モック・オブジェクト・アプローチによるWebアプリケーション単体テストの生産性向上

水島 壮太

Productivity Improvement of Unit Test for Web Application by Mock Object Approach

Sota Mizushima

本論文では，MVC（Model-View-Controller）フレームワーク上のJ2EE™・Webアプリケーションにおけるモック・オブジェクト・アプローチによる単体テストを支援するツールを用いてテスト生産性を向上させる手法について論じる．今後，グローバル・リソースを利用した開発の機会が増えるに従って，単体テストの効率化，テスト・ケース保守やエビデンスの管理は品質確保の観点からさらに重要になってくる．しかし，Webコンテナや複雑なリクエスト・オブジェクトに依存するWebアプリケーションのプログラムを保守可能な形で単体テストすることは難しい．本論文ではWebアプリケーション・フレームワークであるWACs（Web Application Components）のテスト支援ツールを拡張し，画面項目定義情報との連携によるテスト・ケース作成工数の削減，モック・オブジェクト・アプローチによるコンテナ非依存なJSP™（Java Server Page）単体テストの実現，テスト駆動開発によるコントローラーおよびJSP単体テストの効率化を提案，その有用性を検証する．

This paper discusses techniques for improving test productivity using a tool that supports unit testing using Mock Object Approach on the MVC (Model-View-Controller) framework of the J2EE Web application. As Web application development using global resources increases in the future, productivity improvement and the maintenance of the test case and its evidence in the unit test will become more important from the viewpoint of quality control. However, it is difficult to carry out the unit test for Web application programs, which depend on the Web Container and complex request objects, in a preservable format. This paper proposes three techniques and verifies theirs utility that work together with the test supporting tool of WACs (Web Application Components), the Web application framework: the reduction in test case creation man-hours through linkage with the page item definition meta-data; the realization of JSP (Java Server Page) unit test non-dependent on the Web Container through Mock Object Approach; and the efficiency improvement of Controller and JSP unit tests through Test-Driven Development.

Key Words & Phrases : 単体テスト, モック･オブジェクト･アプローチ, テスト駆動開発, WACs, Struts
Unit Test, Mock Object Approach, Test-Driven Development, WACs, Struts

1. はじめに

J2EE・Webアプリケーションの単体テストは，テスト対象のプログラムがJ2EE準拠のWebコンテナに依存しているために，次のいずれかのアプローチで実施されることが多い［1］.

① JUnit［2］とWebコンテナのモック･オブジェクトを使ったモック・オブジェクト・アプローチ（以下，MOアプローチ）
② 実際にWebコンテナ上で稼働させるイン･コンテナ･アプローチ（以下，ICアプローチ）

MOアプローチの単体テストは，実際のWebコンテナ上での動作確認ができないため開発者の不安を感じさせる部分があるが，ICアプローチと比較して下記のようなメリットがある.

- テストが高速で回帰テストが容易
- テスト・カバレッジの測定が容易
- テスト・ケースとエビデンスの保守が容易

一方のICアプローチにおいては，Cactus［3］等のテスト・フレームワークを利用すればJUnitからテストを起動できるため，テスト・ケースの保守を容易にすることはできる［4］. また，ブラウザーからの打鍵テストの場合は，IBMソフトウェア製品のRational Functional Tester［5］やオープン・ソース・ソフトウェアのSelenium［6］等を利用することでテスト・ケースを保守，回帰テストを自動化することが可能であるが，ホワイト･ボックス･テストを基本とする単体テストの支援ツールとして多用するものではない．また，統合テスト･ケースと同等の網羅性になるため，単体テストで網羅すべき分岐や条件が十分にテストされない結果となる.

今後グローバル・リソースを利用した開発の機会が増えるに従って，テスト・ケース保守やエビデンスの管理，テスト・カバレッジの測定は品質確保の観点からさらに重要になってくるため，MOアプローチの単体テストを実施する必要がある．しかし，実際の開発プロジェクトにおいてWebアプリケーションのプログラムに対するMOアプローチの単体テストが十分に実施されるケースは少なく，各開発者のPC上にアプリケーション･サーバーを起動し，ブラウザーからの打鍵による事前統合テスト的な単体テストのみを行うことがほとんどである.

