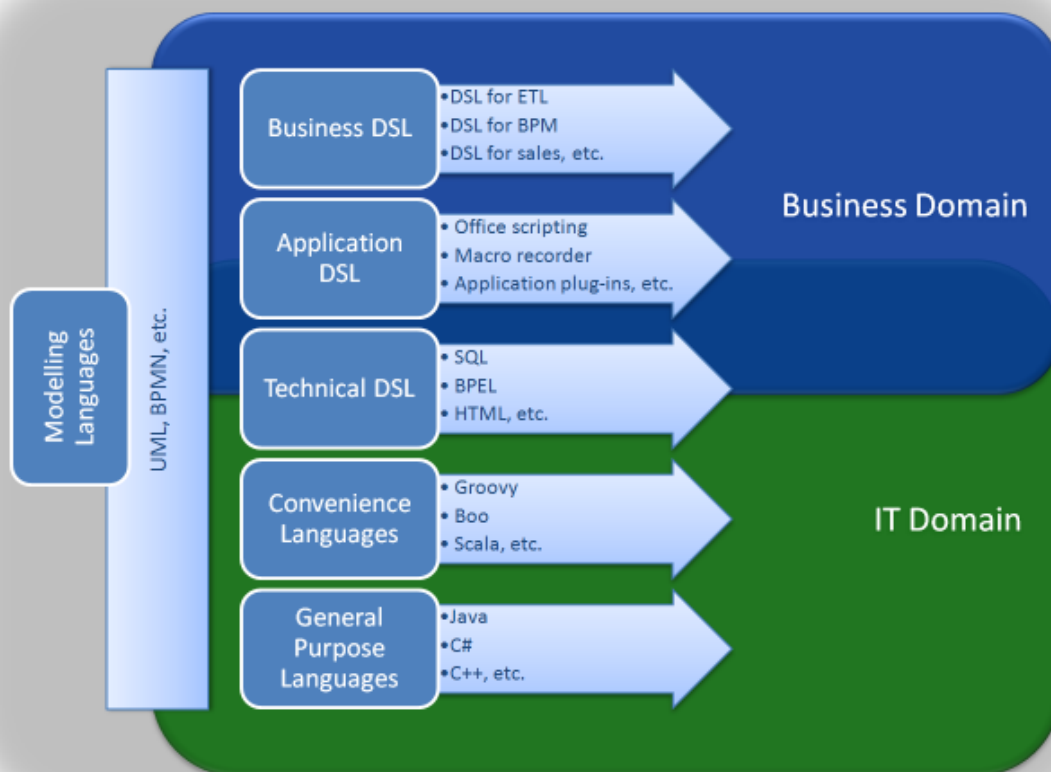


Today's business applications require a high degree of flexibility in order to keep up with frequent market changes. Technologies such as Business Process Management (BPM), Business Rules Management (BRM), Service-Oriented Architecture (SOA) and agile methodologies such as Scrum address these demands. Although the aforementioned are all great approaches, there is still a significant gap between business and IT in most cases. This is where business-oriented Domain Specific Languages (DSL) fits in.

Before we go into detail about the development, it is relevant to look at the current language landscape.



At the very bottom of the diagram are the General Purpose Languages. These are what most people would refer to as programming languages. In the last couple of years several new languages have been invented. Initially called scripting languages, they are convenience languages that make it easier to use the underlying platform such as the JVM or CLR and to add dynamic features to it. Most developers are already used to Domain Specific Languages that have a technical focus, such as SQL, HTML, BPEL, etc. Even in the business domain we can already see Application-DSLs such as office scripting or macros, although a business user would not state that he or she is a programmer. Martin Fowler refers to these people as [Lay Programmers](#). Modelling languages such as UML can be used as DSLs if they are transformed into something executable. This is the core idea of Model Driven Architecture (MDA). BPMN is especially interesting, as with version 2.0 it moves from a pure modelling language towards an executable language.

Business-DSLs target the business user. Initially this might appear to be unusual because IT people tend to think that a business user is not able to write code. The key is that the Business-DSL is

tailored to the users, so that they can manage it. Practice has shown that if the Business-DSL is developed in close collaboration with the business users, it can actually be a huge step towards Business/IT-Alignment. A practical example of what can be achieved with Domain Specific Languages is shown in the [European Patent Office Case Study](#) from SpringSource.

