Regular Paper

Automating End-to-End Web Testing via Manual Testing

Hiroyuki Kirinuki[1,a) Haruto Tanno[1,b)

Received: August 2, 2021, Accepted: January 11, 2022

Abstract: End-to-end test automation for web applications is important in order to release software quickly in accordance with market changes. However, the cost of implementing and maintaining test scripts is a major obstacle to the introduction of test automation. In addition, many testing activities, such as exploratory testing, user-interface testing, and usability testing, rely on human resources. We propose an approach to generate effective test scripts from manual testing, which is indispensable in software development. Manual testing activities are recorded by our tool. The generated test scripts leverage the page-object pattern, which improves the maintainability of test scripts. To generate page objects, our approach extracts operations as methods useful for test automation from the test logs. Our approach also generates test cases that cover features of an application by analyzing its page transitions. We evaluated whether our approach could generate complete test scripts from test logs obtained from four testers. Our experimental results indicate that our approach can generate a greater number of complete methods in page objects than a current page-object generation approach. We also conducted an empirical evaluation of whether our approach can reduce the cost of implementing test scripts for real systems. The result showed that our approach reduces about 48% of the time required to implement test scripts compared with manual implementation.

Keywords: end-to-end testing, web testing, manual testing, test automation

1. Background

Software testing is an important process to evaluating and improving software quality, and developers spend time and effort on software testing [1], [2]. Many researchers aim to make software testing more efficient and detect more bugs [3]. Automated testing is important for improving the efficiency of software testing. It is especially effective for tests repeated many times such as regression testing. It has recently been required that the software release cycle is shortened in order to respond quickly to market changes. Therefore, test automation has become essential for software development.

When testing software with graphical user interfaces (GUIs) such as web applications, it is also necessary to test GUIs from the user perspective. We call this type of testing end-to-end testing, which is what we focus on in this paper. To automate end-to-end testing, tools that automate web-browser operation, such as Selenium [4], are commonly used. End-to-end test automation requires implementing test scripts. In addition, it may be necessary to modify the existing test scripts when modifying the application under test. Christophe et al. [5] investigated the change history of the source code, including the Selenium test scripts, for eight open-source web applications and how modifying applications affects these scripts. They found that 75% of Selenium test scripts were changed at least once every nine commits (once every 2.05 days). Therefore, the Selenium test scripts are updated frequently as the application evolves, so the maintainability of test scripts is important.

[1] NTT Software Innovation Center, Mintao, Tokyo 108–0024, Japan
[a) hiroyuki.kirinuki.ad@hco.ntt.co.jp
[b) haruto.tanno.bz@hco.ntt.co.jp

There are two approaches for automating end-to-end testing, i.e., *record & replay* or *programming*. Record & replay tools (e.g., Selenium IDE) record operations carried out by the tester as a test script and automate the verification by executing the test script. Thus, this approach can easily implement test scripts without the user needing programming skills, but the test scripts tend to be less maintainable because they are described as a simple sequence of recorded operations. The programming approach, on the other hand, implements test scripts as programs using libraries, such as Selenium webdriver, that operate a web browser. This approach can implement test scripts with high maintainability if the developer is skilled, but the implementation tends to be costly. Leotta et al. [6] conducted a comparative experiment on these two approaches. The experimental results indicated that the programming approach took 32–112% more time to implement test scripts than the record & replay approach. In contrast, it took 16–51% less time to modify test scripts. Considering the total cost of implementation and modification, the results also indicate that the programming approach is less costly in most cases when more than three modifications are required.

In their experiment, the participants who adopt the programming approach used the page-object pattern to implement the test scripts. The page-object pattern is a design pattern for end-to-end test automation and improves the maintainability of test scripts by separating test cases and page-specific code [7], [8]. From the above results, the programming approach with the page-object pattern is suitable for software repeatedly released in a short period. However, implementing highly maintainable test scripts by using the programming approach requires skilled developers. Therefore, the difficulty of test script implementation hinders the introduction of end-to-end automated testing into software development. To solve this problem, Stocco et al. [9] proposed an approach to automatically generate page objects. This approach generates page objects by crawling applications, but it is a challenge to crawl large-scale applications and generate complete page objects.

We propose an approach to automatically generate a test script that uses the page-object pattern from manual testing logs. The key point is that test scripts can be generated by simply performing manual tests without users being aware of test automation. Automated testing is becoming more commonly used in industry, but not all testing can be automated. This is because many development projects hesitate to introduce test automation due to the high implementation cost, and automatic verifications are difficult in some tests such as user-interface testing, usability testing, and so on. Manual testing approaches can be broadly categorized as scripted testing or exploratory testing [10], [11], [12]. Scripted testing is an approach with which developers design tests in advance by considering what kind of test should be conducted and execute them in accordance with the test design. Developers often document the test design and if necessary, make test-procedure manuals that describe detailed procedures for the tests. On the other hand, exploratory testing is an approach with which developers do not design tests in advance but conduct test execution, test design, and learning simultaneously. Exploratory testing cannot currently be replaced by automated testing because testers' expertise is important for it.

The proposed approach can generate a test script almost without any prior preparation by recording manual testing, which is indispensable in software development. Our approach can solve the problems of the manual implementation of test scripts and those with other approaches to generate page objects. With our approach, operations carried out by the tester are regarded as the use of a feature on a web page. Our approach converts the operations into a method of the page object.

The generated test scripts include not only page objects but also test cases that cover features of an application by analyzing page transitions of that application. In general, a test case is a specification of the test and includes a set of operations executed on an application to determine if it satisfies software requirements. Note that test cases generated with our approach are automated. Our approach lowers the barrier to introducing end-to-end test automation to software development. To evaluate the effectiveness of our approach, we asked four testers to conduct manual testing on an open-source web application and evaluated whether our approach generates useful test scripts. A test script here refers to a set of page objects and test cases. The experimental results indicate that our approach can generate a greater number of complete methods in page objects than a current approach for page-object generation. We also conducted an empirical evaluation of whether our approach can reduce the cost of implementing test scripts for real systems. The result showed that our approach reduces about 48% of the time required to implement test scripts compared with manual implementation. Contributions of this paper are as follows:

- We propose a technique to generate automated test scripts with page objects from manual testing activities, which is indispensable in software testing.
- We evaluated whether the generated test scripts were complete or not and showed that our approach can generate a greater number of complete page objects than an existing page object generation technique.
- Our empirical evaluation showed that the costs of test script implementation can be reduced more than with other practical approaches used in real-world software development.

