Vol.2012-SE-175 No.15
2012/3/16

Webアプリケーションを対象としたテストスイートの評価 ～ミューテーションテストにもとづいた手法～

深谷雅洋[†1]　Rogene Lacanienta [†1]
高田眞吾[†1]
丹野治門[†2]　張暁晶[†2]　星野隆[†2]

Webアプリケーションがより多く開発されるようになったため，Webアプリケーション向けのテスト手法の重要性が高まっている．今までに提案されてきたテスト手法の評価は，挿入されたバグを発見する実験などを通して行っている．しかし，バグの挿入は網羅的に行っているわけではなく，人が手動で挿入するか，過去に存在したバグを再度挿入するといった方法をとっている．そこで，本稿では，ミューテーションテストが網羅的にバグを挿入しようとする点に着目し，Webアプリケーションを対象に生成したテストスイートを評価するフレームワークについて述べる．実装したフレームワークをサンプルアプリケーションに適用した実験結果および考察を述べる．

Evaluating test suites for Web applications – A mutation test based approach –

Masahiro Fukaya ,[†1] Rogene Lacanienta ,[†1]
Shingo Takada ,[†1] Haruto Tanno ,[†2]
Xiaojing Zhang [†2] and Takashi Hoshino[†2]

Web application testing has become increasingly important with Web applications becoming more prevalent. Many proposed test tools have been evaluated through experiments using programs that have been seeded with bugs. However, such bugs are seeded manually or seeded using past bug information, and as a result they are not seeded exhaustively. In this paper, we focus on mutation testing where bugs are seeded exhaustively. We describe a framework for evaluating test suites that have been generated for Web applications. We conducted an experiment where we applied the implemented framework to a sample application. We discuss the experimental results.

1. はじめに

近年，Webアプリケーションが増えている．その理由の一つとして，Webアプリケーションがどのコンピュータにもインストールされている Web ブラウザを使用するため，ユーザが新たにインストールしなくても利用できる点があげられる．その気軽さがゆえに今後も増え続けるであろうが，これはその信頼性を確保するためのテスト手法の重要性が増すことも意味する．2003 年時には 72.5%ものショッピング Web サイトにおいて何らかの故障を起こしており，当時のテストツールが不十分であることを示唆している，と Kam らは述べている[1)]．そこで，Webアプリケーションのために様々な観点から様々なテスト支援手法やツールが提案されてきている[2)3)4)5)6)7)]．

各手法に対する評価方法として，以下の種類がある．

- ソースコードや設計モデルに対するカバレッジ
- 手作業を含む他の手法・ツールを用いて作成したテストスイートとの比較
- 手作業と比較した場合の工数削減効果
- バグ発見能力

このうち，バグ発見能力が最も直接的に効果を測定できる評価方法として考えられる．しかし，既存の評価では，挿入されるバグの網羅性に疑問が残る．ほとんどの場合，次の二つの方法によりバグを挿入している．

- 開発者が犯すであろうと思いついたバグを挿入する．
- アプリケーションの過去のバグの履歴が残っている場合，そのバグを再び挿入する．

つまり，バグを系統的に網羅的に挿入しているわけではなく，挿入されたバグに対しては有効であることが分かっても，異なるバグの場合の有効性が不明である．

バグを系統的に網羅的に挿入しようとする方法としてミューテーションテストがある．ミューテーションテストは結合効果（coupling effect）と呼ばれる基本仮説に基づいている．結合効果によると，複雑なバグは単純なバグの組み合わせであり，すべての単純なバグを見つけることができれば，ほとんどの複雑なバグも見つけられる．そこで，ミューテーショ

†1 慶應義塾大学
Keio University
†2 NTT サイバースペース研究所
NTT Cyber Space Laboratories

ンテストでは，アプリケーションに対して，様々な単純なバグを個別に挿入することにより，ミュータントと呼ばれるバージョンを複数生成する．つまり，各ミュータントは，元のコードと比較して一か所のみ異なるという単純なものである．テストスイートがより多くのミュータントを検出できれば，より強力なテストスイートであることとなる．

