情報処理学会論文誌 Vol. 49 No. 7 2360–2371 (July 2008)

業務アプリケーションに適用するAjaxフレームワーク

松 塚 貴 英[†1]

近年，Ajaxという方式により，ブラウザの標準機能だけで高度なユーザインタフェースを提供する枠組みが注目されている．しかし，業務アプリケーションでAjaxを利用するためには，多くのJavaScriptコードを効率的に開発しなければならないなど，さまざまな問題がある．そのため我々は，効率的な機能拡張の仕組みや，MVCモデルをサポートし，大規模業務アプリケーション開発に適したフレームワークを研究し，開発した．

A Framework for Ajax-enabled Business Applications

Takahide Matsutsuka[†1]

Ajax gains public attention these days. Using Ajax, we can provide rich user interface using Web browsers without any extension. To apply Ajax for business applications, however, we have problems such as developing a lot of JavaScript code effectively. Therefore we analyzed current problems and developed an Ajax framework, which provides a function extending mechanism and MVC approach, for Ajax-enabled large scale business applications.

1. はじめに

1.1 背景

昨今，Web 2.0[1)]の技術の流れの中で，さまざまな技術が提案されている．なかでも，Webブラウザの標準機能のみを利用して高度なユーザインタフェースを実現するために，Ajaxという方式が利用され始めている．AjaxはAsynchronous JavaScript + XMLの略で，2005年にJesse James Garrettによって名づけられた．その技術の本質は，XMLHttpRequestというJavaScriptの命令によってユーザの操作を中断することなくサーバと通信を行いつつ，JavaScriptとHTML DOM（Document Object Model）の組合せによって動的に画面を書き換えることにある．Ajaxを使ったサービスで有名なものに，Zimbra[2)]やGoogle Maps[3)]などが存在する．

Ajaxアプリケーションの作成は，JavaScriptや非同期通信などの技術に関する深い知識を必要とするため，難易度が高い．その困難さを緩和するために，オープンソースをはじめとしたツールキットがいくつか提案されている．

これらのツールキットでは，Ajaxアプリケーションの効率的な開発のために，画面部品をリッチにしたり，非同期通信を容易にしたりする方法などを提供しているが，プログラミングの方法については言及しているものは少ない．そのため，有効な使い方をするためには，JavaScriptの広範な知識が必要になってしまっている．

我々は，今後Ajaxが業務アプリケーションのユーザインタフェースで使われていく際に，高い生産性を持つAjaxのフレームワークが必要と考えた．また，業務アプリケーションの開発におけるさまざまな性質をうまく取り込んでいく必要があるとも考えた．これらの視点で従来のAjaxツールキットを評価したところ，いくつか問題点が見つかった．本論文では，これらの問題を解決するためのフレームワークを提案する．

1.2 Ajaxの実現方式

Ajaxにはいくつかの実現方式が存在するが，開発の観点からみると，サーバ中心型とクライアント中心型に分類することができる．サーバ中心型アプローチは，XMLやJSPなどの定義体をサーバに配置し，それをあらかじめ，または実行時にHTMLとJavaScriptに変換してブラウザに送る方法である．代表的な例では，Google Web Toolkit[4)]があげられる．このアプローチでは，つねにサーバが動作しているか，開発環境を利用してプリコンパイルする必要がある．逆に，クライアント中心型は，HTMLまたはXMLなどの定義体をブラウザのJavaScriptで解釈させて，動的にDOMを操作して画面を生成する方法である．Prototype[5)]やscript.aculo.us[6)]，Dojo[7)]などの多くのオープンソースがこのアプローチをとっている．このアプローチでは，開発時にサーバは不要で，ブラウザで動作する定義体記述でのプロトタイピングが可能であるため，サーバを選ばないという利点がある．また，ブラウザで動作するJavaScriptを変更できるため，細かいカスタマイズがしやすい．

業務アプリケーションでは，高いカスタマイズ性が要求される．また，顧客によってサー

†1 欧州富士通研究所
Fujitsu Laboratories of Europe Ltd.

バアーキテクチャが異なる場合があるため，サーバを固定してしまうと適用できない恐れがある．さらに，2 章 **(4)** で議論するように，画面のプロトタイピングがブラウザのみで軽量にできることが望ましい．以上のことから，本フレームワークではクライアント中心型を採用した．

1.3　論文の構成

本論文の構成は以下のとおりである．2 章で，本フレームワークの対象となる業務アプリケーションのユーザインタフェース開発の性質について論じ，3 章で既存の問題点について考察する．それに基づき，4 章と 5 章でそれぞれ提案するフレームワークの設計と実現方式について議論する．6 章では問題解決についての考察を行い，7 章でまとめを行う．

2. 業務アプリケーション開発の性質

本論文での業務アプリケーションとは，主に企業内で利用する基幹系システムや情報系システムなどを指す．そのユーザインタフェースの開発には，以下に述べるような性質がある．

(1)　大規模化

業務アプリケーションは，画面数が多くなり，大規模化する傾向にある．そのため，部品数やコード量などが増えるため，それらを効率良く記述，管理する機構が必要となる．特に，Ajax アプリケーションではブラウザで入力チェックやリアルタイムなエラー表示などのさまざまな処理ができるようになるため，従来の Web アプリケーションよりも JavaScript のコード量が増える傾向にある．これらは，開発時は各画面や部品を分離して開発し，運用時は必要なものを 1 ファイルにまとめることにより通信回数を減らす工夫も必要である．

(2)　開発分担と標準化

開発分担をどうするかも課題である．たとえば，画面のデザインとその動作は別の担当者が開発するということがよく行われるが，そのためにはうまく分担ができるように仕組みを整える必要がある．

