Cop: 部品化を指向した Webアプリケーション開発の枠組

山本 竜太郎[1,a)] 岩崎 英哉[1,b)]

概要：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部品化は, 部品の再利用性を向上させるとともに, 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. そのため, GUI アプリケーション開発の枠組においては, Java の Swing のように部品化を支援する仕組を持つものが多い. 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. 特に CSS に着目すると, 部品化を行っても, 異なる部品間で CSS の定義の名前が衝突し得ることによって, 部品化による利点を十分に得られないことがあるという問題がある. この問題を解決するために, 本稿では, 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 Cop を提案する. Cop では, 記述された部品を JavaScript コード上のデータ構造に置換し, それぞれの部品のカプセル化を行うことで, 問題の解決を図る. また Cop では, 既存の Web アプリケーションの記述とほぼ同じ形で部品の記述が可能であるため, 既存のコード資産に大きな変更を加えず部品化が可能となる.

キーワード：Web アプリケーション, HTML, CSS, JavaScript, 部品化

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. 以下, 本稿ではクライアントサイド Web アプリケーションを単に Web アプリケーションと呼ぶこととする.

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, Web ア

1 電気通信大学大学院情報理工学研究科
a) yryu@ipl.cs.uec.ac.jp
b) iwasaki@cs.uec.ac.jp

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, 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．

そこで本稿では, 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 Cop を提案する．Cop では, 既存の開発言語である HTML, 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．Cop では, HTML, CSS, JavaScript で記述された部品を JavaScript のオブジェクトを用いた表現に変換して取り扱うことで, 部品化を実現している．JavaScript にもともと備わっているデータ構造を利用することによって, アプリケーションの利用者側で特別なプログラムのインストールなどを行うことなく, Cop で記述されたアプリケーションを利用できる．

本稿の構成は以下のとおりである．2 章では簡単な例を用いつつ Web アプリケーションの開発の概観を述べる．3 章では Web アプリケーションの部品化を試みる際に発生しうる問題について述べる．次に 4 章で Cop の概要と設計について, 5 章で Cop を構成するプログラムの実装について説明する．最後に 6 章で関連する研究や成果物について述べ, 7 章で今後の課題と本稿のまとめを述べる．

2. Web アプリケーション

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 Web アプリケーションの例を示す．これは, count という ID が付与された DOM のテキストが 0 となっており, input タグで定義されたボタンをアプリケーションの利用者がクリックするたびに、そのテキストの内容が 1 ずつ増加するアプリケーションである．

Web アプリケーションでは, HTML 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．この例では, 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．これらのタグは, div というタグで囲まれている．div タグで囲むことは, これらのタグがひとまとまりのブロック構造であることを示している．タグには id および class 属性を付与できる．これらは識別子の役割を果たし, これらを用いて CSS や JavaScript から任意のタグに対応する DOM を指定できる．単一 HTML ペー

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

(a) HTML

```
#count { color: gray; }
```

(b) CSS

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

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

(c) JavaScript

図 1: 簡単な Web アプリケーション

ジ中で，`id` 属性は一度しか使用できないが，`class` 属性は複数回使用できる．

CSS は，UI の見た目を定義することに用いられる．この例では，`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 `gray` すなわち灰色に指定している．

JavaScript は，アプリケーションの挙動を定義する．HTML タグの中で定義を与えることで，DOM にイベントが発火した際に指定した JavaScript の関数を呼び出すことが出来る．例えば，図 1(c) にある `input` タグには `onClick="countUp()"` という属性が付与されている．これによって，`input` タグに対応する DOM に対してクリックイベントが発火した際に，JavaScript コード中に定義された `countUp` 関数が呼び出される．また，JavaScript コード中では，`document` というオブジェクトを介して DOM を操作できる．図 1(c) の `init` 関数および `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．部品化はアプリケーションの開発にいくつかの利点をもたらす．本稿ではこの利点のうち，一度開発した部品を再利用することで，開発期間の短縮やバグの減少が期待できる点に着目する．この例として，Java で動作する Swing [2] というライブラリがある．