本稿では，テスト手法やテストツールの有効性を正確に評価するために，Web アプリケーションを対象に生成したテストスイートを，ミューテーションテストにより評価することを提案する．評価するための汎用フレームワークを提案し，その実装を適用した結果を報告する．

以下，2 節では，ミューテーションテストの基本を述べる．3 節では，テストスイート評価フレームワークを提案し，その実装を述べる．4 節では，実験について述べ，結果について考察する．5 節では，本稿をまとめる．

2. ミューテーションテスト

ミューテーションテストは，元々1970 年代初頭に提案され，テストスイートの評価やテストスイートをよりよくするために使用される[8]．ミューテーションテストではバグを挿入するが，それはプログラマが実際に行う間違いを表していることに基づいている．

ミューテーションテストの基本手順は，主に二つのステップからなる．まず，ミュータントを生成する．ミュータントは元のプログラムに対して一か所変更を行ったことによりできるプログラムのバージョンである．つまり，元のプログラムに対して，バグを一つ挿入すると，一つのミュータントが生成される．この変更を行うために，ミュータントオペレータと呼ばれる変換ルールを利用する．ミュータントオペレータをプログラムに適用することにより，ミュータントを生成する．ミュータントオペレータについては様々なプログラミング言語に対して提案されている．例えば，FORTRAN 77 用のミューテーションツールである Mothra[9] は 22 個のオペレータを提供しており，本研究でも用いる Java 用のミューテーションツールである MuClipse[10] は 43 個のオペレータを提供している．ミュータントオペレータの例を表 1 にいくつか示す．

二つ目のステップでは，テストスイート中の各テストケースをオリジナルプログラムとミュータントの双方に対して実行する．テストスイート中のいずれかのテストケースにより，オリジナルプログラムとミュータントの振る舞いの違いを観測できた場合，そのテストスイートはミュータントを kill したと呼ぶ．kill できなかったミュータントに対しては，kill できるようなテストケースを作成し，テストスイートを補強することができる．

表 1 ミュータントオペレータの例
Table 1 Example of Mutant Operators.

オペレータの種類	元のコード例	オペレータ適用後の例
変数の変換	x	x+1, x-1, 0
算術演算子の変換	+	-, *, /
メソッド（関数）呼び出しの削除	y = f(x)	削除
条件の否定	if (a)	if (not a)

```
// Original          // Mutant
...                  ...
int x = y;           int x = y;
if (z < y) // (*)    if (z < x) // (*)
  ...                  ...
```

図 1 等価ミュータントの例
Fig. 1 Equivalent Mutant Example

なお，ミュータントによっては，kill できるようなテストケースが存在しない場合がある．例えば，図 1 のように，オリジナルコードの (*) 行の変数 y を x に変更することによりミュータントを生成した場合，どのようなテストケースを作成しても違いを見つけることができない．このようなミュータントを等価ミュータント（equivalent mutant）と呼ぶ．

最終的に，以下の式を用いて，ミューテーションスコアを算出する．

$$Mutation\_Score = \frac{Number\_of\_Killed\_Mutants}{Number\_of\_NonEquivalent\_Mutants} \quad (1)$$

ミューテーションスコアが 1.0 となった場合，テストスイートはミュータントで示されるバグを検出するのに十分であることを示す．また，1.0 でない場合でも，上で述べたように，kill できなかったミュータントも検出できるようなテストケースを作成することによって，よりよいテストスイートを得ることができる．

3. ミューテーションテストに基づいたテストスイート評価フレームワーク

本研究では，生成されるテストスイートの評価を通して，テスト手法やテストツールを評価する．テストスイートの評価については，ミューテーションテストにより行う．本節では，まずミューテーションテストを行うための汎用フレームワークを提案する．その次に，