また，多くの開発者をかかえるプロジェクトでは，開発者全員が深い技術的背景を持つとは限らない．そのため，開発の標準化ができることが重要である．

(3)　部品のカスタマイズ性

クライアントサーバシステムやホストシステムなどでは，キーボードやマウス操作に対する細かいコントロールを行うことが多い．たとえば，単なるテキストフィールドであっても，数値のみ入力させたい場合や，あるフォーマットに従って空白部分を埋めるなどの制御が多く存在する．これは，部品数の増大を招く可能性がある．

また，ページ全体を 1 つのファイルとして開発するとは限らず，画面の部分を切り出して部品にしたい場合がある．たとえば，よく使われるマスタ検索部を別のファイルに記述するなどである．

(4)　顧客要求の獲得方法

多くの業務アプリケーションは，まず画面のプロトタイプを作成し，顧客との要求仕様を確定させていくことが多い．そのため，画面のプロトタイピングがしやすいことが望ましい．

3. 既存のフレームワークの問題

既存のクライアント中心型の Ajax フレームワークを評価し，2 章で論じた性質を持つ業務アプリケーション開発に適用させるうえで問題となる点を抽出した．なお，評価対象は Ajaxian.com で行われた 2006 年時点で多く使われているフレームワークの調査結果[8] より上位のものを抽出し，主に Prototype，script.aculo.us，Dojo とした．

3.1　クラス依存解決の問題

2 章 **(1)** で議論したように，業務アプリケーションはコード量も多くなる．そのため，既存の Ajax ツールキットの多くは，大量のコードを記述，管理するために，いずれも JavaScript 上でオブジェクト指向的なクラス記述を可能としている．**図 1** は，Prototype の記述例である．

この場合，サブクラス Subclass を定義するためには，スーパクラス Superclass の実体が必要となる．すると，クラスの依存解決のために，開発者が定義順をつねに考慮しなければならない．

この依存解決の問題に対して，Dojo や JSAN[9] では，クラスを複数のファイルに記述し，クラスを読み込み，評価したときに依存するクラスを**図 2** のように動的にロードする仕組みにより解決している．

図 3 は，Dojo の記述例である．依存するクラスを dojo.require 文で要求，取得すること

```
Superclass = Class.create();
Superclass.prototype = { ... };
Subclass = Class.create();
Subclass.prototype =
            Object.extend(new Superclass(), ...);
```

図 1　Prototype の例
Fig. 1　An example of Prototype.

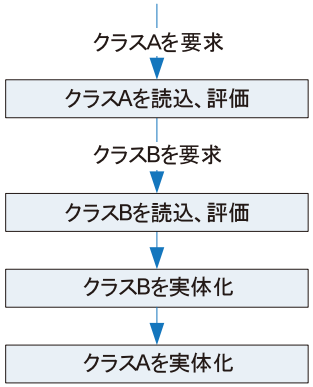

図 2　従来のクラスロード方式
Fig. 2　Class loading of existing toolkits.

```
[Subclass.js]
dojo.require("Superclass");
dojo.define("Subclass", Superclass, ...);
```

図 3　Dojo の例
Fig. 3　An example of Dojo toolkit.

ができる．ここで引数 “Superclass” は文字列なので，この時点では依存クラスの実体を必要としない．ここで取得した Superclass の実体を次の dojo.define 文で参照している．

しかし，require 文は 1 ファイル内での前方参照ができないため，任意の複数クラスを 1 つにまとめることが難しい．なぜなら，単純に複数クラスのファイルを連結しただけでは，前述の Prototype の例で指摘した問題と同様，定義順の問題が発生してしまうためである．

多数のクラス群を効率良く記述，管理するために，別々のファイルでクラスを開発可能なだけでなく，実行環境に合わせて任意の単位でまとめてロードできるための仕組みが望まれる．そのためには，クラスの依存関係の解決を，記述順，ロード順に関係なくできる必要がある．

3.2　ブラウザ依存コード

Ajax ツールキットでは，複数種のブラウザ上での動作をサポートする必要がある．このために一般的にとられる方法は，非互換箇所はブラウザ種別の判定を行ったうえで各ブラウザに依存するコードを実行するという実現方法である（**図 4**）．

```
MyClass = Class.create();
MyClass.prototype = {
    methodA: function() {
        if(IEの場合) {
            //IE依存のコード
        } else if(Mozillaの場合) {
            //Mozilla依存のコード
        }
    },
    ....
};
```

図 4　ブラウザ依存コードの処理例
Fig. 4　An example code of browser-depend processing.

この場合，methodA が実行されるたびにブラウザ種別の判定コストがかかるという問題がある．また，新しくサポートするブラウザを追加する際，他のブラウザで正常に動作しているファイルを編集することになるため，誤って正常なコードを変更してしまう可能性がある．

3.3　拡張部品定義の問題

通常，Ajax ツールキットでは，部品はクラスで定義する．このとき，新たな機能を持つ部品を作成する際は，まったく新しく作成するか，その部品に似た機能を持つ部品を探し，その部品を拡張して実装を行う．

すると，クラス定義の知識が必要となる．JavaScript では，Java などと異なり，クラス定義の方法がツールキットにより異なるため，それを覚える必要がある．業務アプリケーションの実装の際は，各画面の開発者にはなるべくクラス定義をさせずに開発ができることが望ましい．

また，同じ機能を複数の部品に実装したい場合に，部品数の増大を招くという問題もある．たとえば，テキストの入力を数値のみに制限したい場合，対象となる画面部品として，たとえばテキストボックス，テキストエリア，リッチテキスト入力部品がある．クラスを拡張する方式だと，この両方のクラスに数値入力制限機能を設けた新しいクラスを作ることになるため，元の部品（入力制限のないテキストエリアなど）と合わせると部品数が増えてしまう．

