

進化するアーキテクチャーと新方式の設計: 流れるようなインターフェース

イディオムのような領域特有のパターンを抽出する内部 DSL を作成する

Neal Ford

Software Architect / Meme Wrangler
ThoughtWorks Inc.

2010年 7月 13日

シリーズ「[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、新方式の設計でイディオムのようなパターンを抽出する手法を前回に続いて説明します。再利用可能なパターンを識別すれば、コードの残りの部分とは別に、そのパターンを抽出できるはずです。ドメイン特化言語 (DSL) は、簡潔にデータと機能を抽出する数多くの手法を提供します。今月の記事で Neal Ford が紹介するのは、イディオムのようなドメイン・パターンを抽出する内部 DSL を作成する 3 つの方法です。

[このシリーズの他の記事を見る](#)

このシリーズの[前回の記事](#)で話題としたのは、ドメイン特化言語 (DSL) を使用して特定の領域でイディオムのようなパターンを抽出する方法です。今回はその続きとして、DSL を作成するさまざまな手法を具体的に説明します。

Martin Fowler は彼の近刊書『Domain Specific Languages』のなかで、DSL を 2 つのタイプに区別しています (「[参考文献](#)」を参照)。一方は、新しい言語の文法を作成する外部 DSL で、外部 DSL には lex と yacc、または Antlr のようなツールが必要となります。もう一方のタイプは、基本言語の上に新しい言語を作成する内部 DSL です。内部 DSL は基本言語の構文を借りて、それを定型化することによって新しい言語を作成します。この記事では、Java™ を基本言語として使用した内部 DSL を作成し、Java の構文をベースに新しいミニ言語を組み立てる例を紹介します。

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。

これから説明する DSL の作成手法はいずれも、暗黙的コンテキストの概念の上に成り立ちます。DSL (特に内部 DSL) は関連する要素を包含するコンテキスト・ラッパーを作成することによって、「ノイズ」の多い構文を排除しようと努めます。XML ではこの概念が、関連項目をラップする親要素と子要素という形で明確に表れています。この後わかるように、DSL 手法の多くは言語の構文を巧みに使用して、XML の親要素と子要素と同じ効果をもたらしています。

DSL を使用する利点の 1 つは、読み易さです。開発者でなくても読むことのできるコードを作成すれば、開発チームと機能の要求側とのやりとりのフィードバック・ループが短くなります。Fowler の著書で取り上げられている共通の DSL パターンに、「流れるようなインターフェース (fluent interface)」と名付けられたパターンがあります。彼はこのパターンを、一連のメソッド呼び出しに対する命令コンテキストを中継または維持することのできる振る舞いとして定義しています。流れるようなインターフェースにはいくつかのタイプがありますが、まずはそのうちの 1 つ、メソッド・チェーン (method chaining) について説明します。

メソッド・チェーン

メソッド・チェーンは、メソッドからの戻り値を使用して命令コンテキストを中継します。この場合、命令コンテキストは最初のメソッド呼び出しを行うオブジェクト・インスタンスです。こう説明すると実際よりも複雑そうに聞こえるので、具体的な例で、メソッド・チェーンの概念を明らかにしたいと思います。

DSL に取り組むときには、最初に目標とする構文から取り掛かり、それからリバース・エンジニアリングによって構文の実装方法を考えるのが一般的です。DSL では読み易さが極めて重視されるため、最終目標から取り掛かるという方法は理にかなっています。ここで引用する例は、カレンダーのエントリーを追跡する小さなアプリケーションです。このアプリケーションには、目標とする DSL の構文が具体的に示されています (リスト 1 を参照)。

リスト 1. カレンダー DSL で最終目標とする構文

```
public class CalendarDemoChained {

    public static void main(String[] args) {
        new CalendarDemoChained();
    }

    public CalendarDemoChained() {
        Calendar fourPM = Calendar.getInstance();
        fourPM.set(Calendar.HOUR_OF_DAY, 16);
        Calendar fivePM = Calendar.getInstance();
        fivePM.set(Calendar.HOUR_OF_DAY, 17);

        AppointmentCalendarChained calendar =
            new AppointmentCalendarChained();
        calendar.add("dentist").
            from(fourPM).
            to(fivePM).
            at("123 main street");

        calendar.add("birthday party").at(fourPM);
        displayAppointments(calendar);
    }

    private void displayAppointments(AppointmentCalendarChained calendar) {
        for (Appointment a : calendar.getAppointments())
            System.out.println(a.toString());
    }
}
```

```
}
```

上記のリストは、Java カレンダーを扱うために必要なコードで始まっています。続いて、2つのカレンダー・エントリーに値を追加していることから、メソッド・チェーン・タイプの流れるようなインターフェースを使用していることがわかります。1行のコードの (Java 構文から見た) 構成部分のそれぞれをホワイト・スペースで区切っている点に注目してください。このように、内部 DSL では DSL を読み易くするために、基本言語の使用法を定型化するのが通常です。

