ソフトウェア工学 129－3
（2000．11．16）

Web アプリケーションフレームワーク UJI における GUI 構成部品のコンポーネント化技術

野村佳秀† 松塚貴英†

近年，Web アプリケーションの開発基盤技術として JSP/Servlet を利用することが多くなってきている．しかし，その画面構成部品の実装，再利用方法は，実装者によって異なり，何の指針も無しに他のアプリケーションで利用するには難しい．そこで我々は，Web アプリケーション開発，実行環境である UJI フレームワーク上で，画面構成部品のコンポーネント化を支援する以下の２つのフレームワークを開発した．

(1) オブジェクトモデルフレームワークは，部品の表現（レンダラ）とデータの入出力（モデル）を分離することで，それぞれを独立に再利用可能とした．

(2) スクリプティングフレームワークは，Web クライアント上で動作するスクリプト言語による，チェックロジックの呼び出し部と処理部を同時に記述することで，コンポーネント化を可能とした．

本論では，この画面構成部品のコンポーネント化を支援する２つのフレームワークについて解説する．

GUI Component of Web Application Framework "UJI"

YOSHIHIDE NOMURA † and TAKAHIDE MATSUTSUKA†

In recent years, we have been using often JSP/Servlet as a base technology of web application development. But it is difficult to reuse the gui component without any indication of its implementation. We provided two frameworks which are able to make a part of screen into reusable gui component.

(1) "Object Model Framework" makes gui component reusable independently by separating the presentation of screen (view) and data structure (model).

(2) "Scripting Framework" makes client side check logics which is written by a script language like JavaScript reusable by combining script calling part with check logics.

In this paper, we describe these framework technologies which support the reusable gui components.

1. はじめに

1.1 Web アプリケーションの利点

近年，インターネット利用人口の増加に伴って，Web ブラウザをクライアントとする Web アプリケーションとしてのシステム開発を行う場面が増えてきている．

既存のアプリケーションを Web アプリケーションに移行する利点としては，以下の点が挙げられる．

- クライアントアプリケーションが不要．
- スケーラビリティの確保が容易．
- サーバ側のみで機能追加，更新が行える．
- GUI の設計が容易．

また Servlet[5]，JSP[4] など，Java を利用した Web 技術も多く提案され，Web アプリケーション開発のための基盤技術は，ある程度整ってきた．

1.2 Web アプリケーション開発の問題点

近年，Servlet, JSP を用いた Web アプリケーションの開発指針となる J2EE Blueprint[3] が発表されている．この中で Web 層の設計方法についても詳細に解説され，Servlet, JSP の有効な利用方法が提唱されている．

しかし，Web アプリケーションの GUI 開発を行う際，未だ以下のような問題点がある．

- Servlet, JSP 等では，同一ソース内にビューとロジックが混在してしまうため，メンテナンス性が悪い．
- 同じ画面項目に対する値の表示と入力値の反映を，それぞれ別に処理する必要がある．
- Web クライアント上で動作するスクリプト（JavaScript 等）を利用したページを作成すると，メンテナンスが困難となる．
- レガシーアプリからの移行の際，GUI のデザインポリシーが変わる．（ダイアログ表示制御方式の違い，部品化の単位の違い，など．）

我々はこれらの問題を解決するため，Web アプリケーションの開発，実行フレームワークである UJI を開発した．

† (株) 富士通研究所
Fujitsu Laboratories Limited

2. Web アプリケーションフレームワーク UJI

UJI は JSP をベースとする，Web アプリケーションの開発，実行環境である．

UJI は，データ（Model），ビュー（View），コントロール（Control）を分離することにより，メンテナンス性を向上させる．また，それらの間をマップファイルで疎に連携させることにより，画面ごと，データ項目ごとの再利用を容易にしている．

UJI の実行エンジンは，フロントコンポーネントと呼ぶ JSP ファイルから，カスタムタグを利用して起動される．

データは JavaBean で表現され，基本的に JSP 内から `<jsp:getProperty/>` で参照する．また入力時は，フィールドに入力された値が，ページ毎に対応する JavaBean に自動的に設定される．

