部品化を指向したクライアントサイド Web アプリケーション開発の枠組の設計と実装

山本 竜太郎　岩崎 英哉

GUI アプリケーション開発において, UI を構成する要素を適切に部品に分割することは, 部品の再利用性を向上させ, アプリケーションの開発効率を向上させる上で重要である. 一方, クライアントサイド Web アプリケーション開発で用いられる HTML や CSS といった言語は, 名前空間や依存関係管理の機能を言語仕様として持っていない. そのため, 部品化を行っても, 異なる部品間で名前が衝突するという問題が生じ, 部品への分割の利点を十分に得ることが出来ないという問題点がある. この問題点を解決するために, 本研究では, HTML や CSS の文法をベースとして名前空間および依存関係管理の機能を追加した新たなアプリケーション開発の枠組を提案する. 提案する枠組を用いれば, 既存のコード資産に大きな変更を加えず, アプリケーションの部品化が可能となる.

1　はじめに

近年, GUI アプリケーションを構築する際に, Web 技術を用いることが増えている. こうしたアプリケーションでは, HTML や CSS を用いてユーザインタフェース (UI) を設計し, JavaScript を用いて挙動を定義することで, Web ブラウザ上で GUI アプリケーションとしての機能を実現する. Web 技術を用いることで, Web ブラウザさえあればプラットフォームに依存することなくアプリケーションを動作させることができ, 環境への依存性の低いアプリケーションを実現できる. また, Web 技術はネットワークとの親和性が高いため, アプリケーションにネットワークを利用した機能を容易に追加できるという利点もある. これらのアプリケーションは, クライアントサーバモデルのクライアント側で動作するアプリケーションであるため, クライアントサイド Web アプリケーションと呼ばれる.

GUI アプリケーションの開発においては, UI を構成する要素を分割して開発する部品化と呼ばれる手法が, 開発効率の向上に有用であることが知られている [3]. この手法を応用することで, クライアントサイド Web アプリケーションの開発効率の向上が期待できる. しかし, アプリケーション開発に用いられる HTML や CSS, JavaScript といった言語は, 言語仕様として部品化を行うのに十分な機能を持っていないという問題がある. 例えば, CSS は名前空間の概念を持たないため, ある部品の CSS を異なる部品からも参照できてしまう.

そこで本研究では, クライアントサイド Web アプリケーションの部品化を実現するアプリケーション開発の枠組を提案し実装することを目指す. この枠組では, 既存の開発言語である HTML や CSS, JavaScript に対して必要最小限の変更を加え, アプリケーションの部品化を実現する. この枠組で取り入れられた変更は非常に簡単なものであるため, 既存のコード資産に大きな変更を加えずに, アプリケーションを部品化できる.

本稿の構成は以下のとおりである. 2 章と 3 章では, 簡単なプログラム例を用いつつ, クライアントサイド Web アプリケーションの開発と, その際に生じうる問題について述べる. 次に 4 章で本研究で提案する枠組の概要について, 5 章で枠組を実現するための

Ryutaro Yamamoto, Hideya Iwasaki, 電気通信大学情報理工学研究科, Department of Information and Computer Science, The University of Electro-Communications.

```
/* HTML */
<html>
<body onLoad="init()">
<div>
<span id="count"></span>
<input type="button" onClick="countUp()">
    Count UP
</input>
</div>
</body>
<html>

/* CSS */
#count { color: #555555; }

/* JavaScript */
var cnt;
function init() {
    cnt = 0;
    document.getElementById("count")
        .innerHTML = cnt;
}

function countUp() {
    cnt = cnt + 1;
    document.getElementById("count")
        .innerHTML = cnt;
}
```

図 1　簡単なクライアントサイド Web アプリケーション

ライブラリの設計と実装について説明する．最後に 6 章で関連研究について述べ，7 章で今後の課題と本稿のまとめを述べる．

2　クライアントサイド Web アプリケーション

クライアントサイド Web アプリケーションは，HTML と CSS, JavaScript によって記述される．

図 1 に簡単なクライアントサイド Web アプリケーションの例を示す．このアプリケーションは初期状態では，`count` という ID が付与された DOM のテキストが 0 となっている．そして，アプリケーションの利用者がボタンをクリックするたびに，そのテキストの内容が 1 ずつ増加する．