Swing を用いて開発した簡単な GUI アプリケーションの例を，図 2 に示す．このアプリケーションに使われているボタン，チェックボックス，テキストボックスが部品である．これらの部品は，Java 上ではクラスとして取り扱われる．Swing を用いた Java コードの断片を図 3 に示す．コード中にある `JButton` というクラスが，ボタン部品に対応するクラスである．`new` 演算子を用いてインスタンス化することによって，Java コード上でボタン部品を取り扱うことができる．このコード中では，`button` インスタンスの `setText` というメソッドを呼び出し，ボタンに「This is a button」という文字列を表示するように設定している．

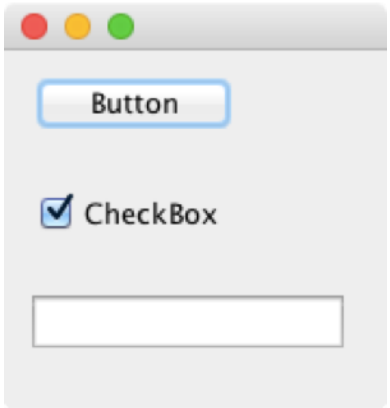

図 2: Swing で書かれた GUI アプリケーション

```
JButton button = new JButton();
button.setText("This is a button");
```

図 3: Swing を用いた Java コード

3.2 部品に求められる性質

部品の再利用性を保つために，部品には満たすべき性質が存在する．それは，予め定められたインタフェースによってのみ部品の外とのやり取りができるという性質 [1] である．

部品 A と部品 B が使用されているアプリケーションを考える．このアプリケーションにおいては，部品 A が部品 B を操作するようなコードが含まれているとする．もし，部品 B に予めインタフェースを定めておかなければ，部品 B が持つ全ての状態や実装を，部品 A などの外部から自由に操作できてしまう．これは，部品 B からすれば部品 A によって自らのどの部分が操作されているのかわからないということを意味している．この状況で，部品 B の実装を変更しようとすると問題が発生する．部品 B は部品 A によってどのように操作されるのかわからないため，部品 B の実装を変更すると部品 A が動作しなくなる可能性があるためである．これでは部品を再利用することができず，部品化の利点が大きく損なわれてしまう．一方で，部品 B の側に予めインタフェースを定めておいた

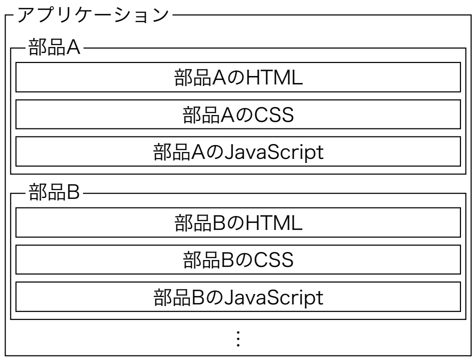

図 4: 部品化された Web アプリケーションの構造

状況を考える．この状況では, 部品 B はそのインタフェースの変更さえしなければ, 実装を自由に変更することが出来, 部品の再利用性を保つことができる．

Java の Swing においても, UI を構成する部品は, 予め対応するクラスにおいて定められたメソッドや公開されたフィールドを介してのみ操作可能なように制限されている．これによって Java の Swing で定義されている部品は, 再利用性が保たれている．

3.3　Web アプリケーション開発における部品化

Web アプリケーションにおいて部品化を行うと, アプリケーションの構造は図 4 のようになる．Web アプリケーションは, HTML, CSS, JavaScript を用いて定義された部品を組み合わせたものである．部品の定義に含まれる HTML, CSS, JavaScript は, それぞれ部品が表現する UI の論理的構造, 見た目, 挙動を定義するものである．

3.4　CSS の問題

Web アプリケーションを部品化しようとした時, 3.2 節で述べた性質を保つことが難しい．この難しさは 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．本稿ではまず CSS の言語仕様に起因する問題に着目して説明する．

3.4.1　名前空間の問題

部品 A と部品 B がそれぞれ図 5 に示すような

```
#wrapper {
    background: gray;
}
```

(a) 部品 A の CSS

```
#wrapper {
    background: red;
}
```

(b) 部品 B の CSS

図 5: 名前の衝突

CSS を持っており, 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 の両方が, 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有しておらず, 全ての読み込まれた CSS がページ全体に適用されるため発生する．