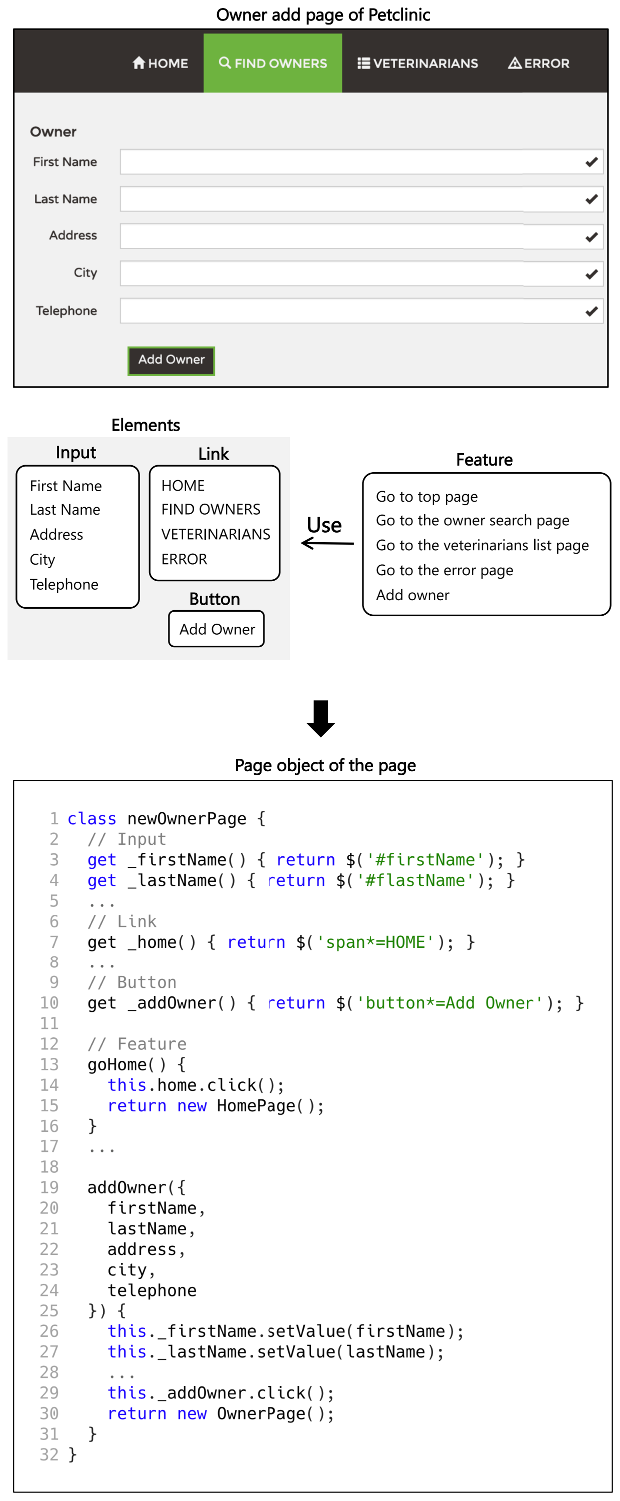
2. Page Object

A page object is a representation of each web page as an object in object-oriented programming. In this study, we define a page object as a class that contains accessors and methods. An accessor obtains a reference to a web element specified by a locator. The body of a method is a sequence of operations such as clicking web elements, entering a value to input fields, or selecting an item from drop-down lists, and the method returns the page object of the destination page. Web elements to be operated in the methods are specified using an accessor.

Figure 1 shows an example of the owner add page in an open-source web application PetClinic [13] and its page object. The owner add page consists of five input fields, four links, and one button. The page also has a feature of transition to another page or adding an owner. The page object in Fig. 1 is implemented in JavaScript with WebdriverIO [14], a test-automation framework for web or mobile applications that makes the test code description more concise than plain Selenium webdriver. The `_firstName` accessor indicates an input field for a first name. `$('#firstName')` captures a web element, the id of which in HTML is "firstName" on the web page. Information for uniquely identifying a web element on the web page, such as `#firstName`, is called a locator. Users can use id, name, text, XPath, etc. as a locator. The defined accessors are only called from methods within the page objects. The *addOwner*() method in Fig. 1 receives values to be inputted in each input field as arguments. This method inputs the values in each input field then clicks the add owner button. When we write test cases to carry out an operation on the owner addition page, we use methods defined in the page object. The return value of the method is generally the page object of the destination of the page transition. This enables us to write test cases using the method chain. The page-object pattern makes it possible to separate test cases and the page-specific code due to modularizing operations and locators. Thus, the page-object pattern can minimize changes to test cases because it is only necessary to modify accessors or methods in the page object when modifying web pages or features under test.

3. Related Work

Stocco et al. [9] proposed APOGEN to generate page objects automatically. It generates page objects by crawling web applications under test and automatically extracts web elements from the pages. APOGEN clusters the web pages on the basis of similarity and integrates the pages belonging to the same cluster into one page object. If multiple pages are functionally similar, the page object should integrate the pages. This is because multiple similar page objects reduce the modularity of the test script, which



```
class newOwnerPage {
  // Input
  get _firstName() { return $('#firstName'); }
  get _lastName() { return $('#flastName'); }
  ...
  // Link
  get _home() { return $('span*=HOME'); }
  ...
  // Button
  get _addOwner() { return $('button*=Add Owner'); }

  // Feature
  goHome() {
    this.home.click();
    return new HomePage();
  }
  ...

  addOwner({
    firstName,
    lastName,
    address,
    city,
    telephone
  }) {
    this._firstName.setValue(firstName);
    this._lastName.setValue(lastName);
    ...
    this._addOwner.click();
    return new OwnerPage();
  }
}
```

Fig. 1 Owner add page of PetClinic and its page object.

would undermine the strength of the page object.

APOGEN can reduce the cost of implementing page objects but has two problems regarding page-object generation. The first problem is that APOGEN requires a certain amount of preparation to use. When an application requires specific input values for page transitions, it is necessary to teach the crawler the locators of input fields and the input values for them in advance. In addition, if APOGEN does not propose the proper cluster, the users need to fix the cluster manually. Our approach, however, requires almost no preparation because it uses logs of manual testing, which are indispensable for software development. Chen et al. [15] improve the accuracy of the page clustering for page-object generation by considering CSS styles and the attributes of web elements, but the problems caused by crawling have not been solved.

The second problem is the completeness of page objects generated with APOGEN. If the crawling cannot cover certain web pages, APOGEN cannot generate page objects for the missing web pages. APOGEN generates methods that operate web elements enclosed in `Form` tags as a feature of the web page, but this technique may not be applicable on some web pages. This is because using features of web pages is not always equivalent to operating web elements enclosed in `Form` tags. In addition, because APOGEN converts all possible page transitions into methods, it would generate too many methods that are not used in the actual test if the application has many links or buttons. On the other hand, our approach can accurately extract features from pages regardless of the page structure by using tester operations, so it is likely to generate useful methods in automated testing. Moreover, APOGEN generates only page objects, but our approach can also generate test cases that leverage the page objects.

Yandrapally et al. [16] proposed an approach to modularizing test scripts automatically to improve the maintainability of test scripts generated by a record and replay tool. Their approach identifies operations to be modularized by analyzing the test scripts and the document-object model of an application. Their experimental results indicated that the number of steps can be reduced by 49–75% by converting parts of the test scripts into a subroutine. However, this approach does not take into account how the subroutines follow the actual use cases of the application.

Crawling-based techniques for end-to-end test-script generation have been proposed to minimize the cost of end-to-end testing [17], [18], [19]. These techniques are used to generate test scripts that cover all features of an application by dynamic exploration. However, test scripts generated with these techniques are not complete and require modifications such as adding assertions by the developer, and the maintainability is not taken into consideration. Thus, it is difficult to incorporate them into continuous development as they are. Although our approach may also require some modifications to generated test scripts, it is easy to modify them due to page objects. GUI ripping is the approach to traverse GUIs automatically and generate their model for regression testing [20], [21]. This approach also could have the same problem as crawling-based techniques.

