Vol.2012-SE-176 No.5
Vol.2012-EMB-25 No.5
2012/5/21

Webアプリの動的部分に着目したグレーボックス統合テストの提案

坂本 一憲[1,a)]　海津 智宏[2,b)]　波村 大悟[2,c)]　鷲崎 弘宜[1,d)]　深澤 良彰[1,e)]

概要：Web アプリケーションは様々なブラウザやプラットフォーム上で動作する上，複数のシステムから構成される非常に複雑なソフトウェアである．そのため，Web アプリケーションの検証に莫大なコストが必要であり，効率の良いテスト手法の確立が大きな課題である．これまでホワイトボックス単体テストとブラックボックス結合テストを組み合わせて，Web アプリケーションがテストされてきた．しかし，前者のテストでは本番環境でテストできず，後者のテストではテストすべき箇所が不明瞭であるという問題がある．そのため，テストすべき箇所を明瞭化した上で，本番環境でテストを実施することが困難である．そこで，HTML テンプレートを解析して，HTML 文章中の動的に生成される部分に着目するグレーボックス結合テストを提案する．本手法は，本番環境でのテストを可能として，さらに，テストすべき箇所を明瞭化することで，従来のテスト手法よりも効率の良いテストを実現する．

キーワード：ソフトウェアテスト, Web アプリケーション, テストカバレッジ, テンプレートエンジン, グレーボックステスト，結合テスト

A Proposal of Gray-box Integration Testing for Web Applications

Sakamoto Kazunori[1,a)]　Tomohiro Kaizu[2,b)]　Daigo Hamura[2,c)]　Hironori Washizaki[1,d)]
Yoshiaki Fukazawa[1,e)]

Abstract: Web application is very complex software because it runs on various browsers and platforms. It requires high costs to verify web applications. Thus, constructing efficient means for web application testing is one of important issues. Existing approaches for web application utilize the combination of white-box unit testing and black-box integration testing. However, the former can not conduct testing on the real environment and the latter can not find the part that should be tested. Therefore, it is hard to do testing on the real environment finding such part.
In this paper, we propose gray-box integration testing focusing on dynamic parts of html documents. Our approach can conduct testing on the real environment finding the part that should be tested.

Keywords: Software testing, Web application, Test coverage, Template engine, Gray-box testing

1. はじめに

インターネットの普及に伴い Web アプリケーション（以降，Web アプリ）の社会的な重要性は増す一方である．Web アプリはサーバークライアントモデルによって構成されており，サーバーサイドとクライアントサイドの両方でプログラムが協調して動作する．サーバープログラムは様々なシステムから構成されている上，サーバーおよびクライアントプログラムの動作環境が多様化している．Web アプリは非常に複雑なソフトウェアであるため，効率良く品質を保証するためのテスト手法が求められている．

[1] 早稲田大学 Waseda University
[2] グーグル株式会社 Google Inc.
[a)] kazuu@ruri.waseda.jp
[b)] tkaizu@google.com
[c)] daigoh@google.com
[d)] washizaki@waseda.jp
[e)] fukazawa@waseda.jp

Web アプリのサーバープログラムは様々なシステムと相互に作用する．例えば，Google が提供する Gmail では，サーバープログラムが配置されている Web サーバー，ユーザのアカウント情報を管理する認証サーバー，メールデータなどを管理するデータベースサーバがそれぞれ協調して動作している．また，クラウドコンピューティングの普及に伴い，サーバープログラムの動作環境が多様化している．そのため，Web アプリを稼働させる本番環境と Web アプリを開発する開発環境とでは，動作環境の差異から Web アプリの挙動が異なるケースが存在する．

Web アプリのクライアントプログラムの動作環境として，様々なブラウザやプラットフォームが挙げられる．例えば，Microsoft の Internet Explorer や Mozilla の Firefox，Google の Google Chrome といったブラウザが存在する．また，上記のブラウザが動作するプラットフォームとして，Windows や MacOS などの OS が搭載されたパソコンと，Android や iOS などの OS が搭載されたスマートフォンが存在する．しかし，これらのブラウザやプラットフォーム間には互換性がなく，あるブラウザ上で正しく動作する Web アプリが，異なるブラウザもしくは異なるプラットフォーム上で動作しない問題が生じている．

ソフトウェアテストはいくつかの観点から分類することができる．テスト対象の粒度という観点からは，ソフトウェアを構成するモジュール単体を対象とした単体テスト，モジュールを組み合わせたソフトウェア全体を対象とした結合テストに分類される[*1]．また，テストに利用する情報という観点からは，ソースコードを参照してプログラムの構造などに着目するホワイトボックス（以降，WB）テスト，ソースコードを参照せずにソフトウェアの仕様書を元に実施するブラックボックス（以降，BB）テスト，一部のソースコードを参照する両者の中間に当たるグレーボックス（以降，GB）テストに分類される．