Business-DSLs have the potential to dramatically improve a business' involvement in IT development. They can empower business users to adapt parts of their IT systems without the help of the IT department. It must be highlighted that this has to happen in a well-defined sandbox environment where it will not cause any serious damage. A staging environment with automated testing can reduce the potential risk.

DSL development is often considered to be difficult. This is not true, particularly if DSL is based on existing technologies. These DSLs are called internal DSLs.

In this tutorial we develop an internal Domain Specific Language on the .NET platform, including an editor with syntax highlighting and auto completion. I have chosen Boo as the language due to its dynamic features and .NET integration capabilities. As an internal DSL, it can utilise the full power of the underlying platform. The effort required for implementation is reduced to a minimum.

Everything shown in this tutorial can also be achieved on the Java platform, simply replace Boo with Groovy, .NET with JEE and SharpDevelop with Eclipse. The principles are identical.

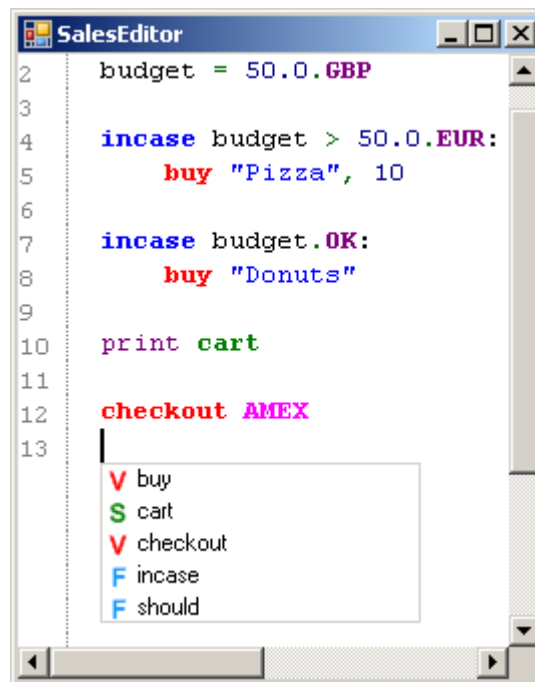
In order to run the code in this tutorial you need the following software:

1. [.NET runtime](#)
2. [Boo](#)
3. [SharpDevelop IDE](#)

Boo and SharpDevelop are both Open Source with liberal licenses MIT/BSD and LGPL. The .NET runtime is free to use.

The DSL we are going to implement is called the sales language which is used to buy products. It is kept simple intentionally, so it is possible to only focus on the technical aspects.

The final result looks like this:



Real DSLs are tailored to the business domain, so your DSL might look totally different.

THE DOMAIN MODEL

At the heart of a DSL is the domain model. It represents the objects which are manipulated by the verbs of the DSL. We implement a shopping cart which can contain arbitrary products.

```
class Product:
    [property(Name)]
    _name as string
    [property(Qty)]
    _qty as int
    def ToString():
        return _name + ":" + _qty
```

A product only has two properties: name and quantity. The sales language is implemented as a .NET class in Boo. The benefit of this is that most DSL verbs can be implemented as simple methods in the class. Likewise, our shopping cart is just a property.

```
public abstract class SalesDSL:
    _shoppingCart = []
    cart:
        get:
            return _shoppingCart
```

Standard types can be extended in order to improve the usability. In addition to the method extensions that exist in C#, in Boo, it is also possible to add property extensions, as shown in the next example.

```
[Extension]
static EUR[value as double]:
    get:
        return value * 1

[Extension]
static GBP[value as double]:
    get:
        return value * 1.18
```

Now we can write the following in our DSL:

```
incase 50.0.GBP > 50.0.EUR
    print "GPB is more than EUR"
```

THE VERBS

The DSL verbs define the behaviour of the DSL. Verbs can be implemented in three different ways.

Plain method verbs

If a verb is implemented as a simple method, it can directly be called from within the DSL.

```
final AMEX = "American Express" # Constant
def checkout(creditCard as string):
    print "Checking out using ${creditCard}"
```

The useful thing with Boo is that it is possible to omit the parenthesis. This creates a more natural feel for the DSL user, as shown below.

```
checkout AMEX
```

AMEX is just a string constant in the *SalesDSL* class. If you need block statements you can use the callable type which allows for the passing of code blocks (closures) to the method.

```
def incase(condition as bool, block as callable):
    if condition:
        block()
```

In the DSL, the method with the callable parameter can be used like this:

```
incase true:
    print "Yes"
```

