information

MDPI

Article

Implementation Support of Security Design Patterns Using Test Templates

Masatoshi Yoshizawa [1], Hironori Washizaki [1], Yoshiaki Fukazawa [1], Takao Okubo [2], Haruhiko Kaiya [3], Nobukazu Yoshioka [4]

[1] Waseda University, Tokyo Japan ; E-mail : yoshizawa-masa@asagi.waseda.jp (M.Y.); E-mail: washizaki@waseda.jp (H.W.); E-mail : fukazawa@waseda.jp (Y.F.)
[2] Information Security Division Institute of Information Security, Yokohama Japan; E-mail: okubo@iisec.ac.jp
[3] Kanagawa University, Kanagawa Japan; E-mail: kaiya@kanagawa-u.ac.jp
[4] GRACE Center National Institute of informatics, Tokyo Japan; E-mail: nobukazu@nii.ac.jp

Abstract: Security patterns are intended to support software developers as the patterns encapsulate security expert knowledge. However, these patterns may be inappropriately applied because most developers are not security experts, leading to threats and vulnerabilities. Here we propose a support method for security design patterns in the implementation phase of software development. Our method creates a test template from a security design pattern, consisting of an "aspect test template" to observe the internal processing and a "test case template". Providing design information creates a test from the test template with a tool. Because our test template is reusable, it can easily perform a test to validate a security design pattern. In an experiment involving four students majoring in information sciences, we confirm that our method can realize an effective test, verify pattern applications, and support pattern implementation.

Keywords: Security Patterns; Model-based Testing; Test-driven Development; Aspect-oriented Programming;

1. Introduction

Due to the increasing number of business services on open networks and distributed platforms, preventing and resolving security issues is vital [1]. Security concerns must be considered in every phase of software development from requirements engineering to design, implementation, testing, and deployment [2]. However, addressing all security concerns is difficult due to the sheer volume and the fact that not all software engineers are security specialists.

Patterns are reusable packages that encapsulate expert knowledge. Specifically, a pattern represents a frequently recurring structure, behavior, activity, process, or "thing" during the software development process. Many security design patterns have been proposed. For example, Schumacher et al. presented 25 design-level security patterns [3]. Because security design patterns are currently abstract descriptions, they are difficult to implement. Additionally, it is hard to validate security design patterns in the implementation phase because an adequate test case is required. Hence, a security design pattern can be inappropriately applied, leading to serious vulnerability issues.

Herein we propose a method to support the implementation of security design patterns using a test template.

The preliminary version of this paper has been presented at the conference [ARES 2014]. In this paper, we added a tool for creating test and an evaluation of our method.

Our method creates a test template from a security design pattern, which consists of an "aspect test template" to observe internal processing and a "test case template". By providing design information in the test template, a test is created to evaluate the system in the early implementation stage using a tool and fixing the code. As the test can be executed repeatedly, it can validate whether a security design pattern is appropriately applied in the implementation phase.

In this paper, we address the following research questions:

- **RQ 1: Is our method more efficient at finding defects when implementing a security design pattern than an existing method?**
- **RQ 2: Does our method create a more effective test to find implementation defects in a security design pattern than an existing method?**
- **RQ3: Compared to an existing method, does our method more effectively create a test to find implementation defects in a security design pattern and allow for a regression test?**
- **RQ 4: Is our method more effective at correcting implementation defects in a security design pattern compared to an existing method?**

Our contributions are as follows:

- We create a reusable test template derived from security design patterns.
- Implementation defects in a security design pattern are efficiently determined.
- Our method supports the appropriate implementation of security design patterns.

The remainder of this paper is organized as follows. Section 2 provides the background and problem with security design patterns. Section 3 describes related works. Section 4 details our proposed method. Section 5 shows a case study using our method. Section 6 discusses the evaluation results. Finally, Section 7 concludes the paper.

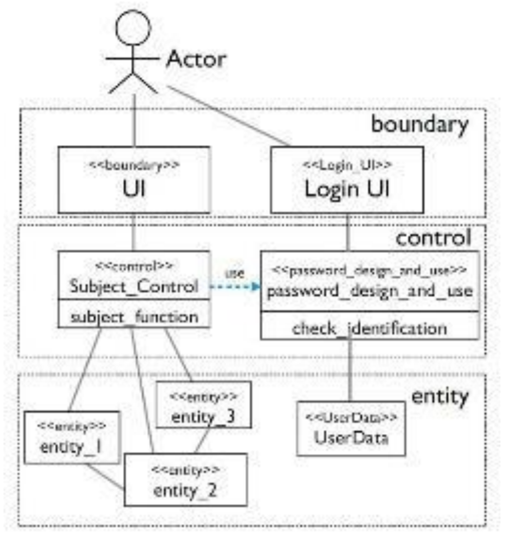
2. BACKGROUND AND PROBLEM

2.1. Security Design Patterns

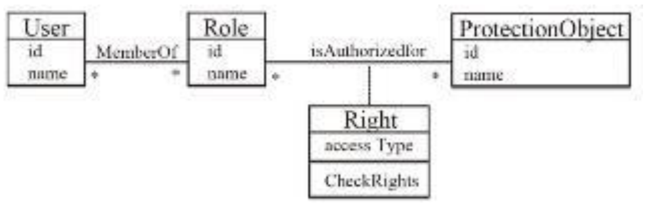
A security design pattern is a reusable documented security problem and corresponding solution that can be used to make decisions on the conceptual architecture and detailed design of a software system. In the design phase of software development, security functions should be designed to satisfy the security properties of the assets identified in the requirement phase. Security design patterns include "Name", "Context", "Problem", "Solution", "Structure", "Consequence", and "See Also". The "Structure" usually contains some design models described by the Unified Modeling Language (UML). UML is a widely accepted and standardized language to model software applications [4]. These models may contain constraints on the design in the form of OCL descriptions. OCL [5, 6] stands for Object Constraint Language, which is a semiformal language that is used to express constraints and other expressions in UML and other modeling languages. For example in [7], we successfully defined UML models with OCL descriptions for various security design patterns.

These patterns can be reused in multiple software systems. Figure 1 shows two examples of security design pattern structures. The Password Design and Use pattern [3, 8] describes the best security practice to design, create, manage, and use password components. In addition to configuring or managing passwords, engineers and administrators use password constraints to build or select password systems. Figure 2 shows the corresponding OCL description for Figure 1 specifying the security requirement that should be satisfied in the structure. The OCL description means that the ID and Password inputted from the login screen must agree with the ID and Password of the User Data in order for a user to be deemed as a regular user and granted access to an asset. If this condition is not met, the user is considered a non-regular user and denied access to an asset. Moreover, Figures 3 and 4 show the expected behavior in the form of a UML communication diagram.

The Role-based Access Control (RBAC) pattern [3, 8], which is a representative pattern for access control, describes how to assign precise access rights to roles in an environment where access to computing resources must be controlled to preserve confidentiality and the availability requirements.



(**a**)



(**b**)

Figure 1. (a) Structure of a security design pattern (Password Design and Use pattern) and (b) structure of a security design pattern (Role-based Access Control pattern)

```
context subject_controller
 inv check_id_and_pass:
  if self.password_design_and_use.User_Data->exists(p |
       p.id = self. password_design_and_use.Login_UI.id and
       p.pass = self. password_design_and_use.Login_UI.pass)
  then
     self.UI.actor.regular_user = true  and self.subject_function = true
  else
     self.UI.actor.regular_user = false  and self.subject_function = false
  endif
```

Figure 2. OCL description (Password Design and Use pattern)

