Webアプリケーションフレームワーク

Webアプリケーションフレームワーク

- これまで見た、Servlet、JSP、フィルタを用いることで、Webアプリケーションを実装できることが分かりました。
- しかしWebアプリケーションには：
  - 共通の処理があります。
  - 共通の処理の流れがあります。
- これらを毎回実装するのは無駄です。これらを共通部分を提供して生産性と品質を上げる目的で使用されるのがWebアプリケーションフレームワークです(以降、単にWebフレームワーク、あるいはフレームワークと呼ぶ)。

共通の処理

- Webアプリケーションでは例えば、以下のような共通の処理があります。
- リクエストパラメータのパース(変換)処理。
  既に見たようにリクエストパラメータは文字列で渡されます。しかしJavaの中では整数なら、intのような数値型を使いますし、例えば従業員や取引といったエンティティ単位でクラスを使用します。このため、リクエストを変換する処理が必要となります。
- リクエストの検証(バリデーション)処理。
  リクエストの内容は検証が必要です。例えばクレジットカード番号のチェックディジットが合っているか、日付のフォーマットが正しいか(例：月が1-12か)といった検証を行い、正しくない場合にはエラー画面を表示して再入力を促す必要があります。
- 国際化(i18n)、ローカライズ処理(l10n)。
  Webアプリケーションは、インターネット上で公開されるため、複数の言語に対応する必要があるかもしれません。その国の言語に対応したメッセージの内容、金額や日付のフォーマットを自動的に選択できる必要があります。

共通の処理

- 画面のフロー制御。
  複雑なアプリケーションでは、複数の画面をまたいで処理を行う必要があるかもしれません。これらの画面の流れを管理する仕組みが必要となります。

- 例外処理。
  多くのアプリケーションサーバでは、開発の際にはアプリケーションでエラーが発生した場合、ブラウザにスタックトレースを表示します。これは開発者にとっては有用ですが一般の人には意味不明ですし、攻撃者に不用意にシステムの情報を与えてしまう危険があります。アプリケーションでエラーが発生した際には適切なエラー画面を表示する必要があります。

- 画面の共通化。
  通常、Webアプリケーションのデザインは統一されています。これを個別に作成するとコードが重複し、デザインの変更が必要となった際に修正量が画面数に比例して膨大になります。画面の各部をパーツ化して再利用できる必要があります。

- その他。
  その他にも例えばボタンの2度押しを検出するとか、セキュリティ脆弱性を防止するなどの共通処理が挙げられます。

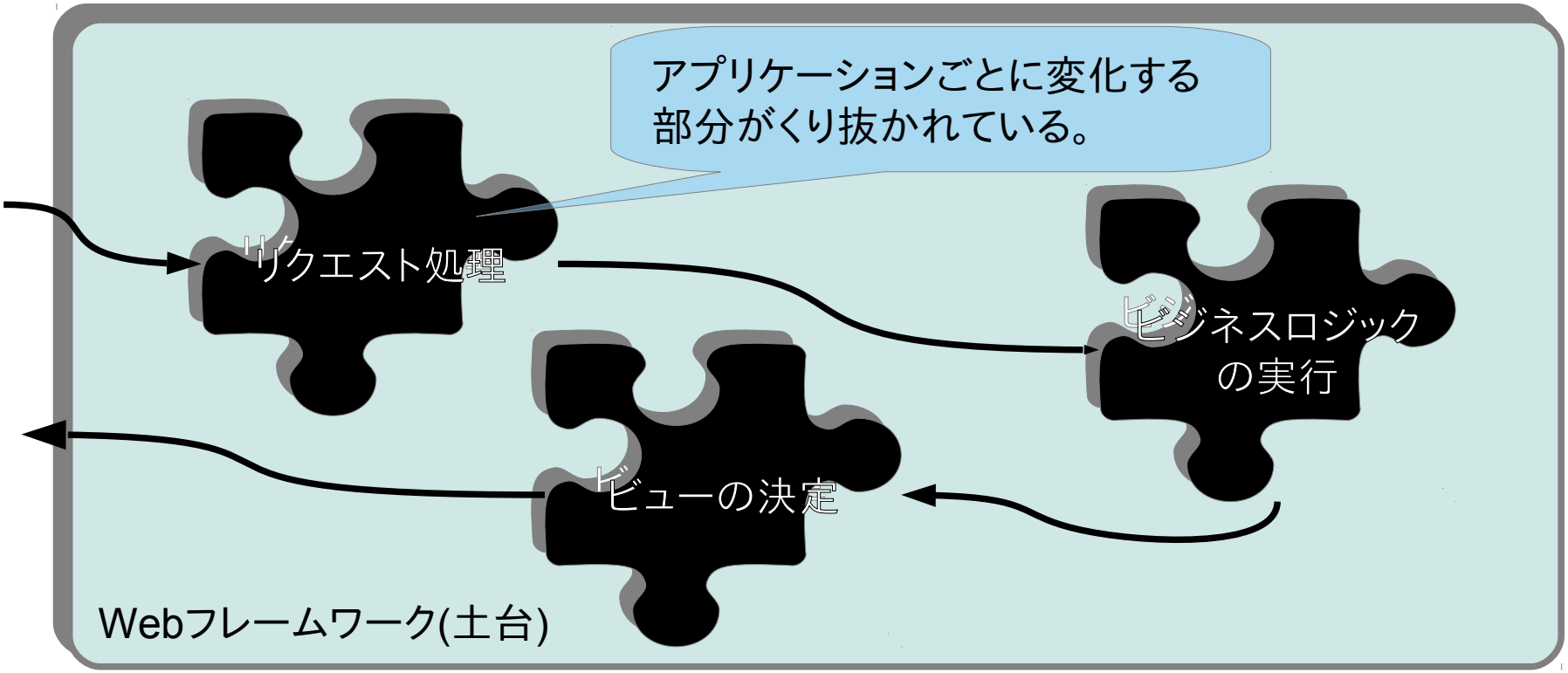
共通の処理の流れ

- Webアプリケーションでは次のような共通の処理の流れがあります。
  - リクエストのパース、バリデーション
  - 実行するビジネスロジックの決定、実行
  - ビュー(表示内容)の決定、生成
  - レスポンスの送信

- こうした共通の流れを処理するための枠組みを提供するのも、Webフレームワークの役割です。

共通の処理の流れ

- フレームワーク概念図



- Webフレームワークは、共通の土台を提供し、各アプリケーションで変更される可能性のあるところが置き換え可能となっている。
- このため穴埋め問題を解く要領でアプリケーションが実装でき、大きな間違いを犯しにくい。

参照実装

- 参照実装(Reference implementation)
  - 参照実装はフレームワークに付属する、穴埋め部分の実装例。多くのフレームワークが提供している。
  - 穴埋め部分をどのように実装すれば良いのかを示す道標となる。
  - アプリケーションで独自実装が不要ならば、そのまま参照実装が使用できる。

まとめ(pros and cons)

- フレームワークの利点
  - 共通部分が提供されるので、生産性、品質が向上する。
  - 共通の枠組みが提供され、穴埋め問題を解く要領で開発できるので、おかしな実装をしてしまうリスクが低減する。

まとめ(pros and cons)

- フレームワークの欠点
  - 共通で提供される枠組みが要件に合っていないと逆に足を引っぱる。
  - 学習コストがかかる(使用するフレームワークの勉強が必要)。
  - フレームワーク自体の出来が悪いと、逆に生産性、品質が悪くなる(実装の品質が悪い、設計が悪い、バッドプラクティスを開発者に押し付ける、テストしにくい)。
  - フレームワーク自体の保守が必要となる。
    - 外部からフレームワークを調達している場合は、保守が終了してしまうと梯子を外された状態になってしまう。
    - 内部で作成した場合、フレームワーク担当者が離任すると保守が困難になる可能性がある。
  - 陳腐化しやすい。
    互換性維持のために設計を変えられず、数年経過すると、あっという間にアーキテクチャが陳腐化して負の資産となり易い。アプリケーション・サーバの進化で新たな機能が提供されるようになっても、フレームワークが対応できずに使用できなくなってしまうことがある。

まとめ(考慮点)