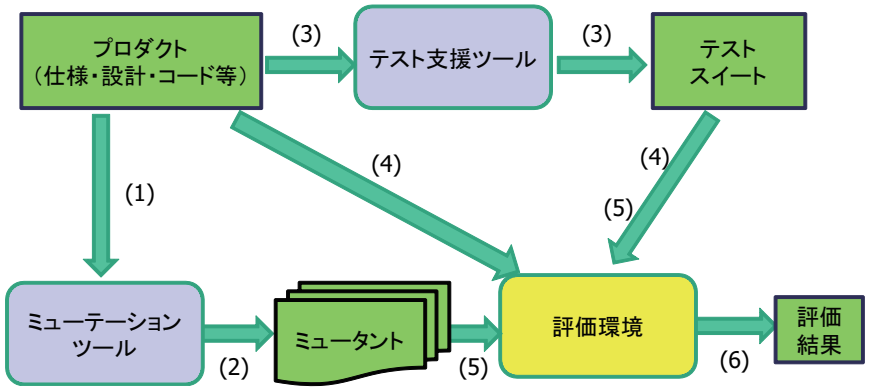

図 2 汎用フレームワーク
Fig. 2 Generic framework.

フレームワークの実装について述べる．

3.1 汎用フレームワーク

ミューテーションテストを通して評価を行うための汎用フレームワークを図 2 に示す．

- プロダクト，テストスイート，ミュータント，評価結果は，各ツールや環境への入力または出力である．
- テスト支援ツールおよびミューテーションツールは，評価時に使用したいツールを選べばよい．テスト支援ツールはテストスイートを生成するものであれば，基本的に何でもよい．ミューテーションツールに関しては，ミュータントを生成できるものであればよく，Java 用のツール[11]，Ruby 用のツール[12] などがある．また，C 言語用の手法[13] や Fortran 用の手法[9] なども提案されている．さらに，通常コードをミューテーションの対象とするが，設計を対象にする場合[14] もある．
- 評価環境は，実際にテストスイートに対してミュータントを実行し，テスト支援ツールの評価を行う．

処理の流れを図 2 に沿って述べる．まず，開発者により仕様，設計，コードなどのプロダクトが作成されているとし，以下のように評価のための準備を行う．

(1) ミューテーションツールに対して，プロダクトを入力する．
(2) ミューテーションツールは，ミュータントを生成する．
(3) テスト支援ツールは，プロダクトを入力として受け取り，テストスイートを生成する．

以上の事前準備が完了したら，評価環境において実際の評価が行われる．

(4) 評価環境は，オリジナルのプロダクトに対して，テストスイートを実行する．
(5) 評価環境は，各ミュータントに対して，テストスイートを実行する．
(6) 評価環境は，オリジナルの実行結果と各ミュータントの実行結果を比較し，kill されたか否かを評価結果として出力する．

評価環境内の詳細については次小節で述べる．

本フレームワークの主な使い方は次の四つである．

- テスト支援ツールの基本評価
  テスト支援ツールを変更せずに，様々なプロダクトおよび様々なミューテーションツールを用いた場合，テスト支援ツールの基本評価ができる．ミューテーションツールを変更せずに一つしか用いなかった場合でも，使用するミュータントオペレータを変更することにより生成されるミュータントを変更できる．その結果，テスト支援ツールがどの種類のバグに弱いかなどもわかり，テスト支援ツールの改良の指針になる．
- 複数のテスト支援ツールの基本比較
  ミューテーションツールを変更せず，様々なプロダクトおよび複数のテスト支援ツールを用いた場合，各テスト支援ツールにより得られるミューテーションスコアを比較することにより，テスト支援ツールの比較が可能となる．
- 複数のテスト支援ツールの詳細比較
  複数のテスト支援ツールの基本比較ではミューテーションツールを変更しなかったが，詳細比較ではミューテーションツール（またはミュータントオペレータ）も変更する．その結果，どのテスト支援ツールがどのミュータントオペレータに有効かまたは有効でないのかがわかる．
- オペレータの検証
  複数のテスト支援ツールの詳細比較から得たデータを別の角度から検証すれば，どのオペレータに対してテストケースが生成しやすいのか，または生成しにくいのかがわかる．