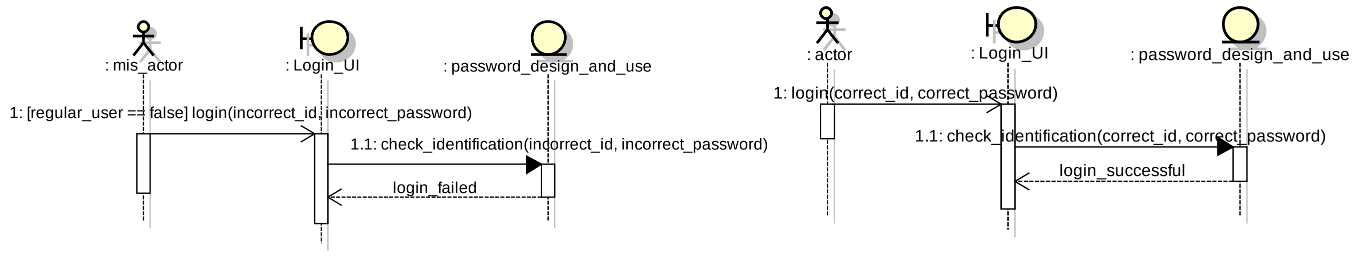

: mis_actor
: Login_UI
: password_design_and_use
1: [regular_user == false] login(incorrect_id, incorrect_password)
1.1: check_identification(incorrect_id, incorrect_password)
login_failed
: actor
: Login_UI
: password_design_and_use
1: login(correct_id, correct_password)
1.1: check_identification(correct_id, correct_password)
login_successful

Figure 3. Login behavior (Password Design and Use pattern)

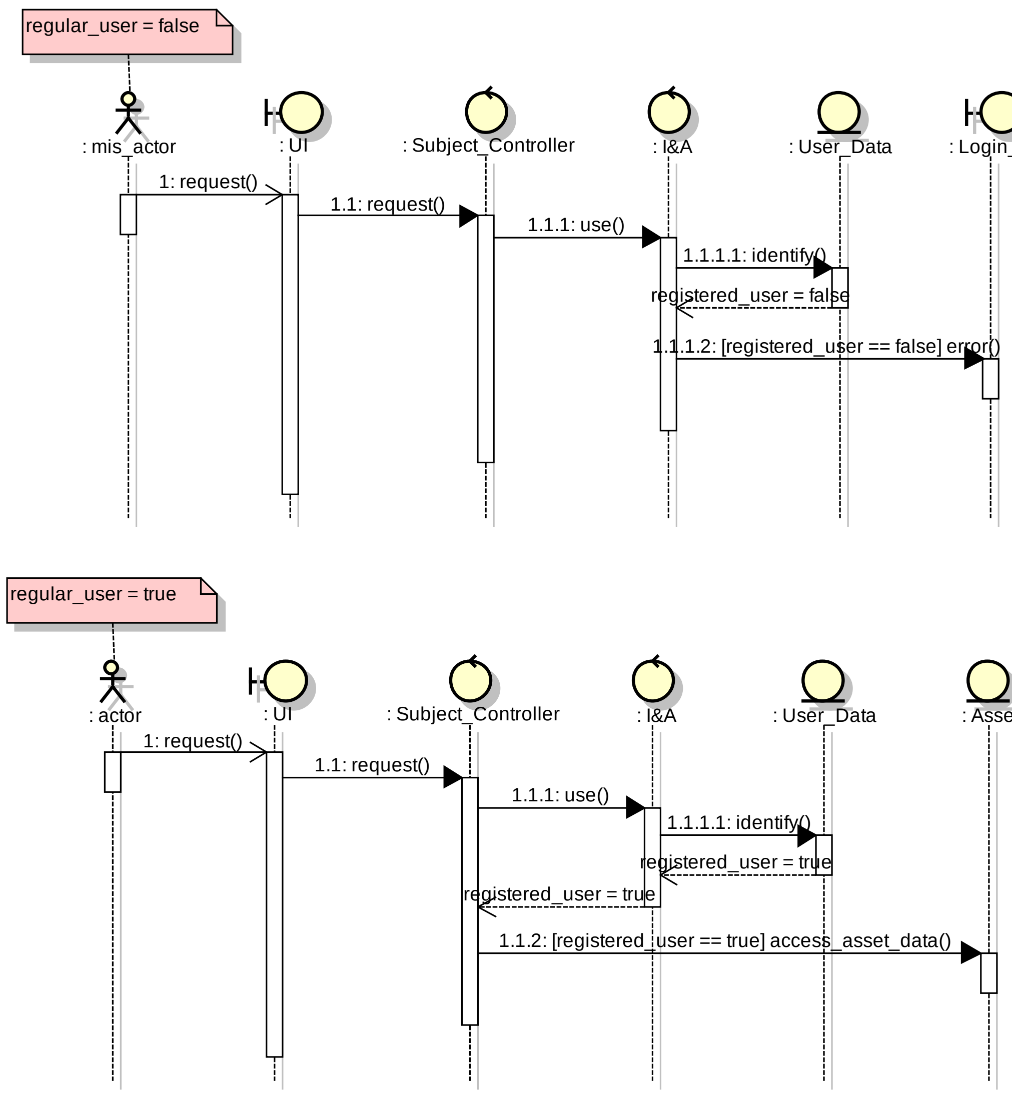


Figure 4. Behavior to access an asset (Password Design and Use pattern)

2.2. Motivating example of implementation problem

As an example of a pattern application, Figure 5 shows a portion ("login") of a UML class diagram.

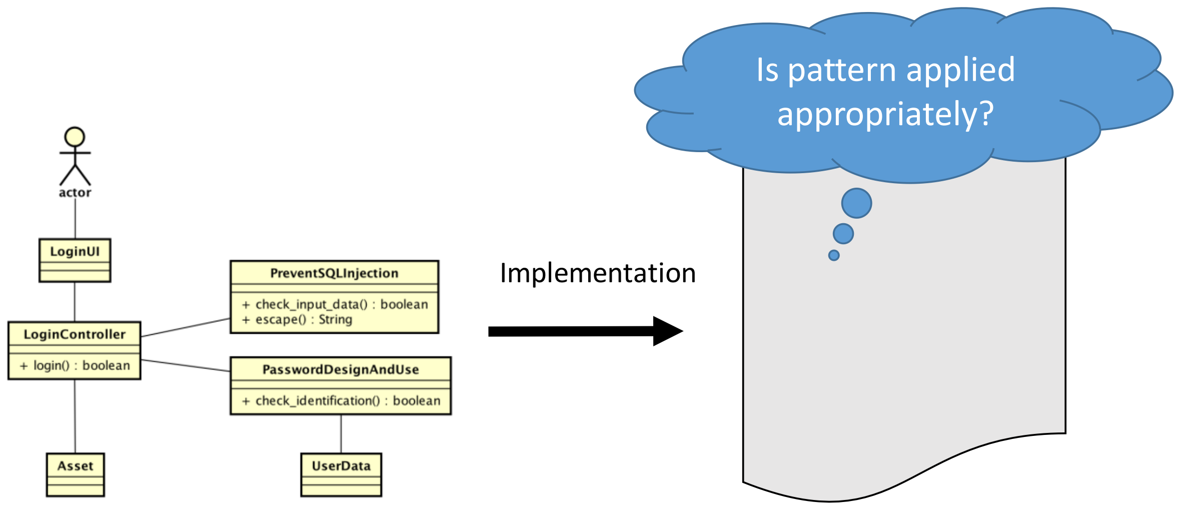
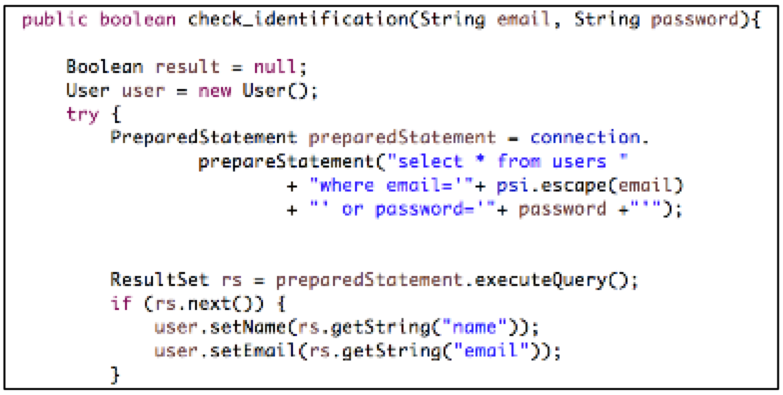


Figure 5. Implementation of the "login" portion of a class diagram for payment processing

Although the class diagram in Figure 5 appropriately applies the Password Design and Use pattern and the Prevent SQL Injection pattern [3], it is incomprehensible in the implementation phase. How to implement the selected pattern is unclear because the relation between the security design pattern and implementation is not defined. Figure 6 shows part of the implementation of the diagram in Figure 5 with defects, which are a miswritten SQL description and an omitted escape process. Consequently, the system may be vulnerable. Thus, the applied pattern must be verified via a test in the implementation phase.



```
public boolean check_identification(String email, String password){

    Boolean result = null;
    User user = new User();
    try {
        PreparedStatement preparedStatement = connection.
                prepareStatement("select * from users "
                        + "where email='"+ psi.escape(email)
                        + "' or password='"+ password +"'");


        ResultSet rs = preparedStatement.executeQuery();
        if (rs.next()) {
            user.setName(rs.getString("name"));
            user.setEmail(rs.getString("email"));
        }
```

Figure 6. Part of the implementation of the class diagram in Figure 5

A conventional test only detects vulnerabilities due to known coding bugs. It cannot determine if a security design pattern is appropriately applied, making validation extremely difficult in the implementation phase. Similar to our work, Munetoh et al. proposed a method to verify the completeness of the implemented security features [9], but their method is limited to access control for Ruby-on-Rails web applications.

