第57回 プログラミング･シンポジウム 2016.1.8-10

Web Components開発における ドキュメント同時生成手法の提案

海老澤 雄太[1] 丸山 一貴[2] 寺田 実[3]

概要：現在 W3C で仕様策定中の Web Components は，Web ページの要素をコンポーネント化するための技術であり，利用する側のマークアップの簡潔化や要素の再利用を促進することを可能にする．しかしながら，ドキュメント生成の観点からみると Web Components は，ソースコード中のコメントからドキュメントを生成する従来の手法をそのまま適用することが難しい．そのため，現状はドキュメントを個別に作成するなどの別の手法を取らなければならない．そこで本研究では，Web Components の作成に関して，1 つの記述によって 1 つの Web Components のソースコードとドキュメントを同時に作成するための手法を提案する．提案手法では，構造化された文書を記述するのに用いるテキスト記法である Markdown 記法を利用し，ドキュメントの中にコードを埋め込むことでそれを実現する．

キーワード：Web Components, ドキュメント, コメント, Markdown

1. はじめに

HTML5 が登場してから，Web ページ上で可能となった操作・表現が増えるに従い，Web における開発手法も大きく変遷を遂げてきた．特に，通常のソフトウェア開発で活用され築き上げられてきた手法が，急速に Web における開発にも用いられるようになった．例えば，Angular.js[*1] や Vue.js[*2] などのフレームワークでは，MVVC モデルでの開発やデータバインディングの利用を可能としている．さらに，先日仕様が公開された ECMAScript2015[*3] をはじめ，いくつかの JavaScript 処理系でモジュール化とそれを使用するための仕組みが含まれるようになった．

これらの例として，Web Components[*4] を挙げることができる．これは，Web ページ中のボタンや検索フォームといった構成要素をコンポーネント化し，再利用するための技術である．現在は仕様策定の段階にあるものの，従来使われてきた方法の問題点を解決し，コンポーネントをより使いやすくするものであるため，今後普及していくと考えられる技術である．Web Components の技術を用いて作成されたコンポーネントは現在，コンポーネント化する部分のマークアップ構造（HTML）と，それに対する CSS によるスタイル定義と JavaScript による動作定義を内包した HTML ファイルとして提供される．

コンポーネントの作成者の観点から見ると，コ

1 電気通信大学大学院 情報理工学研究科 情報・通信工学専攻
2 明星大学 情報学部 情報学科
3 電気通信大学 情報理工学部 情報・通信工学科

*1 https://angularjs.org/
*2 http://jp.vuejs.org/
*3 http://www.ecma-international.org/publications/standards/Ecma-262.htm
*4 http://webcomponents.org/

ンポーネントを作成し公開するにあたり，それを使用するための使用方法や API を記述したドキュメント（以下，特に断りがなければ「ドキュメント」とは，このような解説文書としての意味で用いる）を作成する必要がある．Web Components において必要だと考えられるドキュメントの内容は次のようなものだと考えられる．

- 制作者に関する情報
- ライセンス情報
- 依存関係
- HTML での使用方法
- JavaScript から操作可能な API に関する情報

従来，ドキュメントの生成を効率的に行うための手法として，いくつかの方法が用いられてきた．1 つは Knuth が提唱する文芸的プログラミング [4] であり，あるいは Java における JavaDoc[*5] を代表とするソースコード中のある記法に則ったコメント（ドキュメントコメント）からドキュメントを生成するものである．

ドキュメントコメントを用いた手法は有用な手法ではあるが，Web Components においては現状，あまり有効ではないと考えられる．Web Components 自体は 1 つの HTML であるが，この HTML 内には再利用する部分のマークアップ構造とそれに対するデザインを規定する CSS，動作を記述した JavaScript が含まれている．つまり，文法の異なる複数の言語が同時に存在するソースコードで記述されている．ドキュメントコメントを用いた手法は単一の言語で構成されるソースコードに対して適用されてきたため，既存のやり方ではうまく対処することができないと考えられる．これに関する対応策として，HTML や JavaScript などのパーサを組み合わせて，単一の言語として分割した結果からドキュメントに関する記述を抽出する方法も考えられる．しかし，それぞれの言語部ごとにドキュメントコメント書き方が異なってしまう問題は解決できない．また，ライセンス情報など特定の言語のドキュメントコメント内に記述する必要のない内容については，ドキュメントコメントをつけようとする制作者やコード自体を読もうとするユーザを不必要に混乱させてしまう可能性がある．

また，文芸的プログラミングは有効的な手法となりうるが，そのためのシステムがないために採用されていない．一部，ソースコードのコメント中に詳細なドキュメントを記述する例もあるが，この場合は先に述べたように複数の文法の異なる言語で構成されているために適用が難しい．Knuth が提唱する WEB システムが生成するドキュメントについても，ソースファイルの記述それ自体を整形したものとなるため，ドキュメントコメントを用いたものよりも利用者にとって不都合なものになってしまう可能性がある．そのため，ドキュメントの作成については，個別に直接作成するなどの手段をとる必要がある．

