◆ Web プレゼンテーション

ここでは Web プレゼンテーションに関するパターンについて解説します。ここに出てくるパターンは、1 度でもそれなりの規模の Web アプリケーションの開発を行った経験のある人にとっては、お馴染みのものばかりです。その場合は、これまでの知識の整理として、自分のやってきた事を俯瞰してみると良いでしょう。

モデル、ビュー、コントローラ(Model View Controller)

Web アプリケーションをモデル、ビュー、コントローラの 3 つの層に分離する手法です。モデルは、ドメインに関する情報を表わすオブジェクトで、UI に関する処理を除いた全てのデータと振る舞いを提供します。ビューは、モデルの視覚表現を表わします。コントローラは、ユーザの入力を受け取り、モデルを操作し、その結果をビューに反映する役割を持ちます。

このパターンではコントローラとビュー間の分離、ビューとモデル間の分離が行われます。実際に重要なのは後者の方で、これは以下のような理由で必要とされます。

- ビューを作成する場合は、UI の仕組みや UI コンポーネントの配置を考え、モデルを作成する場合は、ビジネスのやり方とかデータベースとのやり取りを考える。これらは大きく異なる内容で、人によって好みが分かれる場合が多く、分離した方が生産性を上げやすい。
- 1 つのモデルに複数の見せ方(Web、リッチクライアント、リモート API、コマンドラインインターフェースなど)を与えたい場合があり得る。ビューを分離しておけば、これを比較的容易に実現できる。仮に Web しか使用しないとしても、同一アプリケーションの中で、同じモデルを違う見せ方で表示するケースがあり得る。
- 一般にユーザとの対話が必要な UI コンポーネントよりも、モデルのような非表示オブジェクトの方がテストは容易である。ビューとモデルを分離しておけば、モデル単独で容易にテストを行うことが可能となる。

ビューはモデルに依存し、モデルはビューに依存しないように設計します。これによりモデルの作成をビューとは無関係に行うことができます。

ページコントローラ(Page Controller)

ページコントローラは、ページ上のリンク、ボタンに対して、それぞれコントローラオブジェクトを割り当てます。CGI やサーブレットのようなモジュールや、ASP や JSP のようなサーバページ(PofEAA では、これらを全てを総称してスクリプト(script)と呼んでいます)にも実装可能ですが、サーバページに実装する場合はスクリプト要素でコントローラを書かないといけないので、コントローラのロジックが複雑だと扱いにくくなります。このため主要なロジックをヘルパオブジェクトに抜き出すと良いでしょう。ヘルパはコントローラからリクエストを受け取って処理した後、必要に応じて別のページにフォワードするようにします。ページコントローラは以下のような処理を行います。

- URL をデコードし、リクエストからすべてのフォームデータを取り出す。
- モデルオブジェクトを生成して、取り出したデータをまとめて渡す。
- 表示すべきビューを決定して、モデルのデータをビューに渡す。

ページコントローラは後述するフロントコントローラに比べると単純で分かり易いのが長所です。サイトのコントローラロジックが比較的単純な場合に用いると良いでしょう。

フロントコントローラ(Front Controller)

ある程度複雑なサイトになってくると、ページコントローラでは収拾がつかなくなってきます。大量のページで構成されている場合、そこには似通った処理、例えばセキュリティ、国際化、特別なユーザのためのビューの提供などが存在し、ページコントローラのやり方だと、どうしてもそれらが複数のページコントローラクラスに重複して実装されてしまいがちですし、コントローラの動作を実行時に変更することも困難です。フロントコントローラは

一旦全てのリクエストを 1 つのハンドラオブジェクトが受けて、それらをコマンドオブジェクトに振り分ける仕組みです。この時にデコレータパターンを用いてリクエストの処理を実行時に拡張することを可能とします(図 3-4-1)。その後の最終的なビューの決定は、各コマンドが行います。フロントコントローラはページコントローラに比べると複雑になりますが、その分、次の利点が得られます。

