NZ .NET

THOROUGHLY WATERPROOF

CODE CAMP

Domain Specific Languages

With a side order of Boo

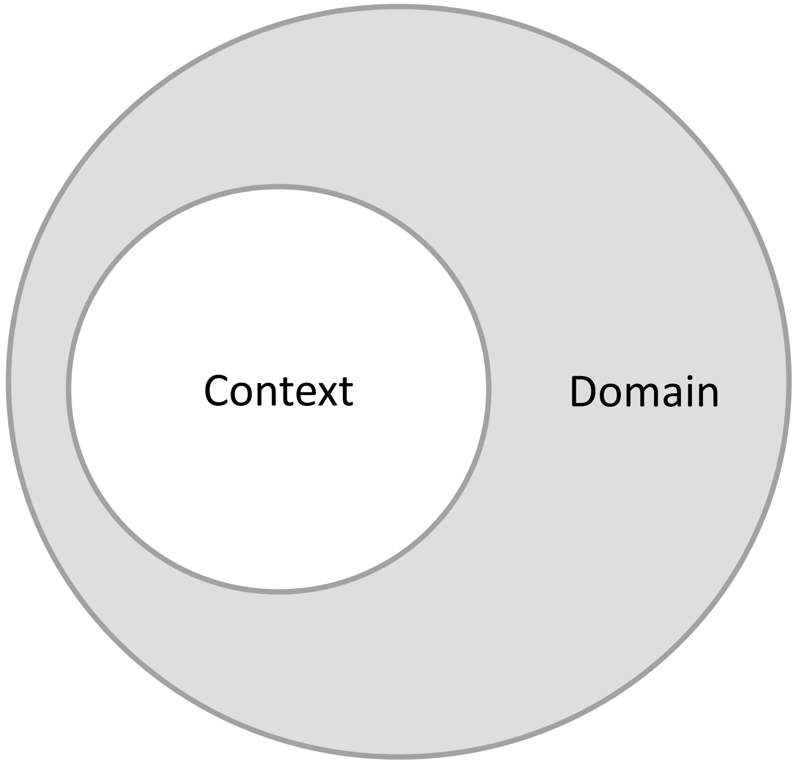
A little about Me…

- Passionate Developer
- Auckland Architecture Chat Organiser
- Sometimes Blogger
- Work/Play at DevDefined Limited

Agenda

- The context and the usual issues we face.
- Some real life examples of domain specific languages.
- Boo's DSL Capabilities.
- Integrating a DSL into your Application.
- Considerations to remember when designing your own DSL (If we get time)

Domain & Context



Who has the knowledge?

- Subject matter experts
- Business Analysts
- Customers
- Clients
- Everyone… But the developer.

What are the problems?

- Our languages are too **general purpose** – too many ways to solve the same problem.
- We don't share a **common metaphor** of understanding between developers and subject matter experts.
- Our **domain experts can't help** with the design of the business logic of an application.
- Business code is **full of boilerplate** and needless ceremony.
- Our business logic is **not cleanly separated** from our application code.
- Business rules **tied to the release cycle** of the application.

DSL: A Possible Solution

- DSL : a Domain-Specific Language.

- From Wikipedia: “***A Domain-Specific Language is a programming language designed to be useful for a specific set of tasks***”

- A DSL is a language that models a certain domain of experience, knowledge or expertise – and those concepts are tied to the constructs of the language.

How to Spot A DSL

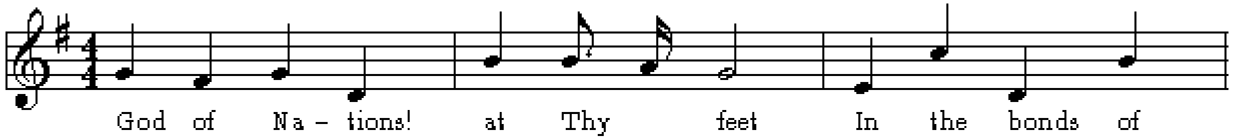
- Quite possibly not **Turing complete**.
- It covers a **particular domain** of knowledge (not general purpose).
- Has a **form**: textual or graphical.
- Produces a **result**:
  - Could represent data, configure objects, calculate discounts etc.
- Can be internal or external.
- Has certain attributes emphasized over others – the ility's ;o)
  - Readability, usability, testability.

Examples of DSL's - Technical

- Regular Expressions
  - **\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b**
- SQL Statements
  - **SELECT * FROM CodeCampSpeakers WHERE FirstName Like "Alex%"**
- Build Engines
  - **<Target Name="BuildAll" DependsOnTargets="Clean;Compile" />**
- Cascading Style Sheets
  - **#left_col { float: left; }**

Examples of DSL's - Notation

- Notations – obviously pre-dating computer programming itself...
  - Chess Moves:
    - 1.e4 e5 2.Nf3 Nf6
  - Knitting:
    - 1st Row: K1, * k1, k2together, yfwd, k1, yfwd, ssk, k2; rep from * to end. 2nd and every alt Row: Purl.
  - Music:



Business

- Insurance – **Policy structure, Risk Calculation, rating etc.**
- Telecommunications – **Call plans, specials, routing calls.**
- Local Government – **Development Contribution Rules, Asset Management, especially for Water Services.**
- Customer Relationship Management – **Rules for calculating lead/opportunity rankings, event handling and workflow.**
- Human resources – **skill set evaluation / rankings etc.**
- E-Commerce applications - **discount and promotional rules.**
- Simulations: **Anti Malaria drug resistance simulations, Artificial Intelligence Simulations such as *Noble Ape*.**
- Products Customisation – **any time you plan to sell the same piece of software to many companies.**

Categories of DSL - External

- External DSL
  - External to your application language.
  - Normally defined by a Grammar i.e. BNF, and converted to AST using a tool like ANTLR.
  - Checkout Oslo's M-Grammar and Quadrant, or Eclipses XText for the java space. Funnily enough both of these are DSL's for defining DSL's ☺
  - Generally more complex to implement and integrate.
  - More syntax freedom.
  - Requires more specialised skills.