3.2 実　　装

本研究では，ミューテーションの対象を，サーバ側にデプロイされている Web アプリケーションとする．具体的には，実装において MuClipse[10] というミューテーションツールを利用した．MuClipse は MuJava[11] という Java を対象にしたミューテーションツールを Eclipse のプラグインとして実現したものである．近年の多くの Web アプリケーションは Java で開発されていることによる選択である．

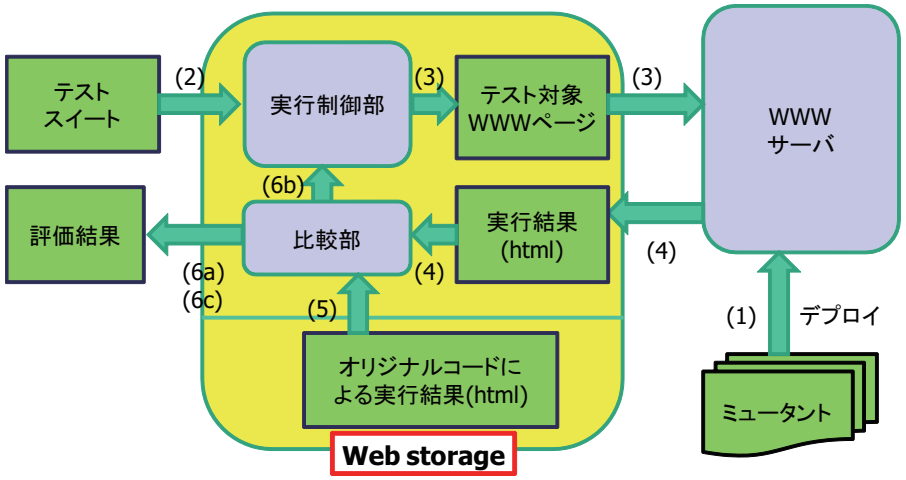

図 3 評価環境の概要
Fig. 3 Overview of Evaluation Environment.

なお，本実装では，Web ページそのものをミューテーションの対象としない．近年 Web ページにもロジックを記述するような Web アプリケーションが増えてきているが，基本は Web アプリケーションのインタフェースという立場をとり，本稿では対象とせず，Web ページそのもののミューテーションを今後の課題とする．

次に，テスト支援ツールとして，TesMa[6)7)] を使用した．TesMa は，設計書に基づいて境界値や代表値を含むテストケースをほぼ網羅的に自動生成するツールである．Web アプリケーションを対象にしていることもあり，本研究において選択した．

なお，ミューテーションツールおよびテスト支援ツールは，本実装において上記の通りそれぞれ具体的なツールを決定したが，前小節にも述べた通り，目的に応じて上記以外でも使用可能である．

評価環境の概要を図 3 に示す．まず事前処理を行う．テスト対象は Web アプリケーションであるが，Web ページを通してアクセスするため，対象となる Web ページを選択する．そして Web アプリケーションのオリジナルコードを Web サーバ上にデプロイし，テストスイート中の各テストケースを実行し，実行結果の html ファイルを Web ストレージ上に保存する．

次に，以下のように，ミュータントに対してテストを実行し，テストスイートを評価する．

(1) ミュータントを WWW サーバ上にデプロイする．
(2) 実行制御部はテストスイートを受け取る．
(3) 実行制御部は，テストスイート中のテストケースを一つ選び，そのテストケースを実行する．つまり，テストケースにある入力データを用いて，テスト対象 Web ページを通して，WWW サーバ上にあるアプリケーション（ミュータント）を呼び出す．
(4) 評価環境内の比較部は実行結果（html ファイル）を受け取る．
(5) 比較部は Web ストレージ内にあるオリジナルコードの対応する実行結果を取得する．
(6) 比較部はオリジナルコードの結果とミュータントコードの実行結果を比較する．
  (a) 異なる場合，そのミュータントが kill されたことを評価結果とする．
  (b) 同じ場合，かつテストスイート中にテストケースが残っている場合，次のテストケースを実行するように，実行制御部に知らせる．
  (c) 同じだがテストスイートにテストケースが残っていない場合，そのミュータントが kill されなかったということを評価結果とする．