2 章 **(3)** で議論したように，業務アプリケーションでは，似ているが少しだけ異なる機能

を持つ部品が大量に必要になることが多い．その場合に，それぞれクラスの実装をしなければならない，または JavaScript でアドホックな追加ロジックを実装しなければならないのはコストが増加する要因となってしまう．

3.4　MVC の分離

大規模アプリケーションなどで，生産性や保守性を高めるため，Java や C++などのオブジェクト指向言語では MVC モデルを利用して作成することが普通に行われている．しかし，JavaScript 自身の大規模な活用が始められて日が浅いことなどから，従来のツールキットでは，MVC を分離するなど，プログラミングアーキテクチャに言及しているものはほとんどない．

2 章 **(1)**，**(2)** で議論したように，今後，Ajax を採用した業務アプリケーションなどでは，入力チェックなどブラウザ側において多くのプレゼンテーションロジックを実装することが予想される．そのため，従来の画面部品のツールキットの枠を超えて，体系化された実装の仕組みを持ち，分担された実装者が型にはまった開発ができるようなフレームワークが必要と考える．

3.5　開 発 環 境

現在主流で使われている Ajax ツールキットには開発環境が整っていないという問題がある．特に，タグが独特であるため，WYSIWYG での開発環境を構築するのが困難である．

4.　Ajax フレームワークの全体構造

3 章で述べた問題を解決し，業務アプリケーションを実装するのに適したフレームワークの設計を行った．

まず，1.2 節で述べたように，本フレームワークではクライアント中心型を採用している．そのため，フレームワーク本体は JavaScript で実装されるクラスライブラリとし，アプリケーションは HTML と JavaScript で記述する方式とした．

図 5 にフレームワークの全体像を示す．本フレームワークは次のコンポーネントで構成される．

4.1　クラスメカニズムとブートストラップ

クラスの基本構造やフレームワークの起動を管理する．クラスは実行時に動的に追加できるものとするため，クラスの動的ロードメカニズムを組み込む．また，依存するクラスやスーパクラスの記述をクラス定義の実行文から排除することにより，開発者が記述順を意識せずに済むようにする．

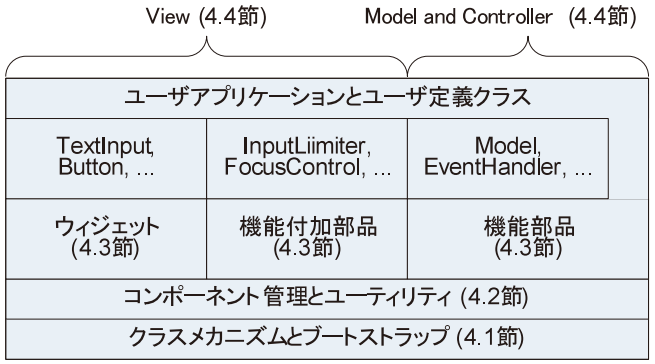

図 5　フレームワークの全体構造
Fig. 5　An overview of the framework.

4.2　コンポーネント管理とユーティリティ

上位のすべてのコンポーネントの基盤となるクラス群と，共通のユーティリティで構成される．ブラウザ種別により非互換が発生するクラスに対して，ブラウザ非依存の共通クラスと各ブラウザ依存のクラスを分けて記述する仕組みを導入することにより，ブラウザごとの if 文による分岐を不要にする．

4.3　ウィジェット，機能付加部品，機能部品

部品数の増大を抑え，最小のコストで高機能な部品を実現するための仕組みとして，部品を 3 種類に分割した．

部品は，ウィジェットと機能付加部品，機能部品の 3 種類から構成される．ウィジェットは，テキストフィールドやボタンなど，通常の HTML でも用いられるコントロールを含む画面に表示される部品である．機能付加部品はウィジェットに機能を追加する部品で，たとえば入力を整数のみに制限する機能，入力の自動補完をする機能，フォーカス移動を管理する機能などを実装する．機能付加部品がウィジェットと異なるのは，それ単体で画面に表示される部品ではないことで，必ず 1 個以上のウィジェットに関連付けられる．機能部品は MVC モデルを実現するための補助的な部品と，モデルを表現する JSON（JavaScript Object Notation），コントローラを実装する JavaScript オブジェクトから構成される．

4.4　MVC モデル

従来形式の Web アプリケーションでは，⟨form⟩ タグの内部に存在する ⟨input⟩ などのタグにより表示内容を保持し，それを submit することでサーバと通信を行っていた．

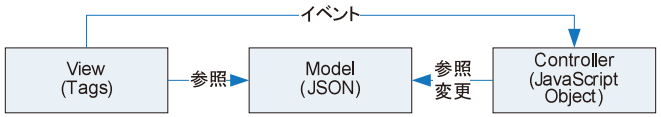

図 6　フレームワークにおける MVC の構造
Fig. 6　The structure of MVC in our framework.

しかし，Ajax アプリケーションにおいてブラウザ側でさまざまな処理をさせたい場合，画面に表示されている内容と，それをサーバとやりとりするためのデータとして扱うものが異なる場合がある．また，さまざまなタイミングで入力内容のチェックや変更を送信前に行いたいことがある．

そのため，ビューとモデルを分離し，モデルにはデータの値を検証するためのスキーマを関連付ける仕組みを考案した．ビューの属性（表示する内容など）はモデルの値を参照することにより，双方向の同期を実現させる．すなわち，モデルの値を変えれば参照しているビューの属性値が変わり，ユーザがビューに何かの操作をすれば，その変化が参照しているモデルに伝播する．これにより，コントローラではモデルの値のみを操作すればよいことになるため，コンポーネント間の独立性が高まる．