3.4.2　カプセル化の問題

背景色が灰色であるようなボタンである部品が存在したとし, この部品を利用するアプリケーションを開発するとする．さらに, アプリケーションの開発者がこの部品の背景を灰色から赤色に変更することを試みることを考える．すると開発者は, アプリケーションに新たな CSS を読み込ませて, この部品に含まれる CSS を上書きすることになる．ところが, 開発者がこのような新たな CSS を記述するには問題が存在する．なぜなら, 部品の背景色を上書きするような CSS を適切に記述するには, 部品がどのように実装されているのかを知る必要があるためである．この問題は, 単に HTML, CSS, JavaScript を組み合わせて部品を実装するだけでは, 部品のインタフェースにあたる仕組を用意することができないことに起因している．

3.5　JavaScript の問題

Web アプリケーションの実装においては,

JavaScript を用いた DOM の動的な操作が頻繁に用いられる．一般に DOM の操作は次のような手順で行われる．

(1) `document.getElementById` などのメソッドを用いて, 対象とする DOM を選択し, 対応する DOM オブジェクトを得る．
(2) 得られた DOM オブジェクトに対して操作メソッドを呼び出したり, フィールドへの代入を行ったりして, DOM の操作を行う．

この方法は, 部品化の実現にあたって問題を起こす可能性がある．`document.getElementById` などの DOM を選択する関数は, DOM セレクタと呼ばれる文字列を用いて DOM を選択する．例えば, `abc` という `id` 属性を持つ DOM であれば, `#abc` という DOM セレクタを用いることで選択できる．ところが, この DOM セレクタによる選択は HTML ページ全体を探索の対象としている．そのため, ある部品の中で実行された DOM 選択関数が, 異なる部品に含まれる DOM を選択してしまう可能性がある．これは, ある部品に含まれる DOM が別の部品によって全く予想外の変更を受ける可能性を意味している．

4. Cop の概要と設計

Cop でのアプリケーションの開発手順は大きく 2 つの工程に分けられる．

まず部品を開発する工程である．部品は, UI の構成要素を適切に分割した結果得られた HTML, CSS, および JavaScript の断片によって定義される．部品の開発者は, この定義を変換器と呼ばれるプログラムに入力する．変換器はこれらの断片を解釈し, 適切な変換を行った上で, 1 つの JavaScript コードに結合し出力する．

続いてこれらの部品を組み合わせ, 実際に動作するアプリケーションを構築する工程である．この工程では, 各部品同士をどのように連携させるかを定義し, 部品同士が連携しアプリケーションとして動作するようにする．