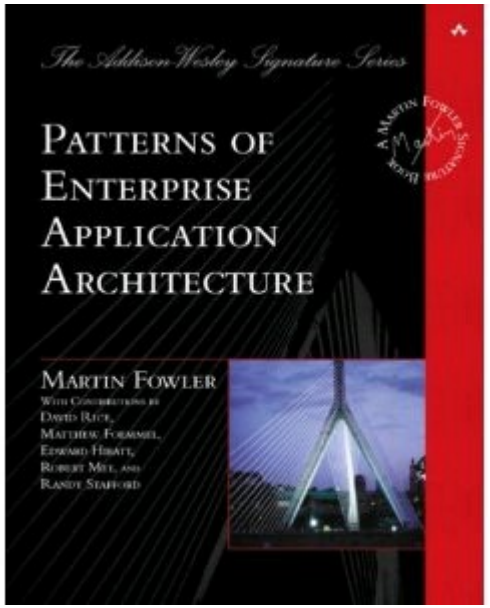
- 逃げられるか？
  - いざとなったら捨てて他のフレームワークに乗り換えられるか？ 手厚いフレームワークほどロックインされやすい(例：データアクセスや特定ミドルウェアに依存してしまっている)。
- 実装したい機能を無理なく実装できるか？
- 非機能要件は大丈夫か？
  - セキュリティ、パフォーマンス、メモリ使用量、トランザクション、障害解析(ロギングなど)。
- 開発メンバに使いこなせるか？
- (OSSの場合)活発に開発が行われているか？ いざとなったらソースを取得して自分達で保守可能か？
- (製品の場合)いつまでサポートしてくれるか？ 将来のアプリケーション・サーバのバージョンアップに追随してくれそうか？
- テストしやすいか？

Patterns of Enterprise Application Architecture

Patterns of Enterprise Application Architecture

- Martin Fowlerが2002/11に出版した、企業向けアプリケーション(Webベース)の設計パターンを解説した書籍。
- エンタープライズ系のWebアプリケーションをやるのであれば、バイブル的存在の必読書。
- 多くのWebフレームワークは、この書籍のプレゼンテーションパターンの影響を少なからず受けている。

- 抽象的な設計パターン集ではなく、実際のJavaや.Netでの実装例を含んだ、すぐに適用できる実装パターン集。



Webフレームワークに関係の深い設計パターン

- ドメインロジック・パターン
  - トランザクション・スクリプト
  - ドメインモデル
  - テーブル・モジュール
  - サービスレイヤ
- Webプレゼンテーションパターン
  - モデル・ビュー・コントローラ
  - ページ・コントローラ
  - フロント・コントローラ
  - テンプレート・ビュー
  - 変換ビュー
  - 2段階ビュー
  - アプリケーションコントローラ

Patterns of Enterprise Application Architecture
ドメインロジック パターン

トランザクション・スクリプト(Transaction Script)パターン

- ビジネスロジックをプロシージャとして実装する。1つのプロシージャが、1つのリクエストを処理する。
- もっとも構造的には単純。
- 検証から、データアクセスなどを含めて1つのプロシージャで処理する。
- 共通の処理は共通のプロシージャに抜き出す。
- 例) ホテル予約の場合以下のようなプロシージャを用意するかもしれない。
  - 空き室検索
  - 料金計算
  - 結果確認
- 実装にあたっては、1つのクラス内のメソッドとして実装するケースと、トランザクション・スクリプトごとに別のクラスを用意する方法がある。

トランザクション・スクリプト(Transaction Script)パターン

- どういう時に使用するか
  - アプリケーションが単純な場合。
  - 開発メンバにOOの素養が無い。
- 利点
  - 単純である。
- 欠点
  - 複雑なアプリケーションでは、コードがスパゲッティになりやすく、また膨大なコード重複を招く。
  - 長大なプロシージャになり易く単体テストしにくい。

ドメインモデル(Domain Model)パターン

- 細粒度で疎結合のオブジェクトの相互作用でビジネスロジックを実装する。
- 構造が柔軟で複雑なロジックを実装するのに向く。
- オブジェクト・モデルはDBの構造に似通ったものになることが多いが、一般にはオブジェクト・モデルの方が細粒度で、継承やストラテジーパターンなどを活用する。

ドメインモデル(Domain Model)パターン

- どういう時に使用するか
  - アプリケーションが複雑な場合。
  - 開発メンバにOOの素養がある。
- 利点
  - 柔軟である。
  - テストし易い。
- 欠点
  - メンバにOOの素養が無いと使いこなせない。

テーブル・モジュール(Table Module)パターン

- ビジネスロジックを、DBのテーブル、ビュー単位にまとめる。
- 行ではなく、テーブル単位にモジュールが生成されるので、DB上のキーを一緒に渡す必要がある。
  例：anEmployeeModule.getAddress(long employeeID)。
- 照会に対してはファクトリメソッドで対応する。
- DBのレコードセットをラップし、業務ロジックをメソッドとして用意する。
- 必ずしもテーブルと1対1に対応させる必要はない。複雑なクエリに対して用意することもできる。

テーブル・モジュール(Table Module)パターン

- どういう時に使用するか
  - 表操作が主体。
  - 開発メンバにOOの素養がない。
  - ビジネスロジックが単純(基本CRUDを提供するだけのアプリケーションには向く)
- 利点
  - 単純。
  - (トランザクション・スクリプトよりは)テストし易い。
- 欠点
  - 複雑なビジネスロジックには向かない。

サービス･レイヤ(Service Layer)パターン

- アプリケーションの外部との境界に、粗粒度のインターフェイスを提供する層を作成する。
- アプリケーション内部の細かな動作をカプセル化して隠蔽する。
- 通常、ビジネスロジックは「ドメインロジック」と「アプリケーションロジック」に分かれる。
  - ドメインロジック
    純粋なビジネス上の問題を扱う(例： 在庫を引き当てる)。
  - アプリケーションロジック
    アプリケーションとしての1まとまりの責務を実行(売上が更新されたら、取引担当者、別のアプリケーションに通知を送る)。別名：ワークフローロジック
- これらの両方をドメインモデルのオブジェクトに入れてしまうのは望ましくない。
  - アプリケーションロジックはアプリケーション固有のものが多い。
  - アプリケーションロジックは、複数のドメインクラスを扱うので、どこかのドメインクラスに入れるのはおかしい。

サービス･レイヤ(Service Layer)パターン

- サービス･レイヤは、ひとまとまりの処理を提供するので粗粒度となる。
- これはリモートプロシージャコールのエントリーポイントとして良く合うが、ローカルインターフェイスで十分なら、リモートにすべきではない(コストが高いため)。必要になったらリモート･ファサードを追加すれば良い。
- トランザクション境界としても利用できる(複数のドメインオブジェクトの更新を、atomicに実施)。

- どういう時に使用するか
  アプリケーションが単純なCRUDのみを提供し、クライアントが1種類のみなど余程単純な場合を除き、基本的には常に用意する。

モデル･ビュー･コントローラ(Model View Controller)パターン

- Webアプリケーションをモデル、ビュー、コントローラに分割して開発する設計パターン。単にMVCと略される。
- モデル
  ビジネスドメインに関する情報、振舞いを表現するう(取引や顧客など)。
- ビュー
  モデルの視覚表現。
- コントローラ
  ユーザ入力を受け取り、モデルを操作し、結果をビューに反映する。
- コントローラとビューを分離する。

モデル･ビュー･コントローラ(Model View Controller)パターン

- ビューとモデルを分離する。
  - 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、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する。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る。