コントローラの起動に関しても考察する．もともと HTML では，onXXX という形式でタグ内にイベントハンドラを記述することができる．これは，小規模のアプリケーションでは記述が少なく有効な手段だが，大規模になるとタグが複雑になるため，可読性に悪影響を及ぼすほか，MVC の分離の原則にも反する．また，JavaScript においては addEventListener，または attachEvent という命令によりタグとは分離してイベントリスナを設定することができるが，実行文として記述するため，これも多数になると可読性が悪くなるという問題がある．

そのため，コントローラにはイベントの処理を宣言的に定義できる機構を導入する．これにより，ビューに “onclick” 属性などで直接ロジックが入り込むのを防ぐ．

MVC 全体の様子を**図 6** に示す．Model，View，Controller のそれぞれが明確に役割を分担し，参照方向も型決めされていることが分かる．

5. Ajax フレームワークの実装

4 章で述べた設計に基づき，フレームワークの実装を行った．

```
{
REQUIRES: ['foo.Superclass', 'foo.MyClass'],
CLASS_DECL: {
    CLASS_NAME: 'foo.Subclass',
    SUPER_CLASS_NAME: 'foo.Superclass',
    CLASS_INIT: function(classdecl) {},
    STATIC: {},
    INSTANCE: {
        message: 'Hello World!',
        sayHello: function() {
            foo.MyClass.baz(this.message);
        }
    }
}
}
```

図 7　クラス定義例
Fig. 7　An example of class definition.

5.1 宣言的なクラス記述

クラスを宣言的に記述するための記述法と，そのクラスをロードし，依存解決するための実装を行った．

図 7 では，スーパクラス Superclass を継承するサブクラス Subclass の定義をしている．ここで，依存するクラスの宣言（REQUIRES に記述）やスーパクラスの宣言（SUPER_CLASS_NAME に記述）は文字列リテラルで指定するため，評価時にスーパクラスの実体を必要としない．

また，このためのクラスロード方式として，従来のクラスファイルから直接クラスを生成する方法ではなく，未解決のクラスを定義体のままスタックに積んで解決する方法を用いた（**図 8**）．これは，次のように動作する．

まず，クラスが要求されると，findClass が実行される．ここでクラスがまだ存在しない場合，クラス定義が読み込まれ，REQUIRES による依存があるかどうかを確認する．依存があれば，そのクラス名をスタックに積む．スタックにクラスが存在すれば，再帰的に findClass を呼び出してクラスを取得する．

依存が解決できたら，スタックからクラスを取り除き，必要な場合にブラウザ依存クラスを読み込み（次節参照），クラスを実体化する．JavaScript におけるクラスは次のように構成されるため，定義体からの変換を行う．

foo.Subclass = function() { ... }
foo.Subclass[静的メンバ] = { ... }
foo.Subclass.prototype[動的メンバ] = { ... }

この時点では foo.Subclass が依存するクラスの実体が必要だが，それはすでに終わっているため，クラスの実体化は参照エラーなく実行することができる．図 7 のクラス定義を実体化した例を**図 9** に示す．

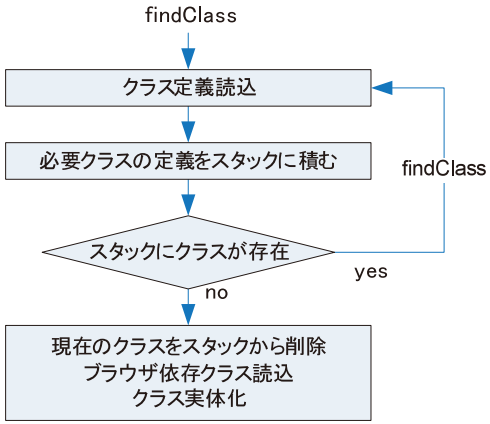

図 8　クラスローダの動作
Fig. 8　A procedure of class loader.

```
foo.Subclass = function() {
  foo.Subclass.prototype.initialize.apply(this, arguments);
}
foo.Subclass.prototype = {
  initialize : function() {}
  // foo.Superclass.prototypeの定義がここに入る
  message: 'Hello World!',
  sayHello : function() {
    foo.MyClass.baz(this.message);
  }
};
```

図 9　実体化したクラス
Fig. 9　A working class.

この仕組みにより，クラスの記述順，ロード順が任意でよくなる．また，複数のクラスを 1 ファイルにするには，クラスファイルを順不同で連結するだけでよいため，依存を考慮して定義順を決める必要がない．

5.2　ブラウザ依存コード

ブラウザ依存クラスを非依存クラスとは別クラスとして定義し，ブラウザ非依存クラスが実体化されるときに依存クラスが非依存クラスを上書きする仕組みを開発した．また，依存クラス名の命名規約を下記のように決め，ブラウザ依存クラスの発見を自動化した．

〈クラス名〉_〈ブラウザ種別〉[_〈ブラウザバージョン〉]

例を**図 10** に示す．ここではブラウザ非依存クラスに COMPATIBLE_SUPPORT というキーワードがあり，これが true に設定されている場合は，上記の命名規約によりブラウザ依存コードを探し，見つかったら非依存クラスへ定義の追加または置き換えを行う．具体的には，Internet Explorer® で実行する場合，BrowserEvent クラスの_registerEvent メソッドの定義が，クラスを実体化するとき（図 8 参照）に BrowserEvent_ie クラスの_registerEvent メソッドの内容で上書きされる．これにより，実行時はつねに Internet Explorer 用の_registerEvent

```
ブラウザ非依存クラス
CLASS_DECL: {
    CLASS_NAME: 'BrowserEvent',
    COMPATIBLE_SUPPORT: true,
    STATIC: {
        _registerEvent: function(...) {
            throw new Error('unsupported operation');
        }, // ...
    }, //...
}
Internet Explorer®依存クラス
CLASS_DECL: {
    CLASS_NAME: 'BrowserEvent_ie',
    STATIC: {
        _registerEvent: function(source, ...) {
            source.attachEvent(...);
        }
    }
}
```

図 10　ブラウザ依存コードの例
Fig. 10　An example of browser-depend code.

