D2.1 - An architectural blueprint and a software development process for security-critical lifelong systems

Ruth Breu, Mukhtiar Memon, Frank Innerhofer-Oberperfler, Manuela Weitlaner, Michael Breu, Michael Hafner (UIB)
Riccardo Scandariato, Koen Yskout, Koen Buyens (KUL)
Benjamin Fontan (THA)
Federica Paci, Elisa Chiarani (UNITN)

Document information

Document Number	D2.1
Document Title	An architectural blueprint and a software development process for security-critical lifelong systems
Version	2.2
Status	Final
Work Package	WP 2
Deliverable Type	Report
Contractual Date of Delivery	31 January 2010
Actual Date of Delivery	19 January 2010
Responsible Unit	KUL
Contributors	KUL, UIB, THA, UNITN
Keyword List	Reference Architecture, Process Model
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	17 Sep 2009	Draft	K. Yskout (KUL)	Outline
0.2	15 Oct 2009	Draft	R. Breu, M. Hafner (UIB)	D. 2.1
0.3	5 Nov 2009	Draft	M. Hafner (UIB)	Enhanced Structure
0.4	5 Nov 2009	Draft	M. Hafner (UIB)	Draft Section 3.1
0.5	6 Nov 2009	Draft	K. Yskout (KUL)	Draft section 1, 2, 4
0.6	13 Nov 2009	Draft	M. Hafner (UIB)	Draft Section 2.4
0.7	17 Nov 2009	Draft	B. Fontan (THA)	Draft Section 3.3
0.8	18 Nov 2009	Draft	R. Breu (UIB)	Enhanced Meta Model
0.9	20 Nov 2009	Draft	R. Breu (UIB)	Revision of 3.1
0.10	20 Nov 2009	Draft	M. Hafner (UIB)	Minor revisions on section 5
0.11	20 Nov 2009	Draft	K. Yskout (KUL)	Enhanced section 4
0.12	9 Dec 2009	Draft	K. Yskout (KUL)	Restructuring, introduction
0.13-0.14	10 Dec 2009	Draft	M. Hafner (UIB)	Added HOMES Case Study to Sec. 5
0.15	10 Dec 2009	Draft	B. Fontan (THA)	Update Conceptual Models of DSML; Define Global model of DSML
0.16	10 Dec 2009	Draft	M. Hafner (UIB)	Finished HOMES Case Study in Sec. 5; related Sec. 5 to Sec 3 and 4
0.17-0.18	13 Dec 2009	Draft	R. Breu (UIB)	Executive Summary and Conclusion
0.19	14 Dec 2009	Draft	F. Innerhofer-Oberperfler (UIB)	Enhanced introduction and section 3 with integrated process
0.20	14 Dec 2009	Draft	K. Yskout (KUL)	Background, section 4, bibliography
0.21	15 Dec 2009	Draft	F. Innerhofer-Oberperfler (UIB)	Section 3: case study, refinement of process, integration of UIB with THA process
0.22	16 Dec 2009	Draft	M. Hafner (UIB)	Completed Section 2.3.1
0.23-0.24	16 Dec 2009	Draft	K. Yskout (KUL)	Extended section 1.2, 1.3, 2.2.2 and 4.6
0.25	17 Dec 2009	Draft	R. Breu (UIB)	Minor changes (typos etc.)
0.26	18 Dec 2009	Internal review	K. Yskout (KUL)	Version for internal review
1.0	4 Jan 2010	Draft	F. Paci (UNITN)	Internal review

1.1	07 Jan 2010	Draft	B. Fontan (THA)	Add connections between WP2 metamodel and THA metamodel in section; Fix some spelling mistakes; Simplify THA_Change_Model
1.2	08 Jan 2010	Draft	R. Breu (UIB)	Changes concerning the internal review comments
1.3	08 Jan 2010	Draft	F. Innerhofer-Oberperfler (UIB)	Replaced Figure 5
1.4	08 Jan 2010	Draft	E. Chiarani (UNITN)	First quality check completed; minor remarks.
1.5	11 Jan 2010	Draft	K. Yskout (KUL)	Implemented remarks from first quality check.
1.6	12 Jan 2010	Draft	K. Yskout (KUL)	Implementing internal review comments: general, section 4, glossary, bibliography
1.7	13 Jan 2010	Draft	K. Yskout (KUL)	Additional implementation of internal review comments for section 2 and 4.
1.8	14 Jan 2010	Draft	M. Hafner (UIB)	Added glossary for section 5
1.9	14 Jan 2010	Draft	F. Innerhofer-Oberperfler (UIB)	Added glossary for section 3
1.10	14 Jan 2010	Draft	B. Fontan (THA)	Updated glossary
2.0	14 Jan 2010	Quality review	K. Yskout (KUL)	Prepared deliverable for quality check
2.1	19 Jan 2010	Draft	E. Chiarani (UNITN)	Final quality check; minor remarks
2.2	19 Jan 2010	Final	K. Yskout (KUL)	Implemented final quality check remarks

Executive summary

WP2 is both concerned with security engineering processes and security architectures for evolving systems. In this deliverable we present **three novel paradigms** which contribute to achieve an integrated framework for management and design of long living security-critical systems.

The **SecureChange security engineering process** (Section 3) is revolutionary in the respect that it is fully change driven. The view of existing security engineering processes as sequences of actions (e.g. risk analysis and requirements elicitation) performed on the whole system has been replaced by the view of change events causing change propagation and state changes in the security engineering artefacts. This change of paradigm provides for the first time a systematic way of handling changes based on dependencies between artefacts. Beyond that the SecureChange process incorporates concepts for the collaboration of different stakeholders in security engineering, ranging from the IT manager and requirements engineer to the security architect and system administrator. The goal of this collaborative approach is to support continuous security management and to achieve an adequate level of security at any time in the software lifecycle.

The SecureChange process is generic in the respect that it is independent of artefacts (e.g. Risk Model, Requirements Model, etc.). As a reference model we present a general meta model of artefacts and their dependencies. At the same time we elaborated a first version of a meta model integrating the artefact structure and change perspectives of the whole project. While development of rigorous tool support of the process will be launched in the second year, one of the partners (THA) already started to materialise the SecureChange process in its existing tool environment.

While the SecureChange process is independent of artefacts and kinds of changes, the concept of **Change Patterns** (Section 4) provides guidance for the architectural changes within the process. Thus, change patterns build the bridge between the security engineering process and security at architectural and design level.

A change pattern guides the architect in designing an architecture that is resistant against certain foreseen evolutions of the requirements and assumptions. A change pattern explicitly records the change of requirements it supports. Applying a change pattern then consists of two steps. First, preparing the architecture 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. A catalogue of change patterns for changing trust relationships is described, and its use is illustrated.

Change patterns provide guidance for architectural changes. Additionally, it is important to have a generic blueprint of an architecture that is designed to accommodate a broad set of changes, and that can serve as a starting point for applying change patterns. This research question has been addressed by the **Security as a Service Architecture** (SEASS) approach (Section 5). Our goal has been to develop an architectural blueprint for a pluggable security architecture which supports evolution by applying similar mechanisms that have been shown fruitful in the functional parts of architectures (e.g. separation of abstraction layers, model-based configuration, and orchestration of services).

The partners of WP2 have been involved in the conceptual design of the SecureChange case studies ATM and HOMES. Section 2.3.2, 4.6.2 and 5.7 summarise the results.

Index

1 Introduction

Engineering a secure software system is hard. Ensuring that the system remains secure throughout its lifetime within a dynamically evolving environment is even harder. This can partly be attributed to the unpredictability of the world in which the system is functioning. Additionally, however, security is a discipline that involves multiple stakeholders, which have to collaborate as a well-oiled machine. Not all of these people are security experts, or even software engineers. For instance, a change in the legal or business context can have a negative impact of the security of the system, if that change is not adequately dealt with.

In this deliverable, the impact of change on developing secure software will be studied. First, this will be done from a broad viewpoint, i.e., the development process. Then, the focus will be placed on a specific phase from the development process: architectural design.

1.1 Process

Traditional secure software development processes focus on activities that have to be performed, described in a step-by-step process guide. When an application evolves, the activities from such development process need to be re-executed, requiring assistance of all stakeholders. This can be inefficient, especially if changes occur often. Therefore, traditional secure software development processes are less suitable for lifelong adaptable, secure systems, because the impact of multiple occurring changes can be (too) large.

In this deliverable, an alternative for an activity-centric process is proposed. The alternative is described as change driven process. For this process, the system is modelled as a set of tightly coupled artefacts. Each artefact contributes to a certain viewpoint and a certain level of abstraction. Evolution is now characterized by changes in the artefacts. Since the artefacts are tightly coupled, a change in one artefact can propagate to other artefacts. If a stakeholder, depending on his viewpoint, is associated with a set of artefacts, he can be notified when a change occurs for which he needs to take action. In Section 3.1 we describe a concrete process which is Work Package 2 specific.

To support this process, a model of the artefacts needs to be available. This model is described by a metamodel of an integrated process which provides an abstract description of change and change propagation overarching all the solutions provided by the various SecureChange work packages. The integrated SecureChange process is still work in progress, but the final goal is to provide Work Package independent concepts of change and to outline how different software engineering artefacts relate to each other. The dependency relations between these different artefacts provide the means to propagate change and describe different handling of different classes of change. In Section 3.2 a strategy for the development of an abstract integrated process is described which provides an integration of the solutions of the different SecureChange Work Packages.

1.2 Architecture

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 first of all possible because of problems or constraints that only arise in the implementation or deployment phase 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 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. 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.

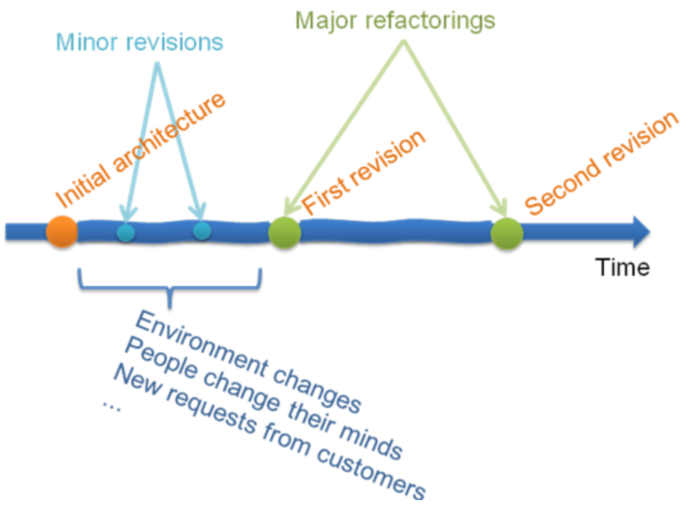


Figure 1 Changes in an application

A change can impact the architecture to different degrees, as illustrated by Table 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. 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.

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 Possible impact on the architecture of a change

A change can have impact in two dimensions. First, the impact of the change during the *development* (design) of the architecture. That is, given that the architect wants to support a given evolution scenario, what is 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. Note 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 it in the future. 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 support 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).

1.3 Outline of the deliverable

This deliverable concerns both security engineering processes and security architectures for evolving systems. After presenting related work (Section 2), three paradigms are presented which contribute to achieve an integrated framework for management and design of long living security-critical systems.

The **SecureChange security engineering process** (Section 3) is a fully change driven process. This provides for the first time a systematic way of handling changes based on dependencies between artefacts. Beyond that, the SecureChange process incorporates concepts for the collaboration of different stakeholders in security engineering. The goal of this collaborative approach is to support continuous security management and to achieve an adequate level of security at any time in the software lifecycle.

While the SecureChange process is independent of artefacts and kinds of changes, the concept of **Change Patterns** (Section 4) provides specific guidance for the software architect who is using the process. Thus, change patterns build the bridge between the security engineering process from Section 3 and security at architectural and design level. A change pattern guides the architect in designing an architecture that is resistant against certain foreseen evolutions of the requirements and assumptions. A catalogue of change patterns for changing trust relationships is described, and its use is illustrated.

Change patterns provide guidance for architectural changes, but require that an initial architecture is available. As such, it is also important to have a generic blueprint of an architecture that is designed to accommodate a broad set of changes, and that can serve as a starting point for applying these change patterns. This research question has been addressed by the **Security as a Service Architecture** (SEASS) approach (Section 5). Our goal has been to develop an architectural blueprint for a pluggable security architecture which supports evolution by applying similar mechanisms that have been shown fruitful in the functional parts of architectures (e.g. separation of abstraction layers, model-based configuration, and orchestration of services).

2 Related work

2.1 Security engineering processes

There are several processes for secure software development in the field. Based on a survey on security engineering processes [1], we present three of the best known processes: OWASP´s CLASP [2], Microsoft´s SDL [3] and McGraw´s Touchpoints [4]. All of them provide an extensive set of activities covering a broad spectrum of the development life-cycle.

2.1.1 CLASP

Originally defined by Secure Software and later donated to OWASP, CLASP is a lightweight process for building secure software. It includes a set of 24 top-level activities, which can be tailored to the development process in use. Key characteristics include:

Security at the center stage: The primary goal of CLASP is to support the construction of software in which security takes a central role. Furthermore, the activities of CLASP are defined and conceived primarily from a security-theoretical perspective and, hence, the coverage of the set of activities is fairly broad.

Limited structure: CLASP is defined as a set of independent activities that have to be integrated in the development process and its operating environment. The choice of the activities to be executed and the order of execution is left open for the sake of flexibility. Moreover, the execution frequency of activities is specified per individual activity and, hence, the coordination and synchronization of activities is not straightforward. Two road maps ('legacy' and 'greenfield') have been defined to give some guidance on how to combine the activities into a coherent and ordered set.

