進化するアーキテクチャーと新方式の設計: イディオムのようなパターンの抽出

新方式の設計手法をまとめて適用してイディオムのようなパターンを発見し、利用する

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2009年 12月 08日

今回の記事では、シリーズ「進化するアーキテクチャーと新方式の設計」のこれまでの記事で説明した新方式の設計についての概念を 1 つの事例研究でまとめ、コードのなかに隠れた意外な設計要素を見つけ、抽出し、利用する方法を紹介します。設計要素を識別する方法を理解しさえすれば、その知識を生かしてコードの設計を改善することができます。新方式の設計によって、今までコード・ベースの重要な部分になるとは予想もしなかったコードの側面を発見できるようになります。

このシリーズの他の記事を見る

このシリーズの最初の記事、「Investigating architecture and design」では、あらゆる規模のあらゆるプロジェクトに、誰一人として予想していなかった設計要素が含まれていると断言しました。問題を詳しく探っていくと、難しいと思っていたことが意外と簡単であったり、あるいはその逆だったりすることは珍しくありません。その後の記事では、コードに隠された興味深い設計要素を明らかにする手法を説明しました。今回の記事では、今まで説明した概念を 1 つの広範な事例研究としてまとめ、ツールと手法によって、コード・ベースのなかで見逃されがちながらも重要な部分を発見する過程を紹介します。

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。

イディオムのようなパターンという概念は、「Composed Method と SLAP」のなかで紹介したとおり、Gang of Four が著した『Design Patterns』(「参考文献」を参照) によって一般的になった

正式なデザイン・パターンとは対照的に、すべてのプロジェクトに適用されるわけではありません。しかしイディオムのようなパターンは至るところに登場し、コードに潜む共通の設計イディオムを表しています。こうしたパターンには、純粋に技術的なパターン (例えば、あるプロジェクトでのトランザクションの処理方法など) もあれば、その領域に特有の問題に対応するためのパターン (例えば「受注処理を進める前に必ず顧客の信用情報をチェックする」など) もあるなど、非常に幅があります。これらのパターンを発見することこそが、新方式の設計の鍵となります。

Big Design Up Front 設計手法 (前もって大規模な設計を行う手法) の支持者たちは、コーディングを開始する前に、作業対象のアプリケーションに必要な設計要素のすべてを決定しようと多大な時間を費やします。文書化される内容の大部分は、ソリューションの全体設計にとって重要なものです。けれどもソフトウェアを実装してみると、驚くべき事態が明らかになります。実装するあらゆる設計要素は他の設計要素と結び付けられており、依存関係と相互関係はとても複雑に入り組んだクモの巣のようになります。そしてシステムに必要な他のすべての部分を実装するなり、平凡だと思っていたコードの側面が複雑になってきます。このことから、コードに含まれる個々の設計要素間の複雑な相互作用を理解することができなければ、ソリューションを完成するために必要な作業を見積もることが極めて困難になります。設計要素の結合と相互作用からなるこの複雑なクモの巣を理解するのは難しく、したがって分析するのも難しいからこそ、ソフトウェアでの作業を見積もることは、未だに黒魔術のような領域となっています。

新方式の設計を利用したアジャイル手法で試みているのは、それとは異なる取り組みです。アジャイルなアーキテクチャーと設計の実践者たちは、コーディングの前に設計するという方法を避けているわけではありませんが、全体像のなかの簡単ではない部分を実装するまでは、問題を完全に把握することはできないことを理解しています。新方式の設計でのスキルを磨くことで、より詳しいコンテキストがわかるまで、設計に関する重要な決定を遅らせることができるようになります。リーン・ソフトウェアを推し進める動き (「参考文献」を参照) には、最終責任時点と呼ばれる優れた概念があります。これは、決定事項を際限なく遅らせるのではなく、最終的に判断が必要な時点まで遅らせるという概念です。設計に関する重要な決定事項を後に延ばすことができれば、それまでに蓄積されたより多くの情報によって、微妙なコンテキストに沿った決定を行うことができます。

