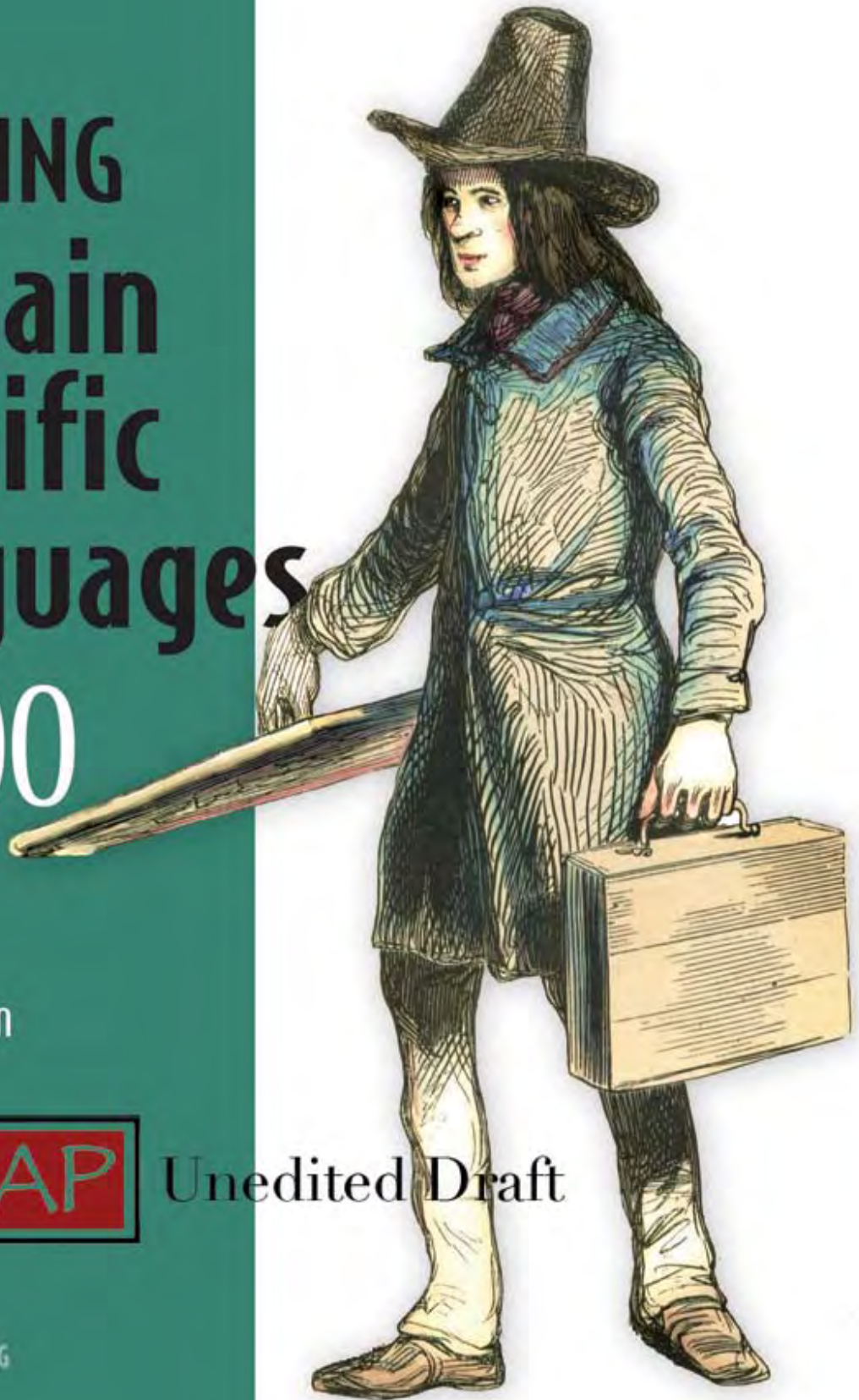


BUILDING Domain Specific Languages IN BOO

Ayende Rahien



Unedited Draft





**MEAP Edition
Manning Early Access Program**

Copyright 2009 Manning Publications

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

Contents

Preface

Chapter 1 What are Domain Specific Languages?