3. RELATED WORKS

3.1. Security Patterns Veirification

Several articles focus on verifying security pattern applications. Abramov at al. have suggested using a stereotype for a database application to validate security patterns at the design level [10]. Although their method can validate an applied pattern structurally, it cannot confirm whether the pattern behavior in the implementation code resolves vulnerabilities to threats. In contrast, our method tests the pattern behavior at the implementation level.

Dongs at al. have proposed an approach to verify the compositions of security patterns using model checking [11]. They presented guidelines to specify the behavior of security patterns in the model specification language. They defined a synchronous message, asynchronous message, and alternative flows of a UML sequence diagram, which were transformed into CCS specifications. Although this approach formally defines the behavioral aspect of security patterns and provides transformation scripts to check the properties of the security patterns by model checking at the model level, it does not conduct an analysis at the implementation level. Therefore, even if the security patterns application can be verified at the model level, it does not guarantee that threats and vulnerabilities are resolved in the corresponding code. On the other hand, our method tests the pattern at the implementation level.

Similarly, Hamid et al. proposed a method to validate the application of patterns at the model level by UML and OCL [12]. However, it is not applicable to implementation level analysis.

Modeling techniques such as UMLsec [13] and SecureUML [14] have been proposed to address security concerns. UMLsec is defined in the form of a UML profile using standard UML extension mechanisms. Stereotypes with tagged values are used to formulate the security requirements, while constraints are used to verify whether the security requirements hold during a specific type of attack. SecureUML focuses on modeling access control policies and how these policies can be integrated into a model-driven software development process. It is based on an extended model of RBAC and uses RBAC as a meta-model to specify and enforce security. RBAC lacks support to express access control conditions that refer to the state of a system, such as the state of a protected resource. In addressing this limitation, SecureUML introduces the concept of authorization constraints. Authorization constraints are preconditions for granting access to an operation. Although these techniques can support modeling security-related concerns in UML models, they cannot be directly used to check the correctness of security pattern applications at the implementation level.

3.2. Model-based Security Testing

Model-based Testing (MBT) is technique to generate part or all of a test case from a model [15]. A model is an abstract representation expressing an operation, structure, behavior or any other concern to be realized by the system or related to the system. Especially in the context of security testing, the process of deriving tests tends to be unstructured, irreproducible, undocumented, lacking detailed rationales for the test design, and dependent on the ingenuity of single testers or hackers, motivating the use of explicit models [16]. Tretmans and Brinksma proposed an automated MBT tool [17], while Felderer et al. proposed a method of Model-based Security Testing (MBST) that relies on models to test whether a software system meets its security requirements [18]. Moreover, there are many other publications on MBST techniques since MBST is a relatively new research field with potential in industrial applications [19]. Felderer et al. proposed a taxonomy for MBST approaches and systematically classified 119 publications using MBST [16]. However, none of these publications focus on security design pattern applications.

We have developed a formal test template based on MBT. Our test template, which is used to create a security design pattern, is abstract and reusable. Thus, it can be applied to various systems. According to the taxonomy proposed by [16], our method can be classified as an approach adopting “Functionality of Security Mechanisms” as a “Model of system security”, “Threat Models” as a “Security model of the environment”, “Explicit Test Case Specifications” as “Explicit test selection criteria”, “Prototype” as the “Maturity of evaluated system”, both “Effectiveness Measures” and “Efficiency Measures” as “Evidence measures”, and “Executable” as the “Evidence level”.

3.3. Test-driven Development (TDD) for Security

Test-driven Development (TDD) is a software development technique that uses short development iterations based on prewritten test cases, which define desired improvements or functions. Here our testing process uses TDD, which requires development prior to writing the actual code [20]. A test case represents a requirement that the program must satisfy [21]. Although some approach implement security requirements by adopting TDD [9][22], they do not specifically handle security design patterns and consequently require developers to manually prepare test cases when applying and testing security design patterns. This is highly burdensome.

On the other hand, our method focuses on security design pattern applications. Our method uses Selenium [23, 24] and JUnit [25, 26], which are tools for testing applications. First a test is created where the security requirement of the corresponding pattern is satisfied. Then a test is quickly executed (first test) to detect vulnerabilities in the code. Afterwards the code is updated so that it passes the test. Finally the test is re-executed to confirm that the vulnerabilities are resolved.

3.4. Decision Table Testing

A decision table is a tool to capture certain kinds of system requirements and to document internal system design [27]. The general form of a decision table consists of conditions and actions; "conditions" represent various input conditions [27] and "actions" are the actions that should be taken depending on the combination of the input conditions.

Decision tables can serve as a guide to create test cases. Such a black-box test design technique based on decision tables is called "decision table testing". Decision table testing is commonly used to research various software targets, including as web applications [28] and web services [29] at various phases such as integration testing [30] and system testing [31].

Our method adopts decision tables to describe the specifications of target security design patterns. These tables are then used to derive the test templates and corresponding test cases.

3.5. Aspect-oriented Programming

Aspect-oriented Programming aims to improve the modularity of software by providing constructs to modularize the so-called crosscutting concerns, which are concerns where the code representation cannot be modularized using traditional software development mechanisms [32]. For example, in the pointcut-advice model of aspect-oriented programming, which is embodied in AspectJ [33], a crosscutting behavior is defined by pointcuts and advice. A pointcut is a predicate that matches program execution points, which are called join points, while advice is the action to be taken at a join point. An aspect is a module that encompasses a number of pointcuts and advice [34].

Montrieux et al. presented a tool that generates Java and AspectJ code from a UML model with a verified UMLsec property [35]. Although the tool can be utilized for testing at the implementation level, it handles only RBAC as the target security design pattern, enforcing the security property to be checked. In our method, developers can register and handle various security design patterns to be tested.

In our method, a test template, which is created via AspectJ, embodies the test, observes the internal processing, and supports vulnerability validation. Because the internal processing does not display the result on the screen such as escape processing in Figure 3, it is hard to validate that the processing is executed and the result is correct using browser-based testing.

4. IMPLEMENTATION SUPPORT OF SECURITY DESIGN PATTERNS

4.1. Overview

Our method verifies the applications of security design patterns and supports their implementation. Figure 7 outlines the process of our method. A test template is derived from a security design pattern by providing design information. Then the developer executes a test to validate the applied security design pattern in the implementation phase of software development. Figure 8 shows the entire picture of our method that clarifies necessary inputs and outputs by using our method's implementation tool, called TESEM (TEst driven SEcure Modeling tool). The developer can create a test case by inputting correct input data, incorrect input data, and the testing environment Selenium code into TESEM. Our tool allows a developer who is not a security specialist to create the test case.