そこで，本論文では，Webアプリケーションに対してMOアプローチの単体テストが実施されない原因を整理し，IBMのグローバルIPアセット（ライセンシング可能なIBMの知的財産）であるWACs（Web Application Components）のテスト支援

提出日：2007年9月3日　再提出日：2008年2月25日

ツールを実際に拡張することで，MO アプローチの単体テストを実践しやすい環境を整える手法について論じる．AspectJ を利用して MO アプローチの単体テストを実践する手法や［7］，MO アプローチの単体テストが容易な POJO（Plain Old Java Object）に対するテスト駆動開発［8］に関する文献はいくつか存在するが，本論文ではこれらの MO アプローチによる単体テストの手法が実際の Web アプリケーション開発プロジェクトにて実施されにくいという問題を解決することを目的とする．

以下，2 章で Web アプリケーションに対する MO アプローチの単体テストの課題を整理し，その解決のための 3 つの方法を 3 章から 5 章で紹介する．さらには，6 章でそれらの解決策を統合した単体テスト・ツールを紹介し，7 章でその有用性を検証した結果を報告する．

2. 従来のアプローチとその課題

WACs や Struts 等に代表される MVC（Model-View-Controller）モデル（**図 1**）の Web アプリケーション・フレームワーク上で稼働するプログラムは，Web コンテナだけでなくそのベース・フレームワークにも強く依存しているため，MO アプローチによる単体テストを実施するためにはフレームワーク専用のテスト・ドライバーが必要である．

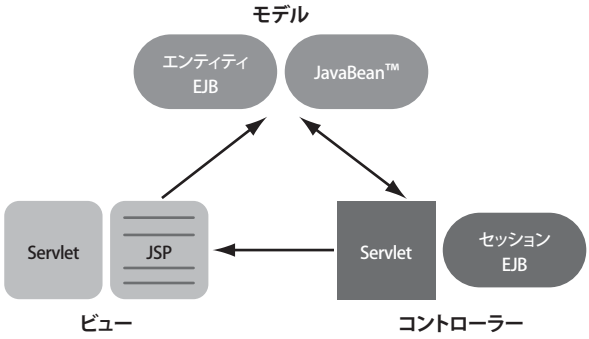

図 1．J2EE アプリケーションにおける MVC モデル
引用元：Sun Developer Connection - Technical Articles
http://sdc.sun.co.jp/java/technicalArticles/spguide/chap3.html

WACs においては，MVC モデル上の表示機能をつかさどる JSP（Java Server Page）の単体テストはできないものの,ユーザーからのリクエストを処理するコントローラーである Reaction クラスに関しては専用の単体テスト・ドライバーが提供されており，Web コンテナに非依存な MO アプローチによる単体テストを実施することが可能である．また，Struts アプリケーションの場合は，StrutsTestCase［9］や S2Struts［10］などのオープン・ソース・ソフトウェアを利用することで MO アプローチによるコントローラーの単体テストが可能だ．

これらのツールが豊富に用意されているにもかかわらず，MVC フレームワークの Web アプリケーションに対して MO アプローチによる単体テストが十分に実施されないのは，主に下記のような課題があるためである．

（1）テスト・ケース作成工数の増大

コントローラーは，リクエスト・オブジェクトやセッション・オブジェクトの入出力処理を主につかさどっているため，1 ケースあたりのテスト・データとその想定結果のデータ定義量が多くなる傾向がある．特にリクエスト・オブジェクトは，実際の Web 画面から送信されるデータを含むため，画面項目数が膨大な画面から呼び出されるコントローラーのテスト・ケースを定義するためには気が遠くなるような単純定義作業を行わなければならない．また，複雑なデータ構造を表現できるように XML などの構造化文書でテスト・データを作成できるようにしたとしても開発者を XML 地獄へと突き落とすだけである．そのため，単体テスト・ドライバーがプロジェクトの基盤機能として提供されていてもテスト・ケースを作る余裕がなく，スケジュールと開発コストを優先させて結局使われないということがしばしばある．

