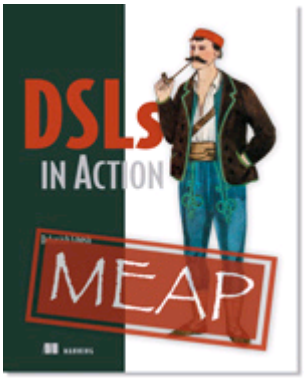
MANNING PUBLICATIONS

Homogeneous and Heterogeneous Integration Patterns

Excerpted from



DSLs in Action
EARLY ACCESS EDITION

Debasish Ghosh
MEAP Release: October 2009
Softbound print: Summer 2010 | 375 pages
ISBN: 9781935182450

This article is taken from the book DSLs in Action*. The author discusses integration in which the involved languages interoperate within the constraints of the underlying virtual machine. In particular, the author focuses on homogenous integration patterns, while briefly covering heterogeneous integration patterns with external DSLs.*

Expressive abstractions lead to easier understanding of the code base and reduce the feedback cycle from the domain experts. But, at the end of the day, however expressive your DSL design, you need to integrate it to the chores of your regular Java application. And there are certain aspects related to this integration that you need to take care upfront.

Ok, so now you have a bunch of DSLs written in multiple JVM languages. How would you consider integrating them seamlessly within your existing application? Remember that DSLs tend to evolve *independently* of the main application. So, your architecture needs to be flexible enough to take care of composing the changing DSLs with minimal impact on the running application.

Depending on the language and the type of DSL you use, you will come across several common patterns that recur in all integration activities. When you integrate an internal DSL with your application, it's mostly in the form of a library written in yet another language for the same runtime. Integration tends to be easier in such cases – we discuss these homogeneous integration patterns. External DSLs may be more challenging to integrate, as you will also see in this article.

Homogeneous integration patterns

You design an internal DSL using the infrastructure of a host language, whereas for an external DSL you design an entirely new language from scratch. We will look at various ways of integrating internal DSLs with your application.

When we talk about virtual machine-based application development and integration of internal DSL with your application, we assume that the DSL is developed in any of the languages that have source code interoperability with the main language of your application. It's not that a Java-based application always integrates with an internal

DSL implemented in Java alone – it can be using any of the JVM languages that can inter-operate with Java. I call this *homogeneous integration* because the languages involved interoperate well enough within the constraints of the underlying virtual machine. Have a look at Figure 1, which illustrates how DSLs developed using Java, Groovy, and Spring configuration integrate homogeneously on the JVM. Each of the DSLs can be deployed as a jar file that the main application can refer to.

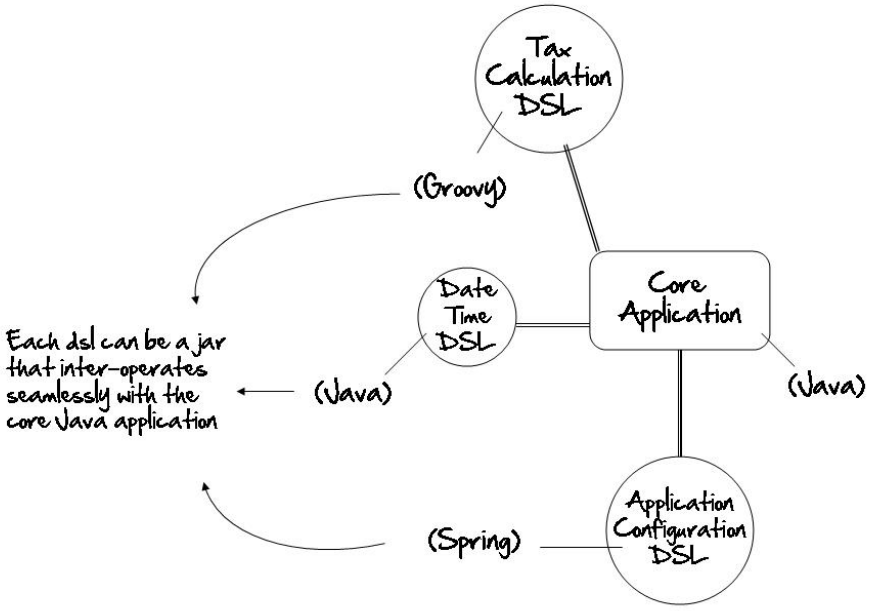


Figure 1 All three DSLs integrate homogeneously with the core application. Each DSL can be deployed as jar files that interoperate seamlessly within the JVM.

Suppose you are developing an application using Java as the primary programming language. But as a polyglot, you choose to use the power of Groovy to implement your XML Builder functionality and a JRuby-based DSL loaded through Spring beans for managing application configurations. How should you integrate your DSLs with your core application? The integration has to be such that it should not incur too much of complexity on the user's part. At the same time, you need to keep the DSL sufficiently decoupled from the core application for managing its own evolution and lifecycle over time. There are quite a few ways in which you can integrate your DSL written in these JVM languages with the Java application. Table 1 lists some of these main patterns.