Categories of DSL - Internal

- Internal DSL (also known as Embedded)
  - Embedded into an existing general purpose language.
  - Not all languages are really suitable for creating internal DSL's.
  - Generally less complex / requires less specialised skills to write.
  - Less Syntax freedom
  - You get things like warning and error support , and existing rich AST and compilation for free.

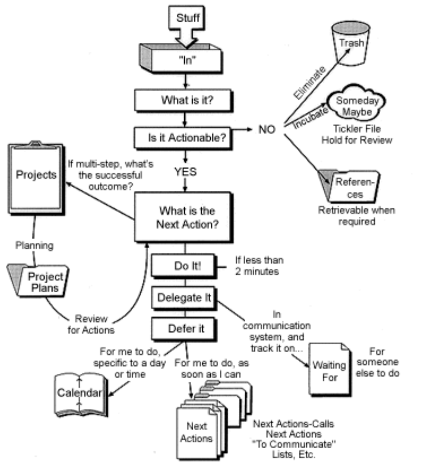
Categories of DSL - Fluent

- I'm loathed to call them DSL's (at best it's an Internal DSL Technique).
- Uses method chaining, operator overloading and abuse of properties and lambdas to achieve code that reads like a sentence.
- Can improve readability, unfortunately shortly past that point you fall off a cliff....

```
Pattern findGamesPattern = Pattern.With
  .Literal(@"<div")
  .WhiteSpace.Repeat.ZeroOrMore
  .Literal(@"class=""game""")
  .WhiteSpace.Repeat.ZeroOrMore
  .Literal(@"id=""")
  .NamedGroup("id", Pattern.With.Digit.Repeat.OneOrMore)
  .Literal(@"-game""")
  .NamedGroup("txt", Pattern.With.Anything.Repeat.Lazy.ZeroOrMore)
  .Literal(@"<!--gameStatus")
  .WhiteSpace.Repeat.ZeroOrMore
  .Literal("=")
  .WhiteSpace.Repeat.ZeroOrMore
  .NamedGroup("state", Pattern.With.Digit.Repeat.OneOrMore)
  .Literal("-->");
```

Categories of DSL - Graphical

- Examples
  - SSIS
  - Workflows
  - Process Management, BPMN
  - UML
  - BizTalk



- Good for high level views, difficult to convey details or cross-cutting concerns.
- Source control and merging changes can be an issue.
- Testing can be a problem.
- Easier to get business buy in (wow factor).

Getting More Technical

Enough Theory....
let’s take a look at the language.

What Is Boo?

- Boo is a .Net Language
- First Appeared Around 2003
- Entirely Open Source
- Python Inspired Syntax
- Very extensible

Why Boo Is Good For DSL's?

- Extensible Compiler and Language
- Syntax (optionally) lacks ceremony
- Syntax Reads well to humans
- Statically Typed
- Compiles Down to Assemblies

... Let's look at some examples...

Basics – Implicit Typing

- Implicit Typing
  - Boo doesn’t need a “var” keyword, by default all variables are implicitly typed, as well as return values from methods etc.
  - Semi-colons are optional

C#

```
var a = 10;
var b = 20;
var c = a + b;
```

Boo

```
a = 10
b = 20
c = a + b
```

Basics – Boolean Operators

- Boolean operators
  - Boo has English words for Boolean operators and more options when it comes to if etc. Statements.

C#

```
if (!(user.IsAdmin || user.InRole("Editor")))
{
  throw new Exception(“Error");
}
```

Boo

```
raise “Error!" unless user.IsAdmin or user.InRole("Editor")
```

Basics – Optional Parentheses

- Calling methods
  - When calling methods, you can drop the parentheses...
  - Also, Boo has some handy literals, i.e. Timespan literals etc. Which makes this even more concise.

C#

```
AddReminder(new TimeSpan(5, 0, 0),
     ”About time you posted to your blog”);
```

Boo

```
AddReminder 5d, ”About time you posted to your blog"
```

Basics – Anonymous Blocks

- Given a method that takes a last parameter as a delegate i.e.
  - Public void OnState(string state, Action action)
- Boo will let you supply that parameter implicitly as a block...

