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

DePoT : Testing Framework for Web Application Test

青井　翔平*　坂本　一憲†　鷲崎　弘宜‡　深澤　良彰§

あらまし　Web アプリケーションは仕様変更が頻繁であるため，変更前は正しく機能していたテストケースが機能しなくなることが頻繁に発生する問題がある．また，テストケースの構造化手法が確立されていないため，保守作業にかかるコストが大きくなってしまう問題がある．そこで本稿では，Web アプリケーションの頻繁な仕様変更に対応可能なデザインパターンを利用した保守性の高い内部 DSL を提案する．さらに，それらを用いた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 を提案し，テストケースの保守性の向上を図った．ユーザー実験を行った結果として保守性の向上を達成することができた．

1　はじめに

近年，Web ブラウザ上で動作する Web アプリケーション（以降，Web アプリ）が増加しており，その上で大規模化や多機能化が進んでいる．これに伴い，Web アプリの機能を検証する Web アプリテストの重要性が増している．Web アプリテストとは，テストケースと呼ばれるテスト手順仕様を与えて Web アプリが要求通りに動作するか検証する技法である．テスト手順仕様とは，テスト実行のために一連の手順を定めたドキュメントのことである [1]．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．アサーションとは，検証項目のことを指す．図 1 に Main ページ，Chat ページ，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 の各リンクを指すコードは，各リンクに対応した HTML ソースコードである．また，図 2 に図 1 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 1 の Web アプリでは，Main ページと Diary ページと Chat ページの相互遷移が可能となっている．また，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により入力した文字列がページ中心部にログとして表示される．図 2 における 5〜8 行目がシナリオ部で 9 行目がアサーション部を示す．

Web アプリテストの実行の自動化には，Web ブラウザ上のユーザーの操作をシミュレートする自動ブラウジングツールが必要である．自動ブラウジングツールは様々なものが存在するが，本稿では最も普及しているブラウジングツールである Selenium を用いる [2]．Selenium は，図 2 のようなブラウザのユーザ操作を表すコードを与える事で自動的にブラウジングを実行する．ブラウジング開始ページが Main ページのときに図 2 のコードを Selenium に与えると，5 行目で id 属性が Diary のリンクをクリック，6 行目 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 行目では id 属性が Chat のリンクをクリックする．最後に 9 行目では，Web ページのタイトルが”Chat ページ”となっていることを検証する．ただし，Selenium は自動ブラウジングツールでありテストツールではないため検証する機能がない．よって，検証をする際には JUnit などの単体テストツールを併用する．図 2 の例は JUnit を併用している例であり，9 行目で JUnit を用いた