（2）JSP 単体テスト支援ツールの欠如

ビューである JSP を単体で動作させるためには，Web コンテナ上でサーブレットに変換，およびコンパイルを行うことが前提となる．そのため，Web コンテナを起動させて，ブラウザーからの打鍵による JSP 表示テストを実施する必要に迫られてしまい，コントローラーと統合させてテストしたほうが開発者にとっては一石二鳥であるため，JSP 単体での本来の単体テストが実施されない結果となる．

また，MVC フレームワークを利用している場合は JSP の中にフレームワークが提供するタグ・ライブラリーが多く存在し，また，コントローラーが出力したリクエスト・オブジェクトは内部に独自のデータ構造を作成するため，リクエスト・オブジェクトのモック・オブジェクトを用意する必要がある．しかし，現在 WACs や Struts 等においてこのような JSP 単体テストを支援するドライバーは提供されていないため，JSP を単体でテストすることが難しい．

本論文ではこれらの課題に対し次の 3 つの解決策を提案する．

- 画面項目定義情報との連携によるテスト・ケース作成工数の削減
- MO アプローチによるコンテナ非依存な JSP 単体テストの実現
- テスト駆動開発によるコントローラーおよび JSP 単体テストの効率化

画面項目情報を利用してテスト・ケースの雛形を作成しテスト・ケース工数を削減する試みや，MVC フレームワーク上の JSP を単体でテストする機構は未開拓の領域である．

3. 画面項目定義情報との連携によるテスト・ケース作成工数の削減

3.1 WACs Generator の画面項目定義との連携

WACs Generator とは，WACs の開発支援ツールの一つで，プレゼンテーション・ロジックの品質向上を目的としたコード生成ツールである．WACs Generator は開発過程で入力される各画面の画面項目定義情報を XML 形式のファイルで保持し

ているため，単体テスト・ケース作成時にこの情報を読み込むことで画面からの入力データ，および出力画面における想定結果データの項目名を取得することが可能である．現状のWACsに含まれるReaction単体テスト・ツールの場合，Excel形式の単体テスト仕様書に入力データと想定されるデータのフィールド名およびその値を手作業で入力しなければならないが，画面項目定義と連携し，その手間を省くだけでもテスト･ケース作成工数は大幅に削減できる．

Strutsの場合でも，ActionFormの項目定義を生成・保持するIPアセットのT-Fog（Struts版のWACs Generator）と連携することで，同様にテスト・ケースの雛形を作成することが可能である．

3.2 正常値の自動生成

WACs GeneratorのForm項目定義を読み込む際に，その項目の属性（HTMLのタグ種別，文字種制限，文字長，数値範囲，文字パターン等）を読み込むことでその項目の正常値をランダムに生成することが可能である．単体テスト・ケースにおいて，特に重要な意味をもたない入力データ等がある場合は，ランダムに生成された正常値を自動生成してくれるだけでも工数削減効果は得られる．開発者は，生成された正常値を任意に変更しながら，条件や分岐を網羅できるテスト・ケースを容易に生成できる．

3.3 テスト・ケース仕様書の自動生成

ツール上で保持しているテスト・ケースのデータおよびそのテスト結果をまとめて仕様書として出力することで，単体テストの仕様書作成工数の削減を図ることができる．JUnitを利用したMOアプローチの単体テストの場合は，IDE（統合開発環境）上で，テスト・ケースを作成しテストを実行するため，IDE上で作成したテスト・ケース情報をマスターとし，テスト結果情報と併せて仕様書を自動生成するほうが変更に対してより柔軟でテスト効率が向上すると考えられる．

4. MOアプローチによるコンテナ非依存なJSP単体テストの実現

4.1 JSPコンパイラーの利用

Webコンテナに依存せずにJSPファイルを実行可能なサーブレット・クラスに変換するためには，WebSphere Application ServerやTomcat等のJ2EE準拠のアプリケーション・サーバーに含まれるJspCというAntタスクを利用することができる．テスト実行前にJSPのコンパイルを事前に実施することで，生成されたJava ™クラスに対して単体テストを実施することが可能になる．