Table 1 Integration points published by Internal DSLs

Internal DSL Pattern	Published Integration Point
Java 6 Scripting Engine	Processing DSL written in scripting languages like Groovy through the corresponding scripting engine that comes with Java 6
DSL Wrapper	Wrap Java objects with smarter APIs written in languages like JRuby, Scala or Groovy and use the Java integration capabilities of these languages
Language specific Integration Features	Some languages like Groovy offer direct integration with Java through abstractions that can load and parse scripts directly

Spring based Integration	Spring allows loading of beans written in dynamic languages directly into your application through declarative configuration.

Each of these languages offers a number of options that you can use to do the integration. You need to be aware of the pros and cons of each of these and use the option that best suits your problem. Let's look at some of the options that each of these languages offers for integration with your Java application.

Java 6 scripting engine

Java as a platform has become ubiquitous. And all of us, the representative programmers of the community have been talking about a unification layer that allows interoperability across all languages that the platform embraces. Java 6 scripting functionality allows you to embed scripting languages within Java applications through usage of appropriate engines. You can now integrate DSLs implemented using languages like Groovy or JRuby through the Java APIs defined in `javax.script` package.

Let's look at an example of such integration taking a Groovy script for executing the DSL for order creation. We will look at that DSL integrated and invoked from within a Java application. This will give you an idea of the power of Java scripting as an enabler towards DSL integration.

Let's assume that we have Groovy DSL implementation for processing client orders (`ClientOrder.groovy`), as shown in Listing 1.

Listing 1 ClientOrder.groovy : Order Processing DSL in Groovy

```
ExpandoMetaClass.enableGlobally()

class Order {
  def security
  def quantity
  def limitPrice
  def allOrNone
  def value
  def bs

  def buy(su, closure) {
    bs = 'Bought'
    buy_sell(su, closure)
  }

  def sell(su, closure) {
    bs = 'Sold'
    buy_sell(su, closure)
  }

  def to() {
    this
  }

  private buy_sell(su, closure) {
    security = su[0]
    quantity = su[1]
    closure()
  }
}

def methodMissing(String name, args) {  #1
  order.metaClass.getMetaProperty(name).setProperty(order, args)
}

def getNewOrder() {
  order = new Order()
}

def valueAs(closure) {  #2
```

```
  order.value = closure(order.quantity, order.limitPrice[0])
  order
}

Integer.metaClass.getShares = { -> delegate }  #3
Integer.metaClass.of = { instrument ->  [instrument, delegate] }
```

#1 Hook to intercept non-existent method calls
#2 Inline valuation strategy specification
#3 Meta-programming to inject new methods

In another script file, `Order.dsl` in Listing 2, the DSL user does the scripting, again in Groovy that uses the above implementation to place orders for the client. Note that this script is purely based on the DSL that we designed above and assumes minimal understanding of the programming language machineries. Along with creating the orders, the script also accumulates them into a collection that gets returned to the caller. But who is the caller here? We'll see that shortly, and unravel the mystery that integrates this Groovy DSL with its parental lineage of the Java platform.

Listing 2 order.dsl : Groovy script for Order Placement

```
orders = []
newOrder.to.buy(100.shares.of('IBM')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << order  #1

newOrder.to.buy(150.shares.of('GOOG')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << order

newOrder.to.buy(200.shares.of('MSOFT')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << order
orders   #2
```

#1 Add order to the collection
#2 Return the collection to the caller

Now we come to the interesting part of the story where we will integrate our DSL implementation and the script with the main Java application. The code fragment in Listing 3 shows the snippet of the main application, which expects the collection of order to be returned from the DSL execution so that it can do further processing on it. The Java code in the snippet below uses the scripting engine for Groovy. Similarly, there are available implementations of scripting engines[1] for other JVM languages like JRuby, Clojure, Rhino, and Jython that can be as seamlessly integrated within your Java application.

Listing 3 Java Application code that invokes the Groovy DSL

```
ScriptEngineManager factory = new ScriptEngineManager();  #1
ScriptEngine engine = factory.getEngineByName("groovy");  #2

List<?> orders  (#4) = (List<?>)
  engine.eval(new InputStreamReader(
```

[1] https://scripting.dev.java.net/

```
    new BufferedInputStream(
      new SequenceInputStream(
        new FileInputStream("ClientOrder.groovy"),
        new FileInputStream("order.dsl"))));   #3

System.out.println(orders.size());
for(Object o : orders) {
  System.out.println(o);   #5
}
```

Let's have a more detailed look in the above code snippet at the steps that we followed in achieving the integration. Figure 2 shows the steps in the form of a sequence diagram, annotated with the actions that Listing 3 performs on the DSL script and implementation.