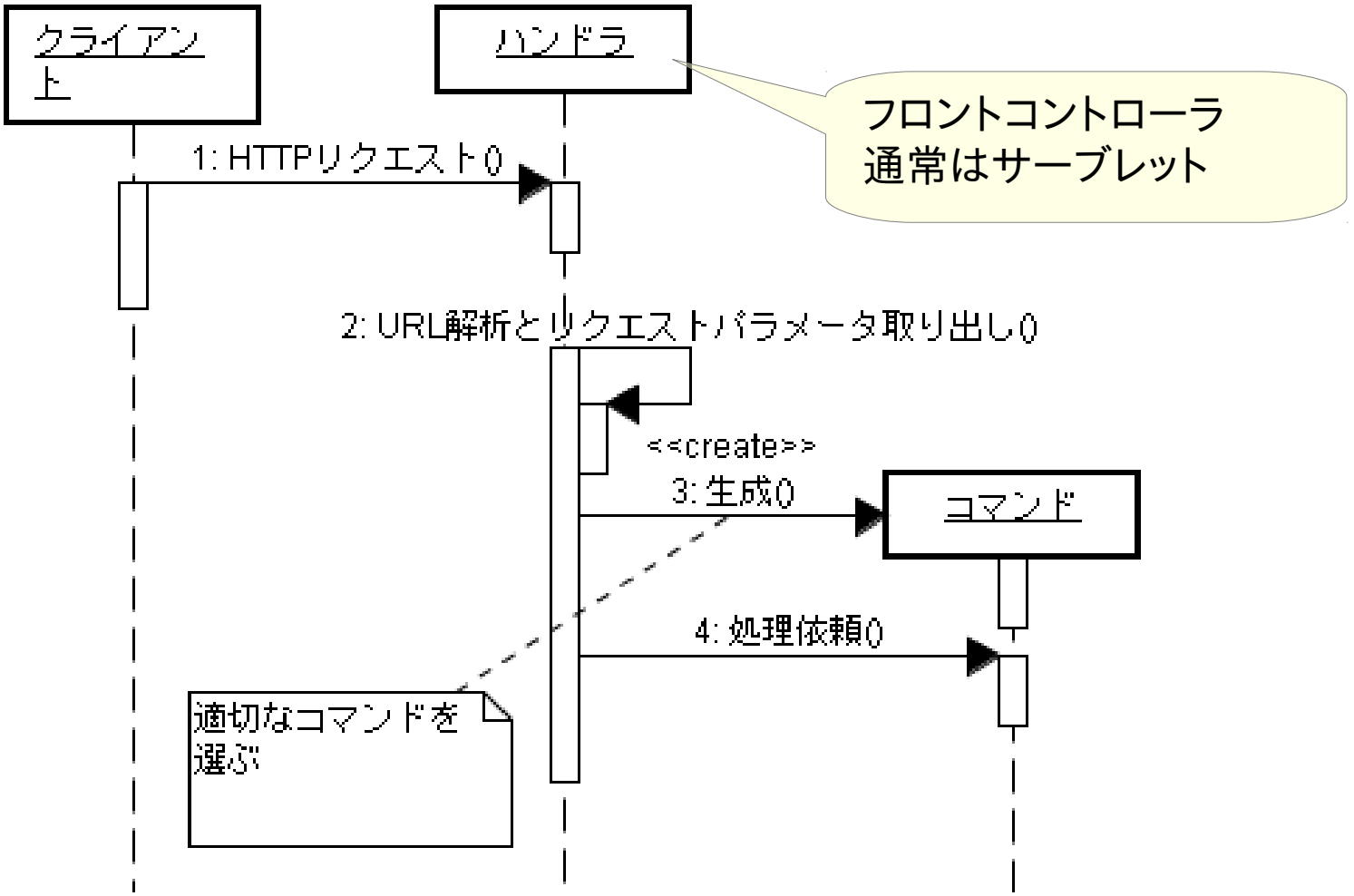
ページコントローラ(Page Controller)パターン

- 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る。
- CGIやサーブレットのようなモジュールだけでなく、ASPやJSPのようなサーバページにも実装可能(ただし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、主要なロジックをヘルパオブジェクトに抜き出してスクリプト側が煩雑にならないよう注意する。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)。
- ページコントローラは以下のような処理を行う。
  - 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(リクエスト・パラメータ)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。
- サイトに複雑なワークフローが無ければ、シンプルで良い。

フロントコントローラ(Front Controller)パターン

- 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くる。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、これらを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、動作を実行時に制御することが困難になる。
- 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。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する。その後の最終的なビューの決定は、各コマンドが行う。

フロントコントローラ(Front Controller)パターン



フロントコントローラ(Front Controller)パターン

- 次のような利点がある。
  - 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(サーブレットは1つのインスタンスが、複数リクエストで共有される)。
  - 動作を実行時に変更することが可能(フロントコントローラで集中制御できる)。
  - 共通処理をまとめられる。

テンプレートビュー(Template View)パターン

- 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るが、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ない。そのようなコードは非常に見にくく直感的ではない。
- 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る。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ないが、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る。
- 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)が挙げられる。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。
- 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う。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、見にくくデバッグもしにくいという欠点があり、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う。

テンプレートビュー(Template View)パターン

- 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るが、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る。同様に繰り返しも問題となる。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。
- 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る。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る。
- 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が困難となる。

変換ビュー(Transform View)パターン

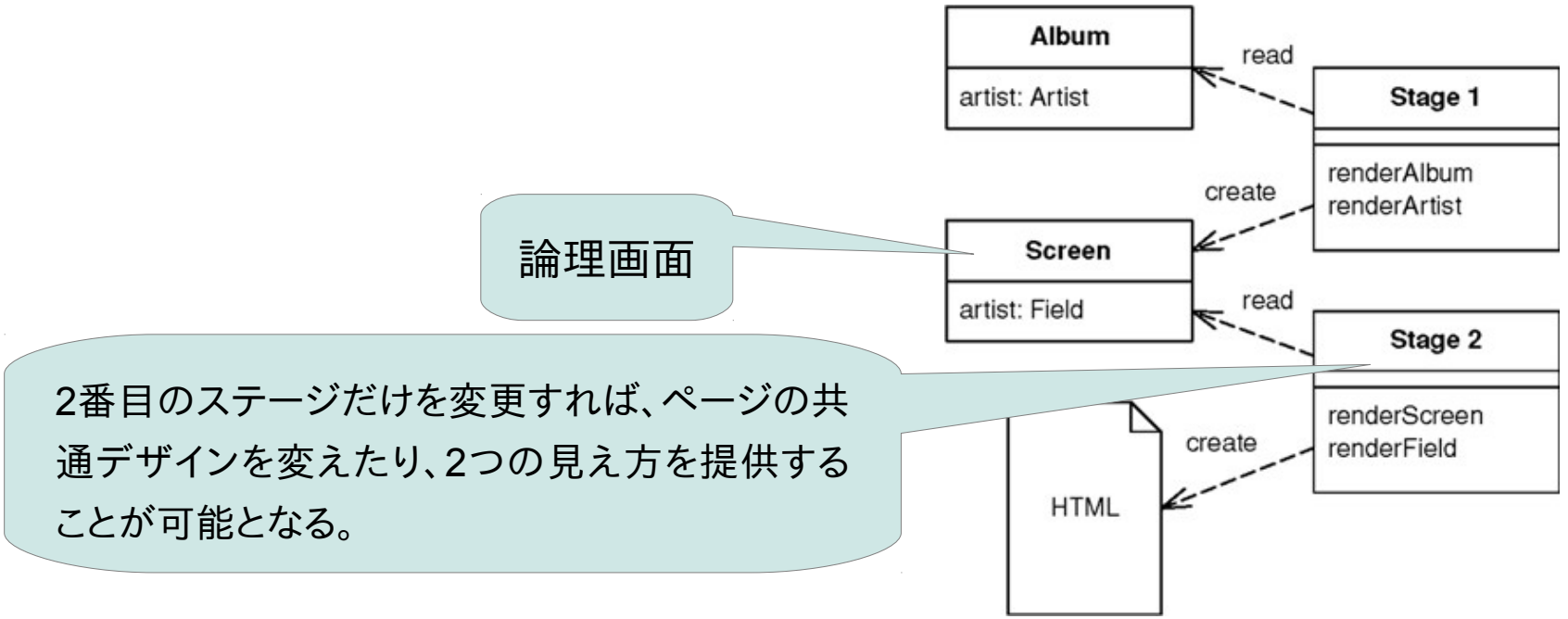
- 変換ビューは、ドメインのデータ項目それぞれを個別にHTMLに変換し、それらを組み合わせてページを生成する仕組み。
- テンプレートビューが表示レイアウトに従って構成されるのに対し、変換ビューはデータ項目の構造に従って構成される。
- 変換ビューはどのような言語でも実装でるが、現時点で最も適しているのはXSLT。
  - まず、ドメイン内のデータ要素をXMLに変換する。
  - 子要素があれば、木を辿るようにして要素を処理すうる。
  - 次にそれをXSLT処理系に渡してHTMLへと変換する。

変換ビュー(Transform View)パターン

- 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る。
- XSLTは習得が難しいという側面もある。
- 長所としては、まずXSLTには移植性という強みがある。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。
- 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。

2段階ビュー(Two Step View)パターン

- 一般にサイトは共通のデザインを持っている。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う。
- 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、この問題を解決する。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する。



2段階ビュー(Two Step View)パターン

- 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る。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る。
- 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に変換する。

2段階ビュー(Two Step View)パターン

- テンプレートビューと組み合わせる例としては、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る。

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

- これをテンプレートの仕組みで最終的にHTMLに変換する。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る。
- 結果として出来上がったサーバページはXHML文書となる
- 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らない。
  - 最初の変換処理はページごとに必要。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。
  - これは提供する見せ方の数だけ用意する必要がある。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る。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るが、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。