4.2 必要になる汎用モック・オブジェクト

JSPをコンパイルすることによって生成されたJavaクラスは，下記のインターフェースの実装オブジェクトに依存しているため，これらのモック・オブジェクトを作成する必要がある．

- javax.servlet.http.HttpServletRequest
- javax.servlet.http.HttpServletResponse
- javax.servlet.http.HttpSession
- javax.servlet.jsp.JspFactory
- javax.servlet.jsp.PageContext
- javax.servlet.jsp.JspWriter

また，MVCフレームワークを利用している場合，JSPはそのフレームワークが規定するコントローラーによって作成されるリクエスト・オブジェクトに依存している．そのため，JSPの単体テストを行うには，リクエスト・オブジェクトの中身をテスト・ケースとして定義し，その定義情報からコントローラーの基盤APIによって規定されたデータ構造を持つjavax.servlet.http.HttpServletRequestのモック・オブジェクトを作成するスキームが必要となる．その作成されたモック・オブジェクトをテスト対象のJSPをコンパイルしたクラスに引き渡すことで，MVCフレームワークが提供するタグ・ライブラリーの処理を正常に行うことができる．

4.3 テスト結果の検証方法

JSPの主な処理はHTMLテキストの文字列をJspWriterオブジェクトに出力する処理であるため，結果をJUnitのAssertionで文字列比較をする検証は効率的ではない．非常に長いテキスト・データであるHTMLの出力結果を検証することにすると，HTMLの想定出力結果を作成する工数が増えるだけでなく，HTMLのデザインが変わるたびにテスト・ケースの期待値を大幅に変更することになり，テスト・ケースの保守にも多大な工数がかかることになる．そのため，JSPの単体テストの結果は，実際のHTMLファイルとして出力し，開発者が設計する検証ポイントを目視で確認する方法が望ましい．また，HTMLファイルとして単体テスト結果を保守しておけば，前回のHTML出力結果との差分をとることによって回帰テストの検証が効率化できるメリットがある．

JSPは，javax.servlet.http.HttpServletResponseのWriterオブジェクトを利用してHTMLを出力するため，そのモック・オブジェクトにFileWriterを設定することにより，JSPが出力するHTMLテキストをHTMLファイルとして保存することが可能である．

5. テスト駆動開発によるコントローラーおよびJSP単体テストの効率化

5.1 コントローラーの単体テスト・ケースの再利用

機械猫モッカー［11］のように，モック・オブジェクトを作成する場合にモック・オブジェクトの振る舞いをそのオブジェクトのテスト・ケースにて定義・生成する手法がある．この手法を今回のJSP単体テストに当てはめて考えると，コントローラーのテスト・ケースを利用してJSPの動作に必要なリクエスト・オブジェクトのモック・オブジェクトを定義・作成することが可能となる．テスト駆動の考え方を応用し，コントローラーの開発に先行してコントローラーのテスト・ケースを作成することで，コントローラーの開発だけでなく，JSPの開発および単体テストに再利用することが可能となる．

5.2 テスト・ケースの作成手順

前述の通りMVCフレームワークを利用している場合，JSPはコントローラーによって作成されたフレームワークが規定のリクエスト･オブジェクトに依存している．そのため，ボトム･アップ･アプローチで単体テストを行う場合には，依存関係上の下位モジュールであるコントローラーのテストを先行して行う必要がある．また，トップ・ダウン・アプローチで単体テストを行う場合は，依存関係上の下位モジュールであるコントローラーのモックが必要になるため，5.1で述べたようにコントローラーのテスト・ケースを先行して作成する必要がある．故に，ボトム・アップ，トップ・ダウンどちらのアプローチでもまずはコントローラーのテスト・ケースを作成することが最優先であると言える．