イディオムのようなパターンの抽出

新方式の設計では、既存のコードから設計要素を見つけることができます。これらの設計要素は、再利用の可能性を秘めた事実上の抽象化として捉えることができます。イディオムのようなパターンを抽出する 1 つの手法は、メトリックの組み合わせを使用することです。この手法を説明するため、これから (これまでの記事と同じく) Apache Struts のコード・ベースを使用します (「参考文献」を参照)。Struts を使用している理由は、これに欠陥があると考えているからではなく (事実、それとは反対です)、よく知られたオープンソースだからです。私はすべてのコード・ベースにはイディオムのようなパターンが潜んでいると強く主張します。つまり、あらゆるプロジェクトに、イディオムのようなパターンが存在するということです。

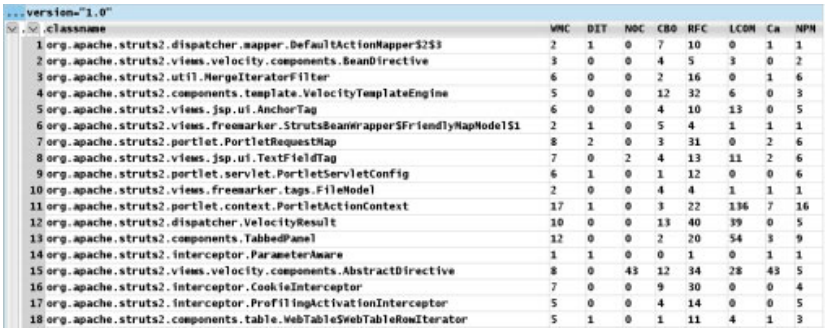
再びメトリックを活用する

「Emergent design through metrics」では、メトリックを使用して、よく知らないコード・ベースのなかで設計の改善につながるリファクタリング対象を見つける方法を説明しました。説明のなかで使用したメトリックは、循環的複雑度と求心性結合の 2 つです。循環的複雑度は純粋に、

あるメソッドの別のメソッドに対する相対複雑度を測る手段です。したがって、このメトリックは他の循環的複雑度の測定値と比べる場合にのみ意味があります。ただし一般的に、循環的複雑度の低いメソッドはあまり複雑ではないと言えます。一方、求心性結合は、対象とするクラスをフィールドやパラメーターによって参照している他のクラスの数を表します。私は Struts コード・ベースでこれらの数値を収集するために、CKJM (Chidamber and Kemerer Java Metrics) メトリック・ツール (「参考文献」を参照) を使用しました。

この 2 つのメトリックを Struts 2 コード・ベースに対して実行すると、対象とする 2 つのメトリックを示す図 1 の結果が得られます。

図 1. 表形式での ckjm メトリックの結果



	...version="1.0"								
	.classname	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM
1	org.apache.struts2.dispatcher.mapper.DefaultActionMapper$2$3	2	1	0	7	10	0	1	1
2	org.apache.struts2.views.velocity.components.BeanDirective	3	0	0	4	5	3	0	2
3	org.apache.struts2.util.MergeIteratorFilter	6	0	0	2	16	0	1	6
4	org.apache.struts2.components.template.VelocityTemplateEngine	5	0	0	12	32	6	0	3
5	org.apache.struts2.views.jsp.ui.AnchorTag	6	0	0	4	10	13	0	5
6	org.apache.struts2.views.freemarker.StrutsBeanWrapper$FriendlyMapModel$1	2	1	0	5	4	1	1	1
7	org.apache.struts2.portlet.PortletRequestMap	8	2	0	3	31	0	2	6
8	org.apache.struts2.views.jsp.ui.TextFieldTag	7	0	2	4	13	11	2	6
9	org.apache.struts2.portlet.servlet.PortletServletConfig	6	1	0	1	12	0	0	6
10	org.apache.struts2.views.freemarker.tags.FileModel	2	0	0	4	4	1	1	1
11	org.apache.struts2.portlet.context.PortletActionContext	17	1	0	3	22	136	7	16
12	org.apache.struts2.dispatcher.VelocityResult	10	0	0	13	40	39	0	5
13	org.apache.struts2.components.TabbedPanel	12	0	0	2	20	54	3	9
14	org.apache.struts2.interceptor.ParameterAware	1	1	0	0	1	0	1	1
15	org.apache.struts2.views.velocity.components.AbstractDirective	8	0	43	12	34	28	43	5
16	org.apache.struts2.interceptor.CookieInterceptor	7	0	0	9	30	0	0	4
17	org.apache.struts2.interceptor.ProfilingActivationInterceptor	5	0	0	4	14	0	0	5
18	org.apache.struts2.components.table.WebTable$WebTableRowIterator	5	1	0	1	11	4	1	3

