情報処理学会論文誌 Vol.56 No.3 1–12 (Mar. 2015)

DePoT: Webアプリケーションテストにおけるテストコード自動生成テスティングフレームワーク

青井 翔平[1] 坂本 一憲[2] 鷲崎 弘宜[1] 深澤 良彰[1]

受付日 2014年6月30日, 採録日 2014年11月28日

概要：Web アプリケーションは大規模化に伴いテストコードの作成コストが増大している上，保守コストが膨大であるという問題がある．特に，Web アプリケーションは他のアプリケーションよりも仕様変更の頻度が高く，また，アサーションの記述に関して構造化する手法が存在していないため，可読性および変更容易性の低下により，保守コストが増大しがちである．
本稿では，Web アプリケーションの頻繁な仕様変更に対応可能なページオブジェクトデザインパターンを利用した保守性の高い内部 DSL および，内部 DSL に基づくテストコードを自動生成するテスティングフレームワーク DePoT を提案する．DePoT はテストコードの自動生成によりテストコードの作成コストを削減して，また，ページオブジェクトデザインパターンと内部 DSL により保守コストを削減する．我々は被験者実験にて人手でテストコードを記述する従来手法との比較を通して，DePoT の有用性を確認した．

キーワード：Web アプリケーション，テスト，Selenium，不変条件，ページオブジェクトデザインパターン

DePoT: Testing Framework for Web Application Test

Shohei Aoi[1] Kazunori Sakamoto[2] Hironori Washizaki[1] Yoshiaki Fukazawa[1]

Received: June 30, 2014, Accepted: November 28, 2014

Abstract: The sizes of web applications are increasing, thus, development costs to write test code for web applications are also increasing. Moreover, specification changes of web applications are frequently occured and there are no method to modularize assertions of test code. Such situation poses high maintenance costs by decreasing understandability and changeability.
We propose a novel testing framework for web applications, named DePoT, which provides an internal DSL on the basis of the Page Object design pattern. DePoT reduces the development costs of test code by generating a skeleton test code using the internal DSL and reduces the maintenance costs of test code by using the Page Object design pattern and the internal DSL. We evaluate DePoT by comparing traditional testing methods.

Keywords: Web application, test, Selenium, invariant, PageObject design pattern

1. はじめに

近年，Web ブラウザ上で動作するアプリケーションである Web アプリケーション（以降，Web アプリ）が増加している．本稿における Web アプリは，Web サーバに配置されたプログラムから受け取った HTML 文章をクライアントのブラウザが表示することで動作するアプリケーションのことを指し，特に，Web アプリを構成する画面が Web ページの単位で区別できるアプリケーションを指す [*1]．

Web アプリは大規模化および多機能化の傾向にあるため [1]，Web アプリの品質を保証する上で，機能を検証す

[1] 早稲田大学
Waseda University
[2] 国立情報学研究所
National Institute of Informatics

[*1] 多くの Web アプリは各 Web ページに対して URL を割り当てるため Web ページの区別が容易である．しかし，Ajax により，同一の URL で様々な種類の画面を表示する Web アプリも存在する．本稿では明確に Web ページという単位を人が認識できる Web アプリを対象としており，Ajax の利用によりその区別が難しい場合は適用対象外として議論しない．

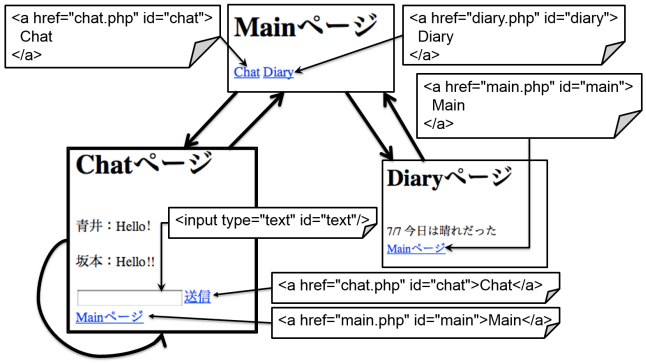


図 1 Web アプリケーションのページ構成例
Fig. 1 Example of page structure of web application

```
1  @Test public void test() {
2    // シナリオ部
3    WebDriver d = new FirefoxDriver();
4    d.get(topPageUrl);
5    d.findElement(By.id("diary")).click();
6    d.findElement(By.id("main")).click();
7    d.findElement(By.id("chat")).click();
8    d.findElement(By.id("text")).sendKeys("Hello!");
9    d.findElement(By.id("chat")).click();
10   // アサーション部
11   assertThat(d.getTitle(), is("Chatページ"));
12   assertTrue(d.getPageSource().contains("Hello!"));
13 }
```

図 2 Selenium を用いたテストコードの例
Fig. 2 Example of test code using Selenium

る Web アプリテストの重要性が増している．Web アプリテストは，テストケースと呼ばれるテストの実行手順と期待される動作結果のペアであり，テストを実行することで，Web アプリが仕様通りに動作するかどうかを検証できる [2]．本稿では，ブラウザ操作ツールである Selenium [3] を利用した Web アプリテストを対象としており，機能テストもしくは受け入れテストの粒度の高いテストを対象とする．

本稿が扱うテストケースは，Web アプリの操作を記述するシナリオ部と検証を記述するアサーション部で構成される．アサーションはテストオラクルを表現しており，本稿では DOM 要素の状態を検証するテストオラクルを対象とする．図 1 と 2 で，シナリオとアサーションを説明する．図 1 で Main，Chat，Diary ページの 3 つの Web ページから構成される Web アプリを示す．図 1 の Web アプリでは，Main ページから Diary ページと Chat ページへの相互遷移が可能となっている．Chat ページではテキストボックスに文字列を入力して送信リンクをクリックすることで，入力した内容をページ中心部にメッセージとして表示できる．

図 2 で図 1 の Web アプリに対するテストケースをコードで表現したテストコードの例を示す．図 2 における 2-9 行目がシナリオ部で 10-12 行目がアサーション部を示す．図 2 は Selenium を利用しており，ブラウザの操作をテストコードで表現することで，テスト実行を自動化する．Main ページに対して図 2 のテストコードを実行すると，5 行目で `id` 属性が `diary` のリンクをクリック，6 行目で `id` 属性が `main` のリンクをクリック，7 行目で `id` 属性が `chat` のリンクをクリック，8 行目で `id` 属性が `text` のテキストフィールドに `Hello!` と入力，9 行目で `id` 属性が `chat` のリンクをクリックする．最後にアサーション部の 11 行目で Web ページのタイトルが `Chat` ページに，12 行目で `Hello!` という文字が存在することを確認する．なお，Selenium はテストの成否を判定するアサーションを記述する機能を有していないため，一般に，JUnit などのテスティングフレームワークを併用する．図 2 は JUnit を併用しており，11 行目と 12 行目は JUnit が提供するアサーションのメソッドを利用している [4]．なお，以降で説明するテストコードは，同様に Selenium と JUnit の両方を使用している．