UJI は，ビューを構成する画面部品をコンポーネント化し，再利用するためのフレームワークを提供する．以下ではこのフレームワークについて解説する．

3. オブジェクトモデルフレームワーク

3.1 概　　要

UJI では，構造をもつ画面項目の入出力を行うためのフレームワークを提供している．これをオブジェクトモデルフレームワーク（以下 OMF）と呼ぶ．

JavaBean と JSP の `<jsp:getProperty/>` を併用することにより，簡単なデータとプレゼンテーションは分離することができる．しかし，繰り返しや階層構造など，データに構造がある場合，スクリプトレット（`<% .. %>` 内に記述する Java のソース）で出力の制御を行わなくてはならず，データとプレゼンテーションを完全に分離することは難しい．

OMF の基本的な考え方は，**図 1** のように，構造のあるデータに対応するビューを導入することによって，プレゼンテーション（レンダラ）とデータとを分離することにある．

つまり，データに対して簡単なインタフェースで制約を与え，このインタフェースに従って，ビューが自動的にデータの読み込みを行う．そして，その読み込んだデータの表現方法を，レンダラと呼ぶ JSP タグでカスタマイズさせることで，構造のあるデータの表示を行う．また，表示されたデータが更新された場合，アップデータと呼ぶクラスによって，同じインタフェースを通してデータの更新が行われる．

OMF は，JSP プログラムの以下の問題点を解決する．

- 繰り返しや階層構造など，構造を持つデータでも，ビュー（JSP）の記述内にロジックが入らなくなるため，メンテナンス性が良くなる．
- ビューやモデルの記述において，データの入力と出力を同時に記述することができる．

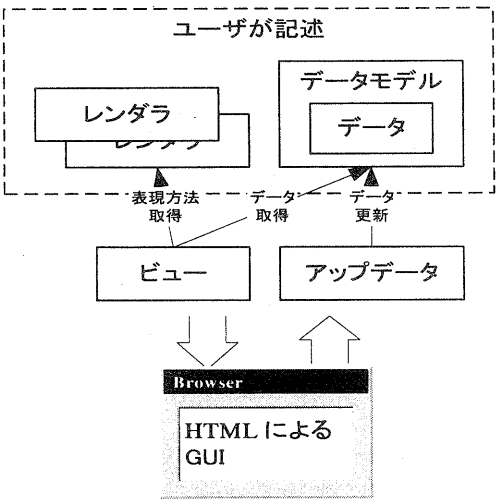

図 1　OMF の考え方

OMF の画面部品は，表現を行う JSP のカスタムタグ（ビュー）と，データを提供するモデルオブジェクト（モデル）で記述し，ビューとモデルは，モデルのインタフェースで連携する．そのため，同じインタフェースに関連するビューとモデルは，それぞれ別個に再利用可能となる．

3.2 内 部 構 成

OMF が標準で提供する画面部品は以下のものである．

テーブル	テーブル形式の入出力．
ツリー	ツリー形式の入出力．
リスト	リスト形式の入出力．
選択	選択形式のデータの出力．
	専用のモデルは持たない．
複合 Bean	入れ子形式のデータの入出力．
	専用のモデルは持たない．

以下ではリストモデルを例に挙げて詳細に説明する．

3.2.1 オブジェクトモデルの表示

リストモデルを構成するのは，ビューを表現する ListTag クラス，ListRendererTag クラス，およびモデルを表現する ListModel インタフェースである．

リストモデルのビューを表現する JSP の実際の記述例が**図 2** である．

```
<ul>
 <uji:list bean="main">
  <uji:listRenderer>
   <li><uji:value/></li>
  </uji:listRenderer>
 </uji:list>
</ul>
```

図 2　リストモデルのビュー記述例

ListTag クラス，ListRendererTag クラスは JSP のカスタムタグとして実装され，それぞれ <uji:list> タグ，<uji:listRenderer> タグに対応する．"uji:" は JSP taglib の名前空間名であり，通常の HTML タグと区別する以上の意味はない．

