# 進化するアーキテクチャーと新方式の設計: Groovy で DSL を作成する

## より表現の豊かな言語でイディオムのようなパターンを抽出する

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2010年 8月 17日

内部ドメイン特化言語 (DSL) を Java™ 言語で作成することは可能ですが、Java 言語の構文は制限されているため、かなり面倒です。内部 DSL を作成するには、JVM をベースにした他の言語のほうが適しています。「進化するアーキテクチャーと新方式の設計」の今回の記事では、Groovy を使って内部 DSL を作成するときに活用できる機能、そして直面すると思われる問題について説明します。

このシリーズの他の記事を見る

先月の記事では、コードの中で共通の設計イディオムとして定義されたイディオムのようなパターンを、ドメイン特化言語 (DSL) を使用して抽出する方法を具体的な例で説明しました (イディオムのようなパターンの概念については、「Composed Method と SLAP」で紹介しています)。DSL は宣言型であること、「通常」のソース・コードよりも読み易いこと、そして抽出したパターンを周囲のコードと差別化できることから、パターンを捉える有効な手段となります。

DSL を作成するための言語手法では、巧みなトリックを頻繁に使用して、暗黙的にコンテキストの中にコードをラッピングします。別の言葉で言い換えると、DSL はコードを読み易くするために、ベースとなる言語の機能を使用してノイズの多い構文をなるべく「隠そうとする」ということです。DSL は Java 言語でも作成できますが、この言語にはコンテキストを隠すための構成要素が限られていること、しかもその構文には柔軟性がなく厳格であることから、DSL を作成するには適していません。けれども他の JVM ベースの言語であれば、この不足を補うことができます。今回から 2 回にわたり、皆さんの DSL 作成ツールの幅を広げるために Java プラットフォーム上で実行される、より表現の豊かな言語で DSL を作成する方法を説明します。今回その第一弾として、Groovy (「参考文献」を参照) を取り上げます。

### このシリーズについて

このシリーズの目的は、ソフトウェアのアーキテクチャーと設計という、繰り返し議論されていながら捉えどころのない概念を新しい視点で捉えなおすことです。Neal Ford が示す具体的な例をとおして、進化するアーキテクチャーと新方式の設計におけるアジャイル・プラクティスの確固たる基礎を学びます。アーキテクチャーと設計に関する重要な決定事項を最

> 終的に必要な瞬間まで遅らせることで、アーキテクチャーと設計が必要以上に複雑にならないようにし、ソフトウェア・プロジェクトが強固なものでなくなる事態を避けることができます。

Groovy には DSL を容易に作成できるようにするいくつもの機能が備わっています。その 1 つ、数量のサポートは DSL では一般的な要件です。7 インチ、4 マイル、13 日間など、何事にも数量を必要とするのは人の常ですが、Groovy ではオープン・クラスによって数量のサポートを直接追加できるようになっています。オープン・クラスでは、既存のクラスを再オープンし、メソッドを追加、削除、あるいは変更することによってそのクラスに変更を加えることができます。これは、強力でもあり、危険でもあるメカニズムですが、幸い、この威力を安全に実装するいくつかの手段があります。それが、Groovy がオープン・クラスでサポートする 2 種類の構文すなわち、カテゴリーと `ExpandoMetaClass` です。

## カテゴリーを手段としたオープン・クラス

カテゴリーの概念は、Smalltalk や Objective-C (「参考文献」を参照) などの言語から拝借したもので、コードの呼び出しを包含するラッパーを作成し、`use` ブロック・ディレクティブを使用して、その中に 1 つ以上のオープン・クラスを含めるという仕組みです。

カテゴリーを理解するには、実例を見るのが一番の近道です。そこで、`String` に追加された `camelize()` という新しいメソッドを実際に実行するテストをリスト 1 に記載します。このメソッドは、アンダーバーで区切られたストリングをキャメル・ケースに変換します。

### リスト 1. `camelize()` メソッドを実際に実行するテスト

```
class TestStringCategory extends GroovyTestCase {
    def expected = ["event_map" : "eventMap",
             "name" : "name", "test_date" : "testDate",
             "test_string_with_lots_of_breaks" : "testStringWithLotsOfBreaks",
             "String_that_has_init_cap" : "stringThatHasInitCap" ]

    void test_Camelize() {
        use (StringCategory) {
            expected.each { key, value ->
                assertEquals value, key.camelize()
            }
        }
    }
}
```