そこで，本研究では Web Components に対して従来手法とは別に，1 つの記述によって 1 つの Web Components のソースコードとドキュメントを同時に作成するための手法を提案する．また，実際にこの手法を用いた開発を行うためのツールを作成し，この手法の有用性を検証していく．

2. Web Components

現在，Web Components は

- Shadow DOM[*6]
- Custom Elements[*7]
- HTML Imports[*8]

の 3 つの仕様から構成されている，Web ページ中の構成要素をコンポーネント化し再利用するための技術である．これらの仕様はそれぞれ，マークアップ構造・スタイルのカプセル化，独自タグの登録と利用，外部リソースの読み込みに関する仕様であり，Web ブラウザから JavaScript API などを通じて提供される機能である．

図 1 は，Web Components の実体の構造を示す抽象的なコードである．30 行目から 32 行目において Shadow DOM の機能を用いてカプセル化され

[*5] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

[*6] http://www.w3.org/TR/shadow-dom/

[*7] http://www.w3.org/TR/custom-elements/

[*8] http://www.w3.org/TR/html-imports/

た環境の中にマークアップ（2 行目から 12 行目）の展開をする処理を記述し，38 行目と 39 行目で Custom Elements の機能を用いて独自タグの登録を行っている．また，Web Components は JavaScript から操作するために様々なフィールド値を定義することができる．以下ではフィールド値の型が関数であるものをメソッド，それ以外をプロパティと呼ぶ．

作成した Web Components は HTML および JavaScript の両方からインスタンスを生成することができる．HTML から使用する場合は，図 2 のように通常の HTML タグと同じように使用することができる．JavaScript から呼び出す場合も，標準の API である document.createElement を使って生成できる（図 3，2 行目）．また，document.registerElement の戻り値は作成した Web Components のコンストラクタとなるため，図 1 の 38 行目の Tag を用いてインスタンスを生成することもできる（図 3，4 行目）．

Web Components が現れる前から，構成要素のコンポーネント化とその使用は行われていた．従来のコンポーネント化とその使用は JavaScript ライブラリを用いて行っていた．図 4 は，従来の方法と Web Components でのコンポーネントの使用方法を比較したものである．従来の JavaScript ライブラリによるコンポーネントは使用者側がコンポーネントの展開先を用意した上で，その展開先にコンポーネントのマークアップ構造を挿入するためのライブラリ独自の機能を呼び出すコードも記述する必要があり，使用方法が煩雑であった（下段左側）．また，同一の DOM（Document Object Model）ツリー上にマークアップが展開されるため，出力する内容によっては他の要素とのデザイン指定と競合を起こす問題をはらんでいた．それに対し，Web Components ではコンポーネントを独自タグを用いて使用するため，利用者側は容易に使用することができる（下段右側）．加えて，出力するマークアップやデザインはカプセル化された状態で DOM ツリーに展開されるため，他のデザイン指定との競合を起こすことなく使用することができる．

```
1  <!-- markup definition -->
2  <template>
3    <!-- styling -->
4    <style>
5      #hoge { background: yellow; }
6      p {font-weight: bold; font-size: 16px;}
7    </style>
8    <!-- markup -->
9    <div id='hoge'>
10     <p>Hello, world.</p>
11   </div>
12 </template>
13 <script>
14   //define components
15   (function() {
16     //define methods or properties
17    var proto =
18       Object.create(HTMLElement.prototype);
19     proto.methodName = function(...){...};
20     proto.propName = ...;
21
22     //Shadow DOM create
23     var doc =
24       document.currentScript.ownerDocment;
25     proto.createCallback = function() {
26       var tmpl =
27         doc.querySelector("template");
28       var tmplCode =
29         doc.importNode(tmpl.content, true);
30       var sroot =
31         this.createShadowRoot();
32       sroot.appendChild(tmplCode);
33       ...
34     }
35     ...
36
37     //register element
38     var Tag = document.registerElement(
39       'tag-name', {prototype: proto});
40   })();
41 </script>
```

図 1 Web Components のコードの概形

```
<tag-name></tag-name>
```

図 2 Web Components の HTML からの利用

仕様策定の段階にある現在，Google Chrome[*9] や最新のブラウザであれば Polymer[*10] などのライブラリを利用することで，Web Components を使用することが可能である．

[*9] https://www.google.co.jp/chrome/
[*10] https://www.polymer-project.org/1.0/


```html
<html>
  <head>
    <link rel="stylesheet"
          href="audio-player.css" />
    <script src="audio-player.js" />
  </head>
  <body>
    ...
    <div class="container"></div>
    ...
    <script>
      var elm = ...;
      var option = ...;
      ...
      addAudioPlayer(elm, option);
      ...
    </script>
  </body>
</html>
```

JavaScript Library

```html
<html>
  <head>
    <link rel="import"
          href="audio-player.html" />

  </head>
  <body>
    ...
    <audio-player option="..."></audio-player>
    ...

  </body>
</html>
```