C#

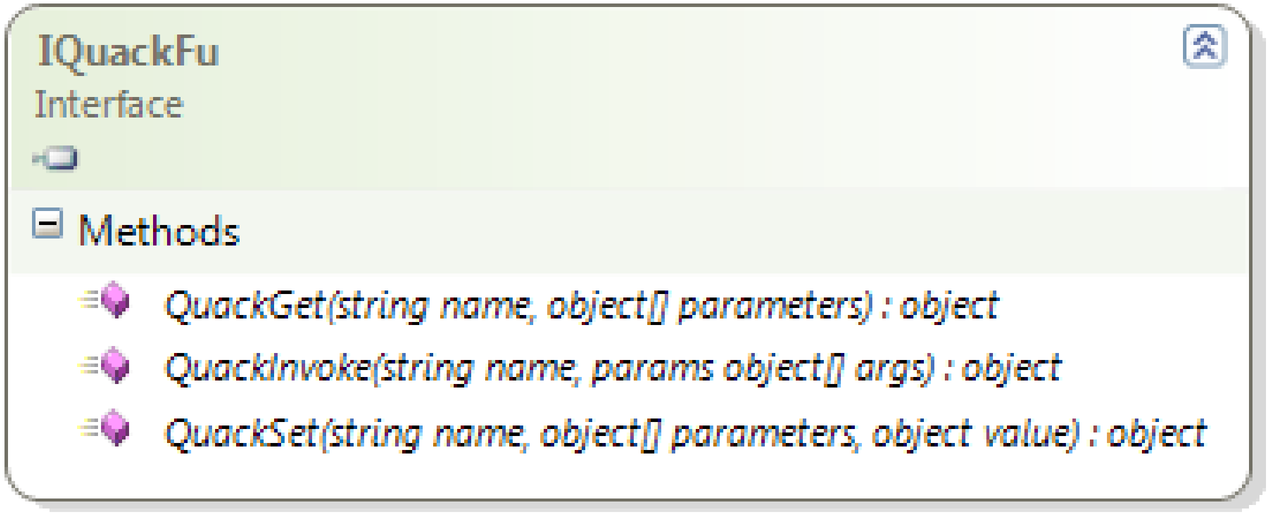
```
OnState("paid", ()=>
                  {
                    ship_goods();
                    mail_receipt();
                  });
```

Boo

```
OnState "paid”:
    mail_receipt
    ship_goods
```

Basics – Duck Typing

- Boo is a static language, like C#, but it does support duck typing – similar to **dynamic** in c# 4.0.
- To support duck typing, your class needs to implement the **IQuackFu** interface.



IQuackFu
Interface

Methods

QuackGet(string name, object[] parameters) : object
QuackInvoke(string name, params object[] args) : object
QuackSet(string name, object[] parameters, object value) : object

Basics – Duck Typing

- Duck typing lets us work against data etc. As if it were methods or properties... Such that if customer implements IQuackFu, and does not have a property called “Gold” then:

```
if customer.Gold:
       apply_discount 15.percent

Becomes...

if customer.QuackGet("Gold", null):
   apply_discount 15.percent
```

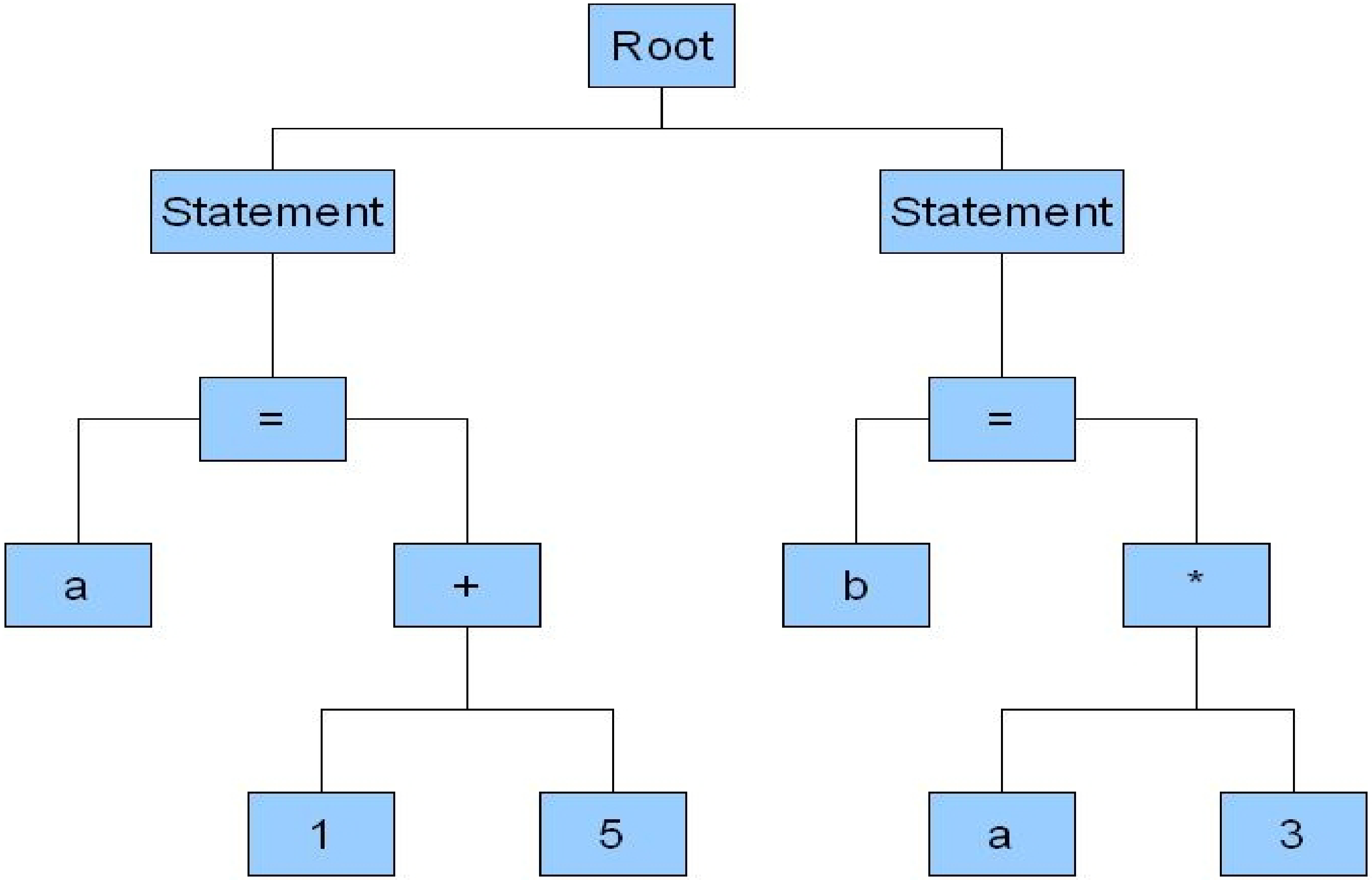
- At this point our code can handle the QuackGet request in our customer class, perhaps returning true after a database lookup.