Web アプリ開発ではしばしばテンプレートエンジンが利用される．テンプレートエンジンとは，HTML テンプレートに対して，プログラムを実行した際に得られた値の文字列表現を埋め込むことで，HTML 文章を生成するシステムである．テンプレートエンジンはサーバーサイドで動作するソフトウェアとクライアントサイドで動作するソフトウェアに分かれており，前者の例として Velocity が，後者の例として Closure Templates [*2]が挙げられる．

Web アプリは静的に HTML 文章を出力するだけではなく，上述のようなテンプレートエンジンなどを用いて，静的に得られる部分とプログラムの実行状態に依存して動的に生成される部分を併合させて，ユーザが閲覧する HTML 文章を生成する．最終的に得られる HTML 文章において，静的に得られる部分は常に同じ文章になっているため，容易にバグを検出することが可能である．一方，動的に生成される部分はプログラムの実行状態に依存して変化する．そのため，大部分のケースでは正常に HTML 文章が生成されても，一部のケースで正常に HTML 文章が生成されない可能性があり，静的に得られる部分と比較してバグの検出が困難である．しかし，WB 単体テストでは本番環境でのみ再現可能なバグを検出することができない．一方で，BB 結合テストでは静的な部分であるか動的な部分であるかの判断が難しく，動的に生成される部分に対して十分にテストが行われない可能性がある．

我々は，プログラムの実行状態によって HTML 文章が動的に変化する部分に，バグが含まれやすくテストすべきであるという考察に基づいて，HTML テンプレートを解析することで，Web アプリケーションの動的な部分に着目した GB 結合テストを提案する．さらに，ユーザのテストコード記述を支援するために，PageObjects デザインパターンに基づいて，動的に生成される部分へのアクセスメソッドを備えたスケルトンテストコードを自動生成するツール POGen を開発した．その上で，生成したアクセスメソッドに基づいて，新しいテスト十分性の指標としてテンプレートカバレッジを提案する．現在，POGen はオープンソースソフトウェアとして公開中である [1]．なお，本論文では Web アプリケーションがテンプレートエンジンを利用することを前提に論じる．

2. 従来のテスト手法と問題点

Web アプリ開発では，開発したコンポーネントの一部を対象とした単体テストを行った上で，実際に Web アプリを動かす本番環境で結合テストを行うというプロセスが取られる． 単体テストは Web アプリの開発者自らが行う場合が多く，ソースコードを理解した上で実施する WB 単体テストが利用される．一方，結合テストは Web アプリの開発者以外が行う場合が多い．特に，Web アプリを開発した組織とは異なる組織が実施するケースもあり，ソースコードを参照しない BB 結合テストが利用される．表 1 に Web アプリにおけるテスト技法の比較を示す．

2.1 ホワイトボックス単体テスト

図 1 に Web アプリの構成と WB 単体テストの関係を示す．サーバープログラムと HTML テンプレートが存在しており，これらを動作させることで HTML 文章が生成され，あるプラットフォーム上のブラウザを通して閲覧することで，クライントプログラムが動作するという構成を考える．なお，PHP のように HTML テンプレートとサーバープログラムを明確に分離できないケースも存在するが，本手法で単に HTML テンプレートとして扱う．

[*1] さらに，システムテストや受け入れテストといった分類も可能だが，本論文では簡単のためにモジュールを結合するか否かで単体テストか結合テストかに分類する．
[*2] サーバーサイドとクライアントサイドの両方で動作する．

表 1 Web アプリにおけるテスト技法の比較
Table 1 A comparison of web application testing

	WB 単体テスト	BB 結合テスト
言語依存	依存	非依存
ソースコード	必要	不要
本番環境でのテスト	困難	可能
定量的なテスト十分性の測定	可能	困難
テスト設計における属人性	低い	高い

WB 単体テストでは Web アプリのソースコードを参照して，テストコードを記述することで自動テストを実施する．その際，xUnit と呼ばれる単体テストのテスティングフレームワークを利用することが多い．そのため，サーバープログラムが Java で記述されている場合は，例えば，JUnit を用いてテストを行うケースが考えられる．