4. Proposed Approach

Figure 2 shows an overview of our approach. It requires manual testing logs as input and outputs test cases and page objects. The generated test cases use the page objects and call methods declared in the page objects. To record manual testing activity, we developed a tool [22] to obtain test logs that consist of operation data. Such data include information of tags, attributes of operated web elements, types of operations (click or input), and page titles/URLs where the operation is carried out. Testers can record the data without being aware of the existence of the tool during a test. **Figure 3** shows a datum of an operation when a pet's birthday is entered on the pet add page of PetClinic. Our approach

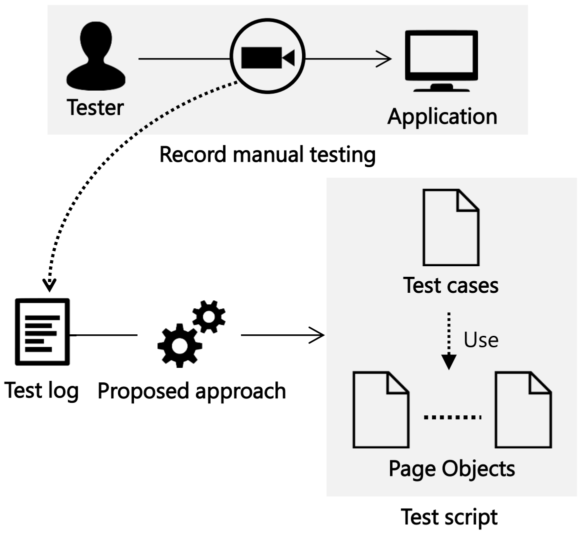


Fig. 2 Overview of proposed approach.

```
"pageInfo":{
    "title":"PetClinic :: a Spring Framework
demonstration",
    "url":"http://localhost:8080/owners/1/pet
s/new"
},
"operation":{
    "type":"input",
    "input":"2020/3/3",
    "elementInfo":{
        "tagname":"INPUT",
        "text":"",
        "xpath":"/HTML/…/DIV/INPUT",
        "attributes":{
            "class":"",
            "id":"",
            "name":"birthdate",
        }
    }
}
```

Fig. 3 Datum of operation in test log.

generates page objects and test cases by using the page objects from test logs and consists of two phases: page-object generation and test-case generation. In the page-object-generation phase, our approach generates page objects by using the data of operated web elements and those of the operation procedure. In the test-case-generation phase, our approach selects test cases to cover all page transitions by analyzing page transitions obtained from test logs and constructs test cases that leverage the page objects.

4.1 Page-object Generation

The proposed approach generates page objects of all web pages visited during a test. Some web applications (e.g., single-page applications) do not have obvious page transitions, so we clarify the definition of a page. In this study, testers could select a title match or URL match as the definition of page equality. They also could use regular expressions and regard web pages that have titles or URLs matching the regular expressions as the same.

The following describes the content of the generated page object. Accessors in the page objects access web elements operated at least once during a test. The accessors return web elements

Algorithm 1: Method generation for page objects.

```
   Input: a test log
   Output: methods for each page object
1  page set $P \leftarrow \emptyset$;
2  foreach operation in the test log do
3      add the page where the operation is executed on $P$;
4  end
   /* Now we have pages $p_1, p_2, \ldots, p_n$                */
5  foreach page $p_i$ in $P$ do
6      let $S_{p_i}$ is the operation sequences for $p_i$;
7      $S_{p_i} \leftarrow \emptyset$;
8  end
9  operation sequence $s \leftarrow \emptyset$;
10 foreach operation in the test log do
11     add the operation to $s$;
12     if there is a page transition, and the previous operation is
         executed on $p_i$ then
13         add $s$ to $S_{p_i}$ for the page;
14         $s \leftarrow \emptyset$;
15     end
16 end
17 foreach page $p_i$ in $P$ do
18     let the sequences adopted as methods for the page object of $p_i$ be
         $M_{p_i}$;
19     $M_{p_i} \leftarrow \emptyset$;
20     sort $S_{p_i}$ in descending order by length;
21     foreach operation sequence $s$ in $S_{p_i}$ do
22         if $s$ is not included in any other $s' \in M_{p_i}$ then
23             add $s$ to $M_{p_i}$;
24         end
25     end
26     convert $M_{p_i}$ to methods;
27 end
```

specified by locators via a function of WebdriverIO. To improve the robustness against application modification, we use locators with priority in the order of (i) id, (ii) name, (iii) text, and (iv) absolute XPath. This is because XPath locators are known to change more frequently than other locators. Text locators are only used for web elements that can contain texts in them (e.g., <a> and <button>). The text locators identify web elements by whether the link text and inner text match the given string.

In the page-object pattern, a method contains a sequence of operations in a web page and represents a feature provided by the page. An operation o is defined as

$$o = \langle t, i, e, p \rangle$$

where t is a type of operation (input or click), i is an input value, e is an operated web element, and p is a web page where the operation is carried out. We regard a sequence of operations carried out from the time a tester comes to a certain page until the time they left as use of a certain feature of the page. We call such operations an *operation sequence*. Algorithm 1 describes the detailed algorithm to generate methods for page objects. We need to first obtain the page set that testers visited by analyzing a test log (lines 1–4). Suppose we have pages $p_1, \ldots, p_n$. We next extract operation sequences carried out on each page as the candidates of the methods (lines 5–16). Let S_{p_i} be the operation sequences for p_i. By scanning the test log, we can retrieve operation sequences carried out on p_i.

If we converted all operation sequences into methods, many duplicate methods would be generated. Therefore, we suppress the generation of duplicated methods by rejecting operation sequences when an operation sequence is included in one of the other operation sequences (lines 17–27). This process aims to generate only versatile methods. For example, we can replace not operating an input field with the operation of entering an empty string into the input field. Let us define operation sequence s_1 and E_{s_1} as the set of web elements operated in s_1 and define s_2 and E_{s_2} in the same manner. We assume that "operation sequence s_2 includes s_1" means that the destination of s_1 and s_2 are the same, and E_{s_2} includes E_{s_1}. For example, suppose that a web page has web elements $e_1, \dots, e_4$, and let us set $E_{s_1} = \{e_1, e_2, e_4\}$ and $E_{s_2} = \{e_1, e_2, e_3, e_4\}$, where E_{s_2} includes E_{s_1}. Also suppose that the destinations of s_1 and s_2 are the same. In this case, s_1 is not adopted as a method because s_2 includes s_1. We take only operated web elements into account regardless of the input value to determine the inclusion. Let M_{p_i} be the operation sequences adopted as methods in the page object for p_1. Finally, the algorithm converts each operation sequence in M_{p_i} into JavaScript code that uses the APIs of WebdriverIO. Since the algorithm converts operation sequences into methods in this manner, the roles of the generated methods are not likely to overlap.

We next present how to determine identifiers of classes, accessors, and methods in the page objects. Class names are determined by the title or URL used to define the page. When we use regular expressions to define web pages, each web page can have a user-defined alias. Accessor names are determined by the id, name, or text of the web element. Method names are "go<class name of the destination>" when the method clicks a link at the end of it; otherwise, it is "do<accessor name called lastly>". If the name of the generated identifier conflicts with other identifiers, this algorithm adds a serial number to the end of the identifier name.

4.2 Test-case Generation

The proposed approach generates not only page objects but also test cases using the page objects. Our approach first determines paths of page transitions to be checked in each test case (a path is represented as a sequence of pages). A test case is constructed by combining methods defined in the page objects and is executed along with one of the determined paths. We note that our test case generation algorithm does not take into account the states of the target application, so the tests are not always executable. This limitation is discussed in Section 4.3.

The following presents the algorithm to determine paths of page transitions. Our approach selects paths that satisfy the following rules:

(1) The path covers all page transitions checked during manual testing.

(2) If the same pages are visited twice in one path, subsequent pages will not be visited.

(3) Page transitions executed in the other paths are not executed as much.

Figure 4 shows an example of path selection. Web pages $p_1, \dots, p_5$ and page transitions among them are shown. In this case, the rules determine the two paths:

$$\text{Path}_1 = [p_1, p_2, p_4, p_5, p_2],\ \text{Path}_2 = [p_1, p_2, p_3, p_5].$$

Here, Path$_1$ and Path$_2$ obviously satisfy the first rule. Next, p_2 is the last page of Path$_1$, and subsequent pages are not visited. We can see that Path$_1$ follows the second rule. p_5 is the last page of Path$_2$, and the page transition from p_5 to p_2 is not executed. Since Path$_1$ has already passed through the page transition from p_5 to p_2, the third rule rejects the page transition. On the other hand, both Path$_1$ and Path$_2$ pass through the page transition from p_1 to p_2. The page transition is not rejected by the third rule. This is because the page transition from p_1 to p_2 is necessary to cover all page transitions by two paths.