Web アプリは大規模化の傾向にあるため，手動でのテストコードの記述には大きなコストを伴う．その上，Web アプリは一般のアプリケーションと比較して，機能追加やデザイン変更などの仕様変更の頻度が高い特徴があり，一日に数回以上変更されることも珍しくない [5]．そのため，変更前は正しく機能していたテストコードが，変更後に機能しなくなることが多い．その結果，仕様変更に伴いテストコードの修正が頻繁に必要となり，保守コストの増大および保守性の低下を招いている [6]．

本稿では，Web アプリテストのための新しいテスティングフレームワーク Declarative PageObjects Testing Framework（以降，DePoT）[*2] を提案する．DePoT は Web アプリの頻繁な変更に対応可能なページオブジェクトデザインパターン [6–9] を利用した保守性の高い Java 言語の内部ドメイン特化言語（内部 DSL）を有する [*3]．提案 DSL を用いることでテストコードにページオブジェクトデザインパターンを適用した上で，アサーションの記述を文脈依存と文脈非依存に分離することを支援する．また，DePoT はテスト対象の Web ページの URL を与えることで DSL を用いたテストコードを自動生成する機能を有する．

本稿では，2 節で現状の Web アプリテストの問題を述べ，3 節で我々の提案する DePoT による問題解決ついて述べ，4 節で DePoT を用いたユーザ実験を示し，5 節で関連研究について述べ，6 節で本稿をまとめる．

2. Web アプリテストの問題

本節では，テストケースの作成コストと修正コストの 2 つの問題点を説明する．

2.1 テストコード作成コスト

Web アプリは大規模化・複雑化する傾向である [5]．大

[*2] https://github.com/saruhei/DePoT
[*3] 本稿は，重要な評価実験や関連研究について追記して，FOSE2012 で発表した論文を拡張している．

規模化に伴い Web ページ数やページ間で遷移するためのリンクが増大しており，また，複雑化に伴い 1 ページで提供する機能が増えている．したがって，Web アプリテストにおいても，記述すべきシナリオ部およびアサーション部が増大しており，その結果，テストコードを作成する際の記述コストが大きくなる問題が生じている．テストコード作成コストを削減するために，テストコードを自動生成する手法が提案されているものの，既存手法は後述する仕様変更が起きた際の修正コストに対して十分に考慮されていない場合 [10–14] や，自動生成にあたり HTML テンプレートが必要であるという制約がある場合 [9] がある．

2.2 テストコード修正コスト

Web アプリのテストにおいて，テストコードの変更容易性および可読性の低下の 2 点が原因となり，テストコードを修正する際のコストが増大する問題が引き起こされている．

2.2.1 テストコードの変更容易性の低下

Web アプリは仕様変更の頻度が非常に多く，特に，デザインや文言の変更に伴う DOM 要素の追加・削除・移動などの変更が頻繁に行われる [5]．Leotta らはこれらの変更が Web アプリテストの修正コストの増大を招いていることをして，ページオブジェクトデザインパターンを使わない場合と使った場合を比較することで，ページオブジェクトデザインパターンが修正コストを削減できることを明らかにした [6]．本稿でも同様に，デザインや文言等の変更が頻繁に起こる Web アプリを対象とする．

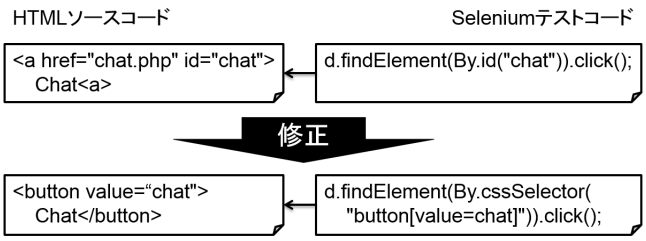


図 3 変更前と変更後の HTML ソースコードと Selenium テストコード

Fig. 3 Selenium test code before / after modification

図 3 で HTML ソースコードおよび Selenium テストコードの変更例を示す．図 3 では，図 1 の Main ページに存在するリンクがボタンに変更された例を示している．図 3 の左上と左下は変更前と後の HTML ソースコードで，右上と右下は変更前と後の Selenium テストコードを示す．

変更前では `id` 属性が `chat` であるリンクを取得していたが，変更後では当該する DOM 要素が存在しなくなったため，新たに追加されたボタンを取得するように Selenium テストコードが修正されている．もし，図 3 の修正前のテストコードが複数のテストケースで利用されている場合，複数箇所での修正が必要であるため，修正に必要な記述コストがさらに増大する．

2.2.2 テストコードの可読性の低下

可読性の低いコードは，修正を困難にして，コードの保守性を下げる要因とされている [15]．Web アプリテストのテストコードの断片はコンテキストによって意味と振る舞いが大きく変わる．例えば，図 2 の 7 行目と 9 行目のステートメントは，7 行目は Main ページ内の Chat ページへのリンク，9 行目は Chat ページ内の送信リンクに対する操作を想定しており，どちらも全く同じステートメントであるのにもかかわらず，実行時の意味および振る舞いが異なる．このような例が多数テストケースの中に存在する場合，テストケースの可読性が低くする原因となり，その結果，修正箇所の特定に要する時間を増大させて，保守性を低下させる．なお，我々が調査した限りは，テストケースの数を削減することで，テストコードの理解に必要な時間を削減する研究は存在しているが [16,17]，ページオブジェクトデザインパターンを除いて Web アプリのテストの可読性を改善する手法は存在しない．

3. DePoT による問題の解決

本稿では，2 節で述べた問題を解決するフレームワークとして DePoT を提案する．DePoT は，前節で述べたテストコード修正コストの問題を解決するために，ページオブジェクトデザインパターンを適用した上で文脈依存および非依存なアサーションを分離記述したテストコードを作成するための内部 DSL を提供する．ページオブジェクトデザインパターンが修正コストを削減することは既に確認されており，本稿では文脈依存および非依存なアサーションを分離記述することでさらなる改善を図る．さらに，テストコード作成コストの問題を解決するため，また，内部 DSL の利用を促すために，DePoT は内部 DSL で記述されたテストケースのテンプレートを自動生成する．

図 4 で DePoT のプロセスを示す．DePoT は Web アプリを Web ページの集合として捉え，各 Web ページに対応する URL を入力すると，内部 DSL で記述されたテストコードのテンプレートを自動生成する．なお，開発者は，具体的なテストケースの内容を生成されたテンプレートに追記する必要がある．

3.1 ページオブジェクトデザインパターン

ページオブジェクトデザインパターンは Selenium の開発者らが提案するテストコードのデザインパターン [7] であり，コードの重複を削除することでテストコードの変更容易性を改善して，また，テストコードのモジュール化を通してクラス名やメソッド名を与えることでコードの可読性を改善する．ページオブジェクトデザインパターンは，各 Web ページに対してクラスを作成して，各ページが提