リスト 1 では、変換前と変換後の大/小文字からなる `expected` という名前のハッシュを作成した後、マップの繰り返し処理を `StringCategory` でラップし、キーのそれぞれがキャメル・ケースに変換されるようにします。`use` ブロック内では、クラスで新しいメソッドを呼び出すための特別な操作は一切必要ないことに注目してください。

`StringCategory` のコードはリスト 2 のとおりです。

## リスト 2. `StringCategory` クラス

```
class StringCategory {

  static String camelize(String self) {
    def newName = self.split("_").collect() {
      it.substring(0, 1).toUpperCase() +  it.substring(1, it.length())
    }.join()
    newName.substring(0, 1).toLowerCase() +  newName.substring(1, newName.length())
  }
}
```

カテゴリーは静的メソッドが含まれる通常のクラスです。この静的メソッドには、引数が少なくとも 1 つなければなりません。その引数はメソッドを実行する皆さんによって渡されるものです。リスト 2 では、メソッドの追加先クラスを表す `String` (従来から `self` という名前が付けられていますが、好きな名前を付けて構いません) を引数として取る 1 つの静的メソッドを宣言しています。メソッド本体に含まれる Groovy コードが、ストリングをアンダーバーで区切って塊に分割し (`split("_")` メソッドによって実行されます)、それからストリングを再び 1 つにまとめ、大文字を適所に配置した上で結合します。そして最後の行では、返されるストリングの最初の文字が確実に小文字になるようにします。

`StringCategory` を使用するときには、`use` ブロック内でアクセスする必要があります。`use` ブロックの括弧内では、複数のカテゴリー・クラスをカンマで区切って使用することもできます。

DSL でオープン・クラスを使用して数量を表現するもう 1 つの例として、リスト 3 のコードを見てください。これは、単純な予約カレンダーを実装するコードです。

## リスト 3. 単純なカレンダー DSL

```
def calendar = new AppointmentCalendar()

use (IntegerWithTimeSupport) {
    calendar.add new Appointment("Dentist").from(4.pm)
    calendar.add new Appointment("Conference call")
                 .from(5.pm)
                 .to(6.pm)
                 .at("555-123-4321")
}
calendar.print()
```

リスト 3 で実装している機能は、「流れるようなインターフェース」で使用した Java の例と同様ですが、ここでは構文が拡張され、Java コードでは不可能ないくつかの工夫が盛り込まれています。例えば、Groovy では場所によっては括弧を省略できることに注目してください (`add()` メソッドへの引数を囲む括弧など)。また、Java 開発者には奇異に見える、`5.pm` のような呼び出しを行うこともできます。これは `Integer` クラスをオープンして (Groovy ではすべての数値が自動的に型ラッパー・クラスを使用するため、実際には `5` でさえも `Integer` ということになります)、`pm` プロパティーを追加する一例です。リスト 4 に、このオープン・クラスを実装するクラスを記載します。

## リスト 4. `IntegerWithTimeSupport` クラス定義

```
class IntegerWithTimeSupport {
    static Calendar getFromToday(Integer self) {
        def target = Calendar.instance
        target.roll(Calendar.DAY_OF_MONTH, self)
        return target
    }

    static Integer getAm(Integer self) {
        self == 12 ? 0 : self
    }

    static Integer getPm(Integer self) {
        self == 12 ? 12 : self + 12
    }
}
```

このカテゴリー・クラスには、`getFromToday()`、`getAm()`、`getPm()` という 3 つの `Integer` 用の新しいメソッドがありますが、実際にはメソッドではなく、新しいプロパティーとして組み込まれていることに注意してください。これらのメソッドを新規プロパティーとして作成した理由は、Groovy でのメソッド呼び出し方法に関係します。引数を持たない Groovy メソッドを呼び出す場合には、一組の空の括弧を使って呼び出さなければなりません。この括弧によって、Groovy は、プロパティーへのアクセスであるのか、メソッド呼び出しであるのかを区別できるためです。上記の拡張をメソッドとして作成したとすると、DSL は `am` 拡張と `pm` 拡張を `5.pm()` のようにして呼び出さなければなりません。それでは、DSL の読み易さが損なわれてしまいます。DSL を使用する主な理由の 1 つは、読み易さを改善することです。したがって、余計なノイズは排除するに越したことはありません。Groovy では、拡張をプロパティーとして作成することによってノイズを排除することができます。プロパティーを宣言するための構文には、Java 言語の場合と同じく `get`/`set` メソッドのペアを使用しますが、これらのメソッドは括弧を使わずに呼び出すことができます。