図 2 に、上記と同じ表を WMC (Weight Methods per Class) でソートした場合を示します。

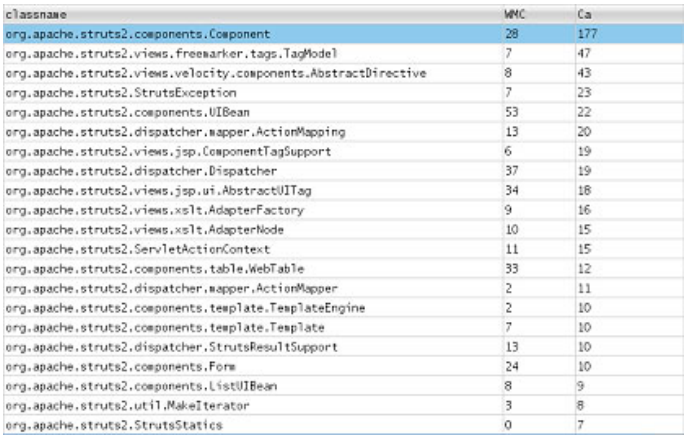
図 2. WMC でソートされた ckjm メトリックの結果

classname	WMC	Ca
org.apache.struts2.components.DoubleListUIBean	66	3
org.apache.struts2.views.jsp.ui.AbstractDoubleListTag	66	2
org.apache.struts2.views.xslt.AbstractAdapterNode	57	4
org.apache.struts2.portlet.util.HttpServletRequestMock	57	1
org.apache.struts2.components.UIBean	53	22
org.apache.struts2.components.Tree	51	3
org.apache.struts2.views.freemarker.tags.StrutsModels	50	1
org.apache.struts2.views.jsp.ui.TreeTag	49	0
org.apache.struts2.components.OptionTransferSelect	44	3
org.apache.struts2.views.xslt.SimpleAdapterDocument	44	2
org.apache.struts2.views.jsp.ui.OptionTransferSelectTag	43	0
org.apache.struts2.dispatcher.Dispatcher	37	19
org.apache.struts2.views.jsp.ui.AbstractUITag	34	18
org.apache.struts2.components.InputTransferSelect	34	3
org.apache.struts2.components.table.WebTable	33	12
org.apache.struts2.views.jsp.ui.InputTransferSelectTag	33	0
org.apache.struts2.components.Component	28	177
org.apache.struts2.components.Autocompleter	26	3
org.apache.struts2.views.jsp.ui.SubmitTag	25	0
org.apache.struts2.components.Form	24	10
org.apache.struts2.views.xslt.AbstractAdapterElement	24	6

この結果だけで判断すれば、`DoubleListUIBean` クラスが Struts コード・ベースで最も複雑なクラスであるということになります。これが意味することは、リファクタリングによってクラスの複雑さを軽減し、抽象化可能な繰り返しのパターンをクラスの中に見つけられるかどうかを調べる対象としては、このクラスが有力候補であるということです。ただし WMC の値からは、設計の改善を目的にこのクラスのリファクタリングに時間をかけることが有効であるかどうかはわかりません。このクラスの Ca (求心性結合) メトリックを見てみると、その値は 3 となっています。つまり、このクラスを使用する他のクラスは 3 つしかありません。そこから判断すると、このクラスの設計を改善するために多くの時間を費やす価値はなさそうです。