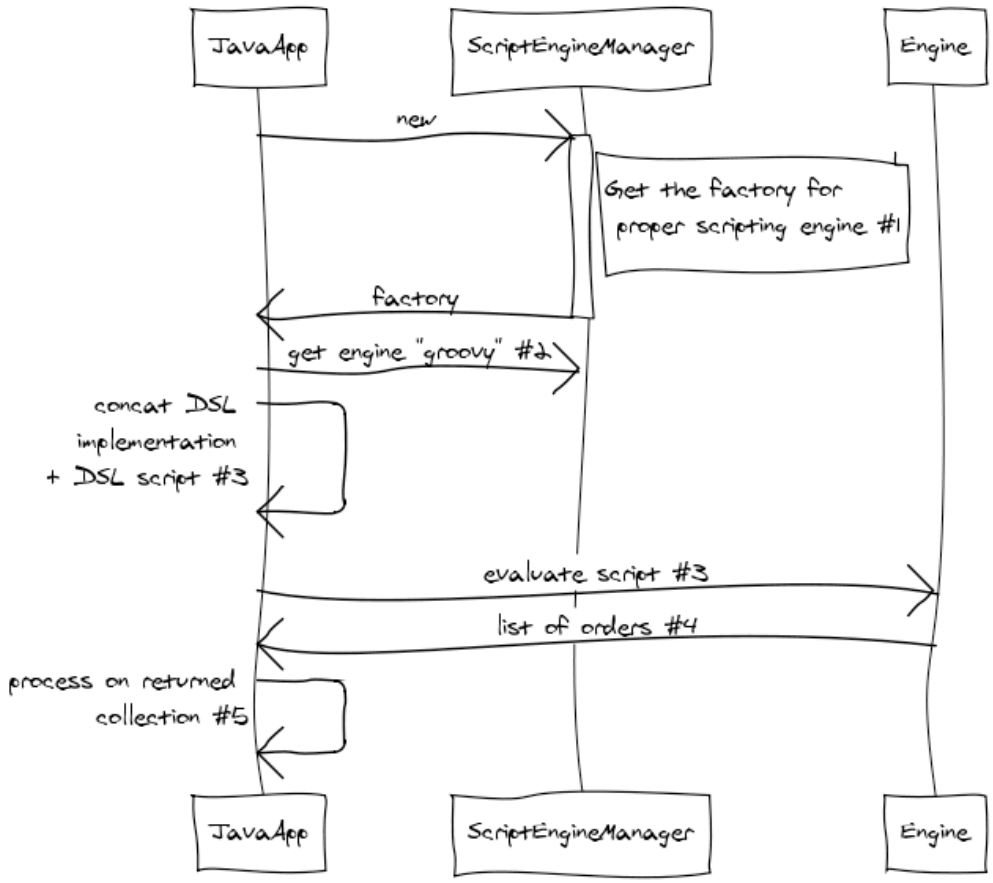


Figure 2 Integrating Groovy DSL through Java 6 Scripting Engine. The interaction diagram shows the steps involved in evaluating the Groovy DSL script within the sandbox of the ScriptEngine.

Java 6 scripting APIs provide a viable option to integrate your DSL designed using almost any JVM language into your Java application. `javax.script` also offers APIs that allow you to set up bindings of variables at various scopes to exchange meaningful information between your DSL and the Java components.

Java 6 scripting is one of the most generic ways to interoperate among JVM languages. And like any other generic strategy, there's always a better option for the specific language that you are working with. Because the DSL script gets loaded in a separate classloader and executes in its own sandbox, you face problems of inter-operability between the Groovy and Java abstractions. Also, since the script executes in the sandbox of the

`ScriptEngine`, in case of any exception, the line numbers mentioned in the stack trace will not match the line number of the source file. This can lead to difficulty in debugging exceptions thrown from the DSL script. We'll explore some of the better options when we talk about more focused integration strategies. *As a note of caution, consider integration using Java 6 scripting API as the last resort when you're working with non-trivial DSL implementations*.

Scripting engines were introduced in Java 6 as a generic way to handle script execution from within your Java programs. The design principles of Java 6 `ScriptEngine`–based APIs cater to all JVM languages that implement JSR-233 compliant infrastructure. If the language that you plan to use for DSL implementation offers its own specific ways of integrating with Java, review it carefully before deciding in favor of JSR-233 compliant implementations. It's more likely that the language specific solution will be more idiomatic, simple, and recommended as a better practice.

A DSL wrapper

You just saw how the scripting APIs of Java 6 work as an enabler towards polyglotism[2] on the Java Virtual Machine. I picked up the example in Groovy because it looked quite seamless how the same DSL could be plugged into the Java application without much of a fuss. You can use similar techniques to plug in DSLs written in other JVM languages like JRuby or Clojure or Rhino into your Java application.

DSL integration can happen at various levels; the Java scripting option that I've discussed earlier allows you to embed your DSL within the execution machineries of the `ScriptEngine` and invoke the DSL scripts. It has the advantage that your DSL is totally decoupled from the application and executes within the sandbox of the `ScriptEngine` context. The disadvantage of this approach is that it's not intuitive to have the DSL components interact easily with the environment of the main application.

Let's look at another approach towards DSL integration where you build the DSL as a wrapper layer on top of the main application components using the rich features that the DSL host language offers. This is a useful approach that you can adopt to make your legacy applications publish smarter APIs. Using the rich language features of yet another JVM language, you can make more expressive domain components based on the legacy abstractions. Not only your domain experts will love them, but also your fellow API users will enjoy using them.