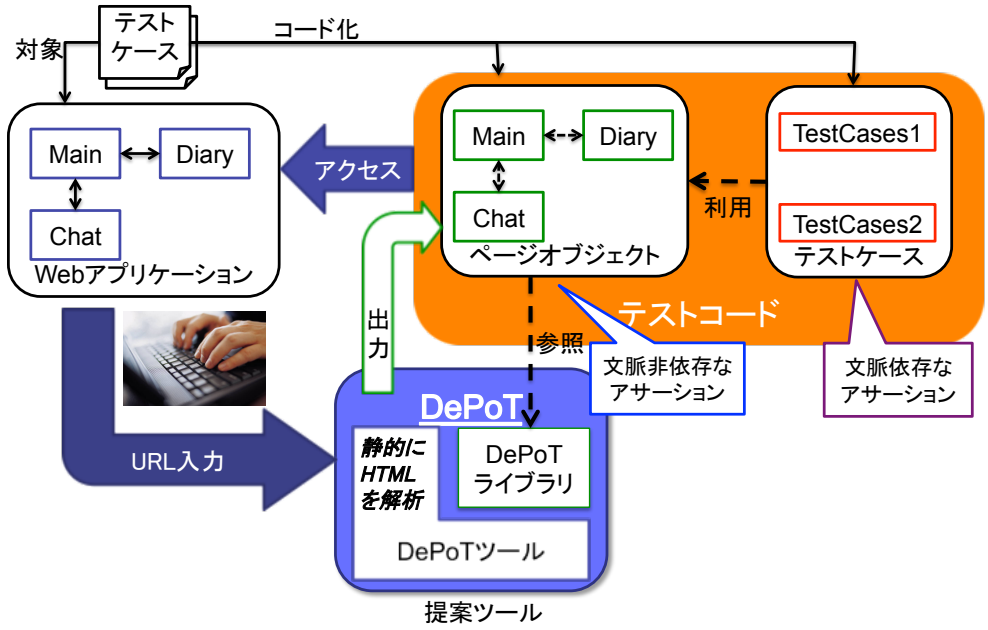


図 4 DePoT のアーキテクチャとふるまい
Fig. 4 Architecture and behaviour of DePoT

供する GUI コンポーネントおよび操作に該当するフィールドおよびメソッドを作成する．図 5 で，図 1 の Web アプリに対して，ページオブジェクトデザインパターンを適用した Selenium テストコードの記述例を示す．

図 5 の上部でページオブジェクトデザインパターンを適用した (1)Main ページに該当するページオブジェクトのクラスを示す．`MainPage` クラスは Main ページの GUI コンポーネントおよび操作を提供しており，このように Web ページ単位でモジュール化されたクラスをページクラス，そのオブジェクトをページオブジェクトと呼ぶ．DePoT におけるページオブジェクトのクラスは，(1.1)GUI コンポーネントのオブジェクトを指すフィールドと，(1.2) 文脈非依存なアサーションの定義，(1.3) フィールドを初期化して文脈非依存なアサーションを実行するコンストラクタ，(1.4) 他のページへ遷移するためのメソッドを有する．文脈非依存なアサーションについては次節で詳細を説明する．

図 5 の下部では，(2) ページオブジェクトを用いたテストケースを示しており，Main ページ，Diary ページ，Main ページ，Chat ページの順で各 Web ページに遷移する．go を接頭辞とするメソッドの戻り値は遷移先のページのページオブジェクトとなっているため，メソッドチェーンを利用して複数回の遷移を簡潔に表現できるようになっている．

ページオブジェクトデザインパターンを利用することで，Web ページ上の GUI コンポーネントを取得するコードはページクラス内に隠蔽される．したがって，Web アプリに仕様変更が起こり，DOM 要素へのアクセス，つまり GUI コンポーネントを取得するコードの修正が必要になった場合，ページクラスそのものの修正は必要になるものの，ページクラスを利用しているテストケースの修正は一切不要である．このことは，従来のテストコードで複数のテストケースに散らばっていたコードがページクラスに集約され，重複が削除されたため実現した．また，図 5 のように遷移の処理に対してメソッドの名前が与えられているため，人間が Web ページの遷移がどのように行われているかを理解しやすく，テストコードの可読性が改善される．

DePoT では，Web ページの URL を入力することで，HTML ソースコードを取得する．得られた HTML ソースコードを解析することで，(1) ページオブジェクトのクラスのうち，(1.1)GUI コンポーネントと (1.3) コンストラクタ，(1.4) 他のページへ遷移するためのメソッドに加えて，(3) ページオブジェクトのクラスの親となる抽象クラスを自動生成する．したがって，ページオブジェクトデザインパターンの適用を促し，上述の利益を享受できる．なお，(1.2) 文脈非依存なアサーションと (2) ページオブジェクトを用いたテストケースは，ユーザが記述する必要がある．

3.2 文脈依存/非依存なアサーション

我々は，Design by Contract における事前条件，事後条件，不変条件の概念 [18] と，既存研究において不変条件を用いた Web アプリのテスト手法 [10,14] を参考に，Web アプリのテストにおけるアサーションを文脈依存/非依存の 2 種類に整理した．我々は文脈を，ある Web ページにおいてアサーションを処理する時点までに，遷移したページや関連するページに対して行われた操作や遷移列の全てもし

```
1  //(1) Mainページに該当するクラス (自動生成)
2  class MainPage extends AbstractPage<MainPage> {
3    // (1.1) GUIコンポーネント (自動生成)
4    @FindBy(id = "chat") public WebElement chat;
5    @FindBy(id = "diary") public WebElement diary;
6
7    // (1.2) 文脈非依存なアサーション (人手で記述)
8    @Override protected void assertInvariant() {
9      assertTitle("Mainページ");
10   }
11   // (1.3) コンストラクタ (自動生成)
12   public MainPage(WebDriver driver) {
13     super(driver);
14     // 文脈非依存なアサーションの実行
15     assertInvariant();
16   }
17   // (1.4) 他ページへ遷移するメソッド (自動生成)
18   public ChatPage goChat() {
19     chat.click();
20     return new ChatPage(driver);
21   }
22   public DiaryPage goDiary() {
23     diary.click();
24     return new DiaryPage(driver);
25   }
26 }
27 // (2) テストケース (人手で記述)
28 @Test public void test() {
29   // StarterはMainページを開く
30   new Starter(new FirefoxDriver())
31     .goDiary()
32     .goMain()
33     .goChat()
34     .write("Hello!")
35     // 文脈依存なアサーション
36     .assertContext(new AssertFunction<ChatPage>() {
37       @Override public void assertPage(ChatPage p) {
38         assertTextPresent("Hello!");
39       }
40     });
41 }
42 // (3) ページオブジェクトの抽象クラス (自動生成)
43 public abstract class AbstractPage
44     <T extends AbstractPage<T>> {
45
46   protected final WebDriver driver;
47
48   protected AbstractPage(WebDriver driver) {
49     this.driver = driver;
50     //(3.1) @FindBy付きフィールドの初期化 (自動生成)
51     PageFactory.initElements(driver, this);
52   }
53
54   protected void assertInvariant();
55   // ... その他、assertion用のメソッド ...
56 }
```