この DSL では単位として「時」を使っているので、例えば `3.pm` の場合には `15` を返さなければなりません。数量を扱う DSL を作成する際には、どの単位を使用するかを決め、(オプションで) その単位を DSL に追加して読み易くしてください。DSL を使用する目的は、領域特有のイディオムのようなパターンを抽出することです。つまり、開発者でなくても、読めるようにするということです。

カレンダー DSL に時刻を実装する方法がわかれば、`Appointment` クラス (リスト 5 を参照) の内容はすぐに理解できるはずです。

## リスト 5. `Appointment` クラス

```
class Appointment {
  def name;
  def location;
  def date;
  def startTime;
  def endTime;

  Appointment(apptName) {
    name = apptName
    date = Calendar.instance
  }

  def at(loc)  {
```

```
    location = loc
    this
  }

  def formatTime(time) {
    time > 12 ? "${time - 12} PM" : "${time} AM"
  }

  def getStartTime() {
    formatTime(startTime)
  }

  def getEndTime() {
    formatTime(endTime)
  }

  def from(start_time) {
    startTime = start_time
    date.set(Calendar.HOUR_OF_DAY, start_time)
    this
  }

  def to(end_time) {
    endTime = end_time
    date.set(Calendar.HOUR_OF_DAY, end_time)
    this
  }

  def display() {
    print "Appointment: ${name}, Starts: ${formatTime(startTime)}"
    if (endTime) print ", Ends: ${formatTime(endTime)}"
    if (location) print ", Location: ${location}"
    println()
  }
}
```

Groovy についての知識がまったくないとしても、上記の `Appointment` クラスは問題なく読むことができるはずです。Groovy では、メソッドの最後の行はそのメソッドの戻り値であることに注意してください。つまり、流れるようなインターフェースがこのクラスで呼び出すのは、`at()`、`from()`、および `to()` メソッドの最後の行 (`this` の戻り値) ということになります。

このように、カテゴリーを使用することで、既存のクラスを制御された方法で変更することができます。それは、カテゴリーでは変更のスコープが `use()` 節で定義されたレキシカル・ブロックに厳格に制限されるためですが、その一方で、オープン・クラスで追加するメソッドに広範なスコープを持たせたい場合もあります。そのような場合に役立つのが、Groovy の `ExpandoMetaClass` です。

## Expando を手段としたオープン・クラス

Groovy のオープン・クラスの構文には当初、カテゴリーしか使用されませんでした。けれども Groovy の Web フレームワークである Grails (「参考文献」を参照) の作成者たちは、カテゴリーに固有のスコープ設定ではあまりにも制約があり過ぎると判断しました。そこで代わりに開発されたオープン・クラスの構文が、`ExpandoMetaClass` です。Expando を使用する場合には、クラスのメタクラス (Groovy が日和見的に自動作成) にアクセスして、そこにプロパティーとメソッドを追加します。リスト 6 は、Expando を使用した場合のカレンダーの例です。

### リスト 6. Expando オープン・クラスを使用したカレンダー

```
def calendar = new AppointmentCalendar()

calendar.add new Appointment("Dentist")
             .from(4.pm)
calendar.add new Appointment("Conference call")
             .from(5.pm)
             .to(6.pm)
             .at("555-123-4321")

calendar.print()
```

リスト 6 のコードは、カテゴリーに必要な use ブロックがないだけで、リスト 3 のコードとほとんど同じように見えます。ただし、`Integer` に対する変更を実装するために、メタクラスにアクセスするという点が異なります (リスト 7 を参照)。

### リスト 7. `Integer` の Expando 定義

```
Integer.metaClass.getAm = { ->
  delegate == 12 ? 0 : delegate
}

Integer.metaClass.getPm = { ->
  delegate == 12 ? 12 : delegate + 12
}

Integer.metaClass.getFromToday = { ->
  def target = Calendar.instance
  target.roll(Calendar.DAY_OF_MONTH, delegate)
  target
}
```