Role-based: CLASP defines the roles that can have an impact on the security posture of the software product and assigns activities to these roles. Roles are responsible for the finalization and the quality of the results of an activity. As such, roles are used as an additional perspective to structure the set of activities.

Rich in resources: CLASP provides an extensive set of security resources that facilitate and support the implementation of the activities. For instance, one of these resources is a list of 104 known security vulnerabilities in application source code (e.g., to be used as a checklist during code reviews).

2.1.2 SDL

As a result of its commitment to trustworthy computing proclaimed in 2002, Microsoft defined the SDL to address the security issues they frequently faced in their products. SDL comprises a set of activities, which complement Microsoft's development process and which are particularly aimed at addressing security issues. SDL can be characterized as follows:

Security as a supporting quality: The primary goal of SDL is to increase the quality of functionality-driven software by improving its security posture. Security activities are most often related to functionality-based construction activities. For instance, threat modeling starts from architectural dependencies with external systems, while an architecture could in fact reduce such threats in the first place. SDL is designed as an add-on to the software construction process.

Well-defined process: The SDL process is well organized and related activities are grouped in stages. Although these stages are security specific, it is straightforward to map them to standard software development phases. Furthermore, several activities have a continuous characteristic in the SDL process, including threat modeling and education. As such, the SDL process incorporates support for revising and improving intermediate results.

Good guidance: SDL does a good job at specifying the method that must be used to execute activities, which, on average, are concrete and often somewhat pragmatic. For instance, attack surface reduction is guided by a flow chart and threat modeling is described as a set of sub-processes. As a result, the execution of an activity is quite achievable, even for less experienced people.

Management perspective: SDL takes a management perspective for the elicitation and description of many activities. This is nice, given the inherent complexity of security, and it shows that security as a quality has to be managed in order to be realized in practice.

2.1.3 Touchpoints

Touchpoints provides a set of best practices that have been distilled over the years out of the extensive industrial experience of its proposer. Most of the best practices, named activities from here on, are grouped together in seven so-called touch points. Touchpoints can be characterized as follows:

Risk Management: Touchpoints acknowledges the importance of risk management when it comes to software security. It tries to bridge the gap by elaborating a Risk Management Framework (RMF) that supports the Touchpoints activities.

Black vs. White: The touch points provide a mix of black-hat and white-hat activities, both of which are necessary to come to effective results. Black-hat activities are about attacks, exploits and breaking software (e.g., penetration testing). White-hat activities are more constructive in nature and cover design, controls and functionality (e.g., code review).

Flexibility: The touch points can be tailored to the software development process already in use. To facilitate this, the documentation provides a prioritization of the different touch points. This allows companies to gradually introduce the touch points, starting from the most important ones.

Examples: Touchpoints is rich on examples. For instance, when describing abuse cases, there is an example giving the reader a good feel about what they might look like in a particular situation.

Resources: To further aid the execution of activities, Touchpoints provides links to resources and also explains how to use them. To this aim, a part of the book is dedicated to security knowledge (which the resources are part of). For instance, attack patterns are provided in order to be used in the elicitation of abuse cases.

2.1.4 Process support for change

The general characteristics of the three processes described before are summarized in the top part of Table 2. We will now discuss how these processes deal with evolution.

	CLASP	SDL	Touchpoints
General			
Focus	Security at center stage	Security as supporting quality	Risk management
Structure	Limited (independent activities)	Well-organized set of activities	Grouped activities (best practices)
Guidance	Rich set of resources	Concrete activities	Rich examples and resources
Evolution			
New security vulnerability	Software updates Security advisories	Software updates Security advisories	*Not supported*
Change in security assumptions	*Not explicitly supported*	*Some continuous activities (e.g., threat modeling)*	*Not explicitly supported*

Table 2 Comparison of CLASP, SDL and Touchpoints

As we know, security is a moving target. Applications change, executing environments change and attackers change. Thus the process should include continuous support to address new security vulnerabilities during the lifetime of an application, under the assumption that previously articulated security assumptions remain valid. This is supported in CLASP and SDL by including activities that focus on software updates and security advisories. Touchpoints does not seem to cover this.

Second, and more challenging, when intermediate results turn out to be incorrect (such as an incomplete threat model), or when security assumptions change after deployment, the process must be backtracked in order to correct the no-longer valid decisions and assumptions. In a process, backtracking can be supported by introducing iterative cycles, or by inserting dedicated checkpoints and feedback loops. This kind of support is limited in the mentioned processes. At least, this would require the explicit documentation of the dependencies between the various activities and their outcome, which none of the processes provide.

Concluding, all the mentioned processes provide a set of actions (e.g., requirements elicitation and risk analysis) on the whole system. However, there is no explicit support for evolution in these processes. The process outlined in Section 3 replaces the view of existing security engineering processes as sequences of actions (e.g. risk analysis and requirements elicitation) performed on the whole system by the view of change events causing change propagation and state changes in the security engineering artefacts. This provides a systematic way of handling changes based on dependencies between artefacts.

2.2 Software architecture and security

2.2.1 Software architecture

In [5], 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 [6], 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 [7]: "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 [6], 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.

The definitions of an architecture introduced above are quite generic, and only talk about 'elements' (or 'components'). There are many different ways to interpret an element, however, which can be captured in the notion of an architectural profile.

A profile defines the vocabulary and rules that can be used to define an architecture using that profile. For instance, a component-oriented profile defines components and connectors, and states that components can only communicate through connectors. A service-oriented profile on the other hand defines services, participants and workflows. An aspect-oriented profile would define architectural elements such as aspectual components and join points. A component-oriented architecture usually suffers from more coupling than a service-oriented architecture, which is loosely coupled by definition. Similarly, an aspect-oriented architecture is designed to reduce coupling even more.

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.

2.2.2 Architecture and security

Security at the architectural level can be looked at from different viewpoints. In this deliverable, 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 [8, 9] are used to this aim. The NFR framework also uses patterns to create secure designs [10, 11]. In [12], 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, in [13] an attack surface metric is proposed, which can be used to measure the security of a design and improve it. Equally, the principle of least privilege can be used as a guidance for improving the security of architectures [14].

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 [15]. Also, UML can be used or extended to represent security properties, as is done in UMLsec [16] and SecureUML [17].

Finally, it can be investigated how an architectural description can be used to perform a security **analysis**. The STRIDE [18] 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 [19].

2.3 Case studies

The concepts in this deliverable will be validated in the second year, by applying them to two case studies from the SecureChange project: the HOMES case study and the ATM case study.

2.3.1 HOMES

Taking the description of the HOMES case study in the document "Description of the Scenarios and their Requirements (D1.1)" as a frame of reference, we are interested in architectural changes to the Home Gateway – a critical component for the enforcement of security policies in the NAC architecture.

More specifically, we plan to investigate two scenarios in the second year of the project, based on the results in this deliverable. In the first scenario, we will focus on the evolution of trust relationships that are present in the HOMES case study (for example, examining the implications of moving the policy decision point (PDP) from the provider side to the Home Gateway). More details are given in Section 4.6.2. The second scenario is the enforcement of a new security requirement (taking non repudiation as an illustrative example) through the deployment and configuration of a new security service onto components (e.g., Home Gateway) of the service oriented infrastructure. Further information on this scenario can be found in Section 5.7.

2.3.2 ATM

To highlight the SecureChange process, we elaborate a concrete example based on a specific change scenario. In concrete we will treat the introduction of a new service in the architecture, the AMAN (Arrival Manager). The introduction of this new service triggers a range of additional changes. Among the most important ones are changes in the architecture, changes in the work procedures and the introduction of new threats and hazards. The scenario is sketched in Section 3.1.6.

Special emphasis of the application of the SecureChange process to the ATM case study will be put on the handling of changes. We will focus on how the states of various model elements are updated following specific activities. Particular focus will be put on the concept of change propagation which is based on state change and the dependencies between different artefacts.

The result of this work will be a complete walkthrough through the SecureChange process, providing concrete examples and instantiations for all the concepts described in Section 3.1.

3 Change-driven security engineering

In this Section we describe the SecureChange security engineering process which is fully change driven. The process is change driven in the sense that every change triggers state changes in the related security engineering artefacts. The dependencies between the various security engineering artefacts provide a frame for change propagation. In addition the SecureChange security engineering process supports the collaboration among different stakeholders which have their specific views on a system. Using the concept of change propagation based on the dependencies between artefacts it is possible to support also the collaboration of different stakeholders which can be notified of change events which impact their specific view.

This section is structured as follows: Section 3.1 describes the SecureChange security engineering process, beginning with a short introduction to the problem. After a classification of different types of changes in the context of security we discuss the requirements for a security engineering framework which supports change. Following the requirement we describe our vision of “Living Security” – a framework supporting a process of secure change. After a description of the core concepts of “Living Security” we outline how the initially mentioned requirements are addressed by this framework.

In Section 3.2 we describe an integrated process which is abstract and generic and encompasses and relates all the solutions provided by the different Work Packages. The integrated process is an abstraction of the SecureChange security engineering process described in Section 3.1 and is independent of any specific artifact. We describe the overall strategy for integrating the different Work Package specific solutions and their artifacts and our strategy for developing a generic description of change and change handling.

Section 3.3 describes the security analysis method of one of the partners of SecureChange (THA), which already started to materialize the SecureChange Process in its tool environment. The Section starts with an outline of the principles of the security analysis method and is followed by an overview of the Security DSML. In the next subsections the Change Model used by THA and the behavior of a Change Request is presented.

3.1 Process

3.1.1 Introduction

The engineering and management of security-critical systems imposes evident requirements onto procedures applied and tools used. The management of information security needs to be effective – producing the desired effect of protection – and efficient – producing effectively with a minimum of effort. But, in an ever-changing world, socio-organizational as well as IT systems become a moving target. They constantly evolve, adapting to changes in their environment. In order to meet the two basic requirements of information security management in the context of evolving security-critical systems we need to rise to three basic challenges.

Relevance. First and foremost we need to identify the kind of events that indicate relevant changes. Change is generally captured indirectly through an event's impact on a system. This can e.g., be expressed as the deviation of the current status from some targeted status. Thus, change is relevant if the event alters (for better or for worse) the capability of some socio-organizational and/or IT systems to work as supposed – which means to comply with functional and non-functional requirements. In security, any deviation from a target may be seen as a change for the worse.

Abstraction. Second, once indicated, change triggers effective action only if it is interpreted appropriately. This means that information about the event has to be meaningful to the person holding a specific role (e.g., CSO, network administrator, software engineer etc.). The person should understand the consequences so that she can carry out necessary action according to her responsibilities (e.g., set up security policy, deploy component, configure network etc.). This entails the need for an appropriate conceptualization and visualization of the event and its impact on the system from a certain angle. This perspective should open a view on information about a system's changing state at an appropriate level of abstraction.

Propagation. Third, change may "materialize" in any of the stakeholders' perspective and percolate to other perspectives, possibly affecting various levels of abstraction each time calling for some action to be taken. As an example we may take a business analyst specifying a new security requirement for a business process in the *IT Management view.* The requirement is translated into a non-functional requirement of the requirements model in the software engineer's view. The engineer can then model and trace the dependencies between the requirement, the component in the software architecture enforcing the requirement and the actual code through the respective layers of his view. The interrelationship between sub-systems may allow change to propagate in unforeseen ways. Thus, to take effective actions, stakeholders need to consider the whole system including IT as well as socio-organizational aspects.

We propose Living Security a framework for the model based development, management and operation of security critical, evolving service oriented systems. The main idea is to facilitate the cooperation of stakeholders in IT management, software engineering and systems operation. The framework links an abstract, but coherent view of a complex system's security status integrating the perspectives of all stakeholders to the running IT-system and implemented organizational procedures.

The link between the models and the technical and organizational controls facilitates a flow of information in both directions. Models and artefacts evolve together. Change thus becomes a first-class citizen in a security process linking security engineering, information security management and risk management.

3.1.2 Change in the context of security

We develop our understanding of relevant change based on the two dimensions as elaborated in [26]. Hence, we focus on two aspects, for one the particular nature of change and, two, its origin.

	Proactive	Reactive
Revolution	*Planned Revolution*	*Imposed Revolution*
Evolution	*Planned Evolution*	*Imposed Evolution*

Figure 2 Four basic types of change

Generally speaking, change can manifest itself as either a gradual evolution of a system in smaller steps or as a radical “revolution” with major impact on the system’s structure and functionality. So, change either comes in the shape of Evolution or Revolution.

As to the origin of change, it is important to know whether change was planned or imposed by the outside. In the first case, a Proactive approach leads to planned change. In case change is imposed, it can only by reacted upon, thus resulting in a Reactive approach.

Classifying change types along these two dimensions, we identify four basic types of change (cf. Figure 2).

Security requirements engineering (as described in [21]) views (r)evolution as emanating from a change in a system’s requirements, specification, and/or context. Living Security follows this engineering approach for supporting the management of the effects of change. This means realigning evolving systems to existing security requirements or to adapt the systems to changing security requirements or context. The handling of each of the four basic types of change is exemplified in a general use case in the context of security management and engineering.

A. Enforcing Security - Planning Small System Changes

In most cases, a system evolves over time according to a specific plan so to meet a specific set of requirements. For example, the anti-virus software component of a network needs a weekly software update. Thus, we anticipate small changes in the system’s context (new virus threats) by planning gradual changes (weekly updates) to enforce a security policy (the requirement of integrity). The framework keeps the network administrator informed about the “effectiveness” of the security measure based system’s current status, whereas IT management can judge on the measure’s “efficiency” by tracing the dependency between the security measure and its contribution to meet business requirements or – thinking inversely – the impact of its failure.

