes():  
    continue unless type.IsAssignableFrom(Controller)  
    component type.FullName, type
```

So we take three lines to register all the controllers in the application. That is quite a bit of expressiveness. It is also a sweet merge between the use of standard operations (for, `Assembly.Load`) and the DSL syntax (component).

Of course, it assumes that you are familiar with the API and its usage, which is not always true, in such situations; you need to provide a far richer environment in your DSL. Usually, we call this type of DSL a Business driven DSL.

1.4.2 Business driven DSL

A business DSL is focused on being (at least) readable to a businessperson with no background in programming.

This type of DSL is mainly expressive in the terms of the domain, and it has a lot less emphasis on the programming features that may still exist. It also tends to be much more declarative than technical DSL in general and a lot more emphasis is placed on the nature of the DSL so the programming feature would not be necessary in most cases.

I can't really think of a good example of a business DSL in the open. There are business rule engines, admittedly, but I wouldn't really call them DSL. They are usually very rigid, at least in my experience.

A good example of a business DSL that I have run into includes a cellular company that had to have *some* way to handle all the myriad of different contracts and benefits that it had. It also needed to handle this with a fairly short time to market, since the company needed 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 this:

Listing 1.12 – A DSL for specifying benefits in a cellular company

```
when call_minutes_in_current_month > 300 and [CA]  
    has_benefit "300 Minutes Free!!!":  
    add_call_minutes -300, "300 Minutes Free!!!"
```

This still looks very much like a programming language, yes. But a business person could read (and write) this very easily. We will see more complex examples in the rest of the book, right now, let us keep this simple.

It was fairly simple to define a small language that could describe most of the types of benefits that the company wanted to express. The rest was a matter of naming conventions and dropping files in a specified folder, to be picked up and processed at regular intervals. We will discuss the structure of the engine that surrounds the DSL itself further along in the book.

A businessperson may not always be able to write actions using a business DSL (more on that later), but they should be able to *read* and *understand* it. After all, it is their business and their domain that you are trying to describe.

Now, why shouldn't a businessperson be able to write actions using a business DSL?

One of the main reasons, as I see it, is of error handling. No, I don't mean in the actual running of the DSL action, I mean when *writing* it. (The other reasons has to do with knowledge and the law of unintended consequences, see section 1.6 for a more detailed discussion on the subject)

A DSL is supposed to be read like a language, but it is still a programming language, and those have little tolerance for such thing as bad casing for keywords, for instance. Certain types of users will simply be unable to go over the first hurdle they face.

It is important to know your audience, and it is even more important not to be contemptuous toward that mythical businessperson. You may not think that this person can understand programming, only to discover that there are quite a bit of automation going on in their life already, powered by VBScript and excel macros.

If you can leverage this knowledge, you have a very powerful combination in your hand, because you can provide that businessperson the tools, and he can provide the knowledge and the required perspective.

1.4.3 Automatic / extensibility driven DSL

I am not quite sure about this classification, but it certainly has its place. Another name for this may be the IT DSL. This type of a DSL it often used to expose the internal of an application to the outside world.

Modern games are usually engines that are being configured using some sort of a scripting language. In fact, I fondly remember building levels in Neverwinters Nights using some variation of JavaScript.

More serious usage for this can certainly be found, such as a DSL that lets you go into the internals of an application and mange it. Think about the ability to run a script that will re-route 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 is possible to do this with shell scripts of various kinds, but most enterprise application can certainly have more visibility into them than already exist, and a DSL that will allow me to inspect and modify the internal state would be very welcome.

I can certainly think of a few administrators who would be grateful to have more power in their hands.

Another way to look at this is to consider all the VBA enabled applications out there. Those move from Office to Autocad to accounting packages and ERP systems. The VBA extensibility enables the users to create scripts that access the state of the system. The same thing can be done using automation DSL (probably for far less in licensing costs).

1.5 Why Boo – Boo’s Domain Specific Language Capabilities

So far, I mentioned Lisp, Smalltalk, Ruby, Python and Boo as languages that are well suited for writing internal DSL.

So, why does this book focus on Boo?

I dislike the term [language/platform] programmer, but I’ll readily admit that most of the time, I am working on software based on the .Net Common Language Runtime. This is a common, stable platform⁷, with a rich set of tools and practices. I have a strong desire to keep to this platform, because I already know most of the quirks and how to smooth them⁸.

As such, I would like to keep my DSL implementation within my preferred platform. This allows me to utilize the aforementioned knowledge to troubleshoot most problems. It also means that I have very little problem when calling a DSL from my code, or vice versa, both the DSL and my code are CLR assemblies, and as such, interoperate cleanly.

Figure 1.5 shows how an application that makes use of several DSL as an integral part of the application itself. Those DSL can access the application logic easily and natively, while the application can just shell out to the DSL for those decisions that require significant amount of flexibility.

⁷ As long as you don’t start to play with Reflection Emit and generics, *there are dragons* in that territory comes to mind at that stage.

⁸ Except, of course, the occasional `ExecutionEngineException` and `InvalidProgramException`, usually this happens right after I tried to work with Reflection Emit and generics. Maybe that is the reason I feel that there are dragons there.



Figure 1.5 – DSL used as an integral part of the application

As much as I like the CLR, the common languages for it are... not well suited for language oriented programming. I believe that I mentioned the term syntactic baggage, but my term for those languages is simpler, rigid. Rigid languages don't offer a whole lot of options to express concepts. You have the default language syntax and that is it.