AST macro verbs

AST macros allow for complex code generation at the compiler level. If a macro is called, the code inside the quasi-quotation block `[|...|]` is yielded to the compiler.

```
macro buy:
    c = 0
    name = "Unknown"
    qty = 1
    for i in buy.Arguments: # Default parameter checking
        if c == 0:
            name = cast (StringLiteralExpression,
                          buy.Arguments[c]).Value
        if c == 1:
            qty = cast (IntegerLiteralExpression,
                       buy.Arguments[c]).Value
        c++
    code = [| # Quasi-Quotation
            block:
                print "Buying " + $(name) + ":" + $(qty)
                product = Product(Name:$(name), Qty:$(qty))
                _shoppingCart.Add(product)
            |].Body
    yield code
```

This example also shows the implementation of the [Convention over Configuration paradigm](#). If the DSL user omits the parameters they are set to meaningful defaults. The user only has to specify specialties, not the standard behaviour.

```
buy "Pizza", 20 # Buy twenty
buy "Donuts"    # buy one
```

The verbs do not usually return values, as the DSL manages the state handling internally. Good DSLs shield the user from complexity as much as possible.

AST macros are the most powerful and most complex means of implementing DSL verbs.

Missing method interception verbs

By implementing the *IQuackFu* interface a class can intercept all undefined methods and properties at runtime. This allows for the implementation of generic DSL solutions in which the verbs are probably unknown during development.

```
public abstract class SalesDSL(IQuackFu):
    def QuackInvoke(name as string, ps as (object)) as object:
        print "Invoking " + name
```

THE ENGINE

At this point our language definition is ready. Instead of writing the DSL inside the *SalesDSL* class we would like to be able to integrate stand-alone scripts. In doing this, we hide the DSL implementation entirely from the user.

We can use the Boo compiler pipeline to merge the DSL script into the *SalesDSL* class at runtime. To do this we need a custom compiler step as shown below:

```
public abstract class SalesDSL(IQuackFu):
    public abstract def Start():
        pass

class BaseClassStep(AbstractTransformerCompilerStep):
    override def Run():
        super.Visit(CompileUnit)

    override def OnModule(node as Module):
        baseClass = [|
            class $(node.Name) (SalesDSL):
                public override def Start():
                    $(node.Globals)
            ]

        node.Globals = Block()

        node.Members.Add(baseClass)
```

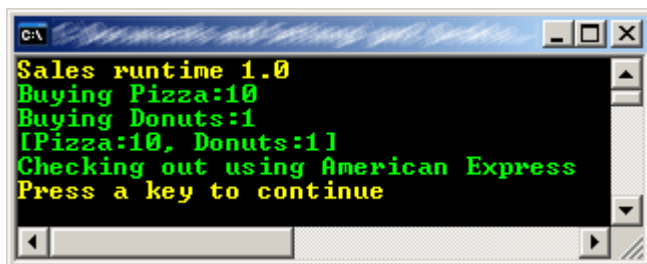
In the *OnModule* method a new class is generated using quasi-quotation. The generated class is derived from *SalesDSL*. The script code is placed in the Start-Method, which overrides the abstract definition in the *SalesDSL* class.

To start the class generation we have to hook up our *BaseClassStep* to the Boo compiler pipeline:

```
compiler = BooCompiler()
compiler.Parameters.Pipeline = CompileToMemory()
compiler.Parameters.Input.Add(FileInput("sales.sdsl"));
compiler.Parameters.Pipeline.Insert(1, BaseClassStep() )
ctx = compiler.Run()
type = ctx.GeneratedAssembly.GetType("sales")
instance as SalesDSL = Activator.CreateInstance(type)
instance.Start()
```

The result is an instance of the *SalesDSL* class with the script content merged into the Start-Method.

Calling Start executes the DSL script, resulting in the following output:



```
Sales runtime 1.0
Buying Pizza:10
Buying Donuts:1
[Pizza:10, Donuts:1]
Checking out using American Express
Press a key to continue
```

As the DSL script is compiled to 100% MSIL it is possible to call arbitrary .NET classes from within the script. To prevent dangerous actions such as *System.Environment.Exit(0)*, the DLS needs to be constrained. As we have already seen, the custom compiler step uses a different and very simple approach for this purpose.

```
def CheckConstraints(result as CompilerContext):
    temp = StringWriter()
    astobject = result.CompileUnit
    s = XmlSerializer( astobject.GetType() )
    s.Serialize( temp, astobject )
    if temp.ToString().IndexOf("Expression\ Name=\"Exit\"" ) != -1:
        raise Exception("Exit is not allowed in sales script")
```

We serialise the AST into a string and perform a simple string search operation for suspicious code.