B. Monitoring Change - Reacting to Minor Changes in Context or Requirements

Often the triggers are small changes in the system’s context or in the requirements specification. In other terms, change occurs in the shape of unplanned events. If the anti-virus component’s weekly update failed, we would face change (lower security) imposed by an unplanned event (failed update). To react appropriately, all stakeholders would have to understand and evaluate the event’s impact onto their domains and take coordinated measures. The latter means that various stakeholders must always bear their activities’ potential impact on other domains in mind. Living Security visualizes the events in the specific views and facilitates coordinated measures by tracing dependencies across views and layers.

C. Scenario Planning - Planning Major System Changes

A system may occasionally have to undergo a major change. For example the merger of two companies may require the integration of two IT systems. In that case, the stakeholders need to plan and anticipate a series of changes to an already existing system. They need to understand the impact of these changes on the systems security. In this respect, Living Security contributes to a clear understanding of the system’s security status “as-is” and facilitates the analysis of security challenges in relation to the various alternatives.

D. Realizing Change - Reacting to Major Changes in Context or Requirements

Living Security would not be able to cope in a reasonable way with large unforeseen change fundamentally impacting a system’s structure or functionality. So we consider this use case as being beyond scope.

3.1.3 Requirements

After the definition of useful categories of change in context of security management and engineering we move on to specify the requirements for a framework supporting secure change. We illustrate these requirements with a running example for an evolving, security-critical large-scale system. The same example will be used in the description of the concepts which realize the framework.

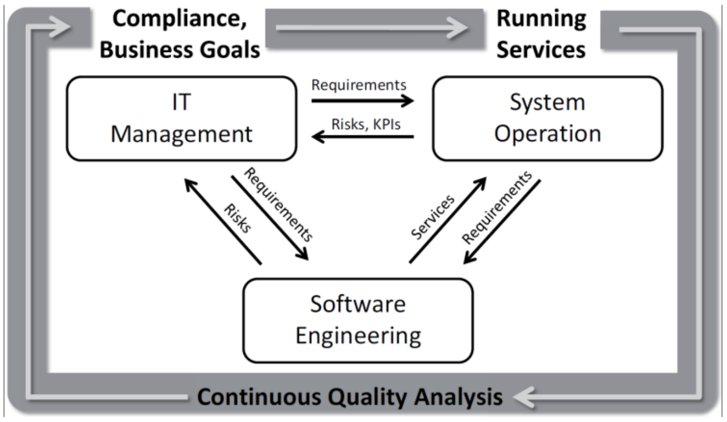


Figure 3 Integrated view

Example 1: The example used in this deliverable is a financial trading platform which allows traders to place orders in specific market segments. The trading platform is directly connected with the systems of major financial institutions who use the platform to place large volume orders. In addition, individual traders can use an online frontend or download a client to use the services. The trading platform is developed and operated by a medium sized specialized company which offers services and support for the platform and sells licenses to its users. The trading platform is developed in house by a team of software developers who develop, deploy and actively manage the systems which are hosted in an outsourced data center. The trading platform uses standardized financial communication protocols and is realized as a SOA. The company is certified according to ISO 27001 to underline their emphasis on security. Currently, the company offers access to specialized niche markets, but is planning to extend its service to stock markets in the near future.

3.1.3.1 R1: Integrated view

To keep a complex and interconnected system running despite change a variety of stakeholders have to collaborate in their daily operations. To support the collaboration of these different stakeholders (e.g., Chief Information Officer, legal experts, system administrators, software developers) an integrated view on the system is required. Such an integrated view needs to incorporate aspects from different disciplines such as IT management, system operation and software engineering. While all these disciplines have their own perspective on a system and use a different level of abstraction, the challenges related to change cannot be tackled from a single perspective. Instead it is necessary to focus on a changing system from all these different angles to keep it running in a secure manner. Figure 3 shows the concepts of an integrated view.

Example 2*:* A financial trading platform has to fulfil a range of security requirements, stemming from special requests of large customers, general legal regulations for the financial services market, and the contractual obligations which arise from docking on to the financial systems of major partners and markets. The company has a Chief Information Officer (CIO) and a Chief Information Security Officer (CISO). In addition the company employs two legal experts who are aware and responsible for all the legal and compliance requirements that have to be fulfilled by the systems. The platform is constantly evolving due to the addition of new services (e.g. access to new market segments can be offered to the clients), the continuous extension of the platform (i.e. 3.000 new accounts are opened every month), and the incorporation of a range of new regulations which were enacted as a result of the financial crisis. Whenever legal requirements change, system capacity needs to be extended or a new service to be deployed in the platform, changes cannot be tackled in isolation but have an impact on many aspects of a system. Therefore it is important to understand which parts of the system are affected by a new compliance requirement, which new services might introduce unbearable risks in the already running platform and what system updates and transitions could cause an interruption of critical services. An integrated view on the systems combines all these aspects and relates these perspectives to each other. A software developer needs to be aware of those requirements which have to be considered in the design and the adaptation of a service. Similarly, a system operator needs to know what the capacity requirements of soon-to-be deployed services may be. An integrated view is therefore an essential instrument to provide a basis for a common understanding of the systems and its dependencies and serves as a platform for communication among all the different stakeholders.

3.1.3.2 R2: Domains and Responsibilities

A framework which supports collaboration and cooperation among different stakeholders needs to take into account the different perspectives and responsibilities of stakeholders. Not every stakeholders needs to understand and analyze every aspect of a system. Instead each stakeholder is equipped with knowledge and experience for a specific domain of the entire system. While it is important to support cooperation among these stakeholders it has to be taken into account that each stakeholder has a tailored view on a system and its security.

A stakeholder carries responsibility for a specific domain, i.e. a subset of the constituting elements of a system. It is important to have a clear understanding of these governance aspects to be able to handle change effectively in an organization. Depending on the type and extent of change, certain stakeholders need to cooperate to provide solutions to handle such change.

Example 3*:* In the example of the trading platform a domain could be the technical infrastructure (physical layer) which is overseen by a senior system and network administrator. The set of all nodes and locations for which he is responsible represents a domain. In a similar way legal experts are responsible for their own domain which could be the set of all compliance requirements defined on the business layer.

Similarly the responsibilities of these stakeholders are differing with respect to handling change. The legal experts are the ones who are informing other stakeholders about new requirements and the available time frame for reaching compliance. The software developers together with the system operators are proposing different solutions for reaching these new compliance requirements and IT management will have to decide together with software architects which solution fits best with the organizations business and technology strategy.

3.1.3.3 R3: Change Propagation

A framework which supports a change-driven security process needs to provide a foundation for propagating change to the right stakeholders in an organization. Change is perceived as an event which triggers a series of consecutive steps. As already outlined in Section 3.1.2 there are different types of changes in a security context.

Change propagation includes on the one hand the registration of relevant change events. On the other hand it requires a systematic way of identifying parts of the system and stakeholders affected by change.

Example 4: In the concrete example of the financial trading platform it is not sufficient to have an integrated view on the system and a clear attribution of responsibilities and domains to various stakeholders, but it is a central requirement to propagate change based on these two foundations. The Living Security framework needs to provide support in the process of identifying what parts of the system are impacted and who needs to be informed, who is consulted, and who has to make decisions in response to a change event.

Continuing the concrete example from above, consider a new legal requirement has been identified by the legal experts and has been attributed to the business process "Place orders". Change propagation includes identifying which parts of the system are related to the specific business process, who the responsible domain owners are, and the steps they need to take to respond properly to the event. The business process "Place orders" processes the two information objects "Order details" and "Account information" and calls two services in the infrastructure, namely "Order authorization" and "Transaction execution". The responsible domain owners are the information architect and the software developer. Both are responsible for maintaining the two services.

In order to be able to identify the interrelationships between the parts of a system and their respective owners, a systematic analysis of the model depicting these interdependencies and responsibilities is required. In addition, a tool-based framework needs to provide communication means for informing the various stakeholders about relevant change events.

3.1.3.4 R4: Bidirectional Flow of Information between Models and Executing System

The bidirectional flow of information from models to executing system and vice versa is essential to ensure the efficient and effective management and engineering of an evolving security-critical system.

On the one hand, the target architecture of a Living Security framework is a security infrastructure equipped with sensors collecting information and feeding it back into the modeling environment. Once there, information is interpreted at the level of Model Elements in context of the System Model. On the other hand, the security infrastructure ought to be configurable from a modelling perspective.

Example 5*:* In the example of the financial trading platform the bidirectional flow of information between models and executing system can be describe using the following two scenarios:

a) Information flow between executing system and models: On the infrastructural and service layer of the system, various sensors which constantly collect information regarding the status of nodes and components in the network can be deployed. Relevant information could be captured through traditional metrics like capacity utilization, throughput, and number of handled orders to name just a few. Using such indicators and putting them in relation with security objectives and requirements extends the integrated view on a system with meaningful key indicators. Consider as an example a service level which is guaranteed to the premium customers of the trading platform. The requirement “Maintain 99.999% uptime per month for the premium services” can be extended with key figures collected from the executing system, which constantly monitor the uptime of these key services.

b) Information flow between models and executing system: If the services are implemented and deployed in a specific target security architecture it is possible to configure the security properties of the system using models. As an example, consider the communication in the business process “Place Order”. Calling the respective services for authorization and transaction execution is designed to be based on using encrypted and authenticated messages. A specific premium customer might require an additional electronic signature by the financial service provider confirming specific details of an order to comply with additional audit requirements. If the service is deployed in a specific target security architecture it is possible to configure such security services using models.

3.1.3.5 R5: Information Consistency and Retrieval

Stakeholders need explicit support to appropriately visualize and query security related information in its various contexts. They need to understand the connection between the various levels of abstraction. This requires appropriate mechanisms that guarantee the consistency of all the information in the model. Consistency can be provided by defining and checking certain constraints in the model of a system.

Information Retrieval goes beyond the mere checking of constraints on model elements in that stakeholders can access semantically enriched information. Depending on his or her background and responsibilities, a stakeholder may require a specific view on a system in a specific representation. While some of the stakeholders might favour a traditional spreadsheet to represent specific information about the system, others might require graphical representations like process maps, or a dependency matrix outlining the relations between services and infrastructural components. Depending on the information a specific stakeholder request the models and specific views can be enriched with additional information which is either collected from sensors in the infrastructure or based on an analysis of the model itself.

Example 6: For example, a CSO would like to check whether every security threat at the technical level is related with some security threat at business level (describing the business impact of the technical threat). Such types of analysis can be run in a model using specific queries and checks. The resulting information can be represented in different ways, e.g. either using a simple table or in a graphical diagram.

Similarly, the security engineer would like to check whether each security requirement is complemented by an appropriate security service at the architecture level. The security engineer might in turn favour a network diagram or another graphical representation outlining which security requirements are not yet complemented by technical security solutions.

3.1.4 The Secure Change framework

Here, we describe our vision of “Living Security” – a framework supporting a process of secure change. We describe the core concepts of Living Security and outline how the before mentioned requirements can be addressed.

3.1.4.1 Common System View

The framework supports stakeholders in their various daily operations. This happens through Stakeholder-Centric Modeling Environments, perspectives on the system’s security status, customized to an appropriate level of abstraction. The analysis of security attributes requires the analysis of interdependencies across the layers ranging from IT management, software engineering and system management. Although the framework also facilitates the cooperation among the stakeholders (Chief Information Officer, Chief Security Officer, Network Administrator, Security Engineer etc.), it does not necessarily need to provide an integrated and homogeneous modeling environment. Rather, these stakeholdercentric modelling environments, rely on a common metamodel, the Common System View.

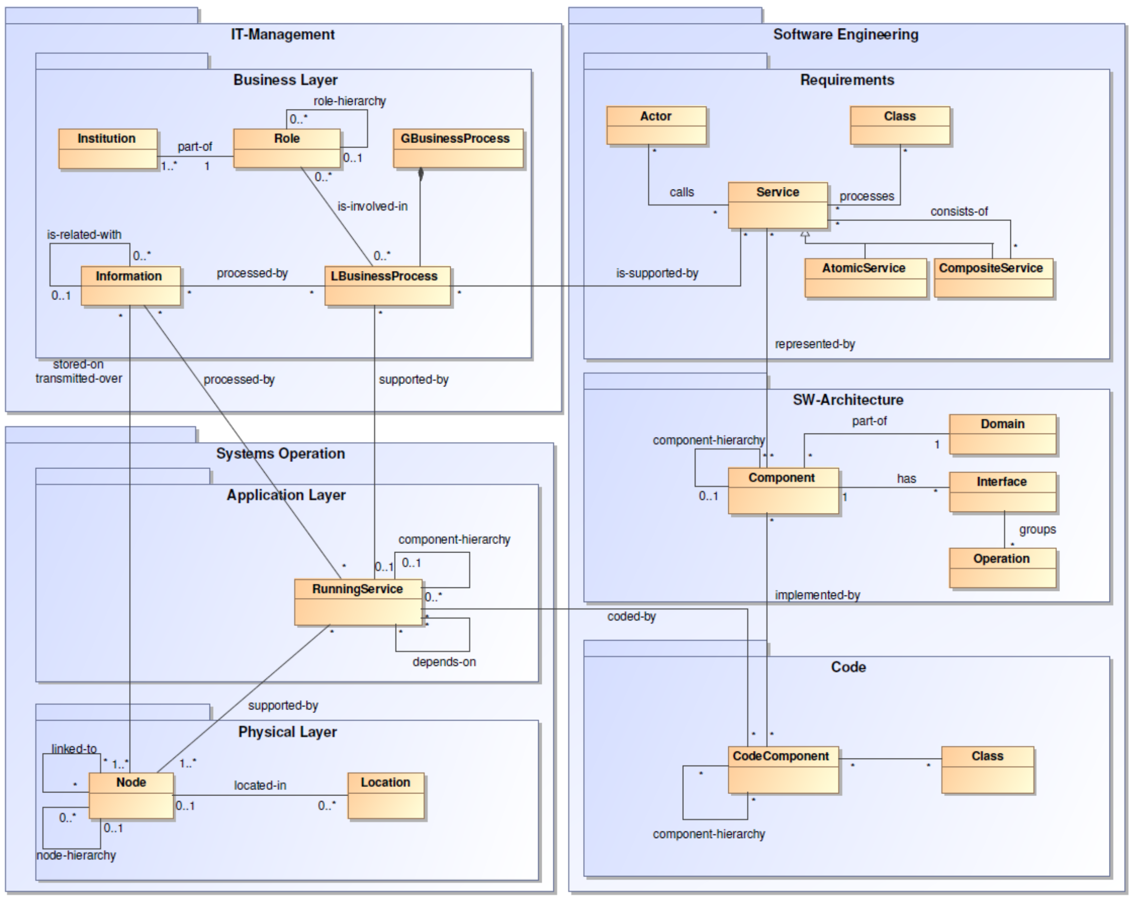


Figure 4 Sample functional system meta model

The Common System View represents the conceptual underpinning for the security management process. Its elements are the conceptual units subject to change. Dependencies between the various concepts are modeled as associations between elements allowing change to percolate through the Model Layers.