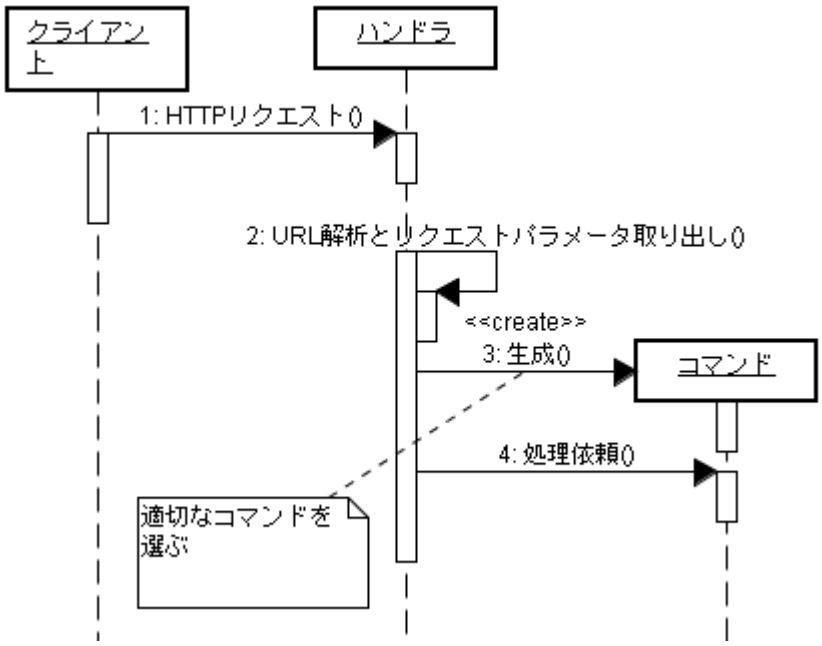

図 3-4-1 フロントコントローラ

- Web サーバに対してはフロントコントローラだけを設定すれば良い。
- コマンドオブジェクトはリクエストごとに生成されるので、スレッドセーフに作成する必要がない。
- 動作を実行時に変更することが可能。

テンプレートビュー(Template View)

Web アプリケーションでは、最終的に HTML を出力する必要がありますが、通常のプログラミング言語で HTML を生成するのは、実際のところ、あまり効率の良いものではありません。そのようなコードは非常に見にくく直感的ではありません。現在、単に静的な HTML を作成するだけならば、WYSIWYG な Web ページエディタが利用可能であり、これを用いることで直感的にページを作成できます。もちろん、このようにして作成した HTML は静的なものに過ぎませんが、この中の動的に変化する部分にマーカ(タグ)を打ち込んで、実行時にそこだけ置き替えれば、より簡単に HTML を生成することができます。これがテンプレートビューの考え方です。

テンプレートビューの実装の代表例としてサーバページ(ASP, JSP, PHP 等)が挙げられます。これらはスクリプト要素を使用すれば、任意のロジックを埋め込むことも可能です。しかし、あまりロジックを埋め込むとサーバページがプログラミング能力のある技術者にしか扱えなくなってしまします。また、そもそもサーバページとプログラミング言語のソースコードとは異質なもので、サーバページの中に書かれたロジックというのは、その構成上、見にくいという欠点があります。乱用すれば、せっかく分離したロジックがサーバページの中に入り込んできてしまい、分離によって得られるメリットが失なわれてしまいます。

特定の条件が成立した時にだけ表示したい要素は、例えば <IF condition="$pricedrop > 0.1"> ... </IF>のようなタグを定義して解決することができますが、これは結局ロジックがページに入り込むのを許してしまうことになります。同様に繰り返しも問題となります。なるべくページ内にロジックを持ち込まないように工夫することが必要です。

テンプレートビューの利点はページの構成を見ながら記述できる点にあります。特にページ設計者がプログラミング知識に乏しい場合には大きな利点となります。欠点として、テンプレートビューを動作させるには一般には Web サーバの機能が必要であり、それ単体でテストすることは困難である点が挙げられます。

変換ビュー(Transform View)

変換ビューは、ドメインのデータ項目それぞれを個別にHTMLに変換し、それらを組み合わせてページを生成する仕組みです。テンプレートビューが表示レイアウトに従って構成されるのに対し、変換ビューはデータ項目の構造に従って構成されます。

変換ビューはどのような言語でも実装できますが、現時点で最も適しているのはXSLTでしょう。まず、ドメイン内のデータ要素をXMLに変換します。子要素があれば、木を辿るようにして要素を処理します。次にそれをXSLT処理系に渡してHTMLへと変換します。