*Shohei Aoi, 早稲田大学

†Kazunori Sakamoto, 早稲田大学

‡Hironori Washizaki, 早稲田大学

§Yoshiaki Fukazawa, 早稲田大学

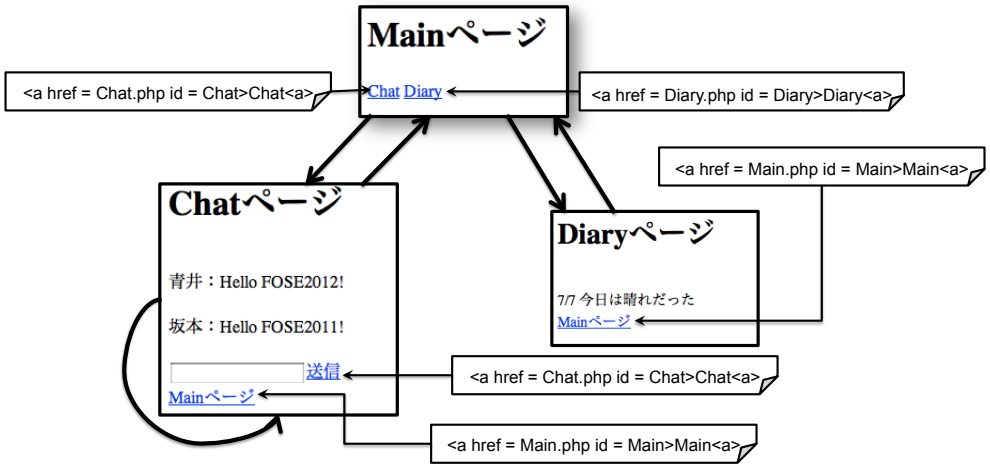

図 1　ウェブアプリケーションの構成

```
1  public class Test {
2    @Test
3    public void testing(){
4    .......
5    driver.findElement(By.id(“ Diary ”)).click(); //シナリオ部
6    driver.findElement(By.id(“ Main ”)).click(); //シナリオ部
7    driver.findElement(By.id(“ Chat ”)).click(); //シナリオ部
8    driver.findElement(By.id(“ Chat ”)).click(); //シナリオ部
9    assertThat(driver.getTitle(), is("Chatページ")); //アサーション部
10   ........
11   }
12 }
```

図 2　テストコード例

検証を行っている [3].

Web アプリは大規模化の傾向にあり，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，機能追加等の仕様変更が非常に多いという特徴があり，一日数回変更されることも珍しくない [4]．そのため，仕様変更以前は正しく機能していたテストコードが修正後に機能しなくなる問題がある．よって，修正のための記述コストも大きくなり保守性の低下を招く問題がある．また，構造化されていないテストコードは可読性を低下させるおそれがあるため，保守性を下げる要因となる．

そこで本稿では，Web アプリケーションの頻繁な変更に対応可能なデザインパターンを利用した保守性の高い内部 DSL を提案する．さらに，それらを用いたテストコードの自動生成を我々が提案し，本稿で提案する Web アプリテストの自動テストコスト削減フレームワーク DePoT により実現する．

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

Web アプリテストの問題として，テストケースの作成コスト，テストケースの修

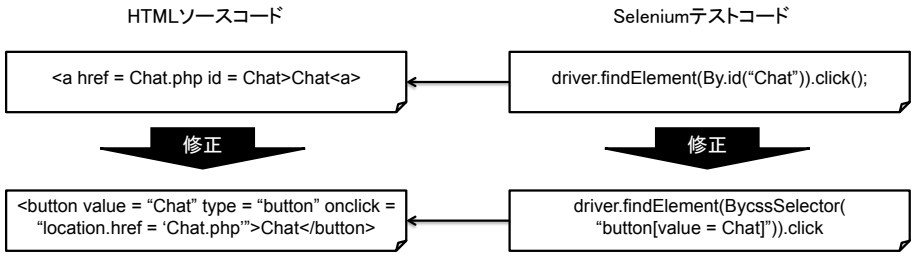

図 3　変更前と変更後の HTML ソースコードと Selenium テストコード

正コストの 2 点を説明する．

2.1　テストコード作成コスト

Web アプリは，大規模化によって Web ページの数やページに遷移するためのリンクの数が増大している．さらに，多機能化によって実装される機能が増え Web ページが複雑化している．したがって，Web アプリテストでは記述すべき遷移シナリオやアサーションが増えてしまい，その結果テストケース生成の記述コストが大きくなってしまう問題がある．

2.2　テストコード修正コスト

以下の 2 点が原因となり，テストコードの修正コストの増加が問題となっている．

- テストコードの変更非容易性
- テストコードの可読性

2.2.1　テストコードの変更非容易性

Web アプリは，仕様変更が非常に多いという特徴がある．そのため，図 3 のように HTML ソースコードが変更されることが多い．左上のコードが変更前で，左下のコードが変更後の HTML ソースコードである．図 3 では，図 1 の Main ページでリンクがボタンに変更されていることを一例に挙げる．