Functional system concepts like business processes, information objects, roles, components etc. are defined in the Functional System Meta Model logically grouped into the various domains of the Modeling Environments. The latter can be composed of Model Layers each one capturing another level of abstraction or degree of granularity. Figure 4 exemplarily shows a sample Functional System Meta Model and the three Layers of the Modeling Environment Software Engineering, namely *Requirements*, *SW-Architecture*, and *Code*.

Security related concepts like threats, risks, requirements etc. are introduced into the meta-model as Meta Model Plug-Ins. Every element of the System Meta Model can be decorated with security-related semantics. Figure 5 shows a sample security meta model that plugs into the Functional System Meta Model as an extension for security. Here we assume that each model element in the Sample system meta model (cf. Figure 4) inherits from the generic class **ModelElement**.

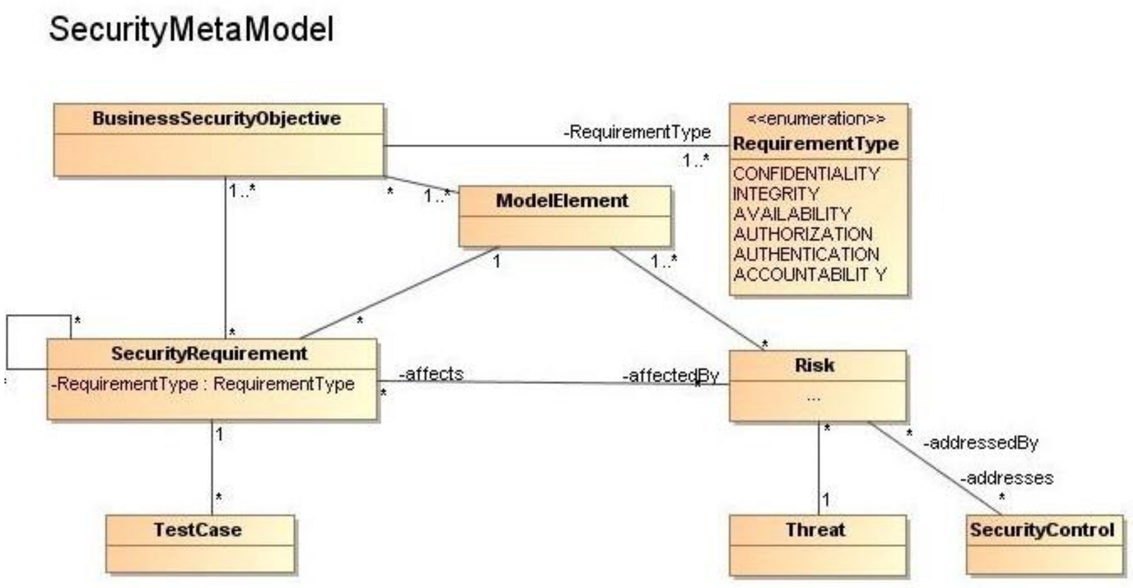
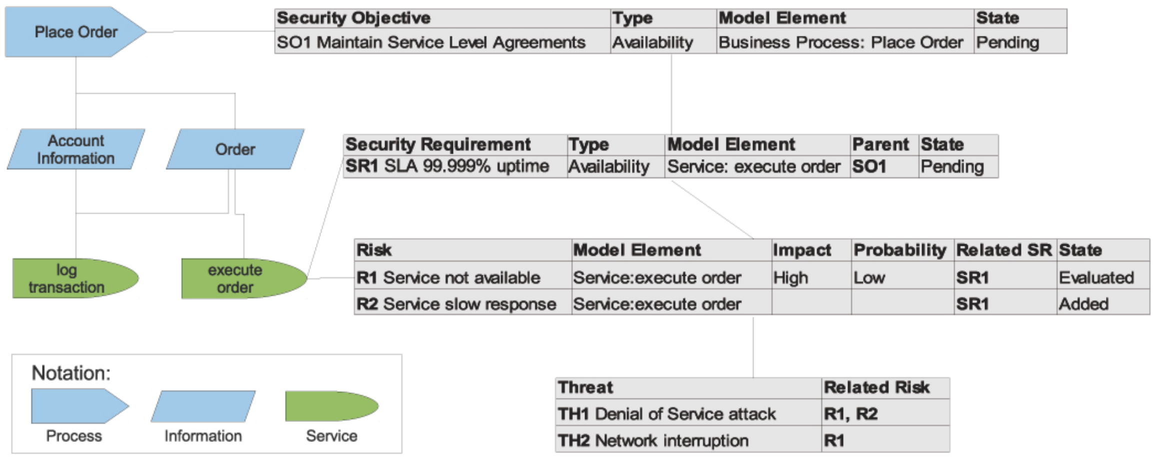


Figure 5 Security plug-In

Example 7: Consider as an example the business process “Execute order” which requires many different running services in the trading platform, which in turn require system and network capacities. If a legal expert identifies a new requirement (e.g. new contractual obligations related to premium customers) which is valid for the business process “Execute order”, then she links this requirement to the concept business process. In the example in Figure 6 the objective is “Maintain Service Level Agreements”. The dependencies and relation between the different layers serve as a means to identify which parts of the infrastructure are impacted by such a new requirement (e.g. the service “execute order” requires an uptime of 99.999%’).



Security Objective	Type	Model Element	State
SO1 Maintain Service Level Agreements	Availability	Business Process: Place Order	Pending

Security Requirement	Type	Model Element	Parent	State
SR1 SLA 99.999% uptime	Availability	Service: execute order	SO1	Pending

Risk	Model Element	Impact	Probability	Related SR	State
R1 Service not available	Service:execute order	High	Low	SR1	Evaluated
R2 Service slow response	Service:execute order			SR1	Added

Threat	Related Risk
TH1 Denial of Service attack	R1, R2
TH2 Network interruption	R1

Figure 6 Example system model extended with a security plug-in

In doing so a system operator might receive notice that certain parts of the infrastructure he is responsible for are impacted by this new requirement. Another advantage of the Common System View is the provision of stakeholder specific perspectives. In this example a software architect focuses mainly on the deployed components which realize the services and how these components communicate with each other.

The System Meta Model together with its plug-ins describes all aspects of the system across the levels of abstraction in a highly interlinked way. Additionally, we define the notion of a *View* (cf. Figure 7). A View consists of a selected set of model elements (together with selected interdependencies) and corresponds to the usual notion of a Model or an artefact in many approaches (like a Security Analysis View/Model, a Software Architecture View/Model or a Requirements View/Model). Note that different views may be related by common meta model elements or interdependencies.

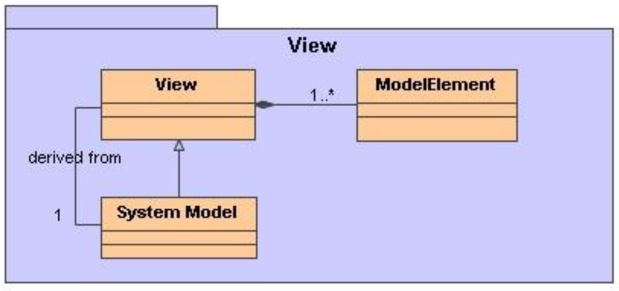


Figure 7 View meta model

3.1.4.2 Model Element States

To depict changes and distinguish different states of an information object, we want to have the possibility to model not only the dependencies between business and technical artefacts, but furthermore we want to differentiate information objects with regard to their life-cycle. We model security relevant milestones in the lifecycle of model element as Model Element States. Changes of Model Elements States can propagate over the complex “network” of model elements as defined in instances of the System Meta Model and its plug-ins.

Typical states for Security Requirements which are basically attached to Model Elements like Role or Operation are e.g., added, pending, evaluated, and implemented.

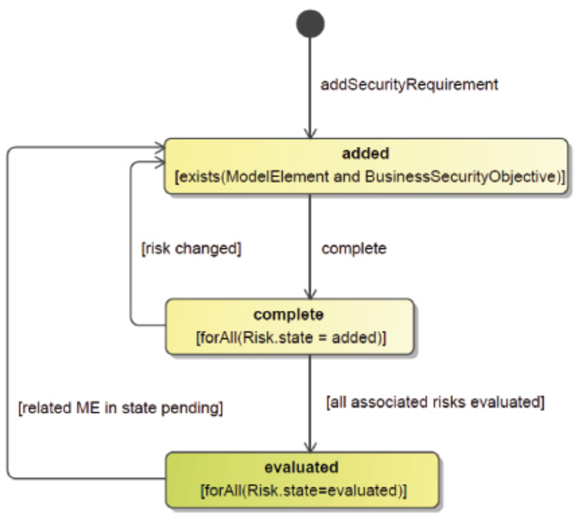

addSecurityRequirement
added
[exists(ModelElement and BusinessSecurityObjective)]
[risk changed]
complete
complete
[forAll(Risk.state = added)]
[related ME in state pending]
[all associated risks evaluated]
evaluated
[forAll(Risk.state=evaluated)]

Figure 8 State machine of meta model element security requirement

The concept is realized through (UML) state machines associated with each meta model element of the System Model and its plug-ins. As an example, Figure 8 shows the state diagram of the meta model element Security Requirement. State transitions are triggered by the following events:

- time events (e.g. triggering analysis actions to be performed periodically), ·
- conditions on the system state (e.g. the state of a security requirement is changed from *complete* to *evaluated* if all associated risks are in state *evaluated*),
- action events initiated by the stakeholders (e.g. with the action event *complete* in Figure 8, the stakeholder declares the set of associated risks to be complete) ·
- change events caused by the modification/ creation/deletion of some model element.

Example 8: In the example outlined in Figure 6 the risks related to the service "execute order" have two different states. The first risk R1 has already undergone a risk evaluation and its state is therefore set to "evaluated". The second risk R2 has only been identified but not evaluated yet, therefore its state is set to "pending".

The related security requirement SR1 will remain in the state "pending" until all related risks (R1, R2) have reached the state "evaluated". Only then the security requirement SR1 will also reach the state "evaluated".

Similarly, if the security requirements were already in the state "evaluated" and a new risk R3 was added, its state would immediately switch back to "pending", thus indicating that a change occurred and additional steps are required to reach a new security state.

In the System Meta Model we extend the class ModelElement by an association to class StateMachine (cf. Figure 9), leaving the structure of a State Machine unspecified at this place.

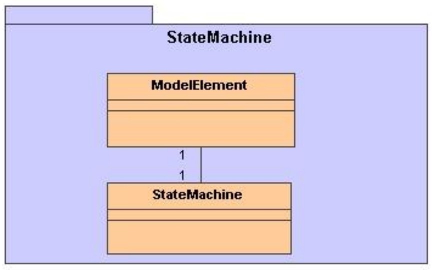


Figure 9 State machine meta model

3.1.4.3 Persistence, Tracing

As instances of the System Meta Model, System Models are the actual targets of change. They capture the status (functional and non functional) of a system at all relevant levels of abstraction. But to indicate change and facilitate planning they have to be made persistent. This allows the definition of system (r)evolution as a sequence of modeled status snapshots. Like the model elements, the System Meta Model itself may undergo change and evolve over time.

Persistence also facilitates model versioning which is a prime requirement for planning. Stakeholders can compare alternatives in terms of their impact (e.g., comparison of safeguards with respect to the trade-off risk reduction vs. costs) by creating new branches, cloning or merging System Models.

Cooperation among stakeholders is facilitated through information traceability which is conceptually anchored in the meta model of the Common System View. Depending on his Role, a stakeholder has a tailored view of a system's security status rendered in his Modelling Environment. The Role Model defines Roles – job functions of the stakeholders and their Rights – and permissions with respect to operations on model elements. The Role Model is associated to the System Meta Model.

Example 9: In the concrete example Persistence can be explained using two examples. First, consider a situation in which new security requirement has been identified as a change event and will be introduced in the common system view. The new requirement triggers a series of actions which are executed by different stakeholders. For instance, the software developer will be re-evaluating whether there are any new potential risks which might be related to the new security requirement. By keeping persistent versions of all the model snapshots, which are reflecting the ongoing security process it is on the one hand possible to provide an audit trail of the analysis and the resulting decisions. On the other hand it is possible to trace specific security solutions which are still in place back to a now possibly obsolete security requirement.

Second, Persistence allows different future scenarios to be modelled. Consider a new security requirement for which several options of security controls might be considered. Using different planning scenarios and snapshots of the model it is possible to evaluate the impact of the planned controls on the current system architecture.

Figure 10 shows a simplified way of handling versions. Every System Model (holding all information of the system) is attached at any point of time with a unique Version object. A tree structure (modelled by the **previous** association) describes the versions of a System Model along the lifetime of the system (including branches).

