Groovy for Domain-Specific Languages

Fergal Dearle

Chapter No.7
"Building a Builder"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "Building a Builder"

A synopsis of the book’s content

Information on where to buy this book

---

About the Author

**Fergal Dearle** is a seasoned software development professional with 23 years of experience in software product development across a wide variety of technologies. He is currently the principal consultant with his own software development consulting company, Dearle Technologies Ltd., engaged in design, development, and architecture for new software products for client companies. In the past, Fergal has worked in lead architect and developer roles for Candle Corporation on the OMEGAMON product, which is now part of IBM's Tivoli product suite, and as development manager for the Unix implementations of Lotus 1-2-3. In the early 1990s, Fergal led the team at Glockenspiel that developed CommonView, the first object-oriented UI framework for Microsoft Windows. The team was awarded one of the first ever Jolt Productivity Awards by Dr Dobbs Journal.

---

For More Information:

I would like to thank the team at Packt for their patience and forbearance during the writing of this book. With special thanks to Doug, Rajashree, Rakesh, and Shubhanjan for keeping me on the straight and narrow. I would also like to thank David Petherick, “The Digital Biographer” for introducing me to the wonders of cask strength malt Scotch whiskey and for his words of wisdom and encouragement both of which gave me the courage to take on this project in the first place. Finally, but by no means least, I want to thank my girls Caroline, Eabha, Nessa, and Sadhbh who put up with my long absences during the writing, without whose love and tolerance I would never have completed this book.

For More Information:
Groovy for Domain-Specific Languages

The Java virtual machine runs on everything from the largest mainframe to the smallest microchip and supports every conceivable application. But Java is a complex and sometimes arcane language to develop with. Groovy allows us to build targeted single-purpose mini languages, which can run directly on the JVM alongside regular Java code.

This book provides a comprehensive tutorial on designing and developing mini Groovy-based Domain-Specific Languages. It is a complete guide to the development of several mini DSLs with a lot of easy-to-understand examples. This book will help you to gain all of the skills needed to develop your own Groovy-based DSLs.

*Groovy for Domain-Specific Languages* guides the reader from the basics through to the more complex meta-programming features of Groovy. The focus is on how the Groovy language can be used to construct domain-specific mini languages. Practical examples are used throughout to demystify these seemingly complex language features and to show how they can be used to create simple and elegant DSLs. The examples include a quick and simple Groovy DSL to interface with Twitter.

The book concludes with a chapter focusing on integrating Groovy-based DSL in such a way that the scripts can be readily incorporated into the reader's own Java applications. The overall goal of this book is to take Java developers through the skills and knowledge they need to start building effective Groovy-based DSLs to integrate into their own applications.

**What This Book Covers**

Chapter 1, *Introduction to DSL and Groovy*, discusses how DSLs can be used in place of general-purpose languages to represent different parts of a system. You will see how adding DSLs to your applications can open up the development process to other stakeholders in the development process. You'll also see how, in extreme cases, the stakeholders themselves can even become co-developers of the system by using DSLs that let them represent their domain expertise in code.

Chapter 2, *Groovy Quick Start*, covers a whistle-stop tour of the Groovy language. It also touches on most of the significant features of the language as a part of this tour.

Chapter 3, *Groovy Closures*, covers closures in some depth. It covers all of the important aspects of working with closures. You can explore the various ways to call a closure and the means of passing parameters. You will see how to pass closures as parameters to methods, and how this construct can allow us to add mini DSL syntax to our code.

For More Information:

Chapter 4, *Example DSL: GeeTwitter*, focuses on how we can start with an existing Java-based API and evolve it into a simple user-friendly DSL that can be used by almost anybody. You'll learn the importance of removing boilerplate code and how you can structure our DSL in such a way that the boilerplate is invisible to our DSL users.

Chapter 5, *Power Groovy DSL Features*, covers all of the important features of the Groovy language, and looks in depth at how some of these features can be applied to developing DSLs.

Chapter 6, *Existing Groovy DSLs*, discusses some existing Groovy DSLs that are in current use and are free to download.

Chapter 7, *Building a Builder*, explains how Groovy provides two useful support classes that make it much simpler to implement our own builders than if we use the MOP. You'll see how to use `BuilderSupport` and `FactoryBuilderSupport` to create our own builder classes.

Chapter 8, *Implementing a Rules DSL*, takes a look at Groovy bindings to see how they can be used in our DSL scripts. By placing closures strategically in the binding, you can emulate named blocks of code. You can also provide built-in methods and other shorthand by including closures and named Boolean values in the binding. These techniques can be used to a great effect to write DSL scripts that can be read and understood by stakeholders outside of the programming audience.

Chapter 9, *Integrating it all*, covers the many different ways in which you can integrate Groovy code into Java. You'll explore the issues around tightly integrating the two languages at compile time. You'll see how this can lead to dependency issues arising when Java code references Groovy classes and vice versa. You'll take a look at how you can use dependency injection frameworks like Spring to resolve some of these issues.

For More Information:

Building a Builder

Builders are a powerful feature of Groovy. The Groovy libraries contain an expanding set of Builders for everything from XML and HTML markup to managing systems via JMX. Even so you will always come across circumstances where the semantics of a builder would be useful in your own application.

