Validating Security Design Pattern Applications Using Model Testing

Abstract

Software developers are not necessarily security experts, confirming potential threats and vulnerabilities at an early stage of the development process (e.g., in the requirement- and design-phase) is insufficient. Additionally, even if designed software considers security at an early stage, whether the software really satisfies the security requirements must be confirmed. To realize secure design, we propose an application to validate security patterns using model testing. Our method provides extended security patterns, which include requirement- and design-level patterns as well as a new model testing process using these patterns. After a developer specifies threats and vulnerabilities in the target system during an early stage of development, our method can validate whether the security patterns are properly applied and assess if these vulnerabilities are resolved.

Keywords-component; Security Patterns; Model Testing; Test- Driven Development; UML;

I. INTRODUCTION

Due to the increased number of business services on open networks and distributed platforms, security has become a critical issue. Software must be supported with security measures (Maruyama, Washizaki, & Yoshioka, 2008), which are addressed in every phase of software development from requirements engineering to design, implementation, testing, and deployment. However, threats and vulnerabilities within a system cannot be sufficiently identified during the early development stage. Due to the vast number of security concerns and the fact that not all software engineers are security specialists, creating software with adequate security measures is extremely difficult.

Patterns, which are reusable packages that incorporate expert knowledge, represent frequently recurring structures, behaviors, activities, processes, or “things” during the software

development process. Many security patterns have been proposed to resolve security issues (Bschmann, Fernandez-Buglioni, Schumacher, Sommerlad, & Hybertson, 2002). For example, Bschmann et al (2006) propose 25 design-level security patterns. By referring to these patterns, developer can realize software with high security level efficiently.

Although UML-based models are widely used for design, especially for model-driven software development, whether the patterns are applied correctly is often not verified (Maruyama, Washizaki, & Yoshioka, 2008). Therefore, it is possible to apply a security pattern inappropriately. Additionally, properly applying a security pattern does not guarantee that threats and vulnerabilities will be resolved. These issues may result in security damage. Thus, we propose an application to validate security patterns using model testing. Our method confirms that security patterns are properly applied and assesses whether vulnerabilities are resolved. Our research aims to answer the following two Research Questions (RQs):

- RQ1: Can our method validate an appropriate application of the security design pattern in a design model?
- RQ2: Can our method validate the presence of vulnerabilities identified at the requirement stage before and after applying patterns?

Herein we answer these two research questions. Because the Security Pattern alone does not provide systematic guidelines with respect to applications, we formally extended existing security patterns. Then we proposed a new testing process to validate applied patterns and a tool to support model testing. Our method provides three major contributions:

- New extended security patterns using Object Constraint Language (OCL) expressions, which include requirement- and design-level patterns;

- A new model-testing process based on Test-Driven Development (TDD) to validate appropriate pattern applications and the existence of vulnerabilities using these extended patterns;
- A tool to support pattern application by creating a script to execute model testing automatically.

This paper is organized as follows. Section II describes the background and problems with security software development. Section III describes related works. Section IV details our new method, which integrates the security patterns. Section V applies our pattern to a case study. Section VI describes the threats to validity of our method. Finally, Section VII summarizes this paper.

II. BACKGROUND AND PROBLEMS

In this section, we overview common existing techniques for secure design.

A. Security Requirement Patterns (SRPs)

The security requirement pattern (SRP) is an existing technique to identify assets, threats, and countermeasures (Kaiya, Okubo & Yoshioka, 2012). A security pattern is reusable as a security package and includes security knowledge, allowing software developers to design secure systems like a security expert. Various types of security patterns exist. For example, SRPs are used at the requirement level, while security design patterns (SDPs), which are described in Section C, are applied at the design stage level.

The “Structure” of a SRP uses the Misuse case with the Assets and Security Goal (MASG) model (Okubo, Taguch & Yoshioka, 2009), which is an extension of the misuse case (Andreas & Sindre, 2000) that provides the structure of assets, threats, and countermeasures at

the requirement level. The MASG model can model attackers, attacks, and countermeasures as well as normal users and their requirements. In addition to the elements of the misuse case diagrams, the MASG model consists of the following:

- Data assets: Assets to be protected;
- Use case assets: Functions related to assets;
- Security goals: Reasons to protect assets.

Identifying assets improves threat recognition, while identifying security goals determines what security measures are important in the target system. The MASG model also contains a security requirement analysis process. First, the assets of the system are identified, and the security goals are defined. Next, threats that may violate the goals are defined, and security countermeasures against these threats are determined (Kaiya, Okubo & Yoshioka ,2012). Finally, security countermeasures that satisfy the security goals are confirmed.

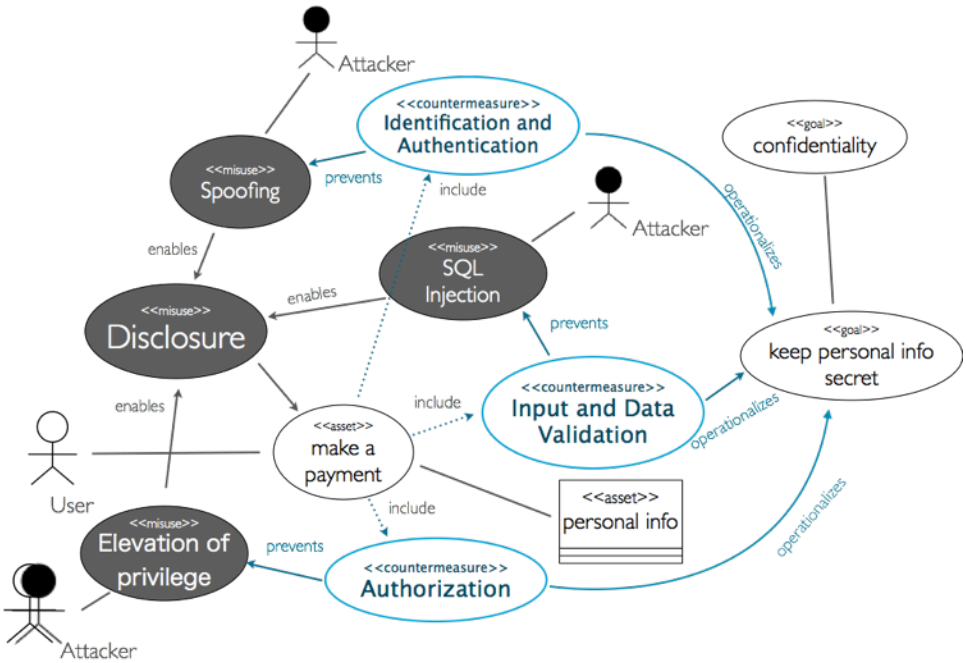

Figure 1. Sample MASG model for a shopping website

Figure 1 shows a typical example of the MASG model: a partially modeled shopping website. The function “make a payment” has several assets, which could be threatened. In the model, "Disclosure" is a threat for "make a payment", while "personal information" is an asset. "Spoofing", "Elevation of privilege", and “SQL Injection” enable Disclosure. In addition, each countermeasure, such as “I&A (Identification and Authentication)”, “Authorization”, or “Input and Data Validation”, are effective to mitigate threats. Although the MASG model helps comprehensively detect security issues at the requirement level, it does not indicate whether the identified threats actually exist in the software system.

B. Security Design Patterns (SDPs)

SDPs are an established technique to satisfy security specifications. A SDP includes “Name”, “Context”, “Problem”, “Solution”, “Structure”, “Dynamics”,“Consequences”, and “See Also”. The pattern descriptions can be reused in multiple systems. As examples of SDPs, Bschmann et al (2006) propose 25 design-level security patterns.

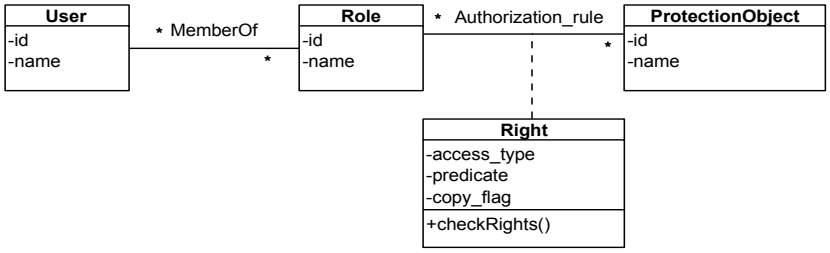

Figure 2. Structure of RBAC

Figure 2 shows the structure of Role Based Access Control (RBAC) as an example of a SDP. The 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.

C. Motivating example

As an example of an applied pattern, Fig. 3 shows a portion ("make a payment") of a UML class diagram that realizes a payment process on the Web.