4. 実　　験

本節では実験について述べる．実験は，提案フレームワークを利用するにあたり，注意すべき点や課題を明確にすることを目的とした．

実験は，Java Pet Store 2.0[15)] に対して行った．Java Pet Store は，元々Sun Microsystems 社が開発したサンプルアプリケーションである．複数の Web ページから構成されているが，本稿では，そのうち，ペットの検索を行う search.jsp に対して実験した結果を報告する．

4.1 実 験 方 法

実験は以下の手順で行った．

(1) コードから設計書を作成
  Java Pet Store にはコードがあるが，テスト支援ツール TesMa に対して入力できる設計書がない．そのため，コードを静的解析および動的実行することによって仕様を理解し，TesMa への入力用の設計書を作成した．
(2) TesMa を用いてテストスイートを生成
  合計 38 個のテストケースからなるテストスイートを生成した．
(3) MuClipse を用いて，コードからミュータントを生成

表 2 実験で対象にしたクラスとミュータント数
Table 2 Classes and the number of mutants used in the experiment.

クラス名	ミュータント数
SearchBean	24
SearchIndex	165
IndexDocument	248
PetstoreConstants	4
合計	441

表 3 実験結果
Table 3 Experimental result.

	Kill できた	Kill できなかった	等価	合計
ミュータント数	223	2	216	441
割合	50.6%	0.5%	48.9%	–

Java Pet Store の 40 クラス中 39 クラス[*1]に対して，合計 4536 個のミュータントを生成した．しかし，これをそのまま評価環境で利用した場合，合計 17 万回以上（38 × 4536）もプログラムを実行しなければならない．そこで，静的解析し，search.jsp とは関係のないクラス，つまり，search.jsp を通して呼び出されることのないクラスを省いた．その結果，関係のあるクラスは 4 つとなり，評価対象となるミュータント数を 441 個に絞り込んだ．なお，関係のあるクラスの一覧とそれぞれから生成されたミュータント数を表 2 に示す．

(4) 生成されたテストスイートとミュータントを用いて，評価環境で評価を行った．

4.2 実 験 結 果

表 3 に実験結果を示す．Kill されなかったのはたった二つのミュータントであり，他はすべて kill できたか，等価ミュータントと判断したものである．この結果，ミューテーションスコアは，0.991 という非常に高いスコアとなった．

4.3 考　　察

本小節では，まず kill できなかったミュータントについて述べ，その次に，等価ミュータントについて考察する．

4.3.1 Kill できなかったミュータント

実験において kill できなかったミュータントは次の二つであった．

(1) PetStoreConstants クラスの中で，変数宣言時の static 修飾子を削除した．

(2) SearchIndex クラスの中で，論理オペレータを反転した．

前者については，宣言が変更された変数を使うようなテストケースがあれば kill できたが，なかったために kill できなかった．これは，ミューテーションテストを行うことによって見つかることが期待される種類の問題である．つまり，テストケースを充実することにより解決できる問題である．

これに対して，後者の方は，テストケースを追加することにより簡単に解決できるような問題ではない．このミュータント内では，論理オペレータを反転させたことにより，無限ループが生じた．無限ループで実行が止まらなければ（ある意味では）kill できたことになるが，途中で Java の例外処理が無限ループを捕獲した．そしてここでミュータントプログラムが終了すれば kill できたことになるが，ミュータントは例外が発生したことをコンソール上に表示するだけで，そのまま実行し続けた．最終的な実行結果（html ファイル）はオリジナルコードによる実行結果と同じになり，kill できなかった．Kill の基準は，オリジナルプログラムとミュータントの振る舞いの違いを観測できたか否かで決まるが，本フレームワークでは実行結果の html ファイルを比較することにより，判断している．そのため，この例のようなミュータントを検出するためには，実行中にコンソールをモニタリングするなどを行う必要がある．