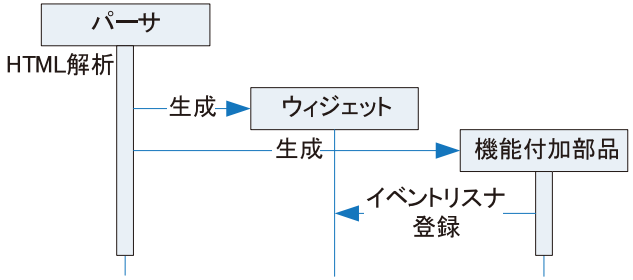

図 11　機能付加部品の初期化
Fig. 11　Initialization of function attachments.

メソッドが起動するため，メソッド内部でブラウザ判定をする必要がなく，そのコストもかからない．

この仕組みにより，ブラウザ非依存クラスの設計は依存部分の排除をメソッド単位で行うため多少難しくなる．しかし，ブラウザ依存クラス開発者は，他のブラウザの影響を考慮する必要がなく，ブラウザ依存クラスを非依存クラスからの差分で開発することができる．クラスの利用者も，インスタンスの生成を非依存クラスで行うため，ブラウザ依存クラスの存在を意識する必要がない．

5.3　機能付加部品

機能付加部品は，初期化の際に関連付けられるウェジェットにイベントリスナとして登録される．**図 11** に初期化時のシーケンスを示す．フレームワークが起動すると，フレームワークのパーサが HTML を解析して各部品を作成するが，機能付加部品が確実にウィジェットにアタッチできるように，初期化時はウィジェットの初期化を先に行い，その後機能付加部品の初期化を行う．すべてのウィジェットにはページ上で発生したユーザイベントに対するイベントリスナがアタッチできるようになっている．

機能付加部品の動作時のシーケンスを**図 12** に示す．対象のイベントがウィジェットで発生したら，機能付加部品が呼び出され，必要な処理を行う．

記述例を**図 13** に示す．この例では，i1，i2，i3，i4 という ID を持つ 4 つのウィジェットが定義されている．FocusManager という機能付加部品により，フォーカスが i1，i2，i4，i1，... という順で移動する．また，AutoCompleter という機能付加部品により，i2 は入力に応じて自動補完候補が i3 のセレクトリストに表示される．

機能付加部品の機構は，機能を付加する対象となる画面部品の種類を固定していないため，同じ機能を複数種の部品に付加することができる．

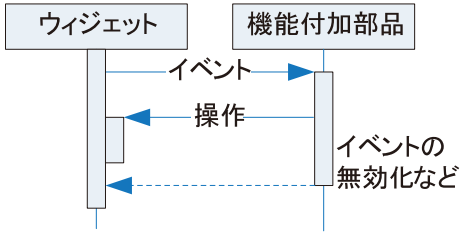

図 12　機能付加部品の動作
Fig. 12　Activation of function attachment.

```
<div rcf:id="i1" rcf:type="TextInput" ... ></div>
<div rcf:id="i2" rcf:type="TextInput" ... ></div>
<div rcf:id="i3" rcf:type="SelectList" ... ></div>
<div rcf:id="i4" rcf:type="CheckBox" ... ></div>
<div rcf:type="FocusManager" rcf:targets="i1;i2;i4"
rcf:rotate="true"></div>
<div rcf:type="AutoCompleter" rcf:input="i2"
rcf:list="i3" rcf:completer=”search”></div>
```

図 13　ウィジェットと機能付加部品
Fig. 13　Widgets and function attachments.

たとえば，「入力を数値に制限する」という部品が欲しい場合，従来方法ではテキストフィールド，テキストエリア，リッチテキスト入力部品など，ベースとなる部品ごとにクラスの拡張が必要となるが，本方式ではその必要がなく，「入力を数値に制限する機能付加部品」をそれぞれの画面部品に付加させればよい．この機能付加部品は「キー入力」のイベントをハンドルして，そのイベントで入力されたキーが数値ではなかった場合，イベントを無効化する（入力がなかったことにする）処理を行う．

5.4　MVC モデル

本節では，本フレームワークにおける JavaScript の MVC モデルの実現方式について述べる．

5.4.1　スキーマつきモデルとビュー

この仕組みでは，タグの属性から，JSON 形式のデータに対し，バインドを行うことがで

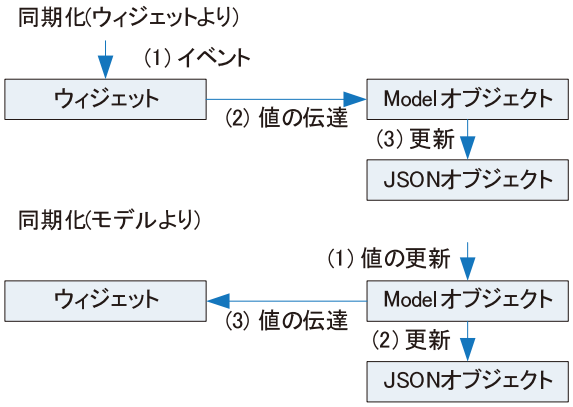

図 14　モデルとビューの同期
Fig. 14　Synchronization of a model and a view.

```
modelData = { name: 'abcde', price: 1000 };
modelSchema = { price: { type: 'integer' } };
<div rcf:id="model1" rcf:type="Model"
  rcf:object="modelData"
  rcf:schema="modelSchema"></div>
<div rcf:id="text1" rcf:type="TextInput"
  rcf:value="{model1.name}"></div>
<div rcf:id="text2" rcf:type="TextInput"
  rcf:value="{model1.price}"></div>
<div rcf:type="Formatter" rcf:target="text2"
  rcf:format="###,0"></div>
<div rcf:type="ValidationHelper" rcf:target="text2"
  rcf:events="blur; focus"
  rcf:onValidationSuccess="target.style.color='black'"
  rcf:onValidationError="target.style.color='red'"></div>
```

図 15　MVC を使ったコード例
Fig. 15　An example code of using MVC.