HTML では，タグを用いてアプリケーションの UI の論理的構造を定義する．例では，`span` タグを用いてカウントされた数の表示領域，`input` タグを用いてカウントアップを行うボタンを定義している．タグには `id` および `class` 属性を付与でき，それを用いて CSS や JavaScript からタグに対応する DOM を指定できる．

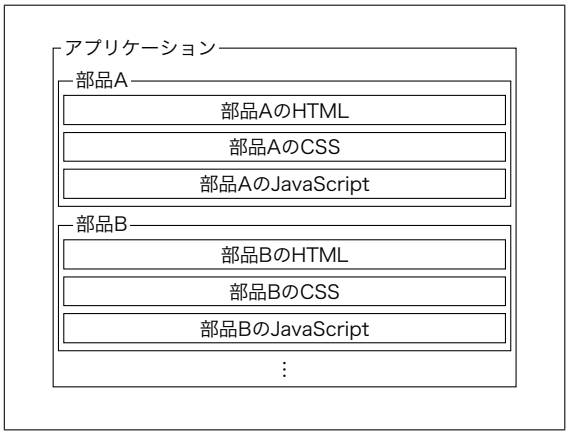

図 2　クライアントサイド Web アプリケーションの部品化

CSS では，UI の見た目を定義する．例では，`count` という ID のついたタグに対応する DOM について，文字色を`#555555` に指定している．

JavaScript では，アプリケーションの挙動を定義する．JavaScript コードは，HTML タグで指定されたイベントが発火した時に呼び出される．例にある `input` タグには `onClick="countUp()"`という属性が付与されているので，この `input` タグで定義されたボタンがクリックされた際に，`countUp` 関数が呼び出される．JavaScript コード中では `document` オブジェクトを介して DOM の要素を操作できる．`init` 関数及び `countUp` 関数では，これを用いて変数 `cnt` の値を DOM に反映している．

3　部品化とその際に起きる問題点

3.1　GUI アプリケーションにおける部品化

GUI アプリケーションにおける部品化とは，アプリケーションの UI を構成する要素を分割して開発することである．部品化によって，アプリケーションの開発において次のような利点が得られる．

- 一度開発した部品を再利用することで，開発期間を短縮し，バグを減らすことが期待できる．
- 部品を複数の人間で分担して開発することで，開発期間を短縮できる．

クライアントサイド Web アプリケーションにおいて部品化を行うと，図 2 のような構造となる．各部品はそれぞれ HTML, CSS, JavaScript を独立して持っ

```
1 /* 部品 A の CSS a.css */
2 #wrapper {
3         background: a.jpg;
4 }
5
6 /* 部品 B の CSS b.css */
7 #wrapper {
8         background: b.jpg;
9 }
```

図 3　名前の衝突

ている.

部品化において実装される部品には, 予め定められたインタフェースによってのみ他の部品とやりとりできるという性質が求められる [2]. これはオブジェクト指向プログラミングにおいてオブジェクトに求められる性質と類似している. この性質が仮に損なわれれば, 部品の利用者は部品の内部実装が変更される度に部品を利用しているコードを書き換えなければならない. 一方, この性質が保たれていれば, インタフェースが変更されない限り, 部品の利用者は部品を利用しているコードを書き換えることなく, 新しい部品を利用できる.

3.2　CSS の問題

クライアントサイド Web アプリケーション開発においては, 3.1 節で述べた性質を保つことが難しい. この難しさは, クライアントサイド Web アプリケーション開発に用いられる言語の仕様に起因している. 本稿では CSS の言語仕様に起因する問題を 2 つ説明する.

3.2.1　名前空間の問題

部品 A と部品 B がそれぞれ図 3 に示すような CSS を持っているとする. この時, 開発者が部品 A と部品 B を同時に利用するアプリケーションを開発することを考える. すると, 部品 A と部品 B 両方が部品 A の CSS と部品 B の CSS を参照できる状態となる. 部品 A の CSS と部品 B の CSS は, 同じセレクタ `#wrapper` を持つので, 部品 A に対して部品 B の CSS が適用されたり, 部品 B に対して部品 A の CSS が適用されたりする可能性が生まれる.