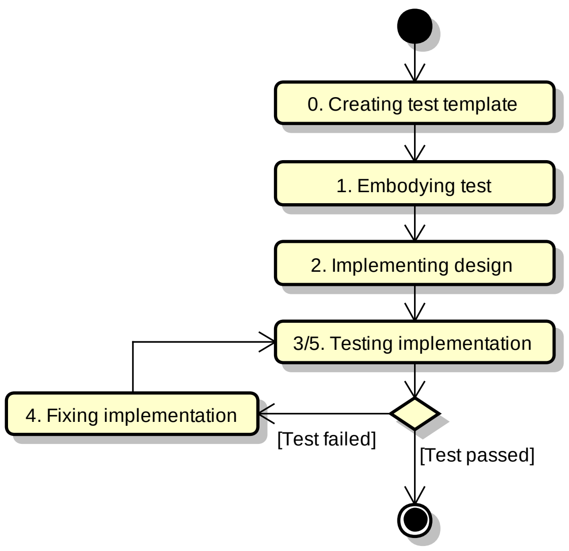


Figure 7. Process of our method

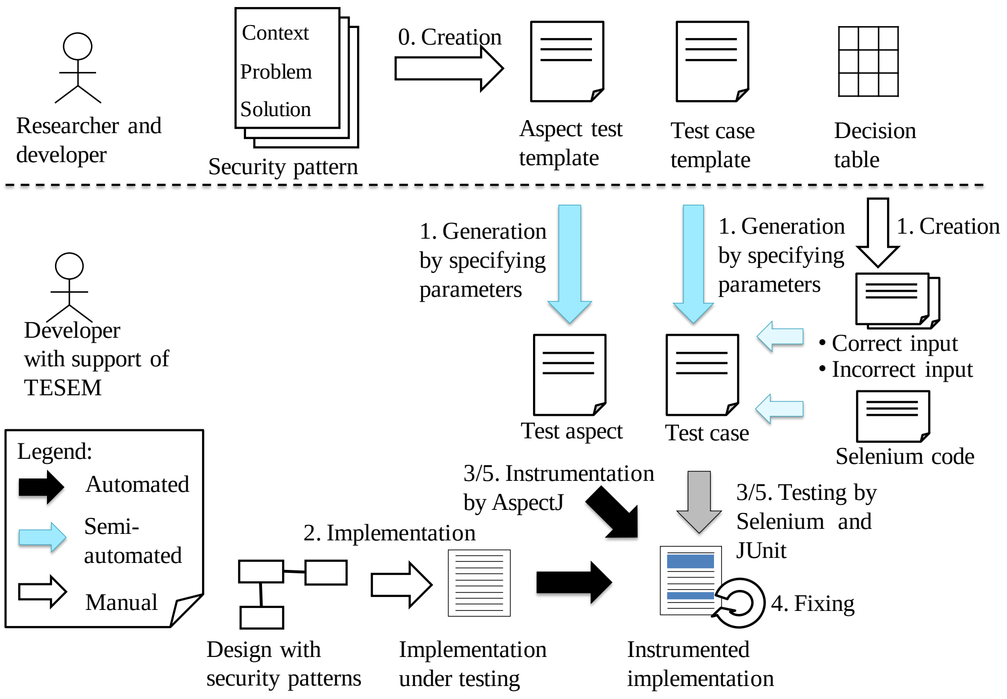


Figure 8. Entire picture of our method

Specifically, our method involves six steps.

Step 0: Create a test template.

A test template is prepared from a security design pattern. The test template consists of an "aspect test template" and a "test case template".

Step 1: Embody tests.

A test is embodied by the given design information in a test template.

Step 2: Implement a design.

Although the pattern application cannot be verified, the intended design for the security design pattern can be implemented.

Step 3: Test and validate the applied patterns.

Based on TDD, a test is quickly executed to validate the applied patterns in the implementation phase. During this step, concrete test aspects generated from aspect test templates are weaved into the implementation under testing for the instrumentation purpose.

Step 4: Fix.

The implementation is fixed based on the defects found in Step 3.

Step 5: Re-test and re-validate applied patterns.

The fixed implementation is re-tested to re-validate the applied patterns. If the test is passed successfully, then the patterns are appropriately applied in the implementation phase. If the test is failed, Step 4 is repeated until the re-test passes.

4.2. Test Template and Test Case

The test template consists of a test case template and an aspect test template. The former generates concrete test cases by specifying a few parameters. The latter generates concrete aspects for instrumentation to observe the internal processing of the code under testing. Testing security design pattern applications often require such internal data access.

Using our previous method [7], we prepared OCL descriptions for each existing security design pattern as test oracles. Then we prepared decision tables and the corresponding test template according to the OCL descriptions.

In our method, templates are prepared and registered to our tool manually for each security design pattern. After the registration, concrete test cases are generated semi-automatically as the parameters must be inputted. The generated test cases check the implementation under testing whether its outputs or internal states meet the expected results stated in the test cases automatically.

The flow to (1) realize a test template and (2) generate test cases is shown below.

(1) For each security design pattern to be tested, we (and the developers) prepare and register a pair of a test case template and an aspect test template using the three steps below. Once registered, the templates can be reused for further development projects.
   a) Developers create a decision table from the OCL description.
   b) Developers create an aspect test template for an internal processing observation from the decision table and the pattern structure.
   c) Developers create a test case template from the decision table and the pattern behavior.

(2) By using TESEM, developers can easily generate concrete test cases as follows:
   a) Developers bind the elements constituting the selected security design pattern and class/interface names in the given design model by specifying a class/interface name for each pattern element as a parameter.
   b) Developers input the Selenium code corresponding to the target security design pattern to be tested, such as a code behaving as a login function to test the login.
   c) Developers specify concrete correct values for the parameters in the test cases, such as concrete values of ID and password for testing login. Developers also specify concrete incorrect values.
   d) Finally, TESEM generates test cases together with concrete test aspects using the parameters and

the information specified in the above steps.

4.3. Tool Implementation and Example

In this section, we explain our method's tool implementation as well as the test template using the Password Design and Use pattern as a concrete example. We implemented our method into TESEM [7, 36-39], which is a modeling and testing tool originally dedicated only to testing design models. Under the environment of JUnit together with Selenium, the generated test cases check whether the implementation's outputs and internal states meet the expected results described in the test cases.

Figures 1(a), 2–4 depict the Password Design and Use pattern. Table I shows a decision table following the OCL description.

Table I. Decision Table (Password Design and Use pattern)

		1	2	3	4
Conditions	Inputted ID agrees with User Data.	Yes	Yes	No	No
	Inputted Password agree with User Data.	Yes	No	Yes	No
Actions	Considered a regular user	×			
	Can access an asset.	×			
	Considered a non-regular user		×	×	×
	Cannot access an asset.		×	×	×

Next, an aspect test template in AspectJ is created from the decision table and structure of the pattern to observe the internal processing. The test should verify the user status (regular or non-regular user) and whether to allow access. In the structure diagram shown in Figure 1(a), part of "considered regular user or non- regular user" is the check_identification method of password_design_and_use class. To observe the internal processing of this point, its pointcut and advice are defined. By observing the internal processing, it is possible to find a critical defect, which cannot be found using browser-based testing.

Figure 9(a) shows a pointcut executing the consideration of a regular or non-regular user. Figure 9(b) shows the advice to observe the pointcut result. Similarly, a pointcut and advice are defined for whether to allow or deny access. In the structure diagram shown in Figure 1(a), whether to grant access is the subject_function method of the Subject_Control class. Figure 10(a) shows the pointcut to allow or deny access to an asset. Figure 10(b) shows the advice to determine the pointcut result.

```
pointcut LoginCheck() :
  call(* *.password_design_and_use.check_identification(..));
```

(**a**)

```
after() returning(Boolean right)  :
     EscapeCheck() {  setTemporary("LoginCheck",right); }
```

(**b**)

Figure 9. (a) Pointcut considering a regular or non-regular user and (b) advice to differentiate between the two types of users

```
pointcut AssetAccess () :
  call(* *.Subject_Controller.subject_function(..));
```

(**a**)

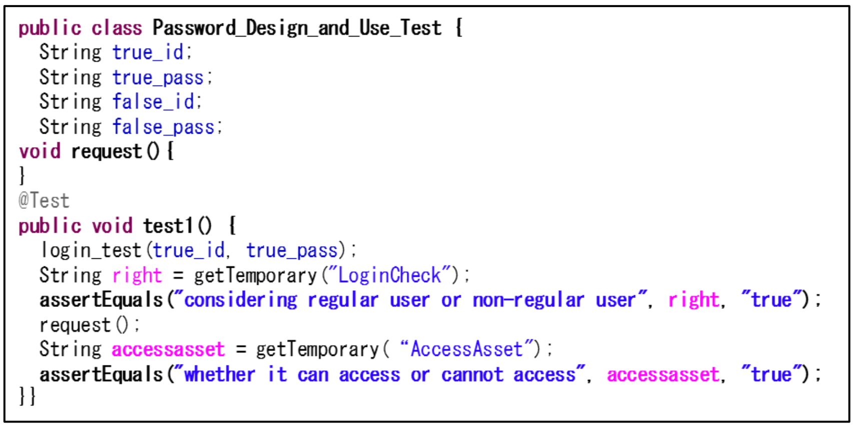
```
after() returning(Boolean right)  :
    AssetAccess () {  setTemporary("AssetAccess",right); }
```

(**b**)

Figure 10. (a) Pointcut judging access to an asset and (b) advice to allow access to an asset

Finally, a test case template is created from the decision table and pattern behavior. Figure 4 shows the behavior of a login to which the Boolean value of a regular_user is returned when an actor inputs an ID and password into a Login_UI as well as the behavior by which an actor sends a request to the subject_controller. A test case template performs these behaviors and confirms whether the internal process is correctly performed as actions of the decision table.

