Verification of Implementing Security Design Patterns Using a Test Template

Abstract—**Although security patterns contain security expert knowledge to support software developers, these patterns may be inappropriately applied because most developers are not security specialists, leading to threats and vulnerabilities. Here we propose a validation method for security design patterns in the implementation phase of software development. Our method creates a test template from a security design pattern, which consists of the "aspect test template" to observe the internal processing and the "test case template". Providing design information creates a test from the test template. Because a test template is recyclable, it can create easily a test, which can validate the security design patterns. As a case study, we applied our method to a web system. The result shows that our method can test repetition in the early stage of implementation, verify pattern applications, and assess whether vulnerabilities are resolved.**

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

I. INTRODUCTION

Security issues have become critical due to the increasing number of business services on open networks and distributed platforms [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 number and the fact that not all software engineers are security specialists.

Patterns are reusable packages that incorporate expert knowledge. Specifically, patterns represent a frequently recurring structure, behavior, activity, process, or "thing" during the software development process. Many security design patterns have been proposed. For example, reference [3] includes 25 design-level security patterns.

Currently, security design patterns are abstract descriptions, making them difficult to implement. Additionally, it is hard to validate the security design patterns in the implementation phase because an adequate test case is required. Hence, a security pattern can be inappropriately applied, leading to serious vulnerability issues.

We propose a method to validate security design patterns using a test template in the implementation phase. Our method creates a test template from the security design pattern. The test template 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 stage of implementation and refactor the code. This test can be executed repeatedly and can validate the applied security design patterns in the implementation phase.

We address the following research questions.

RQ1 Can a test be created from the test template?

RQ2 Can the created test case validate whether the security design pattern is appropriately applied in the implementation phase of software development?

Our contributions are as follows:

- A reusable test template created from the security design patterns.
- Embodiment of the test template by providing design information.
- Validation of the security design patterns in the implementation phase.
- Ease of testing with the test template.

The remainder of this paper is organized as follows. First, we provide background and problems with security patterns in Section II. In Section III, we describe our proposed method. We then discuss the evaluation results of our method in Section IV. In Section V, we describe potential weaknesses of our method. Finally, we provide a conclusion and future works in Section VI.

II. BACKGROUND

A. *Security Design Patterns*

Security design patterns are an existing technique to make decisions on the conceptual architecture and detailed design of system. In the design phase of software development, security functions should be designed to satisfy the security properties of assets identified in the requirement phase. Security design patterns include "Name", "Context", "Problem", "Solution", "Structure", "Consequence", "See Also", and "OCL Description". OCL stands for Object Constraint Language, which is a semiformal language that can be used to express constraints and other expressions in UML and other modeling languages. Patterns can be reused in multiple systems.

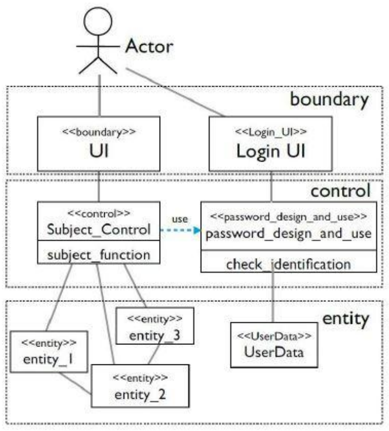


Figure 1. Structure of a security pattern (Password Design and Use pattern)

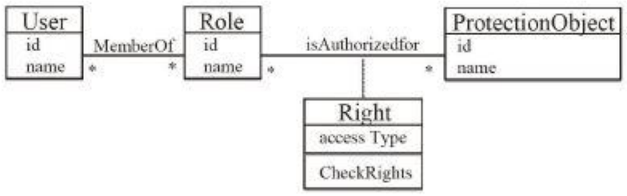


Figure 2. Structure of a security pattern (Role-based Access Control pattern)

Figures 1 and 2 show examples of the security pattern structure. The Password Design and Use pattern 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. The Role-based Access Control (RBAC) pattern, 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.

B. *Motivating example*

As an example of a pattern application, Figure 3 shows a portion ("make a payment") of a UML class diagram, which is implemented to realize a payment process on the Web.