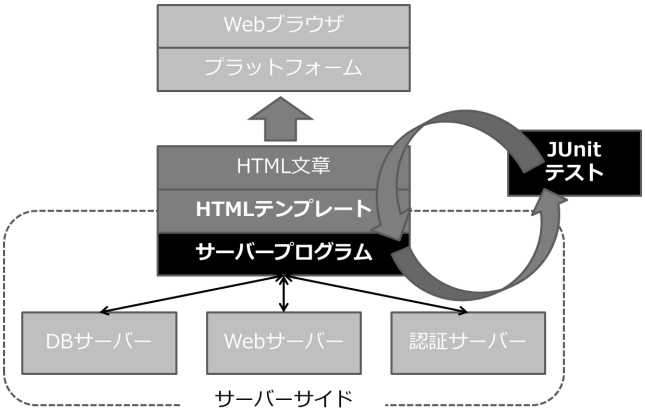

図 1 Web アプリの構成と WB 単体テストの関係
Fig. 1 The relation between the structure of web application and white-box unit testing

xUnit はプログラミング言語毎に開発されていて，例えば，Java に対応する JUnit や.NET に対応する NUnit などが存在する．テスト対象のソフトウェアが記述された言語に対応する xUnit を利用する必要がある．しかし，JUnit と NUnit では検証用に類似したメソッドを提供しているにも関わらずメソッドの引数の順番が異なるように，テスティングフレームワーク毎に仕様の差異が存在する．そのため，表 1 のように言語への依存があり，ユーザは各テスティングフレームワークを学習する必要がある．

WB 単体テストはモジュール単体をテストするため，モジュール間の結合に潜むバグを発見できない．また，Web アプリが複数のシステムと相互に作用する上，様々な実行環境が存在する．そして，システム間の結合においてバグがあったり，開発環境では再現されないバグがあったりする．そのため，実際に Web アプリを稼働させる本番環境でテストする必要がある．しかし，表 1 のように WB 単体テストでは，モックなどを利用してモジュールを単体でテストするため，本番環境でテストすることが難しい．

WB 単体テストはソースコードを参照するため，テストを実施した際に製品コードが実行された割合を示すテストカバレッジを算出できる．テストカバレッジはソースコードと実行履歴から定量的に算出可能なテスト十分性の指標である．一般に，テストカバレッジの目標値を掲げることで，テストの品質を保証できる． そのため，テストカバレッジを利用することで，テストにおける属人性を排除できる．したがって，表 1 のように，WB 単体テストでは定量的なテスト十分性の指標を用いて属人性を低減する．

なお，WB 結合テストは次の理由から実施が困難である．

- Web アプリを構成するシステムが多岐にわたり，全てのシステムのソースコードを入手することは難しい．
- 各システムが異なるプログラミング言語でばらばらに開発されており，全てに対応するテスティングフレームワークやカバレッジ測定ツールが存在しない．
- 全てのシステムを組み合わせると非常に大規模なソフトウェアとなるため，全ての箇所を検証することが難しく，かえってテストすべき箇所が不明瞭となる．

2.2 ブラックボックス結合テスト

図 2 に Web アプリの構成と BB 結合テストの関係を示す．BB 結合テストでは Web アプリの仕様書を参照して，ブラウザを介して Web アプリを操作することでテストを実施する．そのため，Web アプリが動作さえしていれば，どのような形で Web アプリが構成されていようとテストを実施できる．BB 結合テストは Web アプリのソースコードを必要としないため，Web アプリを開発した組織とは異なる組織にテストの業務を外注することが多い，

人手によるテストの実施はソフトウェアによる自動テストよりも高いコストが必要になり，特に，繰り返しテストを行うようなリグレッションテストではコストの差が顕著である．そのため，BB 結合テストでは Selenium などのユーザのブラウザ操作をエミュレートするツールが利用されており，BB 結合テストの実施コストを削減する工夫がなされている．いずれの方法であっても，BB 結合テストはブラウザを介してテストを実施するので，表 1 のようにテストの実施はプログラミング言語に依存しない．

BB 結合テストはブラウザを介してテストを実施するため，表 1 のように Web アプリを本番環境に配置した上でテストできる．そのため，WB 単体テストとは異なり，本番環境で発生する全てのバグを検出することが可能である．

しかし，BB 結合テストは仕様書を元にテストの設計を行うため，テストの質がテストの設計者に強く依存してしまう問題がある．また，テスト十分性を考えた際に，全ての Web ページをテストしたかどうか，ページ間の遷移モデルに基づいて全ての遷移をテストしたかどうか，ページ上に存在する全ての GUI コンポーネントをテストしたかどうかなど，いくつかの指標が存在する [3][4]．しかし，これら

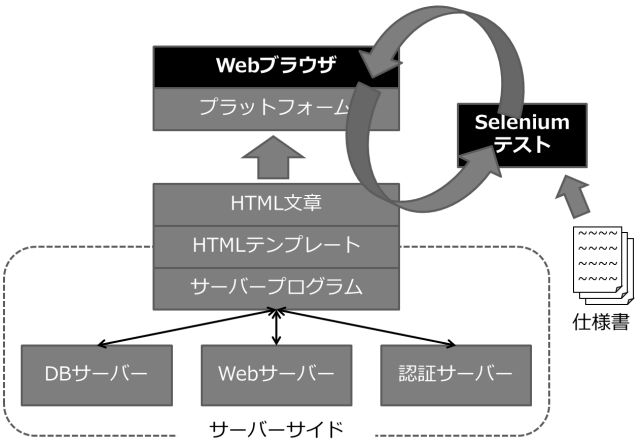

図 2 Web アプリの構成と BB 単体テストの関係
Fig. 2 The relation between the structure of web application and black-box integration testing