We've seen how to build a rudimentary builder by using the Groovy MOP and pretended methods in Chapter 5. Thankfully, the Groovy libraries provide us with easier means of developing our own builders. In this chapter, we will look at some of the ways in which we can use Groovy and the MOP to create our own builder classes.

- To begin with, we will recap the Groovy features that enable the Groovy builder pattern in order to understand how they work.
- We will look at how to build a rudimentary builder with features from the Groovy MOP.
- We will implement our own database seed data Builder by using two of the builder support classes provided in Groovy: BuilderSupport and FactoryBuilderSupport.

Builder code structure
The real beauty of Groovy's builder paradigm is the way in which it maps the naturally nested block structure of the language to the construction process. The process of defining parent-child relationships between objects through nested code blocks is well-established through other markup languages, such as XML and HTML.
The transition from writing XML or HTML to the GroovyMarkup equivalent is an easy one. To make use of a builder, we don't need to have any intimate understanding of the Groovy MOP or of how the builder is implemented. We just need to know how to write the builder code so that it conforms to the correct language semantics. The code structure of the builder pattern relies on just a few Groovy language features.

- **Closure method calls**: The distinctively nested block structure in the builder pattern is facilitated by Groovy's special handling of closures when they are passed as method parameters. This allows the closure block to be declared inline after the other method call parameters.

- **Closure method resolution**: When a method is invoked within the body of a closure and that method does not exist in the closure instance, Groovy uses a resolve strategy to determine which object (if any) should be tried to locate that method.

- **Pretended methods**: The Groovy MOP allows us to respond to method calls that do not exist in a class—in other words to "pretend" that these methods exist.

- **Named parameters**: When we pass a `map` parameter to a method, we can declare the individual map elements alongside the other method parameters, giving the effect of a named parameter list.

- **Closure delegate**: Changing the delegate of a closure allows another class to handle its method calls. When we change the delegate to a `builder` class, this allows the builder to orchestrate how the method calls are handled.

### Closure method calls

When we declare a Groovy method that accepts a closure as its last parameter, Groovy allows us to define the body of the inline closure immediately after the method call containing the other parameters. A method call followed by an inline closure block has all the appearance of being a named block of code. It's when we nest these method calls within each other that we get the distinctive builder-style code blocks.

This style of coding is not unique to builders. We can nest other method calls in the same way. In the following example, we have three methods defined within a script: `method1()`, `method2()`, and `method3()`. Nesting calls to these methods gives us some code that is very similar to a builder block, but is not actually a builder block. The cool thing about the builder pattern is that it uses this existing feature from the language and turns it into a whole new coding paradigm.
The success of building our own builder class by using the Groovy MOP depends largely on understanding the sequence in which these methods get called. The output gives us an idea of what might be happening and what the true sequence of events is. Let's decorate the code a little to show what is happening. The comments show what scope we are running in.

```groovy
// Script scope
def method1(param: "one") {
  // Closure1 scope
  method2 {
    // Closure2 scope
    method3 "hello"
  }
  // End Closure2
  method1 (123) {
    // Closure3 scope
    method1 ("nested") {
      // Closure4 scope
      method3 10
    }
  }
}
```

For More Information:

The main block of code runs within the scope of the script. Each inline closure is in fact an anonymous instance of a Closure object. For the purpose of this exercise we will name these instances Closure1 to Closure4. The first call to method1() occurs in the outer scope of the script so we would expect this method to be passed to the script instance. The subsequent method calls all happen within the scope of one or other of the anonymous closure instances, so we expect these methods to be invoked on the individual closure instances. The following sequence diagram illustrates this:
Resolve Strategy: OWNER_FIRST

The one problem with the previous diagram is that we know that the closure instances don’t implement the method1() to method3() methods. So this sequence diagram shows what methods are initially called, but it does not show what methods actually get called. When Groovy makes a call to a method on a closure, it does not always expect it to be present.

If a method is not present in the closure itself, Groovy will try to find it by looking in the owner of the closure, or its delegate, or both. The order in which this occurs is called the resolve strategy of the closure. The default resolve strategy is OWNER_FIRST, which means that the owner of the closure will be queried first for the method, followed by the delegate. If the owner of the closure happens to be another closure, then the resolve strategy will continue its search for a match in the owner of the owner and so on, until a match is found or the outer scope is reached.

The resolve strategy can be changed for a closure by calling Closure.setResolveStrategy. We can change the resolve strategy to any of the following self-explanatory strategies: OWNER_FIRST, OWNER_ONLY, DELEGATE_FIRST, DELEGATE_ONLY, and NONE.
Building a Builder

Although the preceding sequence diagram reflects the first port of call for each method invocation, what in fact happens is that the resolve strategy kicks in and the method calls will percolate out through the closure instances. A match will eventually be found in the script instance, which is the only place where the actual methods exist. Therefore, the actual calling sequence is better represented as follows:

![Sequence diagram showing the actual calling sequence of method invocations through closures.]

The insight that Groovy designers had when designing the builder pattern was that this natural nesting of closures could be used to map to any construction process that involved a parent-child type of relationship. Even without using a builder class, we can nest closures' method calls to create a pseudo builder. In the next example, we declare three methods that we can use to construct a rudimentary tree structure out of map objects.

