

**Using change patterns to incorporate
evolving trust relationships into a
software architecture**

*Koen Yskout Riccardo Scandariato
Wouter Joosen*

Report CW 576, March 2010



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Using change patterns to incorporate evolving trust relationships into a software architecture

Koen Yskout Riccardo Scandariato
Wouter Joosen

Report CW 576, March 2010

Department of Computer Science, K.U.Leuven

Abstract

When designing a secure software architecture, the architect must take possible evolution of the system and its environment into account. Inevitably, during the lifetime of the application, changes will occur that reduce the security of the system. It is crucial that these changes are anticipated, and that they can be accommodated with minimal impact on the architecture. This report introduces the concept of ‘change patterns’, providing guidance to the architect to achieve this goal.

A change pattern guides the architect in designing an architecture that is resistant against certain foreseen evolutions of the requirements and assumptions. It explicitly records the change of requirements or assumptions it supports. After the architect has selected appropriate change patterns, applying a pattern consists of two steps. First, the architecture needs to be prepared up-front for the evolution (even though it has not yet occurred), based on a likelihood and importance analysis of the evolution. Second, once the evolution occurs, the architect is triggered to perform the necessary steps to update the application such that it conforms to the new situation. These two steps are reflected in the solutions that belong to the change pattern: architectural patterns for the up-front preparation, and change guidance for performing the actual update of the application.

This report contains a description of a change pattern, an outline of a process to use change patterns, and a catalogue of eight change patterns for evolving trust relationships, and an illustration of their use.

Keywords : change patterns, software architecture, security, evolution, trust.

Contents

1	Introduction	4
1.1	Architectural changes	4
1.2	Background	6
1.2.1	Software architecture	6
1.2.2	Architecture and security	6
1.3	Outline	7
2	Change patterns	8
2.1	Motivating example	8
2.2	Change pattern structure	11
2.3	Process description	11
2.3.1	Automation	13
3	Trust evolution	15
3.1	Evolving trust scenarios	15
3.2	Mapping trust requirements to architecture	16
3.3	Change pattern catalogue for trust	16
3.3.1	Evolving trust of execution upon external actor	19
3.3.1.1	Solution 1: Require commitment	20
3.3.1.2	Solution 2: Use monitoring	24
3.3.2	Evolving trust of execution from external actor	27
3.3.2.1	Solution 1: Provide commitment	27
3.3.2.2	Solution 2: Enable monitoring	30
3.3.3	Evolving trust of permission upon external actor	33
3.3.3.1	Solution 1: Apply least-privilege principle	33
3.3.3.2	Solution 2: Attribute-based access control	33
3.3.3.3	Solution 3: Use monitoring	36
3.3.4	Evolving trust of permission from external actor	37
3.3.4.1	Solution 1: Request confirmation	37
3.3.4.2	Solution 2: Enable monitoring	39
3.3.5	Delegate execution of a service to a trusted actor	39
3.3.5.1	Solution: Encapsulate service	40
3.3.6	Delegate permission to a service to a trusted actor	40
3.3.6.1	Solution: Flexible access control	40
3.3.7	Providing additional service with delegated execution	44
3.3.7.1	Solution: Introduce bridge component	44
3.3.8	Providing additional service with delegated permission	44
3.4	Illustration	47
3.4.1	Initial situation	47
3.4.2	Applying change patterns	47
3.4.3	Handling evolution	50
4	Conclusion	52

Chapter 1

Introduction

1.1 Architectural changes

During the lifetime of the system, its requirements will undoubtedly change, as well as the assumptions that were made about the application's environment. These changes may very well have an impact on the security of the system. This is definitely the case when a security-relevant requirement or assumption changes; something in the system will then have to change to ensure that the desired security properties of the system are maintained.

While it may be possible to fulfil the updated requirement without changing the existing architecture, for example by some localized changes to configuration, implementation or protocols, sometimes (significant) alterations to the architecture are necessary. Since in the architectural phase, the most substantial decisions are made regarding the system that is being developed, it is important to understand the nature and impact of changes regarding the architecture.

Security-related changes in the architecture can be triggered by multiple events. Since software design is an iterative process, changes to an architecture are firstly possible because of problems or constraints that only arise during the implementation or deployment of the software. Besides better planning or prototyping, not much can be done to lower this impact, and therefore we will not consider this cause of architectural change any further.

The other important source of architectural evolution, often leading to major adaptations of the architecture, are changes coming from the artefacts generated by the earlier phases of software development, i.e., requirements engineering. This is also true for the security-related aspects. In particular, we discern the following cases.

- **Changes in the functional requirements** When a functional requirement changes, this will often have an impact on the security properties associated with that requirement. For instance, a newly introduced feature of the system may need to be protected from unauthorized users. Also, new features can interact with other features, giving rise to new vulnerabilities.
- **Changes in the security requirements** A changing security requirement will, by definition, lead to a security-related change in order to fulfil it. For example, a previously unprotected piece of information that now needs protection requires that mechanisms are put in place to take care of this protection.
- **Changed assumptions about the environment** The security of an application is always based on security assumptions about the environment in which it would serve. These assumptions may change for various reasons. For instance, the application may simply become deployed in a new environment, in which these security assumptions do not hold. Even within the same environment, the environment's properties can evolve. Equally, a better risk analysis may have been performed, invalidating some assumptions about the target environment (or giving rise to new assumptions, that were not thought to be viable before). All these changes may lead to evolution of the architecture.

The impact of an architectural change can be dramatic, especially when the system is almost entirely implemented, or, worse, already deployed. Unfortunately, it is impossible to create an architecture that

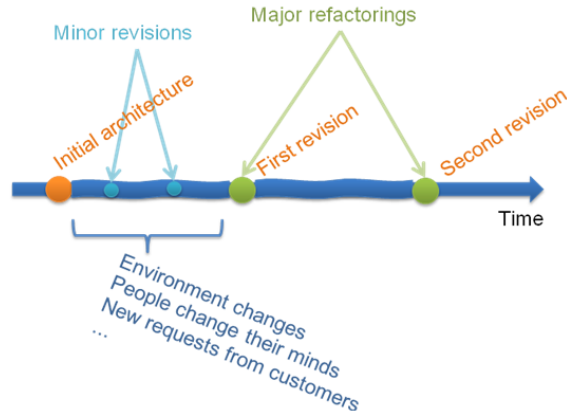


Figure 1.1: Changes in an application

No impact	Local	Non-local	Architectural
/	Confined to single element	Modifies multiple elements, but follows architectural approach	Changes fundamental ways in which elements interact
E.g., change in detailed design, configuration, ...	E.g., change specification of single element	E.g., a change propagating through the system (ripple effect)	E.g., clients in client-server also need to communicate directly

Table 1.1: Possible impact on the architecture of a change

permits all future changes. Therefore, it is important to understand how to design an architecture, such that it supports foreseen, security-related evolution without (or with minimal) impact on the architecture.

Consider Figure 1.1. An application is developed using an initial architecture, and may subsequently be distributed or deployed. In the following period, changes in the environment may lead to minor revisions of the application. If these revisions were foreseen in the initial architecture, or can at least be applied without significant effort, this is no problem. However, due to some unexpected situation, the current architecture may not be able to accommodate one or more necessary changes. At this point, a major refactoring of the architecture is necessary.

A change can impact the architecture to different degrees, as illustrated by Table 1.1. First, a change can have no impact on the architecture at all. For instance, the change can be handled by an adaptation of the detailed design, or by modifying the deployment configuration. Naturally, this is the most desirable situation.

Next, a change can have a local impact on the architecture. The change is then confined to a single element (or a limited number of related elements) of the architecture. For instance, the specification of one component in the architecture may change.

Third, a non-local change modifies multiple elements (typically across the entire architecture). However, the architectural approach itself remains unchanged (i.e., architectural integrity or style is preserved). Examples of this kind of impact are a single change that has a ripple effect throughout the entire architecture, or a change that applies to all connectors in the architecture.

Finally, a change with architectural impact redraws the fundamental ways in which the elements interact, and therefore violates the original architectural approach. An example is the need for clients in a client-server system to communicate directly. This violates the original client-server style of the architecture.

Additionally, a change has an impact in two dimensions. First, the impact of the change during the *development* (design) of the architecture. That is, given that the architect foresaw the change and decides to support that evolution scenario, the impact can be considered to be the impact of the changes that need to be applied to the architecture, such that it is prepared for the occurrence of the scenario in the future. Remark that this does not refer to the impact of implementing the evolution scenario immediately, but only to the impact of implementing the necessary infrastructure to *enable* the implementation of the scenario in the future. Also note that the development impact will typically manifest itself at a point in time where a major refactoring (as discussed before) is made to the system.

In the other dimension, there is the impact of the change during *maintenance* of the application. That is, given that a certain evolution occurs, what is the impact of the changes needed to actually *migrate to* this new situation. This impact can be low (if the architecture is prepared for the change) or high (if the change does not fit the architectural style).

1.2 Background

1.2.1 Software architecture

In [PW92], software architecture is defined as the triple elements, form, rationale. Elements can be processing elements, data elements or connecting elements. Form consists of properties of and relationships between the elements. The rationale, finally, captures the motivation of the architect for the choices that were made.

A similar definition is found in [LPR98], namely “the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them”. The authors, while not ignoring the importance of rationale, do not consider rationale to be part of the architecture itself. They state that an architecture, once created, can be analyzed independently of any knowledge of the process by which it was designed.

Yet another definition of architecture can be found in the IEEE/ISO standard for architecture descriptions [Hil00]: “The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution”. All definitions are similar to a certain extent, and describe an architecture as the collection of elements, their relationships, and some degree of rationale.

Moreover, in [LPR98], the attribute driven design (ADD) development approach for architectures is described. The approach decomposes the architecture based on the quality attributes (non-functional requirements), such that the most important quality attributes of the system are certainly fulfilled. This implies that the main drivers for an architecture are its non-functional requirements.

When an architecture evolves, its elements and/or form will change. Elements may be added, removed, or changed. The architecture’s form, that is, the properties of the elements or the relationships between them, may change as well. In principle, the rationale of an architecture does not evolve, although a change in the supporting claims of the rationale may lead to the re-consideration of the decisions that were made before. This may eventually lead to a change in the architecture. For instance, when a critical assumption that was relied upon when creating an architecture turns out to be false later, the architecture needs to change.

Creating an evolvable architecture thus means creating an architecture such that changes that are likely to happen, will be easy to implement. This implies that architectural changes should be avoided.

1.2.2 Architecture and security

Security at the architectural level can be looked at from different viewpoints.

In this report, we approach architecture from a *constructive* viewpoint. In this respect, the question on how to create an architecture that has certain security qualities needs to be answered. Often, security patterns [YWM08, HAJ07, SYHJ08] are used to this aim. The NFR framework also uses patterns to create secure designs [GY01, Wei06]. In [VL03], Van Lamsweerde proposes a patterns-based approach of creating architectures. Besides patterns, security principles are also commonly used as a guidance for creating secure architectures. For instance, the principle of least privilege can be used as a guidance for improving the security of architectures [BWJ].

Next to the constructive viewpoint, also the *notation* that is used to describe the security properties of an architecture can be studied. Existing architectural description languages (ADL's) can be extended to support security, for example xADL [RT05]. Also, UML can be used or extended to represent security properties, as is done in UMLsec [Jür05] and SecureUML [LBD⁺02].

Finally, it can be investigated how an architectural description can be used to perform a security *analysis*. The STRIDE [HLOS06] risk analysis method is performed using an architectural description as input. Besides providing security-specific notations, UMLsec can also be used to perform a formal analysis on the design. For more background on analysis techniques, we refer to the survey in [DC07].

1.3 Outline

In Section 2, the general concept of change patterns will be introduced. After a motivating example, the structure of a change pattern is presented (Section 2.2). Thereafter, Section 2.3 outlines the process that the architect can follow to apply change patterns correctly.

The main part of this report, Section 3, will investigate how change patterns can be applied to handle changing trust assumptions. A catalogue of eight change patterns, dealing with evolving trust, is presented in Section 3.3. Finally, the catalogue is illustrated using the motivating example in Section 3.4.

Chapter 2

Change patterns

It is the architect's job to design an architecture that can deal with change. Here, we propose one specific technique for the architect to obtain such architecture. A process is outlined to create architectures that are resistant against foreseen security-related changes. To achieve this, we identify the need for a catalogue of the architectural solutions that deal with specific kinds of change: change patterns. The architect can then select the appropriate solutions from this catalogue and apply them to the architecture. After a motivating example, the structure of a change pattern is described. Then, the process of using the patterns is outlined in more detail. Finally, a catalogue of change patterns for dealing with evolving trust relationships is presented, and its use is illustrated.

2.1 Motivating example

Consider an online shop scenario, where clients can order goods from a shop on the Internet, and pay using their credit card. For sake of simplicity, assume the payment data are forwarded by the shop to the credit card company, which will execute the transfer. We will model all samples using a component-based style, using UML 2 structure diagrams. The expected behaviour of the components is self-evident or explained in text; we will not separately depict it in a figure.

For illustration purposes, we will focus on a non-repudiation requirement for the system. By non-repudiation, we mean the inability of a party to deny having performed a particular action. In this scenario, the non-repudiation requirement states that the clients will acknowledge their orders afterwards (i.e., they cannot plausibly deny having placed them), and the shop will execute the orders correctly (i.e., the shop cannot plausibly deny having received an order, and cannot plausibly charge the wrong amount to the client's credit card).

Initially, assume all clients trust the shop to correctly process the orders, and not to abuse their credit card information. The shop, in its turn, is convinced that the clients will acknowledge their orders afterwards. In this situation, the non-repudiation requirement is resolved entirely by trust. The architecture for this system could look like Figure 2.1.

Clients place their orders, and the shop processes them. No additional security measures need to be taken, given the trust assumptions. Although the non-repudiation requirement is fulfilled, resolving the requirement by placing trust upon the appropriate parties may be naive in most real-life cases. Nevertheless, a risk analysis could turn this into an adequate solution.

Assume that, after a while, the trust assumptions turn out not to hold for the shop. The shop cannot present any credible evidence that the client has indeed placed a disputed order. To avoid this from happening again, the shop wants to make sure that suitable evidence exists for all future orders. Therefore, the architecture is modified to resemble Figure 2.2.

Clients now have to digitally sign their orders before the shop will process them. Therefore, they make use of a cryptography module offering digital signatures. Before processing an order, the shop first deals with the added signature. The signature is validated by the shop, using a certification authority (CA) to check the revocation status. If correct, the signed order is time stamped by a third party (the time stamping authority, TSA) to ensure its validity at the time of purchase, even if the client's signature key gets compromised afterwards. Finally, the signed order and timestamp are stored as evidence on a secure

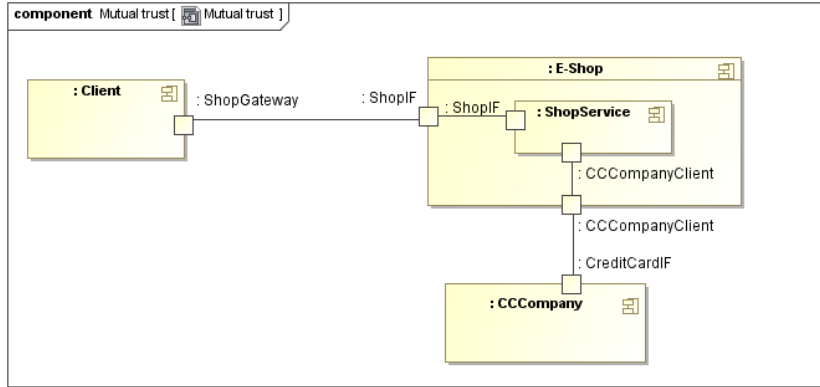


Figure 2.1: Mutual trust

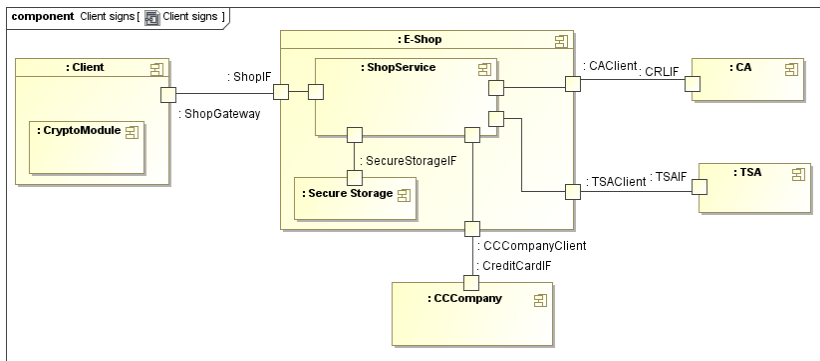


Figure 2.2: Client digitally signs orders

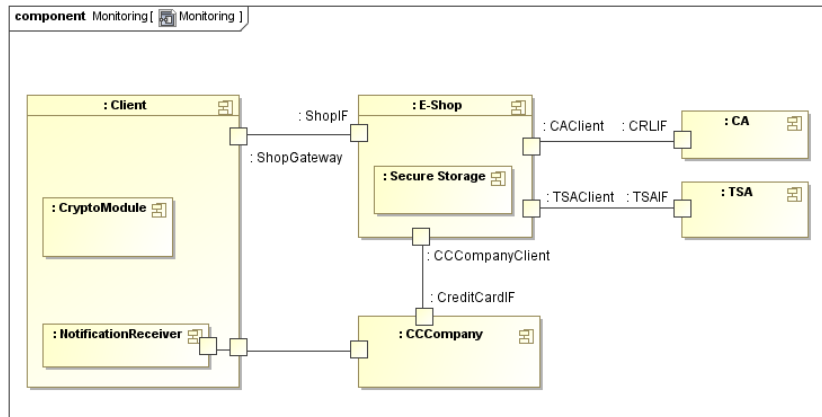


Figure 2.3: Client monitors activities

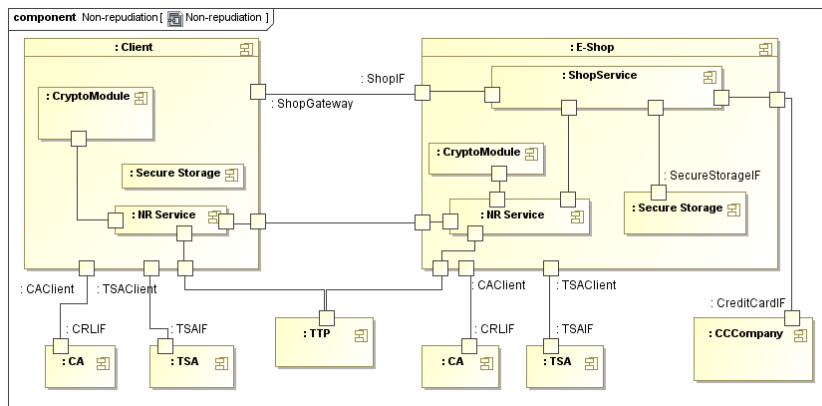


Figure 2.4: Non-repudiation

medium connected to the shop's systems. Note that the shop service component changed significantly. It will now also have to collaborate with the CA, TSA and the secure storage.