Algorithm 2 describes the algorithm for test-case generation. First, our approach analyzes test logs to generate a page-transition

Algorithm 2: Test-case generation.

```
Input: A test log and page objects
Output: Test cases
1  path_list ← ∅;
2  path_c ← ∅;
3  add the start page to path_c;
4  Construct page transition diagram from the test log;
5  Function breadthFirstSearch():
6      queue (of path) ← ∅;
7      Enqueue path_c to queue;
8      while queue is not empty do
9          path_c ← Dequeue from queue;
10         if all destinations from the last page of path are included in
             path_c then
11             Cut off the redundant page transitions at the end of
                 path_c;
12             if path_c includes undiscovered page transitions then
13                 Add path_c to path_list;
14             end
15         end
16         foreach p_a ← adjacent page of the last page of path_c do
17             if p_a is not included in path_c then
18                 path′ ← path_c with p_a appended;
19                 Enqueue path′ to queue;
20             end
21         end
22     end
23 end
24 breadthFirstSearch();
25 foreach path in the path_list do
26     Convert path to a test case that consists of chained methods;
27 end
```

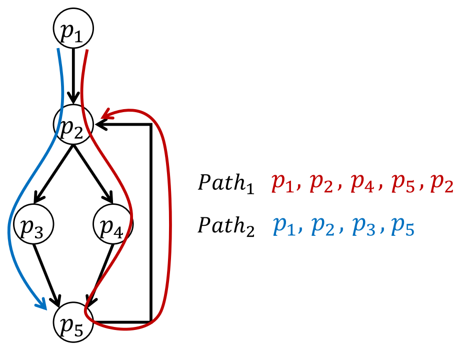
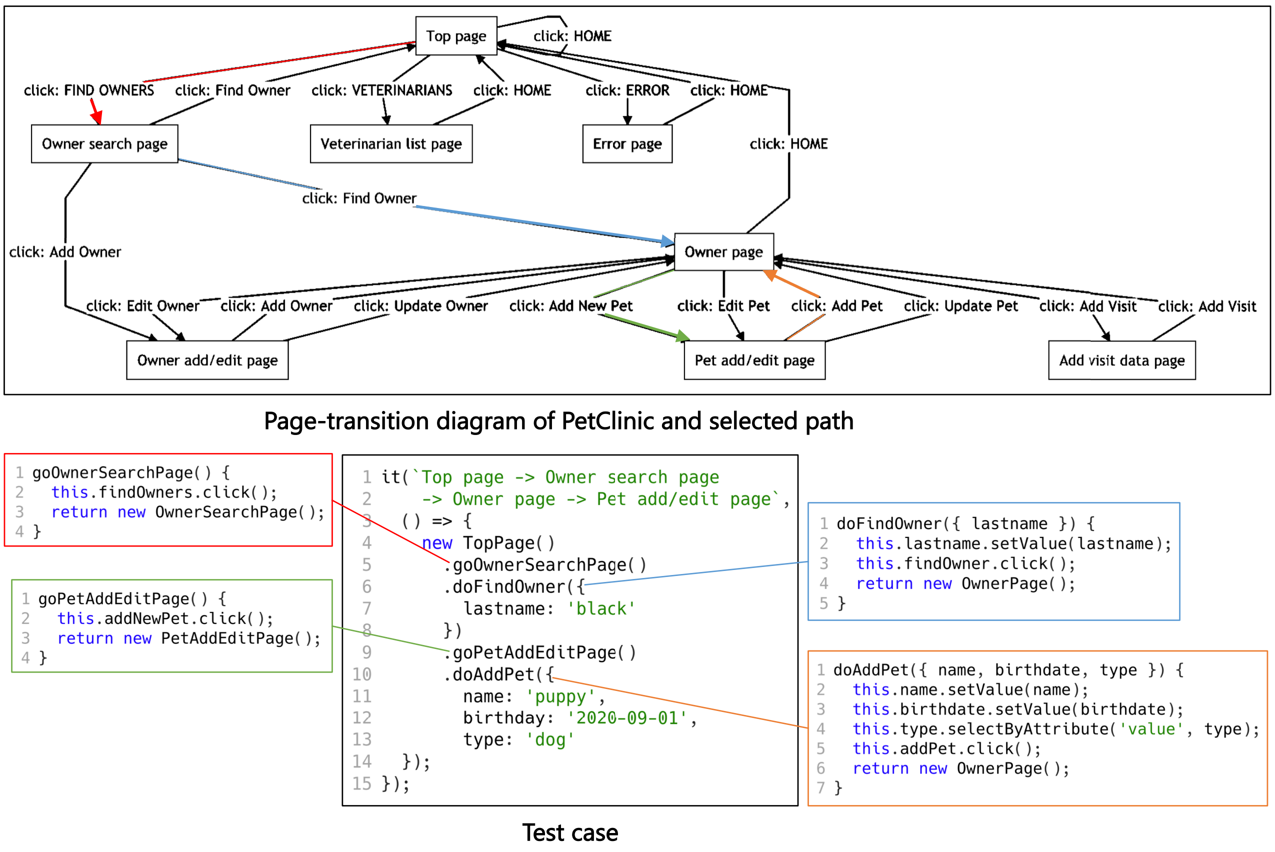

p_1
p_2
p_3
p_4
p_5
$Path_1$ p_1, p_2, p_4, p_5, p_2
$Path_2$ p_1, p_2, p_3, p_5

Fig. 4 Example of path selection for test-case generation.



```
goOwnerSearchPage() {
  this.findOwners.click();
  return new OwnerSearchPage();
}
```

```
goPetAddEditPage() {
  this.addNewPet.click();
  return new PetAddEditPage();
}
```

```
it(`Top page -> Owner search page
    -> Owner page -> Pet add/edit page`,
  () => {
    new TopPage()
      .goOwnerSearchPage()
      .doFindOwner({
        lastname: 'black'
      })
      .goPetAddEditPage()
      .doAddPet({
        name: 'puppy',
        birthday: '2020-09-01',
        type: 'dog'
  });
});
```

```
doFindOwner({ lastname }) {
  this.lastname.setValue(lastname);
  this.findOwner.click();
  return new OwnerPage();
}
```

```
doAddPet({ name, birthdate, type }) {
  this.name.setValue(name);
  this.birthdate.setValue(birthdate);
  this.type.selectByAttribute('value', type);
  this.addPet.click();
  return new OwnerPage();
}
```

Fig. 5 Example of generated test case and methods that it calls.

diagram as far as visited pages in the test (line 4). Next, we obtain a list of paths satisfying the above three rules by carrying out a breadth-first search to the page-transition diagram (lines 5–21). Depth-first search is also a well-known graph-traversal algorithm, but in this case, a breadth-first search is superior. The reason is that a breadth-first search determines the shorter paths first, thus contributing to generating concise test cases. Let the start page of all paths be the start page of the manual testing. When no more page transitions are possible due to the second rule, we cut off the redundant page transitions at the end of the path because of the third rule. A test case consists of chained methods defined in the page objects and executes the page transitions following the path (lines 16–18). For example, suppose that there are three pages p_1, p_2, and p_3. When a path $[p_1, p_2, p_3]$ is converted to a test case, the test case first calls a method defined in the page object of p_1 to go to p_2. The return value of the first method is the page object of p_2; hence, the test case then calls a method defined in the page object of p_2 to go to p_3. In this manner, our approach generates test cases that satisfy the rules by converting each path into a test case. Note that if multiple methods execute the same page transition in one page object, the first generated method is used in the test case. Our approach can also generate arguments and input values for the method. This is because the test logs contain input values when each page transition is carried out by testers.