Trickier Stuff

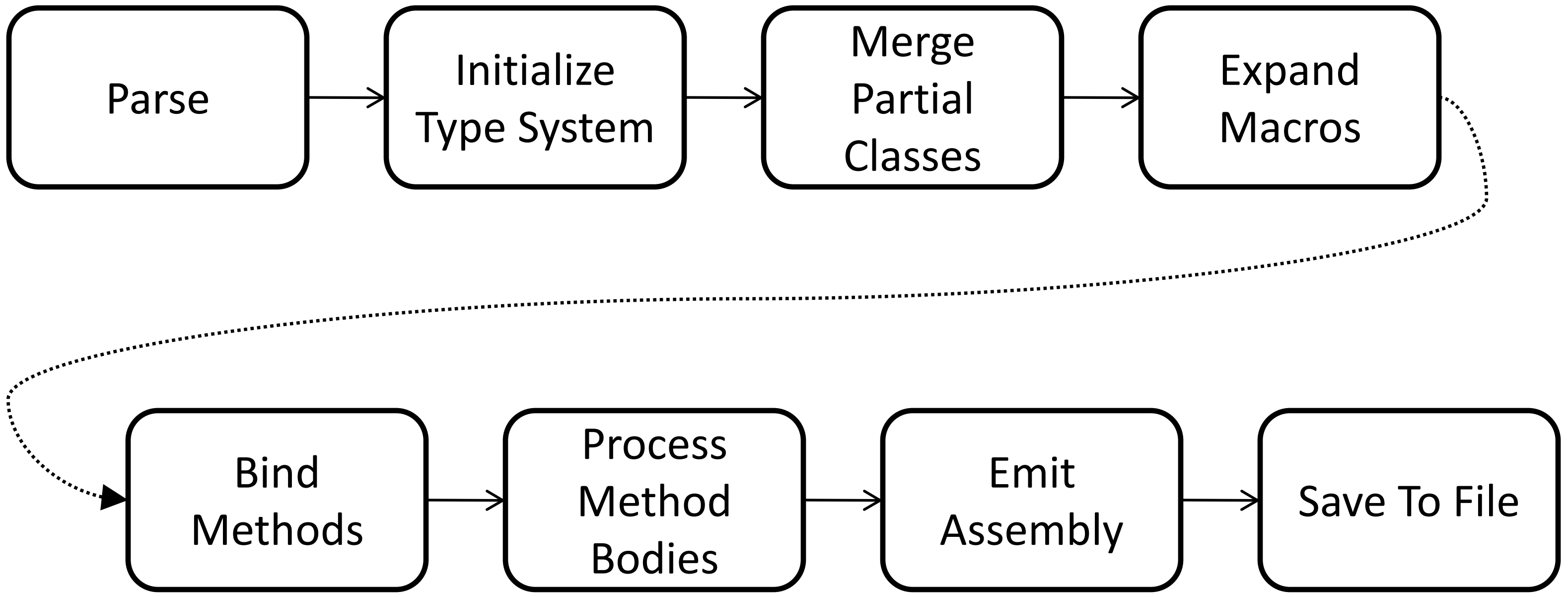
- You should hopefully now have a good taste for the language.
- **However** – Boo offers a lot more than just that to the DSL Author – one of it's claims to fame is **extensibility** – There are 4 key ways to extend Boo:
  - **Meta Methods**
  - **AST Macros**
  - **AST Attributes**
  - **Adding a new step to the compiler pipeline**
- However they really are all means to the same end, so we won't dive to deep on them (and we don't have the time) – but let us instead look at the concepts behind them and how Boo works.

What is an AST?

- AST: Abstract Syntax Tree



Compiler Pipeline (Partial)



The Full Pipeline

- And here are all the classes in the pipeline...