Chapter 2 A short overview of the Boo language

Chapter 3 The drive toward Domain Specific Languages

Chapter 4 Building Domain Specific Languages

Chapter 5 Integrating Domain Specific Languages into your applications

Chapter 6 Digging into advance compiler extensibility approaches

Chapter 7 DSL infrastructure with Rhino DSL

Chapter 8 Testing your Domain Specific Languages

Chapter 9 Versioning Domain Specific Languages

Chapter 10 Creating professional UI for our DSL

Chapter 11 DSL and documentation

Chapter 12 DSL implementation patterns

Chapter 13 Implementing a real world DSL

Appendix A Boo basic reference

Appendix B Boo language syntax

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 one of the most fundamental problems 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 was a problem, and 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:

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

```

        /* chunk is larger, shorten, and return the tail */
        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 are among a decided minority if you can take a single glance at this code and deduct immediately what it is doing. Most developers would have to *decipher* this piece of code.

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 at the assembly level (or near assembly), I have to very carefully instruct the machine on what to do, in excruciating detail.

The piece of code above was taken from FreeBSD's boot loader's malloc method. It has good reasons to look the way it does. But writing at this level has a big cost on productivity and flexibility.

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

We want to achieve the simplest, clearest way to talk to the computer. At some point, we have to leave traditional programming languages behind in order 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 purpose.

What we would like to find are approaches for achieving clear, concise and simple ways to instruct the machine in what we mean, rather than laboriously micro manage it.

1.1 Striving for simplicity

At all times, I want to produce code that is readable, maintainable and simple.

Simple code is much harder to write than complex code. Simple isn't easy. Easy is when you throw code at a problem until it goes away. Simple code is what you get when you remove all the complexity away from the code. That is not to say that writing simple code is complex, it is just noting that writing *complex* code is very easy.

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
------------------------------------	-----------------------------------

```
DateTime.Now.AddDays(14);
```

```
time() + 1209600;
```

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 BCL¹ DateTime API, however.

Using higher level concepts mean that I am concentrating less on how I want things to be done, at 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 make sense, and I usually need to build those with systems with significant size and complexity.

POLITE CODE

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, knows where you live, and has a *very* short fuse.

1.1.1 Creating Clear Code

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 have to 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, 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 in general 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 are using. Programming languages make it easy to tell the computer what it should do; they can be less effective at expressing developer intent.

¹ BCL – Base Class Library – the standard library for the .NET platform

1.1.2 Creating Intention Revealing Code

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, as shown in listing 1.2.

Listing 1.2 – Validating a phone number

```
public bool ValidatePhoneNumber(string input)
{
    if (input.Length != 11)
        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 the code and understand what input it will accept without actually *deciphering* it?