図 3 に、同じ CKJM の結果を今度は Ca でソートした場合の表を示します。

図 3. Ca (求心性結合) でソートされた ckjm メトリックの結果



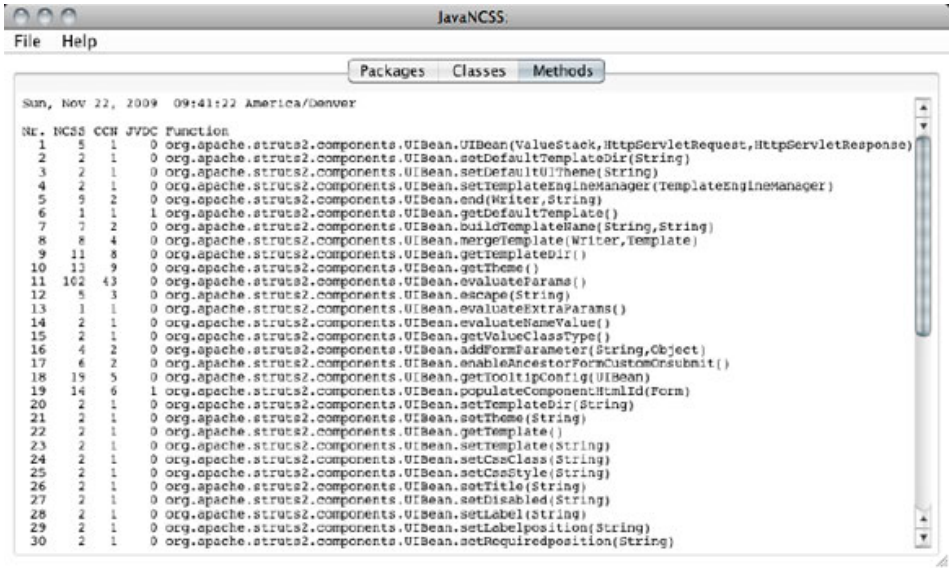
classname	WMC	Ca
org.apache.struts2.components.Component	28	177
org.apache.struts2.views.freemarker.tags.TagModel	7	47
org.apache.struts2.views.velocity.components.AbstractDirective	8	43
org.apache.struts2.StrutsException	7	23
org.apache.struts2.components.UIBean	53	22
org.apache.struts2.dispatcher.mapper.ActionMapping	13	20
org.apache.struts2.views.jsp.ComponentTagSupport	6	19
org.apache.struts2.dispatcher.Dispatcher	37	19
org.apache.struts2.views.jsp.ui.AbstractUITag	34	18
org.apache.struts2.views.xslt.AdapterFactory	9	16
org.apache.struts2.views.xslt.AdapterNode	10	15
org.apache.struts2.ServletActionContext	11	15
org.apache.struts2.components.table.WebTable	33	12
org.apache.struts2.dispatcher.mapper.ActionMapper	2	11
org.apache.struts2.components.template.TemplateEngine	2	10
org.apache.struts2.components.template.Template	7	10
org.apache.struts2.dispatcher.StrutsResultSupport	13	10
org.apache.struts2.components.Form	24	10
org.apache.struts2.components.ListUIBean	8	9
org.apache.struts2.util.MakeIterator	3	8
org.apache.struts2.StrutsStatics	0	7

このようにソートされた表では、Struts のなかで他のクラスに最も使用されているクラスは `Component` であることが示されています (Struts が Web フレームワークであることを考えれば、これは当然です)。`Component` は `DoubleListUIBean` ほどには複雑でないものの、このクラスは 177 の他のクラスによって使用されるため、設計を改善するには有力な候補です。`Component` の設計を改善すれば、他の多数のクラスに波及効果がもたらされます。

図 3 の表では、複雑度と参照の数を並べて確認することができます。設計に問題があるクラスを見つけるには、この表で大きな値の組み合わせを持つクラス (つまり、他の多数のクラスによって使用される複雑なクラス) を探せばよいわけです。私が調査の第一候補として注目したのは、循環的複雑度が 53 で、求心性結合が 22 となっている `UIBean` クラスです。これは他の多くのクラスが使用する複雑なクラスなので、さらに詳しく調査することにします。

ckjm がレポートする循環的複雑度の値は、このクラスに含まれるすべてのメソッドの複雑度を合計した値です。したがって、このクラスを複雑にしている原因を判断するには、個々のメソッドの複雑度の値を調べる必要があります。オープンソースの循環的複雑度ツールである JavaNCSS (「参考文献」を参照) をこのクラスで実行すると、図 4 に示す結果となります。