Boo.Lang.Parser.BooParsingStep
InitializeTypeSystemServices
PreErrorChecking
MergePartialClasses
InitializeNameResolutionService
IntroduceGlobalNamespaces
TransformCallableDefinitions
BindTypeDefinitions
BindGenericParameters
BindNamespaces
BindBaseTypes
MacroAndAttributeExpansion
ExpandAstLiterals
IntroduceModuleClasses
NormalizeStatementModifiers
NormalizeTypeAndMemberDefinitions
NormalizeOmittedExpressions
BindTypeDefinitions
BindGenericParameters
BindEnumMembers

BindBaseTypes
CheckMemberTypes
BindMethods
ResolveTypeReferences
BindTypeMembers
CheckGenericConstraints
ProcessInheritedAbstractMembers
CheckMemberNames
ProcessMethodBodiesWithDuckTyping
PreProcessExtensionMethods
ConstantFolding
NormalizeLiterals
OptimizeIterationStatements
BranchChecking
CheckIdentifiers
StricterErrorChecking
CheckAttributesUsage
ExpandDuckTypedExpression
ProcessAssignmentsToValueTypeMembers
ExpandProperties

RemoveDeadCode
CheckMembersProtectionLevel
NormalizeIterationStatements
ProcessSharedLocals
ProcessClosures
ProcessGenerators
ExpandVarArgsMethodInvocations
InjectCallableConversions
ImplementICallableOnCallableDefinitions
CheckNeverUsedMembers
EmitAssembly
SaveAssembly

Example – Meta Method

```
import Boo.Lang.Compiler.MetaProgramming
import Boo.Lang.Compiler.Ast

[meta]
def assert(condition as Expression):
  return [|
    if not $condition:
      raise AssertException($(condition.ToCodeString()))
  |]

x = null

assert x is null
```

⇩

```
if not x is null:
  raise AssertException("x is null")
```

Example – Compiler Step

- Perhaps in our DSL we want to make it clear what is a string and what is say an application role... so
  - Instead of this: **belongs_to “Administrator”, “Editor”**
  - We’d like this: **belongs_to @administrator, @editor**

```
public class ResolveSymbolsStep : AbstractTransformerCompilerStep
{
  public override void Run()
  {
    Visit(CompileUnit);
  }

  public override void OnReferenceExpression(ReferenceExpression node)
  {
    if (node.Name.StartsWith("@") == false) return;
    var replacement = new StringLiteralExpression(node.Name.Substring(1));
    ReplaceCurrentNode(replacement);
  }
}
```

Bringing it all together

- Now let’s bring it all together – by examining a simple DSL and how it integrates into an application.
- Rather then re-invent the wheel, we’re going to take a look at an example from Ayende’s great book “Writing Domain Specific Languages With Boo”.

Quote Generator - Demo

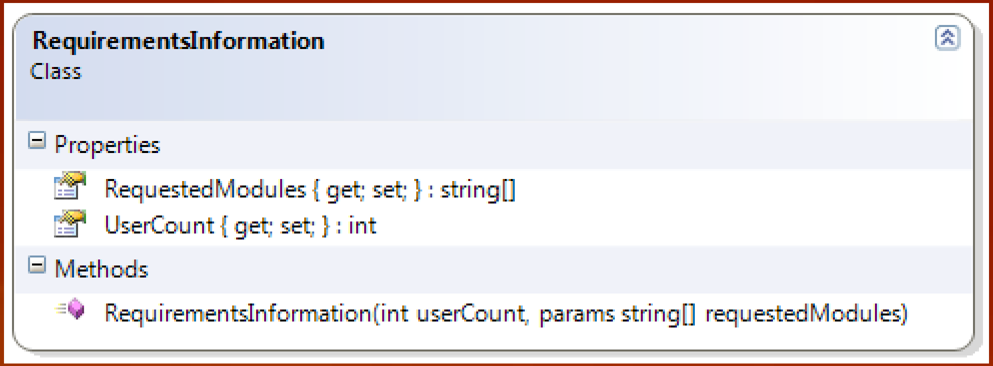
- This code show a hypothetical quote generator.
- Takes in a list of module names for a large App (think of something like SAP) and a total number of users.
- Can return a list of the modules and associated details that will be required (and handles dependencies between modules).
- App can then print a report based on the returned list of modules...

Let's start with a very brief demo then review...

The DSL Syntax

```
specification @vacations:
    requires @scheduling_work
    requires @external_connections

specification @scheduling_work:
    users_per_machine 100
    min_memory 4096
    min_cpu_count 2

specification @salary:
    users_per_machine 150

specification @taxes:
    users_per_machine 50

specification @pension:
    if total_users < 1000:
        same_machine_as @health_insurance
```

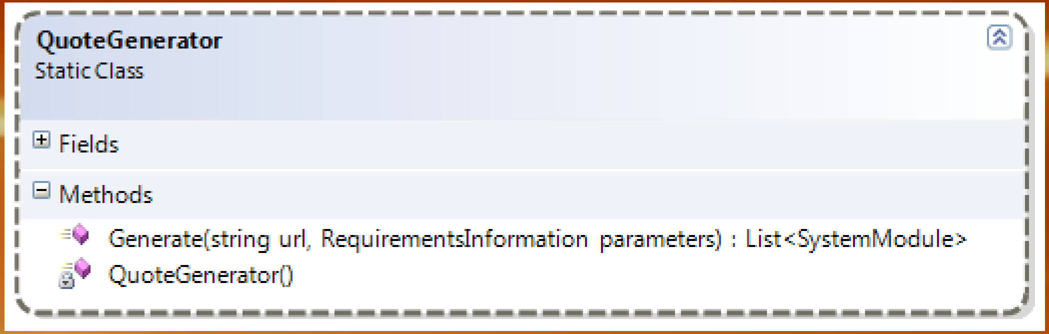
Overview



RequirementsInformation
Class

Properties
- RequestedModules { get; set; } : string[]
- UserCount { get; set; } : int