2段階ビュー(Two Step View)パターン

- 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。
- 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。

アプリケーション・コントローラ(Application Controller)パターン

- 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る。
- こうした画面フローの制御はコントローラの役目だが、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くる。
- 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る。
- 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する。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る。

アプリケーション・コントローラ(Application Controller)パターン

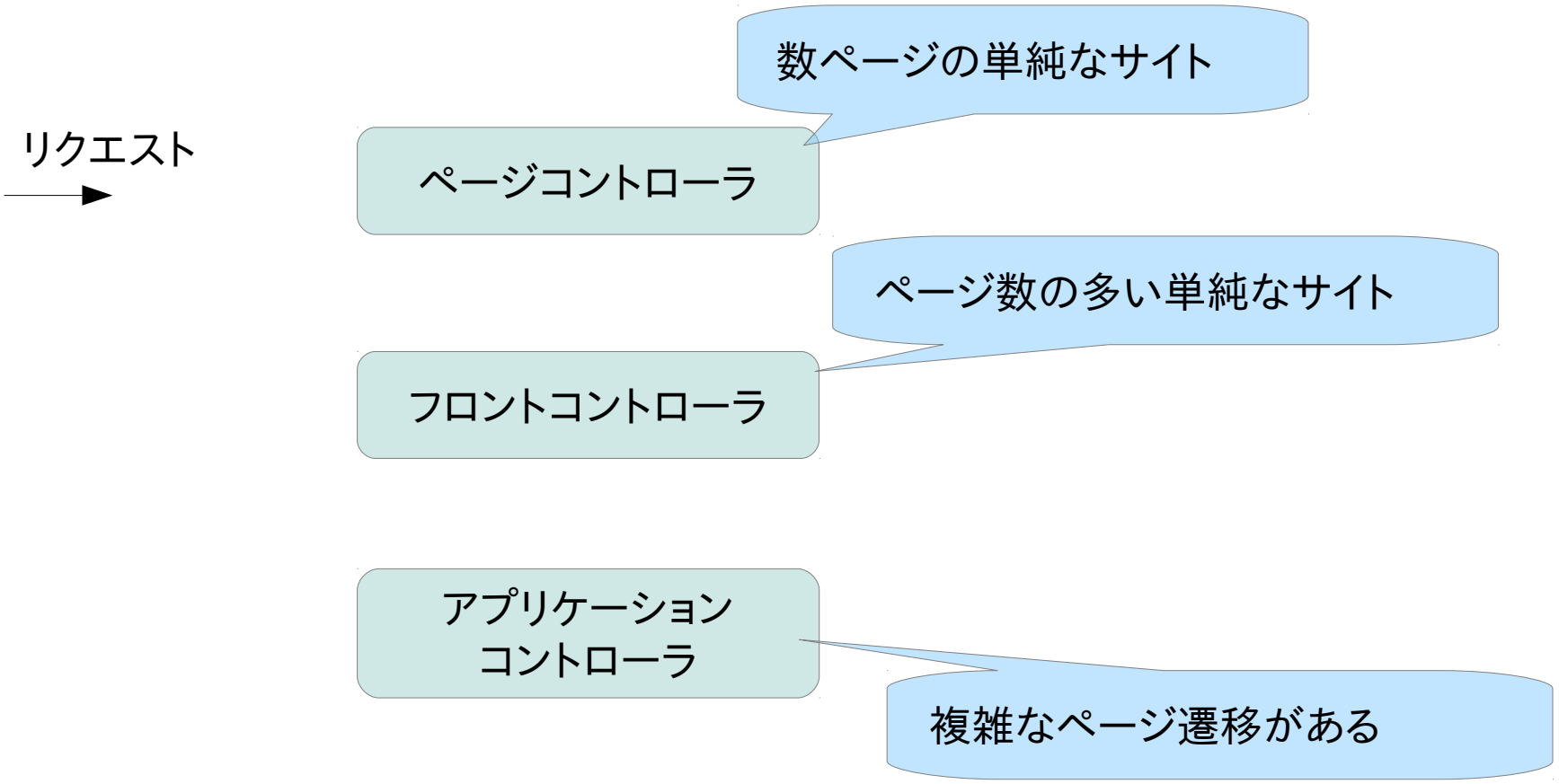
- もしもサイト利用者のナビゲーションが、特に決った画面遷移を辿るようなものでなければ、アプリケーションコントローラは向いていない。
- このパターンの強みは、特定の順番で画面を遷移させたい場合や、条件に応じて違う画面を表示させたい場合に発揮される。
- もしもアプリケーションのフローを変更したい時に、アプリケーションの色々な場所に手を入れなければならなくなってきたら、アプリケーションコントローラを検討すると良い。
- とはいえ実際のところは、例えばJ2EEの開発であれば、非常に小規模なものであっても、StrutsやJSFなどのアプリケーションコントローラを使うケースがほとんどで、逆にアプリケ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