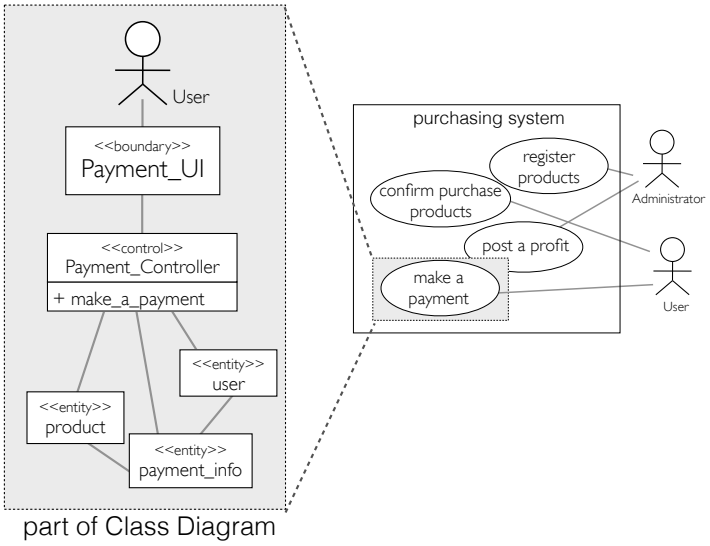

Figure 3. "Make a payment" portion of a class diagram for payment processing

A SDP alone cannot support the development lifecycle because it lacks systematic guidelines with respect to applications in the entire lifecycle (Dong, Peng & Zhao, 2008). Consequently, formally describing what rules must be verified is difficult (Abramov, Shoval & Sturm, 2009). In addition, most SDPs do not specifically mention systematic guidelines until the relations with Security Requirements are defined. Under the present conditions, even if a developer intends to apply a SDP such as RBAC (Fig. 2) to the structul model (Fig. 3), a developer may inappropriately apply a security measure to an identified threat. Additionally, the appropriateness of the applied pattern to the model and the pattern's ability to resolve vulnerabilities are often inadequately verified. Consequently, these situations may lead to inappropriately applying patterns and unresolved vulnerabilities.

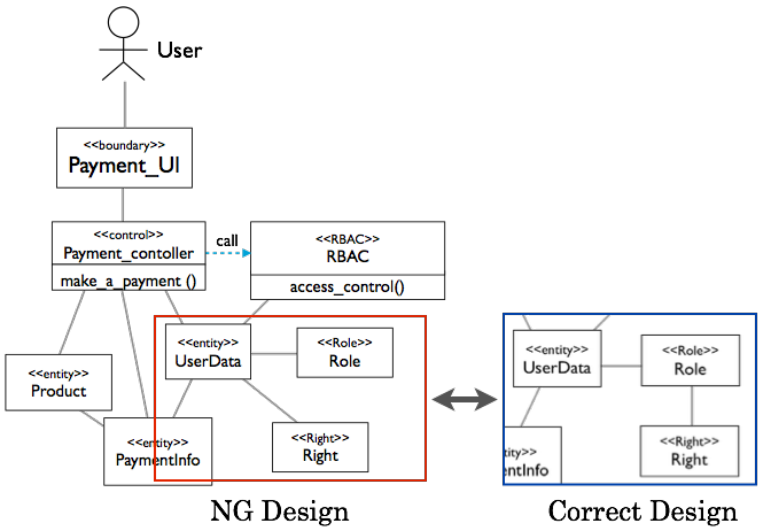

Figure 4. Example of an inappropriate pattern application

Figures 4 shows an example of an inappropriate pattern application where the RBAC is applied to the model shown in Fig. 3. Due to the lack of systematic guidelines with respect to the application and a method to validate patterns, it is possible that a developer may apply patterns inappropriately (e.g., like NG design in Fig. 4). The NG design implies that the access right depends on the user not on the role. Moreover, the appropriate functional behavior of the access control cannot be confirmed until the design model is tested. Thus, the applied measures may not mitigate or resolve the threats and vulnerabilities. Figure 5 shows the conventional pattern application process.

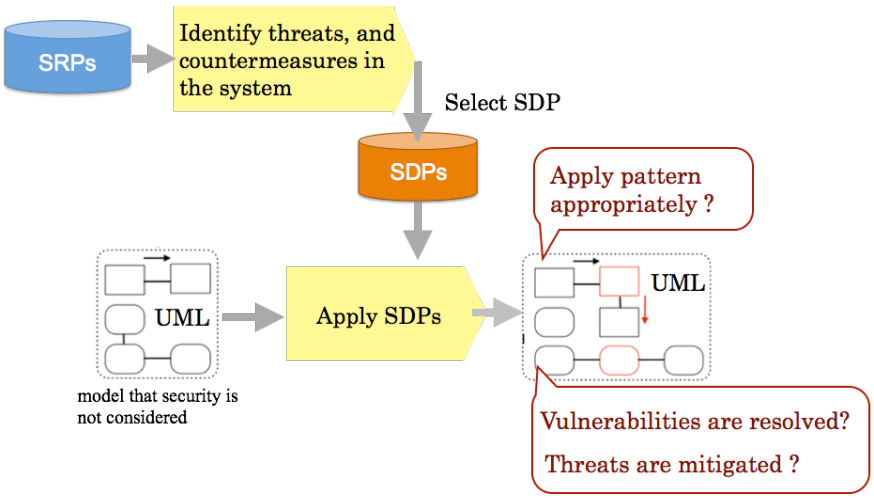

Figure 5. Conventional pattern application process

D. Test-Driven Development (TDD)

TDD is a software development technique that uses short development iterations based on prewritten test cases to define desired improvements or new functions. Here our testing process employs TDD. TDD requires that developers generate automated unit tests to define code requirements prior to writing the actual code (Choi, Kim & Yoon, 2009). The test case represents requirements that the program must satisfy (Astels, Beck, Boehm, Fraser McGregor, Newkirk & Poole, 2003).

Our method employs USE (Büttnera, Gogollaa & Richtersb, 2007), which is a tool in the UML-based simulation environment that runs tests to specify and validate information systems based on subsets of UML and OCL (Kleppe & Warmer, 1999). OCL is a semiformal language that can express constraints for a variety of software artifacts as well as specify constraints and other expressions in modeling languages. USE was initially implemented in Java at Bremen University (Germany) to evaluate OCL expressions via simulations. To verify the OCL constraints, a developer can create an instance of a class in USE and then input a value as a test case.

Our method initially executes test scripts in a design model that does not consider security in USE (Test First). These test scripts are generated automatically. Then our method detects vulnerabilities to threats identified in the requirement stage. Next, SDPs are applied, and the tests are re-executed to confirm that the vulnerabilities are resolved. In our validation method, we use OCL expressions as the requirements, and then we validate that the target model satisfies these requirements.

III. Related Works

Recently, UML-based models have been used for design. Previous research has adopted UML-based models to describe security patterns (Jurjens, Popp, & Wimmel, 2002), (Dam, Foley, Feiner & Hughes, 1995), (Finkelstein, Honiden & Yoshioka, 2004).

To model security concerns, modeling techniques such as UMLsec (Jurijens, 2005) and SecureUML (Basin, Doser & Lodderstedt, 2002) 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 specific types of attacks.

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 role-based access control (RBAC) and uses RBAC as a meta-model for specifying and enforcing security. RBAC lacks support for expressing 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, developers who are not security specialists have difficulty employing these approaches because of their peculiar semantics. Our method adopts security design patterns and it enables developers who are not security specialist to realize secure design easily compared with these approaches. Additionally, our tool enables them to design without using peculiar semantics.

There are several articles about verifying security pattern applications. Abramov at al. (2009) have suggested using a stereotype for a database application to validate security patterns. Although their method can validate applied pattern structurally, it cannot confirm that the pattern behavior in the model resolves vulnerabilities to threats.

Dongs at al. (2009) have proposed an approach to verify the compositions of security patterns using model checking. They presented the guideline to specify the behavior of security patterns in the model specification language. They defined the synchronous message, asynchronous message, and alternative flows of a UML sequence diagram and transform them into CCS specifications. Although this approach formally defines the behavioral aspect of security patterns and provide transformation scripts to check properties of the security patterns by model checking, it does not formally analyze the security requirements of the target system. Therefore, even if the verification of security patterns application can be executed appropriately, it does not guarantee that threats and vulnerabilities are resolved. On the other hand our method sets security requirements and security design requirements firstly, then we validate these requirements.

Model checking is a method to verify a formula against a logic model algorithmically (Clarke, Emerson, Edmund & Sistla, 1986). This verification technique can be automated by model checker. There are several articles about the verification of security specifications using

model-checking tools such as SPIN (Josang, 1995), (Jiang & Liu, 2008), but a specific language, which developers must learn, is necessary for model checking. Additionally, due to the general scale of development, describing security specifications using specific language is time consuming compared with our method.

