

# *DSLs in Boo*

*DOMAIN-SPECIFIC LANGUAGES IN .NET*

OREN EINI

WRITING AS AYENDE RAHIEN



MANNING

Greenwich  
(74° w. long.)

# *contents*

---

<i>preface</i>	<i>xv</i>
<i>acknowledgments</i>	<i>xvii</i>
<i>about this book</i>	<i>xviii</i>
<i>about the author</i>	<i>xx</i>
<i>about the cover illustration</i>	<i>xxi</i>

<i>1</i>	<i>What are domain-specific languages?</i>	<i>1</i>
1.1	Striving for simplicity	2
	Creating simple code	3
	Creating clear code	3
	Creating intention-revealing code	4
1.2	Understanding domain-specific languages	5
	Expressing intent	6
	Creating your own languages	6
1.3	Distinguishing between DSL types	7
	External DSLs	7
	Graphical DSLs	9
	Fluent interfaces	10
	Internal or embedded DSLs	12
1.4	Why write DSLs?	13
	Technical DSLs	14
	Business DSLs	14
	Automatic or extensible DSLs	16
1.5	Boo's DSL capabilities	16
1.6	Examining DSL examples	18
	Brail	18
	Rhino ETL	19
	Bake (Boo Build System)	19
	Specter	20
1.7	Summary	20

- 2 *An overview of the Boo language* 22
  - 2.1 Why use Boo? 23
  - 2.2 Exploring compiler extensibility 24
  - 2.3 Basic Boo syntax 25
  - 2.4 Boo's built-in language-oriented features 29
    - String interpolation 29
    - Is, and, not, and or 30
    - Optional parentheses 30
    - Anonymous blocks 31
    - Statement modifiers 31
    - Naming conventions 32
    - Extension methods 33
    - Extension properties 34
    - The IQuackFu interface 34
  - 2.5 Summary 37
- 3 *The drive toward DSLs* 39
  - 3.1 Choosing the DSL type to build 40
    - The difference between fluent interfaces and DSLs 40
    - Choosing between a fluent interface and a DSL 42
  - 3.2 Building different types of DSLs 43
    - Building technical DSLs 43
    - Building business DSLs 45
    - Building Extensibility DSLs 47
  - 3.3 Fleshing out the syntax 47
  - 3.4 Choosing between imperative and declarative DSLs 48
  - 3.5 Taking a DSL apart—what makes it tick? 51
  - 3.6 Combining domain-driven design and DSLs 53
    - Language-oriented programming in DDD 53
    - Applying a DSL in a DDD application 54
  - 3.7 Implementing the Scheduling DSL 56
  - 3.8 Running the Scheduling DSL 59
  - 3.9 Summary 61
- 4 *Building DSLs* 63
  - 4.1 Designing a system with DSLs 64
  - 4.2 Creating the Message-Routing DSL 65
    - Designing the Message-Routing DSL 65
  - 4.3 Creating the Authorization DSL 72
    - Exploring the Authorization DSL design 73
    - Building the Authorization DSL 76
  - 4.4 The “dark side” of using a DSL 78
  - 4.5 The Quote-Generation DSL 78
    - Building business-facing DSLs 80
    - Selecting the appropriate medium 83
  - 4.6 Summary 84

- 5 *Integrating DSLs into your applications* 86
  - 5.1 Exploring DSL integration 86
  - 5.2 Naming conventions 88
  - 5.3 Ordering the execution of scripts 91
    - Handling ordering without order 91
    - Ordering by name 92
    - Prioritizing scripts 92
    - Ordering using external configuration 94
  - 5.4 Managing reuse and dependencies 94
  - 5.5 Performance considerations when using a DSL 96
    - Script compilation 97
    - Script execution 97
    - Script management 98
    - Memory pressure 98
  - 5.6 Segregating the DSL from the application 98
    - Building your own security infrastructure 99
    - Segregating the DSL 99
    - Considerations for securing a DSL in your application 101
  - 5.7 Handling DSL errors 102
    - Handling runtime errors 102
    - Handling compilation errors 104
    - Error-handling strategies 104
  - 5.8 Administrating DSL integration 105
  - 5.9 Summary 106
- 6 *Advanced compiler extensibility approaches* 108
  - 6.1 The compiler pipeline 109
  - 6.2 Meta-methods 110
  - 6.3 Quasi-quotation 113
  - 6.4 AST macros 115
    - The unroll macro 116
    - Building macros with the MacroMacro 118
    - Analyzing the using macro 120
    - Building an SLA macro 123
    - Using nested macros 124
  - 6.5 AST attributes 126
  - 6.6 Compiler steps 128
    - Compiler structure 129
    - Building the implicit base class compiler step 130
  - 6.7 Summary 132
- 7 *DSL infrastructure with Rhino DSL* 134
  - 7.1 Understanding a DSL infrastructure 135
  - 7.2 The structure of Rhino DSL 136
    - The DslFactory 136
    - The DslEngine 137
    - Creating a custom IDslEngineStorage 139
  - 7.3 Codifying DSL idioms 143
    - ImplicitBaseClassCompilerStep 143
    - AutoReferenceFilesCompilerStep 144
    - AutoImportCompilerStep 144
    - UseSymbolsStep 144