代表的なパターン適用例

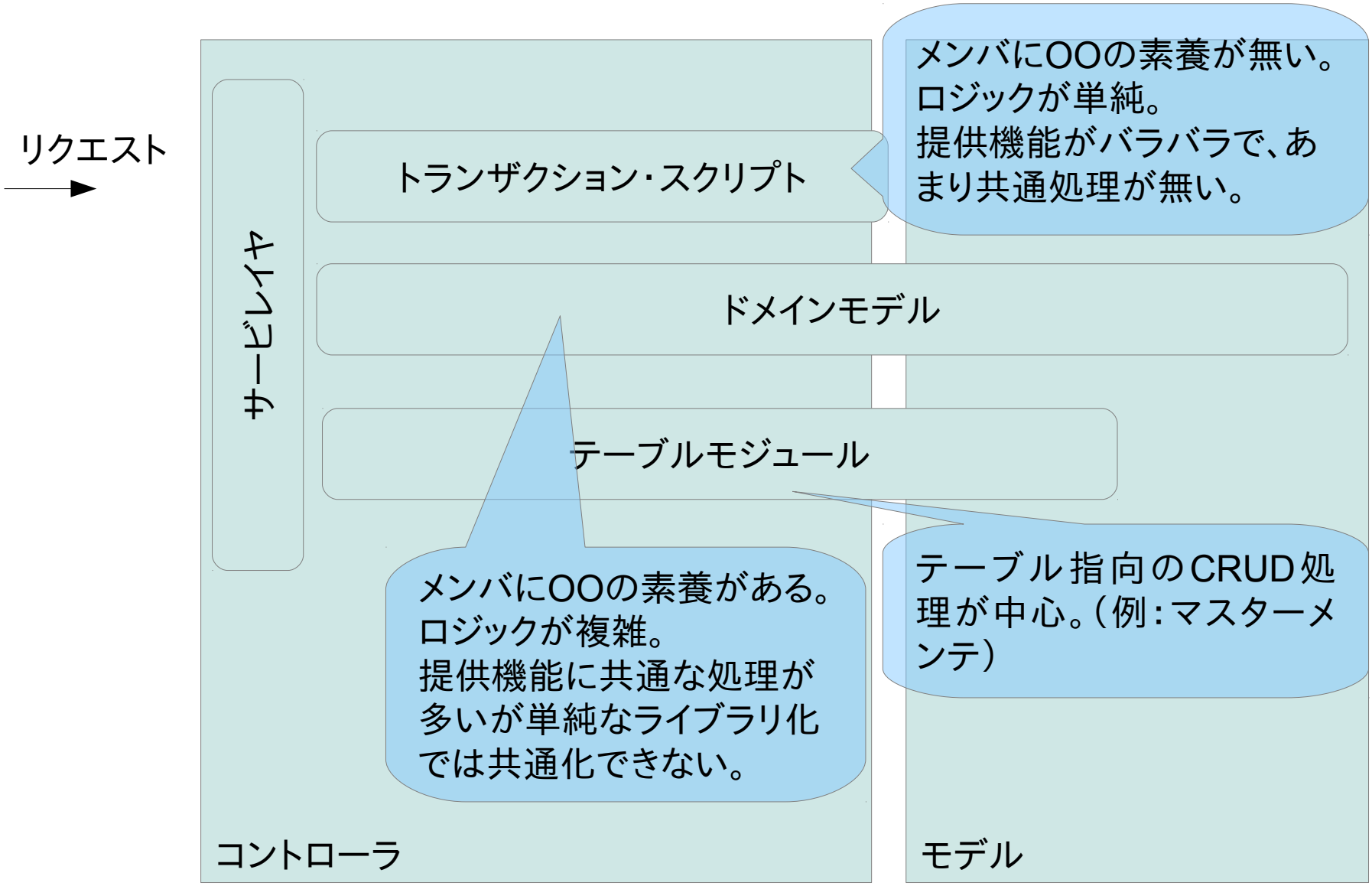
MVCパターン

- MVCパターンと、サービスレイヤパターンは、よほど小さなサイトでなければ必須と考えて良い。
- それ以外のパターンは要件によって取捨選択する。

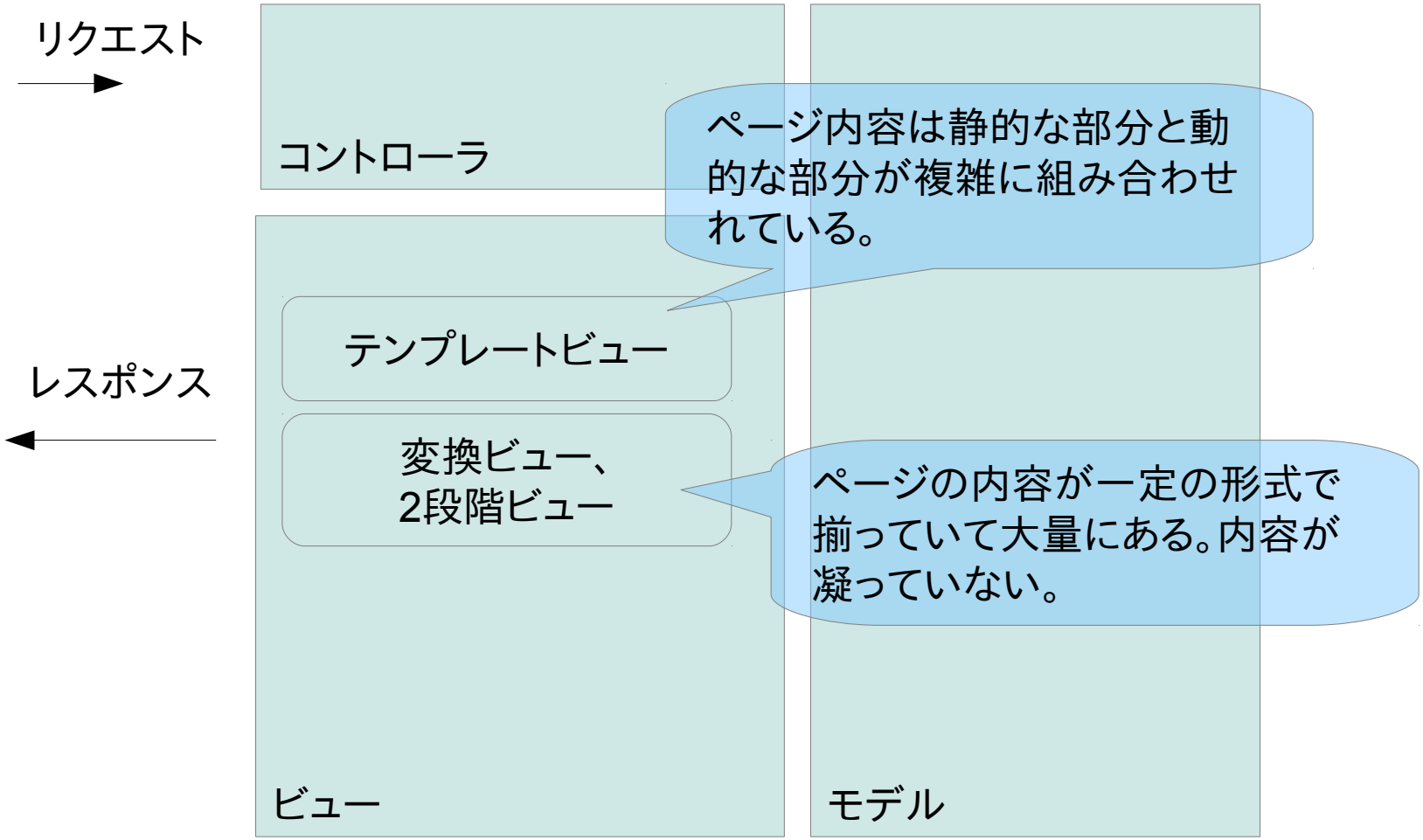
コントローラパターン


リクエスト
ページコントローラ
数ページの単純なサイト
フロントコントローラ
ページ数の多い単純なサイト
アプリケーション
コントローラ
複雑なページ遷移がある

ドメインロジックパターン



ビューパターン


リクエスト
コントローラ
ページ内容は静的な部分と動的な部分が複雑に組み合わせれている。
テンプレートビュー
レスポンス
変換ビュー、
2段階ビュー
ページの内容が一定の形式で揃っていて大量にある。内容が凝っていない。
ビュー
モデル