IV. OUR VALIDATION METHOD

In this section, we explain our method. Firstly, we explain the process of our validation method. Next, we show examples of Ex-SRP and Ex-SDP. Finally, we apply our method to a purchasing system on the Web as an example validation process.

A. Process of Our Validation Method

Figure 6 shows the process of our method. We prepare extended SRPs (Ex-SRPs) and extended SDPs (Ex-SDPs) beforehand. These new SRPs and SDPs are expansions of existing ones, and can be used to validate whether the applied patterns are appropriate and the presence of vulnerabilities in the target model.

In this paper, we adopt UML notation to describe the target system because UML-based models are widely used for system design and most of the security design patterns describe these structure and behavior using UML notation. Additionally, we use a class diagram as a static structure and communication diagram as a dynamic behavior of the system. To separate application's concerns, we describe a system with three types of classes: boundaries, controls and entities. Boundaries are objects that interface with system actors, controls are objects that mediate between boundaries and entities, and entities are objects representing system data (Pilgrim, 2013).

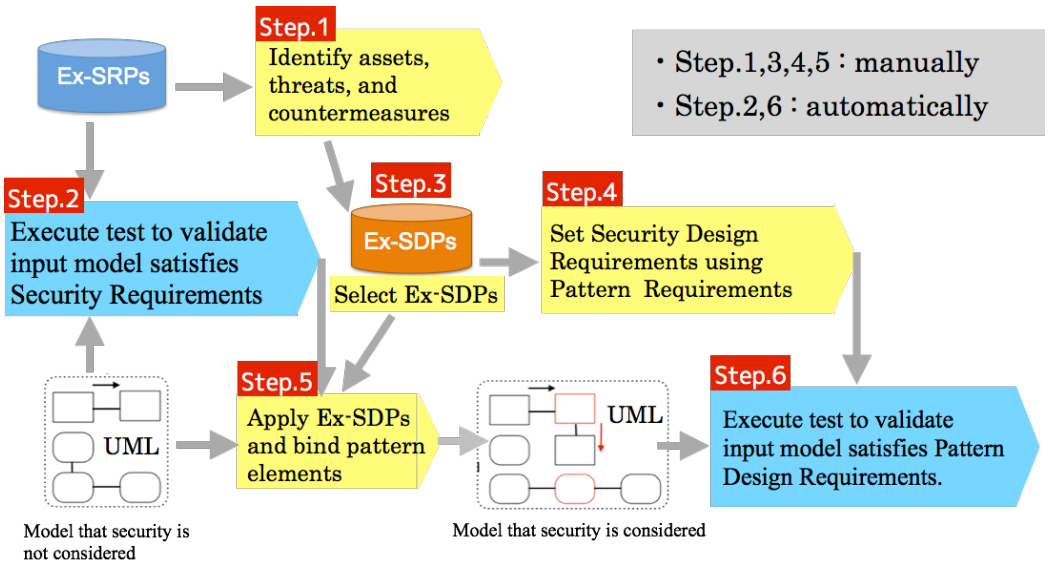

Figure 6. Process of our method

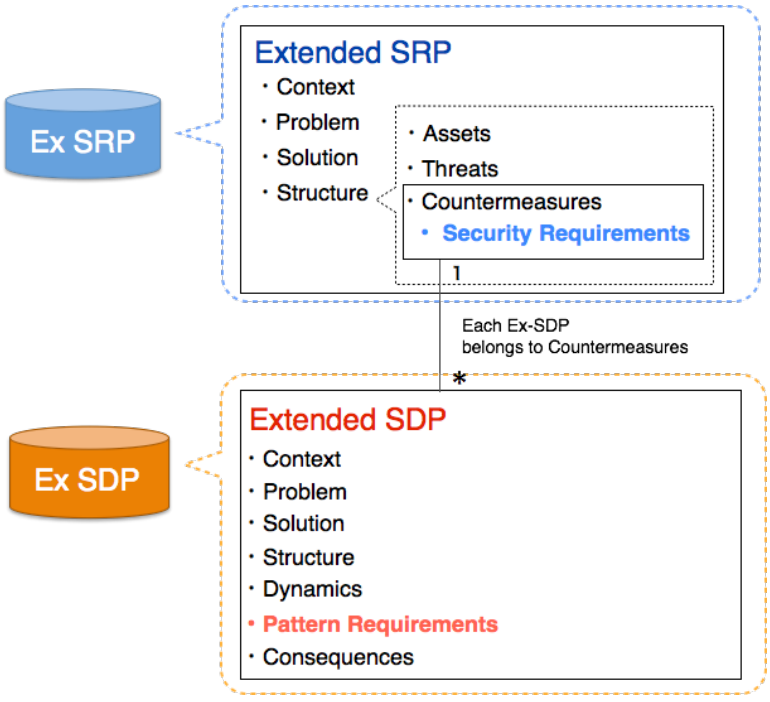

Figure 7. Overall structures of Ex-SRPs and Ex-SDPs

Figure 7 shows the overall structures of Ex-SRPs and Ex-SDPs. In addition to an existing SRP and SDP, extended patterns contain Security Requirements and Pattern Requirements, respectively. The whole purpose of SRP is identifying assets, threats, and countermeasures in the

target system. SRP consists of “Context”, “Problem”, “Solution” and Structure”. The “Structure” of SRP uses the Misuse case with the Assets and Security Goal (MASG) model like Fig. 1, which is an extension of the misuse case (Andreas & Sindre, 2000) that provides the structure of assets, threats, and countermeasures at the requirement level.

On the other hand, Ex-SDP describes how to design security measure in order to realize countermeasures identified by Ex-SRPs. For example, Role-Based Access Control, Multilevel Security and Reference Monitor (Bschmann et al, 2006) are design patterns related to Authorization. Below we briefly describe the Ex-SRPs and Ex-SDPs. Sections IV.B and IV.C provide a concrete example of patterns and the validation process, respectively.

Ex-SRPs

- **Context:** The context for the application of this pattern and system environments:
- **Problem:** The problem raised by threats against the target function:
- **Solution:** The security goal that the application of the pattern is expected to satisfy:
- **Structure:** we adopt MASG model like Fig.1 for representing this section. This model describes assets, threats, and countermeasures at the requirement level. Security Requirements is the requirements each countermeasure must satisfy. If a model does not satisfy the Security Requirements, then the measures do not remove vulnerabilities, and the system may contain threats. In TDD, these requirements represent test cases that must be satisfied.

Herein we assume that there are nine types of countermeasures: “Input and Data Validation”, “Identification and Authentication”, “Authorization”, “Configuration Management”, ”Sensitive Data”, ”Session Management”, “Cryptography”, “Exception Management”, and “Auditing and Logging”. These countermeasures can be referenced in the

Security Frame Category (Mackman & Maher, 2007), which is Microsoft's systematic categorization of threats and vulnerabilities. We assume these nine categories as typical countermeasures are at the requirement level because these categories represent the critical areas where security mistakes are most often made. Developers can use these categories to divide system architecture for further analysis, and to help them identify application vulnerabilities. Table 1 shows the explanation of each countermeasure.

Table 1.
Explanation of each countermeasure

Countermeasure	Description
Input / Data Validation	How do you know that the input your application receives is valid and safe? Input validation refers to how your application filters, scrubs or rejects input before additional processing.
Identification / Authentication	Who are you? Authentication is the process where an entity proves the identity of another entity, typically through credentials, such as a username and password.
Authorisation	What can you do? Authorisation is how your application provides access controls for resources and operations.
Sensitive Data	Who does your application run as? Which databases does it connect to? How is your application administered? How are these settings secured? Configuration management refers to how your application handles these operational issues.
Session Management	How does your application handle sensitive data? Sensitive data refers to how your application handles any data that must be protected either in memory, over the network or in persistent stores.
Cryptography	How are you keeping secrets (confidentiality)? How are you tamper-proofing your data or libraries (integrity)? How are you providing seeds for random values that must be cryptographically strong? Cryptography refers to how your application enforces confidentiality and integrity.
Exception Management	When a method call in your application fails, what does your application do? How much do you reveal? Do you return friendly error information to end users? Do you pass valuable exception information back to the caller? Does your application fail gracefully?
Auditing and Logging	Who did what and when? Auditing and logging refers to how your application records security-related events.