以上を踏まえた上で，コントローラーとJSPのテスト駆動開発の流れは，**図2**のように整理できる．

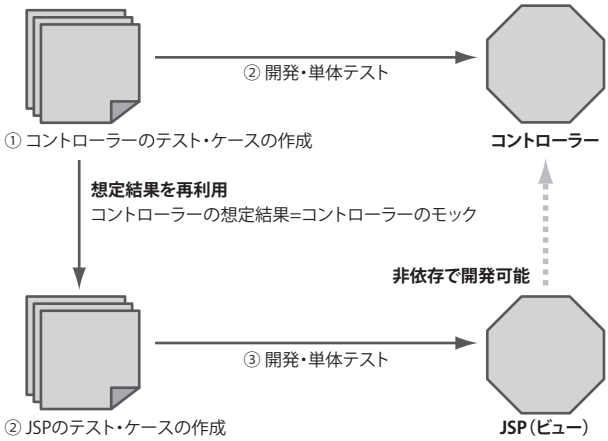

図2. コントローラーのテスト駆動開発の流れ

6. WACsテスト支援ツールの拡張

本章では，3章から5章の解決策を導入したWACsのテスト支援ツールの拡張内容について説明する．

6.1 Reaction単体テスト生成ツールの拡張

3章の解決策を導入したWACsのReaction単体テスト生成ツールの概要を**図3**に示す．

拡張したのは，下記の機能である．

- 画面項目定義からフィールド名の転記機能
- 画面項目定義から正常値の自動生成機能
- テスト・ケース定義情報の内部XML化
- 仕様書エクスポート機能

今までのWACs単体テスト生成ツールは，Excel形式の単体テスト仕様書に入力されたテスト・データをもとにJUnitのテスト・ケース・クラスとテスト・データ・クラスを生成していたため，項目数が多ければ多いほどその仕様書の作成工数が肥大化する問題を抱えていた．上記機能を拡張後は，IDEであるEclipse上で画面項目定義の転記機能や正常値の自動生成機能を利用してテスト・ケースを作成可能であり，最後に仕様書へ出力することでテスト・ケースの作成工数を大幅に削減できる．