図 3 の修正前の HTML ソースコードに対する Selenium テストコードは右上のコードであり，修正後は右下のコードである．

図 3 の修正前のテストコードは id 属性が”Chat”であることを利用している．したがって，図 3 の HTML ソースコードの変更により，a タグが button タグになってしまうと，図 3 の修正後のコードのようにテストコードを修正しなければならない．このように，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，修正のための記述コストが大きくなってしまうおそれがある．

2.2.2　テストコードの可読性

可読性の低いコードは，修正が難しく，コードの保守性を下げる要因となるとされている [5]．我々の調査によると，Mesbah らや Roest らのように Web アプリテストのテストコードの自動生成の研究は存在するが [6] [7]，Web アプリテストのテストコード記述法について論じられていなかった．そのため，Web アプリテストのテストコードにおける構造化手法が確立されておらず，再利用性が低い．また，構造化されていないテストコードは同じステートメントであっても．ブラウジングの状態によってはふるまいが変わる．図 2 の 7 行目と 8 行目に Selenium テストコードの例を示す．7 行目実行前の状態は Main ページ，8 行目実行前の状態は Chat ページであるため，7 行目と 8 行目のステートメントではふるまいが異なり，共に id 属性が ”Chat”のリンクをクリックするユーザー操作を指し示すが指し示されるリンクは異なる．したがって，図 2 では，同じステートメントが同じテストケースの中で異なる意味を持つことになる．このような例が複数のテストケースの中に存在するとテストケースを修正する際，テストケースの可読性が非常に低くなるため，修正

箇所を特定するのに時間がかかってしまい，保守性が低下してしまう．

3　DePoT による問題の解決

本稿では，2 節で述べた問題を，解決するためのツールとして Declarative PageObjects Testing Framework（以降，DePoT）を提案する．図 4 に DePoT のプロセスを示す．図 4 の通り，DePoT は Web アプリを Web ページの集合としてとらえ，その Web ページの URL を与える事で Selenium と JUnit[6] を用いてページオブジェクトデザインパターンを適用した Java テストコードを自動生成するフレームワークである．JUnit とは単体テストの自動化を行うためのフレームワークである．

DePoT は，以下の 3 点により 2 節で述べた問題を解決する

- ページオブジェクトデザインパターンの適用
  Web ページ単位でのモジュール化によるシナリオの重複の除去と再利用性向上，テストコード修正コストの削減
- テストコードの自動生成
  テストコードの生成コスト削減
- 文脈依存/非依存なアサーションの分離
  アサーションの重複除去による再利用性向上，テストコードの修正コスト削減

3.1　ページオブジェクトデザインパターン

Selenium の開発者は，2.2.2 節に示したブラウジングの状況によって同じステートメントで異なるふるまいを持つ問題の回避やテストケース修正における記述コストを削減するために，ページオブジェクトデザインパターン [8] を考案している．ページオブジェクトデザインパターンとは，Web ページ単位でテストケースをモジュール化するデザインパターンである．図 5 に図 1 で示した構成の Web アプリに対し，ページオブジェクトデザインパターンを適用させ Selenium を用いたテストコードの例を示す．