For More Information:
The `root()` method creates the initial tree map and inserts a root element into it. We can nest as many levels deep as we like with the `node()` method, as it will remember its parent node and add sub nodes to it. The `leaf()` method is the only one to take a value and it does not expect to be passed a closure, as it will create the leaf elements in the tree structure.

```groovy
def current
def root (Closure closure) {
    def tree = [:]
    def root = [:]
    tree["root"] = root
    def parent = current
    current = root
    closure.call()
    current = parent
    return tree
}
def node (key, Closure closure) {
    def node = [:]
    current[key] = node
    def parent = current
    current = node
    closure.call()
    current = parent
}
def leaf (key, value) {
    current[key] = value
}

// pseudo builder code
def tree = root {
    node("sub-tree-1") {
        leaf "leaf-1", "leaf object 1"
    }
    node("sub-tree-2") {
        node("node-1") {
            leaf "leaf-2", "leaf object 2"
        }
    }
}
```

For More Information:
assert tree == [
    root: [
        "sub-tree-1": [
            "leaf-1": "leaf object 1"
        ],
        "sub-tree-2": [
            "node-1": [
                "leaf-2": "leaf object 2"
            ]
        ]
    ]
]

Pretended methods

Many Groovy builders rely on the ability to describe arbitrarily-named elements. When we make use of markup builders to generate XML, we need to be able to insert whatever tag names are required to conform to the schema that we are using. Given that elements are created in method calls, we also need to be able to make arbitrarily-named method calls during the markup process.

With Groovy, we can respond to methods that don't exist as concrete methods of a class. The term we use for this type of methods is **pretended methods**. Groovy provides two means for implementing a pretended method.

**invokeMethod**

The `PoorMansTagBuilder` class that we covered in Chapter 5 uses `invokeMethod` as a means of pretending methods. The `PoorMansTagBuilder` class works by handling all method calls to the builder, and invoking the closure argument manually. With `invokeMethod`, we can respond appropriately to any arbitrary method call. In this case, we output the appropriate XML tags.

```groovy
class PoorMansTagBuilder {
    int indent = 0
    Object invokeMethod(String name, Object args) {
        indent.times {print "    "}
        println "<${name}>
        indent++
        args[0].delegate = this // Change delegate to the builder
        args[0].call()
    }
}
```
This is a simple case that we are using just to illustrate the mechanism. Although the technique works for simple cases, extending it to implement a more complete tag builder would rapidly result in complex and hard-to-maintain code.

**methodMissing**

Since Groovy 1.5, an alternative to `invokeMethod` was provided. The `methodMissing` mechanism differs slightly from `invokeMethod`, as it is only called when a method call fails to be dispatched to any concrete method. To update the `PoorMansTagBuilder` for using `methodMissing` instead of `invokeMethod`, all we need to do is replace the method name that we declare.

```groovy
class PoorMansTagBuilder {
    int indent = 0
    def methodMissing(String name, args) {
        indent.times {print "    "}
        println "<${name}>
        indent++
        args[0].delegate = this // Change delegate to the builder
        args[0].call()
        indent--
        indent.times {print "      "}
        println "</${name}>"
    }
}

def builder = new PoorMansTagBuilder()

builder.root {
    level1{
        level2 {
        }
    }
}
```

For More Information:
Closure delegate

Earlier, we looked at how to code a pseudo builder using methods declared within a script. The resolve strategy in that example passed method calls in the nested closure up to the owner of the closure. The `builder` block in the previous example is also in the scope of a script. Let's decorate it as we did before, to identify the various anonymous closure instances.

```groovy
// method root() called on PoorMansTagBuilder
builder.root { // Closure1
    // method level1 called on Closure1 instance
    level1 { // Closure2
        // method level2 called on Closure2 instance
        level2 { // Closure3
            }
        }
    }
}
```

The first method call to `root()` is made against the builder instance, so it will be handled directly by `PoorMansTagBuilder.methodMissing()`. Nested method calls will first be dispatched to the enclosing closure. The `level1()` and `level2()` methods won't be found in the closure instances, so we would normally expect the resolve strategy to dispatch these methods up the chain of owners until a method is found. This normal dispatch chain would end up at the script instance, so these methods would cause a `MethodMissingException` to be thrown.

The secret of how this works is in the handling of the delegate for closure instances. The builder block starts with a direct method call onto the builder instance, `builder.root()`. Anonymous closure, `Closure1`, is passed as a parameter. The call to `root()` will fail and fall through to `methodMissing`. In this simple example, `arg[0]` is always the closure because we are not processing parameters on our tags. A more sophisticated version would need to scan the parameters for the closure instance.