の指標を定量的に測定するツールが存在しない上，そもそもこれらの指標に基づいてテストしてもバグを発見できないケースがある．そのため，表 1 のように定量的なテスト十分性の測定が困難で，テスト設計において属人性が強く残ってしまいテストの品質がばらつく問題がある．なお，上述した指標では不十分であることを次節にて説明する．

ブラウザ操作をエミュレートするツールを使用してテストコードを記述する場合，記述・保守コストが莫大であるため，十分にテストされないケースが存在する．ブラウザ操作の記述は HTML 文章の構造に基づいて記述されることが多い．しかし，Web アプリは配置が容易にでき変更頻度が高い上，そもそもユーザインタフェースの変更は頻繁に行われる．そのため，HTML 文章の構造が変化して，ブラウザ操作が正常に動作しない問題がある．

例えば，二つの値を加算した結果を表示する Web アプリを考える．図 3 は Closure Templates を利用して，二つの値とその加算結果を表示する HTML テンプレートである．Web アプリの実行時に {$left} と {$right} に加算する値，{$answer} に加算結果の値の文字列表現が埋め込まれる．図 4 は Selenium を利用して，加算結果の値をブラウザから取得して検証する Java のテストコードである．図 4 のテストコードは HTML 文章の構造に強く依存しており，例えば，{$answer} を囲むタグを div から p に変更すると，正常に加算結果の値を取得できなくなる．

2.3 従来のテスト手法における問題点

WB 単体テストは従来通り必要な手法である．一方，単体テストだけではバグを発見できないケースがあるため，結合テストも同様に必要である．しかし，BB 結合テストには上述のようなデメリットが存在しており，かといって，WB 結合テストを実施することは難しい．したがって，WB 単体テストと BB 結合テストを組み合わせた従来のテスト手法において次の問題点が挙げられる．

```
1 {template .addition}
2 <p>
3   {$left}+{$right}=<div>{$answer}</div>
4 </p>
5 {/template}
```

図 3 Closure Templates を利用して二つの値を加算した結果を表示する HTML テンプレート
Fig. 3 A html template for printing the result of addition with Closure Template

```
1 WebDriver driver = new ChromeDriver();
2 WebElement element = driver
3     .findElement(By.cssSelector("p > div"));
4 assertEquals("3", element.getText());
```

図 4 Selenium を利用して加算結果の値を検証する Java テストコード
Fig. 4 Java test code verifying the result of addition with Selenium

- 問題 1) BB 結合テストにおいて明確に定量的なテスト十分性の指標が存在しない．そのため，BB 結合テストの属人性が高く，結合テストでしか検出できないバグを見過ごすケースが存在する．
- 問題 2) BB 結合テストにおいてテストコードの記述コストおよび保守コストが大きい．そのため，BB 結合テストにおける自動テストを実施するコストが大きく，網羅的なテストを繰り返し行うことが難しい．

3. 従来手法では発見できないバグの具体例

本節では，従来の WB 単体テストおよび BB 結合テストでは検出の難しい 4 種類のバグについて議論する．

(1) **サーバープログラムと他のシステムもしくは複数のシステム間の結合におけるバグ：**外部の Web アプリと連携して情報を収集して，収集したデータを加工してから，HTML 文章を通して情報をユーザに提示する Web アプリを考える．情報を収集するシステムと情報を加工するシステムが分かれており，収集した情報を蓄積するデータベースと加工した情報を蓄積するデータベースが存在する．

連携している外部の Web アプリの仕様変更のため，収集した情報を蓄積するデータベースのスキーマを変更する必要が生じた場合，情報を収集するシステムにのみ変更が必要だと想定したとしても，変更の影響範囲の分析に抜けがあり，情報を加工するシステムにも変更が必要であるケースが考えられる．つまり，複数のシステムが協調して動作していると，一部のシステムの仕様変更であっても他のシステムに想定外の影響が

生じて，結果的に Web アプリが正常に動作しなくなるバグが起こりうる．
このようなバグは WB 単体テストでは発見することができず，一方，BB 結合テストでは，ページや遷移の網羅をしたとしても，加工した情報を出力する部分のテキストについて検証を行わなければ，上述のようなシステム間のバグを検出することは難しい．