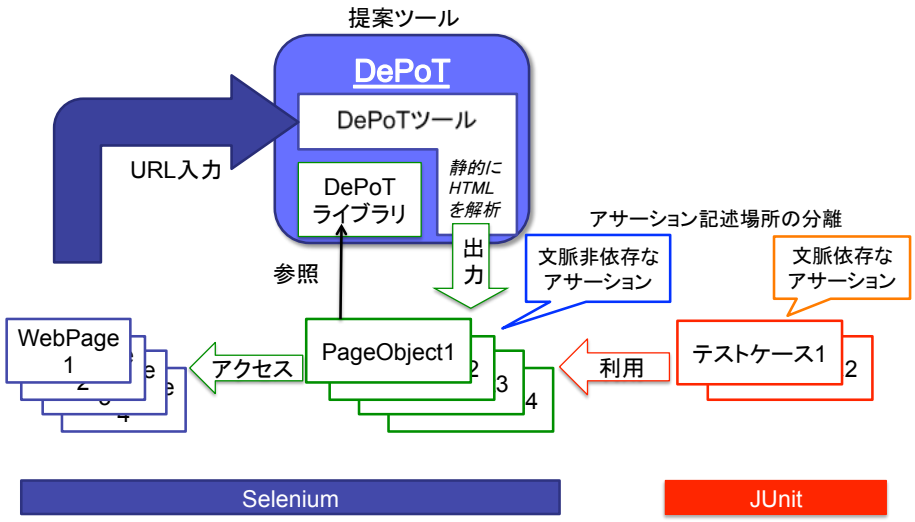

図 4　DePoT のふるまい

図 5 の (2) のコードは，Main ページのテストケースを Web ページ単位でモジュー

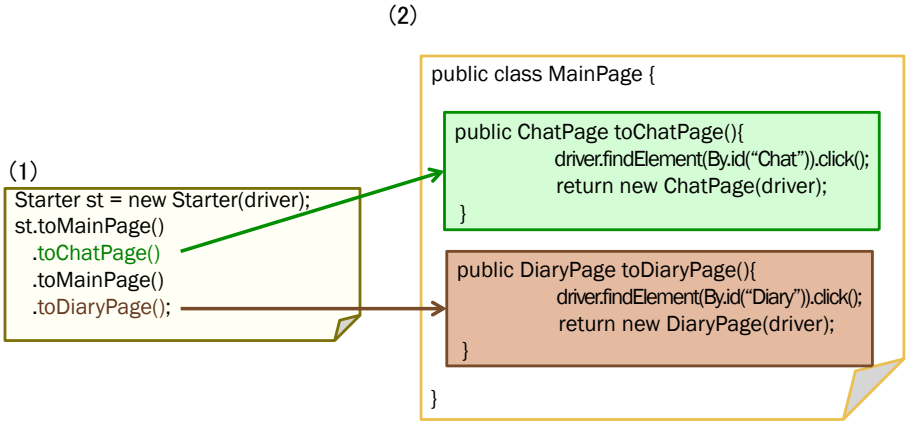

図 5　ページオブジェクトデザインパターン

ル化したコードである．これをページオブジェクトと呼ぶ．ページを 1 つのオブジェクトとしてとらえ，1 つのページの中の操作はそのオブジェクトの機能として提供する．図 5 の (2) では，Main ページ上で実行される操作を全て MainPage クラスの中に記述している．図 5 の (1) のコードはテストシナリオである．図 5 のテストシナリオは，Main ページ→ Chat ページ→ Main ページ→ Diary ページの順に Web ページを遷移している．図 5 のように，ページオブジェクトデザインパターンは，テストシナリオでページオブジェクトに記述された操作を呼び出してメソッドチェーンによって継続的に実行するオブジェクト指向なデザインパターンである．矢印は，呼び出し先のメソッドを示す．また，メソッドチェーンとはメソッドが自らのオブジェクトを返すように設計されたメソッドをつなげることによって，機能を実現する仕組みである．

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，テストケースを追加する際にコードの再利用が可能になるためシナリオ記述の重複を回避できる，また，テストコードを修正する際にページオブジェクトの記述を修正するだけで良い．その上，Web ページ単位でモジュール化するため，2.2.2 節で示すような例が発生しない．また，図 5 のようにシナリオの記述を Web ページ単位で記述できるためシナリオの Web ページ遷移が非常に理解しやすく，テストシナリオの可読性が高くなる．

3.2　DePoT によるページオブジェクトデザインパターンの実装

ユーザー入力により DePoT に Web ページの URL を与えると，DePoT はページの HTML を解析し，出力としてページオブジェクトを自動生成する．図 6 に自動生成されたページオブジェクトのコードの例を示す．また，GUI コンポーネントとはリンクやボタン等 Web ページ上に配置されたオブジェクトのことを指す．