Figure 11 shows part of the test case template used to create a test and define “true_id”, “true_pass”, “false_id”, “false_pass”, and “request”. Because these templates do not contain a specific value, the defined terms must be embodied in order to use them as a test.



```
public class Password_Design_and_Use_Test {
  String true_id;
  String true_pass;
  String false_id;
  String false_pass;
void request() {
}
@Test
public void test1() {
  login_test(true_id, true_pass);
  String right = getTemporary("LoginCheck");
  assertEquals("considering regular user or non-regular user", right, "true");
  request();
  String accessasset = getTemporary( “AccessAsset”);
  assertEquals("whether it can access or cannot access", accessasset, "true");
}}
```

Figure 11. Part of the test case template

4.4. Limitations

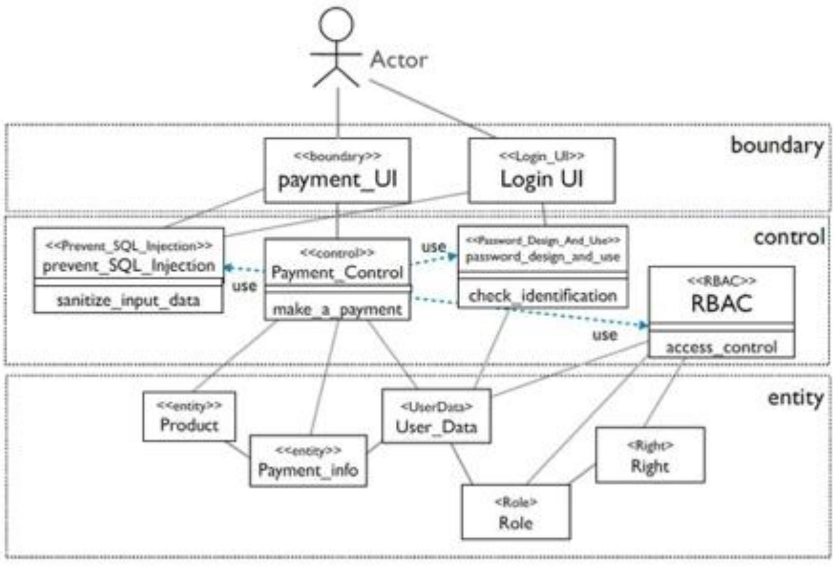
Some expertise about security and OCL may be required to prepare the decision tables and test templates. In relation to that, there is a possibility that developers prepare incorrect test templates against the original expectation of the corresponding security design pattern by misunderstanding the pattern. In the future, we plan to develop a technique and its implementing tool to support transformations from security design patterns to formal decision tables and templates.

There are two possible situations of false positives: (1) developers apply the pattern to the target implementation in a slightly different manner from the case assumed in the test templates and test cases, and (2) developers misunderstand the patterns applied in the corresponding design. Regarding (1), developers should prepare the corresponding test templates of such variant of the pattern application. Regarding (2), since TESEM covers both the design phase and the implementation one; it is easy to avoid such misunderstandings by using our tool beginning in the design phase.

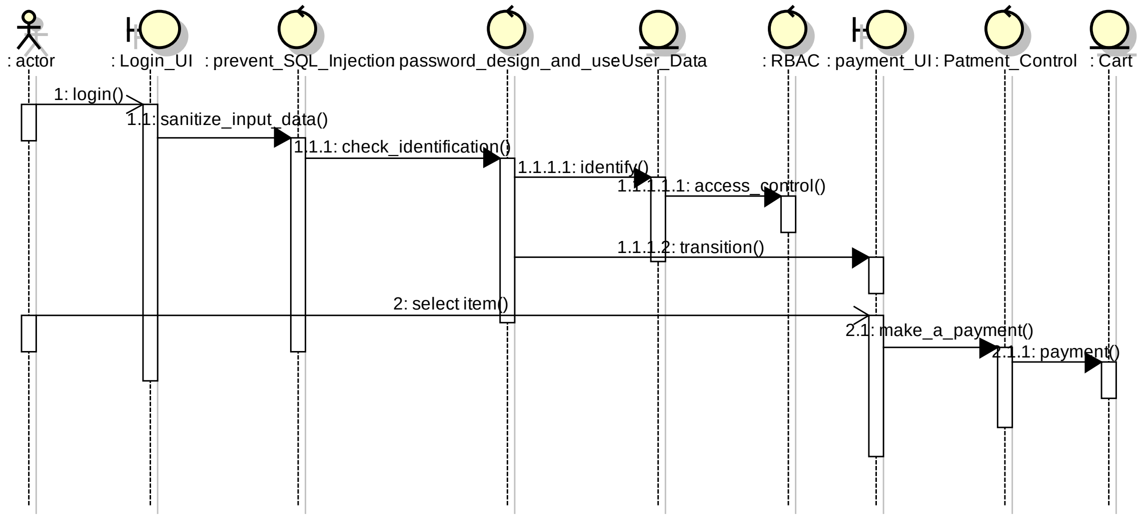
In our method, only security design patterns that are described in general pattern templates with added OCL descriptions are test targets. As mentioned earlier, we successfully defined UML models with OCL descriptions for various security design patterns [7]. In the future, we have a plan to prepare a guideline of describing OCL descriptions for security patterns based on our experience.

5. Case Study

As a validation, we applied our method to an e-commerce system. Figures 12(a) and (b) show the structure and behavior of the e-commerce system, respectively. “Password Design and Use pattern”, “Prevent SQL Injection pattern”, and “Role-based access control pattern” are used as the premises.



(**a**)



(b)

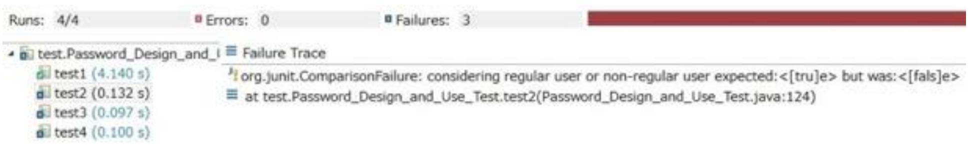
Figure 12. Structure and behavior of a purchasing system

We embody a test case using previously defined design information (e.g., the make_a_payment method of Payment_Control class due to apply “Password Design and Use pattern” in Figure 12(a)) (Step 1). Additionally, “select item and push a button” is used to access to an asset in Figure 12(b). Figure 13 shows part of the created test case. In Figure 13, the “make_a_payment_test” method is embodied in the “request” method in Figure 11 because the “request” method in the test case template corresponds to the “make_a_payment_test” method in the system. Stereotypes, such as <<control>> and <<Login_UI>>, attached to the class diagram show these corresponding relations.

```
public class Password_Design_and_Use_Test {

String true_id="111";
String true_pass="111";
String false_id= "222";
String false_pass= "222";

void login_test(String id,String pass){

driver.get("http://localhost:8080/Purchasing/Login_UI");
WebElement user_id = driver.findElement(By.name("user_id"));
WebElement user_pass = driver.findElement(By.name("user_pass"));
user_id.sendKeys(id);
user_pass.sendKeys(pass);
user_id.submit();
}

void make_a_payment_test(){

driver.get("http://localhost:8080/Purchasing/Payment_UI");
driver.findElement(By.id( "item")).click();
driver.findElement(By.id("purchse")).submit();

}}
```