部品を開発する開発者と, 部品を組み合わせアプリケーションを構築する開発者は, 一般に別である．つまり Cop では, ありあわせの部品を組み

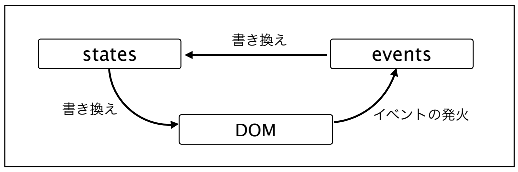

図 6: 部品の動作の流れ

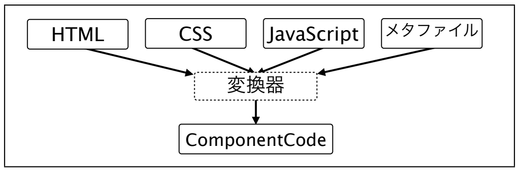

図 7: 部品の開発の流れ

合わせてアプリケーションを開発したり, 部品の開発だけを行ってそれを公開したりするような開発者の存在も想定されている．

4.1 部品の動作の流れ

部品の開発にあたっては, 部品がどのように動作するのかを明確に定めておく必要がある．Cop における部品の動作の流れを図 6 に示す．

部品は DOM と events と states の 3 つの部分から構成されている．DOM は HTML および CSS, events と states は JavaScript を用いて記述する．アプリケーションを利用する人間から直接見えるのが DOM の部分である．DOM はアプリケーションの UI の目に見える部分を構成している要素である．DOM には, 要素のクリックや内容の変更などのイベントが発生することがある．DOM にイベントが発生すると, events が呼び出される．events は, DOM に発生したイベントとそれを処理するハンドラの対応関係を記述したものである．DOM に発生したイベントの種類に基づいて, 対応するハンドラが呼び出され処理が行われる．ハンドラは, 返り値として新たな states を生成する．states は, JavaScript のオブジェクトとして定義されるもので, 部品の状態を持つオブジェクトである．新たな states が生成されると, その states に基づいて DOM の更新が行われる．こうしてアプリケーション内部の処理が UI の目に見える部分に現れる．部品はこのサイクルを繰り返して動作する．

```
<div>
<span class="count" id="count">
  {{count}}
</span>
<input type="button" id="up-button">
  Count UP
</input>
</div>
```

(a) HTML

```
.count { color: gray; }
```

(b) CSS

```
states: {
  'count': 0,
}

events: {
  '#up-button': {
    'click': function() {
      return {
        'count': states.count + 1
      };
    }
  }
}
```

(c) JavaScript

```
{
  'name': 'comp-a',
  'version': '0.0.1'
}
```

(d) メタファイル

図 8: 部品の記述例

JavaScript にネイティブで用意されているイベントハンドリング機構や DOM の構築 API を用いず, このような新たな仕組を用意したのは, 部品がカプセル化された状態を守るためである. JavaScript に用意されているイベントハンドリング機構や DOM の構築 API は, Cop で導入した部品という概念と

```
states: {
  'count': 0,
  'is_even': true,
}

events: {
  '#up-button': {
    'click': function() {
      var nc = states.count + 1;
      return {
        'count': nc,
        'is_even': nc % 2 == 0 ?
          true : false,
      };
    }
  }
}
```

(a) JavaScript

```
{if is_even}
<span>Count is even</span>
{else}
<span>Count is odd</span>
{endif}
```

(b) HTML

図 9: `if` 構文の利用例

は独立に動作するため, これらを利用すると容易にカプセル化された状態を無視したプログラムを書くことができてしまう. Cop では, events と states を介してのみイベントや DOM を取り扱えるようにすることで, これらの操作は原則として部品の中だけに制限され, 部品がカプセル化された状態を保つことを可能としている.

4.2 部品の開発

Cop での部品の開発手順を図 7 に示す. 部品の開発者は, HTML, CSS, JavaScript, およびメタファイルを用いて部品の定義を記述する. Cop で図 1 と同じカウンタの部品を記述した例を図 8 に示す. Cop を使わないもの (図 1) と比較すると,

JavaScript の記述に大きな違いがある．states と events というオブジェクトが定義されているのがその違いである． これは先述の states および events と対応するものである．また, 図 8 の例にはないが, states には_public という特別なオブジェクトを持たせることができる．_public オブジェクトの内容は, 後述する部品の利用の際に外部から書き換えることができる．

あわせて, 部品に関する情報をメタファイルに記述する．メタファイルは JSON 形式で記述されており, 部品の名前やバージョンを記述できる．

Cop においては, より複雑な DOM 操作を支援するため, HTML に特殊な制御構文を記述できる．そのような構文の一つの例として．if 構文を図 9 に示す．この例の場合, is_event の部分が評価され, それが真であるときは"This is inserted if count is even"というテキスト, 偽であるときは"This is inserted if count is odd"というテキストが展開される．cond の部分の評価は, その部分を states オブジェクトに含まれる変数の名前だと解釈して行われる．この例の場合, is_even は count が偶数の場合 true, 奇数の場合 false となるので, count の偶奇によってテキストが変化する．こうした制御構文を用意することで, HTML を記述する際に, 同じ記述を何度も繰り返さなくて良くなり, 見通しの良い実装が可能となる．

続いてこれらの記述した HTML, CSS, JavaScript, およびメタファイルを変換器に入力する． 変換器は記述を解釈し, これらをまとめて 1 つの JavaScript ファイルに出力する． この JavaScript ファイルのファイル名は, デフォルトでは「部品名.js」となる．先の例では, comp-a.js というファイルが出力される．本稿では今後この出力された JavaScript コードを Component Code と呼ぶ．部品を利用してアプリケーションを開発する開発者は, この Component Code を利用する．Component Code には, 元の入力の HTML, CSS, および JavaScript の記述が JavaScript コードの形で含まれている．

```
<html>
<body>
<div id="comp-a"></div>
<script src="comp-a.js"></script>
<script src="cop.js"></script>
<script>
cop.inject('comp-a', '#comp-a');
</script>
</body>
</html>
```