流れるようなインターフェースのメソッドの大部分は、リスト 2 に記載する `Appointment` クラスに含まれています。

リスト 2. `Appointment` クラス

```
public class Appointment {
    private String _name;
    private String _location;
    private Calendar _startTime;
    private Calendar _endTime;

    public Appointment(String name) {
        this._name = name;
    }

    public Appointment() {
    }

    public Appointment name(String name) {
        _name = name;
        return this;
    }
    public Appointment at(String location) {
        _location = location;
        return this;
    }

    public Appointment at(Calendar startTime) {
        _startTime = startTime;
        return this;
    }

    public Appointment from(Calendar startTime) {
        _startTime = startTime;
        return this;
    }

    public Appointment to(Calendar endTime) {
        _endTime = endTime;
        return this;
    }

    public String toString() {
        return "Appointment:" + _name +
            (( _location != null && _location.length() > 0 ) ?
              ", location:" + _location : "") +
            ", Start time:" + _startTime.get(Calendar.HOUR_OF_DAY) +
            ( _endTime != null ? ", End time: " +
              _endTime.get(Calendar.HOUR_OF_DAY) : "" );
    }
}
```

上記のリストを見るとわかるように、流れるようなインターフェースはごく簡単に作成することができます。ここではミューテーター・メソッドごとに、ホスト・オブジェクト (`this`) を返すセッター・メソッドを作成し、`set` 命名規則を読み易い規則に置き換えることによって、標準的な JavaBean 構文のパターンを変更しています。これで、このセクションの先頭にある汎用定義がより理解しやすくなるはずです。さらにメソッド・チェーンによって中継されるコンテキストは `this` なので、一連のメソッド呼び出しを一層簡潔にすることができます。

リスト 3 に、以前の記事「[Leveraging reusable code, Part 2](#)」で紹介した、車両を対象とした API 定義を記載します。

リスト 3. 車両の API

```
Car2 car = new CarImpl();
MarketingDescription desc = new MarketingDescriptionImpl();
desc.setType("Box");
desc.setSubType("Insulated");
desc.setAttribute("length", "50.5");
desc.setAttribute("ladder", "yes");
desc.setAttribute("lining type", "cork");
car.setDescription(desc);
```

車両の問題領域は、コンテンツと履歴の規定による複雑さにあります。この例の元となったプロジェクトでは、[リスト 3](#) のような `set` の呼び出しを何十行も要する多数の複雑なテスト・シナリオを使用していました。そこで、私たちは正しい属性の組み合わせを使っているかどうかをビジネス・アナリストに検証してもらおうとしましたが、彼らはこれらのテスト・シナリオを Java コードと見なし、読む気にならないとして退けました。この問題は最終的に、開発者が詳細を言葉に置き換えて説明するという結果になりましたが、それではもちろん誤りが起こり易く、時間もかかります。

この問題を解決するため、私たちは `car` クラスを流れるようなインターフェースに変換しました。その結果、[リスト 3](#) のコードはリスト 4 に記載する流れるようなインターフェースのコードになりました。

リスト 4. 車両のための流れるようなインターフェース

```
Car car = Car.describedAs()
    .box()
    .length(50.5)
    .type(Type.INSULATED)
    .includes(Equipment.LADDER)
    .lining(Lining.CORK);
```

上記のコードは十分に宣言型であり、Java API バージョンのコードにあったノイズも十分に除去されたことから、ビジネス・アナリストは快く検証を引き受けてくれました。

カレンダーの例に話を戻すと、実装の最後には `AppointmentCalendar` クラスがあります (リスト 5 を参照)。

リスト 5. AppointmentCalendar

```
public class AppointmentCalendarChained {
    private List<Appointment> appointments;

    public AppointmentCalendarChained() {
        appointments = new ArrayList<Appointment>();
    }

    public List<Appointment> getAppointments() {
        return appointments;
    }

    public Appointment add(String name) {
        Appointment appt = new Appointment(name);
        appointments.add(appt);

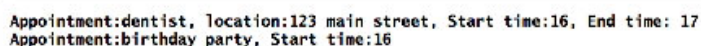
        return appt;
    }
}
```

add() メソッドは以下の処理を行います。

1. Appointment インスタンスを新規に作成して、メソッド・チェーンを開始します。
2. 新規インスタンスをアポイントメントのリストに追加します。
3. 最後に、新規に作成したアポイントメント・インスタンスを返します。これは、以降のメソッド呼び出しがこの新しいアポイントメントで行われることを意味します。