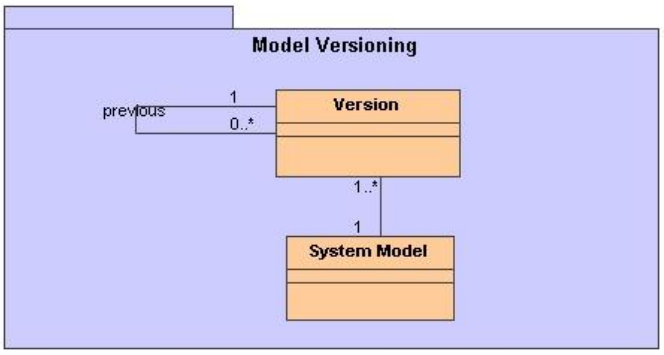


Figure 10 Model versioning

3.1.4.4 Tight Coupling of Models and Code

As the framework aims to support security engineering and management activities targeting the running system, the underlying models have to appropriately reflect the system's current security status. In Living Security, a consistent state between models and evolving system in a changing environment is maintained through the tight coupling between models and the executing system. Together with the Integrated View, this principle provides stakeholders with a Modeling Environment that is directly linked to the executing system.

In Section 5 we present an architectural framework which supports such a seamless transition from models to security architectures.

3.1.5 Meeting the requirements of a change driven security process

The requirements listed in Section 3.1.3 are tackled in our vision of a change-driven security process. The change driven security process contains the classical steps of well established security-processes [27], [28]. What distinguishes our vision of a change-driven security process is that the process steps are initiated by change requests and change events (cf. Figure 11). These change events affect the state of model elements.

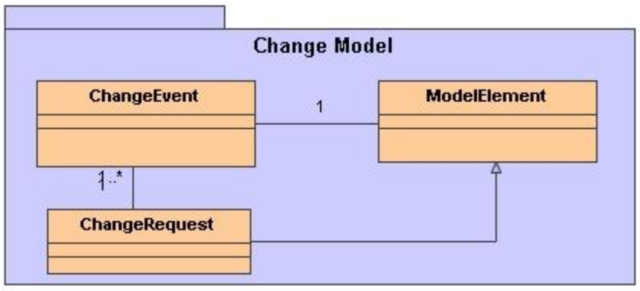


Figure 11 Change meta model

A change request is a general description of some change in the system such as *exchange component X by a technology mitigation* or *implement compliance regulation Y*. Each change request object is a ModelElement which means that change requests have a state and may have associated information (like the risk attached with the change). Each change request triggers one or more change events. Change events are directly related with model elements and cause state changes as described in the process below.

Change is propagated in the System Model based on their interrelationship with other model elements. Change events are sent to the current System Model where actions are triggered and the effects percolate through the different layers . Change is handled according to the following procedure of the Change Driven Process:

A. State transition – A change event may induce a state transition of a model element. For instance, the state of a security requirement is changed from *evaluated* to *added* if the related model element (e.g. a software component) has been modified.

B. Change propagation – The state transition of the model element may trigger state transitions in related model elements according to stated propagation rules. For instance, the modification of a security requirement attached with a business process may cause state transitions in information objects and services supporting this business process. The propagation rules are specific to each meta model element.

C. Modification of task list – Each stakeholder is associated with a task list describing the pending action events of model elements he/she is responsible for. After each state transition new tasks may be pending and have to be added to the task list. Consequently fired action events (e.g. after the evaluation of a model element) are withdrawn from the task list.

Using the concept of change events and model element states it is possible to assign and distribute the tasks of the security process to the according stakeholders. Based on the concept of domains and responsibilities we are able to identify required tasks and assign them to the respective stakeholders.

This implies a distributed security micro-process which is executed by each of the stakeholders within his specific domains. Figure 12 highlights this concept of distributed instances of a security process. Of course the stakeholders do not work independently on their security related tasks, but a lot of coordination and cooperation is necessary. Hence the first three requirements discussed in Section 3.1.3 are realized using the concepts of a common system view, and model element states.

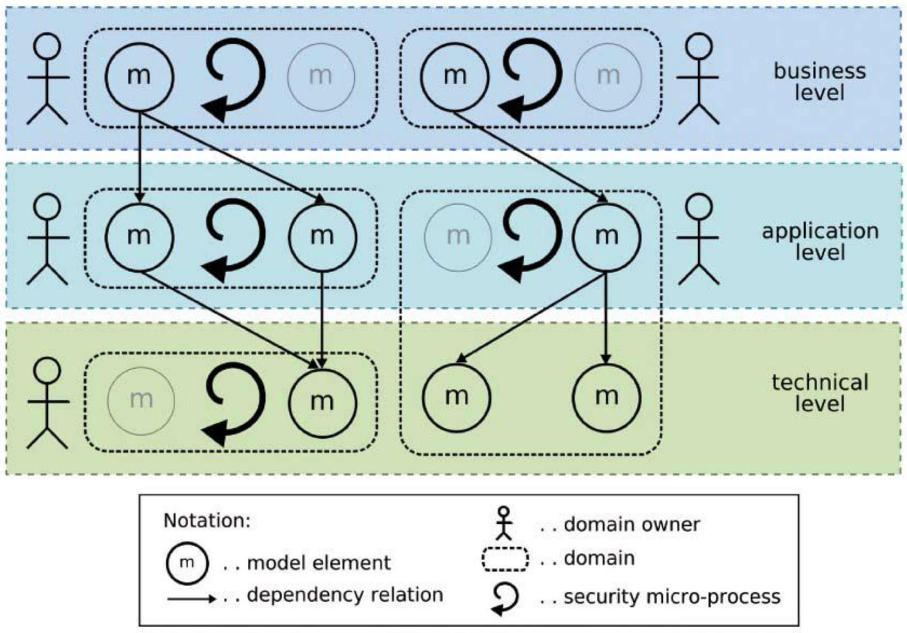


Figure 12 The concept of distributed security processes

Example 10: Using for instance the example outlined in Figure 5, the security process will be schematically described in the following.

The identification of the new security objective “Maintain Service Level Agreements” was brought up by the legal experts based on new contractual obligations with premium clients. The legal experts introduced the new security objective in the common system view using a security plug-in and attached it to the model element “place order”. Based on the dependencies of the system depicted in the common system view, this change event percolates through the processed information objects (eg. “Account information” and “Order”) to the respective services (eg. “log transaction” and “execute order”).

The software architect whose domain and responsibility contains the services and the elaboration of the related security requirements receives a notification to evaluate the existing services according to the new security objective. She or he then identifies and translates the abstract security objective in the concrete security requirement “SLA 99.999% uptime”. This event which was triggered by the introduction of the new security objective by the legal experts again triggers new actions.

In the concrete case, a security engineer whose domain consists of the threats and risks related to services receives the notification to conduct a threat and risk analysis for the service “execute order” since a new security requirement with the status “pending” has been added. The security engineer then introduces two new risks (R1, R2) which are related to the security requirement SR1.

As can be seen in the example, the progress of the steps taken by the security engineer is also reflected as a series of changes in the common system view. She or he has already evaluated the new risk R1, which state is set to “Evaluated”. As soon as the remaining risk R2 will be evaluated, the software architect will receive a notification that his or her security requirement SR1 has too reached the state “evaluated”.

In this manner it becomes possible to translate change events in a series of tasks which have to be fulfilled by different domain owners. The progress of the different distributed actions will be reflected by the model element states and allow to analyze whether or not the whole system has again reached a stable security status.

The change propagation process is related with the versioning concept of Model Versioning (cf. Figure 10) in the following way.

- The lifetime of a change request may involve many versions of the System Model. This reflects the fact that a change request may be a complex task potentially involving several stakeholders.
- Any change event (as well as the automatic change propagation process along the interdependencies between model elements) affects one specific version of System Model and causes modified model elements in the next version (e.g. state transitions or introduction of new model elements).

3.1.6 Case Study

The ATM case study provides a perfect scenario to highlight the SecureChange process as described in Section 3.1 in a concrete manner. The partners of Work Package 2 and Deep Blue have agreed to cooperate in order to build a concrete example of how change is handled in the SecureChange process. The concrete goals of the case study are:

- To outline on the basis of a specific change scenario how the SecureChange process deals with change.
- To provide a practical example of how change propagation works in the SecureChange process.

The introduction of a new service in the ATM network will provide the basic scenario on which we will elaborate the case study. The concrete component which will be treated is the AMAN (Arrival Manager). Introducing such a new service in a information network such as the SWIM architecture provides a good basis to highlight:

- How the introduction of such a new service impacts the architecture and requires changes in the system model.
- With the introduction of this new service comes also a change in the work procedures.
- How the changes in the system model require an updated security analysis of certain aspects of the architecture, as new threats and hazards are introduced.

In the current state we have collected all required information for building a model of the system. In addition Deep Blue provided us with a set of security objectives and requirements.

The following activities will be pursued in Year 2.

1. An initial version of a global and local functional model will be created based on the available documentation.
2. A description of the workflow changes which are required because of the introduction of a new service will be provided.

3. A concrete description of how the related changes are handled following the SecureChange process, with particular emphasis on the concepts of states of model elements and the concept of change propagation using dependencies between different artefacts.

3.2 Integrated Process

In order to integrate the solutions by all the different activities in the Work Packages we enhance the SecureChange process by an integrated view. This integrated view is based on the following concepts:

- Taxonomy: provides a classification of changes and attitudes to changes
- Artefacts: distinguishes on an abstract level the different models and artefacts which are used by the different Work Packages

The taxonomy can be used on the one hand to identify different basic change scenarios. These change scenarios on the other hand can be used to describe how change is handled on an abstract level by the different Work Packages.

The artefacts provide an overview of all the different types of models used by the various Work Packages of SecureChange. These artefacts are described on the meta level and abstract from concrete concepts. That way it is possible to treat method-specific conceptual models as black-boxes and plug-in different methods and approaches to the integrated SecureChange process. Examples for such artefacts are a system model which includes all artefacts related to the system, ie. architecture, code, constraints and others. Other specific artefacts are a verification model, a risk model, a requirements model and a test model.

Independently from which requirements engineering method and model is used, it is clear that a change in a requirement has to trigger some changes in the test model. Using the change scenarios derived from the taxonomy and case studies it can be described how the different artefacts are updated and trigger changes in other models. That way it is possible to outline how the results and solutions provided by one Work Package impact the other Work Packages.

3.2.1 Artefacts and relations between artefacts

As a first basis for distinguishing an abstract approach is followed. This is at the moment a non-exhaustive list which reflects the main types of artefacts which are treated by the different SecureChange Work Packages. The different artefacts are:

- System Model: The System Model includes all artefacts related to the system (from architecture to code, including constraints). It is a placeholder for the system model of Work Package 4 and all types of system models used throughout the other Work Packages.
- Verification Model: The Verification Model contains artefacts which are specific to Work Package 6.
- Risk Model: The Risk Model includes all artefacts related to risk analysis (e.g. assets, vulnerabilities, threats, controls, risk). It is a placeholder for different conceptual models of risk, such as the CORAS model, the THALES risk model

or the ProSecO security model and therefore integrates mainly the artefacts from Work Package 2 and Work Package 5.

- Requirement Model: The Requirement Model reflects all artefacts which are related to requirements engineering. It is mainly related to Work Package 3.
- Test model: The Test Model contains the artefacts related to testing, and is related to Work Package 7.

Figure 13 outlines an overview of these different types of artefacts. Seen from an overall integrative perspective the different artefacts are strongly related to each other. The result of a requirement analysis will provide input for the test-engineers and be used to verify code and infrastructure components.

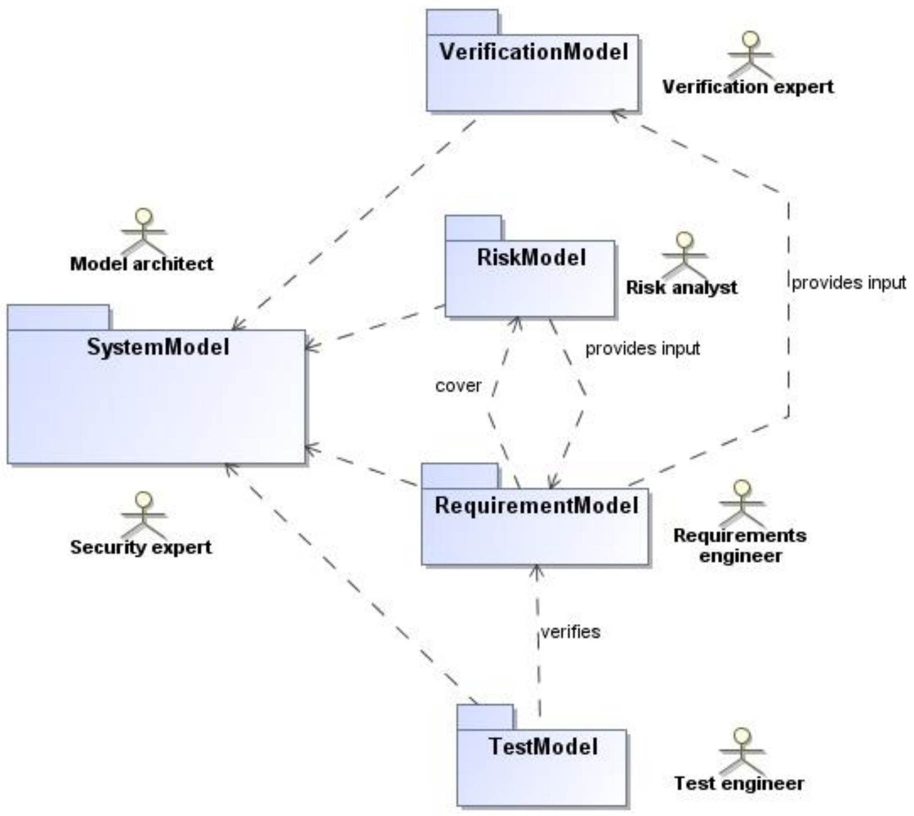


Figure 13 Integrated view of SecureChange artefacts and their dependencies

The meta model which is described in Section 3.1 is a working model in a specific context. It maps to the integrated view of SecureChange artefacts in the sense that it is a specific instantiation of a system model, partly a requirements model and a risk model. That way the specific working model can be mapped to one or more of the artefacts depicted in Figure 13.