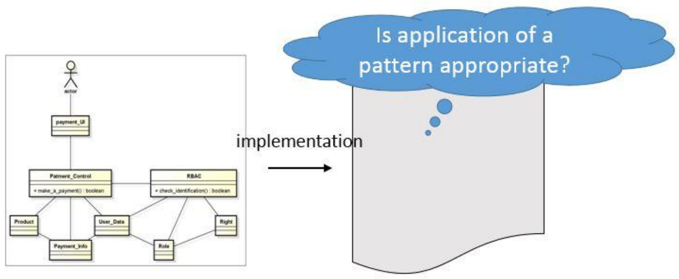

Is application of a pattern appropriate?
implementation
actor
payment_UI
Patment_Control
+ make_a_payment() : boolean
RBAC
+ check_identification() : boolean
Product
User_Data
Right
Payment_Info
Role

Figure 3. Implementation of the "make a payment" portion of a class diagram for payment processing

Although the class diagram in Fig. 3 appropriately applies the security design pattern, it is incomprehensible in the implementation phase. It is unclear how the selected pattern should be implemented because the relation between a security design pattern and implementation is not defined. Consequently, the system may be vulnerable, and the applied pattern must be verified via a test.

However, a conventional test only detects vulnerabilities due to known coding bugs; it cannot determine if a security design pattern is appropriately applied, which is difficult to validate in the implementation phase. Similar to our work, reference [4] has proposed a method to verify the completeness of implemented security features, but this. method is limited to access control for Ruby-on-Rails web application development.

C. *Model-based Testing*

Model-based Testing (MBT) is technique to generate part or all of a test case from a model [5]. A model is an abstract thing expressing an operation that should realize the system. Testing is complicated and expensive. MBT can alleviate these issues. Reference [6] proposes an automated MBT tool, while reference [7] proposes a method of security MBT although a security pattern is not used.

We have created a formal test template based on MBT. To create a security design pattern, the test template is abstract and reusable. Therefore, the test template is applicable to various systems.

D. *Test-driven Development (TDD)*

Test-driven Development (TDD) is a software development technique that uses short development iterations based on prewritten test cases, which define the desired improvements or new functions. Here our testing process uses TDD, which requires development prior to writing the actual code [7]. A test case represents a requirement that the program must satisfy [8].

Our method uses Selenium [9], which is a tool for testing web applications. The first step is to create a test in which a requirement is satisfied. The next step is to quickly execute a test (test first) to detect vulnerabilities in the code. Then the code is updated so that it passes the test. Finally, the test is re-executed to confirm that the vulnerabilities are resolved.

E. *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 [10]. In the pointcut-advice model of aspect-oriented programming, which is embodied in AspectJ [11] for example, a crosscutting behavior is defined by pointcuts and advices. A pointcut is a predicate that matches program execution points, called join points, while advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and advices [12].

In our method, a test template is created via AspectJ. A test, which embodies the test template, observes the internal processing and supports vulnerability validation.

III. OUR VALIDATION METHOD

A. *Overview*

Figure 4 outlines the process of our method. A test template, which is derived from security design pattern, is prepared by providing design information to create a test. Then a developer can execute a test to validate the applied security design pattern in the implementation phase of software development.

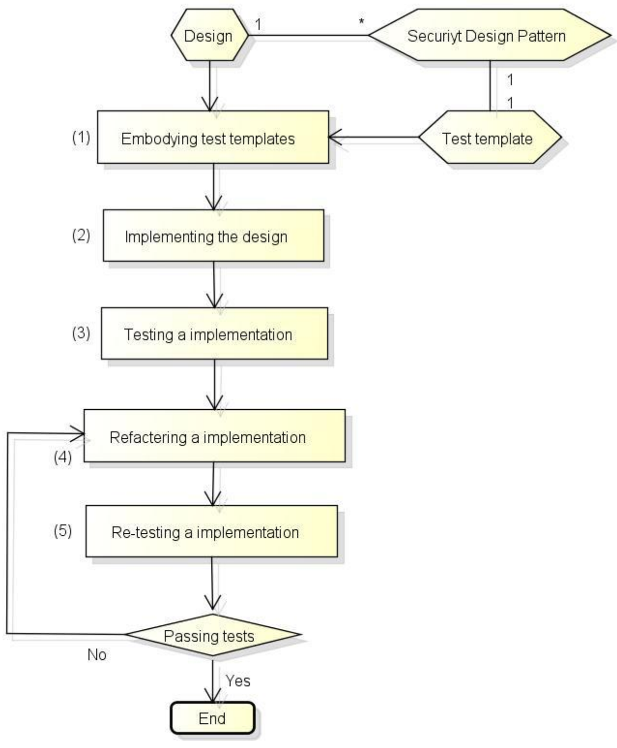


Figure 4. Process of our method

Specifically, our method involves five steps.