Figure 13. Part of the created test case

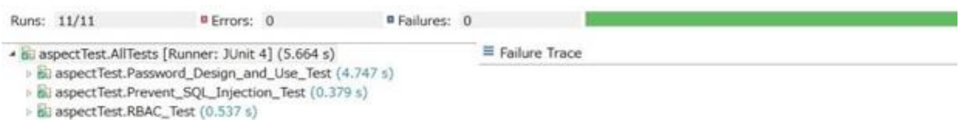
After implementing the design (Step 2), a test is executed to validate the application of the patterns in the implementation phase (Step 3). Figure 14 shows the test result for the "Password Design and Use pattern".



```
Runs: 4/4          Errors: 0          Failures: 3

test.Password_Design_and_l   Failure Trace
  test1 (4.140 s)            org.junit.ComparisonFailure: considering regular user or non-regular user expected:<[tru]e> but was:<[fals]e>
  test2 (0.132 s)            at test.Password_Design_and_Use_Test.test2(Password_Design_and_Use_Test.java:124)
  test3 (0.097 s)
  test4 (0.100 s)
```

Figure 14. Test results (Password Design and Use pattern)

Next, the implementation is fixed (Step 4). The error message in Figure 14 indicates that "whether to allow or deny access" cannot be determined. Finally, the implementation is retested and the applied patterns are re-validated (Step 5). Figure 15, which shows the re-test results, confirms that "Password Design and Use pattern", "Prevent SQL Injection pattern", and "Role-based Access Control pattern" are applied appropriately.



```
Runs: 11/11          Errors: 0          Failures: 0

aspectTest.AllTests [Runner: JUnit 4] (5.664 s)        Failure Trace
  aspectTest.Password_Design_and_Use_Test (4.747 s)
  aspectTest.Prevent_SQL_Injection_Test (0.379 s)
  aspectTest.RBAC_Test (0.537 s)
```

Figure 15. Re-test results

6. EVALUATION

6.1. Experimental Overview

To evaluate the Research Questions, we conducted experiments involving four students in information sciences. We estimated the defect detection and correct implementation of security design patterns by our technique and an existing method for a login system of an e-commerce system and EMSsec [40]. EMSsec is a student information management system, which was developed as a measure to address a common problem regarding security and privacy. We injected three defects similar to the ones shown in Figure 3 into these systems.

As the baseline for the comparison, the existing method is one where only general testing environments, including JUnit and Selenium, are provided. Additionally, there is no further support such as test case generation. Thus, the existing method is completely manual.

Students were asked to find and correct the three defects. Our experiment involved the following steps:

Step 1: Complete a questionnaire before the experiment.

Step 2: Listen to an explanation of our method, JUnit, and Selenium.

Step 3: Detect and correct the defects by the existing (our) method.

Step 4: Detect and correct the defects by our (existing) method.

Step 5: Complete a questionnaire after the experiment.

Step 1: Questionnaire before the experiment

To measure the students' ability, they completed a questionnaire prior to participating in the experiment. The questionnaire consisted of the following:

- What is your skill level regarding developing a program?
- What is your knowledge level regarding security?
- What is your knowledge level regarding software testing?
- What is your knowledge level regarding web systems development?

Students selected from the following answers: "very high", "high", "not high", and "low".

Step 2: Listen to an explanation of our method, JUnit, and Selenium.

We explained how to use our method, JUnit, and Selenium.

Steps 3, 4: Detect and correct the defects by the existing method and our method.

Students detected and corrected defects using our method and the existing method. The students were randomly divided into two groups. The first group of students 1 and 2 used the existing method followed by our method, while the second group of students 3 and 4 used our method first.

Step 5: Complete the after-experiment questionnaire.

Upon completing the experiment, students completed a questionnaire about their experience. The questionnaire contained the following three items:

- Which method do you prefer to find defects in the software?
- Which method do you feel is more efficient at finding defects?
- Which method do you prefer to modify defects in the software?

6.2. Experimental Results

The table below shows the result of the before and after questionnaire (Step 1, 5).

Table II. Results of the Before and After Questionnaire (Step 1, 5) (VH: very high, H: high, NH: not high, L: low, O: our method, RO: relatively our method, RE: relatively existing method, E: existing method)

Students	What is your skill level regarding developing a program?	What is your knowledge level regarding security?	What is your knowledge level regarding software testing?	What is your skill level regarding web systems development?	Which method do you prefer when finding defects in software?	Which method do you feel more efficiently finds defects in software?	Which method do you prefer when modifying defects in software?
1	H	NH	H	NH	O	O	O
2	NH	L	NH	NH	O	O	O
3	NH	NH	NH	L	O	O	RO
4	NH	H	NH	NH	O	O	RO

Figures 16 and 17 show scatter plots of the results for each student. Using our method, the detection time decreased for all students. In addition, Student 2 detected more defects, whereas Students 1, 3, and 4 detected the same number of defects using both methods. As for the correction time, the second method (i.e. our method for students 1 and 2, and the existing method for students 3 and 4) required less time regardless of which method was used.

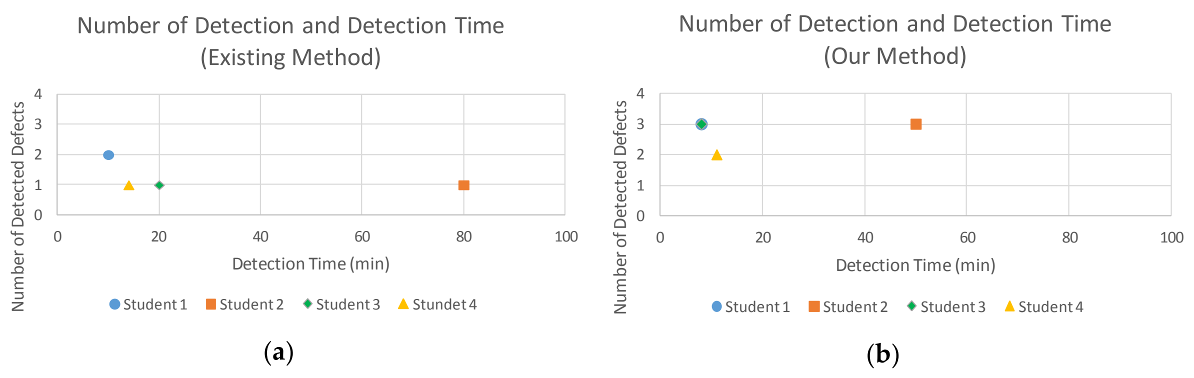


Figure 16. Scatter plots of (a) the existing method and (b) our method for the number detected defects and detection time

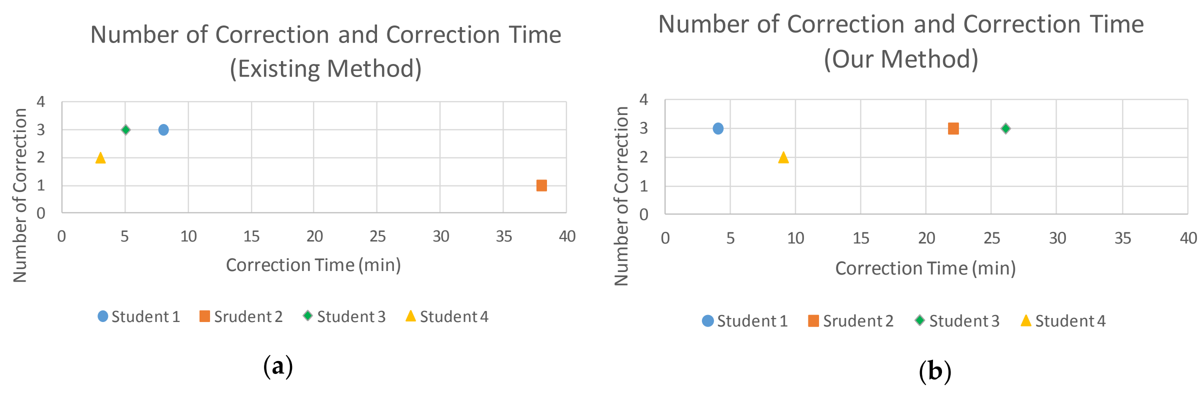


Figure 17. Scatter plots of (a) the existing method and (b) our method for the number of corrected defects and correction time

6.3. Discussion

Based on the results, we answer our RQs.