きる．バインドを行うと，値の変化にともなった双方向同期が自動的に行われる．バインドを行うためのタグの属性値の書式は，

rcf:属性名=”{ バインド先のモデルのパス }”

となる．属性値が ”{}” でくくられている場合がバインドされていることを示す．

図 14 にバインドされているウィジェットとモデルの値の同期の仕組みを示す．JavaScript の言語仕様上，JSON オブジェクトの属性を直接変更しても，それを契機として他の動作を行うことができないため，JSON モデルごとに Model というラッパオブジェクトを生成して，属性を変更する API を与えている．これにより，API 経由で値を変更すれば，その副作用としてバインドしている値の変更を行うことができる．

図 15 において，3 行目のタグ（rcf:type=”Model” で示されるもの）は機能部品の 1 つで，JSON オブジェクト modelData とスキーマオブジェクト modelSchema を model1 という名前のラッパオブジェクトとして参照可能とするためのものである．

これにより，テキストフィールド text1 は model1.name にバインドされており，入力内容は modelData の name 属性に反映する．また，modelData の name 属性を変更すれば，その結果が text1 に即座に反映される．テキストフィールド text2 は 3 桁ごとにカンマをつけるための Formatter 機能付加部品がアタッチされており，modelData の price 属性の値をフォーマットして “1,000” と表示することができる．

さらに，2 行目のスキーマ宣言では price は整数（type: ‘integer’）であると定義されている．スキーマによる検証は 2 種類の起動方法がある．

(1)　すべて検証

Model オブジェクトに対して検証の API（validate メソッド）を呼び出す．この場合，スキーマの内容すべてが検証される．

(2)　1 部品のみ検証

検証の契機となるイベントを，text2 コンポーネントに関連付けられた ValidationHelper 機能付加部品で定義する．text2 のフォーカスイン，フォーカスアウトが発生すると，text2 に関連する値のみスキーマに従って検証が行われる．

検証結果は，ValidationHelper 機能付加部品にある rcf:onValidationSuccess, rcf:onValidationError 属性で示される動作に従う．ValidationHelper 機能付加部品がウィジェットに関連付けられているため，同じモデルをバインドしている複数のウィジェットがあったとしても，検証結果に対する動作をウィジェットごとに別々に与えることができる．text2 コンポーネントのイベントによって起動した検証のシーケンスの例を**図 16** に示す．

スキーマはモデルに属するが，スキーマに従って検証を起動する契機になるのはユーザ操作などのビューのイベントである．また，検証結果を反映する先もビューである．そのため，これらのビューに関する定義は ValidationHelper 機能付加部品にまとめることにより，

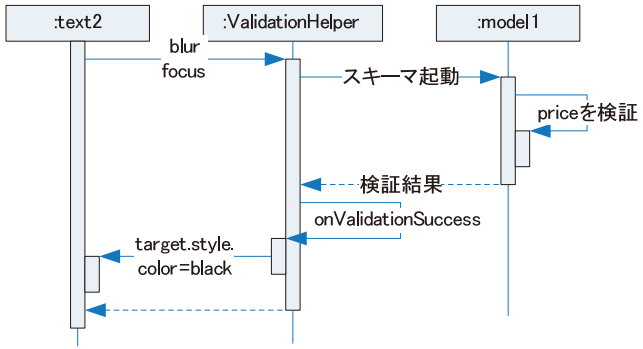

図 16　検証のシーケンス
Fig. 16　Validation sequence.

```
handlers = {
   MyText1 : {
      mouseover: emphasize,
      mouseout: deemphasize,
      click : eventListener
   },
   MyText2_click : function(event) {
      RCF.debug("MyText2 is clicked .");
   }
};
<div rcf:type="EventHandler" rcf:object="handlers"></div>
```

図 17　イベントハンドラ
Fig. 17　An example of event handler.

モデルに関する定義と混在しないようにしている．

5.4.2　コントローラ

コントローラの記述として，タグによる部品記述に加え，タグとは分離して宣言的記述を行うことによるイベントハンドラの定義を設計した．

イベントハンドラには，関数名を記述することもでき，直接実装を含めることもできる．**図 17** の例では，MyText1 というウィジェットに対して mouseover，mouseout，click というイベントに対するハンドラの関数名を設定している．また，MyText2 の click イベントに対してハンドラの実装を直接記述している．こちらは関数名が Visual Basic® のイベントハンドラの名前に近い．また，1 つの部品に定義するイベントハンドラの数が少ない場合はより短く記述することができる．前節で論じた検証結果に対応する動作（rcf:onValidationSuccess など）もここで記述することができる．

なお，最終行で使われている EventHandler という部品は，JavaScript オブジェクト handlers をイベントハンドラとして利用可能にするために必要な機能部品の記述である．具体的には，ハンドラの中身を解析してウィジェットにイベントリスナとして登録する作業を行う．

以上により，モデル，ビュー，コントローラをそれぞれ分離してアプリケーションの設計，実装を行うことができる．

5.5　動的ページ切替え

Ajax アプリケーションでは，規模が大きくなる場合に従来の Web アプリケーションであるようなページ単位の画面遷移を行わないことも多い．たとえば，Single Page Interface（SPI）という概念により，ブラウザ上の単位では 1 画面内だが，ある領域のみ表示を切り替えながらさまざまな処理を行うことがある．このとき，画面のある領域については動的にページを読み込みたいことがある．そのため，HTML を動的に読み込む仕組みをウィジェットとして用意した．

```
⟨div rcf:id=”fragment1” rcf:type=”FragmentContainer”
rcf:src=”fragment.html”⟩⟨/div⟩
```