図 10: Component Code の利用

4.3 部品の利用

4.3.1 アプリケーションでの部品の利用

アプリケーションを開発するには, 前節の手順で部品の開発者が開発した Component Code を組み合わせる工程が必要となる．Component Code を利用する例を図 10 に示す．このコードにおいて読み込まれている comp-a.js が Component Code である．Component Code は JavaScript コードであるため, script タグを用いて読み込む．合わせて読み込まれている cop.js は, Component Code を実際に動作させるために必要な実行時ライブラリであり, Cop が提供しているものである．Component Code を利用するには, 実行時ライブラリの読み込みと同時に cop.inject 関数を呼び出す必要がある．cop.inject 関数には, 第 1 引数に部品の名前, 第 2 引数にその部品を利用する場所を示す DOM セレクタを渡す．部品の名前は, 部品の開発者が部品を開発する際にメタファイルに定義したものである．cop.inject 関数を実行すると, 第 1 引数に与えられた名前を持つ部品が第 2 引数によって示された DOM の直下に展開される．第 2 引数に渡される DOM セレクタは, 特定の DOM を一意に指すものでなければならない．もし複数の要素とマッチする DOM セレクタが指定された場合, 現状の実装では正しいアプリケーションの動作は保証されない．

cop.inject 関数を呼び出す際に, 第 3 引数にオブジェクトを渡すこともできる． この場合, cop.inject 関数は対象となる部品の states オブジェクトの中にある_public オブジェクトに対

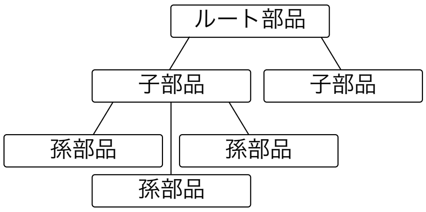

図 11: 部品がなす木構造

```
{
  'name': 'comp-a',
  'verison': '0.0.1',
  'dependencies': [
    './comp-b.js',
    './comp-c.js'
  ]
}
```

図 12: 依存関係を記述したメタファイル

して, 引数に与えられたオブジェクトの内容を追加する. この仕組によって, 部品を利用する際に部品の利用者側で states の初期値を変更することが可能となる. ただし, 変更できる states の内容は, _public オブジェクトに含まれる内容のみと制限されているので, 部品の開発者側が意図しない変更が加えられる恐れはない.

4.3.2 部品の依存関係

, 一般に複数の部品を組み合わせて開発することとなる. Cop では, アプリケーションの開発において組み合わせた部品同士の関係は, 部品同士の関係は図 11 に示すような木構造で表現される. Cop ではルート部品という概念が存在する. これは全ての部品の祖先にあたる部品であり, アプリケーション全体に相当する部品である. Cop でのアプリケーションの開発は, このルート部品を実装することと等価である. それぞれの部品は, 定義に明示的に記述を行うことで, 任意個の子部品を持てる.

ある部品がどのような部品に依存しているかは, メタファイルに記述する. メタファイルの例を図 12 に示す. dependencies が依存先の部品を定義している部分である. dependencies には, 依存している部品の Component Code へのパスを記述する. この例は, comp-b.js と comp-c.js という 2 つの Component Code に依存していることを表している. なお実装の制約上, 依存関係を記述した Component Code を利用する際には, 依存先の部品の Component Code とのパス上での相対的な位置関係を崩してはならない.

```
events: {
  '#comp-a-button': {
    'click': function (event) {
      event.emitToChildren('some-event',
        {'msg': 'some msg'});
      return {};
    }
  }
}
```

(a) 親部品

```
events: {
  '_parent': {
    'some-event': function (event) {
      return {
        /* 'some msg' */
        'msg': event.payload.msg
      }
    }
  }
}
```

(b) 子部品

図 13: 部品の連携

親部品上で子部品を利用するには, 親部品の定義上で cop.inject 関数を呼び出せば良い.