RQ 1: Is our method more efficient at finding defects when implementing a security design pattern than an existing method?

Regardless of whether our method was implemented before or after the existing method, our method required less time as shown in Figure 16. Thus, we conclude that our method can find implementation defects more efficiently than the existing method. By focusing on a particular security design pattern implementation, our method allows test templates to be reused and test cases to be semi-automatically generated; it contributes to such time reduction.

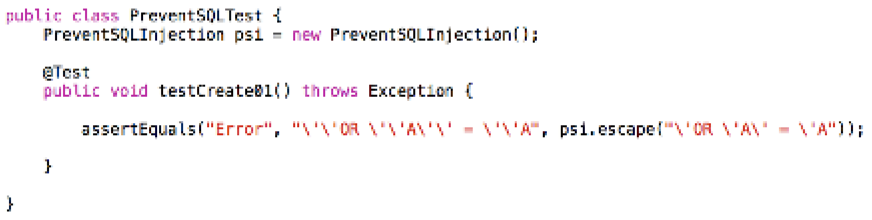
RQ 2: Does our method create a more effective test to find implementation defects in a security design pattern than an existing method?

All students successfully detected more defects using our method than the existing method. Moreover, students 1, 2, and 3 found all three defects with our method, confirming that our method can effectively test implementation defects.

In our method, test cases are semi-automatically generated from test templates by only inputting the necessary information for binding class/implementation names in the given design and elements in the patterns together with correct/incorrect input data and Selenium code. We believe that this feature somewhat mitigates the likelihood of incorrect test cases. Nevertheless, as we mentioned in the limitations section, there is a possibility that developers prepare incorrect test templates against the original expectation of the corresponding security design pattern by misunderstanding the pattern.

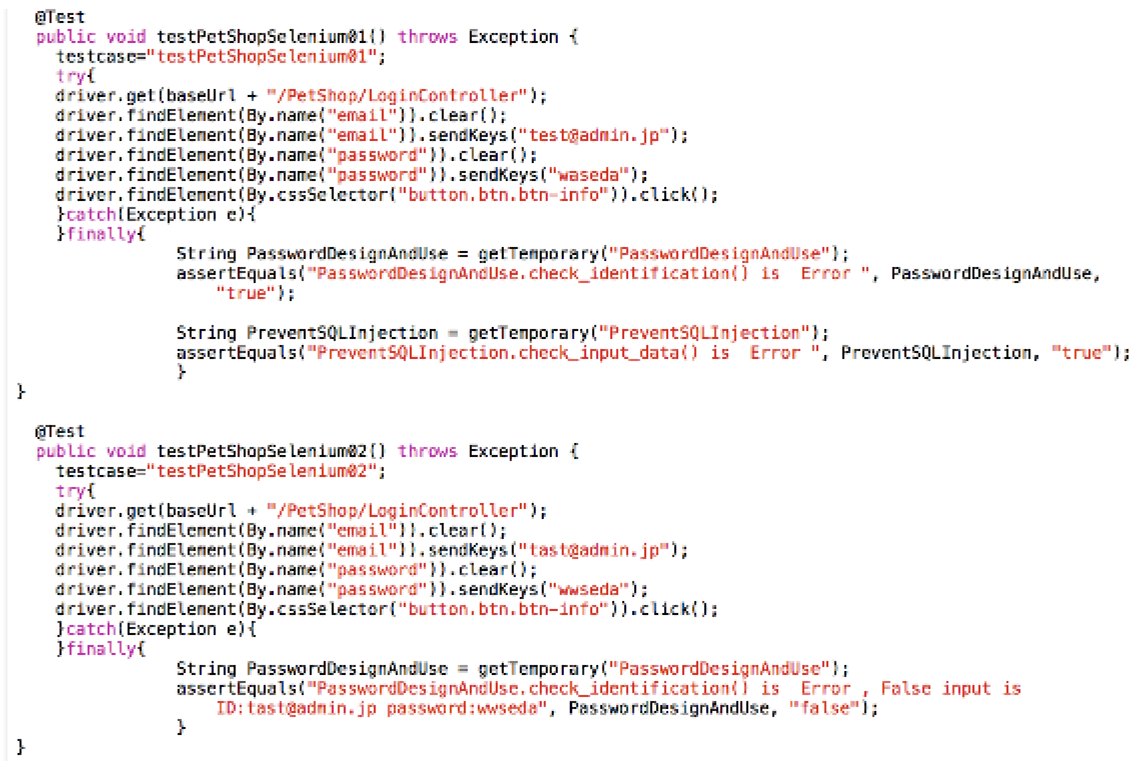
RQ3: Compared to an existing method, does our method more effectively create a test to find implementation defects in a security design pattern and allow for a regression test?

Figures 18 and 19 show the test case developed by Student 1. Figure 18 is the test case, which was rewritten for every test. Because the test case could not be reused, whether the test results are correct cannot be confirmed. In contrast, the test case shown in Figure 19 can be reused, allowing the accuracy of the result to be confirmed. Students 2 and 4 created test cases similar to Figure 18 (Figure 19) when using the existing (our) method. Using our method, all students were able to run a regression test. From these results, we conclude that our method can develop a regression test.



```
public class PreventSQLTest {
    PreventSQLInjection psi = new PreventSQLInjection();

    @Test
    public void testCreate01() throws Exception {

        assertEquals("Error", "\'\'OR \'\'A\'\' = \'\'A", psi.escape("\'OR \'A\' = \'A"));

    }

}
```

Figure 18. Part of the test case for Student 1 using the existing method



```
  @Test
  public void testPetShopSelenium01() throws Exception {
    testcase="testPetShopSelenium01";
    try{
    driver.get(baseUrl + "/PetShop/LoginController");
    driver.findElement(By.name("email")).clear();
    driver.findElement(By.name("email")).sendKeys("test@admin.jp");
    driver.findElement(By.name("password")).clear();
    driver.findElement(By.name("password")).sendKeys("waseda");
    driver.findElement(By.cssSelector("button.btn.btn-info")).click();
    }catch(Exception e){
    }finally{
                String PasswordDesignAndUse = getTemporary("PasswordDesignAndUse");
                assertEquals("PasswordDesignAndUse.check_identification() is  Error ", PasswordDesignAndUse,
                    "true");

                String PreventSQLInjection = getTemporary("PreventSQLInjection");
                assertEquals("PreventSQLInjection.check_input_data() is  Error ", PreventSQLInjection, "true");
                }
}

  @Test
  public void testPetShopSelenium02() throws Exception {
    testcase="testPetShopSelenium02";
    try{
    driver.get(baseUrl + "/PetShop/LoginController");
    driver.findElement(By.name("email")).clear();
    driver.findElement(By.name("email")).sendKeys("tast@admin.jp");
    driver.findElement(By.name("password")).clear();
    driver.findElement(By.name("password")).sendKeys("wwseda");
    driver.findElement(By.cssSelector("button.btn.btn-info")).click();
    }catch(Exception e){
    }finally{
                String PasswordDesignAndUse = getTemporary("PasswordDesignAndUse");
                assertEquals("PasswordDesignAndUse.check_identification() is  Error , False input is
                    ID:tast@admin.jp password:wwseda", PasswordDesignAndUse, "false");
                }
}
```

Figure 19. Part of the test case for Student 1 using our method

RQ 4: Is our method more effective at correcting implementation defects in a security design pattern compared to an existing method?

Although students 1, 3 and 4 found the same number of corrections using both methods, student 2 was only able to modify one defect using the existing method, but modified all three defects using our method. These findings demonstrate that our method can help correct implementation defects of security design patterns for users with low skills about web systems development compared to the existing method. However, our method did not shorten the correction time because it is assumed that more time is necessary to infer the correction contents from the test case generated by a tool automatically.