(2) **JavaScript によって HTML 文章を書き換える際に発生するバグ：**JavaScript を利用してボタンを押すと DOM ツリー上のいくつかの要素が増える機能を考える．このような機能はブラウザ上で実際にボタンを押す操作を行わなければテストできない．また，JavaScript はブラウザによって動作が異なり，例えば，Internet Explorer では `innerText` プロパティが動作するが，Firefox では動作しないといった問題があるため，こうした互換性の検証をするために複数のブラウザを実際に操作してテストする必要がある．
このようなバグは WB 単体テストでは発見することができない．一方，BB 結合テストでは発見することができるものの，ボタン押下で増える要素のうち 1 つの要素だけについて検証をしており，他の増える要素について検証していないといった漏れが生じる．

(3) **HTML 文章中の静的に得られる部分と動的に生成される部分の文字列結合におけるバグ：**ログイン名を表示するような Web アプリを考えたときに，サーバープログラムでは名前と敬称を結合してテンプレートエンジンに文字列を渡すとする．しかし，HTML テンプレートでは受け取った文字列を動的な部分として出力した上で，静的に得られる部分に敬称も記載してしまうケースが考えられる．
このような場合，名前に対して敬称が二重に付与されてしまうが，WB 単体テストではこのバグを発見することができず，一方，BB 結合テストでは発見することが可能であるものの，ログイン名を表示する部分に対するテストを行わなければ発見できない．

(4) **サーバープログラムと HTML テンプレート間の結合におけるバグ：**上述の例と同様にログイン名を表示するような Web アプリを考えたときに，サーバープログラムでは `name` という変数名をテンプレートエンジンに渡していたのに対して，HTML テンプレートでは誤って `account` という名前の変数を出力する場合，想定する文字列とは異なる文字列が表示されてしまうケースが考えられる．
同様に WB 単体テストでは上記のバグを発見することができず，一方，BB 結合テストでは発見することが可能であるものの，ログイン名を表示する部分に対するテストを行わなければ発見できない．

4. グレーボックス結合テストとスケルトンテストコード自動生成ツール POGen

我々はソースコードの一部である HTML テンプレートを解析することでテストすべき箇所を明瞭化した GB 結合テストを提案する．また，テストすべき箇所に基づいたテンプレートカバレッジを提案することで，解決 1) 定量的にテスト十分性を測定する方法を提供する．さらに，解析した情報に基づいて Selenium を利用するスケルトンテストコードを自動生成するツール POGen を開発した．POGen は保守性の高い PageObjects デザインパターンに基づいてスケルトンテストコードを生成して，解決 2) テストコードの記述コストおよび保守コストを低減する．

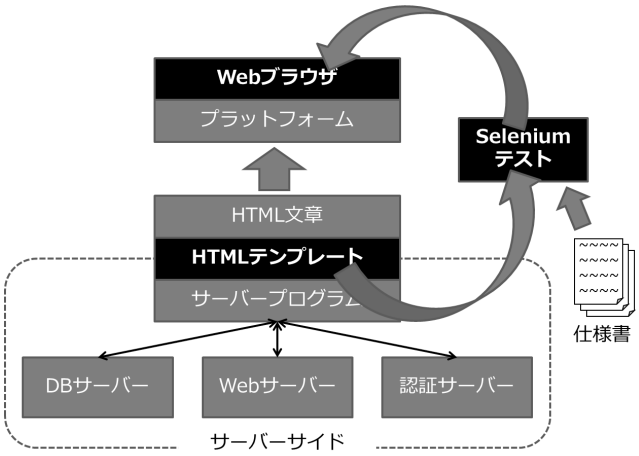

図 5 Web アプリの構成と GB 結合テストの関係
Fig. 5 The relation between the structure of web application and gray-box integration testing

図 5 で Web アプリの構成と提案する GB 結合テストの関係を示す．GB 結合テストでは仕様書のみからテストを実施するのではなく，ソースコードの一部である HTML テンプレートを参考にしてテストを実施する．最終的に得られる HTML 文章において，静的に得られる部分と動的に生成される部分では，動的に生成される部分に注目してテストを実施する必要がある．そのため，POGen では HTML テンプレートを解析することで HTML 文章中の動的に生成される部分を検出して，その部分に対してアクセスするメソッドを持つスケルトンテストコードを自動生成する．

図 6 で POGen のアーキテクチャを示す．POGen は HTML テンプレート解析部，HTML テンプレート変形部，スケルトンテストコード生成部の 3 つの機能部から成る．

- HTML テンプレート解析部は，HTML テンプレートを解析して，HTML 文章中の動的に生成される部分，すなわち，テンプレート変数の文字列表現を出力する部分を検出する．例えば，図 3 の例では，{`$left`}，{`$right`}，{`$answer`} が該当する．その上で，HTML テンプレート解析部に動的に生成される部分の位置を