After some negative experiences with the shop, its clients become more wary of the shop's interactions with the credit card company. The shop, however, does not provide the client with any useful information regarding its actions. Only the order history is available. The credit card company, however, offers the clients a notification system triggered by any activity on their account. The clients will thus monitor the shop's activities indirectly, by comparing the order history provided by the shop with the notifications provided by the credit card company. An architecture for this purpose is displayed in Figure 2.3.

To counter the negative reactions, the shop eventually decides to provide clients with a digitally signed proof of the purchase, including details on all products and prices. The client will need to store this evidence to be able to use it in case of a dispute. Moreover, the shop wants the exchange to be fair, i.e., it should not be possible, nor for the shop, nor for the client, to cheat and receive their evidence without providing the other party with the necessary evidence. A possible architecture for this case is shown in Figure 2.4.

The shop now makes use of a fair non-repudiation protocol between the client and the shop. The protocol is implemented in a non-repudiation component (NR Service), and needs access to a trusted third party (TTP) to always complete successfully. To store the resulting evidence, both client and shop need to use the time stamping services from a TSA and have access to a secure storage medium. Since both parties have to verify the signatures on the evidence, they need access to the CA as well. The client now does not need the notifications from the credit card company anymore.

The evolution of the non-repudiation requirements in this scenario can be captured by an evolution of trust. A summary of the trust situations and chosen solutions is presented in Table 2.1. It is apparent

Trust situation	Chosen solution
Mutual trust between client and shop.	No additional actions necessary.
Distrust from shop to client in acknowledgement of purchase. Client still trusts shop.	Client provides digitally signed purchase evidence to shop.
Distrust from shop to client in acknowledgement of purchase. Distrust from client to shop in correct payment handling.	Client provides digitally signed purchase evidence to shop. Client monitors shop's payment handling actions through credit card company.
Mutual distrust between client and shop.	Client and shop use fair non-repudiation protocol.

Table 2.1: Evolution of trust

that the shop's main component, ShopService, had to be modified multiple times to accommodate this changing trust. We can conclude that it was not designed for this kind of evolution.

2.2 Change pattern structure

To enable the architect to design the architecture such that it can cope with possible evolution scenarios, we propose the usage of change patterns. A change pattern consists of the following parts:

1. A change scenario, expressed at the requirements level, which describes the change in the requirements or environmental assumptions, and for which the change pattern provides a solution. This change scenario will consist of a before and after requirements model.
2. One or more solutions. Each solution consists of
 - An (optional) set of architectural support patterns that describe the infrastructure that needs to be integrated within the architecture in order to use the change pattern.
 - Change guidance, that describes how the change scenario can be implemented, based on the infrastructure introduced in the architectural support pattern.
3. A mapping between the elements from the change scenario (at the requirements level) and the architectural elements in the solution. A mapping can be applicable to a set of change patterns, to a single change pattern or even to only one solution of a change pattern.

Each change pattern thus explicitly describes the evolution scenario it supports. This description is abstract, i.e., situated at the requirements level (including the environmental assumptions), and is expressed as a (situation before change, situation after change) pair. The evolution scenario is described independently of any application context, so interpreting the evolution scenario for a specific application requires performing a translation from the general scenario elements to the specific elements in the application. Besides the change scenario, the pattern provides the description of an architectural solution. Following the change guidance from the solution should enable the evolution scenario to be incorporated without significant impact on the architecture, given that the necessary architectural support patterns are already in place in the architecture. Finally, the mapping clarifies how the entities in the scenario description map to the entities of the solutions at the architectural level.

2.3 Process description

The process in this section describes how the change patterns can be used when designing an evolvable system. The process is to be executed by the architect of the system. Note that it reflects only one possible strategy that an architect can follow within his domain and responsibility.

The first input necessary for the process is a model representing the set of security requirements of the system that is being designed. These requirements do not yet need to be complete. Any part of the system that is sufficiently explored can be used as an input to the process. Of course, the results will

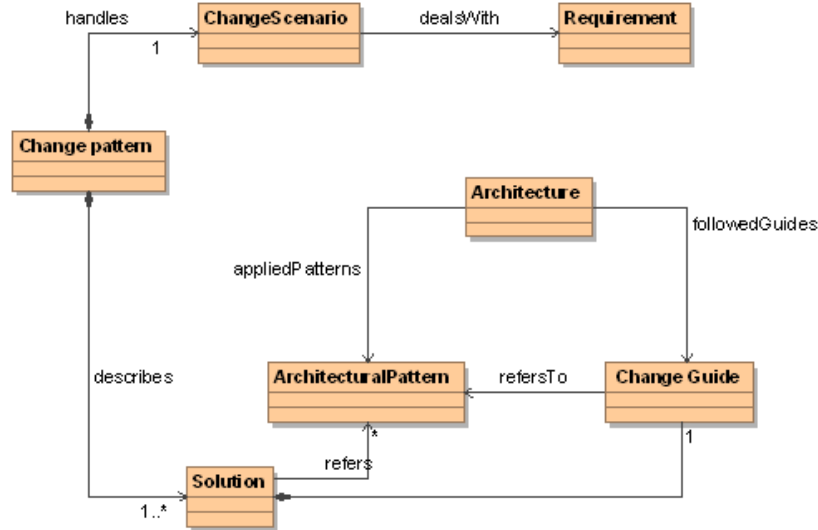


Figure 2.5: A change pattern meta model

then only be limited to this part of the system. Additionally, the process requires a model representing an initial architecture that already supports the security requirements. This architecture is not expected to support evolution of these requirements; supporting this is the outcome of the process. Finally, a catalogue with suitable change patterns is needed.

Given these inputs, the architect can start. First, matches are sought between the security requirements descriptions and the evolution scenarios from the change pattern catalogue. An evolution scenario matches with the requirements if the requirements describe the ‘before’ part of the scenario, and a meaningful transition to the ‘after’ situation can be identified. This transition may occasionally be straightforward to identify, but often it will require some creative thinking and brainstorming by the architect and stakeholders.

For each matching instance, the importance and likelihood of this change scenario is estimated, again by the architect and other stakeholders. This estimation is similar to, and in fact closely related to, the risk analysis of the system: the stakeholders will have to decide whether the evolution scenario currently needs to be supported by the architecture in order to mitigate likely future costs, or whether support for the scenario can be deferred.

The decision to support or discard an evolution scenario, and the reasons for that, should be explicitly documented in the architecture’s rationale. If it is decided that the scenario has to be supported, the architecture is updated by instantiating the change pattern in it: a solution is chosen, and the architect needs to ensure that the architectural support patterns are in place. Later in the lifetime of the application, if the evolution scenario actually manifests itself, the change guidance from the solution is followed to update the application to this new situation. The goal of the change pattern is to help the architect in implementing the change described by the scenario without significant impact on the architecture of the system.

Using the change pattern approach in the SecureChange framework requires a change pattern meta-model, depicted in Figure 27. The metamodel is largely encapsulated in the architectural domain, with some external references. First, it references the requirements domain, by means of the change scenarios. Additionally, the change guide may contain references to elements from other domains like implementation or deployment.

Some of the elements from this metamodel can have state. A change scenario can be in two states: the ‘before’ state, when the real-world situation corresponds with the situation described in the ‘before’ part of the scenario, and the ‘after’ state, in which the real-world situation corresponds with the situation described in the ‘after’ part. An architectural support pattern can be in two states with respect to a particular architecture: the ‘unapplied’ state, i.e., it has not been applied to the architecture, and the opposite ‘applied’ state. Equally, a change guide can be in two states with respect to a particular

Original state			Change		Next state			
Scenario state	Change guide state	Architecture state	World Changes?	Action of architect	Scenario state	Change guide state	Architecture state	
Before	Unfollowed	Matches	No	Nothing	Before	Unfollowed	Matches	
				Follow*		Followed	Overprotected	
			Yes	Nothing	After	Unfollowed	Risk	
		Follow*	Followed	Matches				
	Followed	Overprotected		No	Nothing	Before	Followed	Overprotected
					Undo		Unfollowed	Matches
		Yes	Nothing	After	Followed	Matches		
		Undo	Undo		Unfollowed	Risk		
After	Unfollowed	Risk	No	Nothing	Before	Unfollowed	Matches	
				Follow*		Followed	Overprotected	
			Yes	Nothing	After	Unfollowed	Risk	
		Follow*	Followed	Matches				
	Followed	Matches		No	Nothing	Before	Followed	Overprotected
					Undo		Unfollowed	Matches
		Yes	Nothing	After	Followed	Matches		
		Undo	Undo		Unfollowed	Risk		

Table 2.2: Change patterns and state transitions

architecture: the ‘unfollowed’ state, meaning that the change guide has not been followed by the architect, and the ‘followed’ state for when the change guide has been followed. The architecture itself (which is partially determined by the applied patterns) can be in three states with respect to a particular scenario. First, the architectures ‘matches’ a scenario when the architecture in its current form provides the right guarantees to securely fit in the current real-world situation. Conversely, the architecture is at ‘risk’ when it does not match the state of the scenario, because it does not provide enough guarantees; this leads to a security risk. Finally, the architecture can be ‘overprotected’, meaning that it does not match the state of the scenario, but it provides more guarantees than necessary for the state of the scenario. Changes in the scenario state (real world) are caused by events external to the architectural domain (e.g., stakeholders changing their minds, deployment in a new environment, etc.). Changes in the state of the support patterns, change guides and architecture happen due to actions from the architect. In our context, the actions of an architect are limited to applying or unapplying an architectural support pattern, following or undoing a change guide, or doing nothing. Note that, in order to follow a change guide, the referred architectural support patterns must always be in the ‘applied’ state. All possible transitions for a single combination of scenario, change guide and architecture are summarized in Table 4. Note that, for presentation purposes, we only consider the case where following a change guide increases the security guarantees. Undoing the guide will thus decrease the security guarantees. It is straightforward to extend the table in order to include the converse situation. The term ‘change guide’ in this table refers to a change guide described in one of the solutions that belong to the scenario; ‘architecture’ refers to the architecture with or without the change guide applied to it. Actions marked with a ‘*’ denote actions that require the support pattern to be in the ‘applied’ state.

2.3.1 Automation

In its current form, the architect has to perform all steps outlined above manually. It is interesting to investigate how, and to what extent, the architect can be supported in these tasks by model-driven development and automation. This, of course, requires that both requirements and architecture are expressed using a model.

To further support automation, the catalogue with change patterns needs to be formalized as well. A formalized change pattern consists of the following:

- A formal description of the change scenario at the requirements level that is supported by the pattern. This description is dependent on the formalism (meta-model) used to express the security requirements.
- A formal description of the architectural support pattern that needs to be instantiated in the architecture to support the evolution. This description is dependent on the formalism (meta-

model) used to express the architecture. Note that the pattern description may introduce new structural elements, describe behaviour, and/or can identify roles that later have to be mapped to actual elements from the architecture into which the pattern is instantiated.

- A formal description of the change guidance. The change guidance should relate the necessary changes at the architectural level to the changes at the requirements level. In the description, constructs (e.g., structural elements, behavioural elements, roles) introduced by the architectural support pattern may also be referred to. Therefore, this description is dependent on the formalism used to express the change scenario, the formalism used to express the architecture, and the architectural support pattern.

Furthermore, the change pattern can describe additional characteristics, like qualities, advantages, disadvantages, consequences, etc. that it exhibits. These descriptions are helpful for the architect when choosing a particular pattern and making trade-offs, but play no major role in the automation. Based on this change pattern catalogue, and the current requirements and architecture models, the architect can be supported in multiple ways. For instance, matches between the evolution scenarios and the requirements model can be automatically sought. For each matching scenario, the architect can be prompted whether or not to deal with this scenario. If he chooses not to deal with the scenario, this decision (and its motivation) can be explicitly recorded in the architecture's rationale. If, however, the scenario is chosen, then additional help for the architect can be given, by presenting the set of applicable solutions. The architect can then choose a solution to instantiate. Mappings for the roles of the solution can be determined automatically if possible, or be provided by the architect. Once the mappings are known, an automated transformation can be executed that instantiates the architectural support patterns into the architecture. Also, support can be given in following the change guidance when that becomes necessary.

In what follows, we will attempt to document change patterns in a formal way. We will not, however, elaborate on the automation part any further.

Chapter 3

Trust evolution

The approach outlined above is generic. It can be used for any recurring kind of change, for which a generic solution can be described. To limit the scope of the discussion, this part of the report will illustrate the process using one specific kind of change: evolving trust relationships between the entities in the architecture. The choice for trust is motivated by the following three reasons.

1. Trust is a general but important notion when dealing with security, because the need for security in a system originates from the presence of untrusted entities. Therefore, to be able to effectively secure a system, it is important to know (and explicitly state) which entities are trusted for certain tasks, and which are not. This establishes a strong connection between trust and security.
2. While research has been done on the influence on a software architecture of ‘classical’ security concepts such as confidentiality, integrity and availability, the impact on an architecture of the presence (or absence) of trust relationships between two entities is an underexplored area. This makes trust the most interesting choice from a research point of view.
3. There is a large likelihood of experiencing change in the area of trust over the lifetime of a system. For instance, systems can be moved from a trusted environment to a more hostile world. Additionally, trust relationships between humans (and, by extension, between the companies they work for) are volatile in nature. Such a change will most likely be reflected in the architecture of the software systems. Thus, it is expected that trust evolution will occur, and that it may have a significant impact on the architecture.

The change scenarios provided in the next part originally emerged from analyzing the case studies of the SecureChange project. After this initial analysis, the findings were grouped and abstracted. Finally, the set of scenarios was completed by expressing the scenarios in SI* and eliciting missing variants.

3.1 Evolving trust scenarios

In the remainder of this report, the effect of the evolution of trust on a component-oriented architecture is studied. A catalogue of change patterns for this kind of evolution is presented, and applied to some examples.

To reason about the evolution in trust at the requirements level, we need to explicitly represent the trust relationships. We use the SI* modelling language [GMZ05], adopted by the Secure Tropos methodology [MMZ07]. SI* offers the best support for representing trust, by extending the Tropos language with explicit trust and distrust relationships. We will assume the reader is familiar with the notation; otherwise, we refer to [MMZ07] for an overview of the concepts.

To elicit architectural solutions that can cope with changing trust at the requirements level, we distinguish among various scenarios in which trust changes. We assume that we start from a late requirements model, i.e., there already exists an actor that represents the system. In SI*, the dependencies between that system actor and the other actors define the functional and non-functional requirements of the system [35]. For each distinct trust evolution scenario, a change pattern is defined. This pattern will detail how the architecture of the system should be designed, such that it can deal with the trust

evolution scenario. Also, the pattern describes the necessary changes that have to be applied whenever the evolution scenario occurs. We will focus only on scenarios in which trust decreases. In these scenarios, additional measures will have to be introduced to compensate for the lack of trust. Of course, the inverse evolution is also possible. The proposed change patterns should, therefore, also be capable of handling an increase in trust. This means that every mechanism, proposed by the pattern to remedy the decreasing trust, should subsequently be easy to undo. Finally, recall that both the concept of change patterns and the accompanying process are independent of the used requirements elicitation technique or model. Therefore, the elicitation technique that was simultaneously developed in Work Package 3 could be plugged in as well.

3.2 Mapping trust requirements to architecture

As discussed before, the catalogue needs to provide a mapping between SI* requirement models and component-oriented architectural models, which we will express in UML (version 2.0). This mapping is the same for all change patterns in the catalogue, and is depicted in Figure 3.1.

Note that, contrary to the delegation of execution relationships, delegation of permission relationships are not reflected explicitly in the architectural model. The relationships are reflected only by notes, which can be interpreted as architectural assumptions in this case. At the architectural level, it is assumed that components that fulfil services already have permission to execute the service, or that they acquire the necessary permissions implicitly by means of the received invocations. Mechanisms such as access control may restrict the permissions of a component in the system. In that sense, an architecture is more likely to reflect the absence of a delegation of permission —by the presence of access control mechanisms— rather than the delegation itself.

Remarkably, trust relationships also do not have a companion on the architectural level. It will, again, chiefly be a lack of trust that will influence the architecture: when trust is missing, mechanisms must be put into place to overcome this situation. Trust and distrust relationships are thus mapped to architectural assumptions (for trust relationships) or constraints (for distrust relationships).

3.3 Change pattern catalogue for trust

The catalogue that follows presents change patterns in which a trust relationship (trust of execution or permission) changes, insofar that the relationship crosses the system actor's boundary. This means that the changing relationship represents a change in requirements or assumptions.

We assume that, for each trust (or distrust) relationship between two actors, there is also a corresponding delegation relationship. Strictly speaking, this is not necessary, but trust relationships without a delegation between the actors have little value for our purposes: the presence or absence of the trust relationship without a delegation relationship has no influence the behaviour of the actors, and as such requires no special attention at the architectural level.