In this example, I will use Scala, the statically typed language for the JVM that also has nice interoperability features with Java. Suppose your main application is written in Java and you've all your domain objects implemented as part of the application. Your client is obviously becoming increasingly aware of the hype and fun of DSL-based development and has been asking you to implement some smart DSL features on top of their existing legacy Java trading application. If we previously discussed DSL *integration by embedding*, this is going to be a journey towards *integration by wrapping*.

Financial brokerage systems: client account

In order to do trading, a client needs to open an account with the Stock Trading Organization (STO) called the *trading account*. All trades for the client will be booked in that account and recorded by the STO. Once the trade is done, the settlement process has to be initiated to do the final balancing of the securities and currencies exchanged between the two parties. For example, Client XXX buys 100 shares of SONY at 50 USD per share through STO Nomura Securities. The STO gets those securities from the Stock Exchange where a broker does the sell. In this case, after the trade is made, there has to be a settlement process that will involve an exchange of

[2] The general idea behind polyglotism is that every individual section of a project can be implemented using the language that best suits the problem domain.

100 shares of SONY and approximately 5000 USD between the two counterparties. This settlement has to be done through an account known as the *settlement account*, which can be same as the trading account of the client or it can be a different account. So, the trading account is used for doing the trade and the settlement account is used for settling the trade. They can be the same or a different account.

Consider the following domain model of `Account`, an entity from our friendly domain of securities trading operations. An account is an abstraction through which the firm, its clients and the brokers manage trading and settlement activities. The previous callout gives a brief explanation of the role of accounts and their types in trading and settlement operations. In Listing 4, we have some of the attributes of `Account` that are required to discuss our wrapper approach of DSL integration. `Account` is a Java class on which we will implement Scala wrappers and find out how the usage patterns in client APIs become more succinct and expressive. The standard getter methods of `Account` have been elided for convenience, though some of them will be used later when we implement the DSL on top of this class. Also note that, in real life modeling, an `Account` is a much more complex abstraction than what we discuss here. The current model is only limited in scope for our context of discussing DSL integration features.

Listing 4 Account Domain Object in Java

```
public class Account {
    public enum STATUS { OPEN, CLOSED }

    public enum TYPE { TRADING, SETTLEMENT, BOTH }

    private String number;
    private String firstName;
    private List<String> names = new ArrayList<String>();
    private STATUS status = STATUS.OPEN;
    private TYPE accountType = TYPE.TRADING;
    private float interestAccrued = 0f;

    public Account(String number, String firstName) {
        this.number = number;
        this.firstName = firstName;
    }

    public Account(String number, String firstName, TYPE accountType) {
        this(number, firstName);
        this.accountType = accountType;
    }

    //.. getters ommitted

    public float calculate(final Calculator c) {
        interestAccrued = c.calculate(this);
        return interestAccrued;
    }

    public boolean isOpen() {
        return status.equals(STATUS.OPEN);
    }

    public Account addName(String name) {
        names.add(name);
        return this;
    }
}
```

If you've been programming in Java for quite some time now, I am sure you are neither amused nor surprised with the verbosity and boilerplate stuff that the above model imbibes. Let's try to figure out how we can make the abstraction smart enough so that your client can get some APIs that allow him to express his intents in a far more domain-rich vocabulary. At the end of our exercise, we will come up with a DSL that will integrate nicely into the guts of your Java application.

Let's start with an abstraction `AccountDSL` in Scala that will act as an adapter to the earlier Java class and implement, what we call, smart domain APIs. But, remember, our ultimate aim will be to make our `Account` class smart enough, so that, whatever DSL we design, the client should be able to apply the language on existing instances of the `Account` class. In the following code snippets, I will show you how to enrich the `AccountDSL` abstraction incrementally along with potential uses of the DSL, so that you get a feel of the enrichment as it occurs in the domain abstraction. Always remember one important thing – all of the DSL stuff that we are developing here will be in the same workspace as your Java objects, maybe decoupled into a separate package structure. Hence, the integration aspect is ensured out of the box. Here's the rich version of our `Account` class:

Listing 5 Account DSL in Scala

```
import scala.collection.jcl._

class AccountDSL(value: Account) {

  import scala.collection.JavaConversions._
  def names =                          #1
    value.getNames.toSeq.toList ::: List(value.getFirstName)

  def belongsTo(name: String) = {      #2
    (name == value.getFirstName) || (names exists(_ == name))
  }

  def <<(name: String) = {     #3
    value.addName(name)
    this
  }
  //..
}
```

#1 Convert Java collection to Scala for richer semantics
#2 Domain API
#3 New operator syntax on collection

Let's look at some of the salient features that the above implementation embodies. The code snippet we just discussed uses some of the typical Scala idioms, which I briefly gloss over in the sidebar that follows.

- `AccountDSL` is an adapter to the Java Account class and wraps it as an underlying implementation.
- Scala collections are always semantically richer than Java collections in the sense that we can apply higher order functions to make operations on them more expressive. In #1, we convert the Java collection to a Scala one, which we will use subsequently along with higher order functions. The code snippet shown in Listing 5 uses Scala 2.8 `implicit` conversions between Java and Scala collections. In case you are still working on a pre-Scala 2.8 version, you can use the `jcl` conversion APIs as follows:

```
def names =
  (new BufferWrapper[String] {
    def underlying = value.getNames
  }).toList ::: List(value.getFirstName)
```

- We define a domain API `belongsTo` (#2) using our new Scala collection and higher order functions. Note the succinctness of implementation that Scala offers.
- We define an operator like syntax << to make our DSL more expressive and concise (#3).