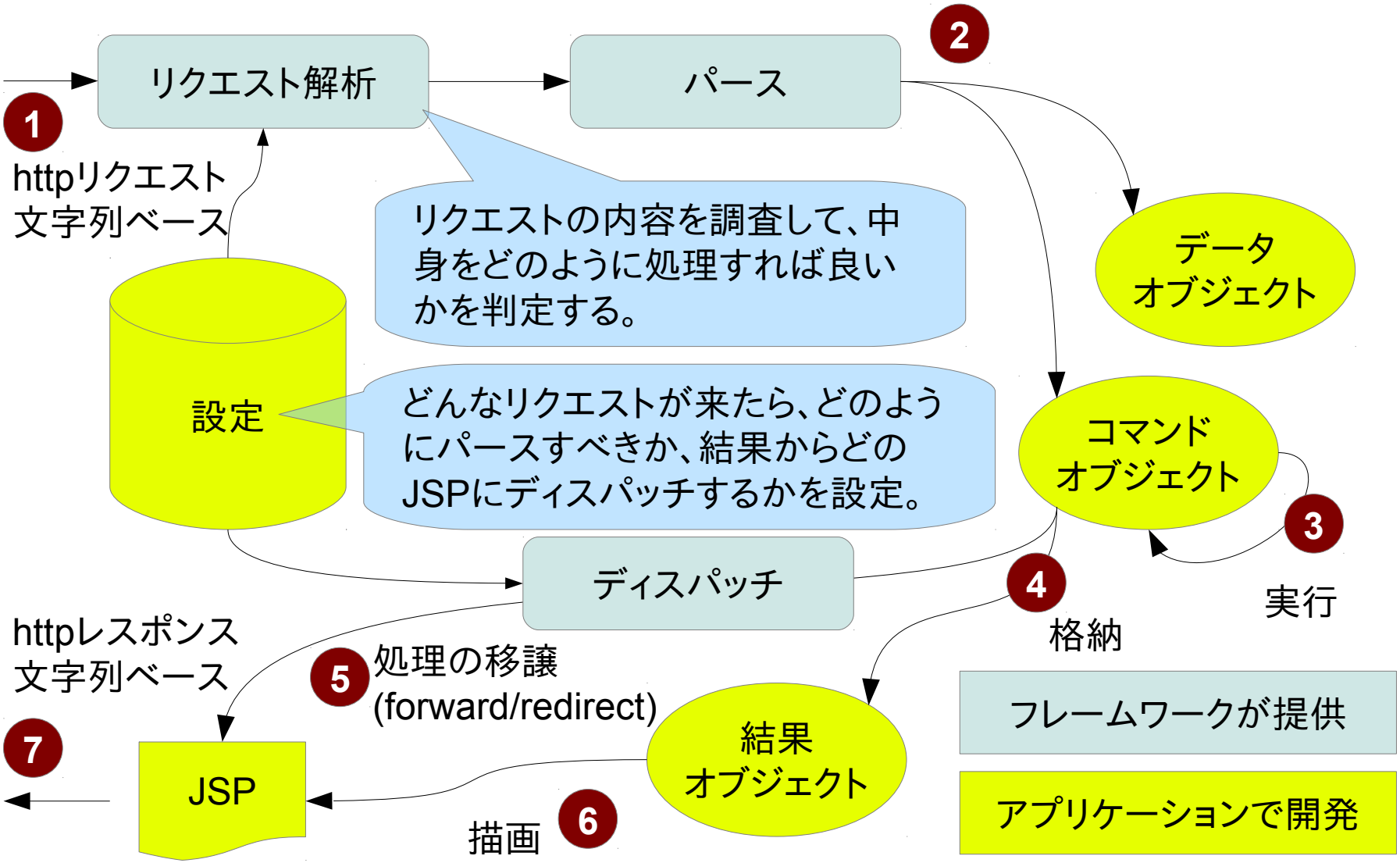
Webアプリケーションフレームワーク「第一世代Webフレームワーク」

第一世代フレームワーク

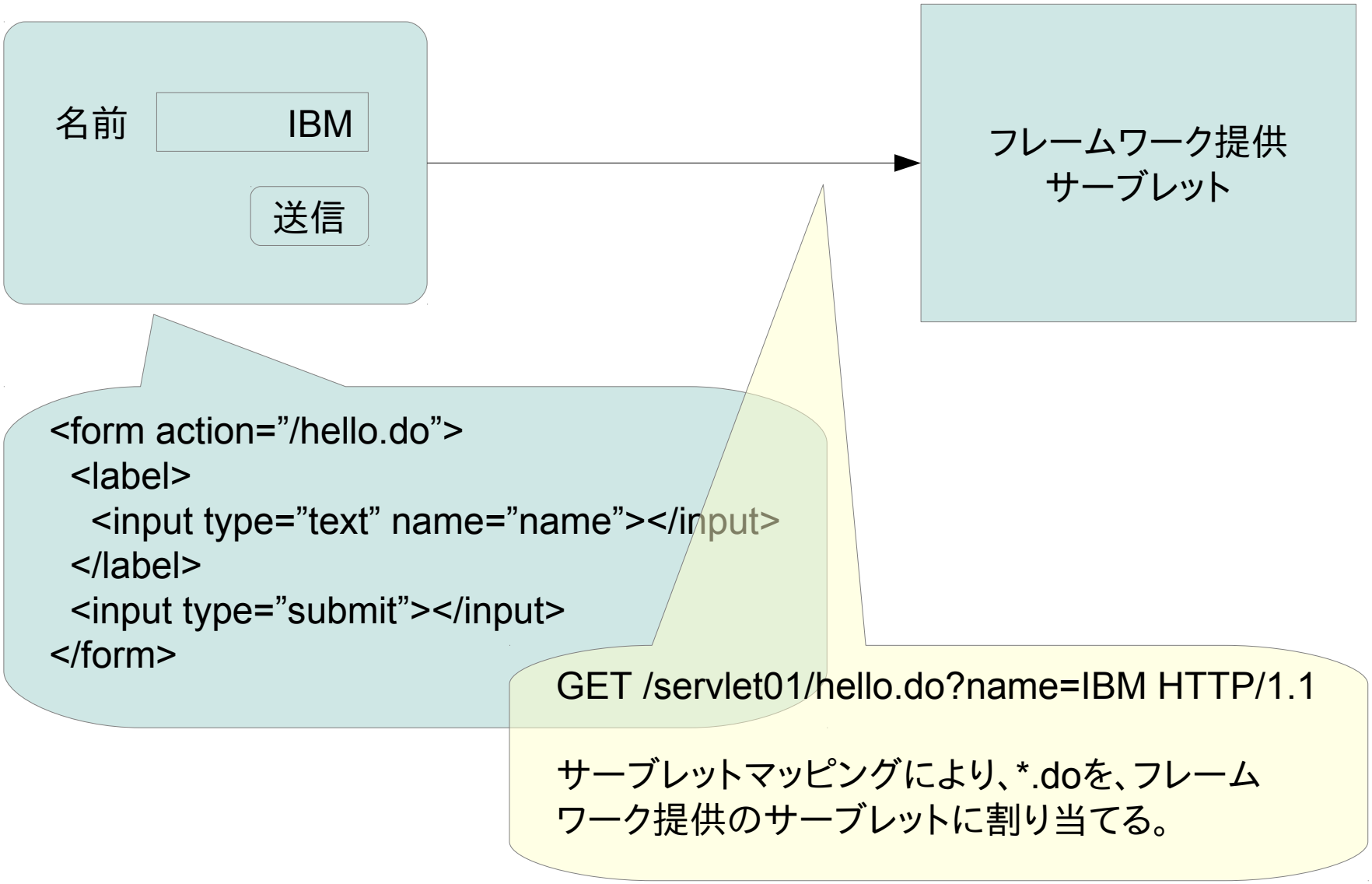
- サーブレットとJSPとの組み合わせ
  - サーブレットは、ロジックを記載するのには向いているが、画面(HTML)を構築するのには向いていない。
  - JSPは、画面(HTML)を構築するのには向いているが、ロジックを記載するのには向いていない。