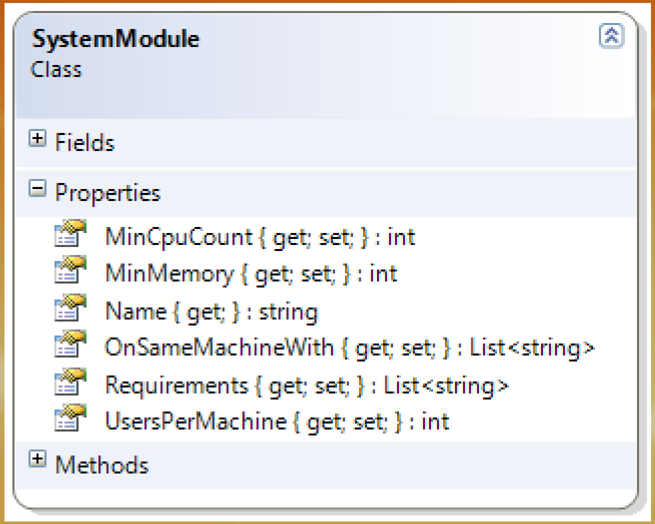
Methods
- RequirementsInformation(int userCount, params string[] requestedModules)



QuoteGenerator
Static Class

Fields

Methods
- Generate(string url, RequirementsInformation parameters) : List<SystemModule>
- QuoteGenerator()



SystemModule
Class

Fields

Properties
- MinCpuCount { get; set; } : int
- MinMemory { get; set; } : int
- Name { get; } : string
- OnSameMachineWith { get; set; } : List<string>
- Requirements { get; set; } : List<string>
- UsersPerMachine { get; set; } : int

Methods

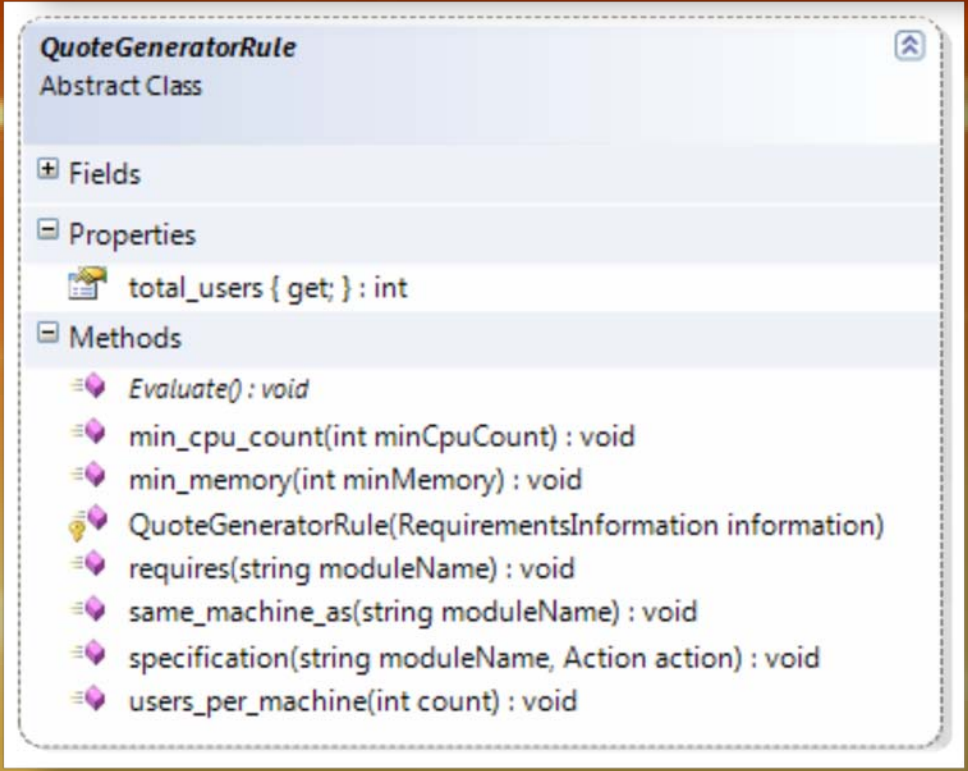
The Quote Generator

```
public static class QuoteGenerator
{
  private static readonly DslFactory dslFactory;

  public static List<SystemModule> Generate(
    string url, RequirementsInformation parameters)
  {
    QuoteGeneratorRule rule =
      dslFactory.Create<QuoteGeneratorRule>(url, parameters);
    rule.Evaluate();
    return rule.Modules;
  }
}
```

HMmmm…. What's this?