At this point we have access to the closure, so we can set its delegate to the builder instance. Now when the `level1()` and `level2()` calls are encountered, the resolve strategy will try the owner first and then try the delegate as follows:

- The `level1()` call will not be resolved in `Closure1`. It won't be found in the owner of `Closure1`, which is the script, but it will be resolved in the delegate, which is the builder instance. `PoorMansTagBuilder.methodMissing` will field the method and also set the delegate for the anonymous closure, `Closure2`.
- The `level2()` call happens in the scope of `Closure2` but will not be resolved there. First its owner, `Closure1`, will be tried, and then its delegate, which once again is the builder instance.
BuilderSupport

Under the hood, all of Groovy's own Builders are implemented by using the `invokeMethod` or `methodMissing` and delegate techniques that we have described above. We could choose to start creating our own builder classes by using these features alone. Perhaps the biggest challenge when creating a builder with these features alone is that the MOP concepts of pretended methods and delegate handling don't fit well with the task at hand—namely the construction of complex objects. It would be nice to have APIs that reflected the task at hand in a better way.

Thankfully, the complexities of managing `invokeMethod` or `methodMissing` calls and figuring out who the delegate should be are encapsulated into the builder support classes provided in the Groovy packages. The most basic support class is `groovy.util.BuilderSupport`.

BuilderSupport hook methods

`BuilderSupport` provides an interface to the building process that nicely mirrors the node-based construction process on which most builder classes are based. Instead of overriding `invokeMethod` as in the initial example, we override the node construction methods provided by `BuilderSupport`.

These methods provide the hooks such that instead of coding at the MOP level with pretended method invocations and delegates, we can code with respect to the node construction process. The important hook methods that we need to know about are listed here. These methods encapsulate the building process out into node creation style events that make far more sense from an object construction point of view. Then we never need to worry about pretended methods or delegates again.

- `createNode(Object name)`
- `createNode(Object name, Object value)`
- `createNode(Object name, Map attributes)`
- `createNode(Object name, Map attributes, Objects value)`
  Called by `BuilderSupport` whenever a pretended method is encountered. The name parameter contains the name of the pretended call. The responsibility of the hook is to return a new object of a type appropriate to the method call. The specific hook to be called depends on what parameters are passed to the method.
- `nodeCompleted(Object parent, Object node)`
  Called after all of the children of a node have been created.
- `setParent(Object parent, Object child)`
  Called after `createNode` for each child node, in order to allow any parent-child relationships to be established.

For More Information:

BuilderSupport takes care of all of the nitty-gritty of handling pretended methods for us. Had our PoorMansTagBuilder worked only for parameter-less tags, BuilderSupport would have detected which type of call is being made and called the appropriate createNode for us. The setParent method is only called if a parent node exists.

How this all hangs together is best illustrated by way of an example. So let's start by creating a really dumb builder that just logs these methods as they are encountered. This will give us a feel for the sequence in which the methods are called.

```groovy
class LogBuilder extends BuilderSupport {
    def indent = 0
    def createNode(name) {
        indent.times {print " "}
        println "createNode(${name})"
        indent++
        return name
    }
    def createNode(name, value) {
        indent.times {print " "}
        println "createNode(${name}, ${value})"
        indent++
        return name
    }
    def createNode(name, Map attributes) {
        indent.times {print " "}
        println "createNode(${name}, ${attributes})"
        indent++
        return name
    }
    def createNode(name, Map attributes, value) {
        indent.times {print " "}
        println "createNode(${name}, ${attributes}, ${value})"
        indent++
        return name
    }
    void setParent(parent, child) {
        indent.times {print " "}
        println "setParent(${parent}, ${child})"
    }
    void nodeCompleted(parent, node) {
        indent--
        indent.times {print " "}
        println "nodeCompleted(${parent}, ${node})"
    }
}
```

For More Information:

To use this builder, all that we need to do is to construct one and start writing some markup with it. Here we have some markup for building customer records, but as this builder does not care what the method tags are, we could write whatever markup we please.

```groovy
def builder = new LogBuilder()

def customers = builder.customers {
  customer{
    id(1001)
    name(firstName:"Fred", surname:"Flintstone")
    address("billing", street:"1 Rock Road", city:"Bedrock")
    address("shipping", street:"1 Rock Road", city:"Bedrock")
  }
}
```

If we run this script, we will generate the following output. We can see from the output exactly what the sequence of calling is, and what parameters are being passed. We've used this simple example for illustrating how the BuilderSupport class works, but it is actually a useful debugging tool in general for using with any builder that's not behaving as expected. By replacing any existing builder instance in your code with a LogBuilder, it will output the construction sequence for you, which may identify the problem.

```groovy
createNode(customers)
createNode(customer)
  setParent(customers, customer)
createNode(id, 1001)
  setParent(customer, id)
nодеCompleted(customer, id)
createNode(name, [firstName:Fred, surname:Flintstone])
  setParent(customer, name)
nодеCompleted(customer, name)
createNode(address, [street:1 Rock Road, city:Bedrock], billing)
  setParent(customer, address)
nодеCompleted(customer, address)
createNode(address, [street:1 Rock Road, city:Bedrock], shipping)
  setParent(customer, address)
nодеCompleted(customer, address)
nодеCompleted(customers, customer)
nодеCompleted(null, customers)
```

**For More Information:**

From this output, we can trace the sequence in which the hooks are called. Nodes are created from the top down. The `createNode` hook for the parent is called first. The `createNode` hook for a child is called next, and `setParent` is called for each individual child after both the parent and the child have been created. The `nodeCompleted` hook is called only after all of the children have been created and their parent-child relations set.