図 5 DePoT が生成するページオブジェクトクラスとテストケースの記述例
Fig. 5 Example of PageObject class and test case

くは部分と定義する．一般に，JUnit 等のテスティングフレームワークを用いて記述するアサーションは文脈に依存しているため文脈依存なアサーション，テスト対象の Web ページが常に満たすべき条件を示す不変条件は文脈に依存していないため文脈非依存なアサーションと考えられる．

図 6 で，図 1 に対する文脈 A と B の二種類の遷移例を用いて，文脈依存/非依存なアサーションを説明する．文脈 A と B どちらも，Main ページから Chat ページへ遷移して，文字列を入力して送信ボタンをクリックするという操作を示す．ただし，文脈 A で入力する文字列は`"Hello!"`であり，文脈 B では`"Bye"`とする．全ての操作が完了した後で，アサーションによる検証を実施する．

図 6 の Chat ページに対する文脈依存/非依存なアサーションは以下のとおりである．

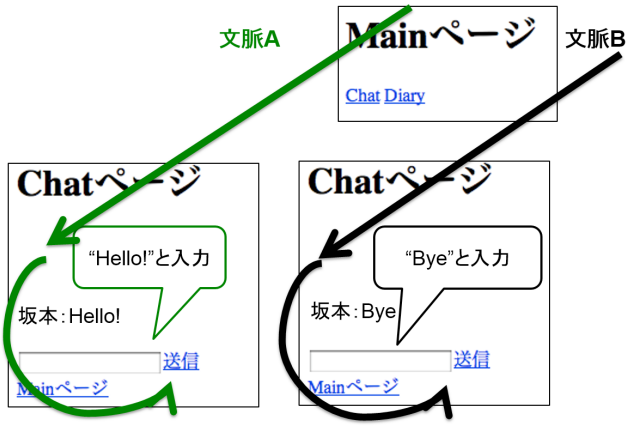


図 6 文脈 A と B による 2 つの遷移例
Fig. 6 Examples of two transitions with contexts A and B

文脈依存なアサーション 文脈 A と B でそれぞれ，Chat ページの真ん中に`"Hello!"`および`"Bye"`と表示される．

文脈非依存なアサーション 文脈 A と B の両方で，ページの上部に `Chat` ページと表示される．

文脈非依存なアサーションは当該 Web ページに遷移した際に，どんな文脈であっても常に満たされるべき条件を用いて検証を行う．従来の方法でテストコードを記述する場合は，文脈依存か文脈非依存かを区別せずに記述するため，当該ページに遷移する度に文脈非依存なアサーションを実行するコードを記述する必要がある．そのため，従来の方法ではコードクローンが大量に発生してしまい，保守性が低下する問題を引き起こす [15]．

我々は文脈依存なアサーションをテストシナリオに記述し，ページクラスに文脈非依存なアサーションを分離して記述することで，文脈非依存なアサーションの記述の重複を削除する手法を提案する．図 5 の (1.2) で，文脈非依存なアサーションの例を示す．DePoT では，ページクラスのオブジェクトが生成される度，すなわち，遷移が実行される度に文脈非依存なアサーションを実行する．図 5 の (2) で，文脈依存なアサーションを含むテストケースの例を示す．DePoT では，従来通りに当該テストシナリオを実行する際に，記述箇所において文脈依存なアサーションを実行する．

3.3 DePoT が提供する内部 DSL

DePoT では，文脈依存/非依存なアサーションを導入することでページオブジェクトデザインパターンを拡張して，拡張ページオブジェクトデザインパターン上でテストコードを記述するための内部 DSL を提供する [19]．内部 DSL のホスト言語は Java で，ドメインは Selenium で記述されたテストコードである．

内部 DSL は特徴 1)Selenium が提供するアノテーション（例えば，`@FindBy`）やメソッド（例えば，`click`）を活用

でき，さらに，特徴 2) ページオブジェクトデザインパターンを拡張して，文脈依存/非依存なアサーションを分離して記述でき，特徴 3) その結果，テストケースをメソッドチェーンのみで記述できるという 3 つの特徴を備える．表 1 と以下で，3 つの特徴を説明する．

特徴 1) Selenium が提供する機能を活用できる 図 5 の上部の (1) ページクラスのように，Selenium が提供する各種機能を活用できる．例えば，(1.1)GUI コンポーネントでは，アノテーションを用いることでフィールドを自動的に初期化する．従来では，Selenium のアノテーションを使っていても，フィールドの初期化を明示的に記述しなければならないが，DePoT では，ページオブジェクトの生成時に自動的にフィールドの初期化を実施することで記述を簡略化する．表 1 のように DePoT が生成するページオブジェクトのクラスは `AbstractPage` クラスを継承しており，(3.1)`AbstractPage` クラスのコンストラクタの中で初期化の処理を実施する．その他に，Selenium の `WebElement` クラスが提供する `click` メソッドや `sendKeys` メソッドを利用できる．

特徴 2) 文脈依存/非依存なアサーションを分離できる DePoT は文脈非依存なアサーションに該当する `assertInvariant` メソッドと，文脈依存なアサーションに該当する `assertContext` メソッドを提供する．表 1 のように `assertInvariant` メソッドはページクラス内で，`assertContext` メソッドはテストケース内で記述でき，両者を分離できる．特に，`assertInvariant` メソッドはテストケース内で記述する必要がないため，常に満たされる条件についてくり返し検査するコードを明示的に記述する必要がない．したがって，従来の方法で大量のコードクローンが生み出される問題を解決する．その他に，アサーションを簡潔に記述するために，`assertTitle` メソッドや `assertTextPresent` メソッドを提供する．

特徴 3) テストケースをメソッドチェーンのみで記述できる 従来のテストケースでは，アサーションのメソッドが静的メソッドで戻り値がないため，メソッドチェーンだけでテストケースを記述することができなかった．例えば，アサーションの記述において，一般に static import 機能を用いてクラス名を省いてメソッドを呼び出すため，メソッド名の先頭の文字を覚えていなければ，IDE の補完機能でアサーションのメソッド一覧を表示できない．その上，アサーションのメソッドを呼び出した後でページに対する操作を行う場合は，改めてページオブジェクトをレシーバーとしてメソッドを呼び出す必要がある．

一方，DePoT では，アサーションのメソッドをページオブジェクトのインスタンスメソッドにして，さらに，文脈依存なアサーションの戻り値をページオブジェクトにすることで，表 1 のようにページオブジェクトをレシーバーとして再指定をせずに，メソッドチェーンのみでテストケースを記述できるようにした．また，文脈非依存なアサーションはコンストラクタで実行することで，ユーザが明示的に呼び出しを行う必要をないようにした．したがって，ユーザは IDE のコード補完の機能と親和性の高い，メソッドチェーンのみでテストケースを記述できる．