4.3.2 等価ミュータント

一般的に，等価ミュータントは，オリジナルプログラムと同じ振る舞いをするミュータントを指す．ただし,「振る舞い」は幅広く解釈ができ，本研究では，最終的な実行結果（html ファイル）に着目した．そこで，216 個の等価ミュータントに対して調査し，実行結果が同じであっても，振る舞いの定義によっては等価ミュータントとして見なせないものがあるか否かを検証した．

検証の結果，本実験で等価ミュータントと判定したミュータントは，以下の三種類に分類できる．

- Type 1：どんな入力を与えても，振る舞いが全く変わらない．
- Type 2：ミュータントオペレータにより変更された部分が，実行時に全く利用されない．
- Type 3：ミュータントオペレータによる変更が，あとの行で“ 訂正 ”される．

Type 1 については，厳密な等価ミュータントであり，kill するようなテストケースを作成できないものである．2 節の図 1 は Type 1 の例である．

Type 2 については，テスト対象を search.jsp という一つの Web ページに限定すると，

*1 対象としなかったクラスは，MuClipse が扱えないユーザインタフェース用のクラスである．

```
 ...
fieldx = indexDoc.getField("modified");
if (fieldx != null) {
    indexDocument.setModifiedDate(fieldx.stringValue());  // (*)
}
 ...
```

(a) オリジナルコード

```
 ...
fieldx = indexDoc.getField("modified");
if (fieldx != null) {
    indexDocument.setDisabled(fieldx.stringValue());  // (*)
}
 ...
```

(b) ミュータント

図 4 Type 2 ミュータントの例
Fig. 4 Example of Type 2 Mutant.

等価ミュータントとしてみなせるものである．図 4 に例を示す．この例では，オリジナルコードの (*) の行の setModifiedDate がミュータントにおいて setDisabled に変更されている．もちろんミュータントそのものの振る舞いはオリジナルコードから変わっているが，search.jsp ページの実行に限った場合，この変更箇所のコードを実行した結果を利用することは決してなく，最終的な実行結果（html ファイル）に伝播しない．その結果検出するようなテストケースを作成できない．そのため，本稿で等価ミュータントとして判定したのは妥当と考える．

Type 3 についても，同様にテスト対象を search.jsp に限定すると，等価ミュータントとしてみなせるものである．図 5 に例を示す．この例では，オリジナルコードの (*) の行の setImage がミュータントにおいて setProduct に変更されている．ミュータントでセットされた product の値は後で利用されるが，その前に，(+) の行で正しい値にセットされる．そこで，この例でも，ミュータントそのものの振る舞いはオリジナルコードから変わっているが，(1) 間違ってセットされた値（product の値）は使用される前に訂正され，(2) セットされなかった値（image の値）は利用されず，実行結果に伝播しない．よって，検出するようなテストケースを作成できず，本稿で等価ミュータントとして判定したのは妥当と考える．

```
 ...
fieldx = indexDoc.getField("image");
if (fieldx != null) {
    indexDocument.setImage(fieldx.stringValue());  // (*)
}
 ...
fieldx = indexDoc.getField("product");
if (fieldx != null) {
    indexDocument.setProduct(fieldx.stringValue());  // (+)
}
 ...
```

(a) オリジナルコード

```
 ...
fieldx = indexDoc.getField("image");
if (fieldx != null) {
    indexDocument.setProduct(fieldx.stringValue());  // (*)
}
 ...
fieldx = indexDoc.getField("product");
if (fieldx != null) {
    indexDocument.setProduct(fieldx.stringValue());  // (+)
}
 ...
```

(b) ミュータント

図 5 Type 3 ミュータントの例
Fig. 5 Example of Type 3 Mutant.