図 6 に示す通り，以下の 3 点が自動生成される．

1. Web ページ上の GUI コンポーネントを表す変数の宣言と初期化
   GUI コンポーネントを変数に格納することにより，クラス内での再利用が可能になりコードクローンを回避することができる．
2. 文脈非依存なアサーションを記述するメソッドの枠組み
   3.3 節以降で詳しく述べる．
3. 異なる Web ページへ遷移するためのメソッド
   ページオブジェクトデザインパターンを適用し，返り値を次の遷移のページオブジェクトにすることによりテストシナリオでのメソッドチェーンによるシナ

```
1  public class MainPage extends AbstractPage {
2    @FindBy(id = "Chat")  //(1)id = "Chat"のGUIコンポーネントの格納
3    public WebElement chat;
4    @FindBy(id = "Diary") //(1)id = "Diary"のGUIコンポーネントの格納
5    public WebElement diary;
6
7    @Override
8    protected void assertInvariant(){ //(2)文脈非依存なアサーション
9      assertTitle("Mainページ");       //内容は任意で記述
10   }
11
12   public ChatPage goChat(){ //(3)ChatPageへの遷移
13     chat.click();
14     return new ChatPage(driver);
15   }
16
17   public DiaryPage goDiary(){ //(3)DiaryPageへの遷移
18     diary.click();
19     return new DiaryPage(driver);
20   }
21 }
```

図 6　ページオブジェクト実装例

リオ記述を実現し，可読性の向上を図る．

DePoT では，DePoT のライブラリを利用するために，図 6 に示す通り，AbstractPage クラスを継承したページオブジェクトのコードを自動生成する．

3.3　文脈依存/非依存なアサーション

Web アプリテストでは，文脈依存/非依存なアサーションがある．ここで，文脈とはある Web ページにおける検証する時点までに実行された当該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くは部分と定義する．以上のことから，文脈依存なアサーションとは文脈に依存し変化するアサーションであり，文脈非依存なアサーションとは文脈に依存せず，検証する Web ページに対する不変条件となる．図 7 の具体的な遷移例を用いて文脈依存/非依存なアサーションについて説明する．

図 7 は，図 1 に対して，文脈 A と文脈 B の 2 つの遷移の例を示したものである．文脈 A と文脈 B 共に，Main ページから Chat ページへ遷移し，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．しかし，文脈 A では文字列は”FOSE”であり，文脈 B では文字列は”DePoT”であると定義する．また，検証は全ての操作が終了した時点で実行する．

ここで，図 7 の Chat ページの文脈依存/非依存なアサーションは以下のようになる．

- 文脈依存なアサーション：
  文脈 A：Chat ページの真ん中に”FOSE” と表示される
  文脈 B：Chat ページの真ん中に”DePoT” と表示される．
- 文脈非依存なアサーション：
  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，そのアサーションが満たされなければいけない検証ページでは，常に満たされるアサーションである．そのため，その検証ページに遷移するたびに実行しなければならない．したがって，テストケース上では安直に実装する場合，何度も重複コードを記述する必要があり，コードクローンが発生してしまい，保守性が低下する問題がある [5]．