If you haven't noticed by now, I consider the need to decipher code a bad idea. It is much safer to avoid this need, 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}$");
}
```

That is still cryptic for many people, however. In this case, the usage of a specialized tool for text processing has made the intent much easier to understand, but you need to understand the tool for text processing. I can glance at the code and, assuming that I know Regular Expressions, I can figure out what input it would like to accept.

Another approach is to define a mask for certain input, which mean that our code will now be:

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

Assuming that # is the character for matching a numeric character, that is even easier to understand than the regular expression approach.

The other side of regular expressions

Regular Expressions are notorious for being a write only tool, which is somewhat true, if you don't write them carefully.

This exposes another issue; the use of a special tool to handle a 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 later in the book; most of chapter 11 is dedicated to techniques that help people get to grips with our custom languages.

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

It is far easier to send a SQL statement to the database and let it sort out how it wants to handle it on its own. We speak at a much higher level of abstraction, because we aren't concerned with the actual details.

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.

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 mean, 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 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 the focus they give us is so 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 Fowler² defines Domain Specific Languages as:

² Martin Fowler – Domain Specific Language - <http://martinfowler.com/bliki/DomainSpecificLanguage.html>

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, starboard and shiver my timbers. There is a whole host of other terms that are used by the sea faring community, specifically in order to be able to communicate clearly even in bad conditions³.

That is also why the weatherman makes such a fuss about the distinction between a breeze, a gale and a storm.

1.2.1 Striving toward better ways to express intent

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 Unix⁴, from UML to BNF, etc.

From the beginning, it was recognized that trying to express what you mean in free text is not really a viable approach. It only because the old rule about "Garbage In, Garbage Out" still applies. You needed to have a clearer, much more focused, way to express your intent. Thus we have code, which is unambiguous (well, most of the time) and easy for the computer to understand.

That led us to another problem, code may be unambiguous to a computer, but it can certainly be ambiguous to me. Understanding code became 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 the code is readable, clear and concise, we make an investment that will benefit us both in the immediate future (simpler, easier to handle software) and in the long term (easier maintainability, less chance of losing your way in a big ball of mud, clearer path for extensibility and growth).

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 category is creating a DSL to allow us to clearly and efficiently express intent and meaning in code.

³ Well, arguably shiver my timbers is not really useful for clear communication.

⁴ Awk, Sed and Grep come to mind immediately, for instance.

1.2.2 Creating our own languages

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

It isn't simple, but it is certainly possible, 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 remains 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 an if statement, operators precedence, loops and the like.

And those are the very basics, mind you. Building a single purpose language is a far easier task, because the scope is much reduced. A good example of that can be seen in rSpec, a Ruby library for creating behavior driven specification. One of its capabilities is a story runner that accepts specifications written in English⁵. I suggest looking at the way they are doing that, it is really quite ingenious in its simplicity.

The problem with that approach is that we hit its limits very quickly. And then we have to accept the additional complexity of building a more full featured language.

Recently I was consulting for a company that had built a 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.

⁵ You can find description of this capability in this URL:

<http://blog.davidchelimsky.net/articles/2007/10/21/story-runner-in-plain-english>

After looking at it for a while, it became apparent that they could have switched the whole thing to an internal DSL (a DSL that is hosted in an existing language, which we will talk about shortly) and save quite a bit of time, effort, and pain.

This is not necessarily the case, because we don't always have to write our own language from scratch. We can utilize an existing language (called the host language or base language) to provide us this facility, while we provide additional syntax and behavior on top of it.

The first popular example that I can think about is probably Ruby on Rails, 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 of 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 often distinguish between several types.

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