<uji:list> タグは表示するデータを保持するモデルの指定，およびレンダラの保持を行う．<uji:listRenderer> タグは，データの実際の表現方法を指定する．レンダラ内では，<uji:value/> というタグで，モデルから読み出されたデータを表す．

この例では， による項目列挙を，リストモデルを用いて出力する．<uji:list> タグで出力に用いるモデルを指定し，個々の項目の出力方法は <uji:listRenderer> タグ内に記述している．ここでは， タグ内に，リストモデルから取得したデータを配置している．出力に用いるモデルは，Servlet の HttpResponse に保存した際のキー名などを指定することで特定する．

モデル側は **図 3** のようなインタフェースを実装し，ビューからの要求に応じて，各項目ごとに表示すべきデータを返すように設計する．(**図 4**)

この例では，単純なリストの表現のみを扱っているが，図 3 の getElementAt() で，リストモデルの項目として，ListModel などのオブジェクトモデルを返すようにし，<uji:listRenderer> 内の <uji:value/> タグの代わりに<uji:list> などのリストモデル表現を記述することによって，オブジェクトモデルを再帰的に表現することも可能となっている．

```
public interface ListModel {
  public int getSize();
  public Object getElementAt(int);
  public String getElementClass(int index);
  public void setElementAt(Object value, int index);
}
```

図 3 リストモデルのインタフェース

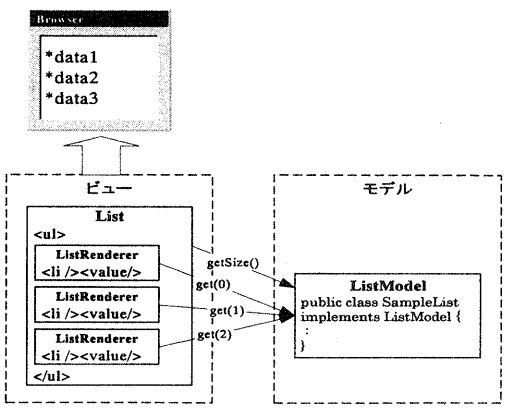

図 4 OMF の出力時イメージ

例として挙げたリスト表現を JSP のみで記述すると，**図 5** のようになる．この場合，一見，記述が短くなったように見えるが，埋め込まれた Java のコードを参照しなければリストであることを判別することができず，可読性が高いとは言えない．

```
<ul>
<% for(int i = 0; i < main.size(); i ++) { %>
 <li> <%= main.elementAt(i); %> </li>
<% } %>
</ul>
```

図 5 JSP のみを用いた例

3.2.2 オブジェクトモデルの入力

各モデルはデータ入力のためのインタフェースも備える．図 3 のリストモデルの例では，setElementAt(..) が入力のためのインタフェースにあたる．

入力を行う場合，ビューは **図 6** のように記述する．

```
<ul>
 <uji:list bean="main">
  <uji:listRenderer>
   <li>
    <input type="text"
           name="<uji:name/>"
           value="<uji:value/>" />
   </li>
  </uji:listRenderer>
 </uji:list>
</ul>
```

図 6 入力を行うリストモデルビューの記述例

OMF を利用した入力では，各項目に対して "uji.model." で始まる特殊な名前が割り当てられる．そして，特殊な名前を持つフィールドデータがサーバに送られると，フレームワーク側が自動的にモデルに対して入力されたデータを反映する．(**図 7**)

図 6 の記述のうち，<uji:name/> が特殊な名前を表すタグとなっている．入力に使用するモデルは，表示と同様に 図 3 のインタフェースを実装し，setElementAt(..) が呼ばれた際，適切な場所にデータを格納する．