残念ながらXSLTを処理するツールはあまり出回っておらず、ツールの入手性ではテンプレートビューに劣ります。またXSLTは習得が難しいという側面もあります。長所としては、まずXSLTには移植性という強みがあります。XSLTさえあれば、どのようなプラットホームでも同じ変換が可能です。変換ビューでは、ロジックがビューに入り込むような心配は不要です。またXMLとXSLTの入力を用意することで簡単にテストが可能です。

2段階ビュー(Two Step View)

一般にサイトは共通のデザインを持っています。何も考慮せずにテンプレートビューや変換ビューを使用すると、共通デザイン部分の構成が、各ページに複製されてしまうため、後から共通デザインを変えようと思うと、全ページを書き直さなければならなくなってしまいます。2段階ビューはページ生成を2つのステージに分ける事で、この問題を解決します。最初の変換でモデルのデータを論理的な画面表現に変え、次の変換でそれを実際のフォーマットへと変換します。2番目のステージだけを変更すれば、ページの共通デザインを変えたり、2つの見え方を提供することが可能となります。

最初の変換で得られる論理的な画面には、例えばフィールド、ヘッダ、フッタ、表といった論理的なパーツが配置されます。もちろん、これらはあくまで論理的なもので、実際のHTMLとは切り離されています。

変換ビューを使用する場合は、最初のXSLTによって、モデルのデータを表現したXMLから表示に特化したXMLへと変換し、次のXSLTによって、それをHTMLに変換します。

テンプレートビューを使用する例としては、サーバーページを直接HTMLで記述するのではなく、以下のような論理表示タグでフィールドを表現するという方法があります。

```
<field label = "Name" value = "getName" />
```

これをテンプレートの仕組みで最終的にHTMLに変換します。このようにページを論理タグのみで構成すれば、HTMLを排除することができます。結果として出来上がったサーバページはXML文書となります。もちろんこの場合には一般的なHTMLエディタを利用することは出来なくなってしまいます。2段階ビューでは2段階分の変換処理を実装しなければなりません。最初の変換処理はページごとに必要です。これは2段階ビューを使用しなかったとしても、結局は必要なものなので、余分に用意しなければならないのは、次の変換処理でしょう。これは提供する見せ方の数だけ用意する必要があります。複数の見せ方を用意するなら、2段階ビューの方が有利になります。もしも2段階ビューを使わないと、通常はページ数 x 見せ方の数だけビューを用意しなければならなくなりますが、2段階ビューならばページ数 + 見せ方の数だけで済ますことが可能となるからです。

もしもサイトが凝ったデザインを採用していて、各ページ間にあまり共通部分が無いのであれば、論理的なパーツの共通化が困難となるので、2段階ビューはあまり有効ではないでしょう。またHTMLの生成がプログラムで行われるため、サイトのデザイン変更には常にプログラマの参画が必要となってしまう点も欠点の1つです。

アプリケーションコントローラ(Application Controller)

アプリケーションの中には、ウィザード形式のような一連の画面の流れを持っていたり、あるいは条件に従って提示される画面が変わったりするものがあります。こうした画面フローの制御はコントローラの役目ですが、アプリケーションの複雑さが増してくると、幾つかのコントローラ間でコードの重複が見られるようになってきます。アプリケーションコントローラの狙いは、こうした画面フロー制御を一箇所にまとめることで、コードの重複を防ぐ

ことにあります。アプリケーションコントローラは、リクエストをどのドメインロジックに割り振るかを決定し、得られた結果を表示するビューを決定します。一般に、こうした制御は状態遷移で表現され、設定ファイルなどで管理されます。

もしもサイト利用者のナビゲーションが、特に決った画面遷移を辿るようなものでなければ、アプリケーションコントローラは向いていません。このパターンの強みは、特定の順番で画面を遷移させたい場合や、条件に応じて違う画面を表示させたい場合に発揮されます。もしもアプリケーションのフローを変更したい時に、アプリケーションの色々な場所に手を入れなければならなくなってきたら、アプリケーションコントローラを検討すると良いでしょう。とはいえ実際のところは、例えば J2EE の開発であれば、非常に小規模なものであっても、Struts や JSF などのアプリケーションコントローラを使うケースがほとんどで、逆にアプリケーションコントローラを使用しないケースは非常に少ないと思われます。