B-032

内部の HTML 構造を自由に設計できるウィジェットの実現方法
A design method of widgets with flexible HTML structure

日野 克哉† 中島 暢康†
Katsuya Hino Nobuyasu Nakajima

1. はじめに

HTML を使用したアプリケーションの画面を開発する方法には、単純な機能をもつ HTML 要素を配置する方法と、ライブラリによって提供されるウィジェットを配置する方法がある。ウィジェットとは、複数の HTML 要素を組み合わせることで複雑な外観が実現され、それに対して JavaScript で高度な機能をもたせた画面部品である。jQuery UI[1]や Dojo Toolkit[2]などのライブラリでは、このようなウィジェットを多数用意しており、HTML での高度な画面開発をしやすくしている。

しかしこれらのライブラリではウィジェットを実現するための HTML 要素の組み合わせ方はほとんど固定されており、ウィジェットを使用する側からは HTML 要素の組み合わせ方を修正することは難しい。HTML 作成者がウィジェットを画面上に配置するには、ウィジェットごとにライブラリで指定された構造をそのまま HTML 内に記述するか、または実行時にライブラリがウィジェットの HTML 要素構造を生成するターゲットの空 HTML 要素、つまりプレースホルダを記述する。ウィジェットを配置するための空の HTML 要素、つまりプレースホルダを記述し、実行時にライブラリが中の HTML 要素を生成する。そのため HTML 作成者はウィジェット内部の HTML 要素構造を HTML 作成時には自由に設計することができない。仮にライブラリで指定された構造と異なる HTML を記述したり、ライブラリによって生成された HTML 要素の構造を実行時にスクリプトなどから動的に変更したりすると、ウィジェットの操作を行う API は期待通りの動作をしなくなる。例えば、ボタン内に装飾目的の画像を挿入する場合、button 要素内に img 要素を挿入することは問題ないが、この状態でボタンに表示されるテキストを変更する API を呼び出すとテキストの変更と同時に挿入した img 要素が消えてしまう（図 1）。これは API があらかじめ決められたウィジェット内部の HTML 要素構造を想定しているからであり、この例ではテキストを表示する HTML 要素が外側の button 要素であると想定されていたからである。

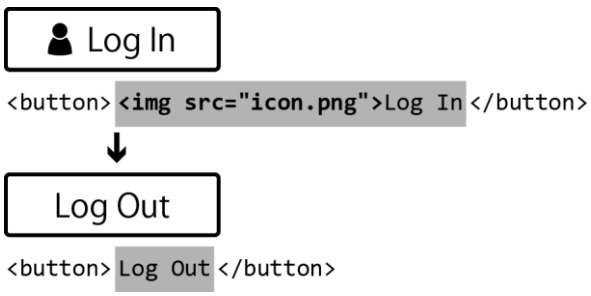


図 1 ボタンのテキスト変更 API を呼び出す例

†（株）東芝 ソフトウェア技術センター,
Corporate Software Engineering Center, Toshiba Corporation

このウィジェット配置方法では、画面の外観を設計する HTML 作成者がウィジェット内部の外観までは設計できないため、画面全体の外観に統一感をもたせづらくなってしまう。上記ライブラリでも HTML 作成者が CSS を作成することでウィジェット内部の HTML 要素の外観を変更できるが、外観の変更は CSS で表現できる範囲に限られる。ウィジェットの内側に凝った装飾やラベルを追加するためには、ウィジェット内部の HTML 構造も変更する必要がある。

本論文では、ウィジェット内部の HTML 構造とウィジェットの動作を分離することで、ウィジェット開発者でなくても内部の構造を自由に設計できるようにし、HTML 作成者が装飾などの目的でウィジェット内部に任意の HTML 要素を追加しても問題なく動作するウィジェットの実現方法を提案する。