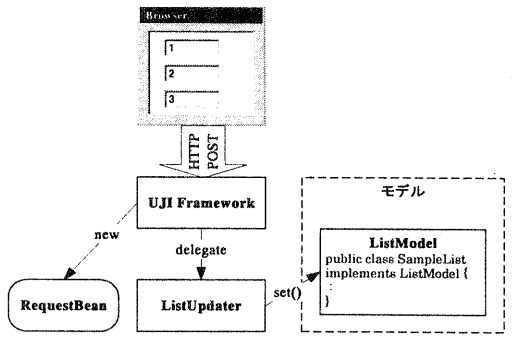

図 7 OMF の入力時イメージ

このように，構造をもつデータの入力と出力を同一インタフェースで行えるようになっていることで，より高度で，かつ再利用性の高い部品化が可能となる．

3.3 オブジェクトモデルの拡張

OMF の部品は，必要に応じて拡張を行うことができる．OMF の拡張は，以下のような場合に行う．

- 標準でサポートするモデルの組み合わせでは実現できないデータ構造の入出力を行う必要がある場合．
- 標準のモデルの組み合わせでは，記述が複雑になってしまうため，それぞれの記述のうち異なる部分だけをタグのプロパティとして与えて，記述をより単純化する場合．
- 既にデータ構造を表すインタフェースが定義されており，標準でサポートするインタフェースとの変換作業が複雑になってしまう場合．

拡張を行う場合，部品作成者は View, Renderer, Model を継承，作成する．

また，入力も制御する画面部品の場合は，Model へのデータのマッピングを行うための Updater も用意する必要がある．

3.3.1 Model

モデルインタフェースは，ビューを構築するために必要な情報を取得可能であれば，ユーザが自由に設計して良い．フレームワーク側では，特に制限は設けていない．

3.3.2 Renderer

レンダラは，ユーザが特にロジックを記述する必要はない．どのような種類のレンダラが存在するのか，などの制限事項を記述する．

3.3.3 View

ビューが一番重要で，ロジックが必要な部分となる．

ユーザは，モデルからデータ構造に応じてデータを取得し，そのデータを表現するためのレンダラを取得して，レンダラに出力を任せるまでのロジックを記述する．

モデルとビューとの関連付け，レンダラの用意，タグの出力などの部分は，フレームワーク側が自動的に行う．

また入力を行う場合，モデルの項目を特定可能となるようなキーを文字列表現で作成し，レンダラに渡す必要がある．

図 8 が，リストビューの実装の一部である．getElementAt() が，データの取得，getRenderer() がレンダラの取得，String name が，モデルの項目を特定可能とするキー，renderer.output() が，レンダラに出力を任せる部分となっている．

3.3.4 Updater

アップデータは，ビューで作成された，モデルの項目を特定するキーから，更新を行うモデルの項目を特定し，値の更新を行う．

フレームワーク側からは，対応するモデルのインスタンスと，該当するモデルのみに関連するキーと値との組が自動的に渡される．

```
public void output(Writer writer, String pre, Object obj) {
  ListModel model = (ListModel)obj;
  for (int i = 0; i < model.getSize(); i++) {
    String clazz = model.getElementClass(i);
    Renderer renderer = getRenderer(null, clazz);
    Object value = model.getElementAt(i);
    String name = pre + String.valueOf(i);
    renderer.output(writer, name, value);
  }
}
```

図 8 ビューロジックの記述例

```
<script language="JavaScript">
<!--
function submitFonc() {
 if (document.main.number.value.length < 10) {
  alert("10 文字以上入力してください");
  return false;
 }
 return true;
}
// -->
</script>
<form name="main" method="post"
      onsubmit="submitFunc()">
 <input name="number" type="text" />
</form>
```

図 9 通常の Script 記述例

4. スクリプティングフレームワーク

4.1 概　　要

UJI では，Web クライアント上で動作するスクリプトの記述，実行を支援するフレームワークを提供している．これをスクリプティングフレームワーク（以下 SF）と呼ぶ．