The default implementation of `BuilderSupport` manages the current node cursor by itself. Two additional hooks to consider are:

- `setCurrent(Object current)`
- `Object getCurrent()`

Certain builder implementations might want to manage the notion of a "current" node object in order to maintain a cursor on the construction process. If so, both of these hooks will need to be implemented.

Now that we understand the building mechanism, it is a trivial matter to change our `LogBuilder` script to create some actual markup. Here, with a few modifications, we can turn our script into `PoorMansTagBuilder 2.0`.

```groovy
class PoorMansTagBuilder20 extends BuilderSupport {
    def indent = 0

    def createNode(name)
    {
        indent.times {print "  "}
        println "<${name}>
        indent++
        return name
    }

    def createNode(name, value)
    {
        indent.times {print "  "}
        println "<${name}" + value
        indent++
        return name
    }

    def createNode(name, Map attributes)
    {
        indent.times {print "  "}
        print "<${name} ">
        print attributes.collect {
            "${it.key}='${it.value}'"
        }.join(' ')
        println ">
        indent++
        return name
    }

    def createNode(name, Map attributes, value)
    {
        indent.times {print "  "}
        print "<${name} 
        print attributes.collect {
            "${it.key}='${it.value}'"
        }.join(' ')
        println ">
        indent++
        return name
    }
}
```

For More Information:
print attributes.collect {
    "\${it.key}='\${it.value}';"
}.join(' ')
println "">" + value
indent++
return name
}
void setParent(parent, child){
    // Don't care since we are just streaming to output
}
void nodeCompleted(parent, node) {
    indent--
    indent.times {print "  "}
    println "</${node}>"
}
}

Once again, this is a simple implementation of a tag builder. We are making no
interpretation of the method tags that are being passed in, so for each createNode
call all we do is output a <TAG> with parameters and attributes if necessary. The
setParent call is not relevant to us because we are just streaming output to standard
out. We will see in the next example where we need to implement this. Finally, the
nodeCompleted call just closes the tag </TAG>.

Now we can apply this builder to the same customers markup script
that we did before, as follows. The only change required is to instantiate a
PoorMansTagBuilder20 in place of the original builder class.

def builder = new PoorMansTagBuilder20()
def customers = builder.customers {
    customer {
        id(1001)
        name(firstName:"Fred", surname:"Flintstone")
        address("billing", street:"1 Rock Road", city:"Bedrock")
        address("shipping", street:"1 Rock Road", city:"Bedrock")
    }
}

For More Information:
Running this script will log some reasonably well-formed XML as follows:

```xml
<customers>
  <customer>
    <id>1001</id>
    <name firstName='Fred' surname='Flintstone' />
    <address street='1 Rock Road' city='Bedrock' >billing</address>
    <address street='1 Rock Road' city='Bedrock' >shipping</address>
  </customer>
</customers>
```

As a markup builder, this falls well short of the features in the Groovy MarkupBuilder, but it does show just how easy it is to put together a quick builder to fit the need of the day. Now let’s consider what we’ve learned, and look at building something a little more useful.

A database builder

Every application that includes a database needs some means of setting up seed, demo, or test data. I have worked on numerous enterprise applications during my career and invariably the management of different data sets becomes as much of an effort over time as the development of the application itself. In my own experience of working with Independent Software Vendors (ISVs), whose applications need to be deployed on multiple customer sites with multiple versions, the problem becomes acute.

ISV companies often have competing needs for data sets. The sales organization needs a predictable data set for its demos to customers. The test department needs data sets that allow them to test specific features of the application. Project management requires specific seed data to be available, which is tailored to each customer site prior to installation. With all of these competing requirements, the IT department has a limited set of database instances available on which to install and test all of these configurations.

There are various ways of managing data sets. The most common is to maintain SQL scripts that take care of the database insertions. Building a simple database will require multiple insertions into a multitude of tables. The SQL script needs to be written in such a way as to maintain the integrity of foreign key references. It’s not an easy thing to do, and requires intimate knowledge of the schema.
Suppose we are working with a Grails application. Take for example the one-to-
many relationship we looked at in Chapter 6:

class Customer {
    String firstName
    String lastName
    static hasMany = [invoices:Invoice]
}

class Invoice {
    static hasMany = [orders:SalesOrder]
}

class SalesOrder {
    String sku
    int amount
    Double price
    static belongsTo = Invoice
}

Grails has a migration plug-in that can be installed. The migration tool will execute
a SQL update script to migrate our database between versions. To use the migrate
tool to add some simple test data for the above classes, we need to know how GORM
maps from the Groovy POGO objects to relational tables.

In Chapter 6, we also saw how these classes in fact map to five tables in the relational
database. There are three main tables that represent the business objects (customer,
invoice, and sales_order) and there are two mapping tables used to manage the
foreign key mappings (customer_invoice and invoice_sales_order) that relate
customers to invoices and invoices to sales orders.

To set up a simple test case with one customer, one invoice and three sales orders
would require nine insertions across these tables. Apart from being error prone
and difficult to maintain, the script will be incomprehensible to anyone who is not
intimately acquainted with SQL. What starts out as a simple data input spec for a
test case becomes a development task for a domain SQL expert who understands the
GORM mapping model.