Boo is a CLR language, with a default syntax that is very much like Python, with some interesting opinions about compiler architecture. Being a CLR language, this means that it will compile down to IL, which is then interoperable with any other .NET code and perform as fast as any other IL based language. The Python syntax doesn't have a lot of baggage (except that indentation thingy, which Boo has a solution for), but that is not what makes Boo special.

What makes Boo special is the fact that the compiler is open. Not open in terms of "you can look at the code", unless you are writing compilers, this kind of open is rarely a useful thing. It is open means that you can, very easily, just go ahead and change the compiler object model *while it is compiling your code*.

As you can imagine, this has some significant implications on your ability to use Boo for language oriented programming. In effect, Boo will let you modify the language itself to fit your needs. I am going to spend most of Chapters 3 and 4 explaining the how and the why in detail. I am sure that you'll be suitably impressed.

What about IronRuby and IronPython, then? Those are CLR implementations of the languages that are already proven to be suited for building DSL.

Well, at the time of writing, neither of those has a stable release out. I have been using Boo for production for over two years now, and it has proven itself well in the field.

But, more importantly, and at the risk of Ninja on Rails attacks. I find that while both Ruby and Python has a lot to offer for language oriented programming, but Boo simply has more to offer.

There is also the question of performance, Boo is (usually) a statically typed language, and those typically significantly outperform most dynamic languages. I have great hopes for the Dynamic Runtime Language, but I have no data so far about the performance characteristics that it is going to have, so I'll remain skeptic until faced with new evidence.

Beyond that, Boo also contains some additional interesting features for language oriented programming. Among them, meta-methods, quasi quotation, AST Macros and AST Attributes.

As I mentioned, we will explore them in full in chapters 3 and 4.

1.6 DSL Anti Pattern - Throwing a DSL over the wall

I am putting this in here as a warning, since it is something that I have heard many times. There seem to be a very common approach to programming and DSL in general, one that dreams of having the developers build the foundation, and then simply handing the tools to the business user and washing our hands from all that nasty business stuff.

This kind of thinking seems to be a recurring theme, from the dreams of the secretary building applications by dragging boxes to the business analyst modifying the workflows. We have seen this happening with CASE tools, and then the 4th generation languages, later with component oriented programming, CORBA, etc, etc.

I disagree with this quite strongly. Giving a user this type of tool is an irresponsible step at best. The users are not developers, and they *will* make mistakes. And the first time that the business will lose money over it, it will be *your* fault.

- It will be your fault when the server is on 100% CPU because the user has created endless loop.
- It will be your fault that Mike from accounting made "just a tiny modification" that suddenly means that you sold a product with for 50% less than you paid for it.

- It will be your fault when the server is down because of stack overflow.
- It will be your fault that Joe just got a paycheck proudly stating -43.9.
- It will be your fault that the company bought the entire fish market for the next three years.
- It will be your fault that no one has any idea what the company's own business processes are.
- It will be your fault that the database contains inconsistent data.
- It will be your fault that there is no control over it.
- It will be your fault that there hasn't been an audit log for the last three months.
- It will be your fault, *period*.

Handing a user a tool like that is wonderful; it is a tool that can enable better communication and rapid response between the IT and the business. A DSL means that you are talking in the same language; that you are less likely to mess up because of translation problems.

It does not mean that you can just throw it over the wall and walk away. A DSL is still code, and like all code, it requires proper handling. A partial list of what this entail includes:

- Code review
- QA
- Change management and source control
- Testing – unit & integration
- Intention revealing semantics
- Performance testing
- Auditing
- Error handling
- Transaction handling

Professional developers are having problems with handling that, giving it to inexperienced users is a mistake. It might work for at first, but this is not a sustainable approach.

Consider the typical departmental application, written by a guy with a Learn Programming in 10 Hours book and Good Intentions. Now consider the usual of quality of such an application. Do you truly intend to bring that, *knowingly*, into your codebase?

To put it in terms that should make it easy for an IT person to understand, would you really let a user of unknown quality access to your database, including the ability to perform potential destructive operations?

The answer for that is obviously not⁹, so why are you willing to give Joe Random Business User the same amount of access to the code?

Please note that I am not talking about malicious users here, but simple honest mistakes that can result in significant problems. But this leads to another question, if I am so strongly opposing to the idea of handing a DSL to the user and be done with it, why am I writing this book in the first place?

All the advantages that I have outlined so far are quite real, but I want to warn against trying to take too much. You want the users to work through their scenarios with the development team so it can validate the total effects of business changes and the technological changes.

Back to the databases example, many places requires a DBA approval before pushing any change to production. The reasons for that are well understood and are considered best practice¹⁰ approach.

You should consider DSL in the same manner. It is a great tool that you can use to help your business and the end users, but at the same time, a tool that needs all the usual checks and balances that normal code goes through.

1.7 Examining Domain Specific Languages Examples

Before we conclude this chapter, I wanted to give you a bit of a taste about what kind of Domain Specific Languages you can create. This is by no mean a conclusive list, and I fully expect that you, dear reader, will surprise me with both ingenuity and innovation, but it is a start.

All the examples were written using Boo, which should give you some idea about the flexibility of the language.

1.7.1 Brail – Text Templating Language

Brail would probably remind you very strongly of classic ASP or PHP.

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

⁹ If it isn't, you are probably needs to consider the IT security procedures.

¹⁰ Albeit, I'll admit they can be *annoying* to the developer