そこで，本稿では文脈依存なアサーションをテストシナリオに記述し，ページオブジェクトに文脈非依存なアサーションをページオブジェクトに分けて記述するこ

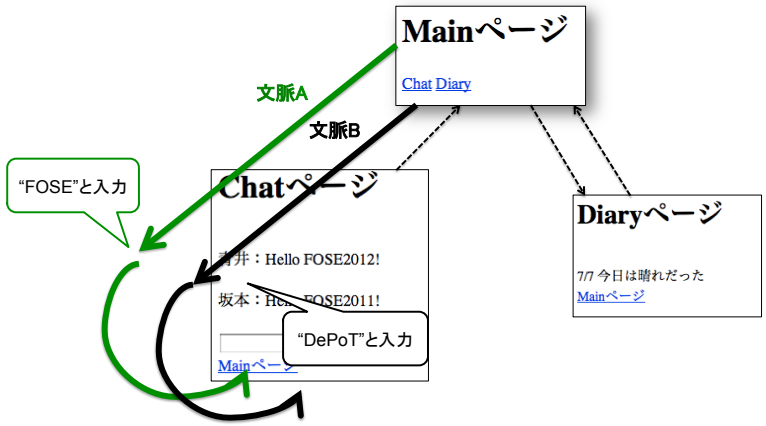

図 7　２つの遷移例

```
1  Stater st = new Starter(driver);
2  st.goMainPage() //(1)メソッドチェーン
3     .goChatPage()  //(1)メソッドチェーン
4     .goDiaryPage() //(1)メソッドチェーン
5     .assertContext(new AssertFunction<DiaryPage>()    //(2)文脈依存なアサーション
6     {
7      @Override
8      public void assertPage(DiaryPage page){
9      assertTextPresent("Diary");
10     }
11    })
```

図 8　テストシナリオと文脈依存なアサーションの例

とで，文脈非依存なアサーションの記述の重複を回避する．その上で，文脈非依存なアサーションにおいてはページオブジェクトを呼び出す際，ページオブジェクトに記述された文脈非依存なアサーションが呼びだされ，実行される構造を提案する．

3.4　DePoT での文脈依存/非依存なアサーションの実装

DePoT では，文脈依存/非依存なアサーションの分離を内部 DSL として定義している．内部 DSL とは，強力な表現力を持つ DSL を汎用言語上で再現する仕組みである． [9]．DePoT では，ホスト言語は Java である．文脈依存なアサーションは，図 6 の (2) のコードのようにページオブジェクトにメソッドとして記述することで実装する．このアサーションは自クラスのインスタンスによりブラウザ操作を実行する際，毎回呼び出される．

文脈依存なアサーションは，図 8 の (2) のようにテストシナリオに直接記述する事で実装する．DePoT では，ページオブジェクトのメソッドを継続的に呼び出し，メソッドチェーンを行う事でテストシナリオを構成する．

図 8 のシナリオは，Main ページ→ Chat ページ→ Diary ページと遷移し最後に文脈依存なアサーションを実行している．

1. メソッドチェーン
   図 8 の (1) のように DePoT のテストシナリオは，メソッドチェーンで記述することができる．Web ページ単位で抽象化したテストシナリオを記述できるため，Web ページの遷移が非常に理解しやすい可読性の高いテストシナリオを記述することができる．

2. 文脈依存なアサーション
   図 8 の (2) のように DePoT では，文脈依存なアサーションをテストシナリオ内部に assertContext メソッドとして記述することができる．メソッドオーバーライドを用いて，assertContext メソッド内部に直接アサーションを記述することができる．また，assertContext メソッドから継続的にメソッドチェーンによってシナリオを記述することができる．

3.5 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 に対して適用する．

Web ページの動的解析 DePoT では，Web ページの静的解析を行うため，ajax などの動的に DOM 要素が変更される技術が利用された Web ページに対しては，開発者が自分で Web ページを解析し，テストコードを記述する必要がある．

検証項目の自動生成 DePoT は，アサーションの具体的な内容は自動生成できない．アサーションを記述する枠組みは DePoT 側で提供するため，開発者はその枠組みに沿ってアサーションを記述する必要がある．

具体的なテストシナリオの自動生成 DePoT では，具体的なテストシナリオは自動生成できない．開発者が手動でシナリオを記述する必要がある．

Web ページ上の GUI コンポーネントの自動取得 DePoT では，属性 id もしくは，属性 name が設定されている GUI コンポーネントのみを自動取得し，変数として格納している．

4 実験

構造化されていない従来のテストコード記述法と本稿の提案手法である DePoT を用いたテストコード記述との比較実験を行い，テストコードの作成時間，修正時間をそれぞれ検証した．利用言語は Java であり，被験者は，公平性の確保のため，プログラミングの経験が Java のみで経験が 1 年程度の大学 4 年生の学生 4 名である．