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

1.3.1 External DSLs

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

Building an external DSL means that we start working from a blank slate. We define the 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 an if statement works. Common tools for building external DSLs include Lex and Yacc, ANTLR, Gold Parser and Coco/R, among others, less well known.

Those tools handle the first stage, of translating text in 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 very few tools to help you there⁶.

Building a rich external DSLs is the same as building a general purpose language. You need to understand compiler theory before starting on that path. For that, I recommend reading *Compilers: Principles, Techniques, and Tools*⁷, which is a classic book on the subject.

This book doesn't cover this subject, it talks about building languages on top of an existing language, 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 we will review the process of building a language from scratch.

The grammar and syntax are often defined using a notation such as BNF or derivative. Afterward you use a 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.

An example will 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.

⁶ One such tool that comes to mind is the Dynamic Language Runtime, a Microsoft project that aims to give us dynamic languages running in .NET. One of the basic underpinning of this project is a set of classes that specify the behavior of a program (the Abstract Syntax Tree, or AST) which the DLR can turn into an executable for us. There are other such tools, for sure, but the DLR is the only one I know in the .NET space.

⁷ *Compilers: Principles, Techniques, and Tools*: By Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman , ISBN: 0321486811

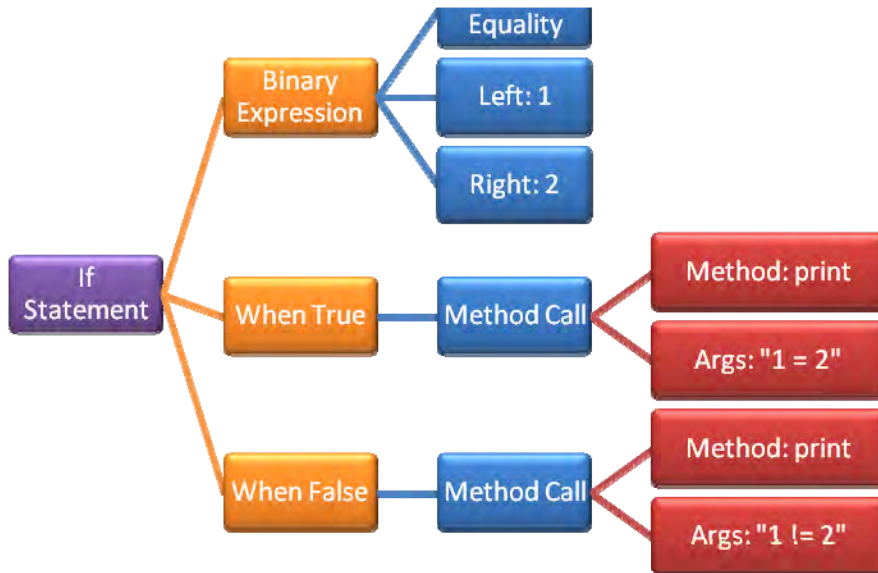


Figure 1.1 – A hierarchical representation of the AST generated from a simple if statement.

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

External DSLs are extremely powerful, but they also carry with them a cost that is not insignificant. In general, I prefer to avoid going with the external DSLs route, mainly because of the cost issue, but also because internal DSLs serve quite well in most cases. As you'll see, you can get quite a bit of flexibility by choosing the right host language, enough to satisfy most needs.

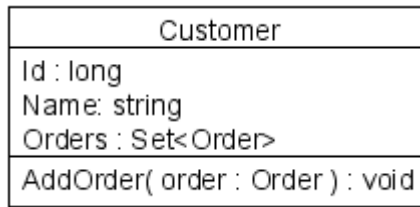
I would use an external DSL for specifying languages that are too far afield from programming languages. SQL is a good example for that. You cannot build that with an internal DSL. You can build something that is similar, but you cannot get it quite right, so an external DSL would be the right approach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

1.3.2 Graphical DSLs

Another form of domain specific language 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 likely already familiar with. Figure 1.2 displays 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 class diagram displaying the customer object



Graphical DSLs are a great way to express a lot of information in a concise way. Often enough, it is much easier to understand a problem when you see it than when it is explained. This is part of the reason whiteboards are an essential tool for developers.

The visualization approach also helps to communicate at a high level because of the physical limitations of the image. The reduction in information means that it is much easier to understand what is going on, because you can see (and the pun is fully intended) the big picture.

There has been also a lot of effort 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 off of them.

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

- UML
- BizTalk Orchestrations and Maps
- Sql Server Integration Services
- Workflow Foundation

I don't like graphical DSLs. To be rather more exact, I love graphical DSLs 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 inherent to the graphical DSLs model.

Any problem of any real complexity would be very hard to build using a graphical DSL. 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 DSLs are visually verbose; they often need a lot of screen real estate to express notions that can take 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 can work with graphical DSLs in a team environment. Most graphical DSLs persist their data into XML files. But while I can pull out a code file and compare two versions easily (called a diff, or diffing), this breaks down for graphical DSLs.

Even assuming that the XML persistence format is human readable (and I haven't seen an example where this was so), comparing the XML 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 I haven't seen any good way to do that.

This makes typical Graphical DSLs a problem in terms of source control. This is a huge issue as far as I am concern, and I have run into 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 DSLs. 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 DSLs. 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. 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.

Linq, in C# 3.0, makes this example look aged. This code was written for C# 2.0, and was used there. I am giving this example as a way to show how far you can push the syntax of a language.

Listing 1.8 is another example would be the fluent interface to configure StructureMap⁸:

Listing 1.8 – Configuring the StructureMap container using Fluent Interface