カテゴリーの場合の例と同じく、`Integer` での `am` と `pm` はメソッドではなくプロパティーにする必要があります (呼び出すときに括弧を使ってアクセスする必要をなくすため)。そこでメタクラスには、新しいプロパティーを `Integer.metaClass.getAm` として追加しています。これらのコード・ブロックでは引数を取ることもできますが、ここでは引数は不要です (そのため、コード・ブロックの先頭には `->` が単独で存在します)。コード・ブロック内では、`delegate` キーワードがメソッドを追加する対象となるクラスのインスタンスを参照します。例えば `getFromToday` プロパティーの中では、新しい `Calendar` インスタンスを作成した後、`Integer` のインスタンスが指定する日数分カレンダーを進めるために delegate の値を使用します。つまり、`5.fromToday` を実行すると、カレンダーは 5 日先に進むことになります。

## カテゴリーまたは Expando の選択

カテゴリーと Expando が同じような表現力を提供するのであれば、一体どちらを選べばよいのでしょうか。カテゴリーの最大の利点は、レキシカル・ブロックに制限されたその固有のスコープです。言語のコア・クラスに抜本的な (そしておそらく破壊的な) 変更を加えるのは、よくある DSL のアンチパターンとなっていますが、カテゴリーを使用すれば、変更に制限が設けられます。一方、Expando は本質的にグローバルであり、Expando コードを実行すると、その変更はアプリケーションの残りの部分にも適用されます。

通常は、カテゴリーを使用するようにしてください。副次作用が考えられる変更を重要なクラスに対して行う場合には、変更のスコープを制限しなければなりませんが、カテゴリーを使用すれ

ば変更のスコープを絞り込むことができます。ただし、同じカテゴリーを使っていくつものコードをラップしていることに気付いたら、Expando に切り替えてください。変更の内容によっては、広範囲に適用する必要があります。そのような変更をブロック内に無理矢理収めるのでは、複雑なコードになりかねません。大まかな目安としては、4 つ以上の個別のコードの塊をカテゴリーでラップしているとしたら、そのコードを Expando にすることを検討してください。

最後の注意事項として、テストはオプションではありません。多くの開発者がサイズの大きなコードでのテストはオプションであると考えているようですが、既存のクラスに変更を加えるコードには例外なく、包括的なテストが必要です。コア・クラスを変更する機能は強力であり、問題に対する見事なソリューションをもたらすはずですが、その威力を使用する場合にはテストを行うという責任を伴います。

## 実際の事例

イディオムのようなパターンを抽出する手段としての DSL についての説明は、これまで多少抽象的だったので、説明の締めくくりとして実際の例を紹介したいと思います。

Groovy ベースのビヘイビア駆動開発テスト・ツールである easyb (「参考文献」を参照) では、開発者でない人にも馴染みのある文体による説明と、テストを実装するためのコードとを組み合わせたシナリオを作成することができます。一例として、リスト 8 に easyb のシナリオを記載します。

### リスト 8. キューをテストする easyb のシナリオ

```
package org.easyb.bdd.specification.queue

import org.easyb.bdd.Queue

description "This is how a Queue must work"

before "initialize the queue for each spec", {
    queue = new Queue()
}

it "should dequeue item just enqueued", {
    queue.enqueue(2)
    queue.dequeue().shouldBe(2)
}

it "should throw an exception when null is enqueued", {
    ensureThrows(RuntimeException.class) {
        queue.enqueue(null)
    }
}

it "should dequeue items in same order enqueued", {
    [1..5].each {val ->
        queue.enqueue(val)
    }
    [1..5].each {val ->
        queue.dequeue().shouldBe(val)
    }
}
```

リスト 8 のコードは、キューに対する適切な振る舞いを定義しています。定義ブロックのそれぞれが `it` で始まり、その後にストリングの説明とコード・ブロックが続きます。`it` のメソッド定

義では以下のように、`spec` がテストについて説明し、`closure` がコード・ブロックを保持するようになっています。

```
def it(spec, closure)
```

リスト 8 の最後のテストでは、`dequeue()` の呼び出しから返される値を検証するために、以下のコード行を使用している点に注意してください。

```
queue.dequeue().shouldBe(val)
```