In the catalogue, each change pattern entry shows the situation before and after the changing trust relationship using the SI* notation. Also, an example gives a concrete illustration of the case. Subsequently, one or more solutions are described. The solutions are also applied to the example given before. It is important to keep in mind that only architectural solutions are considered. For instance, restoring trust by signing an agreement on paper may also be possible, but does not have an architectural impact. Also, multiple solutions to the same problem may sometimes be combined.

It is not claimed that the patterns are the only or best solutions, or that the set of patterns or their solutions is complete. They only offer choice, and provide guidance, to the architect. When choosing and implementing them, other architectural constraints have to be taken into account as well.

Also keep in mind that we will only focus on the architecture of the system that is being developed, and not the architecture of external, connected systems. This means that, for some solutions, the external systems may need to be adapted to work with the chosen solution. These adaptations are not described in detail, but it is clear that change patterns could be used in the design of these external systems as well.

Finally, we stress that requirements and architecture are not two separate phases. This implies that the architectural solutions that are proposed in the next section can usually be expressed by a more abstract requirements model as well. We will do so where applicable.


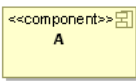
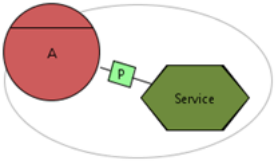
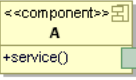
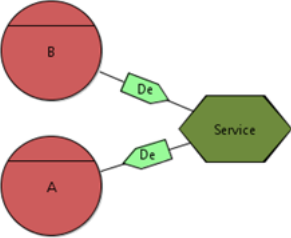
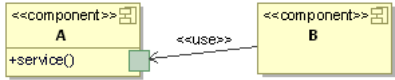
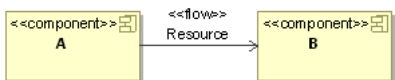
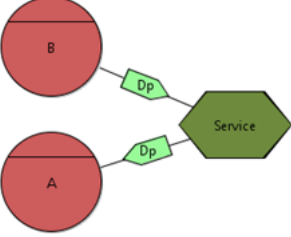
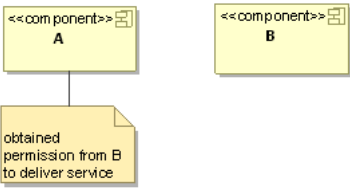
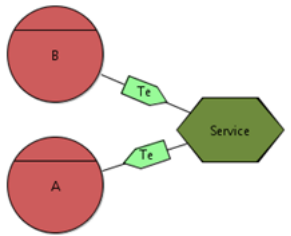
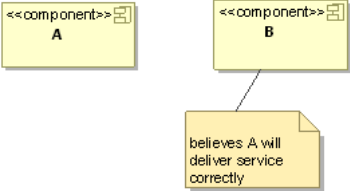
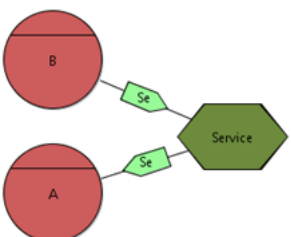
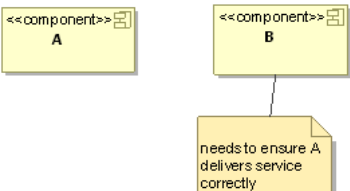
SI*	UML 2.0
<p>Agent A</p> 	<p>Component A</p> 
<p>Agent A, providing service</p> 	<p>Component A with operation and port</p> 
<p>Delegation of execution from B to A</p> 	<p>If the service is a goal or task: operation of A, used by B</p>  <p>If the service is a resource: information flow from A to B.</p> 
<p>Delegation of permission from B to A</p> 	<p>Not modelled explicitly</p> 
<p>Trust of execution from B to A</p> 	<p>Not modelled explicitly</p> 
<p>Distrust of execution from B to A</p> 	<p>Not modelled explicitly</p> 

Figure 3.1: Mapping between SI* and UML

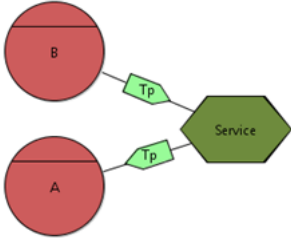
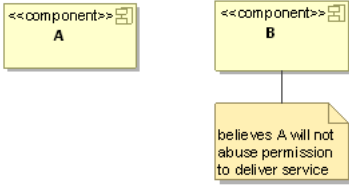
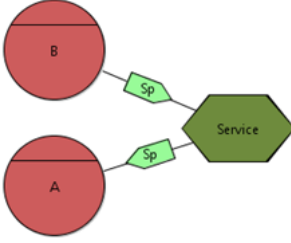
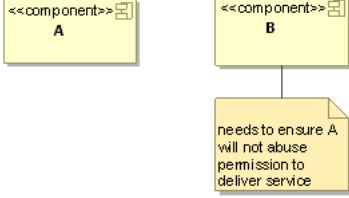
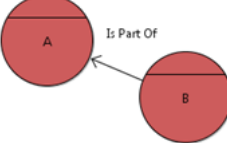
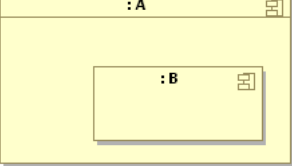
SI*	UML 2.0
<p data-bbox="400 645 780 678">Trust of permission from B to A</p> 	<p data-bbox="879 645 1158 678">Not modelled explicitly</p> 
<p data-bbox="384 947 796 981">Distrust of permission from B to A</p> 	<p data-bbox="879 947 1158 981">Not modelled explicitly</p> 
<p data-bbox="443 1249 735 1283">Actor part of other actor</p> 	<p data-bbox="895 1249 1142 1283">Composite structure</p> 

Figure 3.1: Mapping between SI* and UML (continued)

Change scenario and solutions	Section	Page
Evolving trust of execution upon external actor Solution 1: Require commitment Solution 2: Use monitoring	3.3.1	19
Evolving trust of execution from external actor Solution 1: Provide commitment Solution 2: Enable monitoring	3.3.2	27
Evolving trust of permission upon external actor Solution 1: Apply least-privilege principle Solution 2: Attribute-based access control Solution 3: Use monitoring	3.3.3	33
Evolving trust of permission from external actor Solution 1: Request confirmation Solution 2: Enable monitoring	3.3.4	37
Delegate execution of a service to a trusted actor Solution: Encapsulate service	3.3.5	39
Delegate permission to a service to a trusted actor Solution: Flexible access control	3.3.6	40
Providing additional service with delegated execution Solution: Introduce bridge component	3.3.7	44
Providing additional service with delegated permission	3.3.8	44

Table 3.1: Change pattern catalogue overview

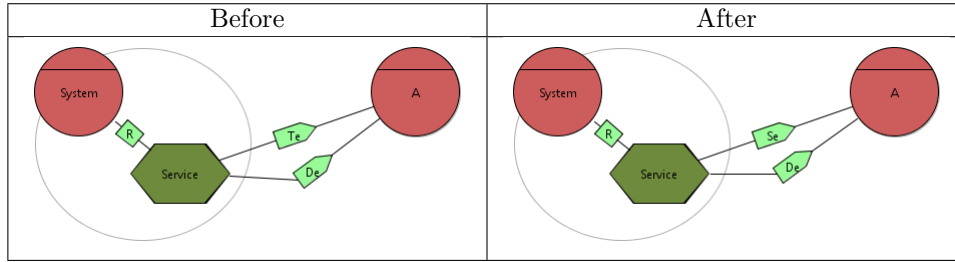


Figure 3.2: Evolving trust of execution upon external actor

For easy browsing, an overview of the change patterns in the catalogue, with references to the corresponding section and page number, is given in Table 3.1.

3.3.1 Evolving trust of execution upon external actor

For the first change pattern, suppose that the system actor relies on an external actor A to execute a certain task (i.e., it delegates execution of the service to A). Originally, the system trusts A to (at least) execute the delegated task. Over the course of time, this trust relationship may change, and the trust relationship can disappear. This is shown in Figure 3.2.

This causes a problem: the system expects A to achieve the delegated goal, but at the same time, does not trust A to do so. To resolve this problem, additional mechanisms will have to reinstate the trust of the system in A.

Example The system to be built is a travel agency system. The system needs to make reservations on flights from an airline. It relies on an external actor, the airline reservation system, to make the reservation when requested. Initially, the travel agency assumes all reservations will be made correctly. After some clients complained because their reservation was incorrect, the travel agency no longer trusts (but still needs) the airline system for making the reservations.

Applying the mapping from requirements to architecture, as given in Figure 3.1, to this example, we obtain the architectures in Figure 3.3 (before and after the trust relationship changes).

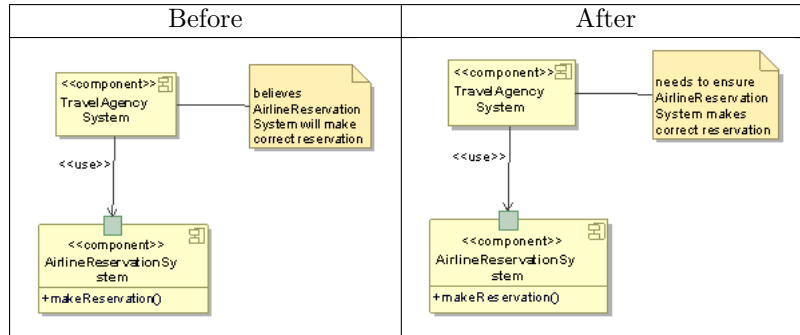


Figure 3.3: Requirements mapped to architecture

Achieving this result by applying the mapping is straightforward, and we will therefore omit the architectures derived from the requirements using the mapping in the other change pattern examples.

3.3.1.1 Solution 1: Require commitment

A first solution for the case above is presented as the ‘non-repudiation pattern’ in [CEKK⁺09]. To re-gain the trust in A, the system will require assurance from A that it will do what is expected. Therefore, before A can fulfil the service, the system requires A to provide a commitment (a piece of evidence) that it will do so. The service is thus split in two: first, checking that a correct commitment from A has been received, and second, fulfilling the actual service.

Note that the system trusts A to deliver the commitment; if no commitment is provided, the system should choose not to rely on A for fulfilling the service. Alternatively, a (fair) non-repudiation protocol could be used between the parties. The protocol then ensures that no party can obtain a benefit over the other.

This solution strategy, expressed in SI* and mapped to the architecture, is shown in Figure 3.4.

Example The travel agency can require a confirmation from the airline system, before the reservation is made. This confirmation should include all data that will be used to make the reservation. The travel agency should check the information in this confirmation, and the reservation should only be made if the travel agency correctly verified the information. In case of later dispute, the confirmation can be used as evidence by the travel agency. This assurance suffices for the travel agency to restore its trust in the airline.

Architectural support pattern At the architectural level, it is clear that this solution requires the system to carry out additional actions and checks before and/or after one of its services is invoked. At a later point in time, these actions and checks might be removed or replaced. The architecture of the application therefore should allow flexible addition and removal of these actions and checks, preferably by reconfiguration.

The architectural support pattern for supporting the above scenario is shown in Figure 3.5, and described as follows. The system component can be associated with a number of registered handler components. The handlers will perform the additional checks and actions that need to be performed. For instance, they could read, modify or delete the request parameters, or append additional information. Moreover, they can block the request altogether, e.g., when a necessary condition is not fulfilled.

All information that the handlers need to do their work is encapsulated in a context class. An instance of the context class exists for both the delegation request and the response. The class should, at least, encapsulate all parameters of the request. The exact definition of the class depends on the operation, and is not elaborated here. The registration of the handlers with the system could, for example, occur in the implementation, or could be handled via configuration options. The exact details are not important, and are therefore not elaborated upon.

Whenever the execution request for the operation is about to be issued, instances of the context class need to be created and populated with the relevant information. Then, the registered handlers are called sequentially. If none of the handlers prohibited the execution of the operation, because of a failed check