Ex-SDPs

- **Context:** The situation in which the pattern applies and the functional design specifications for which the countermeasures are required:
- **Problem:** The problem raised by threats against the target function:
- **Solution:** The objectives for security countermeasures:
- **Structure:** The structure must constantly satisfy the Pattern Requirements:
- **Dynamics:** The behavior must constantly satisfy the Pattern Requirements:
- **Pattern Requirements:** To meet the requirements (constraints), the structure and behavior must be satisfied when a pattern is applied. If a model does not satisfy the Pattern Requirements, then the pattern is applied inappropriately:
- **Consequences:** This section describes how the threats are mitigated by countermeasures.

Our method involves six steps (Fig. 6).

1. Identify threats and countermeasures in the system. Ex-SRPs identify the types of assets, threats, and countermeasures present in the developing software while considering the functional requirements as well as determine their associations at the requirement level.
2. Execute a test to validate that the input model satisfies the Security Requirements. A design class diagram, which does not consider security, is used to execute tests to validate whether the Security Requirements (in OCL) are satisfied because whether the input model satisfies the Security Requirements can be confirmed at this stage (i.e., vulnerabilities to threat identified at the requirement stage can be detected).
3. Select Ex-SDPs. After confirming that the target model does not satisfy the Security Requirements, Ex-SDPs related to the “countermeasures” of Ex-SRP are selected.

4. Set Security Design Requirements using Pattern Requirements. Security Design Requirements, which are combinations of each Pattern Requirement, are set as requirements that the model must satisfy.
5. Apply Ex-SDPs. Specifically, the structure and behavior of Ex-SDPs are applied to the input model that does not consider security by binding pattern elements to the model based on stereotypes.
6. Execute tests to validate the appropriateness of each Ex-SDP (i.e., whether the Security Design Requirements are satisfied is confirmed).

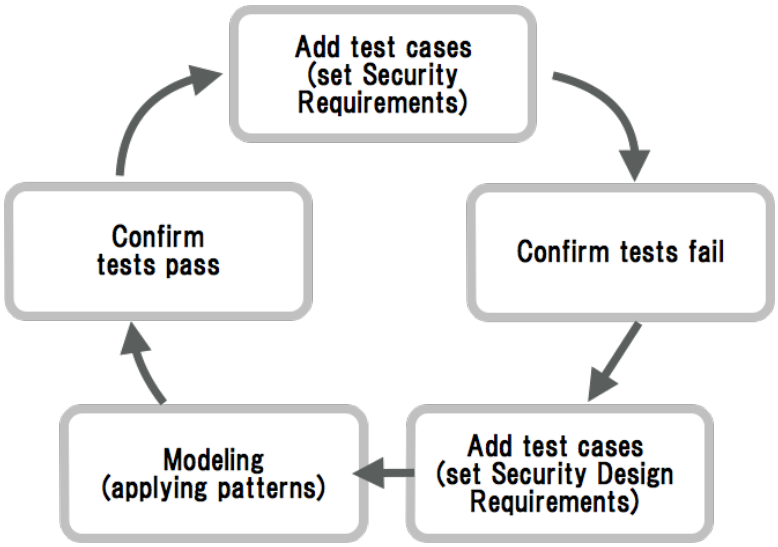

Figure 8. Testing process of our method (conceptual)

Figure 8 shows the conceptual testing process of our method, which is based on TDD. Generally, TDD is used in cord level. However our testing process employs TDD in design level.

B. Examples of Ex-SRP and Ex-SDP

In addition to explaining Ex-SRP and Ex-SDP concretely, we describe how the model uses these extended patterns to satisfy the Security and Pattern Requirements. Expansion details

are described as examples of the Authorization (countermeasure of Ex-SRP) and RBAC (Ex-SDP).

1) Authorization

Table 2 and figure 9 show the Security Requirements of Authorization, which state that only actors with access rights can execute processes that require access control. In Table 2, the concept of Authorization is described by a decision table, while Fig. 9 is an OCL statement of this concept.

Table 2.

Security Requirements of Authorization (conceptual)

		1	2
Conditions	has access right	Yes	No
Actions	execute process requiring access control.	×	
	not execute process requiring access control.		×

```
context controller
  inv Security Requirement :
   if self.UI.Actor.right = true then
    self.subject_function = true
  else
    self.subject_function = false
  endif
```

Figure 9. Security Requirements of Authorization (OCL)

In the Authorization security test, two conditions (“has access rights” and “does not have access rights”) are used to validate whether an actor can execute a target process. If an actor who does not have access rights can execute a process requiring authorization, then the target model does not satisfy the Security Requirements and the Authorization measure for the vulnerabilities

is improperly considered. Consequently, the system may be vulnerable to threats. In IV.C, we explain the concrete conditions for test execution.

Figure 10 shows an example of the test script for model testing. In our tool, developers can set attributes, methods, and the relation of each element. Then design target system is developed using UML notation. Additionally, developer can input concrete test cases into the target model and our tool creates the test script (Fig. 10), which is translated to execute the test in USE. To use this test script, a developer can create instances of an input model and validate whether above Security Requirements (in OCL) are satisfied.

```
----------Create instances
!create Actor_1 : Actor
!create UI : UI
!create Subject_function : Subject_function
!create entity_1 : entity
    .
    .
--------- Insert associations
!insert (Actor_1, UI) into assignedTo
    .
    .
--------- Set Test Case
!set Actor_1.name := 'XXXX'
!set Actor_1.right := true
!set entity_1.attribute := y
    .
    .
--------- Execute Method
!openter Subject_function subject_function()
```

Figure 10. Example of the test script

2) Role-Based Access Control (RBAC)

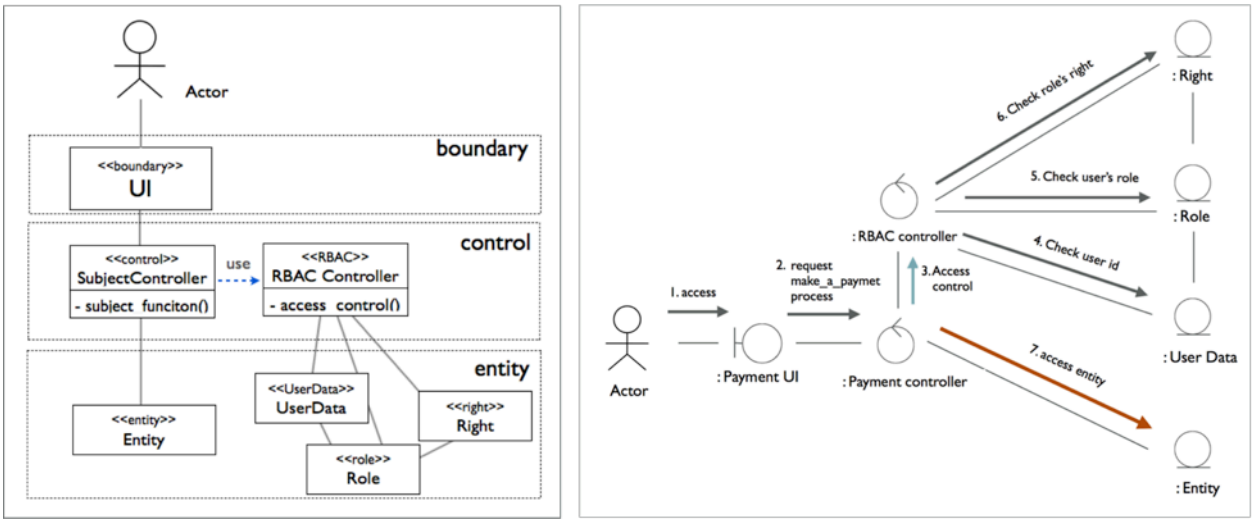

Figure 11. Structure and behavior of RBAC

Figure 11 shows the structure and behavior of RBAC. In our method, we represent pattern elements using stereotypes. In RBAC, stereotypes, such as <<RBAC>>, <<User Data >>, <<Role>> and <<Right>>, are elements of the pattern. When a developer applies a pattern, pattern elements are bound based on stereotypes.

In Fig. 11, “Subject Controller” using the RBAC controller behaves as access control. To employ RBAC, if rights are not specified in the role that an actor belongs, this system assumes that the actor does not have access permission and the actor cannot execute processes requiring Authorization. This security capability is realized because this pattern satisfies the Pattern Requirements of RBAC.

Table 3.

Pattern Requirements of RBAC (conceptual)

		1	2
Conditions	rights are given in <<Role>> which an <<User_Data>> belongs	Yes	No
Actions	<<Actor>> is considered to have access permission.	×	
	<<Actor>> is not considered to have access permission.		×
	execute process requiring access control.	×	
	not execute process requiring access control.		×

```
context subject_controller
 inv access_control:
  if self.RBAC.Right->exists(p |
        p.right = true and
        p.role_id = p.Role.id and
        p.role_id = p.Role.User_Data.role_id )
 then
    self.UI.Actor.right = true and self.subject_function = true
 else
    self.UI.Actor.right = false and self.subject_function = false
 endif
```

Figure 12. Pattern Requirements of RBAC (OCL)

Table 3 and Fig. 12 show the Pattern Requirements of RBAC in the decision table and the OCL statement form, respectively for “if rights are given in role which an actor belongs, then the actor is considered to have access permission and actor can access asset”.