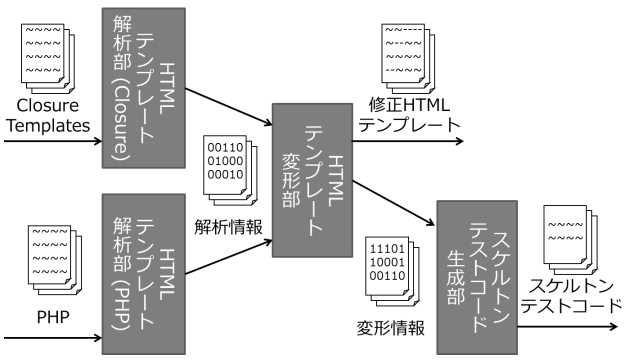

図 6 POGen のアーキテクチャ
Fig. 6 The architecture of POGen

伝える．なお，HTML テンプレート解析部はテンプレートエンジン毎に実装しており，現状では Closure Templates のみに対応している．

- HTML テンプレート変形部は，位置情報を元に動的に生成される部分をテキストとして持つ HTML タグを特定して，特定したタグに該当する DOM ツリーの要素に Selenium から一意にアクセス可能な属性値を追加する．現状では ID 属性が付与されていない場合，一意な ID の属性値を生成して追加している．例えば，図 3 の HTML タグに POGen が ID 属性を追加すると図 7 のようになる．POGen は GB 結合テストを実施する際に元のファイルと置き換えるために，ID 属性を追加した修正 HTML テンプレートを出力する．

```
{template .addition}
<p id="_pogen_1">
  {$left}+{$right}=
  <div id="_pogen_2">{$answer}</div>
</p>
{/template}
```

図 7 HTML テンプレート変形部が ID 属性を追加した HTML テンプレート
Fig. 7 A html template where POGen inserts an ID attribute

- スケルトンテストコード生成部は，追加した ID 属性の情報を元に PageObjects デザインパターンに基づいた Selenium を利用する Java のスケルトンテストコードを生成する．
  PageObjects デザインパターンは 1 ページを 1 クラスで表現することで，保守性の高いテストコードを設計するためのデザインパターンである [2]．ページクラスは，ページ中の GUI コンポーネントや特定の HTML タグに該当する DOM ツリーの要素を参照するフィールド変数と，ページ中の特定のテキストを取得するメソッド，別のページに遷移するためのメソッドなどを持っており，ページクラスが持つメソッドを適切に呼び出すことでテストケースを記述できる．一般に，ページの HTML 構造が変化することで DOM ツリーへの参照方法が変化することは多いが，ページが持つ情報や遷移先が変化することは少ない．そのため，PageObjects デザインパターンでは，仕様変更の影響を受けやすい DOM ツリーの参照処理をページクラスに記述して，それを利用して仕様変更の影響を受けにくいテストケースを記述することで，仕様変更が起きた際の影響を抑えて保守コストを低減する．
  スケルトンテストコードは HTML 文章中の動的に生成される部分にアクセスするメソッド，具体的には，動的に生成される部分を直接のテキストとして持つ HTML タグに該当する DOM ツリーの要素を取得するメソッドを提供することで，テストケースから動的に生成される部分とその親要素へアクセスする手段を提供する．なお，動的に生成される部分の親要素は，静的に得られる部分との結合結果を得る際や，動的に生成される部分が JavaScript で変更されたかどうか判定する際などに必要である．POGen のユーザはスケルトンテストコードを元にテストケースの記述を追加することで，テストコードの記述コストを削減する．例えば，図 7 に対して図 8 のようなスケルトンテストコードが生成される．生成した `AdditionPage` クラスは我々の `AbstractPage` クラスを継承していて，コンストラクタで `FindBy` アノテーションが付与されたフィールド変数が適切に初期化される．

```
public class AdditionPage extends AbstractPage {
  @FindBy(id = "_pogen_1")
  private WebElement _leftAndRight;
  @FindBy(id = "_pogen_2")
  private WebElement _answer;

  WebElement getLeftAndRightElement() {/*省略*/}
  WebElement getAnswerElement() {/*省略*/}
  String getLeftAndRightString() {
    return getLeftAndRightElement().getText();
  }
  String getAnswerString() {
    return getAnswerElement().getText();
  }
}
```

図 8 足し算の結果を表示するページに対して POGen が生成するスケルトンテストコード
Fig. 8 Generated skeleton test code by POGen for the addition page

典型的なテストケースでは，DOM ツリーの要素を取得して操作することで，テキストボックスへの入力やボタンの押下などを通してページの遷移を行い，検証したページにおいて動的に生成される部分に該当する

DOM ツリーの要素を用いて，その要素がテストケースのシナリオを実行した際に期待される状態，すなわち，期待されるテキストを持っているか，期待される属性値を持っているかなどを検証する．検証処理は JUnit などの既存のテスティングフレームワークが提供する Assert メソッドで記述できる．例えば，図 8 に対して図 9 のような JUnit テストケースを記述できる．

```
1 public class AdditionPageTest {
2   @Test
3   public void add1And2() {
4     WebDriver driver = new ChromeDriver();
5     AdditionPage page = new TopPage(driver)
6         .goToAdditionPage(1, 2);
7     assertEquals(page.getAnswerString(), "3");
8   }
9 }
```