```

    <li>${element}</li>
<% end %>
</ul>

```

1.7.2 Rhino ETL – ETL tools based on a Boo DSL

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

```

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

```

1.7.3 Boo Build System

Think about it this way, NAnt, without all that annoying XML, Boo Build System takes a lot of concepts from Rake as well.

```

Task "init build dir":
    Mkdir "build"
    Cp FileSet("lib/*.dll").Files, "build", true

```

1.7.4 Specter

Specter is a Behavior Driven Design testing framework, using it make Boo behave in a way that is far more natural to the way BDD specifies.

```

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

```

Those examples are just to whet your appetite; those are actually on the shallow end of the transformations that you can put Boo code through.

1.8 Summary

By now, you should have developed an understanding of what is a Domain Specific Language. It used to be that the cost and return of investment for creating a DSL only justified themselves for the really big problems, but the tools have grown much better, and this book will help you understand how you can create a non trivial language in a rainy afternoon.

Of all the DSL types presented so far, I most strongly favor internal DSL, for the simplicity, extensibility and low cost, compare to the others approaches. Fluent interfaces are also a good solution, but it often frustrates me that I need to bow to the limit of the languages and find a less clear way to express something when I am creating a fluent interface.

I have a strong bias against graphical DSL, for their visual verboseness and complexity of use and more importantly, for the complete mess they make out of source control. A graphical DSL doesn't really lend itself for diffing and merging, and those are critical in any team scenario. I have had quite a bit of personal pain as a result of just that.

So it is a happy coincidence that this book is not here to talk about graphical DSL, but about internal (or embedded) DSL written in Boo, isn't it?

We are going to explore the reasons for creating a DSL and what kind of structure we need to support it in Chapter 2.

We are going to use Boo for the rest of the book, and we will dedicate Chapter 3 to understand exactly how Boo can help us create our own Domain Specific Language.

And, without further ado, let us get started building our first DSL...