図 4. `UIBean` クラスに含まれる個々のメソッドの複雑度



群を抜いて複雑なメソッドは、複雑度が 43 の `evaluateParams()` です (コード行の中で最大の部分を占めているのも、このメソッドです)。このメソッドは一見したところ、リクエストの一部として Struts コントローラーに追加パラメーターが渡されるという共通の事例を処理し、パラメーターの型を実際の Struts クラスとコンポーネントに渡すようですが、このコードには構造の重複がかなりあります (リスト 1 を参照)。

リスト 1. 構造の重複を示す `evaluateParams()` メソッドのコンテンツ (抜粋)

```
if (label != null) {
    addParameter("label", findString(label));
}

if (labelPosition != null) {
    addParameter("labelposition", findString(labelPosition));
}

if (requiredposition != null) {
    addParameter("requiredposition", findString(requiredposition));
}

if (required != null) {
    addParameter("required", findValue(required, Boolean.class));
}

if (disabled != null) {
    addParameter("disabled", findValue(disabled, Boolean.class));
}

if (tabindex != null) {
    addParameter("tabindex", findString(tabindex));
}

if (onclick != null) {
    addParameter("onclick", findString(onclick));
}
// much more code elided for space considerations
```

このコードには明らかに改善の余地がありますが (次のセクション「コードの改善、パート 1」を参照)、その前に、このコードが存在する理由、そして場合によってはコードがここまで複雑になっている理由について、もう少し検討してみたいと思います。

循環的複雑度と求心性結合の値がどちらも大きい他のクラスを調べてみると、`WebTable` ではメトリックそれぞれの値が 33、12 となっています。このクラスで JavaNCSS を実行してみると、私が思ったとおり、このクラスで 2 番目に複雑なメソッドは `evaluateExtraParams()` です。ここにはパターンが見えます！この複雑な要素がさまざまなクラスで繰り返し現れるということは、パラメーターを中心とした予期せぬ複雑さが大量に存在していることが疑われます。そこで、1 つの実験を行ってみることにしました。この実験では、UNIX® コマンドラインのマジックを少し利用して、Struts に `evaluateParams()` または `evaluateExtraParams()` という名前のメソッドを持つクラスがいくつあるのかを調べます。

```
find . -name "*.java" | xargs grep -l "void evaluate.*Params" > pbcopy
```

上記のコマンドはカレント・ディレクトリーから下に向かってすべての Java™ ソース・ファイルを検出し、検出したファイルそれぞれのなかで、`evaluate` で始まり、`Params` で終わるメソッド定義を検索します。最後のリダイレクト (>) が、結果のファイル・リストを (少なくとも Mac では) クリップボードに貼り付けます。結果を貼り付けたところ、驚くべきことがわかりました。

```
AbstractRemoteCallUIBean.java
Anchor.java
Autocompleter.java
Checkbox.java
ComboBox.java
DateTimePicker.java
Div.java
DoubleListUIBean.java
DoubleSelect.java

File.java
Form.java
FormButton.java
Head.java
InputTransferSelect.java
Label.java
ListUIBean.java
OptionTransferSelect.java
Password.java
Reset.java
Select.java
Submit.java
TabbedPanel.java
table/WebTable.java
TextArea.java
TextField.java
Token.java
Tree.java
UIBean.java
UpDownSelect.java
```

上記のすべてのクラスに、この 2 つのメソッドのどちらか一方、または両方が含まれています。これでついに、イディオムのようなパターンを見つけました。当然、Struts のクラスの多くはパラメーターの処理方法に関する動作をオーバーライドしてカスタマイズする必要があります。そ

してこれらのクラスはすべて、カスタマイズされた事例を独自に処理します。そこで疑問となるのは、改善の方法です。

コードの改善、パート 1

`UIBean` の `evaluateParams()` メソッドには、さまざまな形で構造の重複が現われています。私の同僚の一人は、これを「同じホワイト・スペースの異なる値」と呼んでいます。つまり、同じ構造でクラス名または変数名を置き換えているだけにすぎません。これは基本的に、アプリケーション全体にわたってコードのコピー・アンド・ペーストを行って少し変更を加えるという作業を行ったことを意味するため、コードに問題が存在する兆候として受け取れます。