```
registry.AddInstanceOf<IWidget>()
    .WithName("DarkGreen")
```

⁸ Structure Map is an Inversion of Control container, probably the oldest on .NET. You can find more information about it here:

<http://structuremap.sourceforge.net/>

Inversion of Control container is a tool that helps you manage dependencies and lifetime of object in your application. You can think about it as a very smart factory, although that doesn't do justice for the term. You can read more about this concept here:

<http://www.martinfowler.com/articles/injection.html>

```
.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 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, we have some interesting options to express ourselves. 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 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 especially true when you want to have a way to express business requirements in a way that would make sense even to non programmers. It would be very good indeed if you could show the business people what we are doing, in a way that made sense to them.

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

1.3.4 Internal / Embedded DSL

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 would make 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 can build a good DSL with C++, but it would probably include preprocessor macros galore, and I am not placing bets on how maintainable it can be.

The popular choices for building internal DSLs 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 DSLs? 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 Rake⁹. 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 an XML based build tool, such as NAnt or MsBuild, 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 DSLs 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 an if statement, or redefine operator precedence, for instance. Of course, sometimes that is useful, and in one of my DSLs implementations I *did* redefined the if statement, but that is probably not a good thing to do on general principal.

A DSL should be readable for someone who is familiar with the domain, not the programming language.

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 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 than mere syntax; this is the core reason why you have built the DSL in the first place. Or is it? Why do we need DSLs again?

⁹ 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/>

1.4 Why do we want to write a Domain Specific Language?

So far, I have talked about some examples of DSLs, 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 somewhat hypocritical 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 and we most certainly need to under the why.

There are several reasons that might lead you toward wanting a domain specific language. Those are usually driven by the problems you want to solve. 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 has on the language that we will build.

1.5.1 Technical driven DSL

A technical DSL is supposed to be consumed by someone that understands the development environment. 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 DSLs include Rake, Binsor¹⁰, Rhino ETL¹¹, Watir¹², etc.

¹⁰ Binsor – A domain specific language for defining dependencies for the Windsor Inversion of Control Container.
<http://www.ayende.com/Blog/category/451.aspx>

¹¹ Rhino ETL – A DSL based ETL tool <http://www.ayende.com/Blog/category/545.aspx>

¹² Automation DSL for driving Internet Explorer, mostly used for integration testing.
<http://en.wikipedia.org/wiki/Watir>

The driving force around building a technical DSL is that you want to have richer ways to specify what you want to happen. Technical DSLs are usually easier to write, because your target audience already understands programming, so you have less work to do to create a language that make sense to them.

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.

Listing 1.11 – 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
```

So we take two lines to register all the controllers in the application. That is quite expressive. It is also a sweet merge between the use of the standard language operations (for loop) and the DSL syntax (component).

This works well if your target audience is developers. If they are not, however, you need to provide a far richer environment in your DSLs. 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 the very least) readable to a businessperson with no background in programming.

This type of Domain Specific Language is mainly expressive in the terms of the domain, and it has a lot less emphasis on the programming features that may still exists. It is also tend to be much more declarative than technical DSLs 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 DSLs. They are 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 run into include 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!!!":  
give_free_call_minutes 300, "300 Minutes Free!!!"
```

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

Using a business DSLs requires business knowledge

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

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

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, 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 the user's tolerance for error. No, I don't mean in the actual running of the DSL action, I mean when writing it. A syntax error, for example, can cause a business person to become completely mystified and unable to continue working.

A DSL is supposed to be read like a language, but it is still a programming language, and those have little tolerance for such things as omitting the condition in the if statement, 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 don't assume that mythical businessperson's ability to write code. While you may not think that this person can understand programming, you might very well discover that they already have quite a bit of experience in automating small tasks, using VBA and Excel macros.

If you can leverage this knowledge, you have a very powerful combination in your hands, 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 name, but it certainly has its place. Another name for this may be the IT DSLs. This type of a DSL is often used to expose the internals 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 uses for this style of DSL can certainly be found, such as a DSL that lets you go into the internals of an application and manage 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 applications have a rich internal state that could be partially made visible; certainly 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 options to manage the applications that I put in their hands.

Another way to look at this is to consider all the VBA enabled applications out there. Those range 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 for enterprise applications using an automation DSLs (at far less cost in licensing alone).

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 DSLs.

So, why does this book focus on Boo? And have you even heard about this language? Since Boo has yet to get enough of a mindshare to be a household name (but just you wait), 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 with a python inspired syntax and a special focus on language and compiler extensibility. It is this focus on language and compiler extensibility that makes it ideally suitable for the task of building our own little languages. That it is running natively on the CLR¹³ is a huge plus, since it means that all my existing toolset is still very much relevant.

¹³ The CLR – Common Language Runtime – the technical name for the .NET platform.

Boo run on Java as well, say hello to BooJay

Boo is actually not just a CLR language, it is also a JVM language. I have exactly zero experience with it in that environment, so I'll refrain from commenting on that further.