このアプリケーションを実行すると、構成したアポイントメントの詳細が表示されます (図 1 を参照)。

図 1. カレンダー・アプリケーションの実行結果



Appointment:dentist, location:123 main street, Start time:16, End time: 17
Appointment:birthday party, Start time:16

これまでのところ、宣言型のメソッド呼び出しをはじめ、冗長過ぎる構文を簡潔にするには、メソッド・チェーンが簡単な手段となりそうです。メソッド・チェーンは、新方式の設計におけるイディオムのようなパターンにも効果を発揮します。というのも領域特有のパターンはほとんどの場合、宣言型であるためです。

注意する点として、メソッド・チェーンを使用するとなると、JavaBeans の構文規則には違反せざるを得ません。JavaBeans の構文規則では、ミューテーター・メソッドが set で始まり、void を返すように規定しているためです。流れるようなインターフェースを作成するということは、規則の一部を破ることが理にかなっている場合もあることを知る 1 つの例です。JavaBeans 仕様に従うには読みづらいコードを作成するのもやむを得ないというのであれば、この仕様は何の役にも立っていないことになります。その一方、流れるようなインターフェースを作成、あるいはこれを使用するとしても、流れるようなインターフェースと JavaBeans インターフェースの両方をサポートできないというわけではありません。流れるようなインターフェースのメソッドが方向転換して、標準の set メソッドを呼び出すことは可能です。したがって、フレームワークが JavaBean クラスと対話しなければならない場合でも、流れるようなインターフェースを使用することができます。

終了問題の解決方法

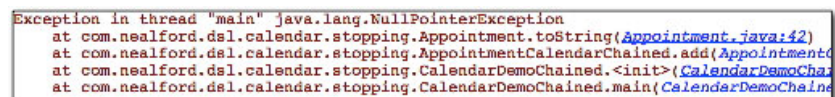
流れるようなインターフェースには、特定の状況下で付いて回る落とし穴があります。それは、終了問題 (finishing problem) という名前で知られている問題です。この問題について説明するため、これから [リスト 5](#) に記載した `AppointmentCalendar` クラスに変更を加えます。おそらく皆さんは、アポイントメントを表示するだけでなく、アポイントメントをデータベース、または何らかの永続化メカニズムに格納したいと思うことでしょう。完了したアポイントメントをストレージに保存するためのコードは、どこに追加すればよいと思いますか？試しに、`AppointmentCalendar` メソッドの `add()` でアポイントメントを返す直前に追加してみてください。リスト 6 では、アポイントメントを出力するだけの単純な目的でアポイントメントにアクセスしてみます。

リスト 6. 出力操作の追加

```
public Appointment add(String name) {
    Appointment appt = new Appointment(name);
    appointments.add(appt);
    System.out.println(appt.toString());
    return appt;
}
```

[リスト 6](#) のコードを実行すると、残念な結果が表示されます (図 2 を参照)。

図 2. `AppointmentCalendar` に内容を追加した後のエラー出力



```
Exception in thread "main" java.lang.NullPointerException
    at com.nealford.dsl.calendar.stopping.Appointment.toString(Appointment.java:42)
    at com.nealford.dsl.calendar.stopping.AppointmentCalendarChained.add(AppointmentCalendarChained.java:42)
    at com.nealford.dsl.calendar.stopping.CalendarDemoChained.<init>(CalendarDemoChained.java:42)
    at com.nealford.dsl.calendar.stopping.CalendarDemoChained.main(CalendarDemoChained.java:42)
```

表示されたエラーは、`Appointment` クラスの `toString()` メソッドで発生した `NullPointerException` です。メソッドは正常に機能しているにも関わらず、このエラーが発生してしまうのが、終了問題の本質です。

エラーが発生した原因は、流れるようなインターフェースの残りのセッター・メソッドが呼び出される前に、アポイントメントのインスタンスで `toString()` メソッドを呼び出そうとしているためです。アポイントメントを出力するためのコードは、アポイントメントのインスタンスを作成してメソッド・チェーンを開始するメソッドの中にあります。 `save()` または `finished()` メソッドを作成して、それをチェーンの最後のメソッドとして呼び出すようにするという方法もありますが、忘れがちな規則を DSL のユーザーに強要したくはありません。実際、この流れるようなインターフェースのメソッドには、順序のセマンティクスを何も設けないようにしています。

真の問題は、私がメソッド・チェーンの手法を積極的に使用し過ぎていることにあります。メソッド・チェーンが最も効果を発揮するのは単純なデータ・オブジェクトを作成する場合ですが、ここでは `Appointment` でのセッター・メソッドに対しても、メソッド・チェーンを開始する `AppointmentCalendar` の中でもメソッド・チェーンを使用しています。