Step0. Create a test template

A test template is created from a security design pattern during a previous step. The test template consists of an “aspect test template” and “test case template”.

Step1. Embody test templates

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

Step2. Implement a design

The design for which the security design pattern was used is implemented, but whether the patterns are applied cannot be verified in this step.

Step3. Test and validate the applied patterns

Based on TDD, a test is quickly executed to validate the applied patterns in the implementation phase.

Step4. Refactor

The implementation is refactored based on the errors found in step3.

Step5. Re-test and re-validate applied patterns

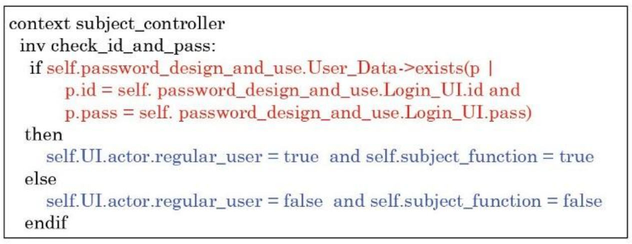
The refactored implementation is re-tested to re-validate the applied patterns. If the test is true, the patterns are successfully applied in the implementation phase. Otherwise step4 is repeated until the re-test is passed.

B. *Test Template*

In this section, we explain the test template using a concrete example of “Password Design and Use Pattern”. The flow to realize a test template is shown below.

1. A decision table is created from the OCL Description.
2. An aspect test template for an internal processing observation is created from the decision table and the pattern structure.
3. A test case template is produced from the decision table and behavior of a pattern.