Now we add the CheckConstraint call to our engine.

```
ctx = compiler.Run()

try:
    CheckConstraints(ctx)
except ex:
    Console.ForegroundColor = ConsoleColor.Red
    print ex.Message
```

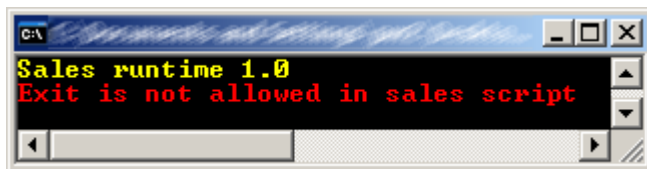
If we try to run a script like the following

```
buy "Pizza"

System.Environment.Exit(0)

Checkout AMEX
```

we will get the following output:



EXECUTION TRACKING

If the DSL runs unattended, for instance on a regular schedule, we need a means to track its execution. The tracking behaviour can be implemented using AST Attributes. These make it possible to interweave all kinds of aspects into your code. For instance logging, security or transactions, just to name a few.


```

[Module]

public class Tracking:

    public static def Track(verb as string):

        Console.ForegroundColor = ConsoleColor.Magenta
        print ("Tracking: " + verb)
        Console.ForegroundColor = ConsoleColor.Green

public class TrackAttribute(AbstractAstAttribute):

    def constructor(expr as Expression):

        pass

    def Apply(target as Node):

        type as ClassDefinition = target

        for member in type.Members:

            method = member as Method

            continue if method is null

            methodBody = method.Body

            methodName = method.Name

            if not methodName == "Start":

                method.Body = [|

                    Tracking.Track($methodName)

                    $methodBody

                |]

```

To implement such an aspect we need a class that derives from *AbstractAstAttribute*. The *Apply* method is used for the adorned element, which in our case is the class in which the AST is manipulated. In the above code the original method body is replaced with one containing the *Tracking.Track* call. *Tracking.Track* writes the DSL verb name to the console. A real DSL one would probably write the tracking data to a database to be shown in an admin console.

To enable the tracking simply adorn the *SalesDSL* class with the new Track-Attribute, as shown below.

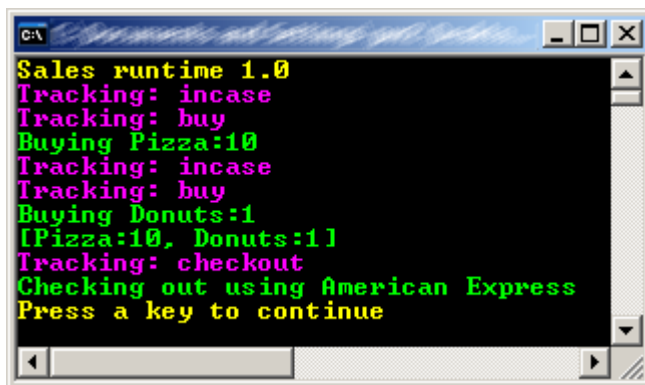
```

[Track(true)]

public abstract class SalesDSL(IQuackFu):

```

Running the tracking enabled DSL shows the following:



```
Sales runtime 1.0
Tracking: incase
Tracking: buy
Buying Pizza:10
Tracking: incase
Tracking: buy
Buying Donuts:1
[Pizza:10, Donuts:1]
Tracking: checkout
Checking out using American Express
Press a key to continue
```

THE EDITOR

An editor can be greatly simplified using the DSL. As developers, we are used to undo/redo, syntax highlighting and auto-completion. Providing these features to the DSL user makes their life much easier. However, building all these features from scratch is a lot of work. Luckily with Eclipse and SharpDevelop we have modular and free editors which are very useful.

The *TextEditorControl* from the *ICSharpCode.TextEditor* assembly of the SharpDevelop distribution is a good example. It can be incorporated in a WinForms application.