We believe that the semi-automated detection of implementation defects supports further corrections since testers or developers can easily confirm whether or not their implementation under testing is truly correct against security design patterns. However, as mentioned previously, our method does not directly support defect correction.

6.4. Threats to Validity

In the experiment, we did not verify whether our method is applicable to any type of system. Additionally, the numbers of security design patterns and developers were insufficient. Hence, it is possible that our method is not applicable to all security patterns. Although we used representative patterns and a typical model for software development to demonstrate the usefulness of our method, we need to examine more general patterns and employ large-scale examples.

7. CONCLUSION AND FUTURE WORK

Because software developers are not necessarily security experts, security design patterns may be inappropriately applied. Additionally, even if security design patterns are properly applied in the design phase of software development, threats and vulnerabilities may not be mitigated in the implementation phase. Hence, we propose a support method for security design patterns in the implementation phase of software development.

Our method offers two significant contributions. First, it can create an effective test to find the implementation defects of a security design pattern irrespective of the user's background. Second, compared to the existing method, our method can help correct defects for users with a low web development ability.

There are several limitations in our method. Some expertise about security and OCL may be required to prepare the decision tables and test templates. Moreover, there is a possibility that developers prepare incorrect test templates against the original expectation of the corresponding security design pattern by misunderstanding the pattern. To mitigate the limitation, we plan to develop a technique and its implementing tool to support transformations from security design patterns to formal decision tables and templates.

Moreover, as our future work, to evaluate the scope of our method, we intend to perform an experiment using a system with more complex defects.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Numbers 25330091, 15H02686, 16H02804 and IISF SSR Forum 2015.

References

1. N. Yoshioka, H. Washizaki and K. Maruyama, "A Survey on Security Patterns" Progress in Informatics, No.5, pp. 35-47, 2008.
2. PT. Devanbu and S. Stubblebine, "Software Engineering for Security: a Roadmap" The Conference on The Future of Software Engineering, pp. 227-239, 2000.
3. M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann and P. Sommerlad, "Security Patterns", Wikey, 2006.
4. Object Management Group. Unified Modeling Language (UML) Version 2.5, 2015, Available on line: http://www.omg.org/spec/UML/
5. Object Management Group. OCL 2.4, 2014
6. J. Warmer and A. Kleppe, "The Object Constraint Language: Precise Modeling with UML," Addison-Wesley, 1998
7. T. Kobashi, M. Yoshizawa, H. Washizaki, Y. Fukazawa, N. Yoshioka, H. Kaiya and T. Okubo, "TESEM: A Tool for Verifying Security Design Pattern Applications by Model Testing," Proceedings of the 8th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2015), Tool Track, pp. 13 – 17, April 2015, Graz, Austria
8. Dr. E. Fernandez-Buglioni, "Security Patterns in Practice: Designing Secure Architectures Using Software Patterns," John Wiley & Sons, 2013.
9. S. Munetoh and N. Yoshioka, "Model-Assisted Access Control Implementation for Code-centric Ruby-on-Rails Web Application Development", The International Conference on Availability, Reliability and Security, 2013, pp. 350 – 359, 1999.
10. J. Abramov, P. Shoval and A. Sturm, "Validating and Implementing Security Patterns for Database Applications," 3rd International Workshop on Software Patterns and Quality (SPAQu), 2009.
11. J. Dong, T. Peng and Y. Zhao, "Automated verification of security pattern compositions," Information and Software Technology, Vol.52, No.3, pp. 274–295, 2010.
12. B. Hamid, C. Percebois and D. Gouteux, "Methodology for Integration of Patterns with Validation Purpose," European Conference on Pattern Language of Programs (EuroPLoP), pp. 1-14, 2012.
13. J. Jürjens, "Secure Systems Development with UML," Springer, 2005.
14. T. Lodderstedt, D.A. Basin and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security," 5th International Conference on The Unified Modeling Language (UML), pp. 426-441, 2002.
15. S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz, "Model-based Testing in practice", The International Conference on Software Engineering, pp. 285-294.

16. M. Felderer, P. Zech, R. Breu, M. Büchler and A. Pretschner, "Model-based security testing: a taxonomy and systematic classification," Software Testing, Verification and Reliability, Vol.26, No.2, pp. 119-148, 2016.
17. J. Tretmans and E. Brinksma, "TorX: Automated Model-Based Testing", The Conference on Model-Driven Software Engineering, pp. 11-12, 2003.
18. M. Felderer, B. Agreiter, R. Breu and A. Armenteros, "Security Testing by Telling TestStories", The conference on Modellierung, pp. 24-26, 2010.
19. I. Schieferdecker, J. Grossmann and M. Schneider, "Model-based security testing," 7th Workshop on Model Based Testing, pp. 1-12, 2012.
20. H. Kim, B. Choi, and S. Yoon, "Performance testing based on test-driven development for mobile applications", The International Conference on Ubiquitous Information Management and Communication, 2009, pp. 612-617.
21. S. Fraser, D. Astels, K. Beck, B. Boehm, J. McGregor, J. Newkirk, and C. Poole, "Discipline and practices of TDD: (test driven development) ", The Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 268-270, 2003.
22. Sonia and A. Singhal, "Development of Agile Security Framework Using a Hybrid Technique for Requirements Elicitation," International Conference on Advances in Computing, Communication and Control (ICAC), pp. 178-188, 2011.
23. Selenium Available online: http://docs.Seleniumhq.org/
24. A. Holmes and M. Kellogg, "Automating functional tests using Selenium", In: Agile Conference, p. 6, 2006
25. JUnit Available online: http://junit.org
26. P. Tahchiev, F. Leme, V. Massol and G. Gregory, "JUnit in action", Manning Publications Co., 2010.
27. L. Copeland, "A Practitioner's Guide to Software Test Design, " Artech House, 2004.
28. G.A.Di Lucca, A.R. Fasolino, F. Faralli, and U. de Carlini, "Testing Web Applications," 18th International Conference on Software Maintenance (ICSM), pp. 310-319, 2002.
29. S. Noikajana and T. Suwannasart, "Web Service Test Case Generation Based on Decision Table," 8th International Conference on Quality Software (QSIC), pp. 321-326, 2008.
30. J. Hartmann, C. Imoberdorf and M. Meisinger, "UML-Based Integration Testing," ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 60-70, 2000.
31. L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," 4th International Conference on The Unified Modeling Language. Modeling Languages, Concepts, and Tools (UML), LNCS Vol. 2185, pp. 194-208, 2001.
32. S. Endrikat and S. Hanenberg, "Is Aspect-Oriented Programming a Rewarding Investment into Future Code Changes? A Socio-technical Study on Development and Maintenance Time", The International Conference on Program Comprehension, pp. 51 - 60, 2011.
33. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, "An overview of AspectJ", The Conference on Object-Oriented Programming, pp. 327-354, 2001.
34. É. Tanter, "Execution levels for aspect-oriented programming", The 9th International Conference on Aspect-Oriented Software Development, pp. 37-48, 2010.
35. L. Montrieux, J. Jürjens, C.B. Haley, Y. Yu, P.Y. Schobbens and H. Toussaint, "Tool support for code generation from a UMLsec property," 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 357-358, 2010.
36. TESEM Available online: http://patterns.fuka.info.waseda.ac.jp
37. T. Kobashi, N. Yoshioka, T. Okubo, H. Kaiya, H. Washizaki, Y. Fukazawa, "Validating Security Design Pattern Applications Using Model Testing," Proceedings of 8th International Conference on Availability, Reliability and Security (ARES2013), pp.62-71, 2013.
38. M. Yoshizawa, T. Kobashi, H. Washizaki, Y. Fukazawa, T. Okubo, H. Kaiya and N. Yoshioka, "Verification of Implementing Security Design Patterns Using a Test Template," 9th International Conference on Availability, Reliability and Security (ARES2014), pp. 178-183, 2014.
39. T. Kobashi, N. Yoshioka, H. Kaiya, H. Washizaki, T. Okubo, Y. Fukazawa, "Validating Security Design Pattern Applications by Testing Design Models," International Journal of Secure Software Engineering, Vol. 5, Issue 4, pp.1-30, 2014.
40. EMSsec Available online: http://lab.iisec.ac.jp/~okubo_lab/Members/okubo/wiki/index.php?EMSSec