The test template consists of this aspect test template and this test case.



```
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 5. OCL Description (Password Design and Use pattern)

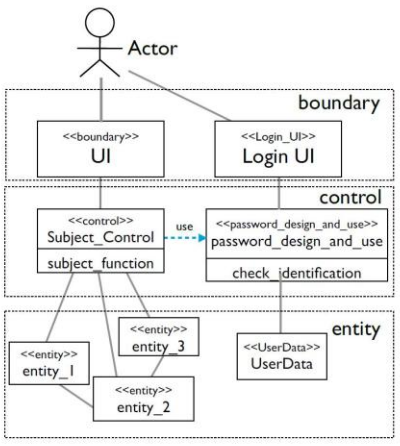


Figure 6. Structure (Password Design and Use pattern)

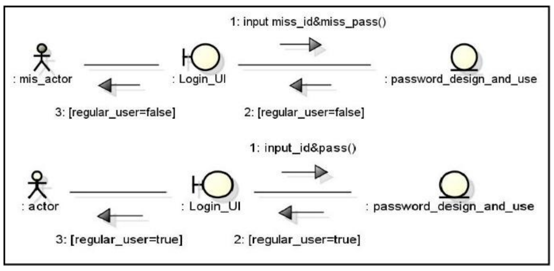


Figure 7. Login behavior (Password Design and Use pattern)

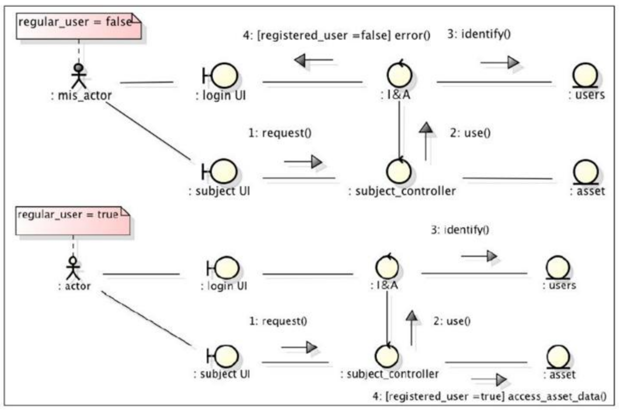


Figure 8. Behavior to access an asset (Password Design and Use pattern)

Figures 5–8 depict the Password Design and Use pattern. The OCL Description (Fig. 5) means that if the ID and Password inputted from the login screen agree with the ID and Password of User Data, then the user is deemed a regular user and allowed access to an asset. Otherwise, the user is considered a non-regular user and denied access to an asset. From this OCL Description, a decision table is created (Table I).

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 to observe the internal processing is created from the decision table and structure of the pattern. The objects to be verified in the test from the decision table are whether to be considered a regular or non-regular user and whether to allow access.

In the structure diagram shown in Fig. 6, part of "considered regular user or non- regular user" is the check_identification method of password_design_and_use class. In order to observe the internal processing of this point, a pointcut and advice are defined.

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

Figure 9. Pointcut considering a regular or non-regular user

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

Figure 10. Advice to observe the consideration of whether a regular or non-regular user

Figure 9 shows a pointcut executing the consideration of a regular or non-regular user. Figure 10 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 Fig. 6, whether to allow or deny access is the subject_function method of the Subject_Contorol class.

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

Figure 11. Pointcut judging access to an asset

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

Figure 12. Advice observing access to an asset

Figure 11 shows the pointcut executing the consideration to allow or deny access to an asset. Figure 12 shows the advice observing the pointcut result.

Finally, a test case template is created from the decision table and pattern behavior. Figure 7 shows the behavior of a login to which a Boolean value of a regular_user is returned when an actor inputs an ID and password into a Login_UI, and Fig. 8 shows 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 processing observed by the aspect is correctly performed as actions of the decision table.

Figure 13 shows part of test case template, which is used to create a test, and defines "true_id", "true_pass", "false_id", "false_pass", and "request". These are templates in which the value is not contained. Therefore, 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 13. Part of a test case template

IV. EVALUATION

To answer the two research questions, we conduced case studies to evaluate our method.

RQ1 Can a test be created from the test template?

RQ2 Can the created test case validate whether the security design pattern is appropriately applied in the implementation phase of software development?

A. *Case Studies*

We applied our method to a purchasing system on the Web in reference [13] as an example validation process. Figures 14 and 15 show the structure and behavior to which our method is applied, respectively. "Password Design and Use pattern", "Prevent SQL Injection pattern", and "Role-based access control pattern" are used as a premise.

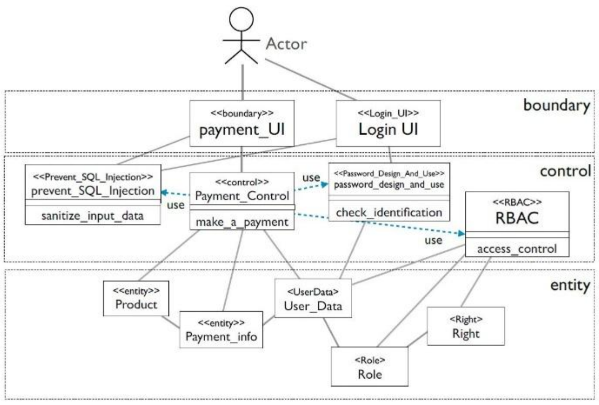
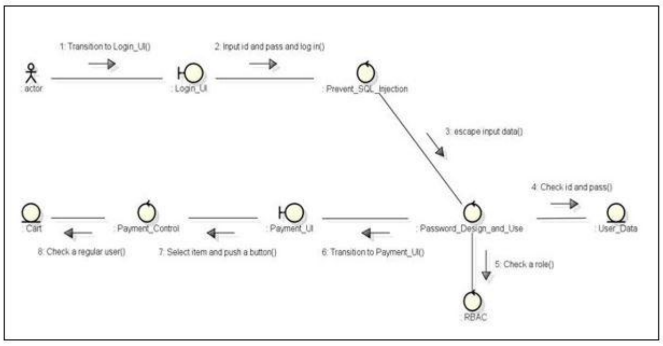


Figure 14. Structure applied to our method



1: Transition to Login_UI()
2: Input id and pass and log in()
: actor
: Login_UI
: Prevent_SQL_Injection
3: escape input data()
4: Check id and pass()
: Cart
: Payment_Control
: Payment_UI
: Password_Design_and_Use
: User_Data
8: Check a regular user()
7: Select item and push a button()
6: Transition to Payment_UI()
5: Check a role()
: RBAC

Figure 15. Behavior applied to our method

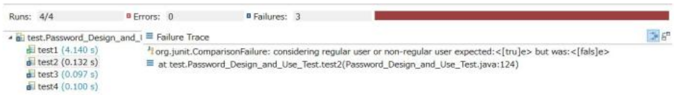
We embody a test case using the design information that is previously defined (e.g., the make_a_payment method of Payment_Control class due to apply "Password Design and Use pattern" in Fig. 14). Additionally, "select item and push a button" is used to access to an asset in Fig. 15. These are used to create a test template, part of which is shown in Fig. 16.