Type 2 や Type 3 を kill するためには，以下の二つの方法が考えられる．

- 厳密には途中の振る舞いが変わっているため，実行中のプログラムの内部状態を観測することにより，振る舞いの変更を検出し，kill する．
- 変更箇所を利用し，それが実行結果に伝播する Web ページを通して，ミュータントを検出し，kill する．

まず前者については基本的に小さなプログラムでない限り非現実的である．ミューテーションテストを行う場合，どこに変更が行われているのかを知らずに実行する．そこで，前者を実現するためには，プログラムの随所に状態チェックを行う必要があるが，ミューテーションテストにおける実行回数（ミュータント数 × テストケース数）のことを考えると，膨大な量のチェックを行うこととなる．

これに対して，後者については，ミューテーションテストの本来の姿であると考えられる．本実験では，search.jsp を一種の単独のアプリケーションとしたため，Type 2 や Type 3 ミュータントが生じた．しかし，通常は，Java Pet Store 全体を一つのアプリケーションとしてミューテーションテストを行う．そのため，Java Pet Store 全体に対して Type 2 や Type 3 の等価ミュータントが発生するとしたら，そもそもその行が不要であることを示唆する．

最後に，表 4 に本実験における等価ミュータントの Type 1，2，3 の内訳を示す．表 4 より，Type 1 ミュータントの数は等価ミュータントの 3.7%（全ミュータントの 1.8%) にすぎない．これに対して，Type 2 および Type 3 は非常に多い．この最大の原因は，本実験では，Java Pet Store 中の一つの Web ページ（search.jsp）を通した実行のみを行っているからである．そのため，他の Web ページを通した場合のみに，意味のある行までも実行される．

ここで注目すべき点として，Jia らのサーベイ[8] では，全ミュータントうち 10〜40%が等価ミュータントであるということを述べている．これはアプリケーション全体を対象にしているため，基本的に上記のうちの Type 1 のみが対象となるものの，単純に比較できない．また，4.1 小節でも述べたが，4536 個のミュータントのうち，search.jsp と関係のないクラスに関わるものを最初から外した．これは実に 9 割も占めるものである．そこで，今後の課題とはなるが，Java Pet Store 全体に対してミューテーションテストを行った場合，等価ミュータントが Jia らのサーベイで示した 10〜40%の範囲内に入るのか否かが興味深い点である．

表 4 等価ミュータントの内訳
Table 4 Equivalent mutants.

	Type 1	Type 2	Type 3	合計
ミュータント数	8	138	70	216
割合	3.7%	63.9%	32.4%	

5. おわりに

本稿では，Web アプリケーションを対象に生成したテストスイートを評価するために，ミューテーションテストを利用した場合のフレームワークを提案した．提案フレームワークをどのように評価に利用できるかを示し，またその実装についても述べた．Java Pet Store の search.jsp を対象に実験を行い，実験結果に対して考察を行った．特に，kill できなかったミュータントおよび等価ミュータントに対して考察を行った．その結果，ミューテーションテストを行うにあたり，kill するための基準となる「振る舞い」の定義，および等価ミュータントの扱いが重要であることが明らかになった．

今後の課題として，まずより多くの実験を行うことである．本稿では，一つの Web アプリケーションの一つのページに対してのみ示した．他の Web アプリケーションでも同様の結果を示すか調査する必要がある．また，本稿では，一つのテスト支援ツールに対して行ったため，他のテスト支援ツールを用いて実験すべきである．

次に，例外発生を本実装がチェックしていなかったために kill できなかったミュータントに対応する必要がある．本文中でも述べたが，実行結果 html のチェックに加えて，コンソールなどをモニタリングすることにより対応できると考えられる．

また，本研究では WWW サーバ上にデプロイされている Web アプリケーションを対象にしており，Web ページを対象にしていない．Praphamontripong らは，Web アプリケーションにおける JSP ファイルと html ファイルに対して 11 個のミュータントオペレータを提案している[16] が，同様に Web ページに対するミューテーションも検討すべきである．