Figure 5 shows an example of a generated test case and methods called from the test case. The page-transition diagram and test case is a part of the output in our experiment using PetClinic described in Section 5. Suppose that we obtain a path that transitions in the following order: the top page, owner search page, owner page, and pet add/edit page. In this case, our approach generates a test case that consists of four methods executing the page transitions. Each method is declared in different page objects. The test cases start from the page object of the top page, and the page object has the *goOwnerSearchPage*() method. Next, *goOwnerSearchPage*() returns the page object of the owner search page, and the page object calls the *doFindOwner*() method. By repeating the same steps, the test case is built. If methods require arguments, the proposed approach extracts a set of input values that caused the required page transition from the test log.

4.3 Limitation

Because our approach does not take into account the state of a target application, it may generate test cases that need to set the application to a certain state in order to execute them. To execute such test cases, we need to insert a process to initialize the database before executing the test cases or to modify input values given in generated test scripts. The problem of state dependency may hinder the practical use of our technique. This problem is also common with most crawling-based test generation techniques and record & replay tools and is out of the scope of this study. There are several pieces of research [23], [24] working on dependency-aware test generation, and these could be used to solve this problem. If the techniques proposed in these studies are not introduced, users will have to modify generated test scripts to solve the state-dependency problem. However, this problem is partially mitigated by the fact that test scripts that use page objects are easier to modify than those that do not. One of the motivations of this research is to generate test scripts that are easy to modify, as it is difficult to generate perfect test scripts for users.

Our approach limits target applications and situations that can be applied. The proposed approach currently cannot generate any assertions; hence, users need to insert assertions to check that the test scripts are correctly executed. However, it is technically feasible to verify that the currently opened web page matches the expected one because we record titles and URLs of web pages where operations are carried out. Moreover, the generated test scripts leverage the page-object pattern and have high maintainability. Therefore, it is even easy to add more detailed assertions.

Our recording tool is currently unable to record operations other than click and input. Hence, it is not possible to carry out any other operations (e.g., drag, mouse hover, etc.) in the generated test scripts. In addition, our approach cannot generate methods including operations that were not performed in manual testing. However, we believe that we can make up for the lack of test logs by keeping logs of not only planned manual testing, but also a little behavior verification and smoke testing.

5. Evaluation

We conducted experiments to determine if our approach can generate effective test scripts. In particular, we address the following research questions.

RQ1. Can our approach generate a greater number of complete methods in page objects than a current approach, i.e., APOGEN?

RQ2. Can the test cases generated with our approach be used without modification and cover the features of the application?

RQ3. Does our approach reduce the initial cost of test script implementation compared to practical approaches used in industry?

5.1 Experimental Setup

Four participants as testers conducted manual testing on Spring PetClinic version 2.2.0, which is an open-source web application. Spring PetClinic is a sample application of the Spring Framework, but it is non-trivial and has more than 6k lines of Java code. **Table 1** lists all web pages and features of PetClinic. Note that we used the database prepared by PetClinic as the initial state. All web pages of PetClinic have a header that contains links to go to the top page, owner search page, veterinarians list page, and error page. We regard the owner add page and owner edit page as the owner add/edit page because these two pages are generated from the same template file of the Spring Framework and have almost the same structure. For the same reason, we also regarded the pet add/edit page as a single page. In addition, each owner has a separate owner page in PetClinic, but we also regarded the owner pages as a single page. Since these page objects must be modified in the same way frequently when the template is modified, we believe it would be better to separate these page objects to experiment in a practical setting.

We used two manual testing approaches to determine whether our approach does not depend on the manner of manual testing. In this experiment, two testers conducted scripted testing, and the other two conducted exploratory testing. These four testers all had over three years of experience in testing, and the two conducting exploratory testing had experience in doing it.

We first had the testers operate PetClinic to grasp its specifications. We assumed that unit tests of PetClinic were sufficiently conducted on both the server-side and client-side as a premise of the experiment. Next, we instructed the testers in how to conduct end-to-end testing on PetClinic regarding functionality and usability. Although PetClinic is a stable application, we asked the testers to test it with the intention of finding bugs. The two testers who conducted scripted testing designed and documented the content of tests as test scenarios in advance before conducting the tests in accordance with the test scenarios. A test scenario here means a sequence of procedures to check a use case of the target application. The other two testers conducted exploratory testing for up to 30 minutes to find bugs by using their knowledge and experience.

All operations carried out in the tests were recorded with our tool explained in Section 4. Let the test logs obtained from testers A–D be test logs A–D, respectively. **Table 2** shows the summary of the tests the testers conducted. The cases of exploratory testing had no documented test scenario because the two testers did not design tests in advance. The number of operations equaled the number of click events and change events that occurred when the testers operated web elements. We applied our approach to the test logs and generated four sets of test scripts. We also merged the four test logs and created one large test log that was equal to four tests conducted consecutively. We also applied our approach to the merged test log in the same manner. The generated test scripts from our approach and APOGEN are publicly available [*1]

5.2 Page-object Generation

We compared our approach with APOGEN to evaluate whether the page-object-generation phase of our approach was able to

Table 1 Web pages and features of PetClinic.

Web page	Feature
Top page	Nothing
Owner search page	Search owners by a last name
Owner search result page	Show the list of owners hit by a search
Owner add/edit page	Input owner data and add or update the owner
Owner page	Add or edit pet data of the owner and add visit data for the pet
Pet add/edit page	Enter pet data and add or update the pet
Visit data add page	Enter visit data for the pet and add them
Veterinarians list page	Nothing
Error page	Nothing

Table 2 Summary of tests by testers.

Tester	Approach	# of test scenario	# of operations
A	Scripted testing	9	135
B	Scripted testing	20	258
C	Exploratory testing	–	378
D	Exploratory testing	–	505

[*1] https://zenodo.org/record/5655786

Table 3 Classification of methods in page objects generated with our approach and APOGEN (the values in parentheses are counted by including the number of methods in the page objects that could not be generated by APOGEN).

Source	Complete	Redundant	To modify	Unnecessary	Header	Total
APOGEN	5	0	6	0	18	31
Test log A	6 (9)	0	3	0	6 (7)	13 (17)
Test log B	8 (12)	0	7	1	8 (12)	24 (32)
Test log C	6 (10)	0	4	6	12 (16)	28 (36)
Test log D	9 (13)	1	6	3	10 (13)	29 (36)
Merged log	12 (16)	0	3	11	18 (23)	44 (53)

generate complete methods. We first examined page objects generated with the proposed approach and APOGEN. PetClinic has nine pages, as shown in Table 1. Note that we define owner pages, owner add/edit page, and pet add/edit page each as one page by regular expression of the URLs. This is because the pages are generated from the same template, as explained in Section 5.1. Therefore, the proposed approach generated nine page objects from each test log. We then applied APOGEN to PetClinic to generate page objects. We gave information to APOGEN's crawler to reach as many pages as possible and classify reached web pages into the pages shown in Table 1 by manual clustering. However, we could not generate page objects for the error page and owner search page due to the limitation of APOGEN. This result is the same as in the evaluation of an existing paper [9]. Thus, we obtained seven page objects with APOGEN, except for the two pages that could not be reached.

We then classified the methods in the page objects in accordance with the following criteria:

Complete Methods have no parts to be modified for arguments, operations, and return values.

Redundant Methods function correctly but contain meaningless operations that do not affect their functionality.

To modify Methods require modification of any of the arguments, operations, and return value in order to use it.