Scala 101

In the method `belongsTo`, we use a predicate as:

```
>> (names exists(_ == name))
```

This is a succinct way of expressing the following in Scala:

```
>> (names.exists(n => n == name))
```

1. In Scala the "." is optional when you invoke methods on a receiver.

2. The "_" that we use is shorthand for anything substitutable in Scala. In the above snippet, "_" is the placeholder for supplying the parameter to the higher order function that `exists` accepts.

3. Note the Scala type inferencer performs the inferencing of the parameter type of the function that `exists` takes.

4. In Scala, operators are methods. Hence, we can define << as the method that adds the `name` to the order object. This may be visually appealing to some. But on a cautionary note, this is highly a matter of personal choice and can lead to unreadable code if not used with proper control.

With the new Scala APIs wrapping our original Java implementation, the client can afford to express his domain intents more succinctly, as we will see shortly. But before that, we need to take care of one other thing that I promised earlier. We need to make `Account` interoperable with `AccountDSL`, so that all the intelligence that we implement on top of `AccountDSL` can also be applied on `Account` instances. We can do this in Scala using the *implicits* feature. Include the following definition in the scope that is visible to the path of execution of the DSL[3]:

```
implicit def enrichAccount(acc: Account): AccountDSL =
  new AccountDSL(acc)
```

Implicits in Scala

Note that in the last snippet we have an `implicit` modifier in front of the method definition `enrichAccount`. In Scala, the `implicit` modifier for a method is used to define the automatic conversion from one type to another. In the above case, the method `enrichAccount` converts an `Account` to an instance of `AccountDSL`. So, instead of using:

```
scala> enrichAccount(acc1) belongsTo("David P.")
```

you can directly use an instance of `Account` to invoke methods of `AccountDSL`:

```
scala> acc1 belongsTo("David P.")
```

It's like all methods of `AccountDSL` have been injected into the class `Account`. Sounds familiar? It's similar to what we do with Ruby monkey-patching that allows you to split open any class and extend it with additional methods.

But there is a difference. In Scala, `implicits` are *lexically scoped*. The automatic conversion between `Account` and `AccountDSL` will be available only in the lexical scope of the method `enrichAccount`. Quite unlike Ruby *open classes* that allow modifications of existing classes on a global scope.

Now that you have transparent conversion from `Account` to `AccountDSL`, you can use the above DSL APIs on `Account` instances. Let's create some instances of our Java class `Account`:

```
val acc1 = new Account("acc-1", "David P.")
val acc2 = new Account("acc-2", "John S.")
val acc3 = new Account("acc-3", "Fried T.")
```

And add a few more account holder names to acc1 using the new operator <<:

```
acc1 << "Mary R." << "Shawn P." << "John S."
```

Note how concise yet expressive the above snippet looks as opposed to what we would have done with our original Java APIs:

[3] This defines an automatic conversion from Account to AccountDSL

```
acc1.addName("Mary R.").addName("Shawn P.").addName("John S.");
```

Let's form a collection of accounts and print the `firstName`s of those accounts which feature "John S." as one of the owners:

```
val accounts = List(acc1, acc2, acc3)
accounts filter(_ belongsTo "John S.") map(_ getFirstName) foreach(println)
```

Expressive indeed! In fact, it is much more expressive than what you would get with our original Java APIs. The reason is the richness of Scala as a language that helps you craft out rich semantics within a reduced surface area of the API. In the above snippet, we use combinators like `filter`, `map`, and `foreach` that operate on higher order functions making code much more concise than what you would get with an imperative Java syntax. Are you having fun? Let's party a bit more!

Get the list of accounts belonging to "John S." and compute the sum of `accruedInterest` for all such accounts for which the accumulated interest is greater than a predefined `threshold`:

```
accounts.filter(_ belongsTo "John S.")
  .map(_.calculate(new CalculatorImpl))
  .filter(_ > threshold)
  .foldLeft(0f)(_ + _)
```

This snippet contains applications of the Scala idiom that I explained in an earlier sidebar. The "_" is a placeholder for the type-inferenced argument that the predicate in `filter` takes as input. Note how the above snippet expresses the domain problem with more clarity than what you would have gotten out of a language that has more verbose syntax, like Java. It's all due to the richness in abstraction design that a more powerful language like Scala offers you by reducing the *accidental complexity* of your code.