4. 評価

本節では DePoT によるテストケースの作成コストおよび修正コストの問題に対する有用性を評価するために実施した実験について説明する．実験では DePoT を用いた場合に対して，ページオブジェクトデザインパターンを用いずに Selenium と JUnit を組合せて，人手で全てのテストコードを記述する場合を従来手法として比較を行った．

4.1 実験概要

本稿では DePoT の有用性を検証するために，以下の 2 つの研究課題 (RQ) を取り上げる．

RQ1 DePoT はテストコードの作成コストを削減できるか？

RQ2 DePoT はテストコードの修正コストを削減できるか？

RQ1 と 2 に回答するため，以下の 3 つの実験を実施した．

- テストコードの内容を理解する実験
- Web アプリの仕様変更に対してテストコードを修正する実験
- テストコードを作成する実験

実験は合計 6 名（A-F）の被験者に対して行った．被験者はコンピュータサイエンスを学ぶ大学生および大学院生であり，プログラミングやテストに対する知識を有している．

被験者の習熟度の影響をできる限り減らすために，事前に 30 分ずつ従来手法と DePoT を用いた手法で，それぞれ Web アプリに対するテスト作成の練習を行った．また，被験者 A-C を X チーム（以降 T_X），被験者 D-F を Y チーム（以降 T_Y）に分けた．各実験では，同じ作業を類似する対象に対して 2 回実施しており，T_X は従来手法を用いた後に DePoT を用いて作業に取り組み，T_Y は DePoT を用いた後に従来手法を用いて作業に取り組んだ．

実験対象は，我々が用意した図 1 と同じ構造の Web アプリとした．なお，実験で記述・修正するテストコードは以下の文脈依存/非依存なアサーションを有することを要件とした．

文脈依存 Chat ページでフォームに入力した内容が反映されるかどうか確認するアサーション

表 1 ページオブジェクトデザインパターンを用いない従来手法 (JUnit + Selenium) と DePoT の DSL で記述するテストコードの比較（ジェネリクスパラメータは省略）
Table 1 Comparison between traditional approach (JUnit + Selenium) without Page Object design pattern and DSL of DePoT

特徴	従来手法 (JUnit + Selenium)	DePoT が提供する DSL

特徴 1)
Selenium が提供する機能を活用できる
（例えば、`@FindBy` アノテーションや `click` メソッド）
	public class Test {
	  @Test public void test() {
	  WebDriver d = new FirefoxDriver();
	  d.get(topPageUrl);
	  d.findElement(By.id(”diary”)).click();
	  }
	}
		class MainPage extends AbstractPage {
		  @FindBy(id = ”diary”) public WebElement diary;
		  public void goDiaryPage() {
		  diary.click();
		  }
		}
		public class Test {
		  @Test public void test() {
		  new Starter(new FirefoxDriver())
		  .goDiary();
		  }
		}

特徴 2)
文脈依存/非依存なアサーションを分離できる
（分離するために、DePoT は `assertContext` メソッドや `AssertFunction` クラスなどを独自に提供する）
	public class Test {
	  @Test public void test() {
	  WebDriver d = new FirefoxDriver();
	  d.get(topPageUrl);
	  d.findElement(By.id(”chat”)).click();
	  d.findElement(By.id(”text”)).sendKeys(”Hello!”);
	  d.findElement(By.id(”chat”)).click();
	  assertThat(d.getTitle(), is(”Chat ページ”));
	  assertTrue(d.getPageSource().contains(”Hello!”));
	  }
	}
		class ChatPage extends AbstractPage {
		  @Override protected void assertInvariant() {
		  assertTitle(”Main ページ”);
		  }
		  // ... 省略 ...
		}
		public class Test {
		  @Test public void test() {
		  new Starter(new FirefoxDriver())
		  .goChat()
		  .write(”Hello!”)
		  .assertContext(new AssertFunction() {
		  @Override public void assertPage(ChatPage p) {
		  assertTextPresent(”Hello!”);
		  }
		  });
		  }
		}

特徴 3)
テストケースをメソッドチェーンのみで記述できる
	public class Test {
	  @Test public void test() {
	  WebDriver d = new FirefoxDriver();
	  d.get(topPageUrl);
	  d.findElement(By.id(”chat”)).click();
	  d.findElement(By.id(”text”)).sendKeys(”Hello!”);
	  d.findElement(By.id(”chat”)).click();
	  assertTrue(d.getPageSource().contains(”Hello!”));
	  d.findElement(By.id(”main”)).click();
	  }
	}
		public class Test {
		  @Test public void test() {
		  new Starter(new FirefoxDriver())
		  .goChat()
		  .write(”Hello!”)
		  .assertContext(new AssertFunction() {
		  @Override public void assertPage(ChatPage p) {
		  assertTextPresent(”Hello!”);
		  }
		  })
		  .goMain();
		  }
		}

文脈非依存 全てのページにおいてタイトルの文字列が適切かどうか確認するアサーション

なお，実験に要した時間およびコード行数を比較して DePoT の有用性の評価を行うが，測定した時間および行数に正規性を仮定できるかどうか分からないため，Wilcoxon の順位和検定によって p 値が 0.05 以下であるかどうかを確認した．

4.2 実験 1：テストコードの内容を理解する実験

図 1 の Web アプリに対して，10 種類のテストケースを有するテストコードとその内容を説明する 10 種類のドキュメントのセットを 2 種類用意した．各セットが扱うテストの内容は異なるが，遷移数およびアサーション数は同程度になっている．

まず，各セットのテストケースとドキュメントの対応関係が分からないように順番をシャッフルした．その後，各セット毎に利用する手法を変えながら，T_X と T_Y に対し

て，各テストケースがどのドキュメントが該当するかを示す対応関係を答えさせた．

なお，DePoT はテストコードの重複を取り除くことができるため，従来手法と DePoT を用いた場合のコード行数は以下のとおりであった．

従来手法 207 行

DePoT 184 行

表 2 で各セットにおいて対応関係の復元に要した時間を示す．F を除いた被験者において DePoT の方が短い時間で対応関係の復元に成功しているが，また，検定結果の p 値が 0.0625 であり，統計的には有意な差が出なかった．多くのケースで DePoT を用いることでテストコードの可読性を改善できると言えるが，一部の利用者にとっては，従来のテストコードの方が可読性が高い可能性がある．

表 2 テストコードとドキュメントの対応関係の復元に要した時間（分）

Table 2 The times to understand test code

	A	B	C	D	E	F	平均
従来手法	4:35	4:56	4:44	2:55	3:40	1:23	3:42
DePoT	1:39	2:01	2:08	2:35	3:27	1:30	2:35

4.3 実験 2：Web アプリの仕様変更に対してテストコードを修正する実験

図 1 の Web アプリに対して我々が仕様変更を行い，被験者に実験 1 で用いたテストコードを修正させた．なお，我々の仕様変更によってテストが通らない状態になっているので，全てのテストが通ることを修正完了の条件とした．