All we need to visualise our DSL script using custom syntax highlighting is shown below.

```
public partial class SalesEditor : Form
{
    public SalesEditor()
    {
        var editorControl = new TextEditorControl();
        editorControl.Dock = DockStyle.Fill;
        editorControl.Text = File.ReadAllText("sales.sdsl");
        Controls.Add(editorControl);
        HighlightingManager.Manager.
            AddSyntaxModeFileProvider(
                new FileSyntaxModeProvider(@"..\\"));
        editorControl.SetHighlighting("Sdsl");
    }
}
```

The highlighting rules are defined in the file Sdsl.xshd. Here is a snippet:

```
<KeyWords name="DSLVerbs" bold="true" italic="false" color="Red" >
  <Key word="buy"/>
  <Key word="checkout"/>
</KeyWords>
<KeyWords name="DSLFlow" bold="true" italic="false" color="Blue" >
  <Key word="incase"/>
</KeyWords>
```

Implementing auto-completion requires a bit more work. It is based on the interface *ICompletionDataProvider*.

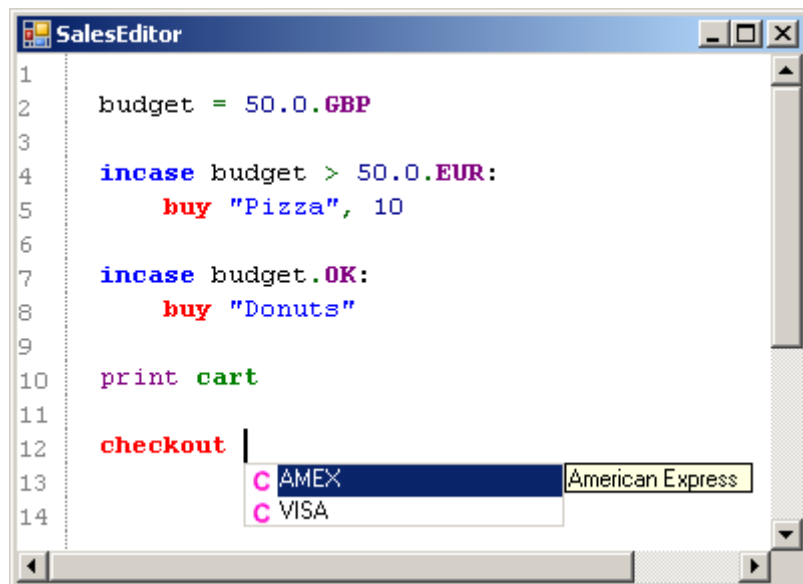
A very basic, non-context aware implementation could look like this:

```
public class SalesCompletionProvider : ICompletionDataProvider
{
    public ICompletionData[] GenerateCompletionData(
        string fileName, ICSharpCode.TextEditor.TextArea textArea,
        char charTyped)
    {
        return new ICompletionData[]
        {
            new DefaultCompletionData("buy", "Buy goods", 0),
            new DefaultCompletionData("cart", "The cart", 3),
            new DefaultCompletionData("checkout", "Checkout", 0)
        };
    }
    ...
}
```

The SharpDevelop *CodeCompletionWindow* can be used to show the suggestions.

```
var textArea = editorControl.ActiveTextAreaControl.TextArea;
textArea.KeyDown += delegate(object sender, KeyEventArgs e)
{
    if (e.KeyCode != Keys.Space)
        return;
    e.SuppressKeyPress = true;
    CodeCompletionWindow.ShowCompletionWindow(this,
    editorControl, "",
    new SalesCompletionProvider(), ((char) e.KeyValue));
};
```

We register a *KeyDown* event handler with the text area of the *TextEditorControl*. The handler shows the completion window in case the space key is pressed. The end result looks like this:



A context aware implementation of *ICompletionDataProvider* can be found in the project attached at the end of this tutorial.

SUMMARY

DSLs are potentially very useful, particularly with regards to bringing IT and business closer together. As this tutorial has shown, it is not too difficult to create your own DSL. The challenge though is to find the appropriate abstraction for your particular business domain. It is strongly recommended that DSLs are designed using an iterative approach. Close collaboration of Business and IT is the key to success.

This article explained the principles of internal Domain Specific Languages and provided a prototypical implementation based on Boo and .NET. The source code can be found here:

[salesdsl.zip](#)

About the author:

Wolfgang Pleus is an independent IT consultant in the area of service-oriented architecture (SOA) and Business Process Management (BPM). His primary interest is the implementation of SOA and BPM concepts at the application-, architecture- and strategy level and is driven by the question of how to realise agility with state of the art technologies. He has been supporting mission critical enterprise projects for more than 15 years. As an author, speaker and trainer he regularly shares his experience nationally and internationally.

Contact: wolfgang.pleus@pleus.net