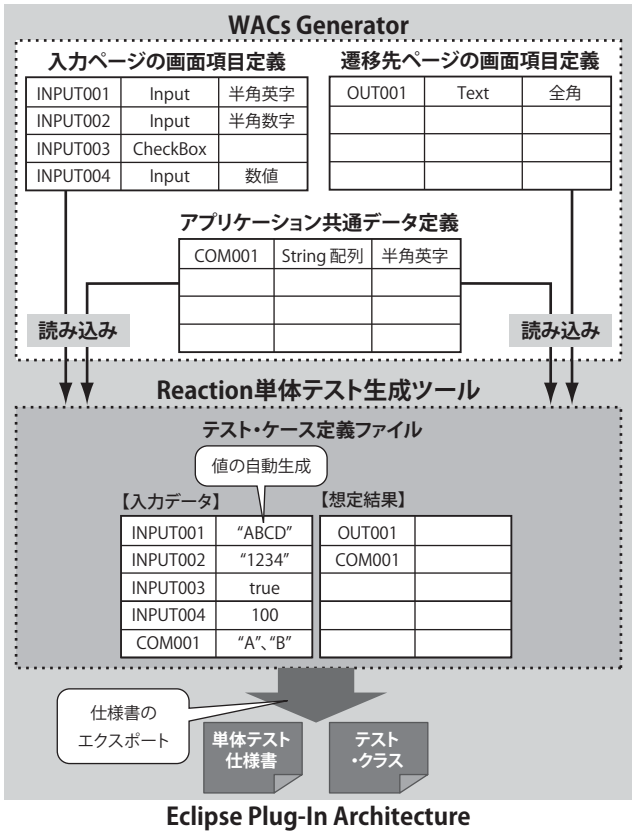

図3. Reaction単体テスト生成ツールの拡張の概要

6.2 JSP単体テスト・ツールの開発

4章にて紹介したWebコンテナに非依存なJSPの単体テスト･ツールの概要を**図4**に示す．

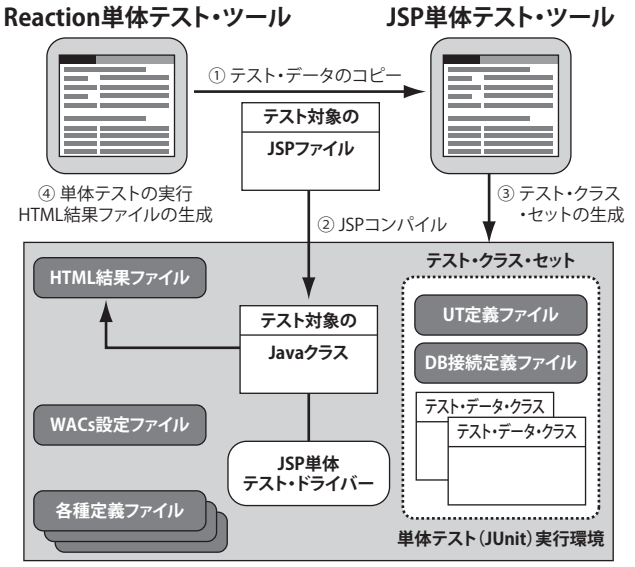

図4. JSP単体テスト・ツールの概要
引用元：WACsガイド『Reaction単体テスト・ツール使用ガイド』

JSP単体テスト・ツールを利用することで以下の手順でJSPの単体テストを実施することができる．

① Reaction単体テストの想定結果のデータをコピー

② JSP ファイルのコンパイル
③ テスト・クラス・セットの生成
④ JUnit から JSP 単体テストの実行

以上で，指定されたディレクトリーに JSP の処理結果である HTML ファイルが保存される仕組みである.

6.2.1 JSP 単体テスト生成ツール

JSP 単体テスト生成ツールは前項の①から③までの手順を支援する以下の機能を持つ.

- Reaction 単体テスト・ツールの想定結果を再利用し，JSP のテスト・データを作成する
- JSP のコンパイル・タスクを起動する
- テスト・クラス・セットとしてテスト・データ・クラスとテスト・ケース・クラスを生成する

これら機能により JSP の単体テストを実施するために必要なテスト・コンポーネントを自動生成することが可能になる.

6.2.2 JSP 単体テスト・ドライバー

次に実際にリクエスト・オブジェクトのモックを作成し，JSP の処理を実行するドライバーが必要になる. Reaction 単体テストのテスト・ドライバーにて，WACs が規定するリクエスト・オブジェクトのモック・オブジェクトを作成する機能があるためにモックの作成機能は流用可能である. 下記は，現状の Reaction 単体テスト・ドライバーが提供しているモック・オブジェクトのインターフェースである.

- javax.servlet.http.HttpServletRequest
- javax.servlet.http.HttpServletResponse
- javax.servlet.http.HttpSession

さらに，JSP の単体テストを実施するためには，JSP が依存している下記クラスの汎用モック・オブジェクトが必要となるため追加開発を行った.

- javax.servlet.jsp.JspFactory
- javax.servlet.jsp.PageContext
- javax.servlet.jsp.JspWriter

また，WACs の現状の Reaction の単体テストの場合，Reaction の処理に入る前の入力データを javax.servlet.http.HttpServletRequest のインスタンスとして作成することが可能なので，同様に JSP の処理に入る前のリクエスト・オブジェクトを javax.servlet.http.HttpServletRequest のインスタンスとして作成する JSPUTRunner クラスを新規に開発した.

JSPUTRunner クラスは，WACs の CPA（Component Plug-in Architecture）の初期化処理を行った後，テスト・データ・クラスの処理を元にリクエスト・オブジェクトを作成し，JSPUTForwarder クラスを呼び出す. JSPUTForwarder クラスは指定された JSP のコンパイル済みクラスを呼び出し，HTML ファイルに出力する PostProcesser の実現として実装した.