しかし `Queue` クラスを調べても、`shouldBe()` メソッドはこのクラスのどこにもありません。このメソッドは一体どこから来ているのでしょうか。

`it()` メソッドの定義を見ると、既存のクラスを増補するためにどこでカテゴリーが使用されているのかがわかります。リスト 9 に、`it()` メソッドの定義を記載します。

## リスト 9. `it()` メソッドの定義

```
def it(spec, closure) {
    stepStack.startStep(listener, BehaviorStepType.IT, spec)
    closure.delegate = new EnsuringDelegate()
    try {
        if (beforeIt != null) {
            beforeIt()
        }
        listener.gotResult(new Result(Result.SUCCEEDED))
    use(BehaviorCategory) {
()
        }
        if (afterIt != null) {
            afterIt()
        }
    } catch (Throwable ex) {
        listener.gotResult(new Result(ex))
    }
    stepStack.stopStep(listener)
}
```

メソッドの中間あたりにある `BehaviorCategory` クラス内で、パラメーターとして渡された closure ブロックが実行されています。リスト 10 に、このクラスの一部を抜粋します。

## リスト 10. `BehaviorCategory` クラスからの抜粋

```
static void shouldBe(Object self, value, String msg) {
    isEqual(self, value, msg)
}

private static void isEqual(self, value, String msg) {
    if (self.getClass() == NullObject.class) {
        if (value != null) {
            throwValidationException(
                "expected ${value.toString()} but target object is null", msg)
        }
    } else if (value.getClass() == String.class) {
        if (!value.toString().equals(self.toString())) {
            throwValidationException(
                "expected ${value.toString()} but was ${self.toString()}", msg)
        }
```

```
    } else {
        if (value != self) {
            throwValidationException("expected ${value} but was ${self}", msg)
        }
    }
}
```

`BehaviorCategory` は、`Object` を増補するメソッドが含まれるカテゴリーです。このカテゴリーが、オープン・クラスの驚くほどの威力を明らかにしています。新しいメソッドを `Object` に追加することで、アプリケーション内のあらゆるインスタンスが追加メソッドにアクセスできるようになります。そのため、すべてのクラス (`Queue` を含む) にいとも簡単に `shouldBe()` メソッドを追加できるというわけです。この芸当は、コアの Java コードでは不可能であり、アスペクトを使用したとしても面倒なことになるでしょう。この例でのカテゴリーの使用は、私の前述のアドバイスを強固に裏付けます。つまり、カテゴリーを使用することによって、`Object` に対する変更のスコープを easyb DSL の `use` 節本体に制限できるということです。

## まとめ

イディオムのようなパターンを抽出したら、そのパターンをコードの他の部分からは際立たせたいものです。DSL は、この目標を達成するための強力なメカニズムを提供します。DSL を作成するには、Java 言語を使用するよりも、DSL を作成するためのサポートを備えた言語を使うほうが遥かに簡単です。組織での外部要因によって Java 以外の言語を利用できないとしても、諦めないでください。Spring フレームワークのようなツールでは、Groovy や Clojure (「参考文献」を参照) などの代替言語のサポートをますます強化してきています。Java を使えないのであれば、これらの代替言語を使ってコンポーネントを作成し、作成したコンポーネントを Spring によってアプリケーションの適切な場所に注入するという方法を採ることもできます。多くの組織では、代替言語に関して慎重になり過ぎていますが、Spring のようなフレームワークを使用して漸進的に進めるという簡単なやり方もあります。

次回の記事では、DSL を領域特有のイディオムのようなパターンを抽出するための方法として使用するという話題の締めくくりとして、JRuby を使用した場合の例をいくつか紹介し、どこまで表現の豊かな言語を使用できるかを説明します。

---

# 著者について

Neal Ford

Neal Ford は世界的な IT コンサルティング企業である ThoughtWorks のソフトウェア・アーキテクトであり、Meme Wrangler でもあります。また彼は、アプリケーション、教育資料、雑誌記事、コースウェア、ビデオや DVD によるプレゼンテーションなどの設計と開発も行っています。さまざまな技術に関する本の著者、編集者でもあり、最新の著書は『プロダクティブ・プログラマ — プログラマのための生産性向上術』です。彼は大規模なエンタープライズ・アプリケーションの設計や構築を専門にしています。また彼は世界各地で開催される開発者会議での講演者としても国際的に有名です。彼の Web サイトをご覧ください。