4.3.3 部品の連携

複数の部品を組み合わせた開発においては, 複数部品間で連携を行う仕組も必要となる. 3.2 節で述べた性質を満たすために, 部品間の連携を行う上である部品が別の部品を直接操作することは望ましくない. そこで Cop では, イベントを利用した部品間の連携を提供している. Cop においてはユーザが任意の名前をつけたイベントを発行する

ことができる．イベントを利用して部品間の連携を実現している例を図 13 に示す．

この例では, 親部品と子部品という 2 つの部品が存在している．この 2 つの部品は前節で述べた方法を用いて親子関係が定義されている．そして, 親部品から子部品に対してイベントを発火させ, 子部品に対して情報を渡している．親部品のコードに含まれる `emitToChildren` メソッドの呼び出しが, イベントの発火に当たる部分である．`emitToChildren` 関数を呼び出すと, 第 1 引数に指定されたイベント名で, その部品の全ての子部品に対してのみ有効なイベントを発火させられる．この例では, 子部品に対して `some-event` という名前のイベントを発火している．子部品では, このイベントを `_parent` という特殊な DOM で発火したイベントとみなして処理する．

連携の実装を単純にするため, `emitToChildren` 関数に自身が持つ子部品を区別する仕組みは用意されていない．この設計は, Cop における部品の連携の仕組みを単純にすることに貢献しているものの, 特定の子部品にのみイベントを送信出来ないため, 同じイベントハンドラを持つ複数の子部品から特定の部品のイベントハンドラだけを選んで呼び出すといった複雑な処理は出来ない．

同様に, 子部品から親部品にイベントを発火させる `emitToParent` 関数も用意されている．この関数によって発火したイベントは, `_child` という特殊な DOM で発火したイベントとみなして処理する．

Cop で部品間の連携を行う方法はこの `emitToChildren` 関数と `emitToParent` 関数によるイベント発火しか存在しない．つまり, イベントが Cop における部品のインタフェースの役割を果たしている．この仕組によって, 部品の内部実装を知らなくても, 部品がどのようなイベントを受け取れるのかさえ知っていれば, 部品を制御することができる．これにより, 部品のカプセル化を実現している．

5. Cop の実装

Cop n 主要な構成要素は, 部品を記述した HTML や CSS, JavaScript を Component Code に変換する変換器と, その Component Code 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である．本節はこれらの実装について述べる．

```
cop.comp('comp-a', {
  events: {

  },
  states: {

  },
  _css: {
    /* CSS */
  },
  _dom: {
    /* DOM */
  }
});
```

図 14: 変換器が中間生成する JavaScript コード

```
.count { color: gray; }
```

(a) 変換前

```
_css: {
  'count' {
    'color': 'gray'
  }
}
```

(b) 変換後

図 15: 変換器による CSS の変換

5.1 変換器

変換器は, まず図 14 のような構成の JavaScript コードを生成する．このコードは, `cop.comp` という関数の呼び出しになっている．この関数は, 第 2 引数として受け取ったオブジェクトを部品の定義とみなし, 第 1 引数に与えられた名前で実行時ライブラリ上で取り扱える形とする関数である．このオブジェクトには, `events`, `states`, `_css`, `_dom` という 4 つのオブジェクトが含まれる．前者 2 つのオブジェクトは, 変換器に入力された JavaScript

コードに定義されていた events と states そのものである．_css は, 変換器に入力された CSS を JavaScript のオブジェクトの形に変換したものである．例えば, 図 15 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 セレクタをクラス宣言のみに制限している．このような制限を課したとしても, 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 セレクタとしてクラス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．_dom は, 変換器に入力された HTML を JavaScript のオブジェクトの形に変換したものである．

Component Code の生成は, 文字列操作として実現されている．CSS や HTML を構文解析し, JavaScript オブジェクトに変換する操作については, 既存の CSS および HTML パーサを用いている．

5.2 実行時ライブラリ

Component Code は, 3.4 節で述べた問題を解決するために HTML および CSS, JavaScript コード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, Component Code 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おおまかに以下の動作をする．

(1) Component Code で cop.comp 関数が呼び出される．実行時ライブラリは, 関数の引数を元に部品を実行時ライブラリ上で取り扱える形とする．
(2) _dom と _css をもとに, DOM を生成し展開する．
(3) DOM へのイベント発火を検知したら, events に定義されたハンドラを実行し, 新しい states を得る．
(4) 新しい states を元に新たな DOM を生成し, 古い DOM と置換する．