2. 提案するウィジェットの実現方法

本論文で提案するウィジェットは、ウィジェットの機能を実現するために必要な HTML 要素の種類を定義しているだけで、内部 HTML 要素の完全な構造を決定しない。ウィジェットの JavaScript ライブラリは、ウィジェットの初期化時に内部の HTML 要素を生成することはせず、既存の HTML 要素の集合に対して、ウィジェットで定義された HTML 要素の操作を行うだけである。例えば一般的なボタンの機能を実現するためには、クリックを受け付けるための HTML 要素とテキストを表示するための HTML 要素が最低限必要だと考えられるので、本手法ではこの 2 種類の HTML 要素さえあればどのような構造の HTML 要素群でもボタンウィジェットとして動作する。またこの 2 種類が同一の HTML 要素に対応していても問題ない（図 2）。なお本論文ではこのような特定の役割をもつ要素を構成要素と呼び、さらにウィジェットのルート要素を識別構成要素と呼んでいる。

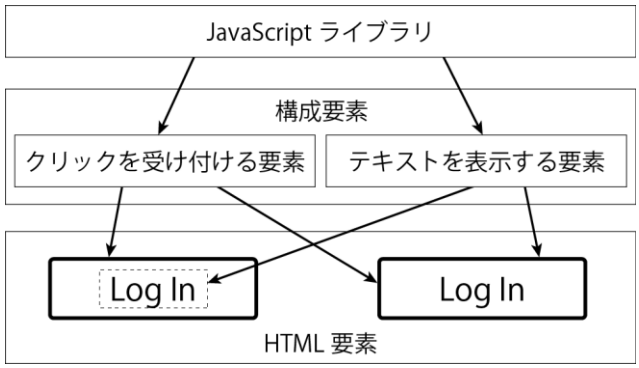
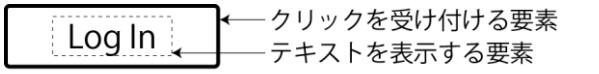
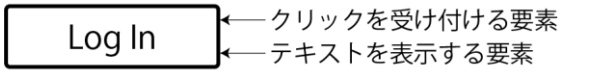


図 2 異なる構造をもつボタンウィジェットに対して JavaScript ライブラリが操作を行う例




```
<button class="ts-button">
  <span class="ts-button-text">Log In</span>
</button>
```




```
<button class="ts-button ts-button-text">
  Log In
</button>
```

図 3 ボタンウィジェットの記述例

各構成要素には制約条件が存在する。本論文のウィジェットでは、構成要素に制約条件を加えることでウィジェットを動作させるためのおおまかな構造を決めている。制約条件は HTML 作成者が記述した内部 HTML 要素構造を解釈するためにも使用される。制約条件にはクラス名、HTML 要素型、要素の存在数、要素の親子関係があり、これらの制約条件に適合した HTML 要素が当該構成要素として認められる。クラス名は HTML 要素に構成要素を明示的に指定するために使用される。例えばボタンウィジェットのテキストを表示する構成要素であったら、「ts-button-text」というクラス名を HTML 要素の class 属性に追加する（図 3）。HTML 要素型は構成要素に指定できる HTML 要素の型を限定する。例えばテキストの入力を受け付ける構成要素は「type 属性が text である input 要素」に限定する。要素の存在数は、その構成要素がウィジェットに必須か任意か、1 つしか存在できないのか複数存在できるのかを指定する。例えばメニューの項目に相当する HTML 要素の存在は任意で複数存在できるが、そのコンテナは必須で 1 つしか存在できない。親子関係は構成要素の親要素を限定する。例えばリストボックスの項目はコンテナの子要素でなければならない。