この終了問題は、アポイントメントを作成するためのコード一式を、アポイントメント・カレンダーの `add()` メソッドの括弧の中にラップするという方法によって解決することができます (リスト 7 を参照)。

リスト 7. パラメーターによるラップ

```
AppointmentCalendar calendar = new AppointmentCalendar();
calendar.add(
    new Appointment("Dentist").
        at(fourPM));
calendar.add(
    new Appointment("Conference Call").
        from(fourPM).
        to(fivePM).
        at("555-123-4321"));
calendar.add(
    new Appointment("birthday party").
        from(fourPM).
        to(fivePM)).
    add(
        new Appointment("Doctor").
            at("123 Main St"));
calendar.add(
    new Appointment("No Fluff, Just Stuff").
        at(fourPM));
displayAppointments(calendar);
```

リスト 7 では、`add()` メソッドの括弧が `Appointment` の流れるようなインターフェース全体をカプセル化しているため、`add()` メソッドは追加に必要な振る舞い (出力、永続化など) を何でも処理することができます。実際、私は `AppointmentCalendar` 自体にも流れるようなインターフェースを若干追加せずにはいられませんでした。それによって、リスト 7 の `add()` をまとめてチェーンングできるようになっています。その実装は、リスト 8 のとおりです。

リスト 8. パラメーターによる `AppointmentCalendar` のラップ

```
public class AppointmentCalendar {
    private List<Appointment> appointments;

    public AppointmentCalendar() {
        appointments = new ArrayList<Appointment>();
    }

    public AppointmentCalendar add(Appointment appt) {
        appointments.add(appt);
        return this;
    }

    public List<Appointment> getAppointments() {
        return appointments;
    }
}
```

流れるようなインターフェースのクラスを組み合わせるときには、常に終了問題が発生する可能性があります。この例で終了問題が浮上してきたのは、アポイントメント・カレンダーを使用してメソッド・チェーンを開始し、構成とラッピングの振る舞いを混在させたためです。構成と初期化を `Appointment` クラスまで遅延させることにより、追加のラッピングの振る舞い (永続化など) を簡単に分離できるようになります。

関数シーケンスによるラップ

これまでに、流れるようなインターフェースの DSL でコンテキストを渡す 3 つの手法のうち、2 つを説明しました。3 番目の手法となる関数シーケンス (functional sequence) では、継承と匿名内

部クラスを使用してコンテキスト・ラッパーを作成します。リスト 9 に、関数シーケンスの手法を適用して作成し直したカレンダー・アプリケーションを記載します。

リスト 9. 関数シーケンスによるラップ

```
calendar.add(new Appointment() {{  
    name("dentist");  
    from(fourPM);  
    to(fivePM);  
    at("123 main street");  
}});  
  
calendar.add(new Appointment() {{  
    name("birthday party");  
    at(fourPM);  
}});
```

リスト 9 には、「[Leveraging reusable code, Part 2](#)」で紹介したパターンが、構造上の重複をなくすという名目で使用されています。この構文が奇妙に見えるのは、二重の波括弧 ({{}}) のせいです。最初の波括弧は匿名内部クラスの構成をラップし、2 番目の波括弧は匿名内部クラスのインスタンス・イニシャライザーをラップしています (この説明では理解しにくかったら、「[Leveraging Reusable Code, Part 2](#)」でこの Java イディオムについての詳細な説明を読んでください)。

流れるようなインターフェースのこのスタイルの主な利点は、その適応性にあります。クラスをこのように使用するために必要なものは唯一、デフォルト・コンストラクターだけです (これにより、クラスを継承する匿名内部クラスのインスタンスを作成することができます)。これは、流れるようなインターフェースのメソッドを、現行の呼び出しセマンティクスを一切変更することなく簡単に既存の Java API に追加できることを意味します。したがって、既存の API を除々に「流れるようなインターフェース」に変えていくことができます。

まとめ

DSL は、イディオムのような領域特有のパターンを簡潔かつ効果的に抽出します。流れるようなインターフェースは、コードの作成方法を変更する単純な方法となり、これまで特定するのに苦労していたイディオムのようなパターンを簡単に見分けられるようになります。また、流れるようなインターフェースによって、コードに対する見方にも多少の変化が必要になります。つまり、コードは単に機能するというだけでなく、読み易くなければなりません。これは特に、開発者でない人がコードの何らかの側面を必要とする場合にはなおさらのことです。流れるようなインターフェースは不要なノイズを構文から取り除き、コードをより読み易くします。宣言型の構造については、その概念をより明確に、しかもより少ないコードで表現することができます。

次の記事でも引き続き、新方式の設計でイディオムのようなパターンを抽出するためのメカニズムとしての DSL 手法について説明します。

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・プログラマー 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。

© Copyright IBM Corporation 2010

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)