それならサーブレットでロジックを実装し、表示でJSPを使用すれば良いのでは？

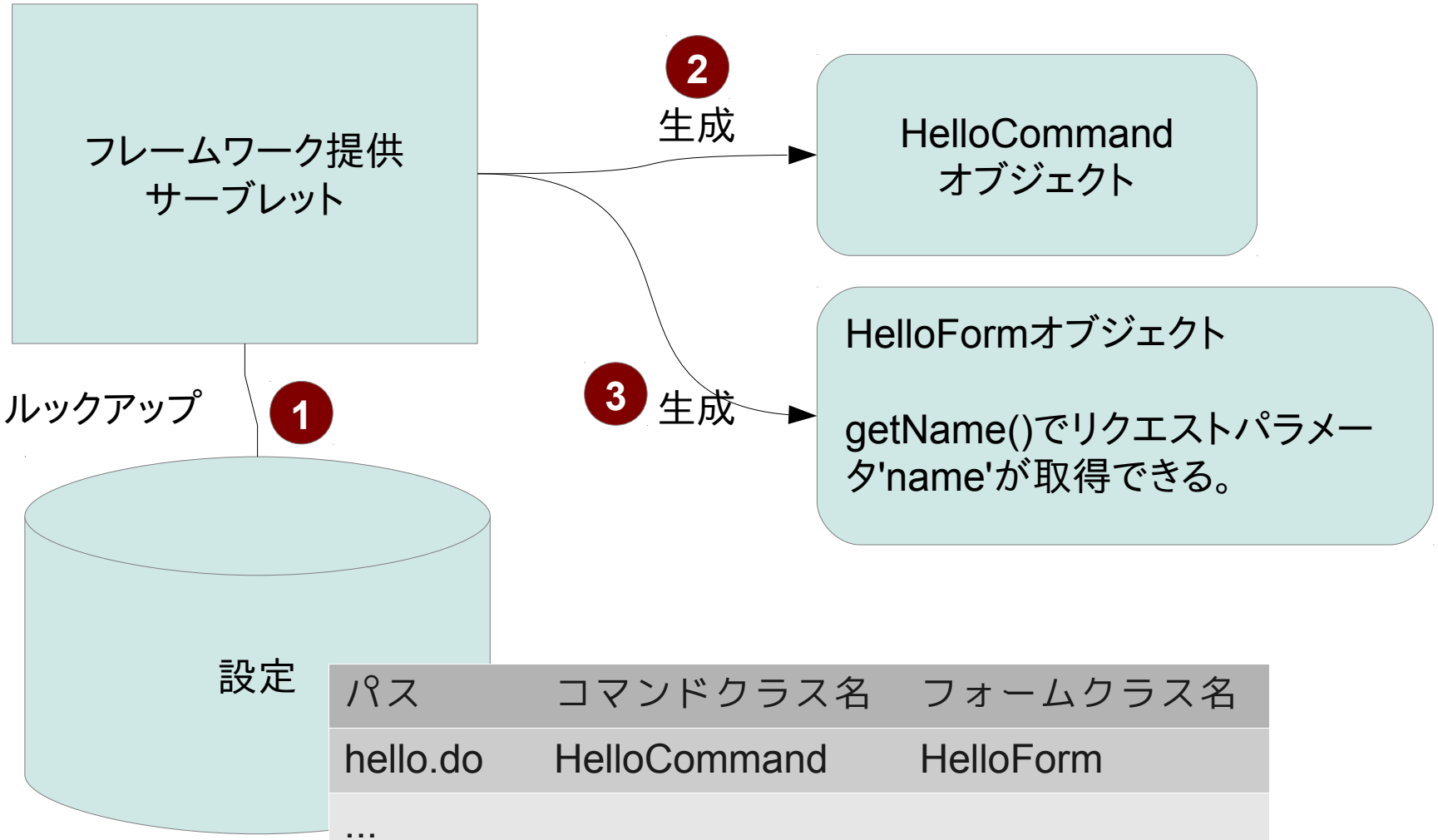
第一世代フレームワークの基本的な処理



第一世代フレームワークの動作例

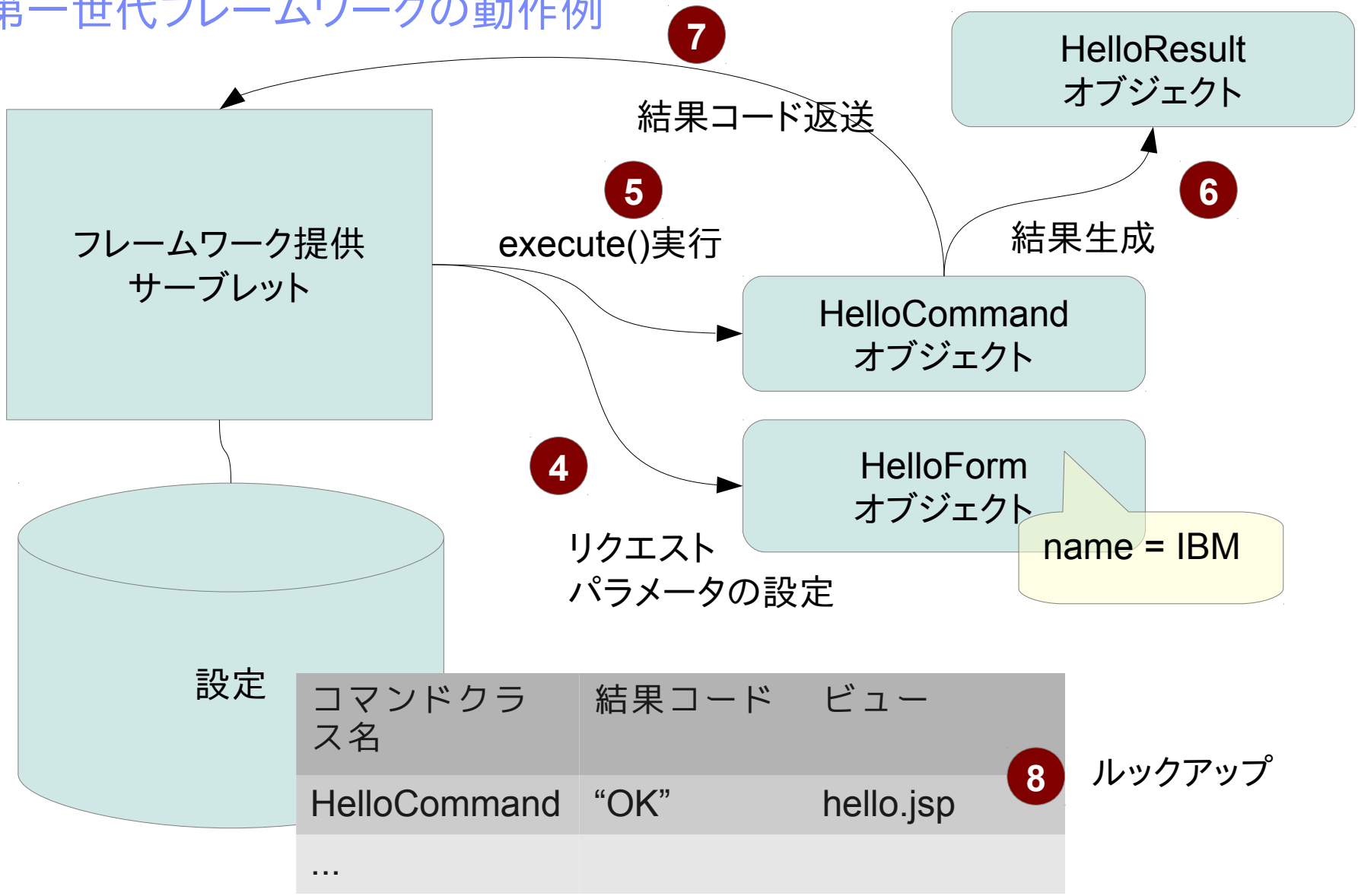


第一世代フレームワークの動作例

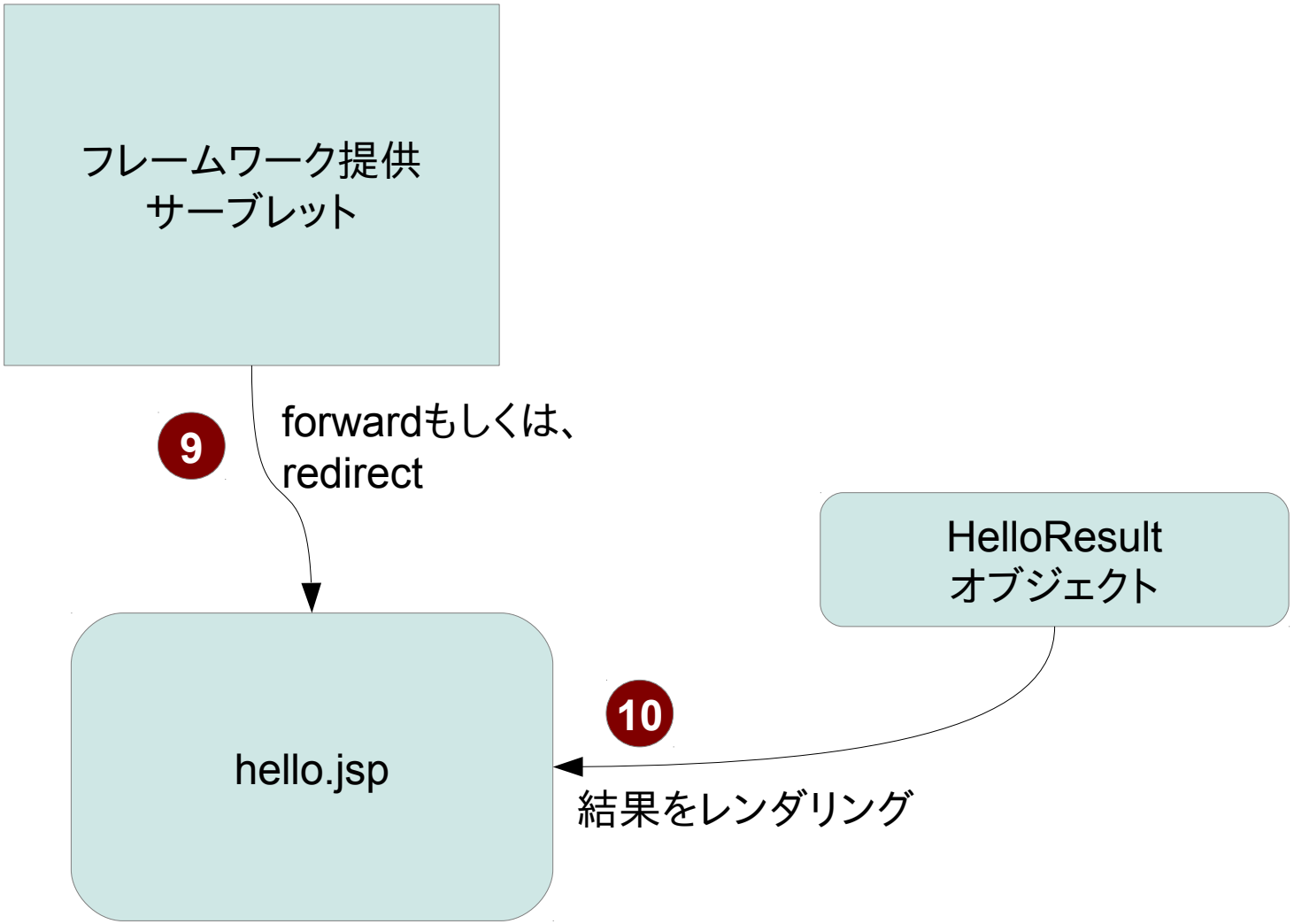


パス	コマンドクラス名	フォームクラス名
hello.do	HelloCommand	HelloForm
...		