実験で行った仕様変更は以下の 5 つである．仕様変更の内容は全て HTML 文章上で行い，DOM 要素に対する変更のみである．

(1) Main ページの名称を Top ページに変更して，テキストリンクの文言を変更
(2) Chat ページの `form` タグ内のテキストボックスの name 要素を変更
(3) Chat ページの `form` タグ内にチェックボックスを追加
(4) Chat ページの送信ボタンをクリックする際に，チェックボックスの入力を必要となるように変更
(5) Diary ページのタイトルを”Dairy ページ”から”Diary ページ”に変更

我々が事前に修正作業を行ったところ，以下の行数のコードの追加・修正が必要であった．

従来手法 31 行

DePoT 5 行

表 3 で上述の修正に要した時間を示す．全ての被験者において DePoT の方が短い時間で修正作業を完了しており，また，検定結果の p 値が 0.03125 と統計的に有意な差が出た．したがって，DePoT を用いることでテストコードの修正コストを削減できることが分かる．

表 3 仕様変更に対するテストコードの修正に要した時間（分）

Table 3 result of reflection test code for modification of web application

	A	B	C	D	E	F	平均
従来手法	6:46	6:45	6:21	6:10	5:46	2:15	5:40
DePoT	2:04	1:15	1:15	3:45	4:08	1:23	2:18

4.4 実験 3：テストコードを作成する実験

図 1 の Web アプリに対して，10 種類のテストシナリオの内容を説明したドキュメントを被験者に与え，ドキュメントに則したテストコードを作成させた．

表 4 でテストコードの作成に要した時間，表 5 で作成したテストコードの行数を示す．全ての被験者において DePoT の方が短い時間・行数で作成作業を完了しており，また，検定結果の p 値が 0.03125 および 0.03501 と統計的に有意な差が出た．したがって，DePoT を用いることでテストコードの作成コストを削減できることが分かる．さらに，表 5 において，DePoT を用いたほうがコード行数のばらつきが少なく，我々はコードの自動生成によって人手で記述する箇所が減った分，属人性の排除に成功したと考える．

表 5 作成したテストコードの行数

Table 5 result of prepare test code(LOC)

	A	B	C	D	E	F	平均
従来手法	251	250	246	247	264	249	251
DePoT	208	210	209	210	210	209	209

4.5 考察

表 2-5 が示すように，全ての実験において，ページオブジェクトデザインパターンを利用せずに Selenium と JUnit を用いて人手で記述する従来手法よりも，DePoT を用いた場合の方が，テストの理解・修正・作成において必要な時間および行数の平均値が低下することを確認した．実験 1 においては，被験者 F のみにおいて従来手法の方が良好な結果を示したものの，実験 2 と 3 においては，全ての被験者が DePoT によって時間および行数を削減できることを確認した．各結果において Wilcoxon の順位和検定を行ったところ，実験 1 は統計的に有意な差が出なかったが，実験 2 と 3 においては，p 値が 0.05 以下であり，統計的に有意な差が出た．以上を踏まて，RQ1 と RQ2 それぞれに対して以下のように回答する．

RQ1：DePoT はテストコードの作成コストを削減できるか？

実験 3 では全ての被験者において，DePoT がテスト

表 4 テストコードの作成に要した時間（分）
Table 4 result of prepare test code(time)

	A	B	C	D	E	F	平均
従来手法	27:13	28:33	28:36	26:57	27:10	18:25	26:09
DePoT	23:46	17:31	21:07	23:46	18:24	14:50	19:51

コードの作成に必要な時間および行数を削減した．我々は，DePoT がテストコードを自動生成することで，従来手法と比べて記述の手間が減ったことが理由だと考える．Wilcoxon の順位和検定においても，統計的に有意な差がでており，したがって，DePoT はテストコードの作成コストを削減して，作成コストの問題を解決する．

RQ2：DePoT はテストコードの修正コストを削減できるか？

実験 1 では被験者 F 以外のテストコードの理解に要する時間の削減に成功したが，被験者 F においては，わずかに提案手法の方が理解に要する時間が短く，その結果，統計的に有意な差が得られなかった．我々は，DePoT がページオブジェクトデザインパターンに基づく DSL を提供したことで，従来手法と比べてコードが簡潔になり，可読性が上がったことが理由だと考えているが，一方で，普段記述するコードとスタイルが異なり，可読性が下がるケースもあるとも考えている．したがって，DePoT は保守コストにおけるテストコードの可読性の低下の問題を完全には解決できなかったものの，ある程度の緩和に成功した．

コードの読みやすさは過去の経験，つまり，過去に読んだコードに依存すると考えられ，被験者 F にとっては DePoT を用いたテストコードよりも従来手法を用いたテストコードの方が読みやすかった可能性がある．今後，どのようなユーザにとって DePoT のテストコードが読みやすいか，大規模な実験を通して評価を行う予定である．

実験 1 の結果にも関わらず，実験 2 では全ての被験者のテストコードの修正時間の削減に成功しており，また，統計的に有意な差が得られた．被験者 F においても大きな削減が見られた．我々は，従来手法と比べて DePoT が変更の必要な箇所を減らして，変更の手間が減ったことが理由だと考える．したがって，DePoT は保守コストにおけるテストコードの変更容易性の低下の問題を解決する．

一般に，テストコードの理解よりも修正の方がコストが掛かることを考慮すると，理解と修正の合計で考えれば DePoT の方が修正コストが少ないと言える．したがって，DePoT はテストコードの修正コストを削減して，修正コストの問題を解決する．

RQ1 と RQ2 における議論を通して，DePoT が 2 節で示した問題を解決できることが分かった．

4.6 制限

DePoT の制限として以下が挙げられる．

- **適用範囲**：現状では，DePoT は Selenium と JUnit を用いたテストコードのみをサポートする．ただし，同様の仕組みを他のテスティングフレームワークやツールに対して実装することで，サポートを広げられる．
- **静的解析に基づくテストコードの自動生成**：DePoT では，Web ページを静的解析してテストコードを自動生成するため，JavaScript など動的に DOM ツリーを変更する技術を利用する Web ページに対しては，開発者が HTML ソースコードのスナップショットを保存して，入力する必要がある．既存研究では，Ajax を利用した Web アプリをクローリングする手法が提案されており [20]，これらの手法を利用することで，HTML ソースコードのスナップショットを収集して，DePoT を適用できる．
- **アサーションの自動生成**：DePoT は，アサーションの具体的な内容そのものは自動生成できない．既存研究では，不変条件を自動生成する手法が存在するため [10, 14]，これらの手法を利用することで，DePoT が提供する `assertInvariant` メソッドの内容を生成できる．
- **テストシナリオの自動生成**：DePoT では，テストシナリオの具体的な内容そのものは自動生成できない．既存研究では，Web アプリケーションをテストする際に必要な入力値を生成する手法 [21] や，ユーザの利用履歴からモデルを作成してテストを生成する手法 [22–24]，クローリングを通してテストを生成する手法 [25] などが存在するため，これらの手法を利用することで，DePoT で実行可能なテストシナリオを生成できる．
- **Web ページ上の GUI コンポーネントの自動取得**：DePoT では，`form` タグおよび，`id` か `name` 属性が設定されている HTML タグのみを解析して，GUI コンポーネントとして操作できるようにページクラスのフィールドを生成している．JavaScript を用いている場合は他にも GUI コンポーネントとして扱うべき HTML タグが存在する可能性があるが，使用する JavaScript ライブラリに依存するため，現状の DePoT ではその全てに対応していない．しかし，GUI コンポーネントとして扱うべきタグの範囲を変更することで今後対応可能である．