構造の重複を取り除く一般的な手法は、メタプログラミングを使用して、繰り返される構造を 1 箇所にカプセル化することです。必要となるさまざまな値を提供するには、リフレクションを使用します。リスト 2 に、この手法を使った新しいメソッドと、`evaluateParams()` メソッドの改善された先頭部分を記載します。

リスト 2. メタプログラミングによって取り除かれた構造の重複

```
protected void handleDefaultParameters(final String paramName) {
  try {
      Field f = UIBean.class.getField(paramName);
      if (f.get(this) != null)
          addParameter(paramName, findString(f.get(this)));
  } catch (Exception e) {
      throw new RuntimeException(e.getMessage());
  }
}

public void evaluateParams() {

  addParameter("templateDir", getTemplateDir());
  addParameter("theme", getTheme());

  String[] defaultParameters = new String[] {"label", "labelPosition", "requiredPosition",
      "tabindex", "onclick", "ondoubleclick", "onmousedown", "onmouseup", "onmouseover",
      "onmousemove", "onmouseout", "onfocus", "onblur", "onkeypress", "onkeydown",
      "onkeyup", "onselect", "onchange", "accesskey", "cssClass", "cssStyle", "title"};

  for (String s : defaultParameters)
      handleDefaultParameters(s);
```

リスト 2 の `handleDefaultParameters()` メソッドは、元のコードで繰り返されていた構造を 1 つの `if` 文にカプセル化します。このメソッドは、Struts のパラメーター名を指定するパラメーターを受け入れ、リフレクションを使って適切なフィールドをプログラムによって取得します。続いて元のコードで行っていたように `null` のチェックをし、最後に Struts の `addParameter()` メソッドを呼び出します。

この `handleDefaultParameters` メソッドがあれば、元のコードの行数 (そして循環的複雑度) を大幅に減らすことができます。私は該当する Struts パラメーター名ごとに `String` の配列を作成し、その配列を繰り返し処理して、それぞれの配列で `handleDefaultParameters()` メソッドを呼び出すようにしました。

すべてのパラメーター・チェックを 1 つの場所にまとめることにより、メソッドのサイズを縮小する以上の成果を上げました。元のメソッドの循環的複雑度は 43 で、以前の `if` ブロックはそ

れぞれ 3 行を占めています (そしてそれぞれが、循環的複雑度を 1 ポイント上げています)。9 行からなる 1 つのメソッド (循環的複雑度は 4) で重複を取り除いたことから、66 行のコード (それぞれ 3 行 x 22 個のパラメーター) が排除されています。つまり、この単純な変更によって、このクラスから 57 行のコードが除去され、新しいメソッドでは循環的複雑度が 18 ポイント (1 CC ポイント x 22 のパラメーター – 4 CC ポイント) も低くなりました。このちょっとした変更で、アプリケーションの読みやすさ、メトリック、サイズ、保守性の大幅な改善を実現したわけです。今後、Struts の `addParameter()` メソッドを呼び出す方法を変更しなければならない場合、その変更は 1 箇所で行えます。

この修正は応急処置的なものですが、単純な変更によってコードが大幅に簡潔になることを実証する目的で紹介しました。けれども、これが自分のコード・ベースだったとしたら、さらに長期的なソリューションを導入します。

コードの改善、パート 2

これが自分のプロジェクトだったとしたら、私はパラメーター処理メカニズム全体を独立した一連のクラスに抽象化して、Struts 内にサブフレームワークを作成すると思います。パラメーターを処理するためのコードの複雑さ、そしてその広範な使用と量を考えると、このコードは Struts 内の第一級市民として扱わなければなりません。その方法は 1 つの記事で説明しきれるものではありませんが、Struts の (メトリックに基づく) 複雑さの大部分は、パラメーターを処理するコードに関連していることは理解できるはずです。