通常，何らかのイベントが発生した際にチェックロジックを実行したい場合，**図 9** のような記述が必要となる．

この記述の問題点として，以下の点が挙げられる．

- submit 時に呼び出される submitFunc 内から，input フィールドを静的参照しているため，フィールド名，フォーム名の変更，追加などがあった場合に，submitFunc を直接編集しなくてはならない．
- イベント発生時にどのフィールドがチェックされるのかは，チェックを行う関数を参照しなくては判断できない．

SF は，onclick, onsubmit など，イベント発生時にスクリプトの呼び出しを行う記述を含むプロパティ（以後，イベントプロパティと表記）と，実行するチェックロジックを含むスクリプト記述とを，1 つのカスタムタグに部品化することで，上記の問題を解決し，以下の効果をもたらす．

- JavaScript を併用した JSP ページのメンテナンス性を向上させる．
- JavaScript による入力値チェックなど，ロジックの呼び出し部，処理部を含めた部品化を行なえる．

```
<uji:form method="post">
 <uji:input name="number" type="text">
  <uji:action event="..submit">
   if (target.value.length < 10) {
    alert("10 文字以上入力してください");
    return false;
   }
  </uji:action>
 </uji:input>
</uji:form>
```

図 10　SF における JSP の記述例

図 10 が SF を利用した場合の，実際の JSP の記述例である．

図 9 と図 10 は，form の submit 時に，number フィールドの文字数を調べ，10 文字に満たない場合は警告ダイアログを表示する，という処理を行う記述となっている．

図 10 では，チェックロジックに対応する<sf:action>タグが，チェック対象の input フィールド (number) 内に記述されている．また，図 9 で記述されていた submitFunc に対応する記述がなくなり，form 内のイベントプロパティ記述もない．このような記述を可能とすることにより，チェックロジックを，チェック対象のフィールド，およびイベント発生元である form と分離し，メンテナンス性を高めている．

4.2　内部構成

SF の画面部品は，主に以下の 3 つの部分からなる．

コンテナタグ　form を表現する．

コンポーネントタグ　input, select など，イベントの発生元やチェックロジックの対象となるタグ．

アクションタグ　イベントの発生条件，イベントが発生した際に実行するチェックロジックなどを表す．

クラス構成は，**図 11** のようになっている．

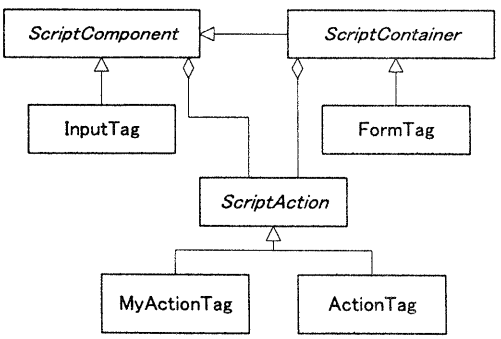

図 11　SF のクラス構成

図 10 では，<uji:form> がコンテナタグ，<uji:input> がコンポーネントタグ，<uji:action> がアクションタグとなっている．

また，<uji:action> タグの event プロパティで，イベントの発生元タグ，およびイベントの種類を指定する．この例では "..submit" となっており，これは，イベントの発生元が form であり，submit イベントが発生した際に input を対象としたチェックロジックを実行する，という記述となっている．

4.3　アクションの拡張

SF では，標準でチェックロジックを内部に記述する形のアクションタグのみ提供している．このタグは，記述を簡素にするためには利用できるものの，チェックロジックを部品化し，再利用するには適さない．

そこで SF では，アクションタグを拡張定義することにより，チェックロジックの部品化を実現可能としている．

図 12 が，実際にアクションタグの拡張を行ったクラスである．

```
public class MyTextActionTag extends CustomActionTag {
  protected int min = 0;
  public void setMinLength(int min) {
    this.min = min;
  }

  public void outputFunctionBody(Writer out) {
    out.write("if (target.value.length < " + min + ") {");
    out.write(" alert(\"" + min + " 文字以上入力してください\");");
    out.write(" target.focus();");
    out.write(" return false;");
    out.write("}");
  }
}
```