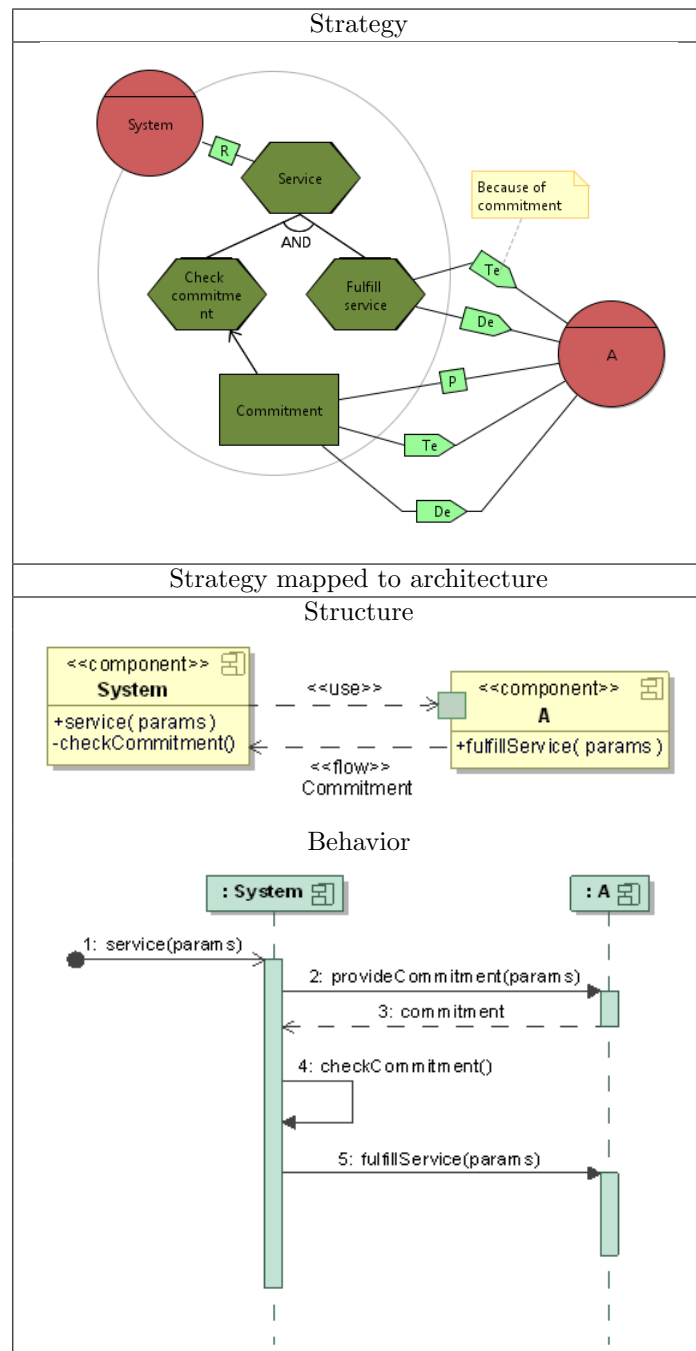


Figure 3.4: Require commitment

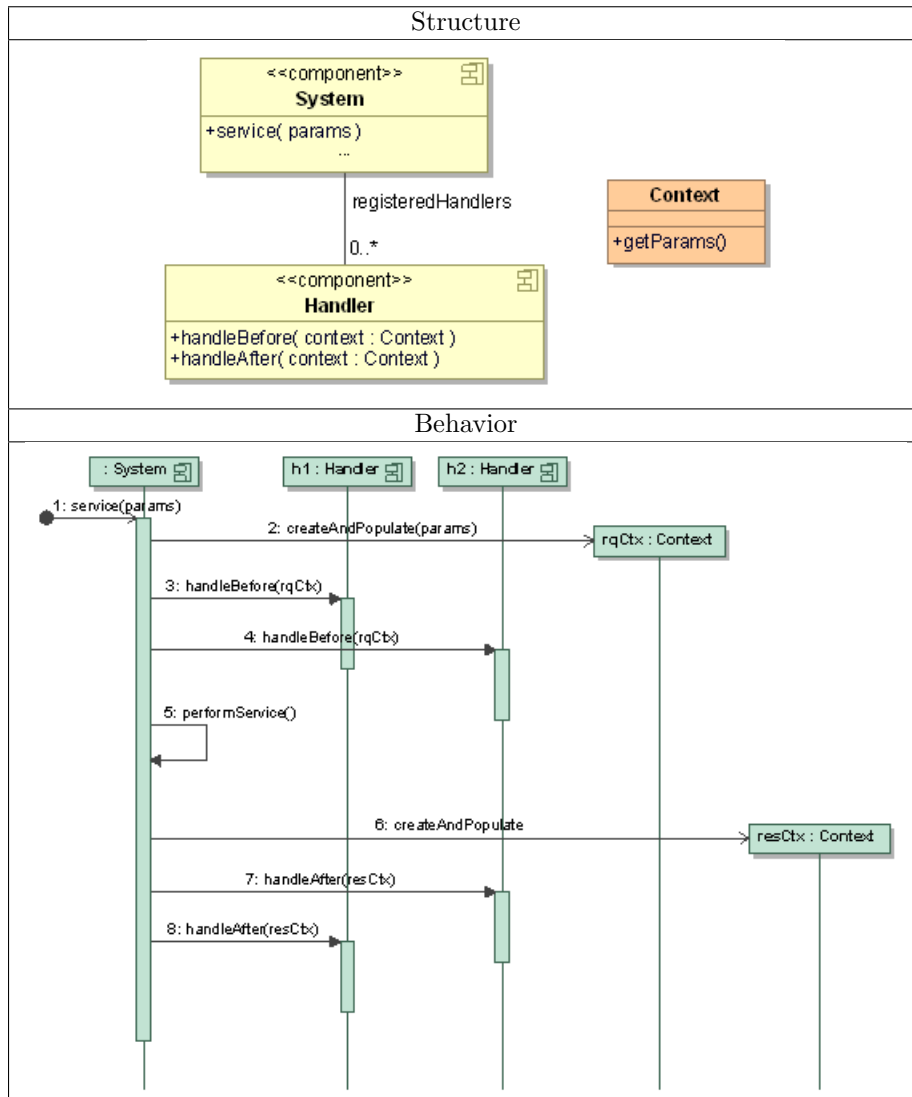


Figure 3.5: Handler architectural support pattern

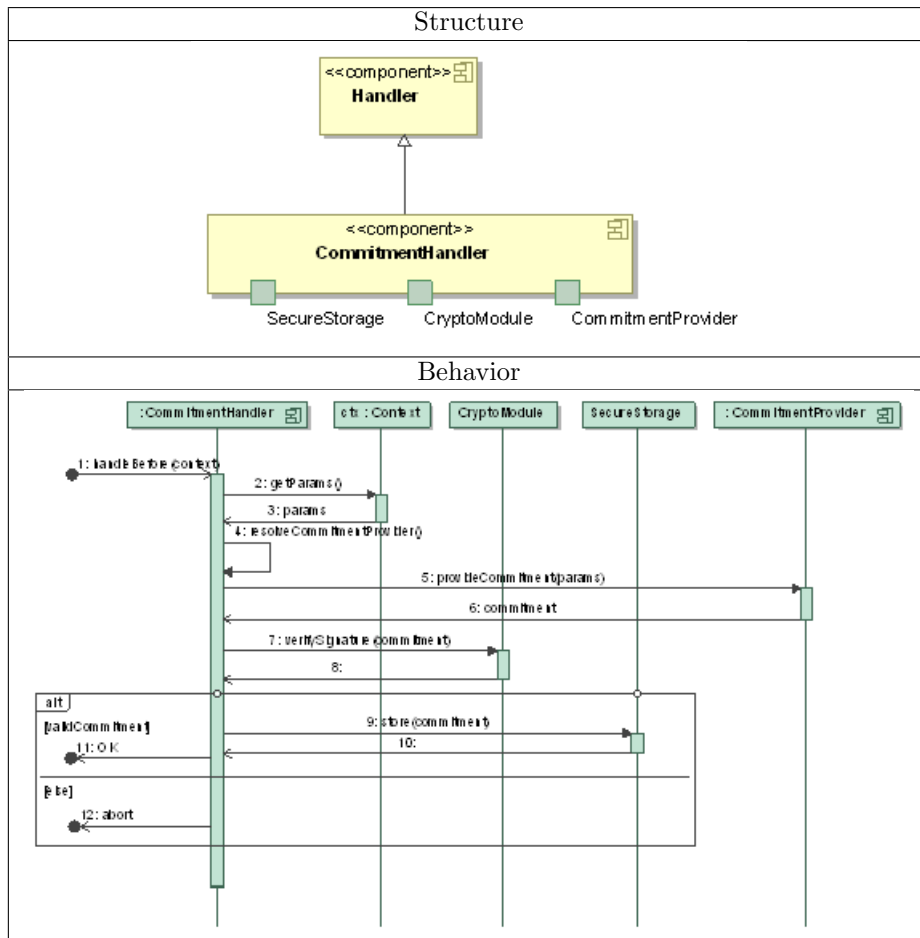


Figure 3.6: Applying the handler architectural support pattern

for instance, then the actual operation runs. After the operation has been executed, but before the result is returned to the system, a context instance is created for the result and the handlers are called again.

Without support from the underlying platform, the implementation has to be done manually for each operation. In this case, it is clear that aspect-oriented technologies can provide a significant benefit. They enable the modular interception technique that is needed for this pattern.

When a middleware platform is used, this functionality is often available by default. For example, in the Java Enterprise Edition, web service clients written using JAX-WS can specify client handlers in a handler chain. These handlers will be called before any operation of the web service is called, and/or before any result is returned to the caller. The handlers have the possibility to, among others, inspect the entire SOAP body and add header fields.

Change guidance To implement the solution when the trust relationship change occurs, a commitment handler needs to be developed (see Figure 3.6). This handler must request a commitment from the other party (called the commitment provider). This commitment must include the values of the (relevant subset of the) parameters that will be used in the actual fulfilment of the service. The commitment handler must resolve a reference to the commitment provider, request a commitment and verify the validity of the returned commitment (both its contents and the digital signatures). If the commitment is valid, it needs to be stored, and the fulfilment of the request can continue. If the commitment is invalid, or has been forged, the request must be aborted.

The component that corresponds to the system must have the architectural support pattern applied. The commitment handler can then be added to the set of registered handlers (in the implementation, or by means of configuration), as shown in Figure 3.7.

The commitment handler should also be configured such that it applies only to the operation that

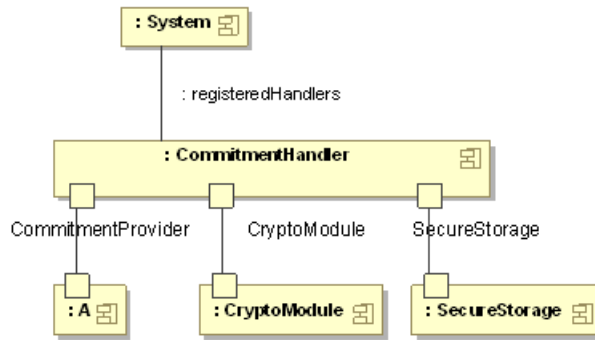


Figure 3.7: Adding a commitment handler

needs protection, i.e., the operation that corresponds with the delegated service as described in the change scenario. The addition of this handler ensures that the negative effects of the change in trust relationships are mitigated.

3.3.1.2 Solution 2: Use monitoring

A second solution in this case is the use of monitoring (Figure 3.8). The system delegates the task of monitoring the execution of the service to a monitor agent. The monitoring gives the system enough assurance to trust upon the execution of the service by A. Note that the monitoring does not prevent A from executing the service incorrectly. However, because of the high chance of failure being detected, A now has more incentive to execute the service correctly.

Monitoring can be performed using communicating software components, but could also be handled by intervening humans (e.g., using e-mails, telephone, letters, etc.). Therefore, the exact monitoring mechanism is not part of the solution. In general, however, monitoring an agent requires that some information from that agent is provided to the agent that performs the monitoring, either spontaneously or upon request.

Example The travel agency system can send an e-mail to inform one of the employees that a reservation with an airline has been made. The employee should then confirm with the airline that the reservation has been made correctly, and if not, contact the airline and make sure the problem gets resolved. From the viewpoint of the system, the reservation will certainly be handled once it has been sent to the airline system and the e-mail to the employee has been sent. Therefore, its trust in the airline system is restored.

Architectural support pattern This solution also requires interception of the service execution, and will require the architectural support pattern introducing configurable handlers as described in Section 3.3.1.1.

Change guidance To implement this solution when the trust relationship changes, a monitor handler needs to be developed (see Figure 3.9). This handler gets invoked before the service from the external component is requested.

The exact implementation of the monitor depends on the type of monitoring chosen, and has to be defined on a case-by-case basis. For instance, the monitor could periodically retrieve information from the external component, or it could send an e-mail to a person responsible for monitoring the service fulfilment.

The monitor handler can then be added to the set of registered handlers (in the implementation, or by means of configuration), as shown in Figure 3.10.

The monitor handler should also be configured such that it applies only to the operation that needs protection, i.e., the operation that corresponds with the delegated service as described in the change scenario.

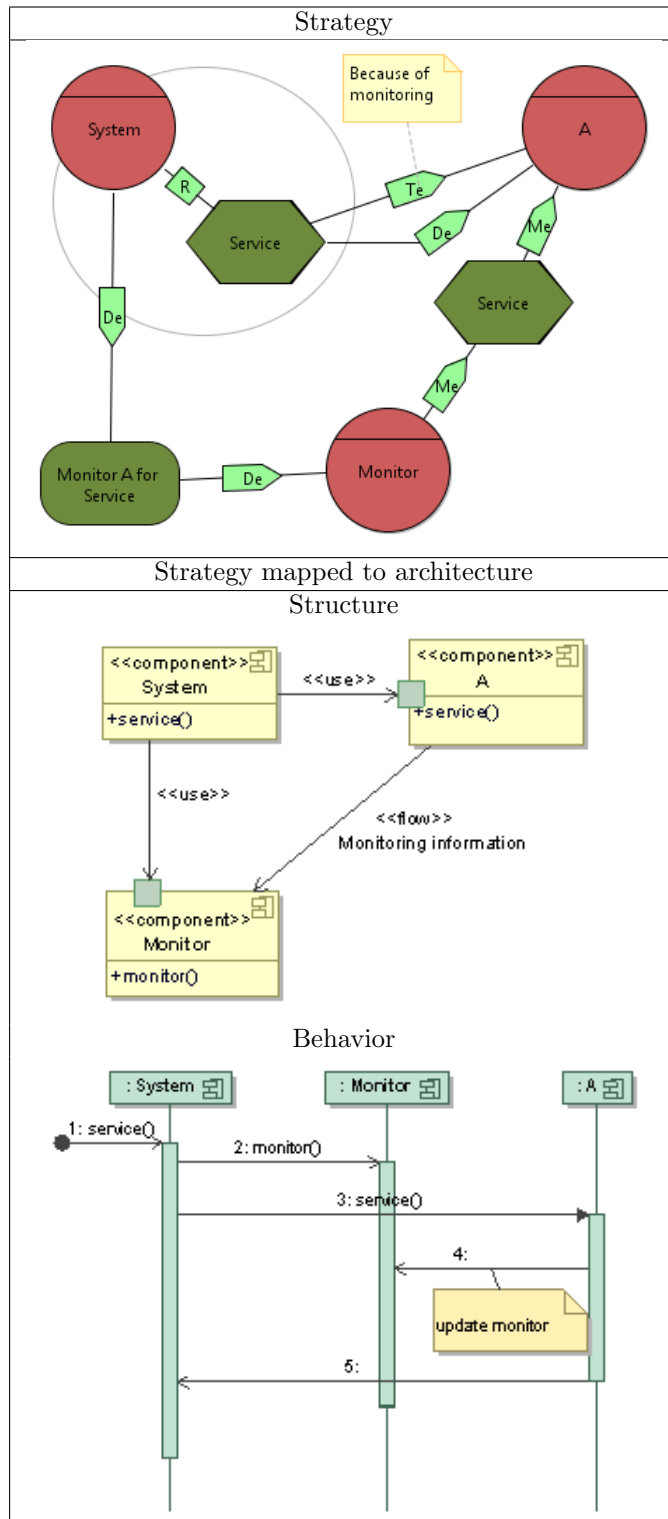


Figure 3.8: Use monitoring

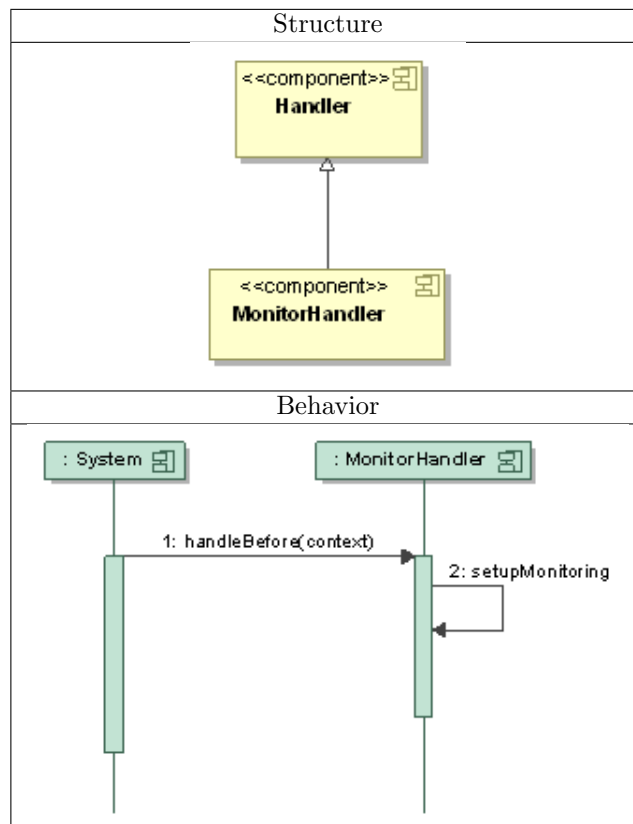


Figure 3.9: Developing a monitor handler

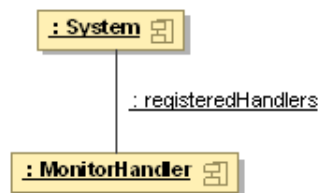


Figure 3.10: Register the monitor handler

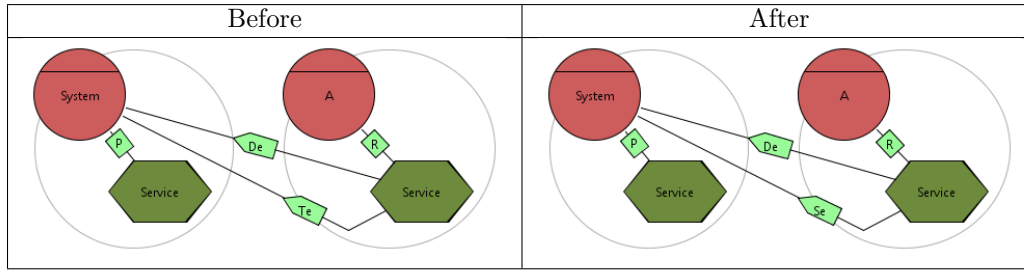


Figure 3.11: Evolving trust of execution from external actor

The addition of this handler ensures that the negative effects of the change in trust relationships are mitigated.

3.3.2 Evolving trust of execution from external actor

In this change pattern, the system provides certain services that are relied upon by an external party. Initially, the external party trusts the system to fulfil the service correctly, but this trust may disappear later during the lifetime of the system. This scenario is depicted in Figure 3.11.

This case is the complement of the previous case. Here, the system actor is the actor that is no longer trusted. As expected, the solutions to this case are related to the previous case. Instead of requiring a commitment, the system will now have to provide one. Similarly, instead of monitoring the external agent, the system will now have to enable monitoring by another agent.

Example The system we are building is an airline reservation system. The system is relied upon by an external actor, the travel agency. Initially, the airline believes all travel agencies trust their reservation system. After some disputes about reservations, though, the travel agency no longer trusts (but still needs) the airline reservation system for making the reservations. The airline reservation system now needs to be changed, such that the trust from the travel agency is restored.

3.3.2.1 Solution 1: Provide commitment

The system can offer commitments for the services requested by other parties, as illustrated in Figure 3.12. The commitment is provided before the actual service is fulfilled. Note that, similar to the converse situation, the external actor trusts the system to deliver the commitment. If this is unacceptable, a (fair) non-repudiation protocol can be used between the parties. The protocol then ensures that no party can obtain a benefit over the other.

Example The airline reservation system can provide a confirmation to the travel agency. This confirmation contains all information that will be used to make the reservation. The travel agency then verifies the confirmation. Only if the verification is successful, the reservation is made. Because the commitment assures the travel agency of the correctness of the reservation, its trust is restored.

Architectural support pattern The system should be designed such that it is easy to validate the parameters used for providing the commitment, i.e., it should be ensured that no commitment is generated for a request that cannot be fulfilled. A possible solution for this is the introduction of a validator component as part of the system, which is responsible for validating a set of parameters for a service. This support pattern is shown in Figure 3.13.

Moreover, to digitally sign the commitment, the system should have access to a component offering cryptographic operations (Figure 3.14).

Change guidance To implement this solution when the trust relationship changes, the system needs to be extended with a service to provide a commitment. This service will contact a validator to ensure the validity of the parameters, before constructing the commitment. If the parameters are valid, the