このタグ定義では，fragment.html という HTML をフラグメントとして読み込む宣言をしている．しかし，この時点ではまだ HTML を読み込んでいない．必要になった時点で，

```
fragment1.activate();
```

を実行すると，実際にこの HTML が読み込まれ，表示可能な状態になる．最初の画面読み込みとは別のタイミングでこれを実行することで，多数の画面を必要とするときも，初期表示の性能を落とすことなく実行することができる．上記では読み込む HTML をタグで指定しているが，API を使うことで，読み込むファイルを動的に指定することも可能であり，読み込んだページを破棄することも可能とした．

6.　評　　価

4 章と 5 章で設計，実装したフレームワークに対し，いくつかの評価を行った．

6.1　実適用による業務アプリケーション要件に対する評価

本節では，2 章で述べた業務アプリケーションの要件に対して，本フレームワークがどのように対処しているかを検証する．この評価は，現在進めている実プロジェクトへの適用状

況からピックアップしている．

(1)　大規模化

本フレームワークでは，5.1 節で議論したクラス管理のシステムにより，クラス数の増大に対処しやすい．また，画面が複雑化した場合に，5.5 節で論じた動的ページ切替えの機能により，HTML を分割して開発することができる．

(2)　開発分担と標準化

多人数開発で開発分担がある場合，ばらばらに作成された多数のクラス群を管理しなければならない．この場合も，5.1 節で議論したクラス管理が有効である．また，ブラウザに依存するコードを記述しなければならない場合に，5.2 節で議論した仕組みにより，if 文を使わずにブラウザごとの独立したコードを差分で記述できるため，管理が容易であることが確認できた．

さらに，5.4 節の MVC モデルにより，実装部分を標準化，極小化できるため，大規模アプリケーションの開発管理が容易になっている．特に，ロジックの実装がイベントハンドラに集約されるため，フレームワークを外れた実装は極力起こらないようになっている．

(3)　部品のカスタマイズ性

5.3 節で論じた方法により，さまざまな制御を持つ機能を機能付加部品としてテキストフィールドなどのウィジェットとは分離して作成し，それをアタッチすることでさまざまな機能を実現することができる．本フレームワークの実装でも，日付や文字列の穴埋め入力や，表示の自動フォーマット，フォーカス移動など，業務アプリケーションで利用される主要な機能を機能付加部品としてサポートすることができた．

また，5.5 節の仕組みにより，多くの部分から構成される画面について，動的に切り替えつつ表示させることが有効であることが分かった．

(4)　顧客要求の獲得方法

クライアント中心方式にしたため，HTML のみでプロトタイプができ，画面の見た目自体も Ajax 機能を有効にした状態で作成し，ブラウザのみですぐ確認することができる．これは JSP などで開発する場合につねにサーバが必要となるのに比べると大きな優位点である．

6.2　試作によるコード量削減率

1 つの例として，Google Maps と JavaServer Faces に付属している Cardemo サンプルをマッシュアップして，言語切替えや非同期サーバ問合せによる動的な価格変更，郵便番号の入力による自動補完機能などを盛り込んだアプリケーションを試作した．

試作したアプリケーションの規模を**表 1** に示す．ここから，以下が読み取れる．

表 1　Cardemo サンプルの行数
Table 1　LOCs of Cardemo example.

フレームワーク	なし(行)	あり(行)	削減率
ロジック	733	276	62%
メッセージリソース	446	446	0%
モデル	0	55	N/A
タグ	260	185	29%
全体	1439	962	33%

- JavaScript ロジックの行数がおよそ当初の 4 割程度と，大幅な削減となった．
- モデルが新たに追加された．
- 画面部品により，タグの抽象度が上がっているため，3 割程度行数が減っている．

6.3　実装における問題点

一方，実装において以下の問題が浮上した．

ブラウザごとの部品の作成は，ブラウザ依存クラスを切り分ける仕組みがあるため差分的に行えるが，スタイルシートの挙動の違いが予想以上であり，複雑な部品において実装に困難を要した．たとえばカレンダー部品は 1 カ月の表示フォーマットを行うために表示に利用する要素が多く，ブラウザごとのスタイルシートの作成に苦労した．これは，スタイルシートの挙動を吸収するライブラリが存在しないことに起因する．これについては現在検討中である．

初期表示性能が当初あまり良くなかった．これは画面定義をブラウザ側で解釈表示したり，MVC モデルの初期化を行ったりしているためである．そのため，さまざまな性能改善策を講じ，初期表示性能を他の Ajax ツールキットと同様まで引き上げた．

6.4　他の Ajax ツールキットとの比較

クラス定義については，開発者は定義順を意識しなくてよい．また，Java における jar ファイルのように，別々のファイルとして作成したクラスを，後から任意の単位でファイルにまとめることが可能であり，Dojo などより制約が少ない．

部品は，機能付加部品を活用することで，画面表示そのものと付加機能を分けて作成することができ，ウィジェットとの組合せにより高機能の部品を実現することができる．他のツールキットでは，これらの機能がないため，部品をそのつど作成するか，JavaScript の外付けロジックで対応する必要がある．

本フレームワークの MVC 方式により，コントローラはイベントハンドラに集約され，そ

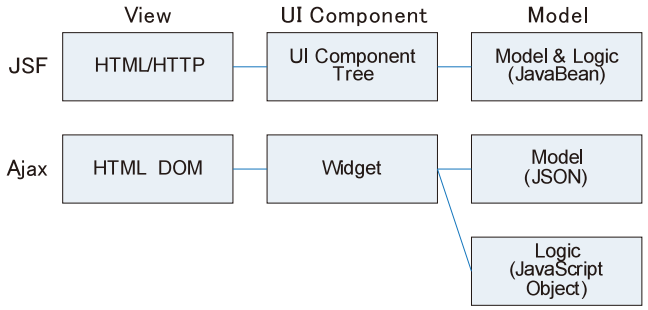

図 18　JSF との対応
Fig. 18　Relation between JSF and our framework.