図 12　アクションタグの拡張例

アクションタグを拡張する場合，CustomActionTag というクラスを継承してクラスを作成し，outputFunctionBody() メソッド内でチェックロジックの出力を行う．チェックロジックの記述方法は，通常の <sf:action> タグ内に記述するものと同じになっている．

この例は，チェック対象の入力フィールドが特定の文字数に達していない場合に，警告ダイアログを出すという簡単なロジックを持つ．また，チェックする文字数をパラメータで変更できるようにし，パラメータに応じてチェックロジックを変化させるようなアクションタグとなっている．

このタグは，JSP の taglib 用定義ファイルを用意した後，以下のようにして利用する．

```
<sf:input type="text">
 <my:textAction event="blur" minLength="10"/>
</sf:input>
```

このように SF を利用して，クライアントスクリプトを，イベント発生元のフィールドや，チェックを行う対象のフィールドから分離して部品化することにより，保守性，再利用性を高めることができる．

5.　評　　価

5.1　オブジェクトモデルフレームワークの評価

OMF の評価のため，OMF を利用したテーブルビューの実装と，JSP のみを利用した実装を用意し，比較を行った．**図 13** が実際に表示までを行ったサンプルアプリケーションである．

以下に，このテーブルを記述する際の，通常の JSP との違いを示す．

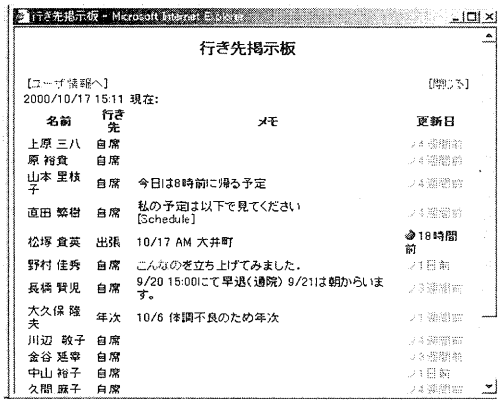

図 13 OMF のサンプル

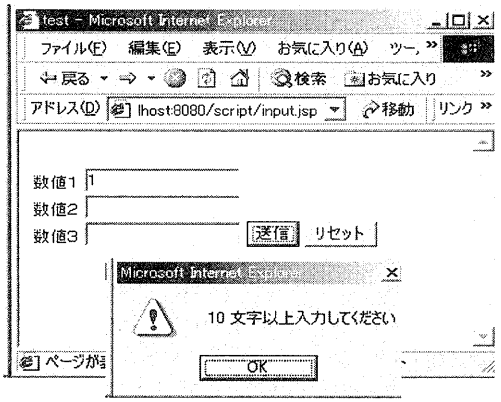

図 14 SF の使用例

	データ (lines)	JSP 表現 (tags)	JSP 内のロジック (tags)
JSP	33	13	9
OMF	43	10	0

データモデルの記述量，タグ数には大きな差はないが，JSP 内に記述されるロジックの量が大きく異なる．JSP 内に記述されるロジックは，基本的にページのデザインには無関係なものであり，保守性を著しく低下させる原因となる．

つまり，OMF を利用することで JSP 内のロジックを無くすことができ，保守性の向上につながると言う事ができる．

5.2 スクリプティングフレームワークの評価

SF の評価のため，送信ボタンを押した際に，3つの入力フィールドに対してチェックが実行され，それぞれについて警告ダイアログが表示されるようなサンプルを用意した．**図 14** がその実行イメージである．

以下に，SF を利用した場合と，JSP のみで記述した場合との違いを示す．

	記述 (lines)	スクリプト (lines)	イベントプロパティ (tags)
JSP	56	42	3
SF	33	15	0

この例では，form の submit イベントが発生した際，3つの入力フィールドのチェックを行う．実際に個々のフィールドのチェックを行うには，それぞれのフィールドチェック用スクリプトを呼び出すためのスクリプト記述が必要になるため，JSP のみの場合はスクリプト記述量が増えてしまっている．しかし，SF を用いると，同様の処理を行う場合も，スクリプトの記述量が増えることがない．