3.2.2 Integrated SecureChange process metamodel

The description of the overall SecureChange process will deliver concepts of change derived from all the solutions of the different SecureChange Work Packages. The goal is to provide an integrated meta model of change related concepts which is independent from any Work Package specific solution (cf. Figure 14).

Consider as an example a change in the infrastructure that requires a change in the system model. The system change triggers a system analysis to analyse the changes with the result of an updated system model. An updated system model might affect the current set of requirements and therefore triggers a requirements analysis resulting in a new updated requirement model. The update of the requirement model and the update of the system model both potentially impact the current test model. Therefore both changes trigger new test engineering with the result of a new updated test model.

The dependency relations between the different types of artefacts are the frame for change propagation.

Currently this change model is in a conceptual development phase and will be elaborated during Year 2 and Year 3. The different change related concepts provide a basis for the description of change handling in the integrated SecureChange process.

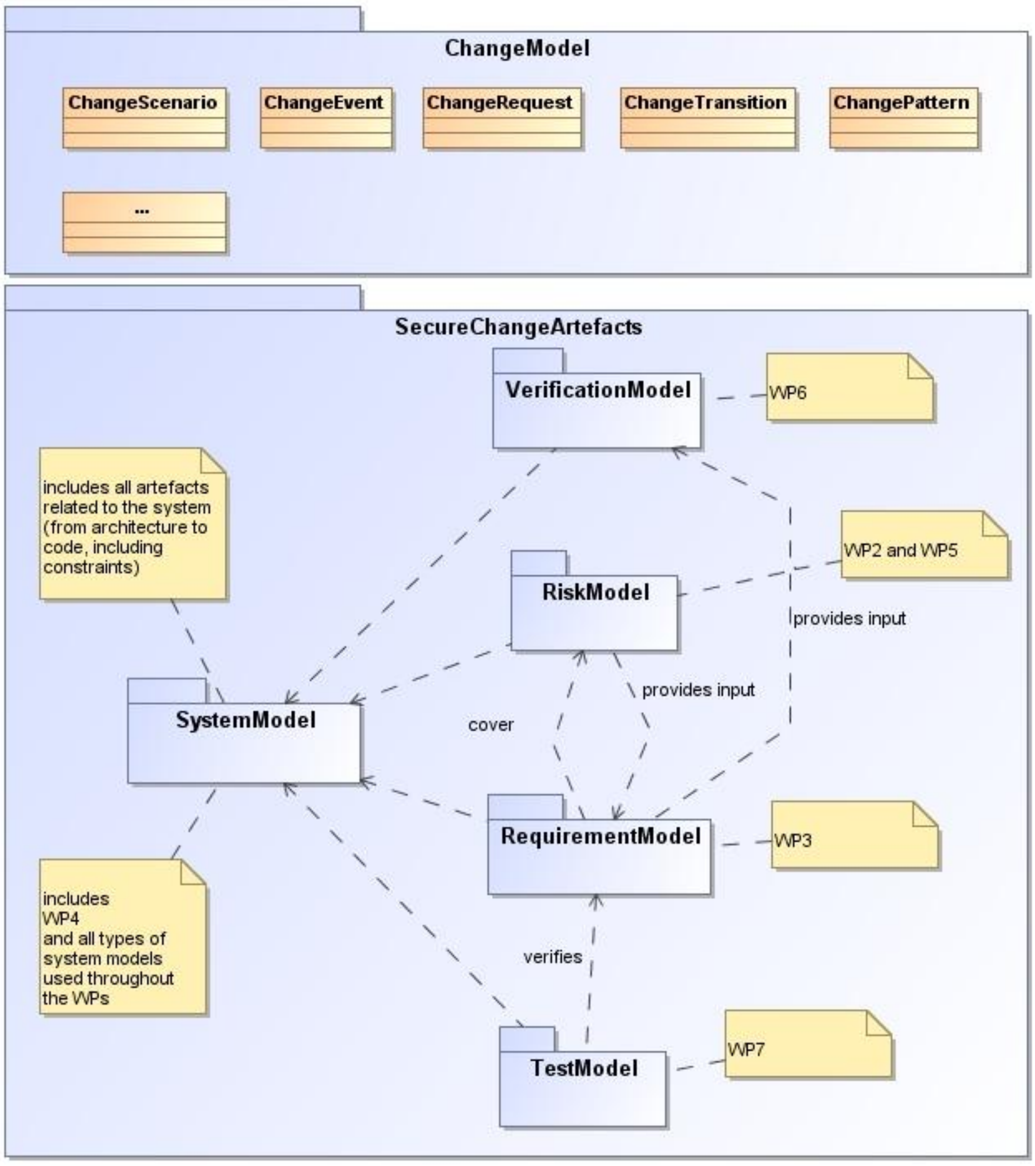


Figure 14 The integrated SecureChange process metamodel

At the moment we have identified the following list of change concepts in the various work packages (cf. Figure 14):

- ChangeScenario: is expressed at the requirements level and describes the change in the requirements. This change scenario will consist of a before and after requirements model.
- ChangePattern: consists of a specific change scenario, one or more solutions and a mapping between the elements from the change scenario and the architectural elements in the solution.
- ChangeEvent: is a general trigger of change which is derived from a set of change scenarios.

- ChangeRequest: is a general description of some change in the system.
- ChangeTransition: is a description of all the differences from one change to another.

Additional concepts which are candidates for the inclusion in the Change Model is the concept of perspectives which is used in Work Package 5, the concepts of Change Line, Version, Change Propagation and others.

Future tasks related to the refinement and further development of the integrated SecureChange process meta model include the collection of additional change related concepts throughout the other Work Packages. All these change concepts will be consolidated as an integration for all the Work Packages. In addition changes need to be classified to provide different basic categories of changes which might require a different handling. Activities in other Work Packages provide a sound basis for the development of such a Change Model, such as the Deliverable D3.2 of Work Package 3 and the Deliverable D5.2 of Work Package 5.

3.3 The security analysis method by Thales

An example for a concrete instantiation of the SecureChange process is the security analysis method by Thales. The method incorporates different concepts of changes in a separate Change Model and different artefacts and their dependencies in a Static Model. In particular the approach supports the concept of change propagation of the SecureChange process by building on the concept of using states of different model elements to track and trigger changes. In this Section the security analysis method by Thales is described providing an overview of the principles, a DSML and how change is handled.

As a long-term industrial initiative, Thales develops a new method to support security risk analysis, closely integrated with the overall engineering process of our critical information systems. This method is building upon model-based engineering techniques [30], it presents a prototype domain-specific modelling language (DSML) that was developed in this context; this DSML aims at supporting the analysis and assessment of security risks for a system, and the specification of requirements for security measures to address those risks. Our objective is to provide adequate and efficient tooling to security engineers for an effective integration of security engineering in the process of critical system design, so as to enable a better targeting of security specifications.

3.3.1 The security risk analysis method: Principles

Our prospective security risk analysis method builds upon model-based engineering methods and techniques. All activities of our method are organised around the building and usage of models, that is formalised, precisely defined, interconnected and integrated representations of the objects under study.

As represented in Figure 15, our proposed method relies on the development of a modelling framework that combines in a synchronised way a set of models that constitute separate viewpoints [29] over the engineering problem:

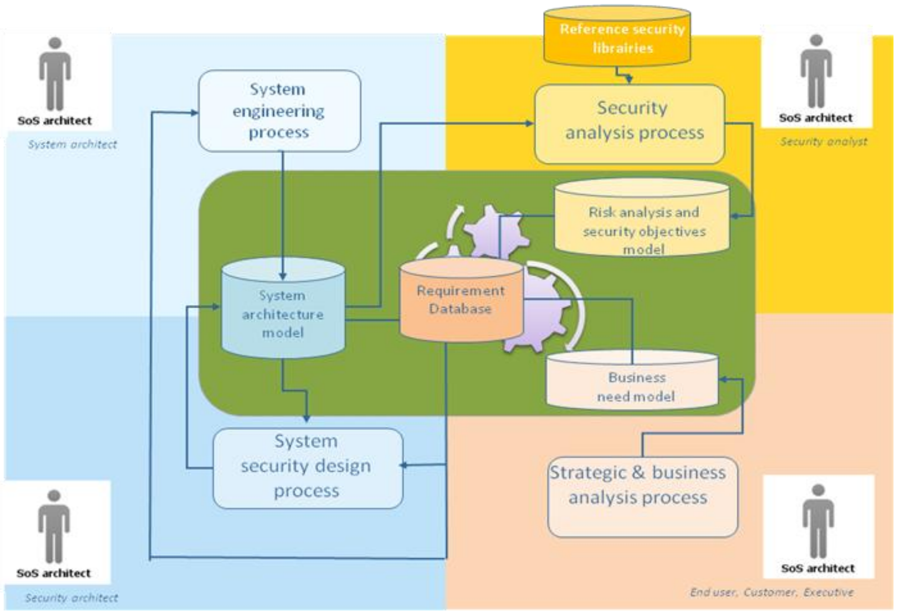

Reference security librairies
SoS architect
System architect
System engineering process
Security analysis process
SoS architect
Security analyst
Risk analysis and security objectives model
System architecture model
Requirement Database
Business need model
System security design process
Strategic & business analysis process
SoS architect
Security architect
SoS architect
End user, Customer, Executive

Figure 15 The security analysis method in Thales context – big picture

- The System architecture model contains the architectural design of the system; this model is developed within the mainstream engineering processes, along at least two dimensions: the functional / logical architecture of the system (functional capacities and data to be realised by the system) and the physical /implementation architecture of the system (actual hardware and software components that realise the functional capacities).
- The Business need model captures a representation of the business context for the system: business process that is supported, underlying business organisation, business objects, key performance indicators, strategic drivers, etc.
- The Risk analysis model and security objectives model capture the results of the security risk analysis method that is proposed in dedicated DSML (presented in next section). These models include a representation of the system architecture that is relevant to the needs of the security analyst, this model is called context model. This model is traced back and maintained in synchronisation with the system architecture model (see [31]). The security risk analysis information is defined as annotations or related new concepts added over the system architecture elements. The risk analysis model and security objectives model may also be traced to elements of information defined in the Business need model.
- The Requirement Database captures all kinds of systems requirements (Security, Safety, Maintainability, Cost ...). Security requirements are derived from the security objectives model of the dedicated DSML (see [32]). This mapping enables to merge security requirements with all kind of requirement addressed for a complex system. The Requirement Database is traced back and maintained in synchronisation with the system architecture model and Business need model.

The System architecture model and the Business need model are part of the architecture modeling framework that we are developing to address service-oriented types of large-scale enterprise integration systems or systems of systems. In the Thales context, the official database of Requirement Management is Rational DOORS with the T-REK add-ons [33] (Dynamic Object Oriented Requirements System).

3.3.2 Security DSML: Overview

The Risk analysis model, security requirements model and context model are expressed in a dedicated DSML[1]. These kinds of models are parts of the ***static model*** (corresponding to SecureChangeArtefact in Figure 14 The integrated SecureChange process metamodel):

- ***The Requirement Model*** describes the specialization of Objectives into several Requirements and links between those and the other elements of the DSML (Risk, Context).
- ***The Context Model*** describes the System Architecture (Essential Elements and/or Target), related constraints which describes how services (described by essential element) are provided by the system and links between those and the other elements of the DSML (Risk, Requirement).
- ***The Risk Model*** describes the risk characterization into threats, damages and vulnerabilities and links between those and the other elements (Requirement, Context).

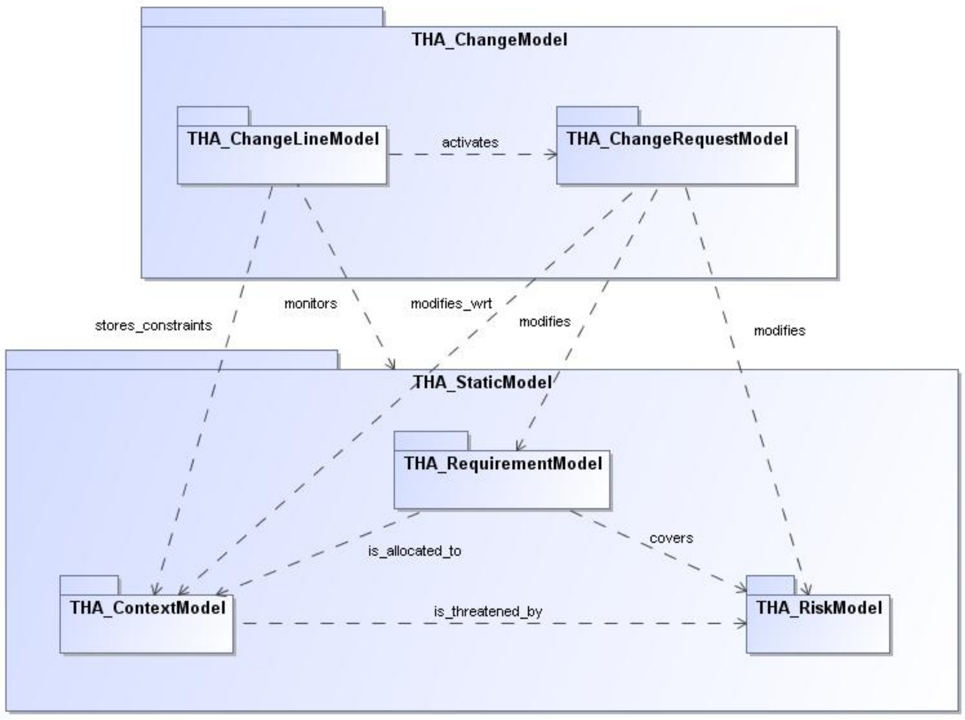


Figure 16 Relationship between Static Models and Change Model

[1] This deliverable cannot be the place for a detailed presentation of the metamodel and syntax of our DSML, more details are provided in [26] [30] [32].