の中でモデルを操作したり画面に入力したりすることによってビューとモデルは自動同期する．この機能は他の Ajax ツールキットに見られない特徴であり，大規模化する傾向のある業務アプリケーションの開発が容易になっていることが実アプリケーションの開発部隊より報告されている．

6.5　サーバフレームワークとの比較

Ajax では類似したフレームワークはないが，サーバのフレームワークでは JSF との類似性が認められる．すなわち，ビュー，UI コンポーネント，モデル・ロジックのそれぞれが JSF と対応する（**図 18**）．

いずれもイベントベースで，View からモデルにバインディングを行って値を同期する点も似ている．

JSF はサーバフレームワークだが，Ajax はクライアントで動作するため，発生したイベントはリアルタイムでモデル・ロジックに伝えられる点が異なる．この特性を生かして，Ajax フレームワークと JSF を協調動作させることも可能と考える．

6.6　通信とサーバフレームワーク

本フレームワークはクライアント中心型のため，サーバを選ばずに利用することができる．しかし，実際にはサーバとの協調により動作させるのが普通である．

サーバについては，我々の開発したフレームワーク UJI[10] があり，すでに多数の業務アプリケーションにおける実適用実績もあるため，非同期でこのフレームワークと通信できるようにした．

なお，通信部分については完全に他部分と独立しているため，DWR[11] などの他のフレームワークを利用することもできる．UJI，DWR とも，JSON オブジェクトを直接送受信できるため，MVC モデルを採用している本フレームワークとの相性が良いことが，試行結果から分かっている．

7. おわりに

7.1　まとめ

Ajax 技術を業務アプリケーションに適用するために，以下のような特徴を持つ Ajax フレームワークを考案，開発した．

- クライアント中心アプローチ
- 定義順を意識する必要がない宣言的クラス定義
- ブラウザ非互換を吸収するための仕組み
- ウィジェットと機能付加部品の分離
- MVC の分離とスキーマの仕組み
- 画面の動的ロード

現在，いくつかの大規模な業務アプリケーション開発プロジェクトにおいて実適用を始めており，フレームワークの効果を確認している．現在はまだ途中段階だが，Ajax の適用が初めてのプロジェクトでも，容易に適用できることが報告されている．

7.2　今後の課題

3.5 節で提起した開発環境と，6.3 節で発見されたスタイルシートの問題については，本論文では解決しきれなかった．現在，画面作成を容易にするため，クライアント中心型の特性を生かし，フレームワークで実装したウィジェットを直接使って WYSIWYG 編集を可能とする開発環境の研究を行っているほか，スタイルシートの違いを吸収するためのさらなる仕組みを模索中である．

また，開発効率化のため，Ajax を使わない Web アプリケーションにおいては，MDA 化の試みが行われている[12)–14)]．しかし，Ajax アプリケーションでは 1 画面内でのインタラクションが多いため，そのまま適用することが難しい．Ajax アプリケーションの画面やインタラクションのモデルをどのように記述するかが課題である．

さらに，Ajax アプリケーションのテスト，検証法について検討している．リグレッションテストを行う場合，通信の非同期性や，HTML ではなくブラウザ内部の文書構造である DOM レベルでの画面変化を考慮する必要がある．このため，非同期な文書構造変化により記録時と再生時でタイミングが変わってしまうような場合に対応できるテスト方法を研究し

ている．

参 考 文 献

1) O'Reilly, What Is Web 2.0 (2005). http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html
2) Zimbra. http://www.zimbra.com/
3) Google Maps. http://maps.google.com/
4) Google Web Toolkit. http://code.google.com/webtoolkit/
5) Prototype. http://www.prototypejs.org/
6) script.aculo.us. http://script.aculo.us/
7) Dojo toolkit. http://Dojotoolkit.org/
8) Ajaxian.com, Ajaxian.com 2006 Survey Results. http://www.surveymonkey.com/DisplaySummary.asp?SID=2402465&U=240246533425
9) JavaScript Archive Network. http://www.openjsan.org/
10) 松塚貴英，野村佳秀：JSP を用いた Web アプリケーション構築のためのフレームワーク UJI，情報処理学会研究報告 2000-SE-129 (2000).
11) Direct Web Remoting. http://getahead.org/dwr/
12) Matsutsuka, T.: Model-Driven Development Approach to Web Applications, *The IASTED International Conference on Software Engineering 2005* (Feb. 2005).
13) 松塚貴英，阿草清滋，山本晋一郎：ラウンドトリップエンジニアリングを目指した Web アプリケーションのための意味モデル，情報処理学会論文誌，Vol.46, No.5 (2005).
14) 堀　雅洋，田井秀樹：モデルに基づく Web アプリケーション開発，情報処理学会誌，Vol.45, No.1, pp.16–21 (2000).

(平成 19 年 10 月 11 日受付)
(平成 20 年 4 月 8 日採録)

松塚　貴英（正会員）

1971 年生．1994 年東京工業大学電気電子工学科卒業，1996 年東京工業大学大学院情報理工学研究科計算工学専攻修士課程修了．同年富士通株式会社に入社．現在，Fujitsu Laboratories of Europe Limited 所属．分散企業システム，Web アプリケーションフレームワーク，ソフトウェアアーキテクチャ等の研究，開発に従事する．2001〜2002 年米 Carnegie Mellon University 客員研究員．