4.1 実験方法

1. 被験者を，A チーム（以降 T_A）と B チーム（以降 T_B）の各 2 名ずつのチームに分割する．
2. T_A に早稲田大学（以降 P_W）の Web ページを従来手法で，T_B に東京大学（以降 P_T）の Web ページを提案手法で，それぞれこちらが与えたテストコードを作成させ，時間を計測する．なお，テストケースはドキュメントファイルに対象となる Web ページの URL とテストシナリオを記述したものを与えた．
3. それぞれのチームに Web ページの仕様変更の情報を与え，テストコードを修正させ，その時間を計測する．仕様変更の内容としては，GUI コンポーネントの id の変更と Web ページのタイトルの変更を仮定した．
4. T_A に P_T の HP を Web ページで，T_B に P_W の Web ページを従来手法で，それぞれこちらが与えたテストケースを作成させ，時間を計測する．
5. 3 の手順と同じことを行う．

P_W，P_T それぞれのテストに利用した範囲の Web ページ数，テストシナリオの数，1 シナリオにおける Web ページの遷移数，修正の際に仮定した仕様変更箇所数は表 1 の通りである．

4.2 実験結果

計測結果の平均は表 2 のようになった．表 2 のテストコード作成時間は，各人がテストケースを作成するのにかかった時間の平均を示す．従来手法ではテストコード記述開始から完成するまでの時間であり，提案手法では DePoT に入力を与えて

表 1　実験対象の情報

	Web ページ数	テストシナリオ数	1 シナリオにおけるページ遷移数	仕様変更数
P_W	8	5	7	2
P_T	7	5	7	2

から DePoT の出力を編集し，テストコードを完成させるまでの時間である．修正時間は修正内容を与えてから修正が完了するまでの時間であり，修正 1 回にかかる時間の平均を算出したものである．表 2 からわかる通り，生成時間と修正時間両方において記述時間が削減された．P_W に比べ，P_T における時間削減効果が小さいのは，P_T の Web ページの HTML が P_W の Web ページの HTML より構造化されていたため，自動生成しなくとも Web ページの GUI コンポーネントの情報を読み取るのが容易であったためだと思われる．作成時間の削減は，テストコードの作成に必要な記述コストが削減されたと考える事ができる．また，修正時間の削減は可読性が向上し修正箇所の捜索に必要なコストが削減され，修正にかかる記述コストが削減されたと考える事ができる．よって，総合的に修正に必要な労力が削減されているため，保守性が向上したと考える事ができる．

実験結果から，DePoT は以下のような成果を挙げたと考えられる．

- テストケース生成コストの削減
- テストケース修正コストの削減，テストケースの可読性の向上

これは，提案手法はにより 2 節で言及した問題を解決したことを示す．よって，提案手法は，保守性の向上を達成した．

表 2　実験結果

	従来手法		提案手法	
	P_W	P_T	P_W	P_T
テストコード作成時間 [分]	85.5	42.5	42	34
テストコード修正時間 [分]	2.5	1.5	0.05	0.2

5　関連研究

Mesbah らは，不変条件をクローリングによって自動検出し，自動的にテストケースを生成し，実行するという手法を提案している [6]．不変条件とは，本稿で述べている文脈非依存なアサーションと同意義である．Mesbah らによる提案ツールでも Selenium が利用されている．彼らの手法では，crawljax というクローラーを用いたクローリングによって文脈依存なアサーションを取得するため，動的な解析を行う事ができる [12]．しかし，テストケースの可読性，保守性を全く考慮に入れていないため，保守作業は非常に困難になると推測される．その上，文脈依存なアサーションも考慮されていないため，文脈依存なアサーションを記述することには向いていない．一方，本稿の提案手法では，ページ単位でテストケースをモジュール化することにより，保守性を向上させ，文脈依存/非依存に対応させた．その上で，可読性の高い内部 DSL を定義した．