HTML 作成者がウィジェットを配置するには、プレースホルダだけでなく内部の HTML 要素構造まで完全に記述する。すべてのウィジェットには識別構成要素を含めた構成要素が 1 つ以上定義されており、HTML 作成者はウィジェットを配置するとき、制約条件を考慮しながら構成要素を含めつつ、ウィジェット内部の HTML 要素構造を設計する。各構成要素には実行時にライブラリから受ける変更があらかじめ規定されている。HTML 作成者はこれらの情報を利用してウィジェット内部の HTML 要素構造を設計することで実行時の意図しない変更を防ぐことができる。例えば、ボタンの表示テキストを格納するテキスト構成要素が、要素の内容が任意のプレーンテキストで置換されると規定されている場合、HTML 作成者はこの HTML 要素の内容に HTML タグを含めるべきでないことを知ることができる。また、構成要素以外の内部 HTML 要素は何の役割ももたず、実行時にライブラリから変更を受けることもない。そのため、装飾のための HTML 要素をウィジェット内部に追加してもウィジェットの動作には影響を与えない（図 4）。

ライブラリがウィジェットを操作するためには HTML 作成者が設計した内部 HTML 要素の構造を把握する必要がある。例えば表示されるテキストを変更する API を備えたボタンウィジェットは、どの HTML 要素がテキストを格納しているかを把握する必要がある。そのためライブラリは内部 HTML 要素をアプリケーションの実行時に解釈し、各構成要素に対応する HTML 要素を、制約条件を手がかりに検索する。具体的にはクラス名と HTML 要素型が一致する要素をすべて検索し、見つかった要素が存在数と親子関係の制約条件を満たしているかを確認する。制約条件を満たしている場合は要素を構成要素として記憶しウィジェットの操作 API から使用するが、そうでない場合は異常終了する。


```
<button class="ts-button">
  <img src="icon.png">
  <span class="ts-button-text">Log In</span>
</button>
```

図 4 ボタンに装飾のための要素を追加する例

3. 効果

本論文で提案したウィジェット実現方法では、構成要素はライブラリから受ける変更が明確に規定されているため、HTML 作成者は構成要素ではない装飾用の HTML 要素を追加するための適切な場所がわかるようになった。これにより、例えば HTML 作成者がウィジェット内部の適切な場所に装飾用の HTML 要素を入れても、実行時にその HTML 要素が保たれるようになった。

また、ウィジェット内部の HTML 要素構造とウィジェットの動作とが分離され、HTML ウィジェット内部の HTML 要素の組み合わせを HTML 作成者が設計できるようになった。これにより HTML 作成者は必要な構成要素さえ正しく記述すればウィジェットのインスタンスごとに内部の HTML 要素構造を変更することもできるようになった。例えば、日付を入力できる入力欄ウィジェットで、あるインスタンスでは日付を選択するためのカレンダを表示するためのボタンを表示したいが、外観設計の都合上、別のインスタンスではこのボタンを表示したくない場合、HTML 作成者はボタンを表示したくないインスタンスだけボタンに相当する構成要素を省略すればよい。ボタンの有無は HTML を記述した時点で反映されるため、HTML 作成者も直感的にボタンの有無を確認できる。

4. 結論

ウィジェットの JavaScript ライブラリから内部 HTML 要素の生成を分離することで、ウィジェット外部からもウィジェット内部の外観を設計できるようになった。画面の外観に統一感をもたせるためにはウィジェット内部の外観についても HTML 作成者が手がける必要であるため、本手法はより統一感のある画面開発に有効である。

本方法の課題は、ウィジェット内部の HTML 要素まで記述することで全体の見通しが悪くなることが挙げられる。この課題はウィジェット内部の HTML を隠蔽する仕組みで解決される。例えば Web Components[3]で提案されているカスタム要素を使用すれば、ウィジェット内部の HTML 構造が隠蔽され見通しがよくなる一方、HTML 作成者はウィジェット内部の HTML 構造も編集できる。

参考文献

[1] jQuery UI, http://jqueryui.com/
[2] Dojo Toolkit, http://dojotoolkit.org/
[3] Introduction to Web Components (W3C Working Draft), http://www.w3.org/TR/2013/WD-components-intro-20130606/