Implicit Base Class

```
specification @pension:
 if total_users < 1000:
   same_machine_as @health_insurance
```


QuoteGeneratorRule
Abstract Class

Fields

Properties

```
total_users { get; } : int
```

Methods

```
Evaluate() : void
min_cpu_count(int minCpuCount) : void
min_memory(int minMemory) : void
QuoteGeneratorRule(RequirementsInformation information)
requires(string moduleName) : void
same_machine_as(string moduleName) : void
specification(string moduleName, Action action) : void
users_per_machine(int count) : void
```

Because of a compiler step – our Boo script is compiled as a sub class of “**QuoteGeneratorRule**” with the contents of the script becoming the body of the “Evaluate” method of the class.

Notice how the method names **match up** with those we’ve used in our DSL – this is still just the CLR, No magic here.

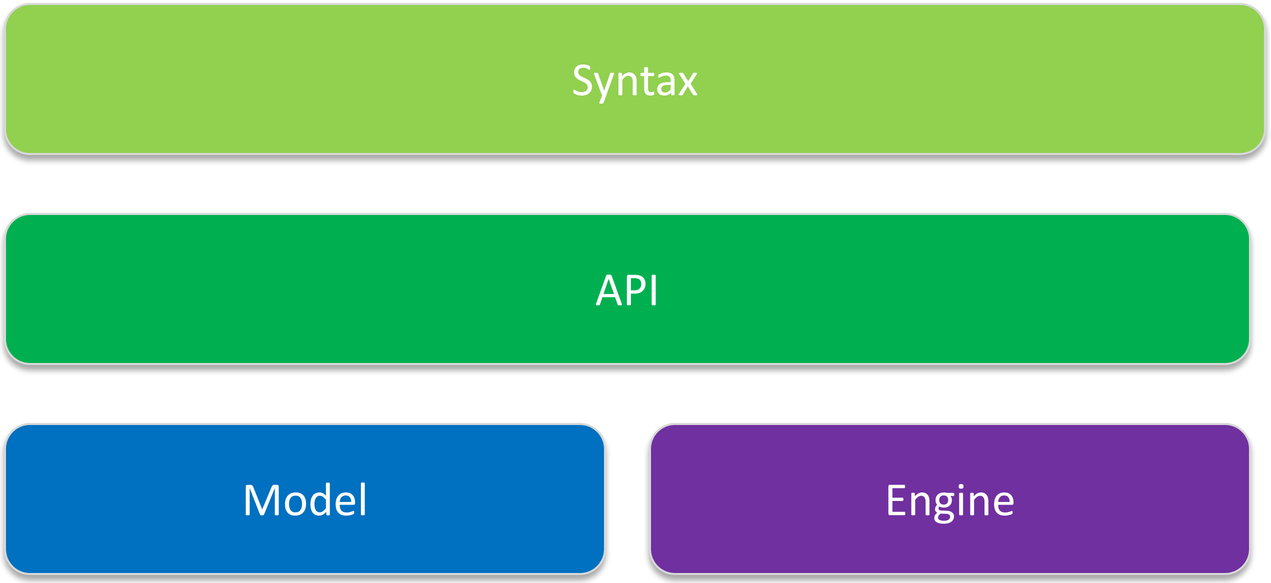
Customising the Compiler

```
public class QuoteGenerationDslEngine : DslEngine
{
  protected override void CustomizeCompiler(
    BooCompiler compiler,
    CompilerPipeline pipeline,
    string[] urls)
  {
    pipeline.Insert(1,
      new ImplicitBaseClassCompilerStep(
        typeof(QuoteGeneratorRule),
        "Evaluate",
        "BooDslExampleApp.QuoteGeneration"));

    pipeline.Insert(2, new UseSymbolsStep());
  }
}
```

Using the DSL Factory

```
public static class QuoteGenerator
{
  private static readonly DslFactory dslFactory;

  static QuoteGenerator()
  {
    dslFactory = new DslFactory();

    dslFactory.Register<QuoteGeneratorRule>(
      new QuoteGenerationDslEngine());
  }
```

Review - Typical DSL Stack



Wrapping Up

We’re almost done...

let’s quickly wrap up with some tips & thoughts.

Stay Iterative

- Never start with a DSL in mind, Iterate towards it, if it feels right.
- See how far you can go with tools like “Fluent” DSL’s.
- Ensure you have tests which test all aspects of your language, not just the expected path (i.e. Expect your users to create invalid/broken scripts).
- Make sure anything which can be achieved via DSL code can also be achieved by directly interacting with your API.