この問題は, CSS が名前空間の概念を持っておらず, 全ての読みこまれた CSS がページ全体に適用されるため起きうる.

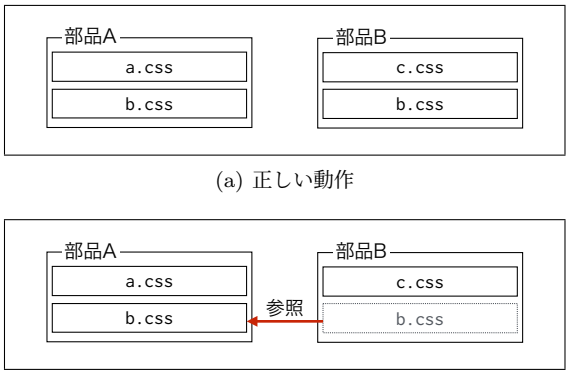

図 4　依存関係の問題

3.2.2　依存関係の問題

図 4(a) に示すような部品が存在する状態を考える. ここで, 部品 A の `b.css` と部品 B の `b.css` は, 偶然同じファイル名であるが内容は異なるものとする. この状態で, 部品 A と部品 B を利用するアプリケーションを構成しても, 問題は発生しない.

しかし仮に図 4(b) のように, `b.css` を部品 B に含め忘れたとする. 部品 B のみを利用するアプリケーションであれば, `b.css` が見つからないというエラーが発生するので, 問題にすぐ気づくことが出来る. しかし, 部品 A と部品 B を両方使うアプリケーションの場合, 部品 B は部品 A の `b.css` を読み込んで動作してしまうため, ファイルが欠如しているという問題にすぐに気づくことが出来ない.

この問題は, 各部品が依存する CSS を管理する仕組が存在しないことに起因する問題である.

4　提案する枠組の設計

4.1　部品の開発

提案する枠組における部品の開発手順を図 5 に示す. 部品の開発者は, HTML および CSS, JavaScript を用いて部品の定義を記述する. 続いて, 記述した HTML および CSS, JavaScript コードを本研究で実装する変換器に入力する. 変換器は記述を解釈し, これらを 1 つの JavaScript ファイルにまとめて出力す

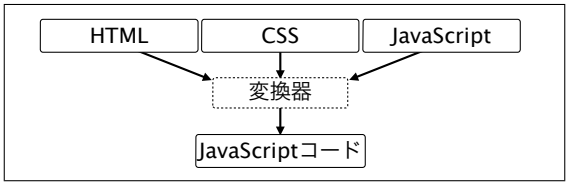

図 5　部品の開発の流れ

る．この出力された JavaScript ファイルを部品として利用する．CSS の内容が JavaScript の中に埋め込まれるため, 各部品が依存する CSS の定義は自明であり, 3.2.2 節で述べた問題が生じることはない．

2 節で紹介した簡単なアプリケーションの例を元に, 提案する枠組で部品を記述した例を図 6 に示す．部品は `a.html`, `a.css`, `a.js` の 3 つのファイルの記述で構成される．

CSS の記述は, これまでの CSS の記述と同等に行える．ただし, `a.css` 内で定義された CSS は, `a.html` に適用範囲が制限される点が, 従来の CSS と異なる点である．これによって, 3.2.1 節で述べた問題が解決される．

クライアントサイド Web アプリケーションの実装においては, JavaScript コードを実行しつつ, 必要に応じて DOM を動的に変更する仕組が必要である．提案する枠組では, この仕組に対する既存のアプローチである Mustache.js [5] を参考に, HTML 上に拡張構文を記述できるようにしてこれを実現している．

HTML からは, `{{}}`という拡張構文を用いることで, JavaScript 中で定義された変数や関数を参照できる．HTML から参照できる JavaScript の変数は, `this.model` オブジェクト中に定義されたものに限定されている．これは JavaScript コードに登場するデータを DOM の構築に必要なものと, そうでない一時的なものに分け, プログラムの見通しを良くする役割を果たしている．