To address change inside this DSML according to Section 3.2.2, we consider a change model which is mapped with all models included in the static model. Figure 16 depicts the relation between different models defined in the DSML and the relation with the change model presented in the next section.

Inside the ***Change Model***, we distinguish two kinds of Models:

- The ***Change Line Model*** represents relationships between several ***Changes*** included in one ***Change Line*** and ***Transitions*** (i.e Change Transition in Section 3.2.2) which describes a set of transformation rules between several changes
- The ***Change Request Model*** which traces changes inside the ***Static Model***. The ***Change Line Model activates the Change Request Model***.

Inside the ***Static Model***, the ***Requirement Model*** must ***cover*** risks expressed in the ***Risk Model*** and requirements ***are allocated to*** system elements (e.g. services, components) defined in the ***Context Model***. The ***Context Model*** is the representation of the system; this model ***is threatened by*** risks expressed in the ***Risk Model***.

The ***Change Request Model*** modifies all models of the ***Static Model*** (represented by the ***<<modifies>>*** dependency relation). The ***Change Line Model*** is described by a set of evolution functions which ***monitors*** the ***Static Model***: context elements are described by several evolution functions, requirements and risk includes evolution functions (e.g. time). The dependency relation ***<<stores_constraints>>*** presents the relationship between constraints and change transitions. Constraints (i.e. contract) describe how services (described by essential element) are provided by the system. These constraints must be stored in change transition in order to respect these constraints inside the ***Change Request Model***. This is why the ***Change Request Model*** modifies the ***Context Model*** with respect to constraints defined in this Model (denoted by ***<<modifies_wrt>>*** relation).

3.3.3 Change Model

To represent traceability between changes and the static model, we add a further Model into the DSML: ***Change Model*** is composed by several ***Change Lines***. As shown by Figure 17, a ***Change Line*** is considered as set of ***Changes*** and ***Change Transitions*** to preserve links and grant consistency between successive changes which compose a Change Line.

Change is described by a ***Change Trigger*** (e.g. discovery of fault or new threat which correspond to Change Event in Section 3.2.2), Change Trigger expresses the rationale of Change and activates a ***Change Request***. It is also possible to activate a Change Trigger by a threshold defined in an ***Evolution Function*** which monitors the static model of the system.

As shown by Figure 17, a ***Change Request*** contains a PUID[2] to identify it, a description a status which represents the state of the Change request (for further detail see next section). After the activation of Change Request by the Change Trigger, Change Request status is first defined in CCB (Configuration Control Board represented by THA_Configuration_Control_Board package[3]). The configuration (or

[2] PUID = Product Unique Identifier
[3] The description of this package is out of scope in this Deliverable

change) control board (CCB) is a periodic meeting between several actors of a development team (client, manager, quality, design, integration …) to define change requests which are accepted, refused or postponed in the next version of the system. The detailed behavior of a Change Request is described in the next section.

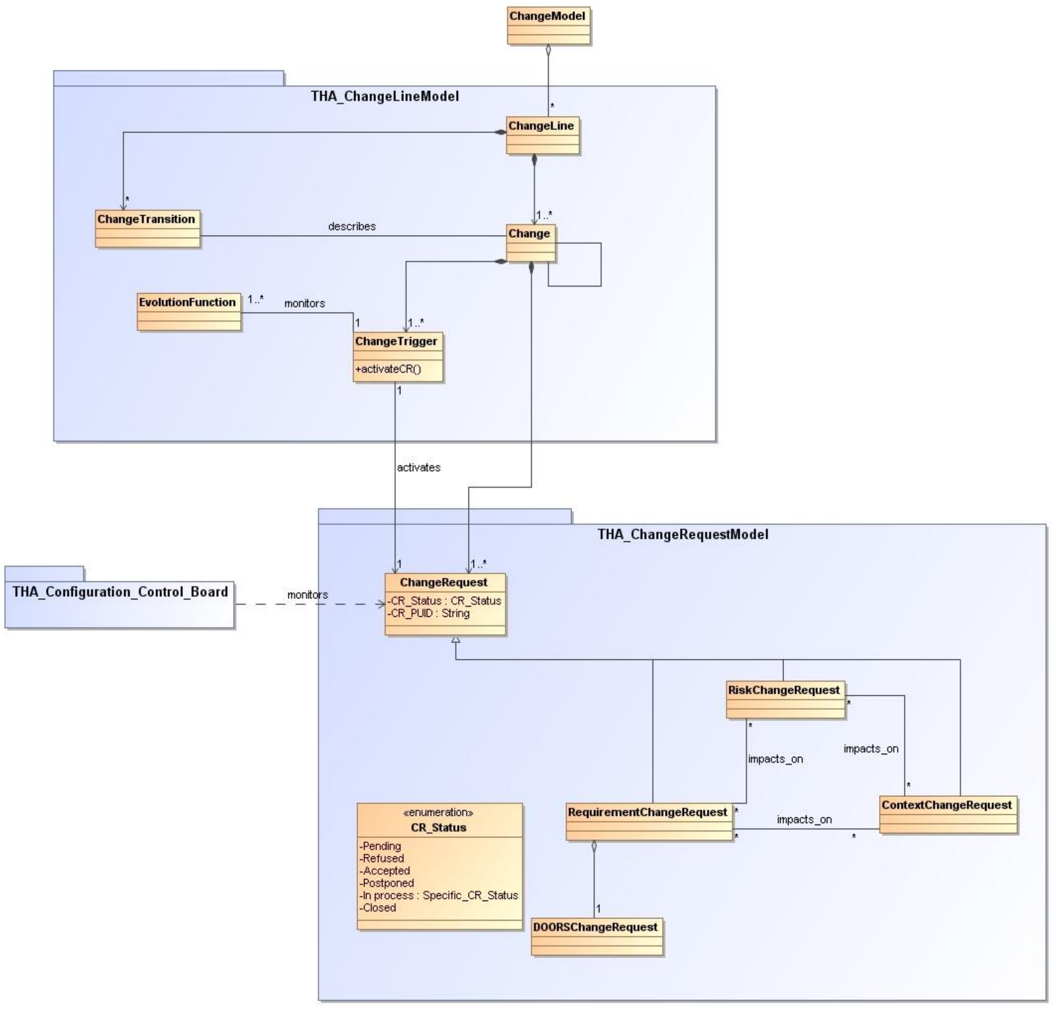


Figure 17 Change Model Conceptual Model

To cover all kinds of static model, ***Change Request*** is specialized into the following kinds:

- A ***Requirement Change Request*** modifies the Requirement Model (Requirement, Objectives). It is possible to map this kind of Change Request with a DOORS Change Request (for further details see [32]).
- A ***Context Change Request*** modifies the Context Model (e.g. system architecture).
- A ***Risk Change Request*** modifies the Risk Model (Risk, Threat, Damage, and Vulnerability).

These three kinds of Change Request are dependant; a Requirement Change Request can impact on a Risk Change Request and a Context Change Request and vice versa.

This is why we consider a traceability relation between those Change Requests. This relation is described by an “impacts_on” association (see Figure 17).

3.3.4 Change Request: Behaviour

For readability sake, Change Request Behavior is described by a UML Statechart Diagram. In first, we present the generic behavior of a Change Request including CCB status relations. In second, we describe the specific behavior of Risk, Requirement and Context Change Request.

3.3.4.1 General Behaviour of Change Request

As suggested by Figure 18, ***Change Request*** (CR) starts after Change Trigger activation (e.g. discover a fault, a new requirement …). The Redactor of a Change Request must define the change and trace it with the impacted elements. Change Request is per default in ***Pending*** State.

A CCB must be planned; it monitors the Change Request Status which can be in the following states:

- ***Refused***, the CR is not relevant; it is not integrated in the system. The Change Request is ended in this state.
- ***Postponed***, the CR is relevant but it is not possible to integrate it in the current version of the system. This CR is planned for the next version.
- ***Accepted***, the CR is integrated in the current version of the system.

If the CR is accepted, it will be in the ***In_process*** macro state. This macro state is specialized for several models of Static Model (Risk, Requirement or Context). Specific Change Request Processes are described in the next section.

A CR is finished if and only if it is closed in CCB with client agreement.

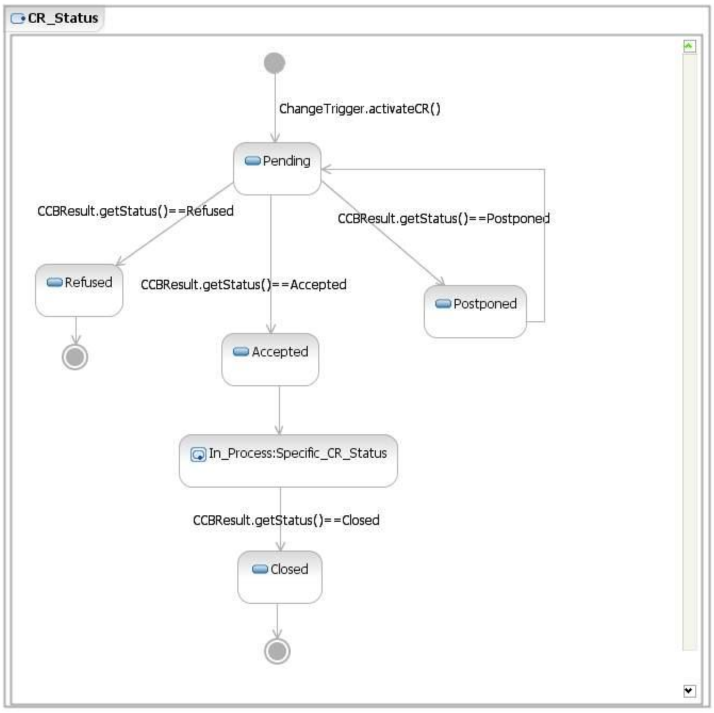

CR_Status
ChangeTrigger.activateCR()
Pending
CCBResult.getStatus()==Refused
CCBResult.getStatus()==Postponed
Refused
CCBResult.getStatus()==Accepted
Postponed
Accepted
In_Process:Specific_CR_Status
CCBResult.getStatus()==Closed
Closed

Figure 18 Change Request Behaviour

3.3.4.2 Specific Behavior of a Risk Change Request

A specific ***Risk Change Request*** Behavior starts after the ***Accepted*** state in the generic behavior. As shown by Figure 19, Risk Change Request Status is represented by the following sequence of states:

- ***In_progress***, the redactor must define the changed risk.
- ***To_be_Evaluated***, the redactor must re-qualify new or changed risk by setting Risk Opportunity and Severity attributes [26] [30].
- ***To_be_Managed***, the redactor of the Risk Change Request must take into account the impact of this change request with the other elements (Requirement and Context) and change them if necessary with new CR(s).

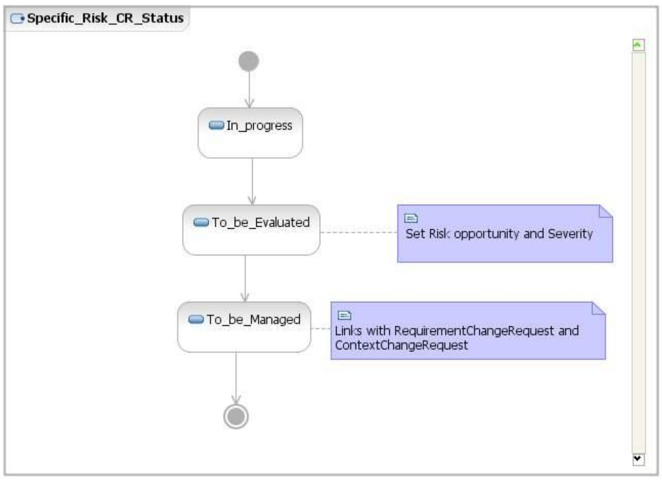

Specific_Risk_CR_Status
In_progress
To_be_Evaluated
Set Risk opportunity and Severity
To_be_Managed
Links with RequirementChangeRequest and ContextChangeRequest

Figure 19 Risk Change Request Behaviour

3.3.4.3 Specific Behavior of a Requirement Change Request

A specific ***Requirement Change Request*** (RCR) Behavior starts after the ***Accepted*** state in the generic behavior. As shown by Figure 20, the Requirement Change Request Status is represented by the following sequence of states:

- ***To_be_Managed***, the redactor of Requirement Change Request must take into account the impact of this change request with the other elements (Risk and Context) and change them if necessary with new CR(s).
- ***In_progress***, the redactor must define changed requirement, the designer must model them, and the developer must implement them.
- ***To_be_verified***, the integrator must take into account these changes in the test campaign (and change the test scenario if necessary).
- ***Resolved***, the RCR Status will reach this state if and only if changed requirement is verified in the test campaign.