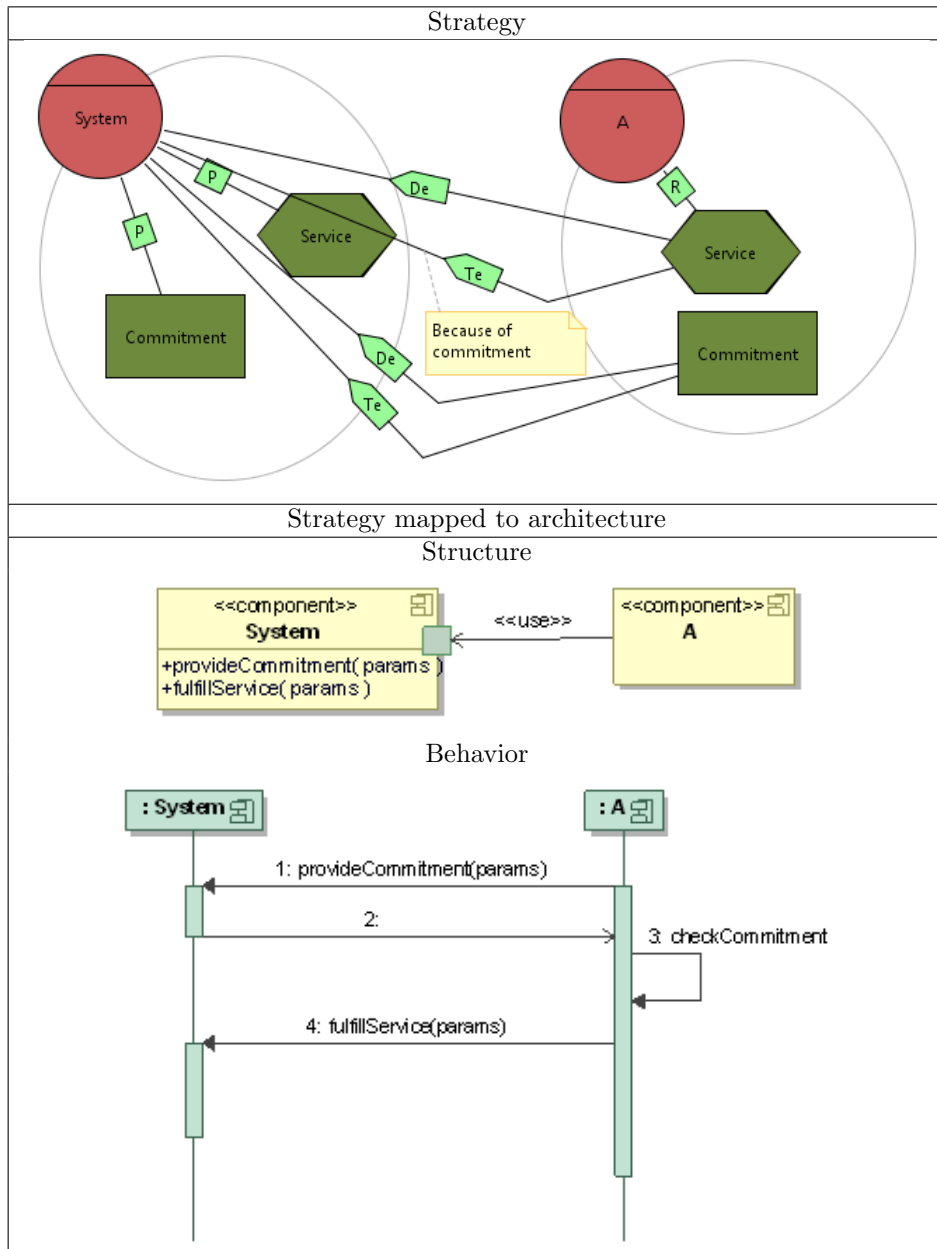


Figure 3.12: Providing a commitment

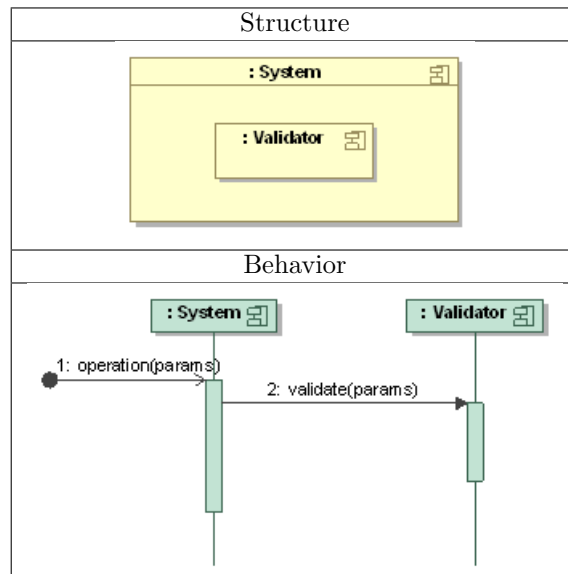


Figure 3.13: Validator architectural support pattern

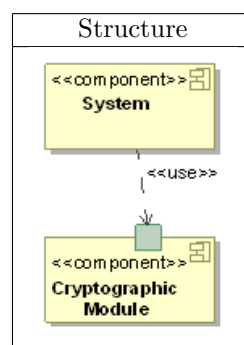


Figure 3.14: System accesses a cryptographic module

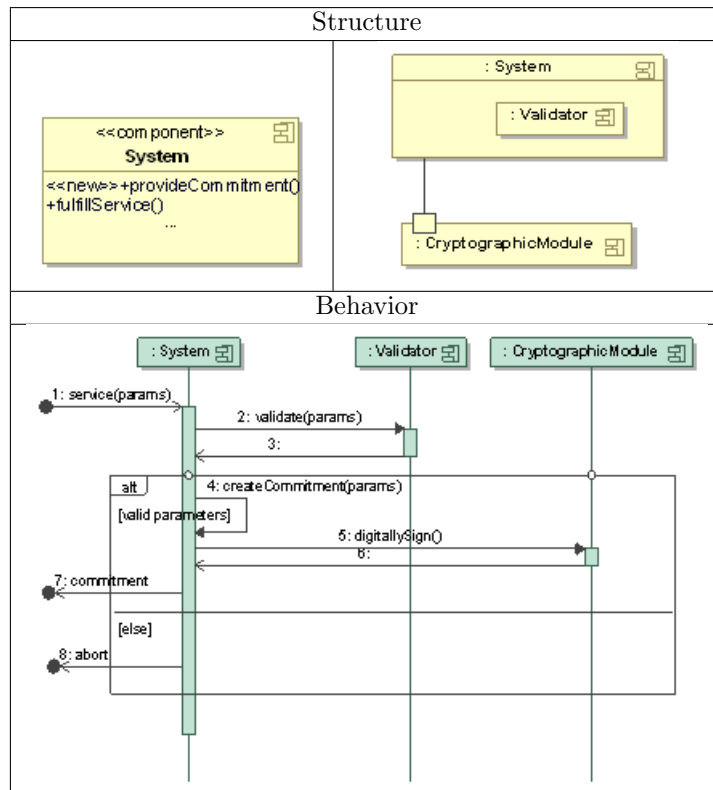


Figure 3.15: Applying the change guidance

commitment is digitally signed and returned. From this point on, the system has committed to fulfil the service with the given parameters. This is shown in Figure 3.15.

3.3.2.2 Solution 2: Enable monitoring

A second solution to restore the trust from the external party is to enable monitoring of the service execution (Figure 3.16). The external party delegates the task of monitoring the system to a monitor actor, which is then responsible for checking the fulfilment of the service by the system. Although this does not ensure the correct fulfilment of the service, but will most likely lead to a bigger incentive for the system actor to ensure a correct execution. Note that monitoring can happen continuously and automatically by using a software monitor, or can be performed occasionally and manually (by an auditing company, for instance).

Example The airline reservation system can enable external actors (i.e., airline employees, or travel agency employees) to observe its actions. For example, a travel agency employee could be able to request information about a reservation, to ensure its correctness. In this way, additional assurance is given to the travel agencies about the reservations, and the trust relationship is restored. Alternatively, an external audit company could be responsible for discovering misbehaviour of the airline reservation system.

Architectural support pattern The system should be extended with a status collector component (Figure 3.17), which gets informed about the status of each service execution, and stores this information. This status collector can be a separate component dedicated to this purpose, or can, for instance, be part of the auditing and logging infrastructure.

Change guidance To implement this solution when the trust relationship changes, the status information obtained by the status collector must be made available to the monitor component (Figure 3.18). This could be achieved entirely using software, or the information could be made available to humans

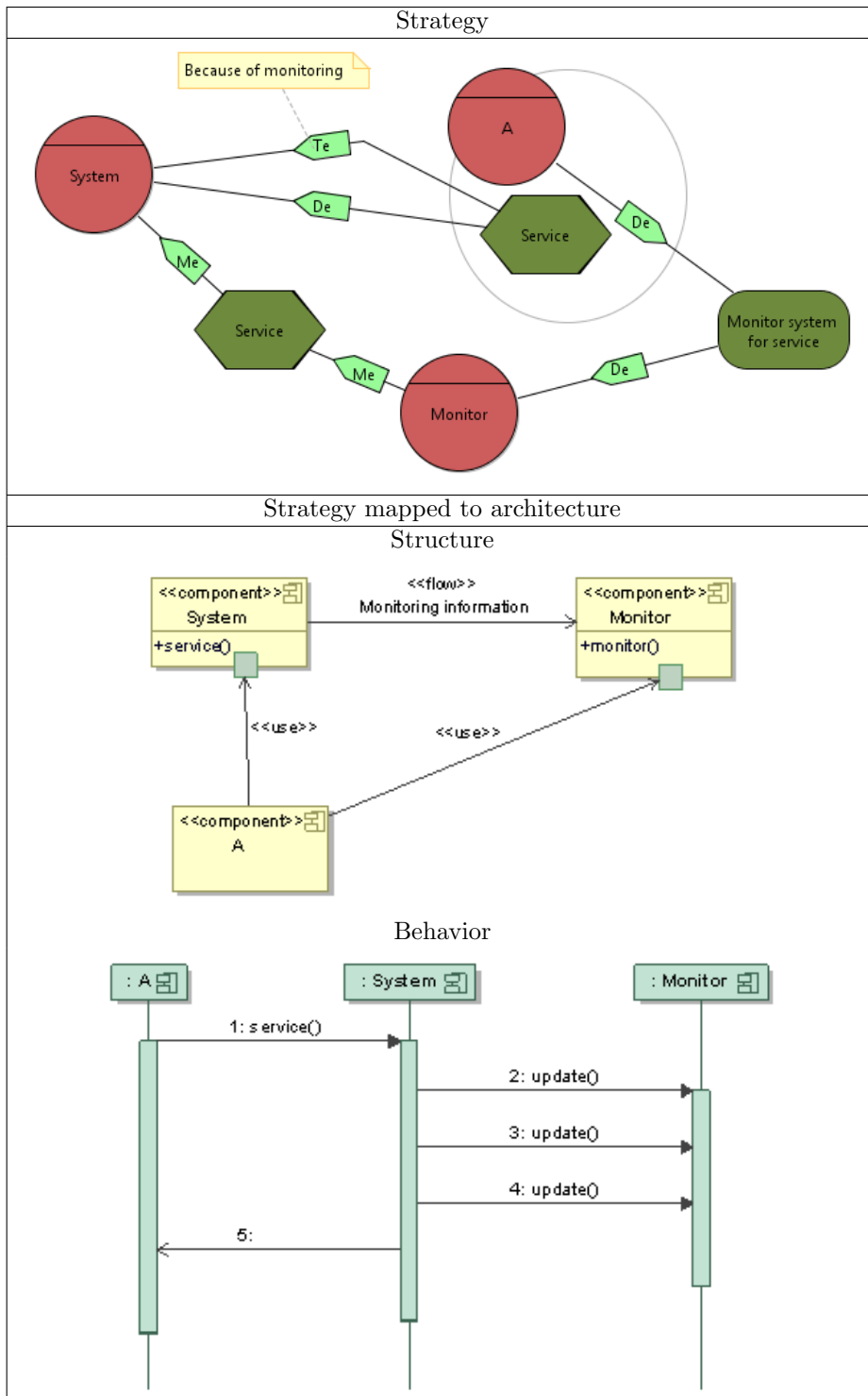


Figure 3.16: Enable monitoring

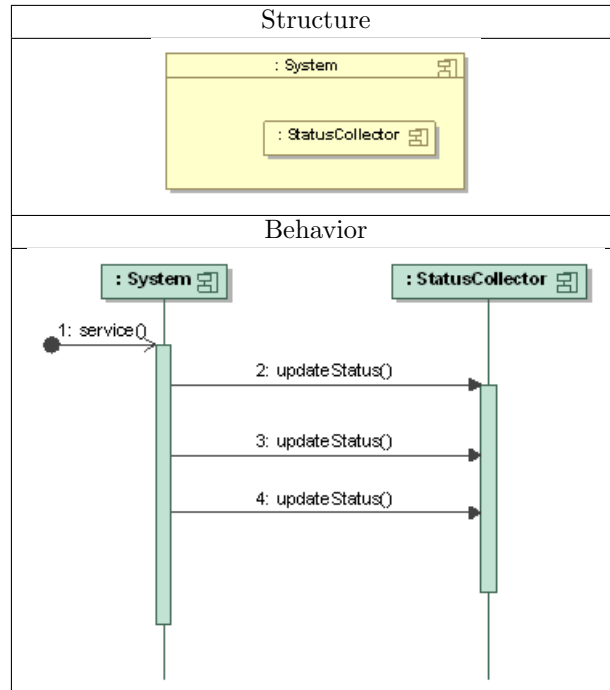


Figure 3.17: Status collector architectural support pattern

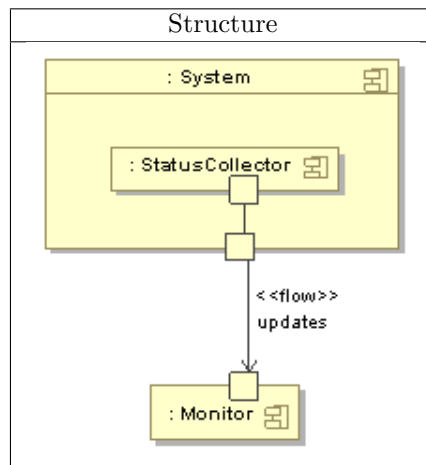


Figure 3.18: Enable monitoring using a status collector

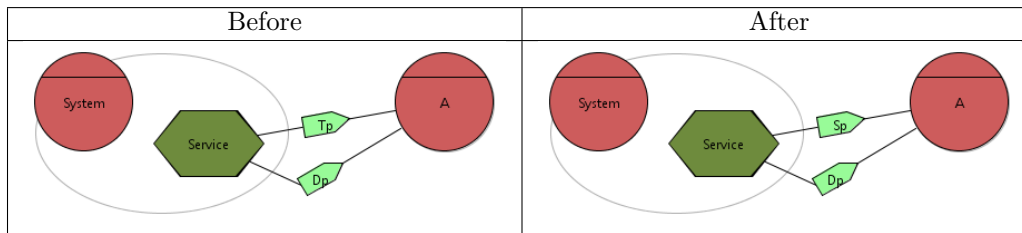


Figure 3.19: Evolving trust of permission upon external actor

(e.g., displaying on a screen). Whatever the used mechanism, the information needs to be accessible to the outside of the system.

3.3.3 Evolving trust of permission upon external actor

This change pattern does not deal with trust of execution, but rather with trust of permission. Suppose a system allows an external actor to fulfil a service, and trusts this actor not to abuse this permission. Over time, this trust relationship can disappear, so that the actor still receives the permission (as it may need it to perform its work), but is no longer trusted with it. This scenario is shown in Figure 3.19.

Example The system we are building is a hospital information system. The system gives permission to another actor, a nurse of the hospital, to access all information about the patients. Initially, the hospital trusts the nurse not to abuse this permission. After the nurse was caught using this permission to gain illegitimate access to a celebrity’s medical records, the hospital system does not trust the nurses anymore.

3.3.3.1 Solution 1: Apply least-privilege principle

Instead of giving the external actor permission to the complete service, the permission can be made more fine-grained. If the service consists of (or can be split into) multiple sub-services, giving permission to execute a small subset of these may suffice; access to all other sub-services can be denied. This corresponds to applying the well-known principle of least privilege, and is shown in Figure 3.20.

Example Instead of having access to all known information about a patient, the nurse in the hospital may now only access the information which she needs for her job. All other information is not accessible to her anymore.

Applying least privilege may have a significant impact on the architecture, however. Therefore, it is hard to create an architecture that can, in the future, be modified to comply with the least privilege principle without requiring significant alterations to the architecture. This means that picking this solution would require the architect to design the architecture from the beginning according to the least privilege principle. If this is impossible, one of the other solutions needs to be chosen.

3.3.3.2 Solution 2: Attribute-based access control

Instead of giving the external party permission to the complete service, an access control policy can be put in place (Figure 3.21). The policy will define permissions based on the identity of the external party. The permissions can be further refined by attaching conditions that depend on the context (e.g., parameters for the service, attributes of the subject or a resource, the current time, etc.).

Example The nurse can only access the patient information of patients for who she is a designated nurse, and only within the nurse’s working hours. A nurse has to identify herself before gaining access to patient information. The context information can be obtained from multiple systems: the designated nurses for a patient are provided by the patient administration system, while the working hours of the nurse are provided by the scheduling system. This restricted access control limits the possible scope of malicious actions of the nurse, so that the trust is restored.