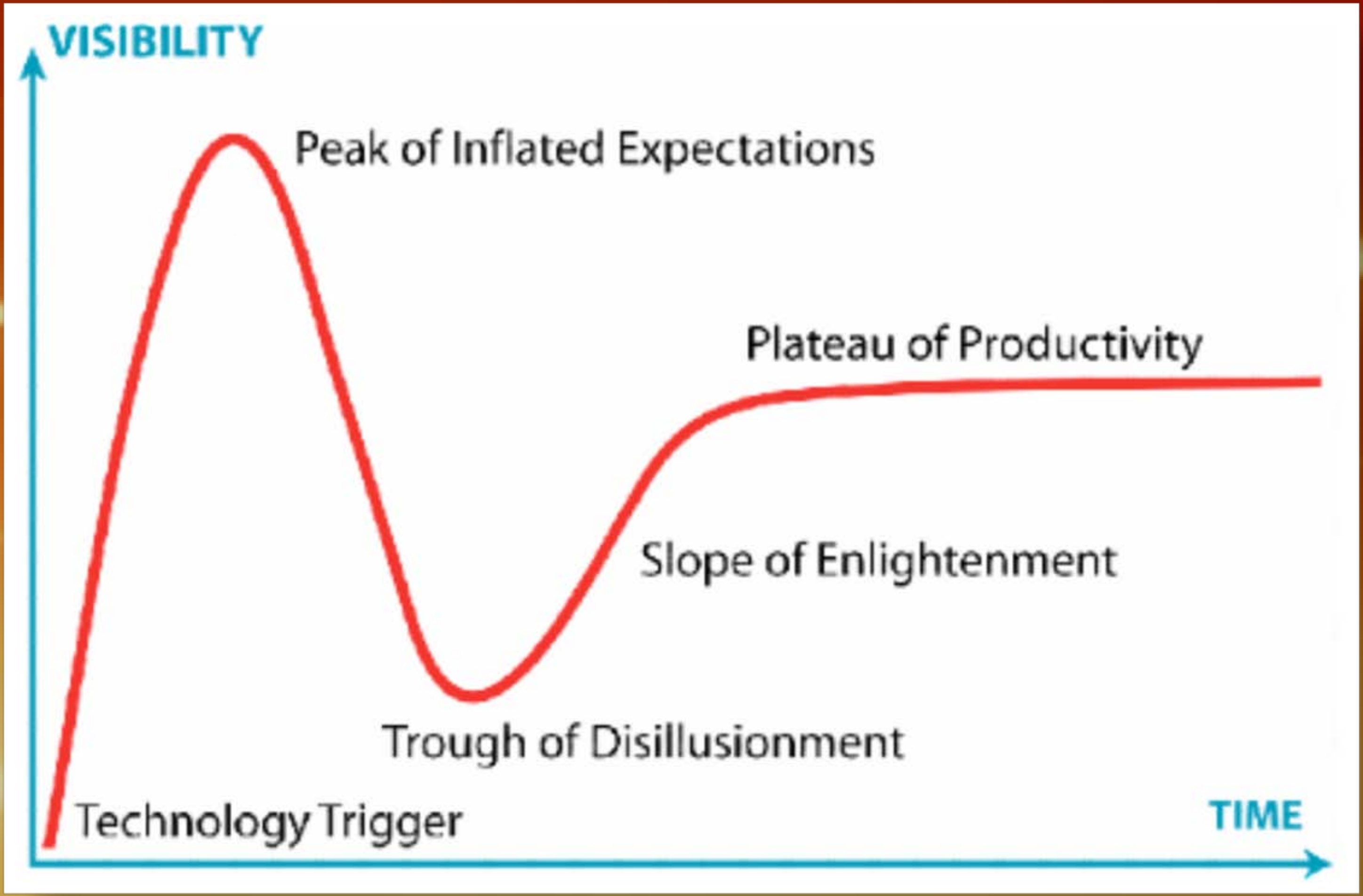
Tooling

I haven't talked about tooling yet...

- Tooling is paramount to adoption.
- Doesn't have to be an Editor With Intellisense...
- Graphical Views (Flow charts etc.)
- Reporting
- Getting simple tooling in place can be relatively easy, but complexity significantly grows with capability.

- Quick Example – if we have time.

Advocacy verses Pessimism



There's lots of think about…

- You need to think about your script lifecycle
  – If people other then developers are editing the DSL, it often wont match the development lifecycle.
- Versioning of your DSL syntax needs to be well thought out – have a strategy.
- If you want live updates, think about the impacts of:
  - Non-atomic changes.
  - Keeping a farm of servers in sync.
- Auditing and security needs to be considered.

Boo for Every Day Use?

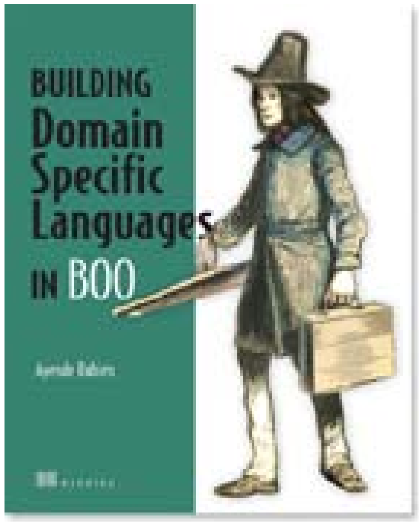
- Visual Studio Support in 2008 is Poor.
  - Hopefully this might change with VS2010.
- Reasonably good support in Sharp Develop.
- MonoDevelop 2.2 looks promising.
- Lack of mainstream tooling support a big problem (this applies to all languages other then C# and VB.Net)

Boo DSL's in the Wild

- Brails – MVC View Engine
- RhinoETL – ETL Tool
- Boo Build System – Nant like DSL
- Spectre – BDD Framework
- Binsor - Windsor Configuration
- Horn – .Net Package Manager

Resources

- "Building domain specific languages in Boo"
  - Written By Ayende Rahien (Oren Eini).
  - Very Thorough, 355 pages!
- Boo Website
- Boo Google group
- Lots of Tutorials On Line
- Source Code & Tests



Questions?

Email: alex@devdefined.com

Blog: http://blog.bittercoder.com/

Twitter: @bittercoder

Microsoft®

DATACOM

NZ .NET

THOROUGHLY WATERPROOF

CODE CAMP

ineta

INTELLIGENT BUSINESS

intergen™

xero