また，この例では，標準で提供されているアクションタグのみを利用しているが，チェックを行うための独自タグを定義することにより，スクリプト記述をさらに3分の1ほどに減らすことができると考えられる．

つまり，チェックスクリプトを使った画面を記述する場合，記述量を減らすには SF は大変効果的であると言える．また，イベント呼び出しを行うためにフィールドを表す input タグなどに挿入する，イベントプロパティ (onclick, onsubmit など) 記述を無くすことができるため，チェックスクリプトの再利用性も向上するといえる．

5.3 関連技術との比較

5.3.1 Struts (Apache)

Struts[7] は，XML によるマップファイルを利用して，プレゼンテーションを行う JSP と，データを表現する JavaBean とを関連付ける．この基本的な考え方は UJI に良く似ている．

しかし，画面項目の部品化という観点からは，Iterator を扱うカスタムタグなどがあるのみで，構造のある項目の再利用までは至っていない．また，スクリプトに関しては考慮されていない．

5.3.2 Brazil (Sun)

Brazil[1] は，Sun が提唱している Web アプリケーションの新しい構築方法で，Servlet よりも粒度の小さいハンドラと呼ぶプログラムで，ユーザからの要求を処理する．

また，ハンドラの動作を BML と呼ぶスクリプト言語で制御する．

`<if>` や `<foreach>` など，ある程度構造を持つデータの表現に関しても考慮してあるが，Brazil では再利用まで含めた部品化までは行えない．

5.3.3 SPFC (Apache)

SPFC[6] は，HTML のタグを JavaBean 化することで，AWT アプリケーションと同じように GUI を構成する．ユーザは GUI を構成する JavaBean とのインタラクションを記述することでアプリケーションを構築する．

アプローチはオブジェクトモデルフレームワークに似ているが，以下の点が異なる．

- SPFC は，GUI のプレゼンテーションまで，Jav-

aBean に対するプロパティとして記述する．つまり，データを提供する部分とプレゼンテーションを行う部分が分離されていない．

- リクエストは Listener インタフェースで受け取ることができるが，リクエストに対するレスポンスページを，リスナが直接用意しなくてはならない．

5.3.4 ECS (Apache)

ECS[2] は，HTML のタグ 1 つを 1 つのオブジェクトとすることで，Java コード内で HTML を扱うための API である．

構造を持つデータの扱いなどは，考慮されていない．

6. ま と め

我々は，JSP をベースとする Web アプリケーションの開発環境において，GUI を構成する画面部品のコンポーネント化を支援する，以下の 2 つのフレームワークを提案した．

オブジェクトモデルフレームワーク 構造をもつデータの表示とデータの保持を行う部品を，それぞれ別に定義することで，記述の独立性を確保し，また再利用性を高めることができた．

スクリプティングフレームワーク クライアントサイドスクリプトの呼び出し部と処理ロジック部を，まとめて部品化するフレームワークを提案．これによって，スクリプト部とコンテンツ部の独立性を保ち，また独自タグを簡単な記述で定義することにより，再利用性を高めることができた．

この 2 つのフレームワークを利用することにより，全体的なコード量が減少するのみではなく，保守性，再利用性が改善することを実証できた．

参 考 文 献

1) Brazil Project, http://www.sun.com/research/brazil/, Sun Microsystems (2000).
2) ECS Project, http://java.apache.org/ecs/, Java Apache Project (1999).
3) Java 2 Enterprise Edition Blueprints, Sun Microsystems (1999).
4) Java Server Pages, http://java.sun.com/products/jsp/, Sun Microsystems (2000).
5) Java Servlet Technology, http://java.sun.com/products/servlet/, Sun Microsystems (2000).
6) SPFC Project, http://java.apache.org/spfc/, Java Apache Project (1999).
7) Struts Project, http://jakarta.apache.org/struts/, Apache Jakarta Project (2000).