新方式の設計と、イディオムのようなパターン

Struts の設計者たちは、パラメーターを処理するコードがこれほど大量に必要になることを予想していたと思いますか？ソフトウェアとは、そういうものです。問題領域に関する理論的知識によって、コードがどれほど複雑になるかを予測できることもありますが、コードを作成するうちに、新たな制約や機会が生まれてきます。こうした制約や機会を前もって予測することは事実上不可能です。実のところ上級開発者でも、難しい問題の発生を予測することに関して能力が上というわけではありません。けれども彼らは、思いもよらない難しい問題が最終的に頭をもたげてくるだろうと想像することにかけては秀でています。

新方式の設計が持つ魅力の一部は、開発者は何が問題になるかを確実に予測することはできないけれども、問題になりそうなものを注意深く見張っていなければならないという考えを具現化していることです。抽象化およびパターンが見つかるはずだという期待を持ってコード・ベースを調べれば、期待するものが見つかりやすくなります。

最後に、私が断続的ながらも取り組んでいる ThoughtWorks プロジェクトに基づく事例研究を紹介します。この大規模な Ruby on Rails プロジェクトが始まって早々に、プロジェクトの技術リーダーはいくつかの独立した事例のなかで非同期の振る舞いが必要であることに気付きました (例えば、大量の画像をアップロードする場合、ユーザーが途中でページを離れても、ページに戻るとアップロード状況を確認できるようにしなければなりません)。当時、私たちが Big Design Up Front という考え方を持っていたとしたら、すぐにメッセージ・キューに取り掛かっていたことでしょう。しかし、プロジェクトに着手した時点では、非同期が必要になるであろうすべての事例を把握することができません。その場合の基本方針としては、最も複雑なメッセージ・キューを見つけ、そのメッセージ・キューが今後の新しい要件に確実に対応できるようにするという方針

が考えられます。しかし技術リーダーは賢くも、そうはしませんでした。彼は目下の作業には、その時点でわかっている情報で十分対処できると判断したのです。

2 年が過ぎました。その頃までにアプリケーションには 3 つの異なる非同期の振る舞いが用意され、その時点でのソリューションがボトルネックとなり始めてきました。今度こそ、メッセージ・キューに取り掛かる段階です。技術リーダーがここまで決定を延ばしたことから、その頃にはメッセージングに関してアプリケーションに何が必要かを正確に把握していました。そのため、必要なジョブを行う最も単純なツールを使用することができました。最終的に判断が必要な時点になるまで待つことで、必要以上に複雑なツールが強いる予想外の複雑さに対処するための時間を省き、コードの簡潔化、新しい機能の高速化、そして対処に四苦八苦する問題の削減という結果に至ったわけです。

コード主導の設計は、開発者が必要なものをより深く理解していることを意味します。設計に関する決定を後に延ばせば延ばすほど、長期にわたり影響が及ぶであろう決定を下さなければならない時点での方針がより明確になります。

まとめ

このシリーズでは多くの場合、実際のメリットが見えてくるように、読者の頭にコンテキストを詰め込んできました。今回の記事では、実質的にシリーズのこれまでのすべての記事をまとめた総集編として、これまで紹介した技術、ツール、考え方を活用しました。新方式の設計には、イディオムのようなパターンと抽象化を見つけて抽出する能力とともに、パターンと抽象化が見えてきたときに、それを利用するツールと技能が必要です。事前の設計は重要な部分を発見するのに役立ちます (そして、アジャイル・プロジェクトでは重要な部分を決定できるだけの設計を事前に行います)。しかし、コーディングを開始した後は、固定概念にとらわれない目と考えを持ち続けることで、驚くほど重要な設計要素が新たに見つかる結果となります。すべてのコード・ベースにはイディオムのようなパターンがあります。私たちは、そのパターンを見つけて対処する方法を学ばなければなりません。

最後の何回かの記事では主に設計と、設計に関する懸念事項を取り上げてきました。次回は、進化するアーキテクチャーについて再び掘り下げ、よくある問題と、アジャイル開発手法がアーキテクチャーに関する概念と結び付くことによって実現するソリューションを紹介します。

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。