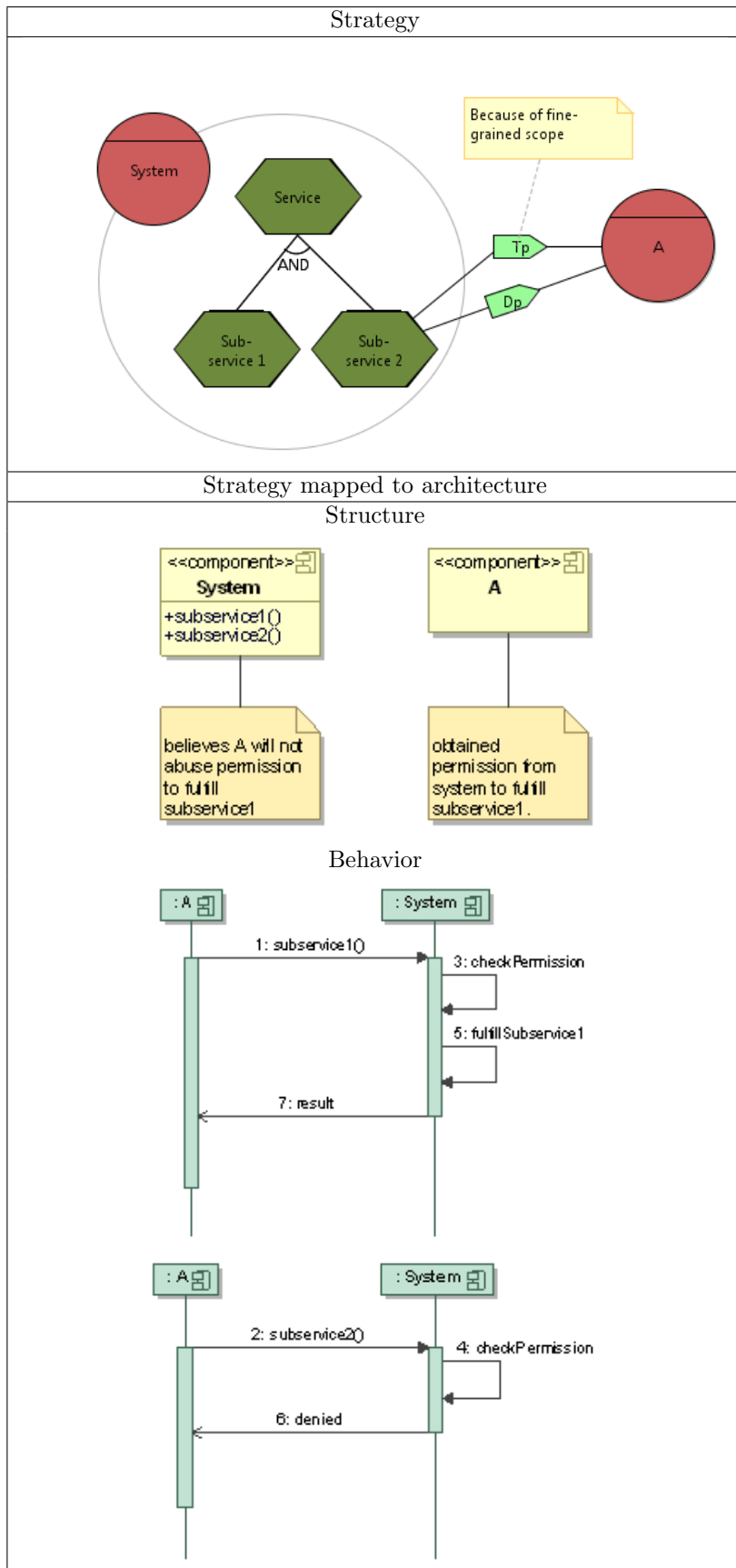


Figure 3.20: Apply least-privilege principle

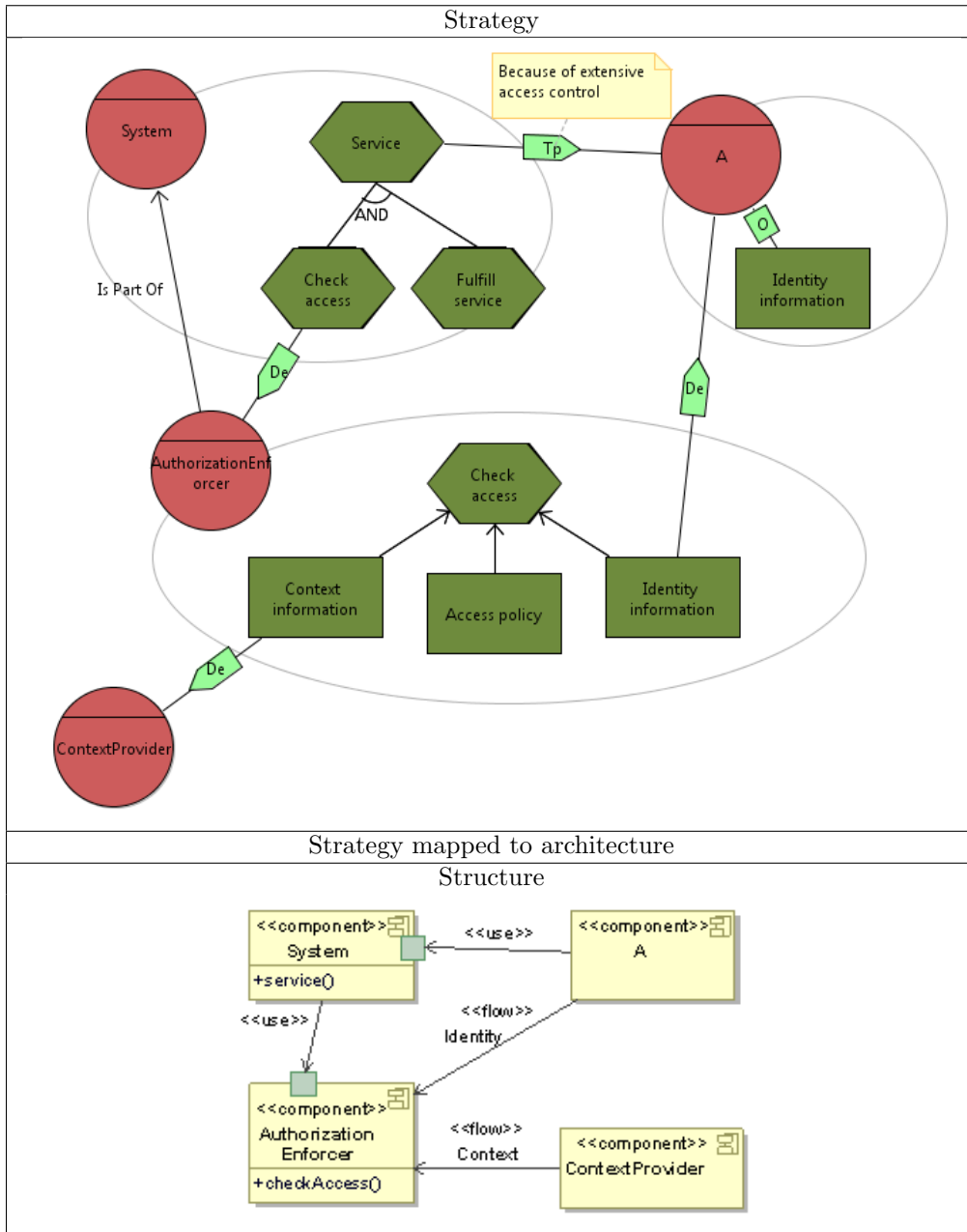


Figure 3.21: Attribute-based access control

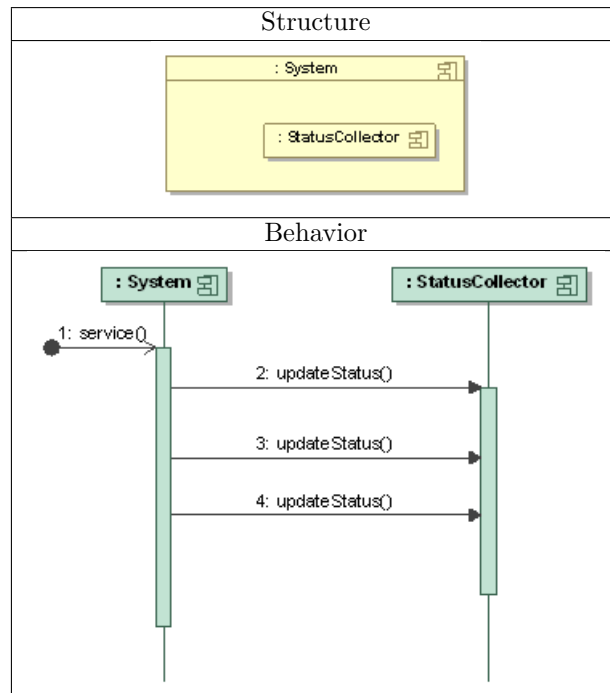


Figure 3.22: Authorization enforcer architectural support pattern

Architectural support pattern This solution requires the presence of an access control infrastructure. This can be done, for instance, by using an authorization enforcer (Figure 3.22). The authorization enforcer will, before an operation is executed, check whether the subject that invoked the operation has permission to do this.

Almost every framework or middleware has built-in support for access control. For instance, in the Java Enterprise Edition framework, role-based access control can be easily enabled for every remotely accessible operation, without writing any custom code.

If access control needs to be implemented by hand, the access control functionality can be developed using the handler pattern as described in Section 3.3.1.1. The authorization enforcer is then implemented as a handler for each operation in the system that needs access control.

Additionally, all information that is needed to make access control decisions need to be available to the authorization enforcer. This means that each component that can provide necessary information needs to act as a context provider (Figure 3.23).

Change guidance To implement this solution when the trust relationship changes, the authorization enforcer in the architecture needs to be updated with the new, attribute-based policy. It also needs to be configured such that it can access all context providers that are necessary to evaluate the new policy.

3.3.3.3 Solution 3: Use monitoring

Alternatively, all service executions can be monitored. The monitor is responsible for verifying that permissions are not abused.

Example A nurse can still access the information from all patients in the hospital, but every access attempt is logged and periodically reviewed by the responsible doctor. Because of the increased risk in being detected when illegally accessing patient files, the trust in the nurses is restored. Note that, in this case, the doctor acts as the monitor.

The technical solution is the same solution as Solution 2: Use monitoring as described in Section 3.3.1.2.

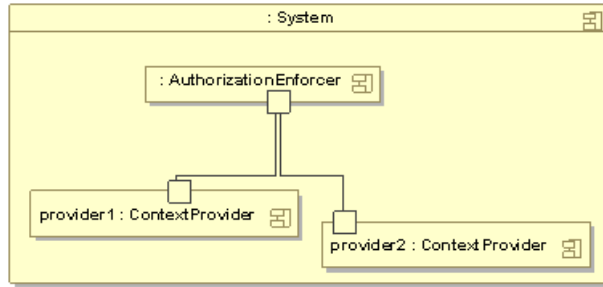


Figure 3.23: Applying the authorization enforcer support pattern

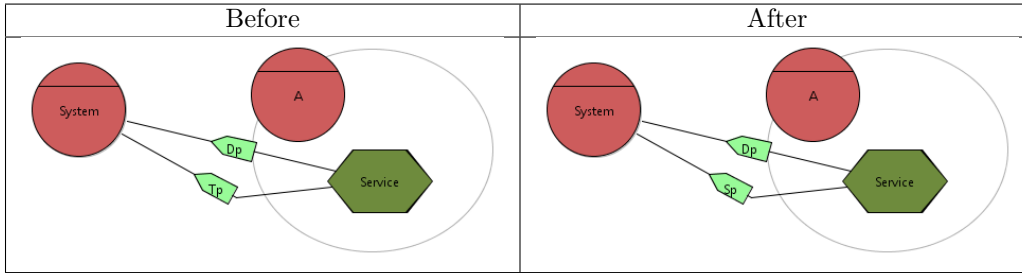


Figure 3.24: Evolving trust of permission from external actor

3.3.4 Evolving trust of permission from external actor

In this change pattern, the system obtained permission from an external actor to fulfil some service. The external party also trusts the system not to abuse this permission. Later, however, this trust disappears. This is shown in Figure 3.24.

Example The system we are building is a hotel booking website. The site obtains permission from its customers to process the credit card information of the user. The users trust the hotel site not to abuse this information. When illegitimate use of the information is detected (e.g., the credit card is used for additional purchases), the users no longer trusts the hotel with their credit card information.

3.3.4.1 Solution 1: Request confirmation

Each time the system performs a service, it needs to request a confirmation from the external party that the service may be fulfilled, as illustrated in Figure 3.25.

This confirmation can take various forms. For instance, it could be digitally signed evidence created by the external party whenever necessary. The system could store the obtained permission, in case of later disputes. Alternatively, if some piece of information that is required to fulfil the service is never stored by the system, and must always be provided by the external system, the submission of this information by the external party can be seen as a confirmation as well. The latter case is what happens with the CVV2 number on credit cards.

Example The hotel booking website needs the CVV2 number to initiate a transaction with the payment gateway. Since the credit card security standards require the system never to store this number, the hotel booking website will need to obtain the CCV2 number from the client for each transaction. By entering this number, the client thus gives a confirmation of the transaction to the hotel booking website.

Architectural support pattern Obtaining and verifying the confirmation needs to be done before the service execution. This can be achieved using the handler architectural pattern from Section 3.3.1.1.

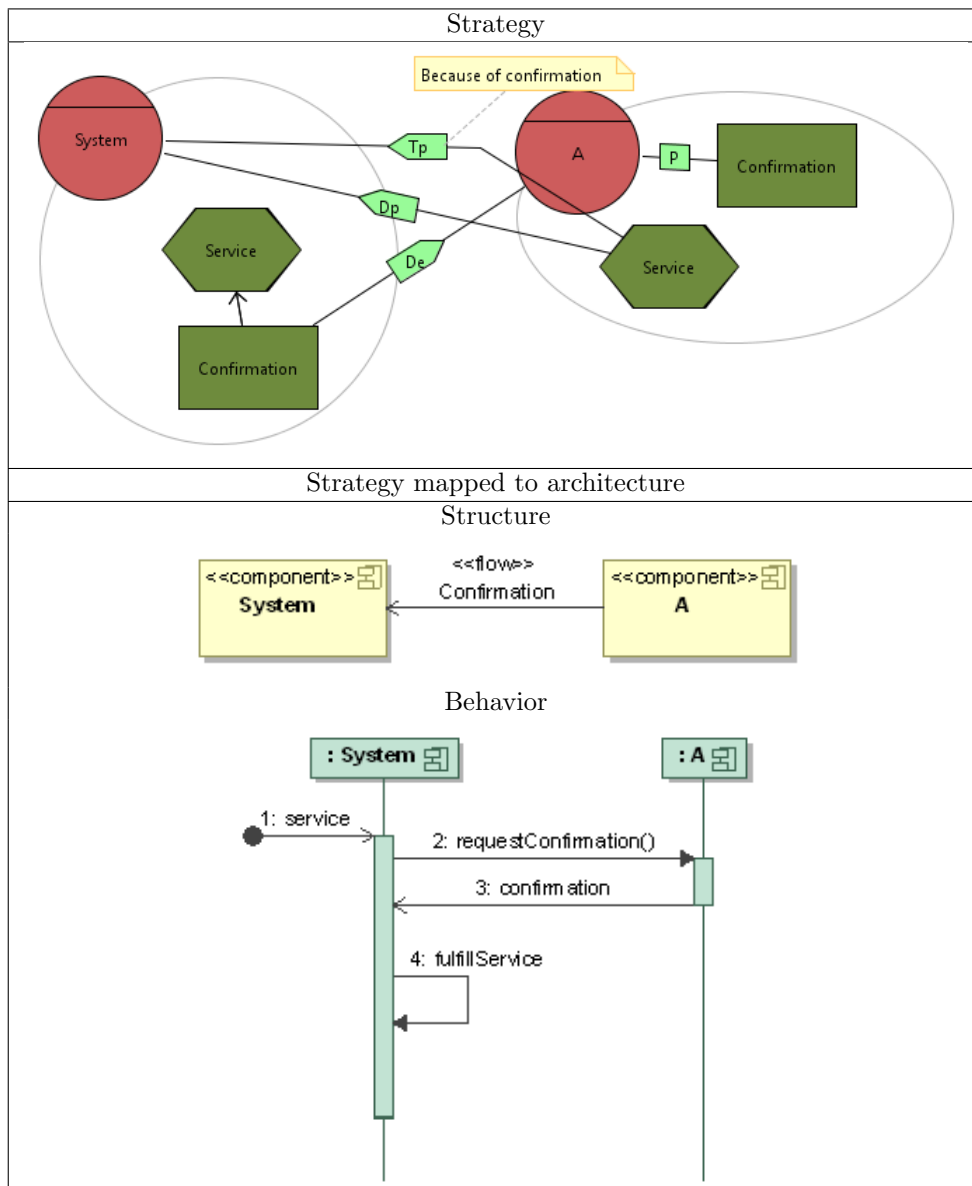


Figure 3.25: Request confirmation

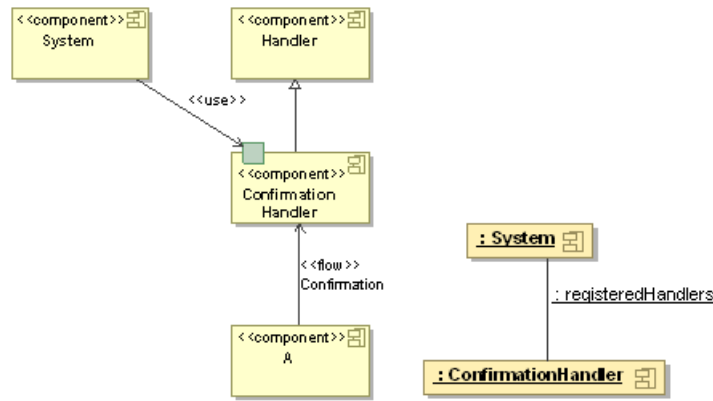


Figure 3.26: Adding a confirmation handler

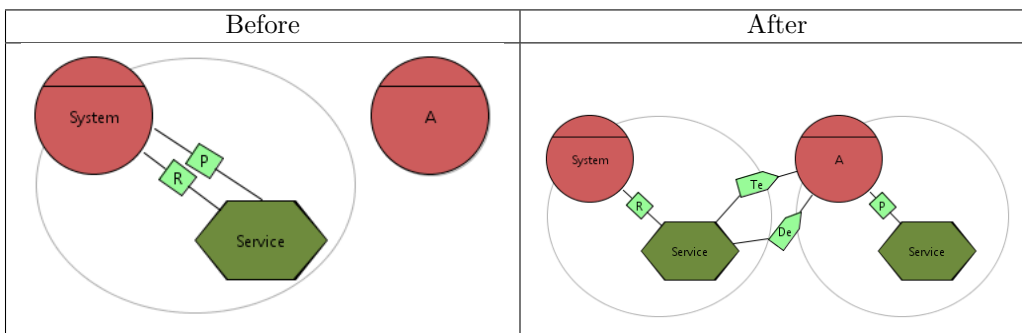


Figure 3.27: Delegate execution of a service to a trusted actor

Change guidance When the system is no longer trusted with the permission for a service by the external party, a confirmation handler is developed and configured for the service (Figure 3.26). This confirmation handler will take care of obtaining and verifying the confirmation, before allowing the service to be fulfilled.

3.3.4.2 Solution 2: Enable monitoring

Alternatively, the system can allow the monitoring of its service fulfilment.

Example The credit card company can monitor the transactions of the hotel booking system. With this information, the credit card company can offer a service to notify the customer of all initiated transactions concerning his credit card.

The technical solution is the same as ‘Solution 2: Enable monitoring’ described in Section 3.3.2.2.

3.3.5 Delegate execution of a service to a trusted actor

For this change scenario, assume the system uses a certain service that it provides itself. Over time, the provisioning of the system may move to an external actor (e.g., outsourcing), as shown in Figure 3.27. We assume that the external party is trusted by the system for executing this service. If this is not the case, the change pattern from Section 3.3.1 needs to be considered together with this pattern.

Example Consider a route planning system. Originally, the system was designed for a single country, and used its custom written software for geo-coding (converting street names to geographic coordinates). When the system is extended to work internationally, an external geo-coding service is used. The system needs to be modified so that this external service is used.

3.3.5.1 Solution: Encapsulate service

To easily enable the transition from the internal to the external service implementation, the service should be encapsulated from the rest of the application, as illustrated in Figure 3.28. This is a well-known design solution for creating maintainable software solutions.

The service can now be changed from the internal to the external implementation, without modifications to the rest of the system. As an additional advantage, if the external actor decides to change the functionality of the service, the service provider can be modified such that the functionality exposed to the system does not change.

Example The route planner system contains a geo-code component. Originally, the geo-code component contains the custom implementation. When the decision is made to use an external service, the implementation is changed to access that service.