JavaScript には, `_init` という特別な関数を定義できる．これは部品が読み込まれた時に自動的に実行される関数である．

```
1  /* a.html */
2  <html>
3  <body>
4  <div>
5  <span class="count">{{ cnt }}</span>
6  <input type="button" onClick={{ countUp }}>
7      Count UP
8  </input>
9  </div>
10 </body>
11 <html>
12
13 /* a.css */
14 .count {
15         color: #555555;
16 }
17
18 /* a.js */
19 /* 最初に必ず実行される特別な関数 */
20 function _init() {
21     this.model.cnt = 0;
22 }
23
24 function countUp() {
25     this.model.cnt = this.model.cnt + 1;
26 }
```

図 6　提案する枠組による部品の記述

```
1  <html>
2  <body>
3  <div id="component-a">
4  <script src="a.js"> /* 部品を読み込む */
5  </div>
6
7  <div id="component-b">
8  <script src="b.js"> /* 部品を読み込む */
9  </div>
10
11 /* 実行時ライブラリを読み込む */
12 <script src="runtime.js">
13 </body>
14 </html>
```

図 7　部品を利用する例

4.2　部品の利用

上のように記述し生成した部品を利用する例を図 7 に示す．ここで, `a.js` と `b.js` は先述の変換器で出力された JavaScript コードのファイルである．アプリケーションの開発者は, HTML において部品を埋め込みたい箇所に `script` タグを用いて部品の JavaScript コードを読み込むことで, 部品を利用できる．`a.js` と `b.js` はそれぞれ別の部品なので, `a.js` の定義を `b.js` から参照したり, その逆を行ったりすることは

```
module {
    \_css: {
        /* CSS */
    }

    ....
    /* javascript functions */
    ....

    _render: function() {
        return /* HTML */
    }
}
```

図 8　変換器が中間生成する JavaScript コード

できない.

例では, `runtime.js` というファイルも合わせて読み込んでいるが, これは部品の JavaScript コードを解釈して動作させるための実行時ライブラリである. 提案する枠組で開発された部品を利用するには, 実行時ライブラリを合わせて読み込む必要がある.

5　提案する枠組の実装

提案する枠組を実現するために, 部品を記述した HTML や CSS, JavaScript を単一の JavaScript コードに変換する変換器と, その JavaScript コードを読み込んで実行し実際に部品として動作させる実行時ライブラリを実装する.

5.1　変換器

変換器は, まず図 8 のような構成の JavaScript コードを生成する.

コードの全体は, `module` ブロックで囲まれている. `module` は, ECMAScript 6 [6] より導入された言語仕様である. このブロックの内部に変数と関数のみを宣言することができ, それらは原則としてブロックの外側から参照することが出来ない. これを用いることによって, 3.2.1 節で述べた問題点を解決することができる. この `module` の中に, 入力された HTML, CSS, JavaScript が全て含まれている. CSS は, JavaScript のオブジェクトの形に変換して `_css` という名前で保持する. 例えば, 図 9 のような変換が行われる. この変換の実装を簡潔にするために, CSS セレクタを class 宣言に制限している. 部品化を十分に行えば, CSS セレクタで DOM を複雑に走査する必要はなくなるので, CSS セレクタとして class 宣言しか使用できなくとも, アプリケーションの記述には支障はない.

```
/* 変換前 */
abc {
    background: #555;
}

/* 変換後 */
\_css: {
    abc: {
        background: "#555",
    }
}
```

図 9　変換器による CSS の変換

HTML は, `_render` という特別な関数の返り値として, 同様に JavaScript のオブジェクトの形に変換して保持される.

JavaScript コードの生成は, escodegen[†1] というライブラリを用いて行う. このライブラリは, JavaScript の AST から実際の JavaScript コードを生成するライブラリである.

以上の手順で生成されたコードは, ECMAScript 6 に準拠したコードであるため, 古いブラウザでは動作させることが出来ない. そこで最後に, Traceur[†2] というライブラリを用いて, ECMAScript 6 準拠のコードを現行のブラウザで実行できるものに変換し, これを最終的な出力とする.