An alternative to this approach is to use the GORM APIs to build the test data. At
least if we do this then we don’t have to concern ourselves with the foreign key
relationships between tables. The script below will set up our simple data set
with one customer, one invoice, and three sales orders:

def fred = new Customer(firstName:"Fred", lastName:"Flintstone")

    fred.save()}
def invoice = new Invoice()

invoice.addToOrders(new SalesOrder(sku:"productid01", amount:1, price:1.00))
invoice.addToOrders(new SalesOrder(sku:"productid02", amount:3, price:1.50))
invoice.addToOrders(new SalesOrder(sku:"productid03", amount:2, price:5.00))

fred.addToInvoices(invoice)

This is somewhat better than the SQL script approach, but it does impose a procedural construction onto the data, where the test data is typically defined declaratively. While I've used GORM to illustrate my point here, the same issues will crop up whatever persistence mechanism we use.

Ideally, we want to be able to describe our data in a declarative style. The syntax of the data definition should as closely match the structure of the resulting data as possible. This is an ideal situation in which to use a builder to take care of construction. With a builder, it should be possible to create a declarative markup style script for building data sets. The builder can take care of the complexities of construction.

Let's first of all imagine how a builder for customers may look in use. We probably want to handle multiple customers, so a top-level customers method is useful. We could have multiple customer blocks nested below that. Nesting is a good way of depicting ownership in a one-to-many relationship, so our Customers markup would probably look something like the following:

```groovy
builder.customers {
    customer {
        invoice {
            salesOrder()
            salesOrder()
            salesOrder()
        }
    }
}
```

We need to be able to set the fields of each entity as it is created. We could have a pretended method for each field as follows:

```groovy
builder.customers {
    customer {
        firstName("Fred")
        lastName("Flintstone")
        invoice {
            salesOrder {
```
This will work. However, it is not immediately clear to a reader of this script that `lastName` is an object attribute and `invoice` is a new subsidiary object. A better option is to set object attributes as mapped parameter values. The following script is far clearer in its intent, so this is the one we will try to implement:

```groovy
builder.customers {
    customer(firstName:"Fred",lastName:"Flintstone") {
        invoice {
            salesOrder(sku:"productid01", amount:1, price:1.00)
            salesOrder(sku:"productid02", amount:2, price:1.00)
            salesOrder(sku:"productid03", amount:3, price:1.00)
        }
    }
}
```

As it happens, the `BuilderSupport` hook methods and their calling sequence work perfectly in step with the GORM APIs that we need in order to construct our customer records.

- The `createNode` method will be called in the correct top down sequence, allowing us to create the appropriate `Customer`, `Invoice`, or `SalesOrder` as required.
- The `setParent` hook is called after both parent and child objects have been constructed, allowing us to call `Customer.addToInvoices` or `Invoice.addToOrders` when we need to.
- The `nodeCompleted` hook can be used to intercept when an object needs to be saved.

For More Information:

The following code snippet contains a rudimentary builder class based on BuilderSupport that constructs customer, invoice, and sales order objects through GORM. The same style of builder could work equally well with whatever other persistence method we choose.

class CustomersBuilder extends BuilderSupport {
    def createNode(name){
        Object result = null
        switch (name) {
            case "customer":
                return new Customer(firstName:"", lastName:"")
            case "invoice":
                return new Invoice()
            case "salesOrder":
                return new SalesOrder(sku:"default", amount:1, price:0.0)
        }
    }
    def createNode(name, value){
        Object result = createNode(name)
        if (value instanceof Customer && result instanceof Invoice)
            value.addToInvoices(result)
        if (value instanceof Invoice && result instanceof SalesOrder)
            value.addToOrders(result)
        return result
    }
    def createNode(name, Map attributes){
        Object result = null
        switch (name) {
            case "customer":
                return new Customer(attributes)
            case "invoice":
                return new Invoice(attributes)
            case "salesOrder":
                return new SalesOrder(attributes)
        }
    }
    def createNode(name, Map attributes, value){
        Object result = createNode(name, attributes)
        if (value instanceof Customer && result instanceof Invoice)
            value.addToInvoices(result)
        if (value instanceof Invoice && result instanceof SalesOrder)
            value.addToOrders(result)
        return result
    }
}
void setParent(parent, child) {
    if (child instanceof Invoice && parent instanceof Customer)
        parent.addToInvoices(child)
    if (child instanceof SalesOrder && parent instanceof Invoice)
        parent.addToOrders(child)
}

void nodeCompleted(parent, node) {
    if (node != null)
        node.save()
}

Here we have implemented all four createNode methods. The method tag is passed as the name parameter to createNode. So we construct a Customer, Invoice, or SalesOrder object based on the tag that we are processing. We allow a parent object to be set in the value parameter to the method. This allows us to construct a child object outside the scope of a parent, and set its parent later.

The setParent method takes care of adding invoices to customers and sales orders to invoices. Testing the instanceof both parent and child ensures that we don't attempt to add an invoice if it is declared outside of the customer.