Architectural support pattern A service provider component should be introduced in the architecture, as shown in the strategy mapped to the architecture above.

Change guidance When the change scenario occurs, the implementation of the service provider needs to be modified such that it uses the external service instead of the internal implementation. The rest of the system remains unchanged.

3.3.6 Delegate permission to a service to a trusted actor

In this change scenario, assume the system owns a certain service. Over time, the system may be opened for external actors. This means that external actors gain permission to fulfil the service. Or, the service may already be available for some external parties, but needs to be available to an additional one. This scenario is shown in Figure 3.29. We assume that the external party is trusted by the system for not abusing this permission. If this is not the case, the change pattern from Section 3.3.3 needs to be considered together with this pattern.

Example Suppose the system we are developing is a social network site. The system owns information about its users. Over time, the social network site wants to allow an advertising company to access this information, in order to deliver personalized advertisements. Of course, other external parties should not be allowed access to this data.

Note that this scenario will often happen together with the scenario in Section 3.3.7.

3.3.6.1 Solution: Flexible access control

A solution for this evolution scenario is putting flexible access control in place (Figure 3.30). Before the service is fulfilled, the request is evaluated by an access control monitor. The policy that is enforced by this actor should be easy to modify, preferable through updating the configuration.

Example The social network site creates an API for obtaining user data, but restricts access to this API to the advertising company.

Architectural support pattern This solution requires the presence of an access control infrastructure, as described in Section 3.3.3.2.

Change guidance To implement this solution when the new trust relationship appears, the policy of the authorization enforcer in the architecture needs to be updated to grant permission to the new external party.

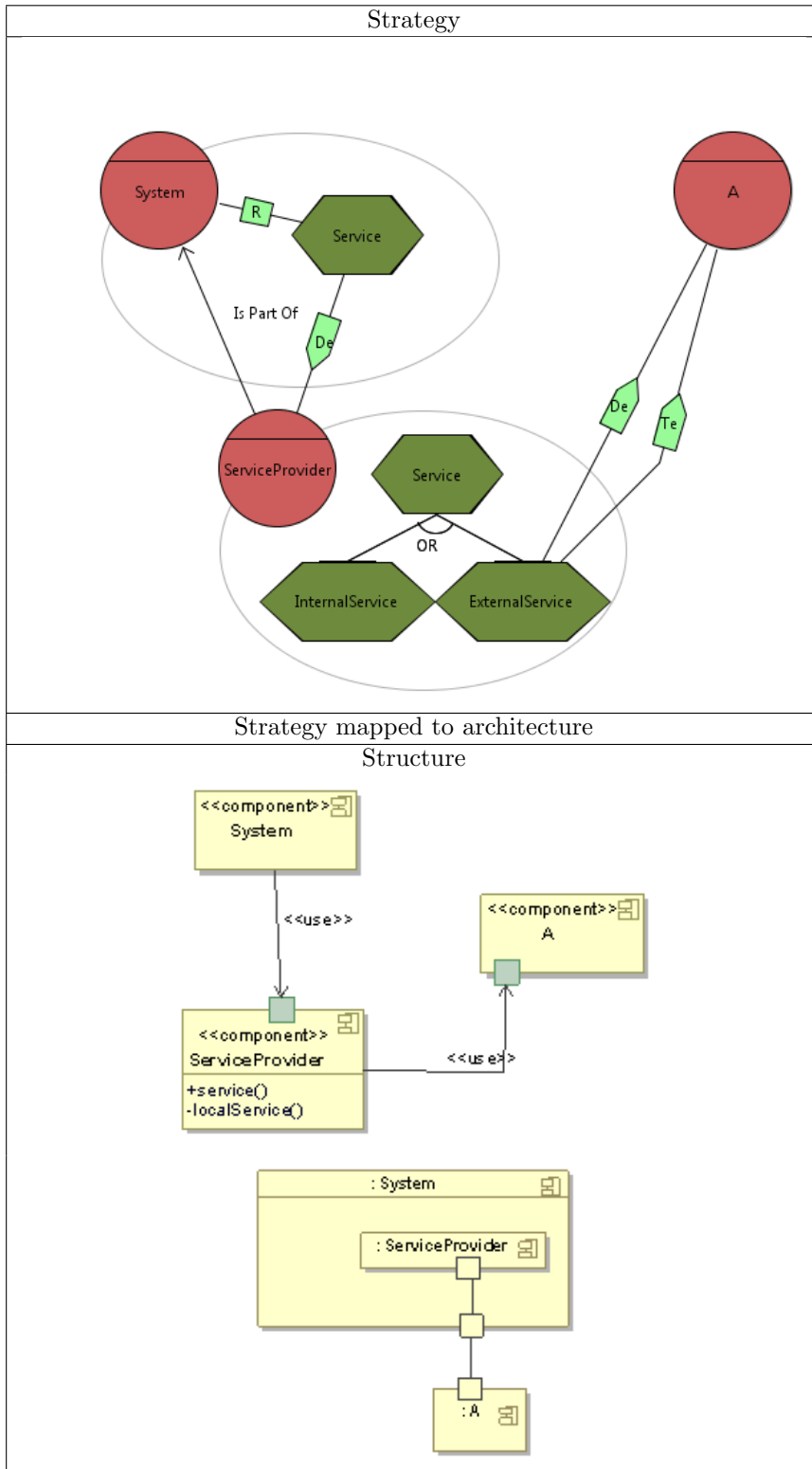


Figure 3.28: Encapsulate service

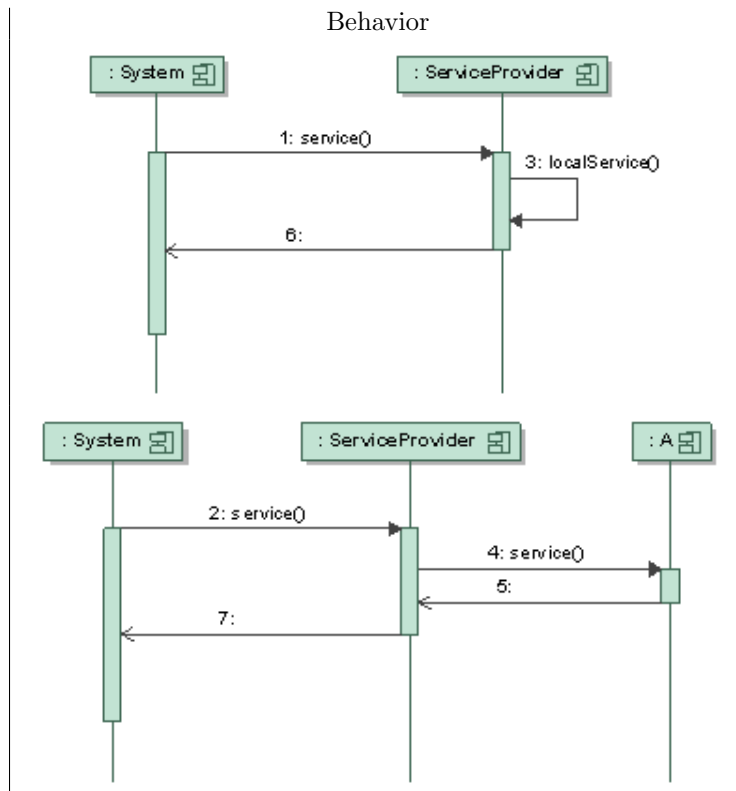


Figure 3.28: Encapsulate service (continued)

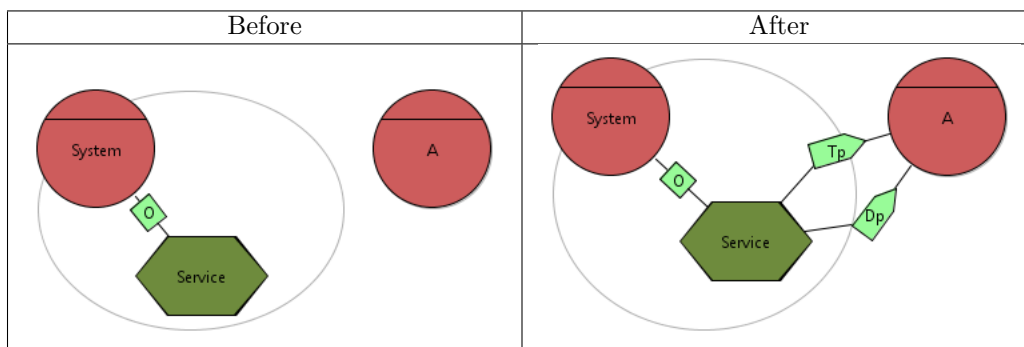


Figure 3.29: Delegate permission to a service to a trusted actor

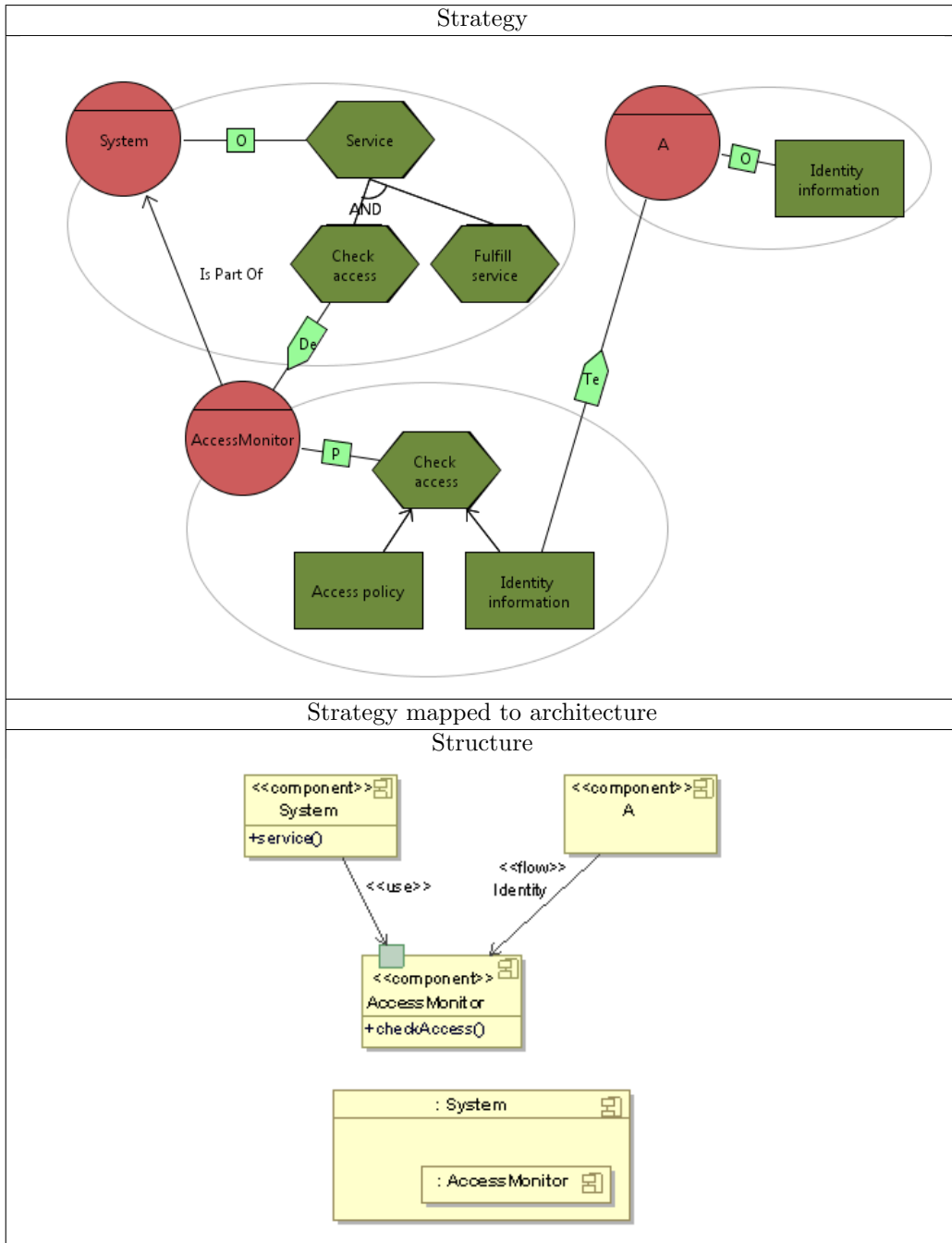


Figure 3.30: Flexible access control

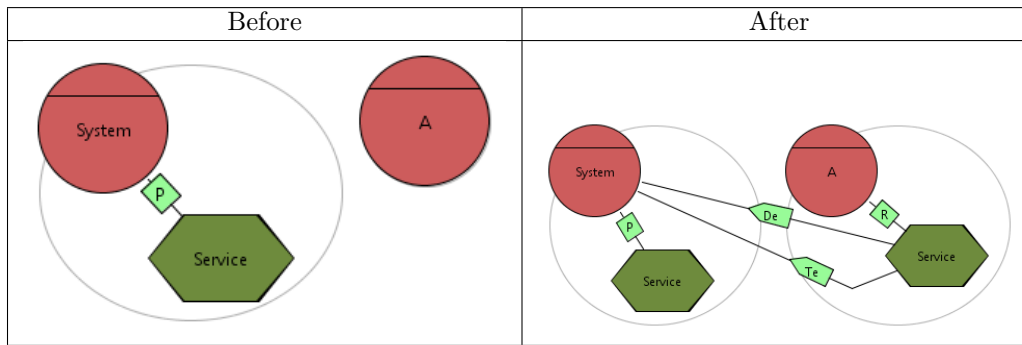


Figure 3.31: Providing additional service with delegated execution

3.3.7 Providing additional service with delegated execution

In this scenario, consider a service of the system originally only used by the system itself. Over time, an external party may decide to rely on the system for this service. This means that the system exposes new (but already implemented) services to its environment. Since external parties do now rely on this service, the service cannot be modified without breaking the external actors. This scenario is shown in Figure 3.31.

Note that we assume that the external party trusts the system for executing this service. If this is not the case, the change pattern from Section 3.3.2 needs to be considered together with this pattern.

Example The system under development is a system for a car rental company. An important service offered by the system is creating a reservation for a car. Originally, the reservation service is only used within the car rental company, but over time the car rental company wants to give travel agencies the opportunity to create car reservations as well. Once the first travel agency is connected to their system, changing the reservation service becomes difficult because the travel agency depends on it.

3.3.7.1 Solution: Introduce bridge component

Introduce a bridge component for the service, which remains stable over time. External systems connect with this bridge component. When the internal implementation of the service changes, the bridge is modified such that it provides the same functionality, but uses the new implementation. It achieves this by converting the input given to the internal service, and result that is returned. In this way, the changed implementation has no observable effects. The solution is illustrated in Figure 3.32. This solution is a widely-known and common technique to create evolvable systems.

Example The car rental company implements a service bridge, which is used by the external systems. When the internal implementation of the service needs to change, the bridge is modified as well so that nothing has changed from the viewpoint of the external systems.

Architectural support pattern A service bridge component should be introduced in the architecture, as shown in the strategy mapped to the architecture above.

Change guidance When the change scenario occurs, the external system is connected to the service bridge instead of to the internal implementation of the service.

3.3.8 Providing additional service with delegated permission

In this final scenario, consider a system that owns and provide a certain service. Over time, the ownership of the service may be transferred to another actor, while the system keeps providing the service, as shown in Figure 3.33.

Since this evolution has no influence on the behaviour of the system (because the system will still provide the service, just as before), no architectural solutions are necessary for this evolution scenario.