Web Components

図 4 コンポーネントの使用方法の比較
例えば上段のコンポーネントを利用する場合，JavaScript ライブラリでは下段左側の，Web Components では下段右側のような記述をすることで使用が可能になる．

```
1 // Pattern A
2 var instance1 = document.createElement("
    tag-name");
3 // Pattern B
4 var instance2 = new Tag();
5
6 // insert DOM tree
7 var body = document.body;
8 body.appendChild(instance1);
9 body.appendChild(instance2);
```

図 3 Web Components の JavaScript からの利用

3. 設計

3.1 方針

ドキュメントとソースコードを同時に生成するためには，これら両方に関する記述がなされている必要があることは自明である．しかしながら，既存のソースコード中に特定の書式でコメントを加えていくドキュメントコメントではうまく対処できないことは先に述べた．文芸的プログラミングも，有効的な手法となりうるが，そのまま適用することは難しい．

そこで，本研究では緩く制限を加えたドキュメントの記述の中にプログラムコードを挿入していくことで，ドキュメントとソースコードを一度に作成する手法を提案する．文芸的プログラミングのように対象とする言語のコメント記法に左右されることなく記述可能で，ドキュメントコメントの方式のように正規化され必要な情報のみを取り出したドキュメントを生成することができる．

この手法を用いたコードとドキュメントの生成の概念図を図 5 に示す．入力は，ドキュメント記述のための記法を用いたテキスト形式のドキュメント（図中①）である．この入力ドキュメントをツールを用いて解析してドキュメント部分とコード部分を抽出・処理し，Web Components のプログラムコード（図中②）とその API ドキュメント（図中③）を出力する．プログラムコードは，ドキュメントの記述と抽出したコードを組み合わせることで作成する．API ドキュメントは，ドキュメントの記述のうちメソッドやプロパティなどの必要な情報のみを抜き出して出力する．

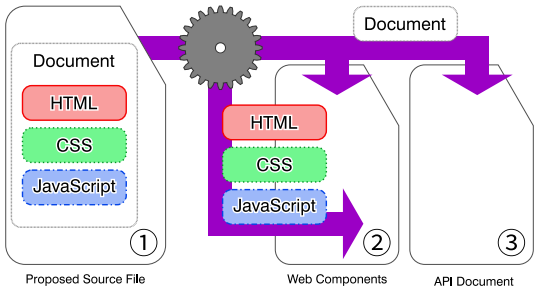

図 5 ドキュメントとコードの生成方針

また，近年の Web 開発ではスタイルチェック・テスト・最適化など，複数のツールも用いて処理された結果を出力として扱うことが多い．そこで，テストコードの記述や，ツールの使用に関する記述についても行えるような方法を考える．

3.2 記法の選択

ドキュメントを記述する記法は XML をはじめ，reStructuredText[*11] など様々なものが存在する．今回，提案手法を実行するためのドキュメント記述のための記法として，Markdown を採用した．

Markdown[*12] は Gruber が提唱した，HTML への変換を主とする簡易マークアップ言語である（図 6）．テキストメールの装飾に似たシンプルな記述によって文書構造を定義するもので，ドキュメントやウェブログなどの様々な文書の記述に広く用いられている．プログラムコードは 14・15 行目のようにインデントで字下げをすることで記述することができる．図 6 を HTML へ変換すると図 7 のような出力が得られる．また，現在多くの独自・拡張記法や解釈の差があるため，CommonMark[*13] として標準化の最中にある．

提案手法を実現するために用いる記法の条件として，(1) 記述がしやすい (2) テキストの状態で読みやすい (3) ドキュメントを記述するのに最低限の文書構造を記述可能 (4) 様々なプログラムコードを埋め込むことが可能，を満たすことを要求した．これらの条件を満たす記法の中で，すでに広く知られており，多くの人が活用している記法であると考えられる Markdown 記法を採用することにした．

(1) (2) (4) の条件については，図 6 を見れば明らかである．キーボードから直接入力できる記号を用いて文書構造を定義するため，XML や LaTeX と比較して大いに記述が簡単であると言える．見た目に関しても，テキストメールなどで使われる装飾や意味合いとほぼ同義であるため，テキスト状態でも文書構造を理解しやすい記法であると考えられる．(3) の条件について，Markdown の活用例として実際にドキュメントの記述に用いられているため，この条件も満たされていると考えた．図 6 においても，「見出し」「段落」「強調」「引用」「リスト」「リンク」「コード」の文書構造を記述しており，十分な文書構造を記述できると考えられる．

Markdown 記法の活用例を見てみると，GitHub[*14] で公開されているリポジトリの ReadMe やドキュメントはしばしば Markdown で記述されており，ドキュメント記法として開発者に馴染み深いものであると考えられる．また，Qiita[*15] をはじめとするナレッジ共有系 Web サービスではプログラムコードを表示することを目的とした文書の記述に Markdown が使われている．このことからも本研究で提案するドキュメントとプログラムコードを同時に記述するための用途に合致していると考えられる．

4. 書式

以下では，提案手法における Markdown での，Web Component のドキュメントとコードの記述方法について述べていく．なお，Markdown には様々な拡張があり，システムによって使用可能な記述が異なる場合がしばしばある．本研究では，一部の広く使われている拡張記法を除き，Gruber が提唱したオリジナルの記法のみで記述できるよう考慮した．