4.7 妥当性への脅威

実験対象とした図 1 の Web アプリは 2.1 節で議論した大規模性を有していない．ユーザの入力により Web アプリの状態が変化しており，ある程度の複雑性は有しているものの，3 ページのみで構成される Web アプリである．DePoT は Web ページの静的解析によって発見した GUI コンポーネントに関連するテストコードを生成するため，DePoT が生成可能なテストコードの範囲は Web アプリの規模に依らない．そのため，我々は規模の大きい複雑な Web アプリであっても，DePoT は実験と同様に有効であり，妥当性に大きな影響を与えないと考えている．今後は，本稿が想定する大規模な Web アプリを対象とした実験を実施して，影響の有無を確認したい．

全ての被験者はプログラミングの知識および経験を有しているものの，Web アプリ開発の経験はほとんどなかった．したがって，現場の開発者とは知識や経験において差があり，利用者によって DePoT の有用性が変化する可能性がある．今後は，Web アプリの開発経験を有する被験者を対象とした実験を実施することで，影響の有無を確認したい．

また，4.3 節の実験において加えた仕様変更数が，現場の開発においても引き起こされるかどうかは検証できていない．ただし，仕様変更の内容はシンプルなものであり，我々は実際に起こる可能性は十分にあると考えている．今後，実験の仕様変更が実際の現場にて引き起こされるかどうかを検証して，また，より複雑な仕様変更が起こった場合においても，DePoT が有用であることを評価する予定である．

5. 関連研究

Mesbah らは，Ajax を利用する Web アプリケーションに対するクローリング手法およびツール Crawljax を提案した [20]．彼らはクローリングツールを応用することで，テストシナリオを自動生成する手法 [25] や，不変条件を推定する手法 [26] を提案している．ただし，彼らの手法を用いて生成したテストは Web アプリの仕様変更に対する対処が行われておらず，テストコードの作成コストの削減はできているものの，テストコードの保守コストの削減は行えない．なお，DePoT はテストシナリオの自動生成を支援していないが，彼らの手法は支援を行っており，それぞれが対象とする領域が異なる．これらは直行した支援を行っており，両者の手法を組み合わせることで DePoT の有用性をさらに高められる．

テストシナリオの自動生成の手法に関しては，テスト入力値を生成する手法 [21] や，ユーザの利用履歴からモデルを作成してテストを生成する手法 [22–24] が存在する．また，SeleniumIDE などユーザのブラウザ操作を記録してテストで使用する手法が存在するが [12]，操作を記録して生成したテストコードの保守性が低いことが指摘されている [6]．したがって，DePoT のようなテストコードの保守性を改善する手法が必要である．

Kung らは，Web アプリテストにおけるテストケースのモジュール化について提案している [27]．彼らは Web ページを基準としたモジュール化手法と，Web アプリの状態を基準としたモジュール化手法の 2 種類を提案している．前者の Web ページを基準としたモジュール化手法はページオブジェクトデザインパターンとは異なるものの，各 Web ページが提供する処理をメソッドとしたクラスにモジュール化するという観点は同じである．なお，後者の Web アプリの状態を基準としたモジュール化手法では，待機状態，スリープ状態，許可状態など，Web アプリの様々な状態毎にクラスとしてモジュール化する手法である．ただし，彼らの手法はアサーションに対するモジュール化を考慮しておらず，アサーションを文脈依存と非依存に分離する点は DePoT の優位性である．

6. おわりに

本稿では，Web アプリの大規模化に伴うテスト作成コストの問題と，Web アプリの頻繁な仕様変更に伴うテスト保守コストの問題を指摘した．特に，テストコードの可読性および変更容易性を改善する方法が確立されておらず，従来手法ではテスト保守コストの削減が困難であった．

本稿では，新しい Web アプリのテスティングフレームワーク DePoT を提案した．DePoT はテスト保守性を改善するページオブジェクトデザインパターンを適用して，また，文脈依存および非依存なアサーションを分離記述可能な内部 DSL を提供する．さらに，Web アプリの Web ページを解析することで内部 DSL を用いたテストコードのテンプレートを生成することで，テストの作成コストを削減する．

評価実験では，提案手法と比較して可読性の改善において統計的に優位な差が得られなかったものの，変更容易性およびテスト作成コストの削減において統計的に優位な差が得られた．ただし，可読性に関しては被験者 1 名以外の 5 名において効果が得られた上，1 名の提案手法と DePoT の差はほとんどなく，変更容易性も含めたテスト保守コストの削減として考えると，DePoT による改善が見られたと考えられる．以上から，DePoT によるテスト作成コストおよびテスト保守コストの削減において有用であることが分かった．

今後の課題としては，Ajax を用いた Web アプリケーションへの対応に加えて，既存手法と組み合わせることで，テストシナリオや文脈非依存なアサーションの自動生成に対応する予定である．

謝辞 本研究は JSPS 科研費 60609139 の助成を受けたものです。

参考文献

[1] Ricca, F. and Tonella, P.: Analysis and Testing of Web Applications, *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, Washington, DC, USA, IEEE Computer Society, pp. 25–34 (2001).

[2] JSTQB: ソフトウェアテスト標準用語集 日本語版 Version 2.2.J02.

[3] Holmes, A. and Kellogg, M.: Automating functional tests using Selenium, *Agile Conference, 2006*, pp. 6 pp.–275 (online), DOI: 10.1109/AGILE.2006.19 (2006).

[4] Hunt, A., Thomas, D. and Programmers, P.: *Pragmatic unit testing in Java with JUnit*, Pragmatic Bookshelf (2004).

[5] Jazayeri, M.: Some Trends in Web Application Development, *2007 Future of Software Engineering*, FOSE '07, Washington, DC, USA, IEEE Computer Society, pp. 199–213 (online), DOI: 10.1109/FOSE.2007.26 (2007).

[6] Leotta, M., Clerissi, D., Ricca, F. and Tonella, P.: Capture-replay vs. programmable web testing: An empirical assessment during test case evolution, *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 272–281 (online), DOI: 10.1109/WCRE.2013.6671302 (2013).

[7] Developers, S.: selenium - The Page Object pattern represents the screens of your web app as a series of objects - Browser automation framework - Google Project Hosting.