より細かな動作について, 以下で説明する．

5.2.1 部品の初期化

comp.inject 関数を呼び出すと, 対象となる部品の Component Code に含まれる cop.comp 関数が呼び出される．cop.comp 関数は, 引数に与えられた部品の定義を表すオブジェクトを元に, 部品の定義を実行時ライブラリ内に登録する．また, その部品の依存関係の情報も実行時ライブラリ内に木構造の形で登録する．この木構造は, emitToChildren 関数や emitToParent 関数によってイベントが発火した際に, イベントを送信する部品を決定する際に用いられる．

部品の情報の登録が終わると, 実行時ライブラリは部品の定義に基づいて DOM の構築を行う．構築された DOM は comp.inject 関数で指定された場所に展開される．

5.2.2 イベントの処理と DOM の再構築

DOM へのイベント発火を検知すると, 実行時ライブラリに含まれている eventRunner という関数が呼び出される．eventRunner は, 対象となる部品の events を参照し, 発火したイベントを処理するハンドラを探索する．目的のハンドラが見つかればそれを呼び出し, その返り値を元に states を更新する．

実行時ライブラリには, eventRunner とは別に statesObserver というオブジェクトが存在している．statesObserver は, states の変化を監視し, states の変化が起きた際に新たな DOM を生成するための関数を呼びだすオブジェクトである．eventRunner によって states が書き換えられると, statesObserver がその変化を検知し states の変更を反映するために新たに構築し直さなければならない DOM を特定した上で, その DOM を構築する関数を呼び出す．

6. 関連研究

Web アプリケーション開発において, 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ら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性が同じであるが, 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, 既に Web アプリケーション開発に広く用いられている言語をベースとしているため, 利用にあたってのハードルが低く抑えられている. さらに, 既存のコード資産を活用しやすいという利点もある.

Han ら [5] は, Pagelet という Web ページの断片を表す概念を導入した Web アプリケーションの開発手法を提案している. この手法は, XML をベースとしており, XSLT[6] と呼ばれる言語を用いて XML で記述されたデータを HTML に動的に変換することで動的なアプリケーションの挙動を実現している. Pagelet という概念は本稿で述べている部品の概念と類似しているが, 必ずしも Pagelet はインタフェースによる外部とのやり取りの制限がされているとは限らない点, XSLT という新たな言語を導入している点が異なっている.

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組である. 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 の適用範囲を制限したりすることができる. 例えば, JavaScript 上で DOM を表現する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 で記述されたアプリケーションを古いブラウザでは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, 古いブラウザでもアプリケーションを動作させることが出来る.

7. おわりに

本稿では, 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, 現行の開発言語に少しの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.

Cop は開発途上の枠組であり, 今後改善すべき余地がある. その一つに, 複数の部品にまたがるような定義の共通化が挙げられる. 例えば, 複数の部品を用いているアプリケーション全体で, 特定の条件を満たした要素に対して何らかの処理を行いたい時, 現状では`_public`や部品の連携機能を用いて, 対象となる部品全体に処理を伝搬しなければならない. この問題点を改善するには, 複数の部品で共通の実装を共有できるモジュールのような機能を用意することが考えられる.

またアプリケーションの実行性能についても検討が必要である. Cop は非常に小さな枠組であるため, Cop 自体がアプリケーションの実行性能に及ぼす影響は小さいと予想されるものの, Cop を使用しない場合と使用した場合でのオーバヘッドの比較, また他の Web アプリケーションフレームワークとのオーバヘッドとの比較は今後の課題である.

参考文献

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

[2] Eckstein, R., Loy, M. and Wood, D.: *Java Swing*, O'Reilly Media (1998).

[3] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Longman Publishing Co., Inc. (1983).

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

[5] Han, H. and Liu, B.: Problems, Solutions and New Opportunities: Using Pagelet-based Templates in Development of Flexible and Extensible Web Applications, *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '10, pp. 679–682 (2010).

[6] Tidwell, D.: *XSLT*, O'Reilly Media (2008).

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