Figure 13 shows the test script to validate whether the model-applied pattern satisfies the Pattern Requirements. Similar to Fig. 10, a developer provides concrete test cases to the target model in our tool, then our tool outputs the test script, which is then translated to execute a test in USE. Both the Security and Pattern Requirements can be validated simultaneously using an OCL statement. The former determines the presence of vulnerabilities at the design level, while the latter confirms if the pattern is appropriately applied.

```
----------Create instances
!create Actor_1 : Actor
!create UI : UI
!create Subject_function : Subject_function
!create RBAC : RBAC
!create User_Data : User_Data
!create Role : Role
!create Right : Right
    .
    .
---------------------------
--------- Insert associations
!insert (Actor_1, UI) into assignedTo
    .
    .
---------------------------
--------- Set Test Case
!set Actor_1.name := XXXX

!set User_Data.id := z
!set User_Data.pass := 'xxxxxx'
!set User_Data.name := 'XXXX'

!set Role.id := y
!set Role.name := 'xxxxxx'

!set Right.id := x
!set Right.role_id := y
       .
       .
---------------------------

--------- Execute Method
!openter RBAC access_control()
!openter Subject_function subject_function()
```

Figure 13. Example of a test script

C. Example of the Validation Process

To confirm that our method realizes secure design, here we Jiang our method to a purchasing system on the Web as an example validation process. We assumed the assets, threats, and countermeasures using the MASG model in Fig. 1.We initially identified and modeled the assets, threats, and countermeasures in the system by referring to the Ex-SRPs of the requirement called "commercial transaction on the Web". Next we executed tests of the model that security is not considered to validate whether vulnerabilities to threats identified by Ex-SRPs are detected.

Figure 14 shows the model that does not consider security and Table. 4 show the explanation of each element in this model. In Fig. 14, the system does not have a function to

check the condition to execute “make a payment” process. In other words, even if this user is not a regular user, the process can be executed.

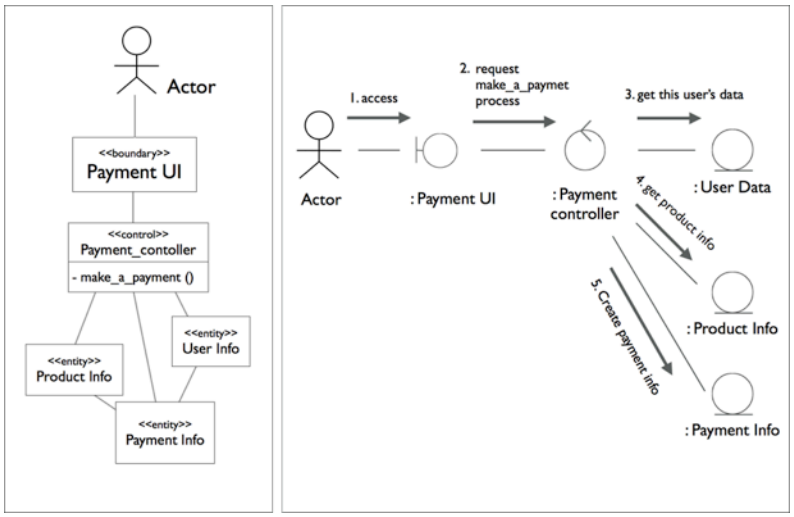

Figure 14. Model that does not consider security

Table 4.

Explanation of each element

Element	Stereotype	Description
Payment UI	boundary	Interface that payment informations are input into.
Payment Controller	control	Element that has a “make a payment” method.
User Info	entity	Element that contains user’s data such as name and gender.
Product Info	entity	Element that contains product data.
Payment Info	entity	Element that contains payment data. when the actor finishes make a payment process, this instance will be created.

After confirming that these vulnerabilities exist in the input model, we set Security Design Requirements and applied Ex-SDPs. Finally we executed tests to confirm that the Security Design Requirements are satisfied due to an appropriately applied pattern.

Step 1: Identify threats and countermeasures in the system.

Referring to Fig. 1, “I&A”, “Input and Data Validation”, and “Authorization” are countermeasures for “Spoofing”, “Elevation of Privilege”, and “SQL Injection” in the “make a payment” process, respectively. For simplicity, each threat has one countermeasure.

Step 2: Execute a test to validate that the input model satisfies the Security Requirements.

Then by referencing the Security Requirements used for each Ex-SRP countermeasure, the set for the Security Requirements should be satisfied in the “make a payment” process. Table 5 and Fig. 15 show the Security Requirements for the “make a payment” process.

Table 5.

Security Requirements for the “make a payment” process (conceptual)

		1	2	3	4	5	6	7	8
Conditions	<<Actor>> is regular user	Yes	Yes	Yes	Yes	No	No	No	No
	<<Actor>> has access right	Yes	Yes	No	No	Yes	Yes	No	No
	valid data is inputed in <<UI>>	Yes	No	Yes	No	Yes	No	Yes	No
Actions	execute “make a payment” process	×							
	not execute “make a payment” process		×	×	×	×	×	×	×

```
context payment_controller
 inv SecurityRequirement :
  if self.payment_UI.User.registered_user = true and
     self.payment_UI.User.right = true and
     self.payment_UI.is_safe_input = true then
   self.make_a_payment = true
 else
   self.make_a_payment = false
 endif
```

Figure 15. Security Requirements for the “make a payment” process (OCL)

The Security Requirements for the “make a payment” process include: “actor is regular user”, “actor has access permission”, and “valid data is inputted”. If these requirements, which are a combination of “I&A”, “Input and Data Validation”, and “Authorization”, are met, then the

actor can execute the “make a payment process”. These requirements represent test cases in the TDD process.

Next, we executed a model test to determine whether the input model that does not consider security satisfies the Security Requirements in Fig. 15 (i.e., we validate whether each test case 1 ~ 8 behaves according to the expected action in Table 5). If the Security Requirements are not satisfied, then the appropriate countermeasures are not taken, and the threats identified using Ex-SRP may exist.

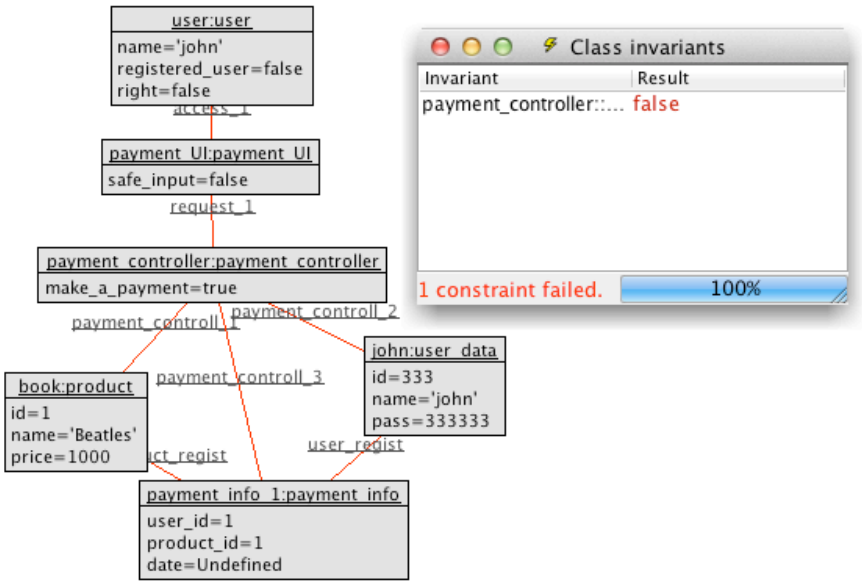

Figure 16. Conditions of the Security Test in USE

Figure 16 shows a case where the “regular user”, “has access permission”, and “uses valid input data” are all “false” (Table 5, test case 8). Because the input model lacks object constraints, an actor may carry out “make_a_payment = true” (i.e., an actor can execute the “make a payment” process without being a regular user or permission). Hence, the input model not considering security does not satisfy the Security Requirements of the “make a payment” process, and the OCL evaluation in USE becomes "false" in Fig. 16.

Table 6 shows the results of the eight test cases where only case 1 satisfies the Security Requirements in Table 5 and Fig. 15, confirming the necessity of countermeasures “I&A”, “Authorization” and “Input and Data Validation”.

Table 6.