5.2　実行時ライブラリ

変換器によって生成された部品ファイルは, 3.2 節で述べた問題を解決するために HTML および CSS, JavaScript を全て一つのファイルにまとめているため, そのままではブラウザで解釈して実行することができない. そのため, 部品ファイルを解釈して実行する実行時ライブラリが必要となる. 実行時ライブラリは以下の動作をする.

1. `_init` 関数を実行する.
2. `_render` 関数の返り値を元に, DOM を生成し

†1　https://github.com/estools/escodegen
†2　http://github.com/google/traceur-compiler

展開する.

3. `model` オブジェクトとイベントの監視を開始する.
4. `model` オブジェクトの変更を検知したら, `_render` 関数の返り値を元に新たな DOM を生成し, 古い DOM と置き換える.
5. イベントの発火を検知したら, 対応するイベントハンドラ関数を実行する.

6 関連研究

クライアントサイド Web アプリケーションにおいて, アプリケーションの部品化をしやすくする仕組については, いくらかの例がある. Google Web Toolkit [4] は, Google が開発している Web アプリケーションフレームワークである. このフレームワークは, UI を構成要素ごとに記述しそれを組み合わせることで Web アプリケーション開発を行う本研究と目指す方向が同じであるが, アプリケーションを Java で記述することを前提としており, Java の知識がないデザイナなどが利用するにはハードルが高いという問題がある. 本研究の手法は, HTML や CSS, JavaScript などの既に Web アプリケーション開発で広く利用されている言語をベースとするため, 利用にあたってのハードルが低く抑えられている.

WebComponents [7] は, W3C が策定を進める Web アプリケーションの UI の部品化のための仕様である. 既存の DOM や JavaScript に拡張を加え, 特定の DOM を外部から参照できないようにしたり, 特定の DOM に CSS の適用範囲を制限することが可能になっている. 例えば, DOM オブジェクトに対して外部から参照できない DOM を追加するメソッドが新たに実装されている. この仕組では, DOM と JavaScript そのものに拡張を加えているため, WebComponents を用いて記述されたアプリケーションを古いブラウザで直接動かすことが出来ないという問題がある. それに対して本研究の手法では, DOM や JavaScript に拡張を加えず, 既存のブラウザで利用できる機能のみを用いて部品化を実現するため, 古いブラウザでも問題なくアプリケーションを動作させることができる.

Cascading Tree Sheets (CTS) [1] は, CTS と呼ばれる CSS に似た文法を持つ新たな言語を導入し, 部品化を容易にしている. CSS に着目している点は本研究と類似しているが, 部品化の実現方法が本研究とは異なる.

7 おわりに

本稿では, クライアントサイド Web アプリケーション開発で部品化を行う際の問題点を提起し, その問題点を解決するための新たな枠組の提案を行った. この枠組では, 現行の開発言語に僅かな変更のみを加えることで部品化を容易にしているため, 既存のコード資産に大きな変更を加えることなく, 部品化が可能である.

現状は, 実行時ライブラリの実装を行っており, 手で記述した簡単な変換後の JavaScript コードを解釈して実行できるという段階である.

今後は, 実行時ライブラリが完全な JavaScript コードを解釈して実行できるように拡張するとともに, 変換器の実装を進める. その上で, 簡単なアプリケーションを本枠組で実装し, 従来の記述と比較して, 記述性や, 実行時性能について評価したいと考えている.

参 考 文 献

[1] Benson, E. and Karger, D. R.: Cascading Tree Sheets and Recombinant HTML: Better Encapsulation and Retargeting of Web Content, Proceedings of the 22nd international conference on World Wide Web (WWW '13), 2013, pp. 107–118.

[2] Chaudron, M. and Crnkovic, I.: Component-based software engineering, *Software Engineering: Principles and Practice*, (2008), pp. 605–628.

[3] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and its Implementation*, 1983.

[4] Google: Google Web Toolkit, `http://www.gwtproject.org/`.

[5] Lerner, R. M.: At the Forge: Mustache.js, *Linux Journal*, Vol. 2011, No. 210(2011).

[6] Rauschmayer, D. A.: *Exploring ES6: Upgrade to the next version of JavaScript*, 2015.

[7] W3C: WebComponents, `http://www.w3.org/TR/components-intro/`.