You can learn more about Boo on Java in the BooJay discussion group:

<http://groups.google.com/group/boojay/>

A good place to start seeing what BooJay is capable of is this screen cast:

http://blogs.codehaus.org/people/bamboo/archives/001751_experience_boojay_with_molipse.html

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 platform¹⁴, 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 them.

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 1.3 shows how an application that makes use of several DSL as an integral part of the application itself. Those DSLs 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.

¹⁴ 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.



Figure 1.3 – 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, able to access the entire base class library and any additional code that you have just lying around. It also means that it performs as fast as any other IL based language. You don't sacrifice performance when you are choosing to use a DSL, at least not if you go with Boo.

In fact, I can debug my DSL in Visual Studio, profile it with dotTrace¹⁵, and even reference my DSL from any .NET language (such as C# or VB.NET).

It also means that *your code* (in any .NET language that you care to name) will be able to make calls into the DSL code. In fact, this is the most common way to execute a DSL, simple call it.

What makes Boo special is the fact that the compiler is open. And 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. What I mean here is 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 Chapter 2 and Chapter 6 explaining the how and the why in detail. I hope that I will be able to show you what has made me choose Boo as the host language for my DSL efforts.

Using IronRuby or IronPython as host languages for DSLs

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

I have two major issues with them as host languages for my DSL. The first being that compared to Boo, I don't have enough control over the resulting language. The second being that both those languages run on the Dynamic Language Runtime, which is a layer on top of Common Language Runtime, which is what Boo is running on.

This means that calling into IronRuby or IronPython code from my C# code is not as simple as “add a reference and make the call”, which is all I need to do in Boo's case.

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 2 and 6. But for now, I want to show you a few real world examples of Domain Specific Languages written in Boo.

¹⁵ dotTrace is a really sweet .NET profiler from JetBrains: <http://www.jetbrains.com/profiler/>

1.6 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 in Boo. This is by no mean a conclusive list, but it should give you some idea about the flexibility of the language.

1.6.1 Brail – Text Templating Language

Brail would probably remind you very strongly of classic ASP or PHP. It is a text templating language in which in you can mix code and text freely.

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

Further reading:

<http://www.castleproject.org/monorail/documentation/trunk/viewengines/brail/index.htm>

!

1.6.2 Rhino ETL – ETL tools based on a Boo DSL

This one was built specifically after I had enough of the pain of using an ETL¹⁶ 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.

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

```
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
```

¹⁶ ETL – Extract Transform Load – a generic term for moving data around in a data warehouse. You extract the data from location A, transform that (probably with data from additional locations) and load that into its final destination. The classic example is moving data from relational database to decision support system.

In this case, get the users list from the source db, split the names, and then save them to the destination database. This is a good example of a domain specific language that requires some knowledge of the domain before you can really utilize it.

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

1.6.3 *Boo Build System*

NAnt is an XML based build system, which works for simple scenarios, but it gets very complex, very fast, when you have a build script of any complexity. The Boo Build System takes much the same conceptual approach (tasks and actions), but it is using Boo to express the tasks and actions in the script.

The result syntax tends to be easier to understand in just about any level of complexity. XML has no natural way to express conditional and loops. And you often need those in a build script, It is much easier to read when you are using a programming language to express that.

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

Further reading: <http://code.google.com/p/boo-build-system/>

1.6.4 *Specter*

Specter is a Behavior Driven Design testing framework. It allows developers to build specifications for the object under test, instead of asserting their behavior. You can read more about it here: <http://behaviour-driven.org/>

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
```

Further reading: <http://specter.sourceforge.net/>

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.

1.7 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 DSLs types presented so far, I most strongly favor internal DSLs, for the simplicity, extensibility and low cost, compared to the other approaches. Fluent interfaces are also a good solution, but it often frustrates me that I need to bow to the limits of the (rigid) host language.

I have a strong bias against graphical DSLs, for their visual verbosity, complexity of use and more importantly, for the complete mess they often make out of source control. A graphical DSL doesn't really lend itself for diffing and merging, and those are critical in any scenario that involves more than a single developer. I have had quite a bit of personal pain as a result of authors of graphical DSL not taking into account integration with a source control system as a first level feature.

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

Chapter 2 will be a short dive into Boo; just enough to get our feet wet and have enough understanding of the knowledge to start building interesting language using it.

So, without further ado, let us start on the path of understanding Boo.