以上で，WACs に依存した JSP の単体テストを実施可能にするテスト・ドライバーが完成する. まとめの意味も含め，JSP 単体テスト・ドライバーの主要クラスのハイレベルなクラス図を**図 5** に示す.

JSP 単体テスト生成ツールで生成された JSP テスト・ケース・クラスが，この JSPUTRunner の execute メソッドを実行することによってテスト・データ・クラスの情報からリクエスト・オブジェクトのモックを作成，JSP 単体テストを実施し，その結果をテスト・ケース毎に HTML ファイルとして保存することが可能となる.

7. 検証

本論文では拡張した WACs テスト支援ツールの有用性を検証するために，次の 2 つの検証を行った.

7.1 Reaction 単体テスト・ケースの作成工数の削減効果の測定

画面項目数の数に応じた Reaction 単体テスト・ケースの作成工数の削減効果を実際に測定する目的で，下記の検証実験を行った.

【実験条件】

① 入力となる画面項目数が 10，20，30，50 項目の Reaction クラスを用意し，正常ケースのテスト・ケースを 1 ケース作成する
② 入力となる画面項目は複数の HTML 種別が混在しているものにする（テキスト・フィールドやチェック・ボックス，ラジオ・ボタン等）
③ 画面項目は，単項目と一覧項目を混在させ，一覧項目に関しては 3 件程度を目安にテスト・ケースを作成する