Note the `CalculatorImpl` object that `calculate()` takes as input. We define `Calculator` as an interface in Java with `CalculatorImpl` as its implementation:

```
public interface Calculator {
    float calculate(Account account);
}

public class CalculatorImpl implements Calculator {

    @Override
    public float calculate(Account account) {
        //.. implementation
    }
}
```

In most of the cases you'll have the same implementation of `Calculator` interface being passed into `Account#calculate()` method. One way of avoiding this repetition is to use dependency injection to inject the implementation dynamically during runtime. Scala offers a better alternative where you can make this parameter *implicit* in all calls of `calculate`.

```
class AccountDSL(value: Account) {

//.. as above

  def calculateInterest(
    implicit calc: Calculator): Float = {  #1
    value.calculate(calc)
  }
}
```

#1 Calculator passed as an implicit parameter

You define an `implicit` argument to the method `calculateInterest` and have the `implicit` default set up in the scope of execution of the DSL #1. You now have an implicit default value for the implementation of `Calculator`, which you need not pass repeatedly to invocations of `calculateInterest`[4]. Have a look at the final version of the above calculation of accrued interest for all accounts belonging to "John S.":

```
implicit val calc = new CalculatorImpl

accounts.filter(_ belongsTo "John S.")
  .map(_.calculateInterest)
  .filter(_> threshold)
  .foldLeft(0f)(_ + _)
```

With support for features like closures and higher order functions, Scala offers you the power to define control abstractions that look like language built-in syntax. Using Java objects as underlying implementations, you can design powerful control constructs that make your DSL succinct and expressive.

It's not that a nonprogramming domain expert will be able to program in any DSL that you design. It's the explicit communicability of the API that matters for a well designed DSL. In the above snippet you will notice functional combinators like `map`, `filter` and `foldLeft` that do not qualify as being very meaningful to the domain person. But, what the domain person will be able to figure out easily from the above snippet are the hotspots like:

- Filtering the account belonging to "John S."
- `calculateInterest` on it
- Filtering only those that are > the threshold value
- Adding the interest values up

And when you offer all these hotspots within a localized surface area of the code base, it becomes easier for the domain expert to comprehend and verify the business logic. With an imperative approach, the same logic would have been spread across a larger code segment, making it much more difficult for someone who doesn't know programming.

Let's define one such control abstraction using our `Account` Java object and the `AccountDSL` that we implemented in Listing 5.

```
object AccountDSL {
  def withAccount(trade: Trade)(operation: Account => Unit) = {
    val account = trade.account
    //.. initialization
      try {
        operation(account)
      } finally {
        //.. clean up
      }
  }
}
```

Note that the underlying abstractions used in the above snippet, `Account` and `Trade` are Java abstractions that may have been enriched using Scala wrappers. Now it's your DSL users' turn to use such abstractions to perform useful domain operations. Have a look at the following DSL code fragment that's possible using the earlier control abstraction and wrapper-based integration of Scala and Java. It's so much more expressive and closer to the domain syntax than what would have been possible with a *Java-only* paradigm.

```
withAccount(trade) {
  account => {
    settle(
```

[4] This is yet another use of `implicit` modifier in addition to the one we discussed in an earlier sidebar

```
      trade using
        account.getClient
               .getStandingRules
               .filter(_.account == account)
               .first)
    andThen journalize
  }
}
```

This sequence of API invocation does the following:

- Takes an account
- Finds its client
- Fetches all standing rules for the client
- Gets the one that matches the account (a client can have multiple standing rules)
- Does settlement for the trade using the standing rule
- Journalizes the trade in the book of accounts

And, it does all these in only a few lines of clear domain specific code. Show this snippet to your domain expert. I am sure he will be able to explain to you what it does. I did the same thing with one of the Bobs of our project team. And can you imagine what happened? Bob gleaned over the snippet and here's the conversation that followed:

> *Bob:* You are picking up the first of the selected standing rules after filtering – right?
>
> *Me:* Yeah!
>
> *Bob:* But, in many cases there may be multiple matches for the same account.
>
> *Me:* How do you then decide which rule to pick up?
>
> *Bob:* In those cases, every rule has a priority tagged onto it, and you need to pick up the one with the highest priority.
>
> *Me:* Great!

The next time your manager talks about up front big investment for DSL implementation, narrate this small anecdote to him. It's a myth that every integration effort with a DSL needs a big upfront investment. The wrapper technique that we discussed in this article is a real-life testimony to this. It actually builds upon your current investment of Java domain model and makes it smarter and closer to the domain experts.

Now, you have seen yet another technique of integrating DSLs into your application using a statically typed language like Scala. We used dynamically typed Groovy. This shows the power of JVM as a platform for integrating multiple languages of different paradigms. In the next two subtopics, we will see the two remaining techniques from Figure 1. When you design DSLs for your next application, you can pick and choose from all these techniques. Just ensure that you use the right tool for the right job.

Language-specific integration features

Let's revisit our original order processing DSL that we integrated with our core Java application. In this subtopic, we'll use an integration technique that's recommended more often by the practitioners of Groovy as more idiomatic.

Suppose that in your trading application you've used Java 6 `ScriptEngine` to integrate your order processing DSL with the Java application. Things were running fine. But one day the client came back with a requirement that needs additional processing to be done on the collection of orders that the script returns to your Java application. His specific need was to compute the total valuation of all orders that have been placed so far, along with some custom displays of order attributes for the customer.

In your earlier approach, you loaded the DSL implementation (`ClientOrder.groovy`) as well as the user defined script (`order.dsl`) as a single Groovy script to be executed within the sandbox of the `ScriptEngine`. This makes the Groovy DSL completely opaque to the Java code – the script was being loaded using a different

classloader than your Java classes, thereby making them completely invisible within your main application. You need to buy some time from your client and try to implement alternate forms of integration of the DSL that make Groovy classes more visible to your Java application. In case you haven't yet figured out the solution, read on. I have something for you in the following paragraphs.