All that remains for nodeCompleted to do is to save the node object that we have created to the database. When we put all of this together, we can make use of our CustomerBuilder to build a simple test database as follows:

def builder = new CustomersBuilder()

def customers = builder.customers {
    fred = customer(firstName:"Fred", lastName:"Flintstone") {
        invoice {
            salesOrder(sku:"productid01", amount:1, price:1.00)
            salesOrder(sku:"productid02", amount:2, price:1.00)
            salesOrder(sku:"productid03", amount:3, price:1.00)
        }
    }
    invoice2 = invoice(fred)

    salesOrder(invoice2, sku:"productid04", amount:1, price:1.00)
    salesOrder(invoice2, sku:"productid05", amount:1, price:1.00)
    salesOrder(invoice2, sku:"productid06", amount:1, price:1.00)
}

For More Information:
Building a Builder

By allowing a parent object to be passed as the value parameter, we have made the markup script more flexible. As you can see from the preceding code, invoice and salesOrder tags can be declared directly as children of a parent object, or they can be declared independently. This gives us a bit more flexibility in what types of mapping relationships we can support where ownership between parent and child might be optional, or in more complex scenarios where many-to-many relationships might need to be declared.

FactoryBuilderSupport

BuilderSupport is the base class for many of the builder classes provided in the Groovy packages. As we can see from the previous examples, it is easy to work with. We have built quite a useful database builder tool in relatively few lines of code.

However, one issue with BuilderSupport is that the hook functions are in effect funnels for handling all of the possible tags that we might like to process in our markup. In our CustomerBuilder we are handling just four different tags.

This is not a realistic scenario for most database schemas. We could expect to have dozens more tag types that we need to handle if we wanted to expand this example into something that would work with a typical database schema for even a modestly sized application. Funneling all of these tags into one createNode would create an unwieldy mess of code.

```groovy
def createNode(name){
    Object result = null
    switch (name) {
        case "customer":
            return new Customer(firstName:"", lastName:"")
        case "invoice":
            return new Invoice()
        case "sales_order":
            return new SalesOrder(sku:"default", amount:1, price:0.0)
        case "another_object":
            return new Something()
        ....... and more!
    }
}
```

For More Information:

Groovy provides a second builder support class that neatly overcomes this problem. The `groovy.util.FactoryBuilderSupport` class is based on the factory pattern, and delegates the handling of individual tag objects to Factory classes. Originally, this support class was just provided as part of the `SwingBuilder`. Because it was clear that this was more generally useful, the code was then refactored to be generally usable as a standalone `Builder` class. Since then, it has become the basis for other builders, such as the recently-added `JmxBuilder`, and is available to us for deriving our own factory-based builders.

`FactoryBuilderSupport` works by orchestrating the construction process in concert with Factory classes. When the `FactoryBuilderSupport` class encounters a method tag, it constructs an appropriate `Factory` object to handle it. The factory provides method hooks that implement the construction of the individual object and the setting up of parent-child relationships between the objects.

To implement a builder with the `FactoryBuilderSupport` class, we must first declare a Factory class for each object type that we wish to process. Factory classes are derived from the `groovy.util.AbstractFactory` class and need to overload some or all of the following methods from the `AbstractFactory` class:

- **newInstance**
  Called by `FactoryBuilderSupport` whenever it wants an object of a particular type to be constructed. It is similar to the `createNode` methods of `BuilderSupport` except that there is just one `newInstance` method, which accepts all argument types regardless of whether a value or attributes are supplied or not.

- **onHandleNodeAttributes**
  Allows the `Factory` class to take over the management of attributes. It can stop the builder from processing attributes by returning `true`.

- **setParent** and **setChild**
  Provide hooks for managing the parent-child relationships between objects.

- **isLeaf**
  We set this method to return `true` if the method tag being handled should be a leaf node and stops the builder treating any subsequent method calls as object declarations.

- **onNodeCompleted**
  Called when a node is completed, in order to allow any finalization of the object to be performed. It is similar to `nodeCompleted` in `BuilderSupport`.

For More Information:

To build a replacement for the CustomerBuilder with FactoryBuilderSupport, we first need to define Factory classes for each of the tag methods that we need to process. The first of these is the customers tag, which is straightforward enough. This tag does not cause any objects to be created, so all we do is return the tag name as the object created.

```java
public class CustomersFactory extends AbstractFactory {

    public boolean isLeaf() {
        return false
    }

    public Object newInstance(FactoryBuilderSupport builder,
                                Object name, Object value, Map attributes
                            ) throws InstantiationException, IllegalAccessException {
        return name
    }
}
```

We then define a factory class for the customer object. The methods that we need to implement are isLeaf (returns false), newInstance (to create the customer object), and onNodeCompleted (to save it).

```java
public class CustomerFactory extends AbstractFactory {

    public boolean isLeaf() {
        return false
    }

    public Object newInstance(FactoryBuilderSupport builder,
                               Object name, Object value, Map attributes
                           ) throws InstantiationException, IllegalAccessException {
        Customer customer = null
        if (attributes != null)
            customer = new Customer(attributes)
        else
            customer = new Customer()
        return customer
    }

    public void onNodeCompleted(FactoryBuilderSupport builder,
                                 Object parent, Object customer) {
        customer.save()
    }
}
```