[*11] http://docutils.sourceforge.net/rst.html
[*12] https://daringfireball.net/projects/markdown/
[*13] http://commonmark.org/
[*14] https://github.com/
[*15] https://qiita.com/

```
# level1 header

## level2 header

Plain text is paragraph.
You can use *italic* or __bold__ style.

> quatation
> like e-mail style

[link](http://www.example.com)

- list
    - can
        - be nested

    // code example in JavaScript
    console.log("Hello, world.");
```

図 6　Markdown 記法の例

```
<h1>level1 header</h1>

<h2>level2 header</h2>

<p> Plain text is paragraph.
You can use <em>italic</em> or <strong>bold</strong> styles.

<p><a href="www.example.com">link</a><p>

<ul><li>list
  <ul><li>can
    <ul><li>be nested</li></ul></li></ul></li></ul>

<pre><code>// code example in JavaScript
console.log("Hello, world.");</code></pre>
```

図 7　図 6 の HTML への変換結果

4.1　文書の概形

ユーザが記述する文書の概形を図 8 に示す．

Markdown で記述する文書構造は，レベル 1 の見出しではじめるものとする．このような文書のはじめに現れるレベル 1 の見出しは，その文書のタイトルとして扱われることが多いが，ここでは作成する Web Components のタグ名として扱う．そのため，ハイフン（-）を含む，数字から始まらないなどの Custom Element の仕様に起因する制限を設ける．また，1 つの文書で 1 つの Web Components を定義するものとする．

```
# component-name

## description
explanation of this component.

## properties
definitions of properties.
...

## methods
definitions of properties.
...

## others
...
```

図 8　提案手法による記述の概形

メソッドや使用方法など，Web Components のコード定義およびドキュメントの記述はレベル 2 の見出しで始まるセクションで記述していく．ここで，セクションとはあるレベルの見出しから，それ以上のレベルの見出しが現れるまでの範囲のことを指す．例えば，レベル 3 の見出しで始まるセクションは，次にレベル 1・レベル 2・レベル 3 の見出しが現れるまでの間とする．

以下では，具体的な記述の方法について記していく．提案手法の記述により生成されるコードについて理解しやすくするため，一般的な記述方法と同時に，次の図 9 に示す Web Components のコードに対応する記述を合わせて記載する．なお，図 9 のコードは Polymer ライブラリを使用しているため，図 1 とは概形が多少異なっている．また，このコードは説明目的のため，本来使用する上で必要となるであろう機能や条件分岐を省いている．

4.2　マークアップ・スタイルの記述

コンポーネント化するマークアップとそれに対するスタイルに対する記述は，図 10 のようにそれぞれ，「structure」および「style」というレベル 2 のセクション内で行う．

セクション内に記述されたプログラムコードを抽出して，コンポーネント化するマークアップ・スタイル部分のコードの生成に用いる．コードの記述方法は，標準的なインデントによるものと拡

```
<link rel="import" href="bower_components/
  polymer/polymer.html" />
<dom-module id="my-counter">
  <template>
    <!-- style -->
    <style>
      #container {
        display: inline-block;
        font-size: 16px;
        height: 20px;
        padding: 0 4px;
        background-color: #4CCFFF;
        border: 1px solid #0084B4;
      }
      #num {
        color: #fff;
      }
    </style>
    <!-- DOM structure -->
    <div id="container">
      <span id="num">0</span>
    </div>
  </template>
  <script>
    Polymer({
      is: "my-counter",
      // definition of properties
      properties: {
        count: {
          type: Number,
          value: 0
        },
      },
      // definition of methods
      reset: function() {
        this.count = 0;
        this.$.num.textContent = this.count;
      },
      increase: function(quantity) {
        this.count += quantity || 1;
        this.$.num.textContent = this.count;
        return this.count;
      },
      attached: function() {
        this.$.num.textContent = this.count;
      }
    });
  </script>
</dom-module>
```

図 9 Web Components の具体例（Polymer ライブラリ使用）

張記法である fenced code block 記法の両方に対応している．

これらのセクションの内容は，作成する Web Components 自体に手を加える場合には必要になる

```
## style
explanation of style.

    .box { background: yellow; }
    p { font-weight: bold; font-size: 16px; }

## structure
explanation of structure.

‘‘‘
<div class='box'>
  <p>
</div>
‘‘‘
```

図 10 マークアップ・スタイルの一般的定義

```
## style
‘inline-block‘ style adjust container width
  automatically.

    #container {
      display: inline-block;
      font-size: 16px;
      height: 20px;
      padding: 0 4px;
      background-color: #4CCFFF;
      border: 1px solid #0084B4;
    }
    #num {
      color: #fff;
    }

## structure
simple div/span structure

    <div id="container">
      <span id="num">0</span>
    </div>
```