In this subtopic, we use an approach whereby we treat Groovy classes as reusable abstractions within your Java application and use `GroovyClassLoader` to load only the order processing script that your user writes. Listing 6 makes the necessary changes that you need to do to make things more Groovyish.

Listing 6 RunScript.java: DSL integration using GroovyClassLoader

```
public class RunScript {
    public static void main(String[] args)
            throws CompilationFailedException, IOException,
                InstantiationException, IllegalAccessException {

        final ClientOrder clientOrder =
                new ClientOrder();

        clientOrder.run();  #1

        final Closure dsl =   #2
            (Closure)((Script) new GroovyClassLoader().parseClass(
                new File("order.dsl")).newInstance()).run();

        dsl.setDelegate(clientOrder);   #3
        final Object result = dsl.call();  #4

        List<Order> r = (List<Order>) result;
        int val = 0;
        for(Order x : r) {   #5
            val += (Integer)(x.getValue());
        }
        System.out.println(val);
    }
}
```

Let's recap what the above listing offers you to make the DSL integration more meaningful in Groovy. We have separated the abstraction `ClientOrder.groovy` and precompiled it to make the `Order` class available to the Java application. Within the Java class, we execute an instance of `ClientOrder` to do the meta-class setup (#1). The DSL script `order.dsl` now returns a `Closure` containing the DSL code (#2). We set up `ClientOrder` as the delegate of the `Closure` to resolve the symbols that the script uses (#3). We call the DSL script to return a list of `Order` objects (#4). And, finally, we can find out the total order valuation by iterating over individual orders (#5).

Note that in the above listing, once we come out of the execution of the DSL script, we get back a list of `Order` objects, which we can use to do further business processing (#5). You couldn't have done this in the earlier case when we used Java 6 scripting API to integrate our DSL with Java application. Now your client is happy, and you have also learned a new way to integrate your Groovy DSL into your Java application.

Here's the DSL script `order.dsl` in Listing 7, changed to return a `Closure` back to the Java application.

Listing 7 order.dsl : The DSL script now returns a Closure

```
{->
orders = []
ord1 =
newOrder.to.buy(100.shares.of('IBM')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << ord1

ord2 =
newOrder.to.buy(150.shares.of('GOOG')) {
```

```
  limitPrice    300
  allOrNone     true
  valueAs       {qty, unitPrice -> qty * unitPrice - 500}
}
orders << ord2

ord3 =
newOrder.to.buy(200.shares.of('MSOFT')) {
  limitPrice    300
  allOrNone     true
  valueAs       {qty, unitPrice -> qty * unitPrice - 500}
}
orders << ord3

println "Orders ..."
orders.each { println it }
}
```

We have been following an iterative approach towards incremental refinement of DSL design. Both of these implementations use the `ExpandoMetaClass` feature of Groovy meta-programming.

Like Groovy, similar language specific strategies are offered by JRuby. Within a Java program, you can create instances of `RubyRuntimeAdapter` and use `eval` to directly evaluate JRuby scripts. Calling Java code from JRuby is quite easy and has been demonstrated in many blog posts and articles. But, when your core application is written in Java and you would like to reach out to Ruby for integrating a DSL that you've implemented, you need the interoperability in the other direction. And this is not a path that's been well explored by practitioners. Using Java 6 scripting API is one viable option, but, as I mentioned earlier, the usage is a bit clunky and looks unwieldy when compared to the elegance of the natural Ruby idioms.

Moving on from language-specific integration, let's now explore a framework-based integration of internal DSLs. Spring offers a suitable platform here; the next subtopic shows how.

Spring-based Integration

We have come to the final piece of integration technique that we summarized in Table 3.1. It's one level up in abstraction level, since it offers integration through a framework as opposed to language level ones that we discussed earlier. How often have you wondered how helpful it would be if some of the business rules that you implemented in Java could have been changed dynamically without having to restart your application? When you design the domain model of an application, you always try to make it maximally expressive, so that your domain expert can go through the implementation and verify for its correctness and completeness. This is, after all, the basic motivation for doing DSL-driven development.

Since 2.0, Spring[5] supports bean implementation using expressive dynamic languages like Ruby and Groovy and offers the feature of making them *refreshable*. A refreshable bean is one that allows itself to reload dynamically when it's underlying implementation changes. Let's consider an example from our domain where an implementation of a `TradingService` needs to look up rules for computing the accrued interest on Coupon Bonds.

```
public class TradingServiceImpl implements TradingService {
    private AccruedInterestCalculationRule accIntRule;  #1

    @Override
    public void doTrade(Trade trade) {
        // .. implementation
    }
}
```

#1 Calculation rule to be injected by Spring

[5] http://www.springframework.org

In the above snippet, the business rules for accrued interest calculation will be injected through dependency injection using Spring at runtime. Using dynamic language support that Spring offers, you can implement these rules using expressive languages like JRuby or Groovy or Jython. This is an area where a small rich DSL makes a great fit. And the benefits that you get are twofold:

- More expressive code because the languages themselves are more rich
- The ability to auto-reload the runtime instance of the bean when the underlying implementation changes

In the current example we can have a Java interface for the rule contract:

```
public interface AccruedInterestCalculationRule {
    BigDecimal calculate(Trade trade);
}
```

While the backing implementation of the rule can be done using a DSL written in Ruby:

```
require 'java'

class RubyAccruedInterestCalculationRule {
  def calculate(trade)
    //.. implementation
  end
end

RubyAccruedInterestCalculationRule.new
```

You can then wire up the whole implementation using the following XML configuration snippet in Spring. Now, when you ask for an instance of `AccruedInterestCalculationRule` from within your Java program, you get an instance implemented using the Ruby DSL. Using Spring you have successfully integrated a Ruby DSL into your Java application.

```
<lang:jruby
  id="accIntCalcRule"
  refresh-check-delay="5000"
  script-interfaces=
    "org.springframework.scripting.AccruedInterestCalculationRule "
  script-source="classpath:RubyAccruedInterestCalculationRule.rb">
</lang:jruby>
```

This model of DSL integration is non-intrusive and keeps the DSL component decoupled from the context where it is used. If you're using Spring as the backbone of your application, you can consider this as an option to implement dynamic reloading of business rule DSLs.

Now that you've seen the homogeneous integration patterns related to internal DSLs, let's look at some of the patterns for integrating external DSLs. External DSLs can be of any form and may be implemented using custom language infrastructure. In the next subtopic, we'll discuss the external DSL implementation and see how each of them publishes explicit integration points for your core application. Note that external DSLs are custom made for specific applications only. Thus our discussion on integration patterns for external DSLs will be limited to a few generic techniques that are used by practitioners today.

Heterogeneous integration patterns with external DSLs

How do you integrate XML with your application? You will scream out *"Using the XML parser!"* Correct! Because XML is not part of the host language that you are using for implementing the application, you need separate machinery to parse and process XML. XML is so commonly used that you get a slew of tools like XPath, XQuery, and a huge number of XML parsers bundled with almost every enterprise solution that you get today. Hence, integrating XML with your application is a no-brainer. Unfortunately, the external DSL that you'll design for your

application is not lucky enough to inherit a rich repertoire of such tools. Integrating your external DSL with your application is often found to rely on specific techniques that cannot be generalized as a pattern.

Judging from what I said in the last paragraph, you must be thinking of integrating external DSLs as a nightmare in the application development lifecycle. It all depends upon the complexity of the DSL and the technique that you used to develop it. If you use standard tools like ANTLR or YACC to develop parsers of your external DSL, integration is often quite straightforward.

Let's have a look at the various patterns of external DSL and try to find out the integration points that each of them publish. Table 2 lists a summary of thoughts of how you would integrate external DSLs with your core application.

Table 2 Integration points published by external DSLs

External DSL Pattern	Published Integration Point
Context driven string manipulation	▪ Converted to host language through tokenization process using techniques like regular expression matching and dynamic code evaluation ▪ The resultant code snippet is the integration point with the application
Transforming XML to consumable resource	▪ XML parsers are the most natural form of integration points ▪ After parsing, XML is converted to host language data structures, which can be directly used by the application
Non textual representation	▪ The nontextual representation is converted to the abstract syntax tree ▪ The abstract syntax tree can be used as the basis for generating multiple forms of concrete syntax trees ▪ One of the concrete syntax trees can be targeted to generate the host language of the application, which becomes the integration point
Mixing DSL with embedded foreign code	▪ The DSL processing engine transforms the DSL into appropriate data structures in the language of the embedded code and plugs in the embedded code snippets as callbacks ▪ The result is a set of data structures in the embedded code that can be directly used in the core application using the same language
Parser combinator based DSL design	▪ Parser combinators are implemented as a library in languages like Scala ▪ The rules that you write to parse the external DSL are combinators in the host language ▪ Using embedded host language snippets you build up data structures that get populated by

the rules and ultimately leads to the making of the language tree for you to use in your application

You must be wondering why we did not go into the details of integration patterns for external DSLs like the ones we did for internal DSLs. Internal DSL integration takes place through a host language, while external DSLs often require a more elaborate stack that depends on the specific application domain. It's difficult to have a generic discussion of external DSL integration patterns without looking into the specific details of what it needs to achieve.

That was quite a journey visiting all integration patterns for internal and external DSLs. You must now be feeling quite comfortable handling problems of DSL integration with your core application. Language design and implementation is always an evolutionary process. The idea was to demonstrate through examples and usability patterns the variations that you can come across in your pragmatic approach towards DSL building.

Summary

In this article, we discussed aspects related to integrating DSLs within your application environment. You've learned the various factors that you need to consider before choosing the underlying language of your DSL. We concentrated mostly on the JVM as a platform for DSL implementation. And, on the JVM, there are multiple languages that can be used to design expressive DSLs. But, when it comes to integration with your core application, choose the language that offers the most flexible endpoints. In doing so, your DSL will never feel like an alien piece bolted on to your application. It will feel like a natural extension of the underlying language.