- UnderscoreNamingConventionsToPascalCaseCompilerStep 145
- GeneratePropertyMacro 146
- 7.4 Batch compilation and compilation caches 146
- 7.5 Supplying external dependencies to our DSL 148
- 7.6 Summary 149
- 8 *Testing DSLs* 150
  - 8.1 Building testable DSLs 150
  - 8.2 Creating tests for a DSL 151
    - Testing the syntax 152
    - Testing the DSL API 155
    - Testing the DSL engine 158
  - 8.3 Testing the DSL scripts 160
    - Testing DSL scripts using standard unit testing 160
    - Creating the Testing DSL 162
  - 8.4 Integrating with a testing framework 166
  - 8.5 Taking testing further 171
    - Building an application-testing DSL 171
    - Mandatory testing 171
  - 8.6 Summary 172
- 9 *Versioning DSLs* 173
  - 9.1 Starting from a stable origin 174
  - 9.2 Planning a DSL versioning story 175
    - Implications of modifying the DSL engine 175
    - Implications of modifying the DSL API and model 176
    - Implications of modifying the DSL syntax 177
    - Implications of modifying the DSL environment 177
  - 9.3 Building a regression test suite 178
  - 9.4 Choosing a versioning strategy 179
    - Abandon-ship strategy 179
    - Single-shot strategy 179
    - Additive-change strategy 180
    - Tower of Babel strategy 181
    - Adapter strategy 182
    - The great-migration strategy 184
  - 9.5 Applying versioning strategies 185
    - Managing safe, additive changes 185
    - Handling required breaking change 187
  - 9.6 DSL versioning in the real world 190
    - Versioning Brail 190
    - Versioning Binsor 190
    - Versioning Rhino ETL 191
  - 9.7 When to version 192
  - 9.8 Summary 193

- 10 *Creating a professional UI for a DSL* 194
  - 10.1 Creating an IDE for a DSL 195
    - Using Visual Studio as your DSL IDE 196
    - Using #develop as your DSL IDE 198
  - 10.2 Integrating an IDE with a DSL application 198
    - Extending #develop highlighting for our DSLs 200
    - Adding code completion to our DSL 203
    - Adding contextual code completion support for our DSL 206
  - 10.3 Creating a graphical representation for a textual DSL 209
    - Displaying DSL execution 209
    - Creating a UI dialect 211
    - Treating code as data 212
  - 10.4 DSL code generation 216
    - The CodeDOM provider for Boo 216
    - Specific DSL writers 217
  - 10.5 Handling errors and warnings 219
  - 10.6 Summary 220
- 11 *DSLs and documentation* 221
  - 11.1 Types of documentation 222
  - 11.2 Writing the Getting Started Guide 223
    - Begin with an introduction 224
    - Provide examples 224
  - 11.3 Writing the User Guide 225
    - Explain the domain and model 225
    - Document the language syntax 227
    - Create the language reference 230
    - Explain debugging to business users 231
  - 11.4 Creating the Developer Guide 232
    - Outline the prerequisites 232
    - Explore the DSL's implementation 232
    - Document the syntax implementation 233
    - Documenting AST transformations 236
  - 11.5 Creating executable documentation 237
  - 11.6 Summary 238
- 12 *DSL implementation challenges* 239
  - 12.1 Scaling DSL usage 240
    - Technical—managing large numbers of scripts 240
    - Performing precompilation 241
    - Compiling in the background 243
    - Managing assembly leaks 243
  - 12.2 Deployment—strategies for editing DSL scripts in production 244

12.3	Ensuring system transparency	246
	Introducing transparency to the Order-Processing DSL	246
	Capturing the script filename	248
	Accessing the code at runtime	248
	Processing the AST at runtime	250
12.4	Changing runtime behavior based on AST information	251
12.5	Data mining your scripts	253
12.6	Creating DSLs that span multiple files	254
12.7	Creating DSLs that span multiple languages	256
12.8	Creating user-extensible languages	256
	The basics of user-extensible languages	256
	Creating the Business-Condition DSL	258
12.9	Summary	262
13	<i>A real-world DSL implementation</i>	263
13.1	Exploring the scenario	264
13.2	Designing the order-processing system	265
13.3	Thinking in tongues	267
13.4	Moving from an acceptable to an excellent language	269
13.5	Implementing the language	271
	Exploring the treatment of statement's implementation	273
	Implementing the upon and when keywords	274
	Tracking which file is the source of a policy	276
	Bringing it all together	276
13.6	Using the language	278
13.7	Looking beyond the code	280
	Testing our DSL	280
	Integrating with the user interface	281
	Limited DSL scope	282
13.8	Going beyond the limits of the language	282
13.9	Summary	283
	<i>appendix A Boo basic reference</i>	285
	<i>appendix B Boo language syntax</i>	302
	<i>index</i>	313

## *preface*

---

In 2007, I gave a talk about using Boo to build your own domain-specific languages (DSLs) at JAOO (<http://jaoo.dk>), a software conference in Denmark. I had been working with Boo and creating DSLs since 2005, but as I prepared for the talk, I was surprised to see just how easy it was to build DSLs with Boo. (I find that teaching something gives you a fresh perspective on it.)

That experience, and the audience's response, convinced me that you don't have to be a compiler expert or a parser wizard to build your own mini-languages. I realized that I needed to formalize the practices I had been using and make them publicly available.

One of the most challenging problems in the industry today is finding a way of clearly expressing intent in a particular domain. A lot of time and effort has been spent tackling that problem. A DSL is usually a good solution, but there is a strong perception in the community that writing your own language for a particular task is an extremely difficult task.

The truth is different from the perception. Creating a language from scratch would be a big task, but you don't need to start from scratch. Today, there are lots of tools and plenty of support for creating languages. When you decide to make an internal DSL—one that is hosted inside an existing programming language (such as Boo)—the cost of building that language drops significantly.

I routinely build new languages during presentations (onstage, within 5 or 10 minutes), because once you understand the basic principles, it is easy. Easy enough that it deserves to be a standard part of your toolset, ready to be used whenever you spot a problem that is suitable for a DSL solution.