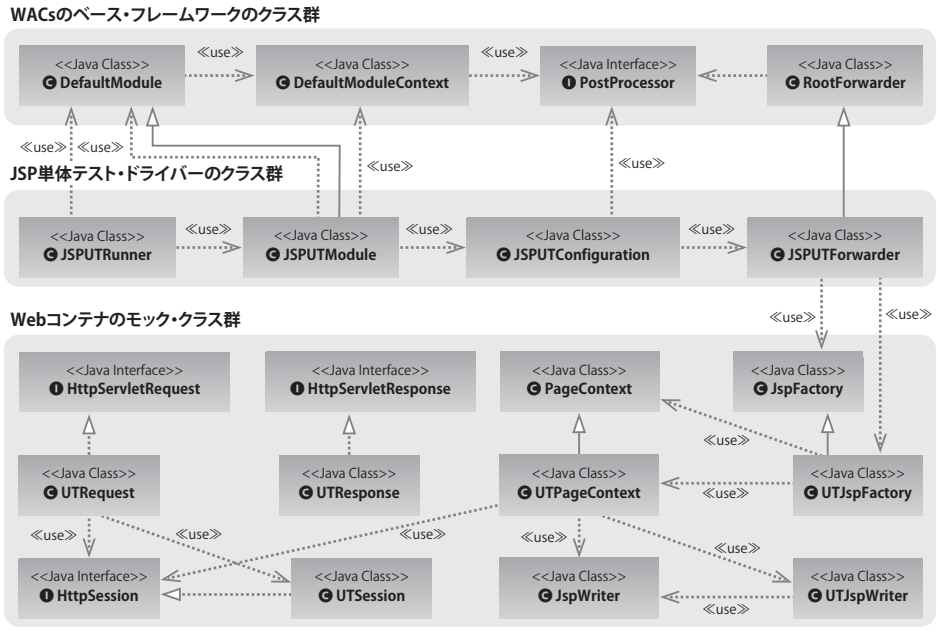

図 5. JSP 単体テスト・ドライバーのクラス図

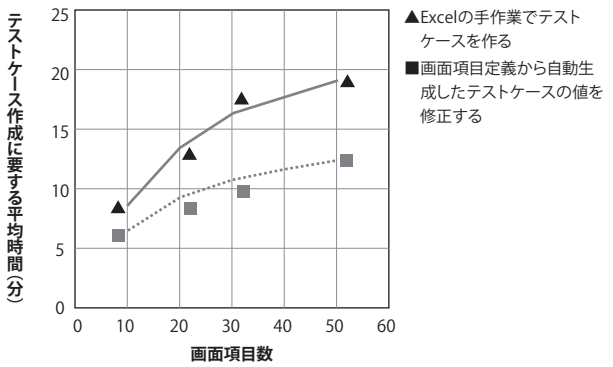

図 6. テスト・ケース作成工数削減効果

④ テスト対象の Reaction は入力された画面項目を単純に if 文で比較・分岐するもので，画面項目の中には分岐に関係のないものを 1 割程度含める
⑤ 従来の Excel 入力によるテスト・ケース作成時間と画面項目定義情報を利用した単体テスト支援ツールによるテスト・ケース作成時間をそれぞれ測定する
⑥ 測定時間は，テスト・ケースの作成を開始してからあらかじめ用意した正解のテスト・ケースを作成し終わるまでの時間を測定する
⑦ 5 名程度の開発者でそれぞれ測定し、その平均値を算出する

上記条件で実験を実施して、**図 6** の結果が得られた.

単体テスト・ケース作成工数を画面項目が 30 項目程度の場合に最大で約 40% 程度削減できることが検証できた. 無論，画面数，画面項目数が増えるにつれて全体のテスト・ケース作成工数の削減効果は大きくなる.

7.2 JSP 単体テスト・ツールの動作検証

JSP 単体テスト・ツールが動作することを確認するために，WACs が提供するサンプル･アプリケーションの 27 個の JSP ファイルに対する単体テストを試験的に実施したところ，正常に HTML ファイルを出力することに成功した. 故に，Web コンテナに非依存な JSP 単体テストを実現できることを確認した.

8. おわりに

本論文では，MVC フレームワーク上で開発された Web アプリケーションの MO アプローチによる単体テストが抱える問題を解決するため，WACs の Reaction 単体テスト・ツールの拡張を提案した.

ツールを実際に拡張し検証を行った結果，コントローラーの単体テスト・ケースの 1 ケース当たりのテスト・ケース作成工数を最大で約 40% 削減できるとともに，コントローラーのテスト･ケースを再利用して MVC フレームワーク上の JSP に対し Web コンテナに非依存な単体テストを実現できることを確認した.

この結果から，MVC フレームワークの Web アプリケーションに対する単体テストにおいて，IC アプローチのみでなく，MO アプローチによる単体テストをより実施しやすくする環境が開発現場に提供可能であることが示された.

謝辞
ご多忙の中 WACs テスト支援ツールの開発にご助力くださった WACs チームの皆様，この場を借りて感謝の意を申し上げます.

参考文献

［1］ Mock Objects vs In-Container testing:
http://jakarta.apache.org/cactus/mock_vs_cactus.html
［2］ JUnit:
http://junit.sourceforge.net/
［3］ Cactus:
http://jakarta.apache.org/cactus/index.html
［4］ Vincent Massol, Ted Husted:JUnit in Action,Manning Publications, ISBN:1-930110-99-5（2003）.
［5］ Rational Functional Tester:
http://www.ibm.com/jp/software/rational/products/test/rft/
［6］ Selenium:
http://selenium.openqa.org/
［7］ Nicholas Lesiecki：“AspectJおよび疑似オブジェクトによる柔軟なテスト,” developerWorks（2002.5.1）.
http://www.ibm.com/developerworks/jp/java/library/j-aspectj2/
［8］ 大月美佳：ソフトウェアテストの最新動向：6.Test-Driven Development（テスト駆動開発）開発手法としてのテスト，情報処理，Vol.49 No.2, pp.162-167（2008）.
［9］ StrutsTestCase:
http://strutstestcase.sourceforge.net/
［10］ S2Struts:
http://s2struts.seasar.org/ja/
［11］ 機械猫モッカー:
http://kikainekomocker.sandbox.seasar.org/

日本アイ・ビー・エム株式会社
GBS ソリューション&アセット
IT スペシャリスト
水島 壮太　Sota Mizushima

［プロフィール］
2005年日本IBM入社.入社後，Web基盤構築アセット群「Web Value Pack」の設計／開発を担当. 現在は金融業界のプロジェクトを中心にJava関連の技術支援，および「Web Value Pack」のAjax対応や提案サポートに従事している.
stmz@jp.ibm.com