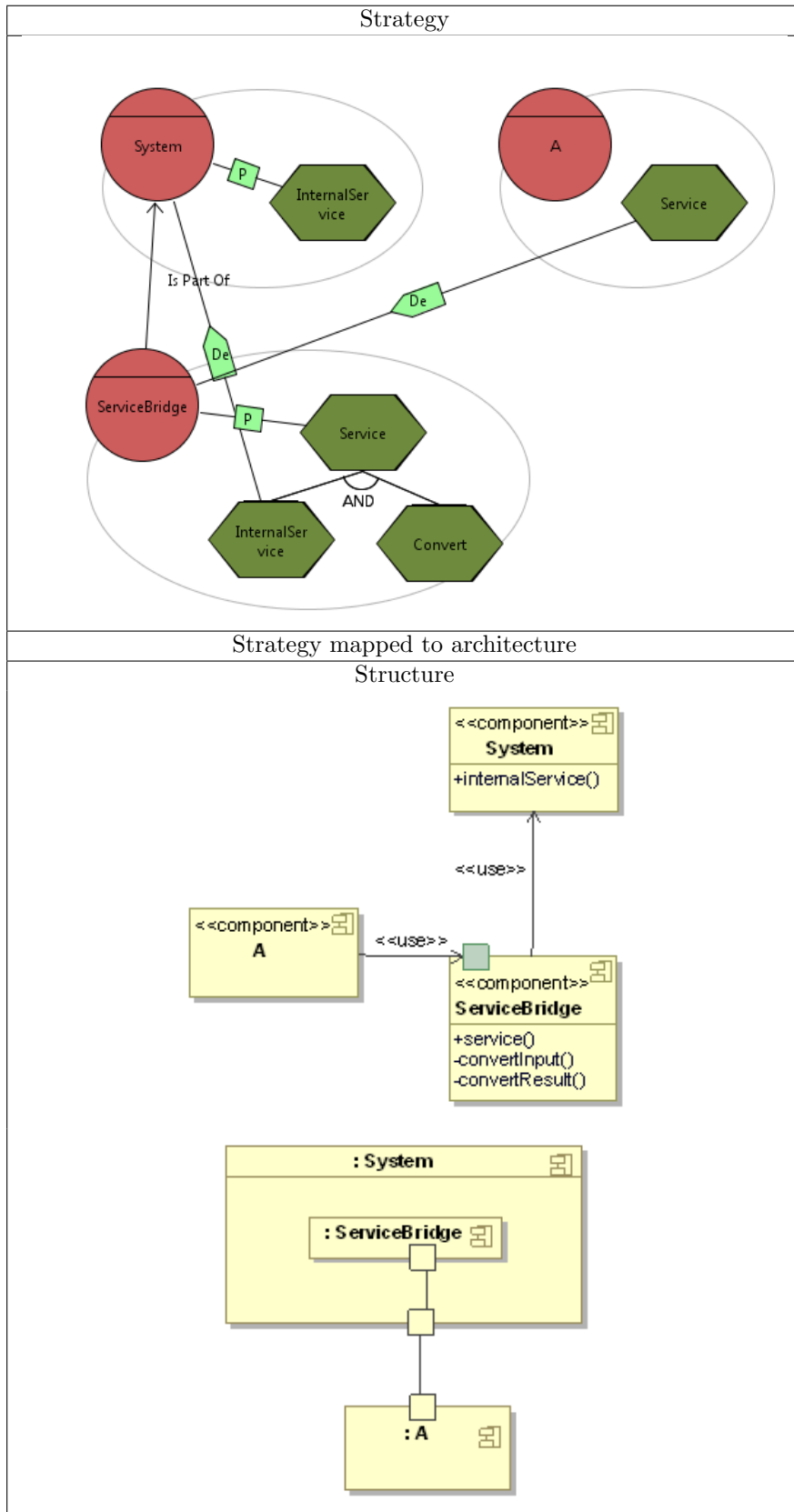


Figure 3.32: Introduce bridge component

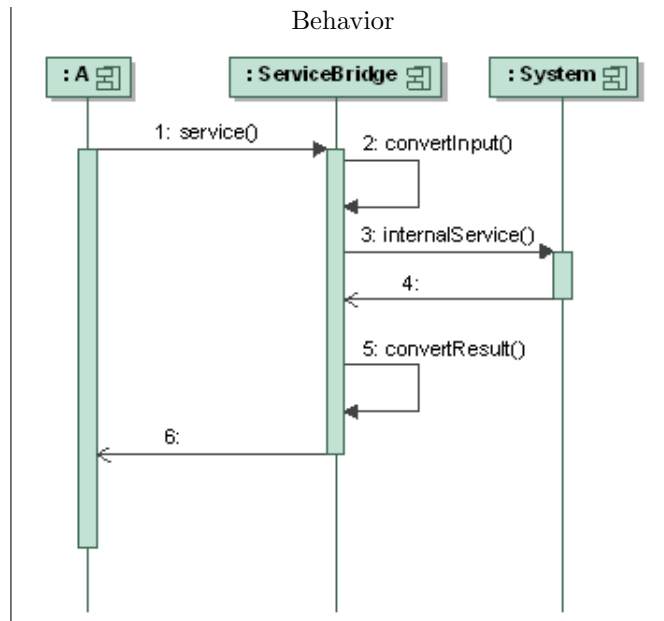


Figure 3.32: Introduce bridge component (continued)

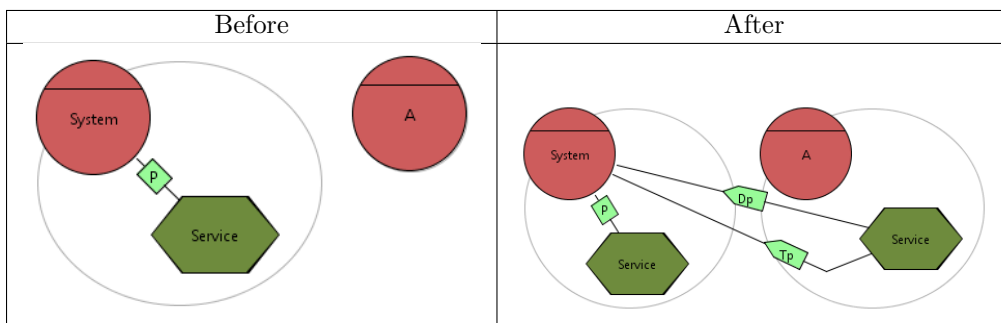


Figure 3.33: Providing additional service with delegated permission

3.3.1 – Evolving trust of execution upon external actor		
×	Current situation:	Shop trusts client to provide purchase order acknowledgement
	Possible change:	Client is no longer trusted to do this
	Likelihood:	Very high
	Current situation:	Shop trusts credit card company to correctly process payment
	Possible change:	Credit card company is no longer trusted
	Likelihood:	Very low
3.3.2 – Evolving trust of execution from external actor		
×	Current situation:	Client trusts shop to correctly handle orders and sell goods
	Possible change:	Shop is no longer trusted to handle orders and sell goods
	Likelihood:	Medium
3.3.3 – Evolving trust of permission upon external actor		
	No matched identified	
3.3.4 – Evolving trust of permission from external actor		
×	Current situation:	Client trusts shop not to abuse credit card information
	Possible change:	Shop is no longer trusted with credit card information
	Likelihood:	High
3.3.5 – Delegate execution of a service to a trusted actor		
	Current situation:	Shop takes care of order shipment itself
	Possible change:	Shipping of orders is outsourced to dedicated shipping company
	Likelihood:	Low
3.3.6 – Delegate permission to a service to a trusted actor		
	Current situation:	Shop does not share customer preferences
	Possible change:	A marketing company is given access to customer preferences
	Likelihood:	Low
3.3.7 – Providing additional service with delegated execution		
	Current situation:	The shop only ships its own orders
	Possible change:	An external company uses the shipping services from the shop
	Likelihood:	Low
3.3.8 – Providing additional service with delegated permission		
	No match identified	

Table 3.2: Result of following the change pattern process

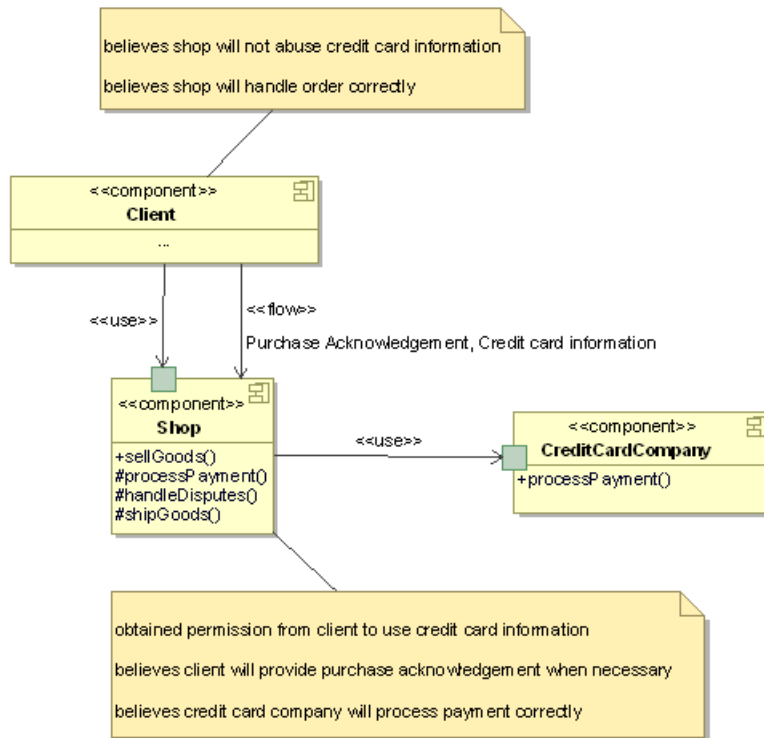


Figure 3.35: Corresponding architecture

upon an external actor disappears. In the shop system, there are two trust relationships from the shop upon external actors.

First, the shop system trusts the client to provide acknowledgements of their purchase. While the initial shop system may be designed to cooperate with a limited number of trusted customers, it is easy to imagine a situation in which this trust relationship is unjustified. Therefore, the likelihood of clients not providing this acknowledgement is very high, and the stakeholders will require that the architecture is prepared for this situation.

The second matching trust relationship is that upon the credit card company for correct payment processing. While it can be imagined that the credit card company will not be trusted anymore, the stakeholders have determined that this is unlikely to happen in the foreseeable future, and as such support for this evolution scenario is not included in the current architecture.

The other rows in the table are obtained in a similar way, by matching with the other patterns from the catalogue. Eventually, the rows with a 'x' were selected as likely and important scenarios by the stakeholders, and will need to be resolved in the architecture.

The architect now chooses solutions for each of the chosen evolution scenarios. This choice is based on architectural trade-offs, and relies on the knowledge and experience of the architect. Suppose the architect prefers the following solutions:

- For the first evolution scenario (where the client is no longer trusted to acknowledge its purchases), the architect chooses the solution based on requiring a commitment of the client (Section 3.3.1.1).
- The second evolution scenario (where the shop is no longer trusted to correctly handle orders), the solution in Section 3.3.2.1 is chosen (providing a commitment).
- For the third evolution scenario, the solution based on monitoring (Section 3.3.4.2) is picked.

The architect now has to integrate all architectural support patterns of these solutions into the architecture of the shop. This is a manual effort, and the result is depicted in Figure 3.36 (only the structural part has been shown). Recall that this is the result before any of the aforementioned evolutions occurred; only the necessary support infrastructure has been put in place. It is apparent that the introduction of

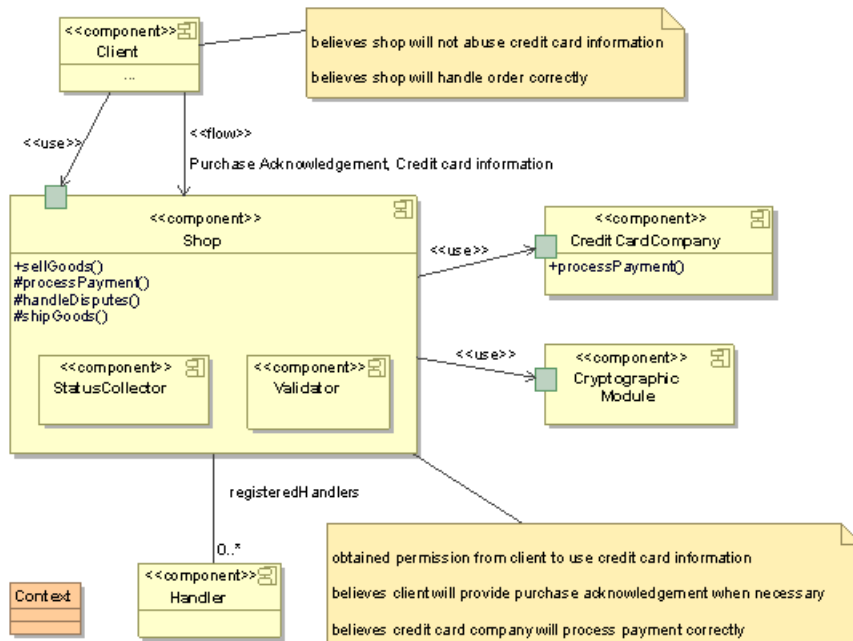


Figure 3.36: Resulting architecture

the architectural support patterns has significantly influenced the architecture. Since no evolution has occurred, however, all the trust assumptions of before (represented using notes) are still in place.

The introduction of the handler pattern enables the registration of handler components with the shop, which will be invoked before (and after) every order made by a client. The newly added status collector keeps track of the order history, and the status of each order. The validator component validates the parameters of a purchase order, before the order can be fulfilled. Finally, one of the change patterns prescribes that the system needs to have access to a cryptographic module, capable of placing digital signatures.

We stress that the system, using this architecture, does not yet comply with any of the evolution scenarios. The architecture of the system has merely been prepared to handle the evolution scenarios, should they occur in the future. However, remark how, for example, the need for keeping an order history has emerged from analyzing possible trust evolutions using change patterns. Of course, this feature would most likely be present in any commerce system, but our analysis has highlighted its relevance with regard to evolving trust.

3.4.3 Handling evolution

In this section, we will replay the evolution from the motivating example at start of the chapter, but now with the architecture created in the previous subsection.

In the first part of the story, the trust between the shop and the client disappears. Since this trust evolution matches with the change pattern in Section 3.3.1 (Evolving trust of execution upon external actor), for which the first solution was chosen, the change guidance from Section 3.3.1.1 needs to be followed. This guidance prescribes the development of a commitment handler, and deploy it as a registered handler for the shop system. This means that a new handler component needs to be developed, which extracts the digitally signed commitment from the order request created by the client, validates the commitment and stores it. This handler then needs to be registered with the shop component. These actions happen in the implementation and deployment domain, and can be performed without modifications to software architecture of the system. Indeed, the introduction of the handler pattern, as prescribed by the change pattern, enables easy addition of this functionality.

For the second evolution, the trust from the client upon the shop for correctly using their credit card information disappears. This evolution also corresponds with a change pattern applied to the architec-

ture, namely “Evolving trust of permission from external actor” (Section 3.3.4). Here, the architect has chosen the solution of enabling monitoring (Section 3.3.4.2). That solution’s change guidance describes that the status information, obtained by the status collector component, must be made available to the monitor. In this example, the monitor is the client component itself, and the status information is the order history. To overcome the decreasing trust, the order history should thus be made available to the clients of the shop, if this was not already the case before. The clients need this order history to align the order history with the notifications from the credit card company.

In the third (and final) part of the scenario, the clients lose their trust in the shop system entirely, and the change guidance from the chosen solution (providing a commitment, Section 3.3.2.1) must be followed. A new service needs to be provided by the shop, which provides a commitment to the clients who request one. Note that this is an architectural change, because a new service is introduced at the side of the shop. However, the newly introduced service is not essential, and clients who do not require the commitment can still place their orders. Old clients do not have to be changed immediately, and therefore the change only has a small impact on the overall software architecture.

The illustration above demonstrates how the use of change patterns for designing an architecture enables evolution with little to no impact on the architecture. Of course, the example consists of a simplified scenario. Additional validation, on a more realistic case, is necessary.

Chapter 4

Conclusion

Change patterns are proposed as a helpful concept for designing secure, evolvable systems. A change pattern is attached to a change scenario, which represents a high-level evolution of the security requirements of the system. The change pattern also contains one or more solutions. Each solution may refer to some architectural support patterns, which prepares the architecture for an upcoming evolution scenario. Additionally, the solution contains change guidance, which describes the necessary steps to update the architecture such that it reflects the new security requirements, once these have changed.

Dealing with evolution of security at the architectural level consists of three steps. First, likely evolution scenarios need to be identified, for instance by questioning whether each of the change patterns is applicable and likely to happen. If so, the second step needs to be executed: picking a solution to handle this case needs and incorporate it in the architecture. This step prepares the architecture for the expected evolution. The third step consists of updating the system once the identified evolution actually happens. The main goal of the change patterns is dramatically decreasing the possible impact on the architecture during this last step.

We presented a catalogue of change patterns that deal with changing trust relationships in a component-oriented architecture. The techniques to deal with these changing trust relationships can be divided in two broad categories. One category are the typical, general principles to achieve maintainability in a software architecture: decoupling the points of variation from the rest of the system, so that they can evolve independently. The second category is more specific for trust relationships, and involves the introduction of a monitoring infrastructure.

The proposed approach requires an explicit representation of all the trust relationships involving the system. Moreover, when following the approach, each trust relationship will be questioned and assessed. This systematic approach helps the architect in achieving a more complete solution when compared to identifying and resolving the likely change scenarios in a more ad-hoc manner.

Bibliography

- [BWJ] K. Buyens, BD Win, and W. Joosen. Resolving least privilege violations in software architectures. In *SESS'09: Proceedings of the 5th International Workshop on Software Engineering for Secure Systems*.
- [CEKK⁺09] L. Compagna, P. El Khoury, A. Krausová, F. Massacci, and N. Zannone. How to integrate legal requirements into a requirements engineering methodology for the development of security and privacy patterns. *Artificial Intelligence and Law*, 17(1):1–30, 2009.
- [DC07] L. Dai and K. Cooper. A Survey of Modelling and Analysis Approaches for Architecting Secure Software Systems. *International Journal of Network Security*, 5(2):187–198, 2007.
- [GMZ05] P. Giorgini, F. Massacci, and N. Zannone. Security and trust requirements engineering. *Lecture notes in computer science*, 3655:237, 2005.
- [GY01] D. Gross and E. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001.
- [HAJ07] M. Hafiz, P. Adamczyk, and R.E. Johnson. Organizing security patterns. *IEEE software*, pages 52–60, 2007.
- [Hil00] R. Hilliard. IEEE-Std-1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE*, <http://standards.ieee.org>, 2000.
- [HLOS06] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack. Uncover Security Design Flaws Using The STRIDE Approach. *MSDN Magazine-Louisville*, pages 68–75, 2006.
- [Jür05] J. Jürjens. *Secure systems development with UML*. Springer Verlag, 2005.
- [LBD⁺02] T. Lodderstedt, D. Basin, J. Doser, et al. SecureUML: A UML-based modeling language for model-driven security. *Lecture notes in computer science*, pages 426–441, 2002.
- [LPR98] B. Len, C. Paul, and K. Rick. Software architecture in practice. *SEI Series in Software Engineering*, Addison Wesley, 1998.
- [MMZ07] F. Massacci, J. Mylopoulos, and N. Zannone. An ontology for secure socio-technical systems. *Handbook of Ontologies for Business Interaction*, page 188, 2007.
- [PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):52, 1992.
- [RT05] J. Ren and R. Taylor. A secure software architecture description language. In *Workshop on Software Security Assurance Tools, Techniques, and Metrics*. Citeseer, 2005.
- [SYHJ08] R. Scandariato, K. Yskout, T. Heyman, and W. Joosen. Architecting software with security patterns. *CW Reports, volume CW515, Department of Computer Science, KU Leuven*, 2008.
- [VL03] A. Van Lamsweerde. From system goals to software architecture. *Lecture Notes in Computer Science*, pages 25–43, 2003.
- [Wei06] M. Weiss. Modelling Security Patterns Using NFR Analysis. *Integrating Security and Software Engineering: Advances and Future Visions*, page 127, 2006.

- [YWM08] N. Yoshioka, H. Washizaki, and K. Maruyama. A survey on security patterns. *Progress in Informatics*, 5:35–47, 2008.