Results of the Security Test

		1	2	3	4	5	6	7	8
Conditions	<<Actor>> is regular user	Yes	Yes	Yes	Yes	No	No	No	No
	<<Actor>> has access right	Yes	Yes	No	No	Yes	Yes	No	No
	valid data is inputed in <<UI>>	Yes	No	Yes	No	Yes	No	Yes	No
Actions	execute “make a payment” process	×	×	×	×	×	×	×	×
	not execute “make a payment” process								

Step 3: Select Ex-SDPs.

We selected Ex-SDP related to the countermeasures of Ex-SRP, and then adding these to the structure to realize security capabilities. We selected Password design and Use, RBAC and Prevent SQL Injection for “I&A”, “Authorization” and “Input and Data Validation”, respectably.

Step 4: Set Security Design Requirements using the Pattern Requirements.

Table 6 and Fig. 17 show the combinations of each Pattern Requirement necessary for the “make a payment” process, which are referred to as “Security Design Requirements”.

Table 6.

Security Design Requirements of the “make a payment” process (conceptual)

		1	2	3	4	5	6	7	8
Conditions	the same ID and Password that are inputted into <<Login_UI>> exist in <<User_Data>> respectively	Yes	Yes	Yes	Yes	No	No	No	No
	rights are given in <<Role>> which an <<User_Data>> belongs	Yes	Yes	No	No	Yes	Yes	No	No
	valid data is inputed in <<UI>>	Yes	No	Yes	No	Yes	No	Yes	No
Actions	<<Actor>> is considered regular user	×	×	×	×				
	<<Actor>> is consider non-regular user					×	×	×	×
	considers that <<Actor>> have access permission	×	×			×	×		
	consider that <<Actor>> does not have access permission			×	×			×	×
	consider that valid data is inputed	×		×		×		×	
	consider that invalid data is inputed		×		×		×		×
	execute “make a payment” process	×							
	not execute “make a payment” process		×	×	×	×	×	×	×

```
context payment_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.Payment_UI.actor.regular_user = true
  else
      self.Payment_UI.actor.regular_user = false
  endif

context payment_controller
  inv access_control:
    if self.RBAC.Right->exists(p |
          p.right = true and
          p.role_id = p.Role.id and
          p.role_id = p.Role.User_Data.role_id )
  then
      self.Payment_UI.actor.right = true
  else
      self.Payment_UI.actor.right = false
  endif

context payment_controller
  inv sanitize_input_data_payment_UI:
    if self.Payment_UI.Prevent_SQL_Injection.sanitize_input_data = true
  then
      self.Payment_UI.valid_input_data = true
  else
      self.Payment_UI.valid_input_data = false
  endif

context payment_controller
  inv sanitize_input_data_login_UI:
    if self.password_design_and_use.Login_UI.Prevent_SQL_Injection.sanitize_input_data = true
  then
      self.password_design_and_use.Login_UI.valid_input_data = true
  else
      self.password_design_and_use.Login_UI.valid_input_data = false
  endif

context payment_controller
  inv security design requirement:
    if self.Payment_UI.actor.regular_user = true and
       self.Payment_UI.actor.right = true and
       self.Payment_UI.valid_input_data = true and
       self.password_design_and_use.Login_UI.valid_input_data = true
  then
      self.make_a_payment = true
  else
      self.make_a_payment = false
  endif
```

Figure 17. Security Design Requirements of the “make a payment” process (OCL)

Step 5: Apply Ex-SDPs and bind pattern elements.

We applied these Ex-SDPs (i.e., “Password Design and Use", "RBAC”, and "Prevent SQL Injection"). During pattern application, we bind these pattern elements to a stereotype in our tool. Figure 18 shows the conditions to apply a pattern using our tool, while figure 19 and 20 show the structure and behavior after applying the patterns to model that does not consider security. In other words, these are models that consider security. As compared with the model in Fig. 14, there are several conditions (see Table. 6) to execute make a payment process. Table 7 shows the explanation of added element.

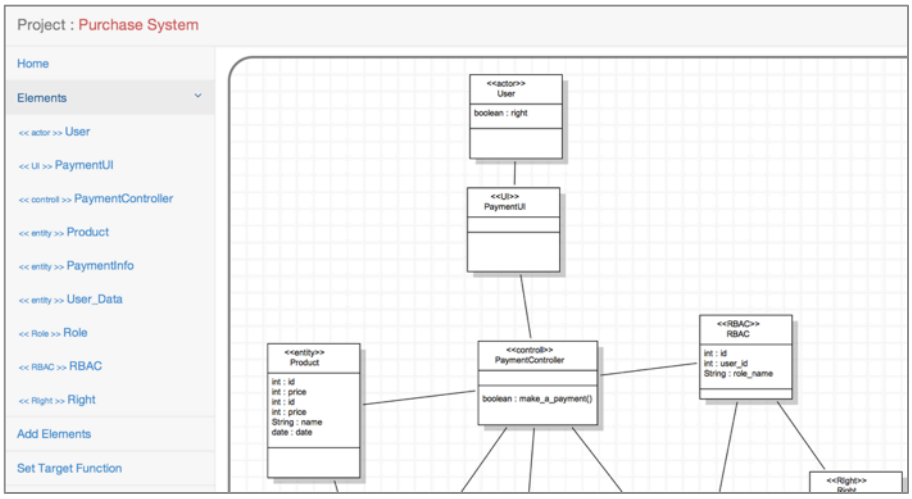

Figure 18. Conditions of pattern application in our tool

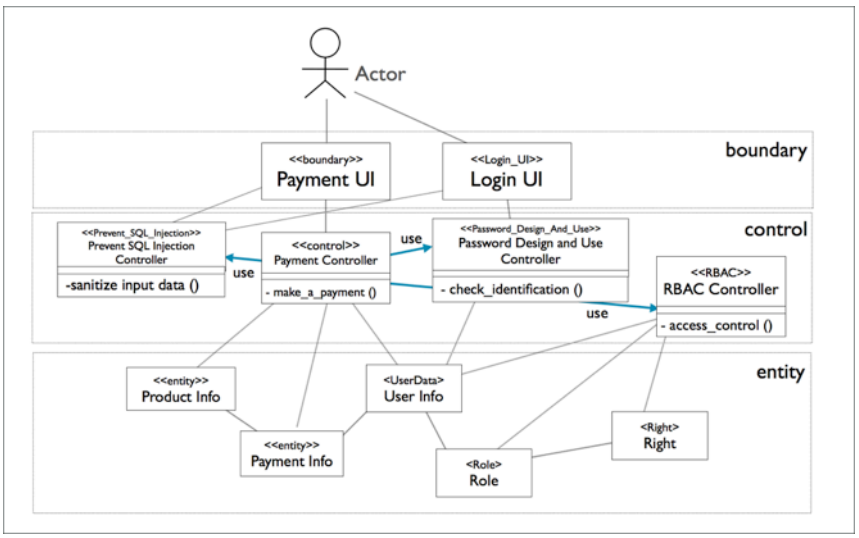

Figure 19. Model that consider security (structure)

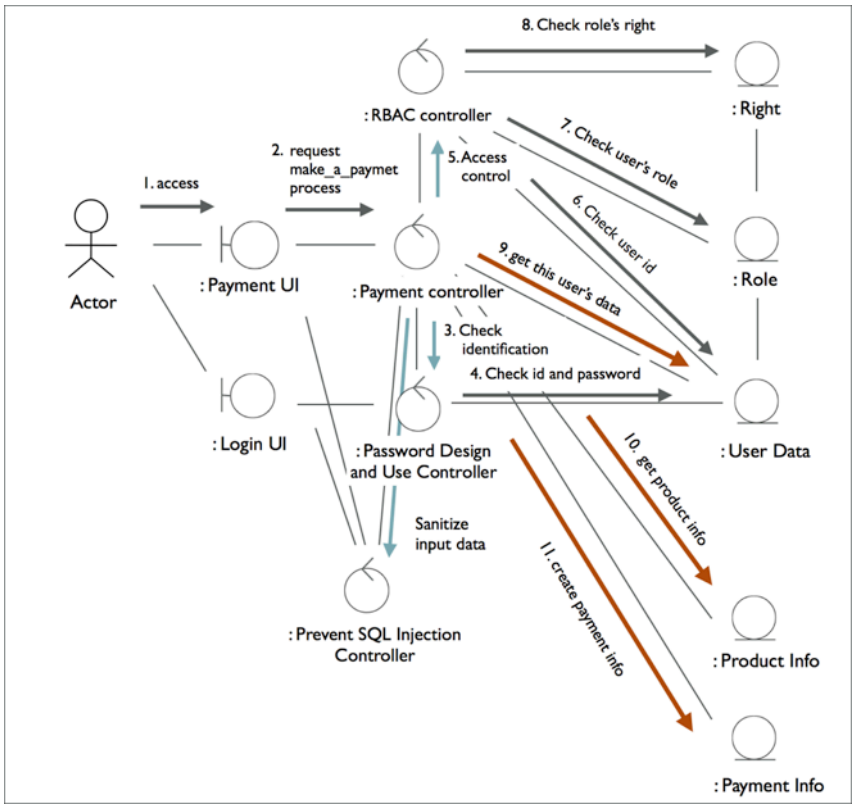

Figure 20. Model that consider security (behavior)

Table 7.