図 11 図 9 に対応するマークアップ・スタイルの定義例

が，ただ利用する際には不要であると考えられる．そのため，出力するドキュメントにはこれらの内容を反映しない．

4.3 プロパティ定義

プロパティに関する記述は，図 13 のように「properties」というレベル 2 のセクション内で行う．

セクション内では，定義リスト記法を用いて，1 つずつプロパティの定義を行う．用語としてプロパティ名を指定し，説明としてデータの型や初期

```
## properties

propertyName1
:    ‘"initial value"‘
:    type String

explanation of propertyName1

propertyName2
:    type Number
:    ‘1234567890‘

explanation of propertyName2
```

図 12 プロパティの一般的定義

```
## properties
count
:    type Number
:    ‘0‘

value of counter.
```

図 13 図 9 に対応するプロパティの定義例

値を記述する．説明の部分はバッククオート（‘）で囲んだ内容を初期値，type TYPENAME の記述を型名の定義として扱う．これらの指定の順番は特に問わない．また，これらの指定がなくても，未定義値を指定するため問題ない．次に定義リスト記法が現れるまでの文書構造は，そのプロパティに対する説明を記述するものである．

なお，定義リスト記法は PHP Markdown Extra[*16] で追加されたオリジナルの Markdown にはない拡張記法である．

「...」のように，HTML の開始タグのタグ名の後に続く href などの，タグの設定を記述するための識別子を属性（アトリビュート）という．ここで定義したプロパティは，Web Components のカスタムタグのアトリビュートとしてマークアップからも指定できるようにコード生成を行う．

4.4 メソッド定義

メソッドに関する定義は「methods」というレベル 2 のセクション内で行う．

[*16] https://michelf.ca/projects/php-markdown/extra/

表 1 指定可能なアノテーション

種類	意味
async	メソッドが非同期処理であることを表す
chainable	メソッドチェーンが可能であることを表す
depricated	非推奨メソッドであることを表す

```
## methods

functionName
:    param {type} arg1 explanation
:    param {type} [arg2] explanation
:    return {type} explanation on return value
:    depricated

explanation of function.

    // definition of function
    return arg1 + (arg2  || 0);
```

図 14 メソッドの一般的定義

プロパティと同様に定義リスト記法を用いて，メソッドを定義していく．説明の部分では，引数・戻り値・アノテーションの指定を行う．引数や戻り値の記述方法は図 17 のように行っていく．引数名を [] で囲む場合は，その引数が省略可能であることを示す．実際の関数定義における引数の順番は，ここでの記述順とする．

アノテーションはコード出力に影響を与えないが，ドキュメントを読むにあたり，ひと目でメソッドの特徴を分かるようにするためのものである．指定可能なアノテーションは表 1 に示す．

なお，無引数かつ戻り値を返さないメソッドでアノテーションをつける必要がない場合は，これらの記述と競合しない適当な説明を加えることでメソッド定義を行う．

次のメソッドに関する定義リスト記法が現れるまでの文書構造のうち，最後に出現したコードブロックをメソッドの処理内容としてコード生成に，それ以外をメソッドに関する動作説明や呼び出し例としてドキュメント生成に使用する．

4.5 ライフサイクルコールバックの記述

いくつかの特殊なメソッドは，インスタンス生

```
reset
:    no arguments

reset ‘count‘ to 0.

    this.count = 0;
    this.$.num.textContent = "0";

increase
:    param {Number} quantity increment of
  counter
:    return {Number} new value of counter.

increase value of counter by ‘quantity‘.
if ‘quantity‘ is 0, Inf or NaN, increase by 1.

    this.count += quantity || 1;
    this.$.num.textContent = this.count;
    return this.count;
```

図 15 図 9 に対応するメソッドの定義例

```
## lifecycle
### created
    console.log("callback - created");
### attached
    console.log("callback - attached");
### detached
    console.log("callback - detached");
### attributeChanged
    console.log("callback - attributeChanged"
      );
```

図 16 ライフサイクルコールバックの一般的定義

```
## lifecycle
### attached
    this.\$.num.textContent = this.count;
```

図 17 図 9 に対応するライフサイクルコールバックの定義例

成時や DOM ツリー挿入時などのタイミングで自動的に呼び出される．このようなメソッドをライフサイクルコールバックと呼び，次の 4 種類のメソッドがある．

created インスタンス生成時
attached DOM ツリーに挿入
detached DOM ツリーから削除
attributeChanged 属性値が追加・削除・変更

これらは先述のメソッド定義によっても定義可能であるが，メソッド名や引数の数などが決まっているため，「lifecycle」というレベル 2 のセクション内で定義可能である．

それぞれのメソッド名と同じレベル 3 のセクションを用意し，セクション内で最初に現れるコードブロックの内容を各ライフサイクルコールバックの処理としてコード生成に使用する．

4.6 テストコードの記述

メソッドなど，作成する Web Components の動作を確かめるためのテストコードの記述方法を 2 種類用意した（図 18）．

1 つは，「test」というレベル 2 のセクション内で，テストコードを定義していく方法である（図 18 後半）．このセクション内のレベル 3 の見出しで始まるセクションをそれぞれテストケースの定義として，テストコードを抽出し，実際にテストを行うためのコードを生成するのに使用する．

別のやり方は，メソッド定義時の定義リスト記法におけるメソッド情報の記述において，テストコードを書く方法である（図 18 前半）．こちらでは，メソッドへ渡す引数と想定する戻り値を記述することで，メソッドが正しくその値を返すかをテストするコードを生成する．この記法は JavaDoc において同様のテスト方法を提案した Vesa らの手法を参考にしている [6].