```
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 16. Part of the created test

In Fig. 16, the "make_a_payment_test" method is embodied in the "request" method in Fig. 13 because the "request" method in test 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 correspondence relations.

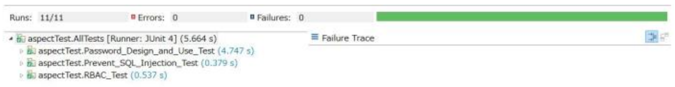
After the design is implemented, a test is executed to validate the application of patterns in the implementation phase. Figure 17 shows the test result for the "Password Design and Use pattern".



```
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 17. Result of a test ("Password Design and Use pattern")

Next, the implementation is refactored. The error message in Fig. 17 indicates that "whether allow or deny access" is impossible. Finally, we re-test the implementation and re-validate the applied patterns. Figure 18, 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 18. Re-test results

B. *Research Questions*

Our case studies deal with four patterns. A test is created from the test template by providing design information. Thus, the proposed method answers RQ1.

Then we validated whether implementation after testing satisfies the Security Design Pattern by repeated testing based on TDD. Consequently, the test validated the existence of vulnerabilities identified in the early implementation phase. Additionally, we confirm qualitatively that the test is quickly created. Thus, the proposed method answers RQ2.

C. *Limitations*

Our method has a few limitations. Because the test is created using design information, we postulate that the security design pattern is appropriately applied in the design. Additionally, verification of vulnerability that is not considered by the design may be out of range.

V. THREATS TO VALIDITY

A. *Threats to internal validity*

Although our test template may eliminate human dependency, the effectiveness of the template should be confirmed when employed by a developer unfamiliar with our method.

B. *Threats to external validity*

We used representative patterns and a typical model for software development. However, we did not verify whether our method is applicable to all types of patterns and models. In the future, we intend to confirm that our method is applicable to more patterns and more general examples.

VI. CONCLUSIONS AND FUTURE WORK

Because a software developer is not necessarily a security expert, patterns may be inappropriately applied. Additionally, even if patterns are properly applied in the design phase of software development, threats and vulnerabilities may not be mitigated or resolved in the implementation phase. Hence, we propose a validation method for security design patterns using a test template in the implementation phase.

This method offers two significant contributions. First, a reusable test template created from a security design pattern is defined, easily creating and executing a test in the early implementation phase. Second, the security design pattern is validated in the implementation phase. Although the test is manually created from a test template, in the future we plan to automatically transform a test template into a test.

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] 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.

[5] 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.

[6] J.Tretmans, E.Brinksma, "TorX: Automated Model-Based Testing", The Conference on Model-Driven Software Engineering, pp. 11-12, 2003.

[7] M.Felderer, B.Agreiter, R.Breu and A.Armenteros, "Security Testing by Telling TestStories", The conference on Modellierung, pp. 24-26, 2010.

[8] Heejin Kim, Byoungju Choi, and Seokjin Yoon, "Performance testing based on test-driven development for mobile applications", The International Conference on Ubiquitous Information Management and Communication, 2009, pp. 612-617.

[9] Steven Fraser, Dave Astels, Kent Beck, Barry Boehm, John McGregor, James Newkirk, and Charlie Poole,"Discipline and practices of TDD: (test driven development) ", The Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 268-270, 2003.

[10] Selenium http://docs.seleniumhq.org/

[11] Endrikat.S, Hanenberg.S, "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.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten,Jeffrey Palm, and William Griswold, "An overview of AspectJ", The Conference on Object-Oriented Programming, pp. 327-354, 2001.

[13] Éric Tanter, "Execution levels for aspect-oriented programming", The 9th International Conference on Aspect-Oriented Software Development, pp. 37-48, 2010.

[14] T.Kobashi, N.Yoshioka, T.Okubo, H.Washizaka and Y.Fukazawa, "Validating Security Design Pattern Applications Using Model Testing", The International Conference on Availability, Reliability and Security, pp. 62-71, 2013.