Explanation of added element

Element	Stereotype	Description
Login UI	boundary	Interface that ID and PASSWORD are input into.
Password Design and Use Controller	control	Element that has a "check identification" method. This method returns true when ID and PASSWORD match.
RBAC Controller	control	Element that has a "access_control" method. This method returns true when access permission is given in Role that an actor belongs.
Prevent SQL Injection Controller	control	Element that has a "sanitize input data" method. This method returns true when all input data are sanitized in user interface.
Role	entity	Element that contains role informations which user belongs.
Right	entity	Element that contains access permission informations which role has.

Step 6: Execute test to validate input model satisfies the Security Design Requirements.

To validate whether the patterns are applied appropriately to the "make a payment" process, each Pattern Requirement must be validated (i.e., the structure and behavior of the model must be confirmed after applying the patterns). We executed model tests to confirm that the model in Fig. 19, 20 satisfies the Security Design Requirements in Fig. 17. Specifically, we confirmed that test cases 1 ~ 8 behave as expected (Table 6). First, concrete test cases are inputted into the model created in Step 4, which generates a test script. Then this script is translated to execute test in USE. Finally the OCL statement using this test script in USE is evaluated. Figure 21 shows the conditions of the Security Design Test in USE.

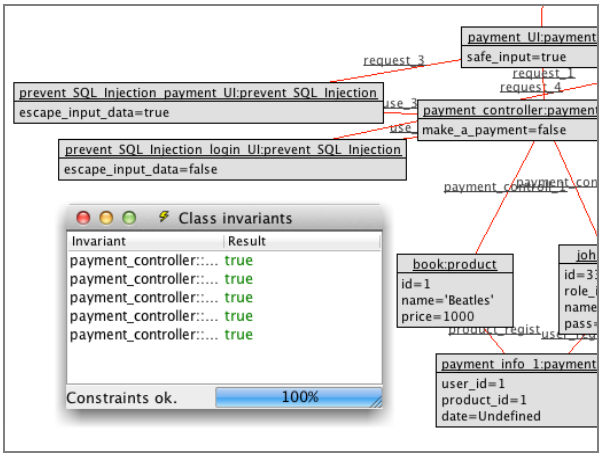

Figure 21. Conditions of the Security Design Test in USE

Figure 21 shows a case where access permission is not given for the “Role” of the actor, but and the system does not sanitize the input data in “Login UI” (Table 6, test case 4). Prior to applying patterns, USE outputs “make_a_payment = true” (i.e., an actor can execute the “make a payment” process, even if the actor does not have permission or inputs invalid data). After the patterns are applied, USE outputs "make a payment = false", and the actor cannot execute the “make_a_payment” process because access permission is not specified in the “Role” and the system assumes invalid data is used in “Login UI”. Consequently, the OCL statements are true in Fig. 21. By executing all the test cases, we confirm that the output model after the pattern application satisfies the Security Design Requirements of the “make a payment” process.

To summarize, we applied Ex-SDPs for the “make_a_payment” process, which requires “I&A”, “Input Data and Validation”, and “Authorization”, and then executed a model test. If patterns are applied appropriately, then the output model will simultaneously satisfy the Security Design Requirements and the Security Requirements. The initial input model did not satisfy the Security Requirements of the “make a payment” process. However, the output model applied patterns to satisfy the Security Design Requirements of the “make a payment” process. In this

manner, the appropriate application of security design patterns and the existence of vulnerabilities to threats identified at a requirements stage can be validated before and after pattern application.

D. Limitations

Our method has a few limitations. Because test cases are created based on threats and countermeasures identified in the requirement stage, the presence of threats not identified in the requirement stage cannot be verified. In addition, the criterion for selecting Ex-SDP may be impractical because the range is influenced by the security policy, platform, and risk analysis.

V. CASE STUDY AND DISCUSSION

Here we apply our method to a student information management system (Kaiya, Kobashi, Okubo, Washizaki & Yochioka, 2013) as a case study and to evaluate RQ1 and RQ2. This system, which does not directly consider security, was created by a masters course student majoring in software engineering. Table 8 shows the scale of the target system, and Figure 22 shows a model of the target system.

Table 8. Scale of the target system

Number of Use Case	24
Number of Class	31 (View : 18, Controller : 5, Model : 8)
Development Time	3 months

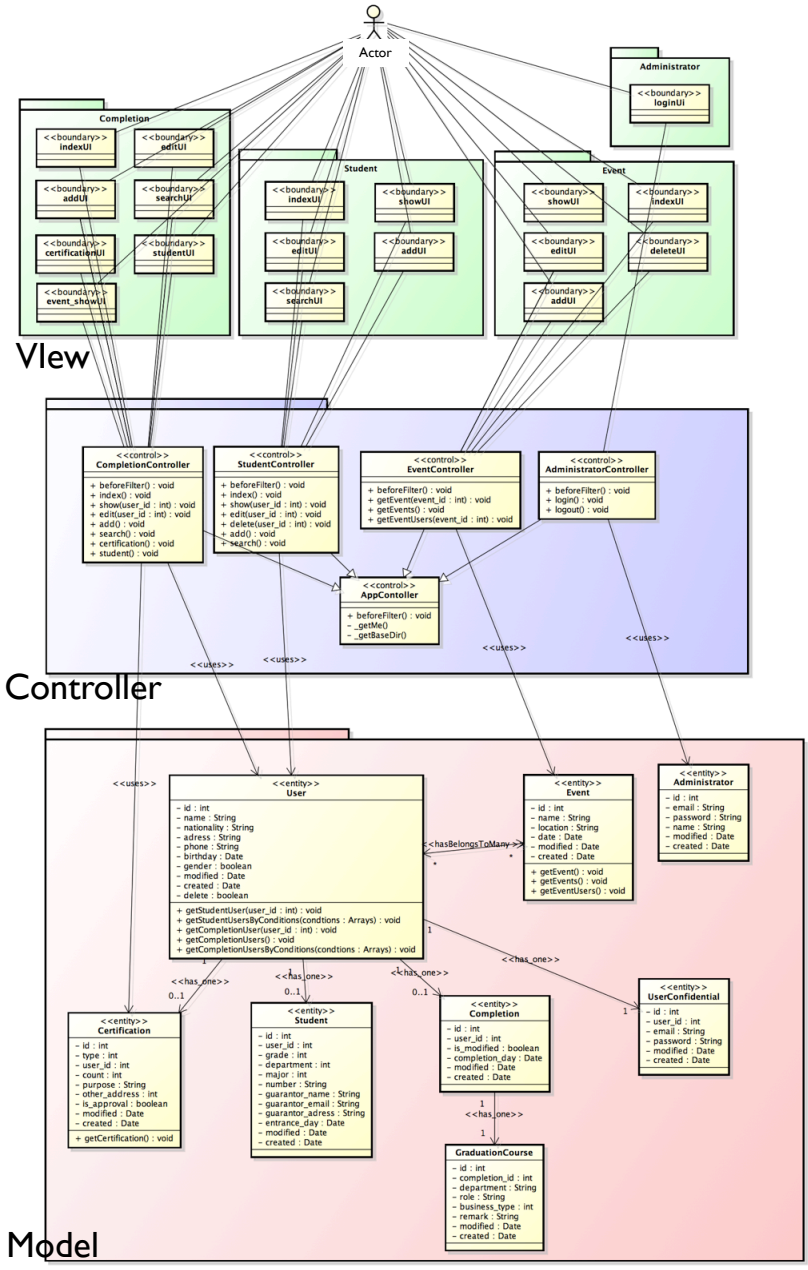

Figure 22. Model of the student information management system

In the case study, we applied our method to the “delete function” of the “Student Controller” (Fig. 22). After referencing the Ex-SRP, we identified threats to the delete process (“Elevation of privilege” and “SQL Injection”) and assumed that “Authorization” and “Input and Data Validation” are effective countermeasures. In other words, when an actor deletes student

data, the system lacks a function to determine whether the actor has a permission to do so or if the inputted data is valid. Referencing Ex-SRP, we set the Security Requirements, which are a combination of "Authorization" and "Input and Data Validation".

```
context StudentController
 inv SecurityRequirement :
  if self.DeleteUI.Actor.right = true and
     self.DeleteUI.Actor.valid_input_data = true then
   self.delete = true
 else
   self.delete = false
 endif
```

Figure 23. Security Requirements of the "delete student data" process (OCL)

We then validated whether the input model satisfies the Security Requirements and whether the vulnerabilities are resolved upon applying the patterns. After confirming that the input model does not satisfy the Security Requirements, we applied Ex-SDPs. We selected "RBAC" and "Prevent SQL Injection" as Ex-SDPs. The model applied patterns to realize an "Authorization" function because "Student Controller" calls elements of <<RBAC>> to verify the actor's role as rights depend on the role. In addition, this model can realize an "Input and Data Validation" function via the sanitizing process in <<Delete_UI>>.