図 9 POGen が生成するスケルトンテストコードを利用した JUnit テストケース

Fig. 9 A test case with the generated skeleton test code by POGen

図 10 で POGen による GB 結合テストの流れを示す．また，POGen による GB 結合テストは次の手順で実施する．

(1) テスト対象の Web アプリの HTML テンプレートを POGen に入力する．
(2) HTML テンプレートを解析して，修正 HTML テンプレートとスケルトンテストコードを生成する．
(3) スケルトンテストコードにテストケースを記述してテストコードを完成させる．
(4) 修正 HTML テンプレートを用いて Web アプリを配置する．
(5) 配置した Web アプリに対して，Selenium と記述したテストコードを用いてテストを実施する．

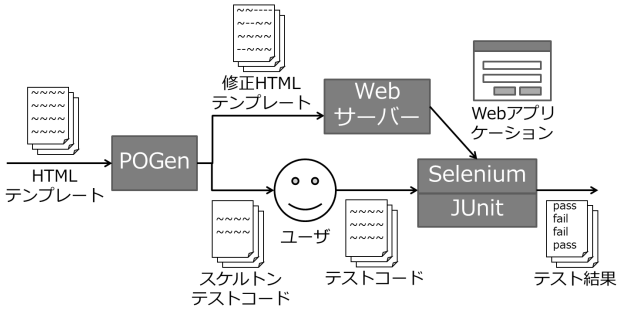

図 10 POGen による GB 結合テストの流れ

Fig. 10 The process of gray-box integration testing with POGen

POGen が生成するスケルトンテストコードのメソッドを利用することで，HTML 文章において動的に生成される部分を全て検証できる．つまり，POGen が生成した `WebElement` を戻り値とするメソッドを全てテストケースで使用した場合，動的に生成される部分については，十分にテストがされていると判断できる．そこで，「テストで実行した `POGen` が生成した `WebElement` を戻り値とするメソッドの数」 / 「`POGen` が生成した `WebElement` を戻り値とする全てのメソッドの数」をテンプレートカバレッジと定義する．なお，図 8 において，`getAnswerString()` メソッドが `getAnswerElement()` メソッドを呼び出すように，POGen が生成するメソッドは `WebElement` を戻り値とするメソッドか，もしくは，既に生成したメソッドを呼び出すメソッドのいずれかである．そのため，POGen が生成した `WebElement` を戻り値とするメソッド数を分母とすれば良い．したがって，テンプレートカバレッジをテスト十分性の指標とすることで，定量的なテスト十分性の指標が存在しないという問題 1 を解決する．

なお，ステートメントカバレッジとブランチカバレッジに包含関係がないように，ページ間の遷移を全て網羅したかどうかという指標とテンプレートカバレッジにも包含関係がない．テンプレートカバレッジの網羅が品質の高いテストであることの十分条件ではないが，必要条件である．

スケルトンテストコードを自動生成することで記述コストを削減して，さらに，PageObjects デザインパターンに基づいてアクセスメソッドをページ単位で生成することで，従来の Selenium を利用したテストコードよりも保守コストを低減する．これにより，記述コストおよび保守コストが大きいという問題 2 を解決する．

5. 評価

第 2.3 節の具体例に対して，提案手法とテンプレートカバレッジを利用することでバグを発見可能か議論する．

(1) **サーバープログラムと他のシステムもしくは複数のシステム間の結合におけるバグ：**テンプレートカバレッジを網羅する形でテストコードを記述すれば，加工した情報が正しいかどうかテストすることができ，バグを発見できる可能性が上昇する．
(2) **JavaScript によって HTML 文章を書き換える際に発生するバグ：**JavaScript で HTML 文章の DOM ツリーを変更する際，変更対象の要素に ID 属性が付加される可能性が高い．POGen ではテンプレート変数を出力する部分を直接のテキストで持つ要素と，もともと ID を持つ要素へのアクセスメソッドを提供するため，テンプレートカバレッジを網羅することで，HTML 文章が正常に書き換えられたかテストでき，バグを発見できる可能性が上昇する．
(3) **HTML 文章中の静的に得られる部分と動的に生成される部分の文字列結合におけるバグ：**テンプレートカバレッジを網羅する形でテストコードを記述すれば，動的に生成される部分を持つ要素に対してテストする

ため，その要素内での文字列結合に対してテストでき，バグを発見できる可能性が上昇する．また，当該要素の親要素の文字列も取得できるため，必要であればより広い範囲で文字列結合のテストが可能である．

(4) **サーバープログラムと HTML テンプレート間の結合におけるバグ：**テンプレートカバレッジを網羅する形でテストコードを記述すれば，テンプレートエンジンによって動的に生成される部分をテストできるため，サーバープログラムとのミスマッチを検出できる可能性が上昇する．