Unnecessary The second or subsequent methods of multiple methods that check the same page transitions.

Header Methods click links on the header to go to another page.

Although methods classified as *header* are all *complete* methods, we decided to distinguish *header* from the others. This is because they are unlikely to be used in actual test cases despite the large number.

Table 3 shows the results of the classification. The table shows both the case where we count only the methods in the seven page objects generated by APOGEN and the case where the two page objects that APOGEN could not generate are included. Our approach generated a greater number of complete methods than APOGEN, even if we excluded the page objects of web pages that APOGEN could not reach. It also generated one *redundant* method *goOwnerSearchPage*() for test log D. This is because tester D carried out the operation sequence to click a link to go to the owner search page after filling in an input field on the owner add/edit page. The operation sequence is converted to the method, but the operation of filling an input field is not necessary to go to the owner search page.

The *to modify* methods were generated with both APOGEN and our approach, but our approach tended to generate fewer. There was no correct page object as a return value in four *to modify* methods generated with APOGEN probably because APOGEN does not take into account the case where different page transitions are performed depending on the input values when using the same feature. The other two *to modify* methods by APOGEN lack operations to enter values when updating information of pets or owners. On the other hand, the proposed technique was able to generate these methods that APOGEN was not able to generate correctly. APOGEN converts operations on web elements enclosed in FORM tags into a method, but PetClinic did not have such a set of web elements. Hence, there was no sequence of operations that the proposed technique could recognize but APOGEN could not.

Most of the *to modify* methods generated with our approach have an insufficient number of arguments and cannot enter values to some input fields. This is because testers did not fill in all input fields in some pages during the tests. If testers conducted a test that attempts to register an owner with a blank name, the generated method did not include the operation to fill in the name input field due to the method-generation algorithm of our approach. However, there are other possible inputs to cause registration to fail, such as not giving an address and inputting incorrect characters. The methods should always have arguments for all inputs to register an owner because missing arguments reduce versatility. If there are no missing arguments and users want to register an owner with a blank name, the users can achieve the operation to give an empty string as an argument.

Our approach generated *unnecessary* methods that click different owners on the owner search result page separately in most cases. Since these methods have the same destination, the second and subsequent methods are classified as *unnecessary*. The *unnecessary* methods were only generated with our approach. However, if APOGEN reached the owner search result page, APOGEN would generate many methods to click each owner and generate more *unnecessary* methods than our approach.

Our approach generated fewer *header* methods, even though it generated more page objects. Most of the methods classified as *header* are not important and would not be used because developers usually just need to make sure the links are valid.

Page objects generated from the merged log had the most *complete* methods and the least methods *to modify*. The reason is that each log fills in the missing operations. Our approach converts operation sequences that include other small operation sequences into methods. Thus, even if a log has an operation sequence that does not operate on all input fields, our approach generates a *complete* method if all input fields are operated on in the other logs. On the other hand, the object generated from the merged log gen-

Table 4 Classification and average length of generated test cases.

Source	Complete	Data-dependent	To modify	Total	Avg. length
Test log A	7	0	1	8	4.25
Test log B	11	0	1	12	4.00
Test log C	10	4	0	14	4.58
Test log D	11	0	2	13	4.50
Merged log	19	0	1	20	4.46

erated many *unnecessary* and *header* methods. This is because the merged log includes many operations of clicking various owners in the owner search result page and clicking links in the header on each page. However, since this problem is largely due to the specification of PetClinic, the problem may not occur in other applications.

To summarize the evaluation of page-object generation and *answer RQ1*, our approach is more likely to generate a greater number of complete methods compared with APOGEN, regardless of the manual testing approach or testers. Generating page objects from a merged log can compensate for the incompleteness of each log but increases the number of extra methods that are likely to be unused.

5.3 Test-case Generation

We next evaluated whether the test-case-generation phase of our approach was able to generate complete test cases. It depends on the project as to what test cases should be automated, but in this experiment, we evaluated whether our approach could generate test cases that check the normal scenario of each feature and do not require modifications. The reason is that such test cases are versatile and can be useful in any project. Moreover, it is easy to implement test cases for exceptional scenarios (e.g., cases in which owner registration fails) by reusing generated test cases and page objects. We classified test cases generated from test logs A–D in accordance with the following criteria.

Complete A test case can be executed without modifying the order of method calls, argument values, and the database state.

Data-dependent A test case can be executed by changing the database state from the initial state or changing the value given by the method argument.

To modify A test case can be executed by replacing some called methods with other methods that have the same transition destination as before.

Table 4 lists the results of the classification of generated test cases and the average length of the test cases. The length of a test case means the length of paths of page transitions checked in a test case. It also equals the number of called methods in a test case because a method call invokes a page transition.

The reason why four test cases of test log C were classified as *data-dependent* is that tester C added a pet to an owner registered during manual testing. Since our approach does not take into account the state of applications as explained in Section 4.3, it generated test cases that add a pet to an owner who does not exist in the initial state of the database. The testers other than C tested the edit feature only for users and pets that are registered by default, so this problem did not occur when using logs other than C. We found that whether or not our technique generates *data-dependent* test scripts depends on the way of testing. *Data-dependent* test cases become *complete* when we replace the method call in the test case with the method to click an initially existing owner or when we change the initial state of the database.

Some test cases were classified as *to modify* because some web pages with different features are defined as one page. For example, adding and editing pets are different operations, but we define the pet add/edit page as one page because the templates of the pages are the same. The method clicking the “Add Pet” button after filling in input fields and the method clicking the “Update Pet” button after that are declared as different methods in the page object. However, our approach did not distinguish the methods when constructing test cases because both methods go to the owner page from the pet add/edit page. As a result, our approach may generate test cases that call the method to update a pet when the method to add a pet should be called. In this case, the test case becomes *complete* if we replace the method call to update a pet with that to add a pet.

Table 5 shows what features were checked from the test cases generated from test logs A–D and merged log (labeled “M”). The features of PetClinic were extracted from the test-case specifications written by testers A and B. In the table, a “✓” indicates that the generated test cases checked the feature, “×” indicates the generated test cases did not check the feature even though the manual test checked it, and “–” indicates that the generated test cases could not check the feature because the manual test did not check the feature. Due to the limitation of our approach, it is not able to generate test cases checking the feature that the manual tests did not check. We assume that one test case can confirm multiple features. For example, we have a test case that adds a pet to an owner found in the owner search after moving from the top page to the owner search page. In this case, we determine that the test case confirms features (1, 6, 12) in Table 5. Note that Table 5 shows the result when the *data-dependent* test cases and *to modify* ones were correctly modified and became complete.

Some features were not checked by the generated test cases although the manual tests checked the features. In most cases, this was due to the fact that our approach generates test cases on the basis of the coverage of page transitions. For example, in a certain test case, if a transition from the owner page to the pet add/edit page was performed by clicking the “Add New Pet” button, the page transition was checked. On the other hand, when the “Edit Pet” button was clicked from the owner page, the transition to the pet add/edit page was also performed. However, since this page transition had already been checked, our approach did not generate the test case to check the feature to edit pet. As a result, for

Table 5 Features confirmed from generated test cases.