For More Information:
The factory for invoices is equally straightforward. The only addition is that we need to take care of the parent-child relationship between customer and invoice. We do this by adding a setParent method, which will call addToInvoices on the customer object if required. We also need to check the value parameter passed to newInstance to see if a parent is being set at this point.

```java
public class InvoiceFactory extends AbstractFactory {

    public boolean isLeaf() {
        return false
    }

    public Object newInstance(FactoryBuilderSupport builder,
            Object name, Object value, Map attributes
        ) throws InstantiationException, IllegalAccessException {
        Invoice invoice = null
        if (attributes != null)
            invoice = new Invoice(attributes)
        else
            invoice = new Invoice()
        if (value != null && value instanceof Customer)
            value.addToInvoices(invoice)
        return invoice
    }

    public void setParent(FactoryBuilderSupport builder,
            Object parent, Object invoice) {
        if (parent != null && parent instanceof Customer)
            parent.addToInvoices(invoice)
    }

    public void onNodeCompleted(FactoryBuilderSupport builder,
            Object parent, Object invoice) {
        invoice.save()
    }
}
```

The factory for sales orders is identical to invoices except that we now return true from isLeaf because a sales order object will always be a leaf node in our tree.

```java
public class SalesOrderFactory extends AbstractFactory {
    public boolean isLeaf() {
        return true
    }
}
```

For More Information:
Building a Builder

```java
public Object newInstance(FactoryBuilderSupport builder,
                        Object name, Object value, Map attributes
) throws InstantiationException, IllegalAccessException {
    SalesOrder sales_order = null
    if (attributes != null)
        sales_order = new SalesOrder(attributes)
    else
        sales_order = new SalesOrder()
    if (value != null && value instanceof Invoice)
        value.addToOrders(sales_order)
    return sales_order
}

class CustomersFactoryBuilder extends FactoryBuilderSupport {
    public CustomersFactoryBuilder(boolean init = true) {
        super(init)
    }

    def registerObjectFactories() {
        registerFactory("customers", new CustomersFactory())
        registerFactory("customer", new CustomerFactory())
        registerFactory("invoice", new InvoiceFactory())
        registerFactory("sales_order", new SalesOrderFactory())
    }
}
```

All of the intelligence of how to orchestrate the construction process is encapsulated in the FactoryBuilderSupport class. So literally all we need to do for the whole builder to work is to register the Factory classes with appropriate tag names.

For More Information:
FactoryBuilderSupport uses reflection at runtime to detect what registration methods to run. By scanning the list of methods in the MetaClass instance and looking for methods that begin with "register", FactoryBuilderSupport detects whether any additional registration methods are provided in the derived builder class. In the preceding example, the only registration method added is registerObjectFactories, but we could well have written:

```groovy
def registerCustomers() {
    registerFactory("customers", new CustomersFactory())
}

def registerCustomer() {
    registerFactory("customer", new CustomerFactory())
}

def registerInvoice() {
    registerFactory("invoice", new InvoiceFactory())
}

def registerSalesOrder() {
    registerFactory("sales_order", new SalesOrderFactory())
}
```

FactoryBuilderSupport would detect all of these and run them in turn. Which method you use is a matter of choice. The only issue that you need to be aware of is that the registration methods will not be called in any predetermined order. If there are dependencies in your registration code, then it's best to group these into a single registration method.

To finish, we can drop this modified builder right where we previously used CustomersBuilder, and it will work in the same way.

```groovy
def builder = new CustomersFactoryBuilder()

def customers = builder.customers {
    fred = customer(firstName:"Fred", lastName:"Flintstone") {
        invoice {
            salesOrder(sku:"productid01", amount:1, price:1.00)
            salesOrder(sku:"productid02", amount:2, price:1.00)
            salesOrder(sku:"productid03", amount:3, price:1.00)
        }
    }
    invoice2 = invoice(fred)
    salesOrder(invoice2, sku:"productid04", amount:1, price:1.00)
    salesOrder(invoice2, sku:"productid05", amount:1, price:1.00)
    salesOrder(invoice2, sku:"productid06", amount:1, price:1.00)
}
```

For More Information:
In terms of management and maintenance, this version is far superior. Adding capabilities now for new tables will simply involve writing a new Factory class and registering it.

**Summary**

In Chapter 5, we discussed how builders worked via the Groovy MOP. In this chapter, we have taken a deeper look at how features of the MOP are used to implement the builder pattern. We've looked at the language features used to create a builder, and seen how they involve implementing pretended methods and influencing how methods calls are resolved. Implementing a builder directly by using the MOP in this way focuses on the nuts and bolts of the semantics of the builder, rather than the construction process.

In this chapter, we have seen how Groovy provides two useful support classes that make it much simpler to implement our own builders than if we use the MOP. We've seen how to use `BuilderSupport` and `FactoryBuilderSupport` to create our own builder classes.

Using these support classes greatly simplifies the implementation of builders. Hopefully, this will inspire you to see opportunities to develop your own Groovy-based builders for your own applications. You can find the full documentation for all of the classes that we covered here on the Codehaus website. The Groovy document for the classes can be found at [http://groovy.codehaus.org/api/groovy/util/package-summary.html](http://groovy.codehaus.org/api/groovy/util/package-summary.html).
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.