4.7 メタ情報の記述

提案手法では，YAML Front Matter を用いてメタ情報を記述する．

YAML Front Matter とは，Jekyll[*17] などの Markdown を用いたドキュメント生成するシステムで使われている，メタ情報などを記述するための手法である．Markdown 文書の先頭に YAML[*18] 形式（図 19）で文書のタイトルや著者，またはシステムの動作オプションを記述することで，後にそれを処理するシステムがこれらの情報を利用して文書を作成する．

この記法によって，Markdown では記述すること

[*17] https://jekyllrb.com/
[*18] http://yaml.org/

```
## method
add
:     param {Number} arg1 lhs
:     param {Number} arg2 rhs
:     return {Number} arg1 + arg2
:     test [1,2] 3

```javascript
return arg1 + arg2;
```

test
suite-1
supplementation

    it('always pass', function(){
      expect(true).to.be.ok;
    });

suite-2

    it('always failed', function(){
      expect(false).to.be.ok;
    });
```

図 18 テストの一般的記述

```
---
authors:
  - ebisawa
  - maruyama
  - terada
keywords:
  - Web Components
  - document
booktitle: Programming Symposium
volume: 2016
month: jan
---
```

図 19 YAML の例

が難しい作者情報やコードの生成オプション，外部ツールの使用に関する設定などを記述する．

5. 実装

5.1 概要

提案する記法に基いて記述した Markdown 文書をもとに，ドキュメントと Web Components のコードを出力するシステムを作成している．現在までに Ruby によるモックアップを作成し，公開に向け

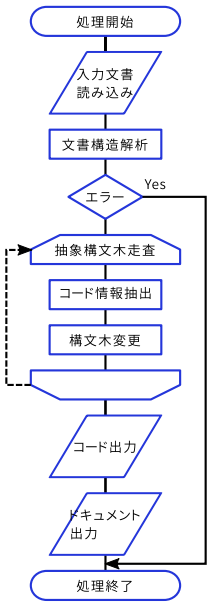

図 20 システムの動作フロー

Node.js[*19] 上で動作するモジュール（CLI）として実装している段階にある．準備が整い次第，ソースコードを GitHub を通じて公開し，システム自体も Node.js モジュールのパッケージマネージャである npm[*20] を通じて利用できるようにする予定である．

システムの動作フローを図 20 に示す．システムは YAML Front Matter 部の抽出と解析をしたのちに，標準の Markdown 文書として解析を行う．解析された Markdown 文書から「properties」や「methods」などの特別なセクションを探しだし，そこから Web Components のコード生成に必要な情報やコード断片を抽出する．その後，抽出したデータから Web Components のコードを生成すると同時に，元の文書はエンドユーザ向けのドキュメントとして必要な情報のみを残し，見やすくスタイリングして HTML ドキュメントとして出力する．

5.2 Markdown 文書の解析

Markdown 文書の構造を解析するために，本システムでは Pandoc[*21] を用いて Markdown の文書

[*19] https://nodejs.org/en/
[*20] https://www.npmjs.com/
[*21] http://pandoc.org/

構造を JSON 形式の抽象構文木として取得する．Pandoc を使用する理由としては，このように抽象構文木を取得できる機能を持つだけではなく，多くの拡張記法に対応しているためである．

Pandoc を用いて取得した JSON 形式の抽象構文木は，そのまま構造の解析とデータの抽出のために走査するには扱いにくい形式をしている．そこで，本システムではこの抽象構文木から DOM に似たデータ構造を用いてシステム内部用の抽象構文木を作成する．このシステム内部向けの抽象構文木を走査することで，出力の生成に必要なデータを収集するとともに，生成するドキュメントに合わせて抽象構文木の構造を変化させる．

5.3 出力の生成

本システムは最終的に Web Components のコードと HTML 形式のエンドユーザ向けのドキュメント，およびテストコードなどの付加的なファイルを出力する．

Web Components のコードは，文書の解析によって得られたプロパティやメソッドの情報やコードの断片を組み合わせて 1 つのコードを生成する．マークアップとスタイルに関するコードは，それぞれセクション内で最初に現れたコードの記述を抽出して使用する．メソッド定義においては，メソッドを定義する次の定義リストが現れる前の最後のコードの記述をメソッドの処理内容として抽出して使用する．このとき，`function(arg1, arg2,...){` などの関数定義の外皮とも言える部分の記述は，それまでのメソッドに関する引数の記述より生成が可能なため不要とする．