Web page	#	Feature	A	B	C	D	M
Owner search page	1	If one hit is made in the owner search, the owner page will be displayed.	✓	✓	✓	✓	✓
	2	If two or more hits are made in the owner search, they will be displayed on the owner search result page.	–	✓	×	×	✓
	3	If nothing is entered in the owner search, all owners will be displayed on the owner search result page.	✓	×	✓	✓	×
	4	Go to the owner add page.	✓	✓	✓	✓	✓
Owner search result page	5	Move to the owner page by clicking an owner name.	–	✓	✓	✓	✓
Owner page	6	Go to the pet add page.	✓	✓	✓	✓	✓
	7	Go to the pet edit page.	×	×	×	×	×
	8	Go to the owner edit page.	✓	✓	✓	✓	✓
	9	Go to visit data add page.	✓	✓	✓	✓	✓
Owner add/edit page	10	Add an owner by filling in input fields.	✓	✓	✓	✓	✓
	11	Edit an owner by filling in input fields.	×	×	×	×	×
Pet add/edit page	12	Add a pet by filling in input fields.	✓	✓	✓	✓	✓
	13	Edit a pet by filling in input fields.	×	×	–	×	×
Visit data add page	14	Add visit data by filling in input fields.	✓	✓	✓	✓	✓
Header	15	Go to the top page, owner search page, veterinarians list page, or error page.	✓	✓	✓	✓	✓

Table 6 Summary of the test scenarios for empirical evaluation.

	System	# of test procedures	# of involved pages	Description
Scenario 1	A	3	4	Check data query feature
Scenario 2	A	2	14	Check data update feature
Scenario 3	A	2	14	Check data lifecycle
Scenario 4	B	2	6	Check standard operation procedures

the pairs of features (2, 3), (6, 7), (10, 11) and (12, 13) in Table 5, only one feature of each pair was checked. However, we believe that we can easily create test cases to check another feature of each pair by slightly modifying the generated test cases.

Finally, we discuss the smallness and simplicity of generated test cases. Table 5 shows that the average length of the test cases was at most 4.58. This indicates that each test case is concise and that users can easily understand the test cases. The interesting point is that test log D had about twice as many operations as test log B, and testers B and D adopted different manual testing approaches, yet the numbers of test cases were almost the same. Since generated test cases depend on the page-transition diagram obtained from manual tests, our approach has the advantage of generating similar test cases no matter how the manual tests were conducted if the page-transition diagrams are similar.

In this experiment, although the test cases generated from test log A were the smallest, the test cases covered most of the features checked in the other test cases. Therefore, we can say that there is redundancy in the test cases generated from the other test logs. This is because the more links on the header are clicked, the more complex the page-transition diagram becomes. Our approach uses the page-transition diagram to make the test cases cover the page transitions executed in tests. However, every page of PetClick has a header, and if testers go to another page by clicking the links on the header, the page transitions are regarded as different. Hence, we found that our approach may generate a redundant set of test cases if applications contain mesh-like page transitions that are interconnected.

To answer RQ2, our approach generated complete test cases in most situations. The generated test cases covered most of the features of the application. However, our approach may generate incomplete or redundant test cases when multiple pages with different features are regarded as the same one or when the application has interconnected page transitions.

5.4 Empirical Evaluation

We evaluated whether our approach is efficient to implementing test scripts using page objects with less cost than existing approaches. As a comparison, we chose to implement test scripts manually and with SeleniumIDE, which are commonly used in real-world software development. Target systems are an internet banking system (System A) and a campaign information management system (System B), which are developed in a real project of a partner company. We prepared three test scenarios for System A and one test scenario for System B. **Table 6** shows the summary of each scenario. Each test scenario has multiple predetermined test procedures.

We asked one developer who belongs to the partner company to carry out tasks that are implementing test scripts to automate these test procedures. The developer is familiar with our approach, SeleniumIDE, and how to implement test scripts with page objects. The developer also has a detailed understanding of the target systems. The condition for task completion is that the developer implements test scripts and confirms that the test scripts automate predetermined test procedures. The test procedures are also complied with when recording the tests with SeleniumIDE and our recording tool. The test script implementation tasks were carried out in the following order.

(i) Manual implementation: The developer implemented test scripts with page objects written in JavaScript.

(ii) SeleniumIDE: The developer recorded tests, exported them as test scripts written in JavaScript, and then rewrote them into test scripts with page objects.

(iii) Our approach: The developer recorded tests, generated test scripts via our approach, and modified them to automate

Table 7 The time to implement test scripts (minute).

	Our approach				SeleniumIDE				Manual			
	Rec[1]	PO[2]	TC[3]	Total	Rec	PO	TC	Total	Rec	PO	TC	Total
Scenario 1	1	7	1	9	1	16	1	18	0	21	1	22
Scenario 2	3	26	2	31	3	41	2	46	0	66	2	68
Scenario 3	3	60	2	65	3	55	2	60	0	94	2	96
Scenario 4	1	5	1	7	1	20	1	22	0	28	1	29
Total	8	98	6	112	8	132	6	146	0	209	6	215

[1] Time to record manual tests
[2] Time to create or modify page objects
[3] Time to create or modify test cases

the predetermined test procedures.
The developer could not divert test scripts implemented in the previous tasks to the later tasks. Carrying out the previous tasks was likely to make the later tasks easier, which may not result in a fair outcome. We will discuss this problem in Section 6.

Table 7 shows how many minutes it took to finish the tasks. The result shows that the proposed approach reduced the time to implementing the test scripts by 48% compared to manual implementation and by 23% compared to using SeleniumIDE. Most of the task time is spent on creating or modifying page objects. The time spent in recording the operations and creating or modifying the test cases was small. The reason why it takes less time to create or modify test cases is that the test cases can be written easily as a combination of methods in page objects, and the number of test cases is small. When using our approach, the largest amount of time (42.3%) was spent on the source code modification for fixing the method in the page objects. The time used correcting locator errors (34.7%) follows this. Other modifications included adding commands to wait for loading web pages, removing unnecessary test steps, and so on.

The reason why the page objects generated by our approach required modifications was due to the complexity of System A. Depending on the internal state of the system, the page transitions may change even if the same operation is carried out. In addition, System A has web pages changing drastically and dynamically by JavaScript. Our approach currently cannot handle such internal states of applications and drastic screen changes. If page objects are not correctly associated with each page, the generated page objects require significant modifications. However, despite the need for modifications to the generated test scripts, the results show that using our approach is more efficient than implementing from scratch. Our approach will be able to solve such problems by making it possible to define screens more flexibly, for example, by defining pages using strings rendered on web pages. Alternatively, it is a reasonable idea to use more advanced screen recognition techniques proposed in several pieces of research [25], [26].

The main reason why the developer needed to fix locators was that the locators generated by our approach did not uniquely identify the web elements in a web page in some cases. Since our approach does not collect any information other than the web elements operated during the manual testing, the generated locators may not be unique. This problem can be solved by considering all web elements in a web page to generate locators during recording manual testing. We believe that these improvements will further reduce the time required to implement test scripts via our approach.

To answer RQ3, our approach has the potential to reduce the cost of test script implementation in real-world software developments. In addition, improving the algorithm of our approach would potentially reduce more costs.

6. Threats to Validity

The external validity of our study concerns the generalization of our findings. First, we used only PetClinic as the target to evaluate the proposed technique. Different results from those in this study may be obtained if we apply the proposed approach to other applications. In this study, we chose PetClinic since it was used in an existing paper [9] to compare our approach with APOGEN. Next, the results of our experiment depended on the content of the testers' manual testing approach. Our experiments showed that our approach can generate a greater number of complete methods and test cases for a variety of testing approaches. However, when other testers conduct manual tests, we may not obtain similar results. In addition, the proposed technique may not work well if manual testing is not sufficiently performed. If the proposed technique is applied to an application more complex than PetClinic, testers may miss features to be tested. Even if the manual testing is sufficient, generated test scripts may not be able to be executed due to the state-dependency problem when some operations in the manual testing depends on past ones.