最後に，今後の課題として最も重要なのは効率的に等価ミュータントを検出することである．正しくミューテーションスコアを算出するためには，kill されなかったミュータントから等価ミュータントを取り除く必要があるが，本実験では等価ミュータントを手動で検出した．ミューテーションテストが現実的なものになるためには，等価ミュータントを自動検出する必要がある．等価ミュータントの検出に関する研究が存在する[17][18] が，まだ解決されているとは言い難い[8]．

参 考 文 献

1) Kam, B. and Dean, T.: Lessons Learned from a Survey of Web Applications Testing, *6th International Conference on Information Technology: New Generations*, pp.125–130 (2009).
2) Ricca, F. and Tonella, P.: Analysis and Testing of Web Applications, *23rd International Conference on Software Engineering (ICSE 01)*, pp.25–34 (2001).
3) Marchetto, A. and Tonella, P.: Search-based Testing of Ajax Web Applications, *1st International Symposium on Search Based Software Engineering*, pp.3–12 (2009).
4) Mesbah, A., van Deursen, A. and Roest, D.: Invariant-based Automatic Testing of Modern Web Applications, *IEEE Transactions on Software Engineering*, Vol.38, No.1, pp.35–53 (2012).
5) Selenium: SeleniumHQ – Web application testing system, (online), available from ⟨http://seleniumhq.org/⟩ (accessed 2012-02-15).
6) 張暁晶, 星野隆：設計モデルを用いたテスト項目抽出とテストデータ生成手法, 電子情報通信学会 信学技報 SS2009-7, pp.37–42 (2009).
7) 丹野治門, 張暁晶, 星野隆：結合テストにおけるテスト項目自動生成手法の提案と評価, 電子情報通信学会 信学技報 SS2010-34, pp.37–42 (2010).
8) Jia, Y. and Harman, M.: An Analysis and Survey of the Development of Mutation Testing, *IEEE Transactions on Software Engineering*, Vol. 37, No. 5, pp. 649–678 (2011).
9) DeMillo, R., Guindi, D., King, K., McCracken, W. and Offutt, A.: An Extended Overview of the Mothra Software Testing Environment, *2nd Workshop on Software Testing, Verification, and Analysis*, pp.142–151 (1988).
10) Smith, B. H. and Williams, L.: An Empirical Evaluation of the MuJava Mutation Operators, *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007 (TAICPART-MUTATION 2007)*, pp. 193–202 (2007).
11) Ma, Y., Offutt, J. and Kwon, Y.: MuJava: An Automated Class Mutation System, *Journal of Software Testing, Verification, and Reliability*, Vol.15, No.2, pp.97–133 (2005).
12) Clark, K.: seattlerb's heckle-1.4.3 Documentation, Seattle Ruby Brigade (online), available from ⟨http://docs.seattlerb.org/heckle/⟩ (accessed 2012-02-16).
13) Agrawal, H., DeMillo, R., Hathaway, B., Hsu, W., Krauser, W. H.E., Martin, R., Mathur, A. and Spafford, E.: Design of Mutant Operators for the C Programming Language, Seminar on Mathematical Sciences SERC-TR-41-P, Purdue University (1989).
14) Hollmann, A.: Model-Based Mutation Testing for Test Generation and Adequacy Analysis, PhD Thesis, Universitat Paderborn (2011).
15) Oracle: Java Petstore 2.0, Oracle (online), available from ⟨http://java.sun.com/developer/releases/petstore/⟩ (accessed 2012-02-15).
16) Praphamontripong, U. and Offutt, J.: Applying Mutation Testing to Web Applications, *2010 IEEE Third International Conference on Software Testing, Verification and Validation Workshops*, pp.132–141 (2010).
17) Offutt, A. and Craft, W.: Using Compiler Optimization Techniques to Detect Equivalent Mutants, *Journal of Software Testing, Verification, and Reliability*, Vol.4, No.3, pp.131–154 (1994).
18) Offutt, A. and Pan, J.: Automatically Detecting Equivalent Mutants and Infeasible Paths, *Journal of Software Testing, Verification, and Reliability*, Vol.7, No.3, pp.165–192 (1997).