本システムはこの処理を，予め用意した雛形に与えられたデータを展開したテキストを生成する機能を提供するテンプレートエンジンというシステムを用いて実行する．Web Components のコードに共通する部分をひな形として用意しておき，それぞれ異なる部分に解析で得られた情報に基いたデータを当てはめていくことで，完成した Web Components のコードを生成する．テストコードなどの付加的なファイルもひな形を作ることが可能なため，解析によって得られた情報から同様に生成する．テンプレートエンジンは様々なものが利用可能であるが，本システムでは Mustache[*22] を使用した．

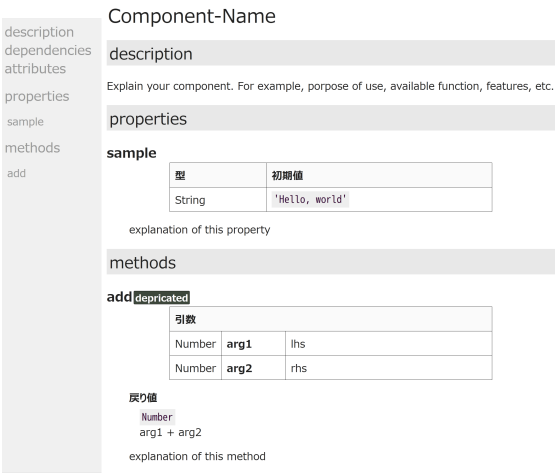

図 21 生成する API ドキュメント

API ドキュメントの生成は，解析と同時に変更を加えた内部向けの抽象構文木を再度 JSON 形式の抽象構文木へ変換し，これを Pandoc を用いて HTML に変換することで実現している．このとき，Pandoc での HTML 変換結果を別に用意した API ドキュメントのひな形に差し込むことで，読みやすいようスタイリングされた HTML 形式の API ドキュメントが生成されるようにしている．生成される API ドキュメントは図 21 のとおりである．なお，この例では生成される API ドキュメントの形式のみを示している．システムでは，はじめに述べた「必要と思われる情報」を網羅しているかどうかのチェックは行わない．

抽象構文木の解析の際，インストール方法や更新履歴など，エンドユーザに有用ではあるが API ドキュメントに含めるべきではない情報は抽象構文木から該当する部分木を取り除く．代わりに，その部分の情報は Markdown のテキストへと変換し，ReadMe テキストへまとめて出力することでエンドユーザが参照しやすいようにする．

5.4 外部ツール連携

近年の Web 開発では，ドキュメントの生成や

[*22] https://mustache.github.io/

コードの変換・チェック・難読化などの様々なタスクがあり，それらを自動的に行うための Grunt[*23] や Gulp[*24] といったツールの利用が盛んに行われている．特に，Web Components の作成に関わるものとしては各種代替言語から HTML/JavaScript/CSS を出力するための処理や，依存関係にある Web Components 群を 1 つのファイルに結合するものが挙げられる．

そこで，本システムでは YAML Front Matter でオプション指定を行うことで，外部ツールを用いた各種タスクを実行可能とした．また，マークアップ・スタイルの定義では fenced code block 記法を用いてコードを記述する際に，そのコードタイプを指定することで HTML・CSS の代わりに代替言語による記述が可能である．このとき，記述に用いられた代替言語から生成される HTML・CSS を使用してコード生成を行う．メソッドの定義に関しては，JavaScript の代わりに CoffeeScript[*25] を用いてコードの記述をするためのオプションを用意した．このオプションを有効にすることで，メソッド定義を CoffeeScript を用いて記述することができる．ただし，JavaScript と CoffeeScript の混在は許可しない．オプションを有効にした場合は，すべて CoffeeScript でメソッドが定義されているものと仮定してコード生成を行う．

6. 関連研究

本研究では対象として Web Components を選択したが，これ以外にも Web ページの構成要素をコンポーネント化し再利用するための仕組みが提唱されてきた．Benson らは Cascading Tree Sheet（以下 CTS）[3] を提案し，その有用性を示している [2]．Web Components とは異なり，CTS では CSS に似た構文を用いて，コンポーネント化したマークアップとの対応関係を記述することでそれを実現する．また，Benson は CTS を用いたコンポーネントを活用する，モックアップ駆動開発という新たな開発手法についても提唱している [1]．

Web Components を対象とした研究は，これがまだ仕様策定の段階にあるために数は少ないものの存在する．Krug らは SmartComposition[5] という Web Components を活用したマルチデバイス向け Web アプリケーションを作成するためのマッシュアップを実装，提案している．

Markdonw 記法の活用例として，Poley の RUMU Editor[8] や Leijen の Madoko[7] が挙げられる．RUMU Editor は Markdown 記法のテキストを HTML へと変換する際に，Web ページ作成用のフレームワークを組み合わせることで，整ったデザインの Web ページの作成コストの削減を行う Markdown エディタである．Madoko は Markdown 記法を拡張した記法を用いて，数式やプログラムコードなどを含む複雑なドキュメントを作成するためのオーサリングシステムである．同様のシステムに MacFarlane の Pandoc があるが，こちらは Markdown をはじめ HTML や LaTeX などの多くのマークアップ言語を入力として，それらの相互変換や PDF や EPUB などを出力するためのツールである．また，WebAPI の記述とドキュメント化をするために用いられる API Blueprint[*26] という Markdown を拡張した記法が存在する．