第一世代フレームワークの動作例



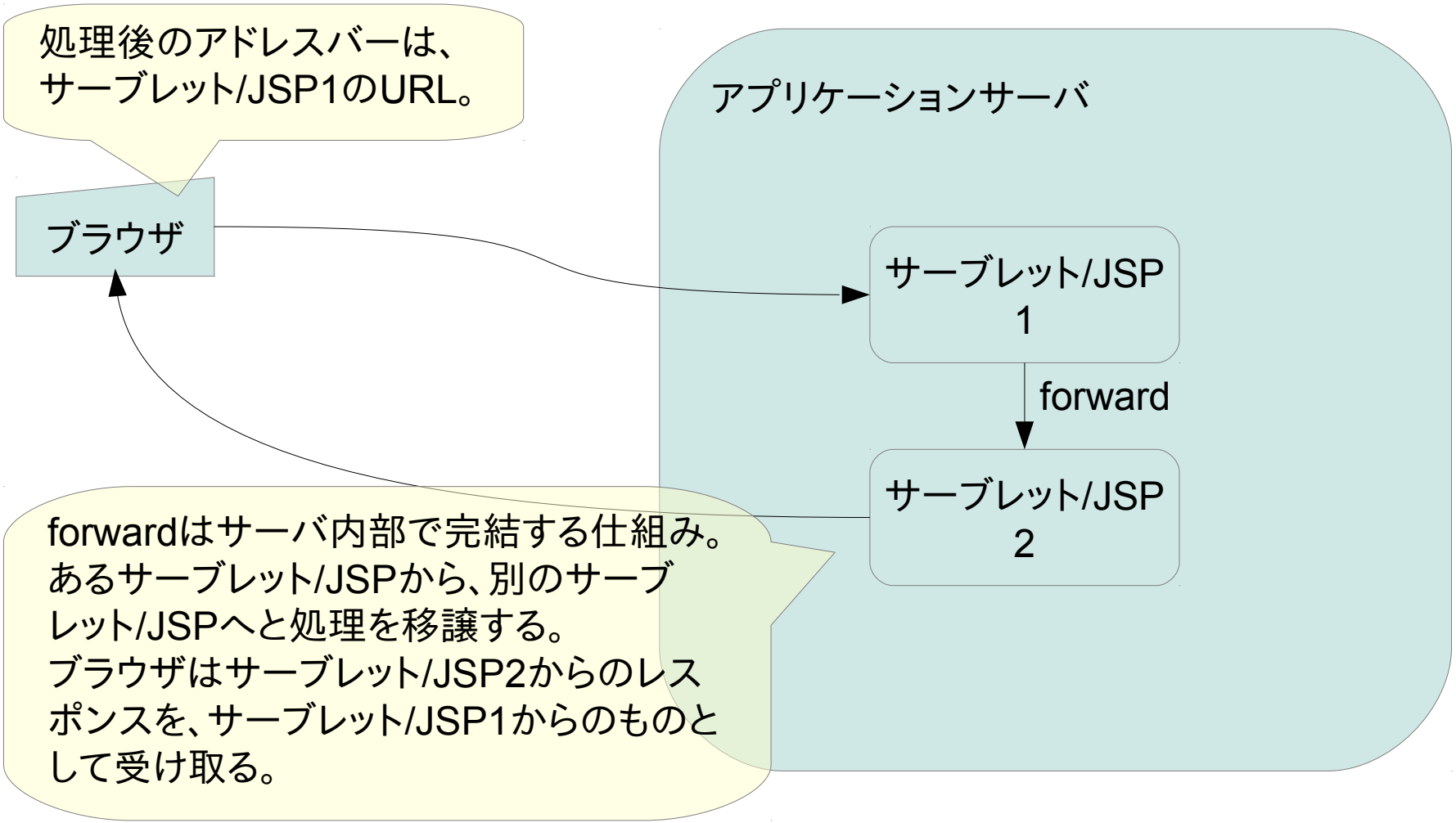
第一世代フレームワークの動作例



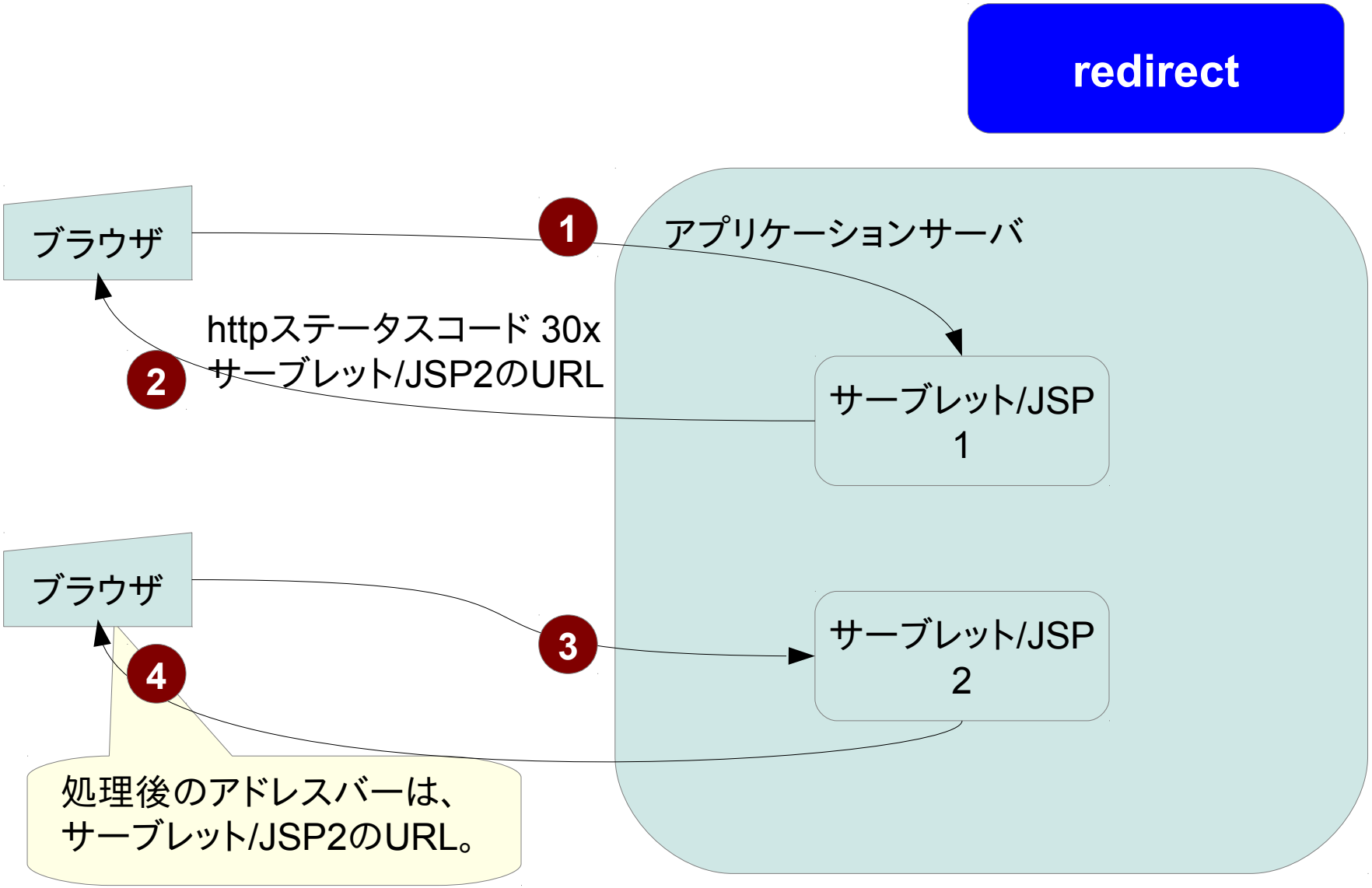
forwardとredirect

- どちらも、ある処理から別の処理へと処理を移譲する仕組み。

forward



forwardとredirect



まとめ(第一世代フレームワークのpros and cons)

- pros
  - サーブレットと、JSPの良いところ取り。
  - 複雑なワークフローを設定で集中管理。
  - リクエストパラメータへの簡単なアクセス(全てが文字列ベースのhttpに対して、数値にはintなどを利用できる)。
  - ユーティリティの提供(二度押し防止、ビューの共通化、例外処理など)。
  - JSP用タグライブラリの提供(分岐、繰り返しなど)。
  - クラスの分割が強制されることで設計が悪化しにくい(Command、Formなど)。
  - 当時はそれしか無かった。Struts1は爆発的ヒット。

まとめ(第一世代フレームワークのpros and cons)

- cons
  - やたらとクラスを書かなければならない(Command、Form、Result、Request... 今回の例には無いが更に取引などのモデルのクラスや、データアクセス用のクラス等々)。
  - XML地獄(設定をXMLに格納するものが多い)。コードよりXMLの方が多かったりする。
  - 当時は脆弱性への対応のためのサポートが不十分なものが多かった。
  - 細部の作りこみが必要で、GUIとの親和性があまり良くない(VBと比べて生産性が悪い)。ボタンをダブルクリックしてロジックを書いておしまいとは行かないし、一覧のような繰り返しデータを表示、ページ制御しようとすると一苦労となる。
  - データアクセス・パターンへのサポートが無いものが多い。