To confirm whether the structure and behavior of the applied patterns operate appropriately, we validated the Security Design Requirements (Table 9) of the "delete student data" process using model tests.

Table 9. Security Design Requirements of the "delete student data" process (conceptual)

		1	2	3	4
Conditions	rights are given in <<Role>> which an <<User_Data>> belongs	Yes	Yes	No	No
	valid data is inputed in <<UI>>	Yes	No	Yes	No
Actions	considers that <<Actor>> have access permission	×	×		
	considers that <<Actor>> dose not have access permission			×	×
	consider that valid data is inputed	×		×	
	consider that invalid data is inputed		×		×
	execute "Delete Student Data" process	×			
	not execute "Delete Student Data" process		×	×	×

We executed model tests for the four test cases in Table 9 and confirmed that the model after pattern application satisfies the Security Design Requirements. Thus, our method correctly applies patterns (**RQ1**) in both the "delete student data" process and in III.C.

Finally we validated whether the model after pattern application satisfies the Security Requirements of the "delete student data" process. To satisfy Security Design Requirements, the target model must satisfy the Security Requirements. Consequently, the first test confirmed the existence of vulnerabilities identified in the requirement stage before pattern application, while the second test validated that the vulnerabilities are removed after pattern applications. Thus, the proposed method answers **RQ2**.

VI. THREATS TO VALIDITY

A. Threats to internal validity

In the case study, a developer familiar with our method or security patterns applies patterns. Therefore, there is a possibility that general developers could proceed with our process inappropriately. Additionally, even if they validate pattern application and resolve vulnerabilities appropriately, it takes long time rather than other existing approaches because of lack of the understanding of our method. Although our tool provides a flow of the validation process and it also executes test automatically, we should confirm that the expected outcome is achieved by developers unfamiliar with our method.

B. Threats to external validity

We did not verify whether our method is applicable to any type of system. Therefore, it is difficult to generalize the case study results. Moreover, the number of security patterns we used and experiment tester is not enough. Hence, it is possible that several security patterns could be not available in our method. Although we used representative patterns and a typical model for software development and confirmed that our method is useful for that, we should confirm that our method is applicable to more general patterns and large-scale examples.

VII. CONCLUSION AND FUTURE WORK

If a software developer is not a security expert, patterns may be inappropriately applied. Additionally, threats and vulnerabilities may not be mitigated even if patterns are applied correctly. Herein we propose a validation method for a security design pattern using a model test in the UML model simulation environment. Specifically, assets, threats, and countermeasures are identified in the target system during an early stage of development. We validated both the

appropriateness of the applied patterns and the existence of vulnerabilities identified in the first stage of the design model.

This method offers three significant contributions. First, Ex-SRP and Ex-SDP, which are new extended security patterns using OCL expressions, include requirement- and design-level patterns. Second, a new model-testing process based on TDD validates correct pattern applications and the existence of vulnerabilities using these extended patterns. Finally, a tool to support pattern application automatically generates a script to test the model. In the future, we intend to experiment using more general and large-scale examples as well as consider applications based on the dependencies among patterns, which should realize more practical uses.

REFERENCES

Maruyama, k., Washizaki, H., & Yoshioka, N. (2008). *A Survey on Security Patterns* (pp. 35-47).

Bschmann, F., Fernandez-Buglioni, E., Schumacher, M., Sommerlad, P., & Hybertson, D. (2006). *SECURITY PATTERNS : Integrating Security and Systems Engineering (Wiley Software Patterns Series)*.

Heyman, T., Joosen, W., Scandariato, R., & Yskout, K. (2007). An analysis of the security patterns landscape. in Proceedings of the Third International Workshop on Software Engineering for Secure Systems, ser. SESS ’07. IEEE Computer Society. doi: 10.1109/SESS.2007.4.

Lai, R., Nagappan, R., & Steel, C. (2005). Core Security Patterns: *Best Practices and Strategies for J2EE, Web Services, and Identity Management*.

Bondareva, K., & Milutinovich, J. (2014). *Unified Modeling Language*. Retrieved September 10, 2014, from http://www.omg.org/gettingstarted/what_is_uml.htm.

Kaiya, H., Okubo, T., & Yoshioka, N. (2012). N. Effective Security Impact Analysis with Patterns for SoftwareEnhancement. IJSSE 3(1): 37-61 2012. doi: 10.1109/ARES.2011.79.

Okubo, T., Taguch, K., & Yoshioka, N. (2009). Misuse Cases + Assets + Security Goals. International Conference on Computational Science and Engineering. doi: 10.1109/CSE.2009.18.

Andreas L., & Sindre, G. (2000). Eliciting security requirements by misuse cases. IEEE Computer Society. doi: 10.1109/TOOLS.2000.891363.

Dong, J., Peng, T., & Zhao, Y. (2008). Verifying Behavioral Correctness of Design Pattern Implementation. SEKE, page 454-459.

Abramov, J., Shoval, P., & Sturm, A. (2009). Validating and Implementing Security Patterns for Database Applications. SPAQu.

Dong, J., Peng, T., & Zhao, Y. (2009). Automated verification of security pattern compositions. Information and Software Technology, vol 52, pages 274–295. doi: 10.1016/j.infsof.2009.10.001

Choi, B., Kim, H., & Yoon, S. (2009). Performance testing based on test-driven development for mobile applications. ICUIMC. doi: 10.1145/1516241.1516349.

Astels, D., Beck, K., Boehm, B., Fraser, S., McGregor, J., Newkirk, J., & Poole. C. (2003). Discipline and practices of TDD *: (test driven development)*. OOPSLA. doi: 10.1145/949344.949407.

Büttnera, F., Gogollaa, M., & Richtersb,M. (2007). USE: *a UML-based specification environment for validating UML and OCL.* Science of Computer Programming (vol 69). doi: 10.1016/j.scico.2007.01.013.

Kleppe. A & Warmer, J. (1999). The Object Constraint Language: *Precise Modeling with UML* (Addison-Wesley Object Technology Series).

Ju¨rjens, J., Popp, G., & Wimmel, G. (2002). Towards using security patterns in model-based system development. PLoP. doi: 10.1.1.18.8894.

Adamczyk, P., Hafiz, M., & Johnson, R. (2007). Organizing security patterns. IEEE Software, vol.24, no.4, pp.52– 60. doi: 10.1109/MS.2007.114.

Finkelstein, A., Honiden, S., & Yoshioka, N. (2004). Security patterns: *a method for constructing secure and efficient inter-company coordination systems*. Enterprise Distributed Object Computing Conference pp.84–97. doi: 10.1109/EDOC.2004.1342507.

Dam, A., Foley, J. D., Feiner, S., & Hughes, J. (1995). Computer Graphics: Principles and

Practice in C (2nd Edition).

Jurijens, J. (2005). Secure Systems Development with UML.

Basin, D., Doser, J., & Lodderstedt, T. (2002). SecureUML: *A UML-Based Modeling Language for Model-Driven Security*. doi: 10.1007/3-540-45800-X_33.

Clarke, E., Emerson, A., Edmund, M., & Sistla, A. (1986). Automatic verification of finite- state concurrent systems using temporal logic specifications. doi: 10.1145/5397.5399.

Josang, A. (1995). Security protocol verification using spin.INRS-Telecommunications, Montreal, Canada.

Jiang, Y., & Liu, X. (2008). Formal analysis for network security properties on a trace semantics. doi: 10.1109/ICACTE.2008.31.

Pilgrim, P. (2013) Java EE 7 Developer Handbook. Paperback, Packt Publishing.

Mackman, A., & Maher, P. (2007). Web Application Security Frame. Microsoft Patterns & Practices. http://msdn.microsoft.com/en-us/library/ms978518.

Kaiya, H., Kobashi.T., Okubo, T. Washizaki, H., & Yochioka, N. (2013). SSR-Project. https://github.com/SSR-Project