7. 終わりに

本研究では，Web Components に対する新たなドキュメントとプログラムコードの同時生成手法を提案した．現在 Web Components を構成する仕様は策定段階にあり，今後どのように Web 標準として成立するのか未確定な部分がある．しかしながら，扱う対象が現在と変わらない限り，今後も提案手法をはじめとする手法が有効になるのではないかと考えられる．

従来手法では対象とするプログラミング言語（のコメント記法）に応じて書き方を多少変更する，異なるシステムを用意する必要があったが，提案手法では実際に挿入するプログラムコードと最終的に出力するソースコードの変更をするのみで多くのプログラミング言語に対応できる利点があると考えられる．また，文芸的プログラミングの流れを

[*23] http://gruntjs.com/
[*24] http://gulpjs.com/
[*25] http://coffeescript.org/
[*26] https://apiblueprint.org/

汲んでいるため，ドキュメント駆動開発や Readme 駆動開発 [*27] などの開発手法との親和性が高いと考えられる．

近年ではエディタおよび統合開発環境の機能向上により，シンタックスハイライト（予約語などの色分け表示）やコードの推薦・補完などの作業効率を向上するための機能が設けられている．そのため，提案手法においても普及させることを考えると，このような機能を開発環境側が少ないコストで提供可能であるかどうかが問題となる．

始めにシンタックスハイライトについて考える．提案手法のベースとなっているのは Markdown 記法であるが，この記法については多くのエディタでシンタックスハイライトが可能である．また，vim[*28] や Sublime Text[*29] などの高機能なエディタにおいては，Markdown 中のコード記法内のコードについても，コードの種類を明示的に指定することでシンタックスハイライトが可能となるものもある．これらは，エディタが元々有している機能か，有志が公開するプラグインの導入によって提供される機能である．以上のことから，シンタックスハイライトに関しては汎用的な仕組みによって解決が可能であると考えられる．

同様に，コードの推薦や補完機能について考える．一般的に提供されているこれらの機能は，対象言語の予約語や，使用するライブラリやユーザが定義した識別子の掲示によって成り立っている．提案手法では，シンタックスハイライトの議論と同じくコード記法により記述したコードの種類を明示的に指定することで，その言語における予約語（例えば HTML のタグ名や CSS のプロパティ）や提案手法で記述されたものではない外部ライブラリの識別子を提示することは可能だと考えられる．しかしながら，提案手法によって記述している Web Components のメソッドやプロパティに関しては，Markdown 記法によって記述されているため，既存の仕組みによって提示することはできないと思われる．

以上の議論により，開発環境に要求される機能をある程度は既存の仕組みによって提供することが可能であると考えられるが，普及を目指すためにはこれらの機能を提供するための機能拡張システムなどの開発を要すると考えられる．

提案手法は Markdown を用いて記述を行うが，メソッドやプロパティの定義などにおける定義リストの利用など，一部の文書構造に意味を持たせているために十分なドキュメントの記述を阻害してしまう可能性がある．現状はあまり広く使われていないと考えられる定義リスト記法を利用することで，その影響を最小限にとどめるようにしているが，ユーザのドキュメントの書き方などによっては独自記法の導入を含め，記述方法の調整を行う必要があると考えられる．

今後は，実際にこの手法が Web Components に対しどの程度有用であるのかを定量的・定性的に検証していきたい．また，Web Components に限らず，別のプログラミング言語に対しても提案手法を適用できないか検討したい．

[*27] http://tom.preston-werner.com/2010/08/23/readme-driven-development.html
[*28] http://www.vim.org/
[*29] http://www.sublimetext.com/

参考文献

[1] Benson, E.: Mockup Driven Web Development, *WWW '13 Companion*, pp. 337–342 (2013).

[2] Benson, E.: Quantifying and Reducing the Cost of Web Edits, *CHI EA '13*, pp. 2659–2664 (2013).

[3] Benson, E. O. and Karger, D. R.: Cascading Tree Sheets and Recombinant HTML: Better Encapsulation and Retargeting of Web Content, *WWW '13*, pp. 107–118 (2013).

[4] Knuth, D. E.: Literate Programming, *Comput. J.*, Vol. 27, No. 2, pp. 97–111 (1984).

[5] Krug, M. and Gaedke, M.: SmartComposition: Enhanced Web Components for a Better Future of Web Development, WWW '15 Companion, pp. 207–210 (2015).

[6] Lappalainen, V., Itkonen, J., Isomöttönen, V. and Kollanus, S.: ComTest: A Tool to Impart TDD and Unit Testing to Introductory Level Programming, *ITiCSE '10*, pp. 63–67 (2010).

[7] Leijen, D.: Madoko: Scholarly Documents for the Web, *Proceedings of the 2015 ACM Symposium on Document Engineering*, DocEng '15, New York, NY, USA, ACM, pp. 129–132 (2015).

[8] Poley, E.: RUMU Editor: A non-WYSIWYG Web Editor for Non-technical Users, *CHI EA '10*, pp. 4357–4362 (2010).