David 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 [10]．彼らは，２種類のモジュール化について提案している．Web ページを基準にしたモジュール化と，Web アプリの状態を基準にしたモジュール化の２種類である．Web ページを基準にしたモジュール化とは，本稿も利用している手法で，1 つの Web ページの手続きを 1 つのオブジェクトで完結させる手法である．彼

らは，その上でダイアグラム化して可視性の向上も行っている．状態を基準にしたモジュール化は，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にモジュール化する手法である．この手法は，主にサーバーに有用な手法である．彼らの手法は，状態基準のモジュール化でサーバー側，ページ基準のモジュール化でクライアント側でそれぞれに適したモジュール化を行う事ができる．しかし，彼らの手法では，文脈依存/非依存なアサーションについてまったく考慮されていない．一方，本稿の手法では，文脈依存/非依存なアサーションを考慮したデザインパターンを利用している．

テストコードを自動生成するツールとしては，ブラウザ上での手動の手続きを記録し，その操作を繰り返し実行する SeleniumIDE [11] というツールが存在する．しかし，このツールで自動生成されるテストコードはモジュール化されておらず，また文脈依存/非依存なアサーションについても考慮されていない．一方，本稿の手法では，文脈依存/非依存なアサーションを考慮したデザインパターンを考慮している．

6 おわりに

本稿では，Web アプリテストのテストケース生成，修正における記述コストを問題とした．この問題に対し，Web アプリの頻繁な仕様変更に対応可能なデザーンパターンを利用し，アサーションを考慮した保守性の高い内部 DSL を提案し，それら用いたテストコードを用いたテスティングフレームワークである DePoT の作成した．また，Web アプリの頻繁な仕様変更に対応可能なデザインパターンとして，Web ページ単位でテストケースをモジュール化するページオブジェクトデザインパターンを改良し，文脈依存/非依存なアサーションに対応させ，テストケースのシナリオ記述，アサーション記述それぞれに対し，重複をさける記述を提案した．その上で，提案ツールによる従来手法との比較実験では，従来手法に対する優位性を示した．

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や，文脈非依存なアサーションの自動生成が挙げられる．

参考文献

[1] JSTQB - ソフトウェアテスト標準用語集 日本語版 Version 2.1.J01 : http://jstqb.jp/dl/JSTQB-glossary.V2.1.J01.pdf.

[2] Selenium - Web Browser Automation : http://seleniumhq.org/.

[3] JUnit - Welcome to JUnit.org! — JUnit.org : http://www.junit.org/

[4] Jozaeri, M. Some Trends in Web Application Development, *Trans. ICSE 2007, Future of Software Engineering*, pp.199-213 , 1991.

[5] Steve MacConnell（(株) クイープ訳）Code Complete 第 2 版，日経 BP ソフトプレス ,2005.

[6] Ali Mesbah,Arie van Deursen and Danny Roset. Invariant-Based Automatic Testing of Modern Web Applications, *Software Engineering, IEEE Transactions on*, vol38 Issue1 ,2011.

[7] Danny Roset,Ali Mesbah,Arie van Deursen. Regression Testing Ajax Applications: Coping with Dynamism, *In Proceedings of the 3rd International Conference on Software Testing(ICST'10)*, IEEE Computer Society ,2011.

[8] selenium -The Page Object pattern represents the screens of your web app as a series of objects - Browser automation framework - Google Project Hosting , http://code.google.com/p/selenium/wiki/PageObjects .

[9] Fowler M. Language Workbenches:The Killer-App for Domain Specific Languages? http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf

[10] David C Kung, Chien-Hung Liu, Pei Hsia. An Object-Oriented Web Test Model for Testing Web Applications, *In Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, IEEE Computer Society ,2000.

[11] Selenium IDE Plugins : http://seleniumhq.org/projects/ide/.

[12] Crawljax : http://crawljax.com/.