In the empirical evaluation, only one developer carried out test script implementation tasks. We may obtain different results from this evaluation by having more developers carry out the same tasks. In addition, doing the previous tasks may make the later tasks easier, so it is possible that the time to carry out tasks via our approach is shorter than it should be. We believe that the effect of the previous tasks on the evaluation is small because the tasks assigned to the developer are simple compared to usual test script implementation. In usual test script implementation, developers often implement test scripts through trial and error. In our experiment, on the other hand, the test procedures were predetermined and the developer understood the details of systems under test. Moreover, the developer did not spend time on properly naming identifiers and refactoring, other than converting predetermined test cases into test scripts. This would have made it clearer how to implement test scripts in many parts.

A threat to internal validity is that we defined the web pages and features of PetClinic ourselves and classified the generated page objects and test cases. The definition of the features was based on the test specifications written by testers A and B, so we believe that the definition is objective to some extent. We will make the output of our approach publicly available so that other researchers can verify the results.

7. Conclusion

We proposed an approach to generate test scripts using the page-object pattern from manual testing logs. Through experiments, we showed that our approach solved the problems with current approaches and generated a greater number of complete methods in page objects. Our approach also generated test cases that leverage the generated page objects and covers most of the features of the application under test. The generated test cases and page objects are reusable, and users can add new test cases easily. Our empirical evaluation also showed the potential for reducing the cost of test script implementation in real-world software development. The proposed approach is effective to reduce the cost of implementing and maintaining test scripts by generating useful test scripts through the conducting of only manual testing, which is essential in software development.

For future work, we aim to generate more complete page objects and test cases by converting operation sequences into methods more precisely using the testers' knowledge contained in the test logs effectively. We also would like to make our recording tool publicly available so that everyone can record testing activities. If everyone can obtain test logs, researchers will be able to mine the test logs and use them for various purposes other than test automation.

References

[1] Yang, B., Hu, H. and Jia, L.: A Study of Uncertainty in Software Cost and Its Impact on Optimal Software Release Time, *IEEE Trans. Software Engineering*, Vol.34, No.6, pp.813–825 (2008).

[2] Bertolino, A.: Software Testing Research: Achievements, Challenges, Dreams, *Future of Software Engineering*, pp.85–103 (2007).

[3] Orso, A. and Rothermel, G.: Software Testing: A Research Travelogue (2000–2014), *Proc. Future of Software Engineering*, pp.117–132 (2014).

[4] Selenium, available from ⟨http://www.seleniumhq.org/⟩ (accessed 2021-08-02).

[5] Christophe, L., Stevens, R., Roover, C.D. and Meuter, W.D.: Prevalence and Maintenance of Automated Functional Tests for Web Applications, *IEEE International Conference on Software Maintenance and Evolution*, pp.141–150 (2014).

[6] Leotta, M., Clerissi, D., Ricca, F. and Tonella, P.: Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution, *20th Working Conference on Reverse Engineering*, pp.272–281 (2013).

[7] Leotta, M., Clerissi, D., Ricca, F. and Spadaro, C.: Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study, *IEEE 6th International Conference on Software Testing, Verification and Validation Workshops*, pp.108–113 (2013).

[8] Ricca, F. and Stocco, A.: Web Test Automation: Insights from the Grey Literature, *47th International Conference on Current Trends in Theory and Practice of Computer Science* (2020).

[9] Stocco, A., Leotta, M., Ricca, F. and Tonella, P.: Clustering-Aided Page Object Generation for Web Testing, *Web Engineering*, pp.132–151, Springer (2016).

[10] Ghazi, A.N., Petersen, K., Bjarnason, E. and Runeson, P.: Levels of Exploration in Exploratory Testing: From Freestyle to Fully Scripted, *IEEE Access*, Vol.6, pp.26416–26423 (2018).

[11] Shah, S.M.A., Alvi, U.S., Gencel, C. and Petersen, K.: *Comparing a Hybrid Testing Process with Scripted and Exploratory Testing: An Experimental Study with Practitioners*, pp.187–202 (2014).

[12] Itkonen, J., Mantyla, M.V. and Lassenius, C.: Defect Detection Efficiency: Test Case Based vs. Exploratory Testing, *1st International Symposium on Empirical Software Engineering and Measurement*, pp.61–70 (2007).

[13] PetClinic, available from ⟨https://github.com/spring-projects/spring-petclinic⟩ (accessed 2021-08-02).

[14] WebdriverIO, available from ⟨https://webdriver.io/⟩ (accessed 2021-08-02).

[15] Chen, Y., Li, Z., Zhao, R. and Guo, J.: Research on Page Object Generation Approach for Web Application Testing, *The 31st International Conference on Software Engineering and Knowledge Engineering*, pp.43–48 (2019).

[16] Yandrapally, R., Sridhara, G. and Sinha, S.: Automated Modularization of GUI Test Cases, *Proc. 37th International Conference on Software Engineering*, pp.44–54 (2015).

[17] Iyama, M., Kirinuki, H., Tanno, H. and Kurabayashi, T.: Automatically Generating Test Scripts for GUI Testing, *IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp.146–150 (2018).

[18] Fard, A.M., Mirzaaghaei, M. and Mesbah, A.: Leveraging Existing Tests in Automated Test Generation for Web Applications, *Proc. 29th ACM/IEEE International Conference on Automated Software Engineering*, pp.67–78 (2014).

[19] Dallmeier, V., Pohl, B., Burger, M., Mirold, M. and Zeller, A.: WebMate: Web Application Test Generation in the Real World, *IEEE 7th International Conference on Software Testing, Verification and Validation Workshops*, pp.413–418 (2014).

[20] Memon, A.: GUI ripping: Reverse engineering of graphical user interfaces for testing, *Proc. 10th Working Conference on Reverse Engineering*, pp.260–269 (2003).

[21] Amalfitano, D., Fasolino, A.R., Tramontana, P., Carmine, S.D. and Memon, A.M.: Using GUI ripping for automated testing of Android applications, *2012 Proc. 27th IEEE/ACM International Conference on Automated Software Engineering*, pp.258–261 (2012).

[22] Kirinuki, H., Kurabayashi, T., Tanno, H. and Kumagawa, I.: Poster: SONAR Testing - Novel Testing Approach Based on Operation Recording and Visualization, *2020 IEEE 13th International Conference on Software Testing, Validation and Verification* (*ICST*), pp.410–413 (2020).

[23] Biagiola, M., Stocco, A., Ricca, F. and Tonella, P.: Dependency-Aware Web Test Generation, *2020 IEEE 13th International Conference on Software Testing, Validation and Verification* (*ICST*), pp.175–185 (2020).

[24] Biagiola, M., Stocco, A., Mesbah, A., Ricca, F. and Tonella, P.: Web test dependency detection, *Proc. 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019*, pp.154–164, Association for Computing Machinery (2019).

[25] Roest, D., Mesbah, A. and van Deursen, A.: Regression Testing Ajax Applications: Coping with Dynamism, *2010 3rd International Conference on Software Testing, Verification and Validation*, pp.127–136 (2010).

[26] Yandrapally, R., Stocco, A. and Mesbah, A.: Near-duplicate detection in web app model inference, *Proc. ACM/IEEE 42nd International Conference on Software Engineering*, *ICSE '20*, pp.186–197, Association for Computing Machinery (2020).

Hiroyuki Kirinuki is currently a researcher in the Software Innovation Center at Nippon Telegraph and Telephone Corporation (NTT). He received an M.E. degree in 2015 from Osaka University. He joined NTT in 2015. His research interests include software testing and empirical software engineering. He is a member of the IPSJ.

Haruto Tanno is currently a researcher in the Software Innovation Center at Nippon Telegraph and Telephone Corporation (NTT), Tokyo, Japan. He received an M.E. in 2009 and Dr. Eng. in 2020 from The University of Electro-Communications, Tokyo. He joined NTT in 2009. His research interests include software testing and debugging. He is a member of the IPSJ.