6. 関連研究

Staats らは Gourlay らのフレームワークを拡張することで，テスト対象のプログラム，テストコード，テストオラクルの関係について整理した [5]．また，彼らはテストの品質について議論する際に，テストカバレッジだけではなくテストオラクルについても言及すべきだと主張している．本論文で提案したテンプレートカバレッジは，単にテストを実行した際にプログラム中で検証した範囲を示すだけではなく，テンプレートエンジンによって動的に生成される部分を持つ要素が期待通りの状態であるか，つまり，動的に生成される部分を検証するようなテストオラクルを用いたかどうかを示している．こうした観点から，我々が提案するテンプレートカバレッジは，Staats らが主張するテストオラクルの品質を表す新しい指標と捉えられる．

Sprenkle らは Web アプリにおけるユーザの実行履歴を解析することで [3]，ページ間の確率的な遷移モデルを構築して，その上で，テストコードを自動生成する手法を提案した．彼らの手法はブランチカバレッジやパスカバレッジなどのプログラム構造に基づくテストカバレッジの考え方に似ていて，遷移モデル上でユーザがよく利用する経路を重点的にテストする手法である．一方で，我々の提案手法はテストオラクルの考え方に類似していて，Staats らが主張するように両者の観点からテストを設計することで品質の高いテストを実施できる．

小高らはアスペクト指向を利用することで JSP に出力する値をテストする手法を提案した [6]．彼らの手法では，JSP においてテンプレートから動的に生成される部分を抽出して，JSP ファイルのコンパイル結果に AspectJ でアスペクトをウィーブすることで，動的な部分だけの文字列を取得できるような特殊な HTML コメントを HTML 文章に埋め込むように修正する．さらに，埋め込んだ HTML コメントを介して動的な部分の文字列を取得して，期待値と一致するか比較すテスティングツールを提供する．しかし，彼らの手法では動的な部分の文字列が期待値と等しいかどうかだけしか検証できない．そのため，テスト可能な範囲が限られており，特に JavaScript のようなクライアントサイドで動作するプログラムの検証が難しい．一方，我々の提案手法では動的な部分のテキストを持つ DOM ツリー上の要素を取得することができ，その要素のテキストや属性値，兄弟や親子の要素について期待通りの状態かどうか検証できる．このことは，JavaScript を利用して DOM ツリーを動的に書き換えるような Web アプリにおいても，柔軟にテストできることを示す．

7. まとめと今後の展望

本論文では，従来の WB 単体テストと BB 結合テストでは発見できないバグについて議論して，Web アプリにおいて動的な部分に着目して上記のバグを発見可能な GB 結合テストを提案した．さらに，HTML テンプレートを解析することで，PageObjects デザインパターンに基づき，動的な部分に対するアクセスメソッドを備えたスケルトンテストコードを自動生成するツール POGen を開発した．その上で，POGen が生成したアクセスメソッドを用いて，テンプレートエンジンを利用する Web アプリに対する新しいテスト十分性の指標，テンプレートカバレッジを提案した．

テンプレートカバレッジは品質の高いテストの必要条件であり，結合テストのためのカバレッジをさらに考案する必要がある．ブランチカバレッジとコンディションカバレッジを組み合わせるように，テンプレートカバレッジに他のカバレッジを組み合わせることで，品質の高いテストの十分条件と成るようなカバレッジを提案する予定である．

参考文献

[1] Sakamoto, Kazunori and Kaizu, Tomohiro：PageObjectGenerator - Google Project Hosting, 入手先 ⟨http://code.google.com/p/pageobjectgenerator/⟩ (2012.04.16).

[2] Simon Stewart：PageObjects - Selenium - Google Project Hosting, 入手先 ⟨http://code.google.com/p/selenium/wiki/PageObjects⟩ (2012.04.16).

[3] Sprenkle, Sara and Pollock, Lori and Simko, Lucy: A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications, *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, pp. 230–239 (2011).

[4] 坂本一憲, 東海政治, 村上裕子, 宮原里枝, 奥村有紀子, 秋山浩一, 鷲崎弘宜, 深澤良彰: Web アプリケーション開発における画面仕様書およびテスト仕様書の自動生成手法と開発プロセス改善の提案, ソフトウェア品質シンポジウム 2011, (2011).

[5] Staats, Matt and Whalen, Michael W. and Heimdahl, Mats P.E.: Programs, tests, and oracles: the foundations of testing revisited, *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pp. 391–400 (2011).

[6] 小高敏裕, 上原忠弘, 片山朝子, 山本里枝子: アスペクト指向を利用した Web アプリケーションテストの自動化, *IPSJ SIG Notes 2007(33)*, pp. 97–104 (2007).