[8] Hüttermann, M.: Specification by Example, *DevOps for Developers*, Apress, pp. 157–170 (online), DOI: 10.1007/978-1-4302-4570-4_10 (2012).

[9] Sakamoto, K., Tomohiro, K., Hamura, D., Washizaki, H. and Fukazawa, Y.: POGen: A Test Code Generator Based on Template Variable Coverage in Gray-Box Integration Testing for Web Applications, *Fundamental Approaches to Software Engineering*, pp. 343–358 (online), DOI: 10.1007/978-3-642-37057-1_25 (2013).

[10] Mesbah, A., van Deursen, A. and Roest, D.: Invariant-Based Automatic Testing of Modern Web Applications, *IEEE Trans. Softw. Eng.*, Vol. 38, No. 1, pp. 35–53 (online), DOI: 10.1109/TSE.2011.28 (2012).

[11] Roest, D., Mesbah, A. and Deursen, A. v.: Regression Testing Ajax Applications: Coping with Dynamism, *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, Washington, DC, USA, IEEE Computer Society, pp. 127–136 (online), DOI: 10.1109/ICST.2010.59 (2010).

[12] Sprenkle, S., Gibson, E., Sampath, S. and Pollock, L.: Automated Replay and Failure Detection for Web Applications, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, New York, NY, USA, ACM, pp. 253–262 (online), DOI: 10.1145/1101908.1101947 (2005).

[13] Di Lucca, G. A. and Fasolino, A. R.: Testing Web-based Applications: The State of the Art and Future Trends, *Inf. Softw. Technol.*, Vol. 48, No. 12, pp. 1172–1186 (online), DOI: 10.1016/j.infsof.2006.06.006 (2006).

[14] Pattabiraman, K. and Zorn, B.: DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing, *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, Washington, DC, USA, IEEE Computer Society, pp. 191–200 (online), DOI: 10.1109/ISSRE.2010.17 (2010).

[15] McConnell, S.: *Code Complete, Second Edition*, Microsoft Press, Redmond, WA, USA (2004).

[16] Harrold, M. J., Gupta, R. and Soffa, M. L.: A Methodology for Controlling the Size of a Test Suite, *ACM Trans. Softw. Eng. Methodol.*, Vol. 2, No. 3, pp. 270–285 (online), DOI: 10.1145/152388.152391 (1993).

[17] Sampath, S., Mihaylov, V., Souter, A. and Pollock, L.: A Scalable Approach to User-Session Based Testing of Web Applications Through Concept Analysis, *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, Washington, DC, USA, IEEE Computer Society, pp. 132–141 (online), DOI: 10.1109/ASE.2004.6 (2004).

[18] Meyer, B.: Applying "Design by Contract", *Computer*, Vol. 25, No. 10, pp. 40–51 (online), DOI: 10.1109/2.161279 (1992).

[19] Fowler, M.: Language workbenches: The killer-app for domain specific languages, (online), available from ⟨`http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf`⟩ (2005).

[20] Mesbah, A., Bozdag, E. and Deursen, A. v.: Crawling AJAX by Inferring User Interface State Changes, *Proceedings of the 2008 Eighth International Conference on Web Engineering*, ICWE '08, Washington, DC, USA, IEEE Computer Society, pp. 122–134 (online), DOI: 10.1109/ICWE.2008.24 (2008).

[21] Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H. and Su, Z.: Dynamic Test Input Generation for Web Applications, *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, New York, NY, USA, ACM, pp. 249–260 (online), DOI: 10.1145/1390630.1390661 (2008).

[22] Elbaum, S., Karre, S. and Rothermel, G.: Improving Web Application Testing with User Session Data, *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, Washington, DC, USA, IEEE Computer Society, pp. 49–59 (online), available from ⟨http://dl.acm.org/citation.cfm?id=776816.776823⟩ (2003).

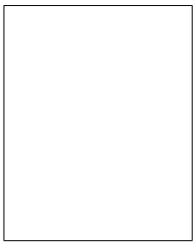
[23] Elbaum, S., Rothermel, G., Karre, S. and Fisher, M.: Leveraging user-session data to support Web application testing, *Software Engineering, IEEE Transactions on*, Vol. 31, No. 3, pp. 187–202 (online), DOI: 10.1109/TSE.2005.36 (2005).

[24] Sprenkle, S., Pollock, L. and Simko, L.: A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications, *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pp. 230–239 (online), DOI: 10.1109/ICST.2011.34 (2011).

[25] Roest, D., Mesbah, A. and van Deursen, A.: Regression Testing Ajax Applications: Coping with Dynamism, *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 127–136 (online), DOI: 10.1109/ICST.2010.59 (2010).

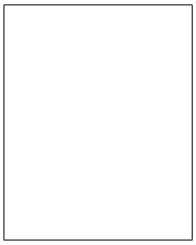
[26] Mesbah, A. and van Deursen, A.: Invariant-based Automatic Testing of AJAX User Interfaces, *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Washington, DC, USA, IEEE Computer Society, pp. 210–220 (online), DOI: 10.1109/ICSE.2009.5070522 (2009).

[27] Kung, D., Liu, C.-H. and Hsia, P.: An object-oriented web test model for testing Web applications, *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pp. 111–120 (online), DOI: 10.1109/APAQ.2000.883784 (2000).



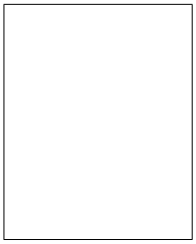
青井 翔平

2012 年早稲田大学基幹理工学部情報理工学科卒．2014 年同大学大学院基幹理工学研究科修士課程修了．2014 年より株式会社 VOYAGE GROUP に所属．現在に至る．



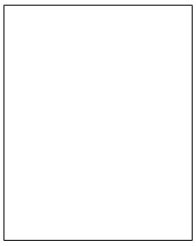
坂本 一憲 （正会員）

2013 年早稲田大学大学院博士後期課程修了，博士（工学）．2013 年より国立情報学研究所特任助教．2014 年より同研究所助教．同年 10 月より独立行政法人科学技術振興機構さきがけ研究員（兼任）．現在に至る．情報処理学会，電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．



鷲崎 弘宜 （正会員）

2003 年早稲田大学大学院博士後期課程修了、博士（情報科学）．同大学助手，国立情報学研究所助手を経て 08 年より同大学理工学術院准教授，同研究所客員准教授．再利用と品質保証を中心にソフトウェア工学の研究や教育に従事．情報処理学会代表会員，情報規格調査会 SC7/WG20 小委員会主査，日科技連 SQiP 研究会運営委員長．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．



深澤 良彰 （正会員）

1976 年早稲田大学理工学部電気工学科卒．1983 年同大学大学院博士課程修了．同年相模工業大学工学部情報工学科専任講師．1987 年早稲田大学理工学部助教授．1992 年同教授. 工学博士．主として，ソフトウェア再利用技術を中心としたソフトウェア工学の研究に従事．情報処理学会，電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．