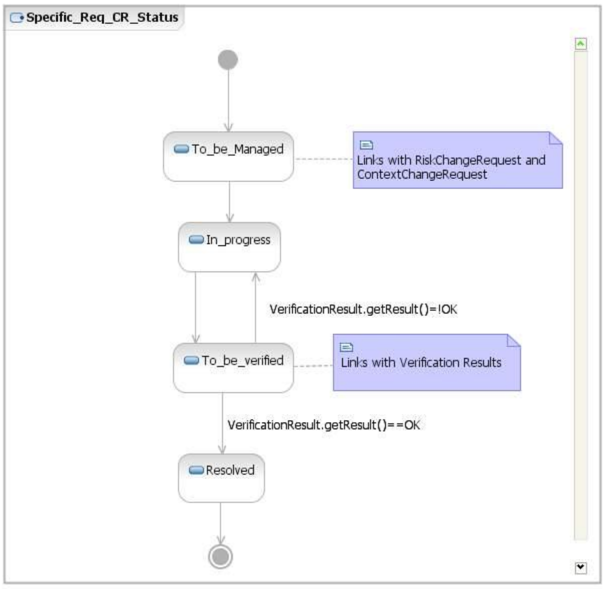

Specific_Req_CR_Status
To_be_Managed
Links with RiskChangeRequest and ContextChangeRequest
In_progress
VerificationResult.getResult()=!OK
To_be_verified
Links with Verification Results
VerificationResult.getResult()==OK
Resolved

Figure 20 Requirement Change Request Behaviour

3.3.4.4 Specific Behavior of a Context Change Request

A specific ***Context Change Request*** (CCR) Behavior starts after the ***Accepted*** state in the generic behavior. As shown by Figure 21, the Context Change Request Status is represented by the following sequence of states:

- ***To_be_Managed***, the redactor of the Context Change Request must take into account impact of this change request with the other elements (Risk and Requirement) and change them if necessary with new CR(s).
- ***In_progress***, the redactor must define changed components, the designer must integrate it into models, and the developer must implement or use it.
- ***To_be_qualified***, the integrator must take into account these changes in the qualification process.
- ***Resolved***, RCR Status will reach this state if and only if the changed component is qualified.

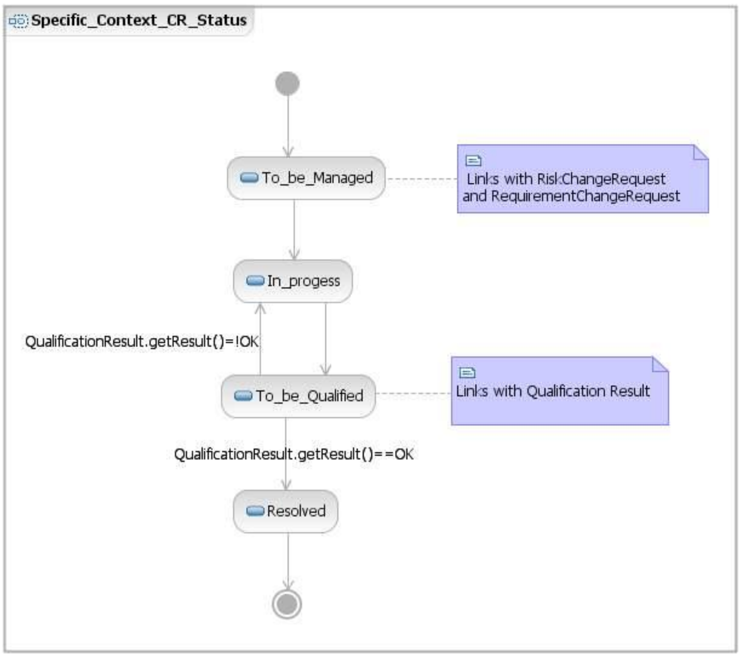

Specific_Context_CR_Status
To_be_Managed
Links with RiskChangeRequest and RequirementChangeRequest
In_progess
QualificationResult.getResult()=!OK
To_be_Qualified
Links with Qualification Result
QualificationResult.getResult()==OK
Resolved

Figure 21 Context Change Request Behaviour

4 Change patterns

The previous section described an overarching security process. In this process, when a change occurs, it propagates throughout the whole model of the system. Each stakeholder is triggered to perform appropriate actions if that change affects the stakeholder’s view on the system. In this process, the actions that the stakeholder needs to execute are not specified, however. In the remainder of the deliverable, we focus on what one specific stakeholder, namely the architect, can do to deal with evolution.

In the current section, we propose one specific technique for the architect to deal with change. 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, and the process is connected to the overarching security process from the previous section. Finally, a catalogue of change patterns for dealing with evolving trust relationships is presented, and its use is illustrated.

In Section 5, a generic architectural blueprint is outlined that is designed to accommodate a broad set of changes. The architect can design the architecture using this blueprint, and can use it as a starting point for applying change patterns.

4.1 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 22.

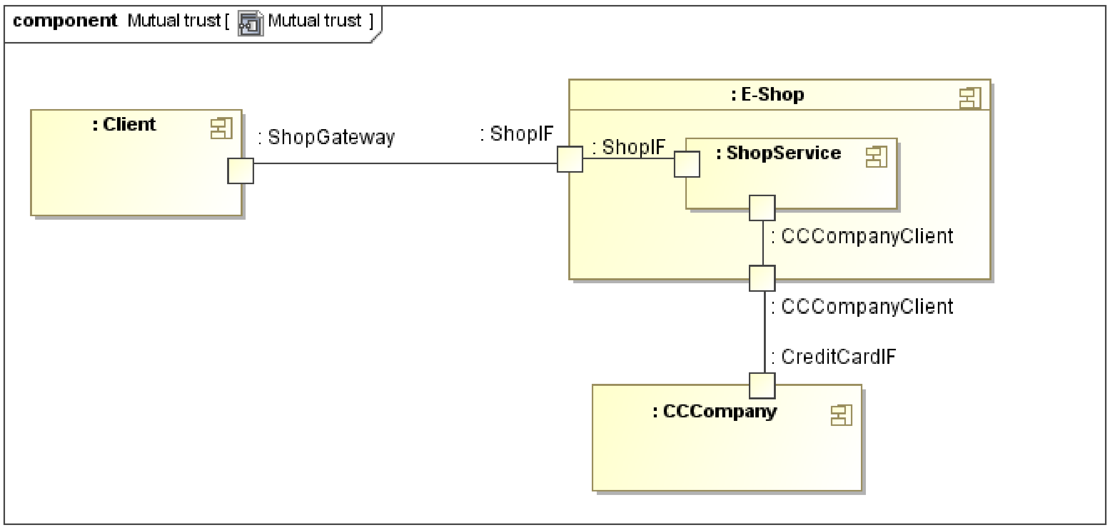


Figure 22 Mutual trust

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.

The trust assumptions turned out not to hold for the shop, though, and after a while the shop gets wound up in its first lawsuit filed by an unhappy customer. There, it becomes painfully clear that the shop cannot present any credible evidence that the client has indeed placed the 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 23.

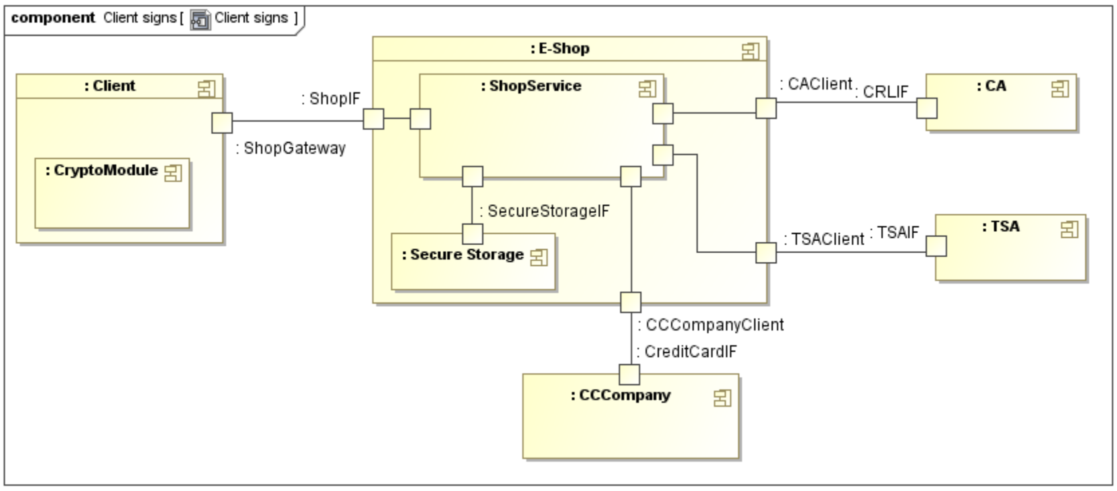


Figure 23 Client digitally signs orders

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 or purchase, even if the client's signature key gets compromised afterwards. Finally, the signed order and timestamp are stored as evidence on a secure 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. Luckily, the credit card company 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 24.

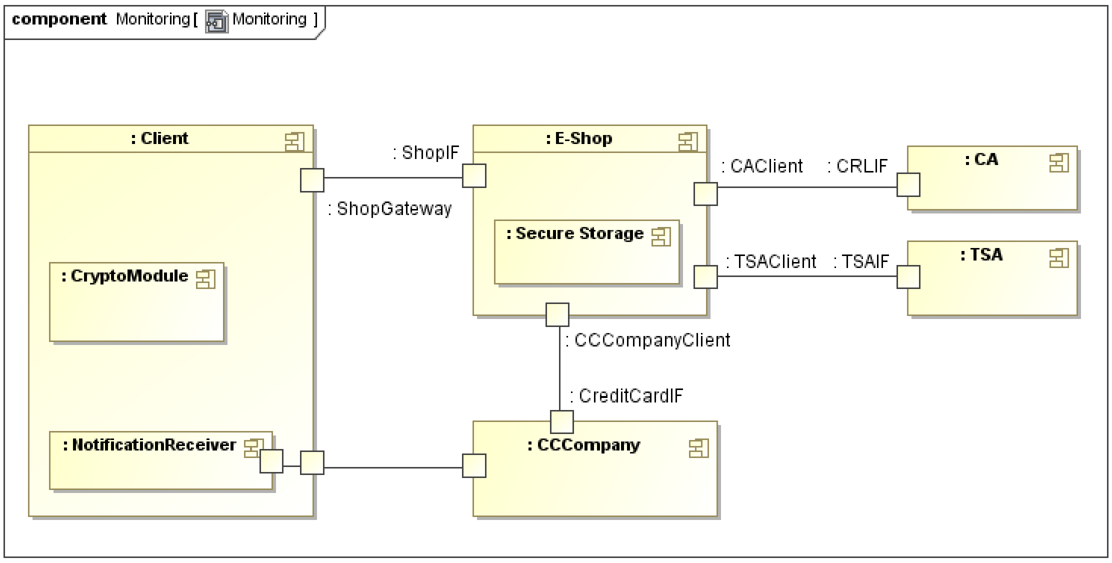


Figure 24 Client monitors activities

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 25.

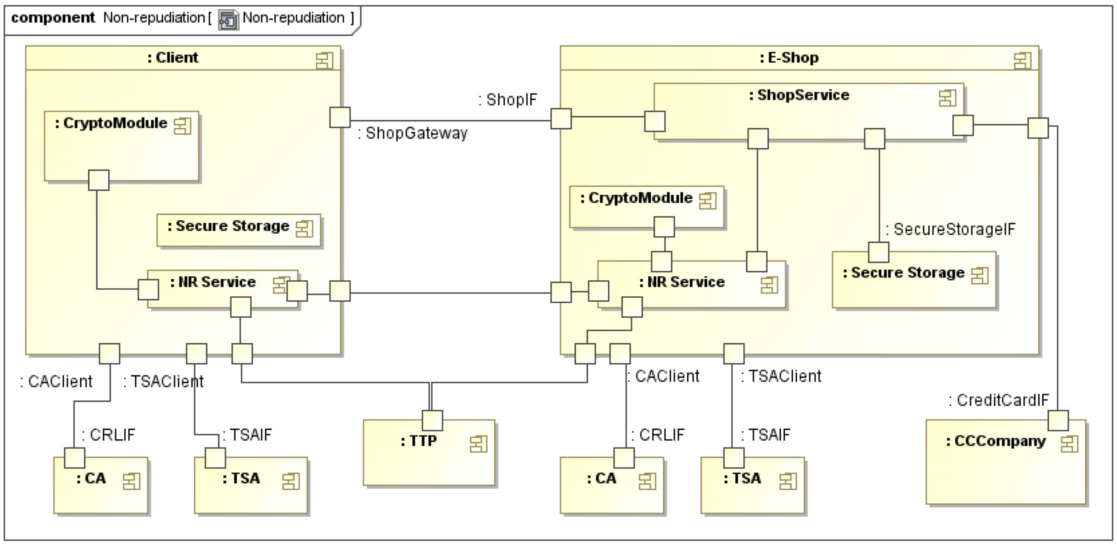


Figure 25 Non-repudiation

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 3. It is apparent 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.

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 3 Evolution of trust

4.2 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[4] 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.

4.3 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. Within the overall process presented in Section 3, it reflects one possible strategy that an architect can follow within his domain and responsibility. The process is triggered by changes occurring outside this domain, for instance in the requirements domain. In its turn, following the process will eventually trigger changes in the implementation or deployment domain.

[4] For brevity, we will from this point onwards refer to both requirements and environmental assumptions simply as ‘requirements’.

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 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.

4.3.1 Place in the SecureChange Framework

A change pattern is related to the SecureChange framework from Section 3 in the following way. The common system view from the framework consists of meta model elements describing the entire system. Each stakeholder has a specific view on these elements: the requirements engineer focuses on the elements related to requirements, the software architect on the architectural and design elements, and so forth. The change pattern approach describes a process for the architect to deal with change at the architectural level.

In Figure 26, we schematically depict how a change pattern is related to the occurrence of changes, i.e., a transition from one system state to another. A change pattern consists of a before/after pair at the requirements level, that is, it describes the change scenario which it supports at the requirements level. To enable the use of the change pattern in the architecture, the architect needs to instantiate the architectural support patterns that belong to the change pattern into the architecture. Note that this needs to happen before the actual change occurs, as the architectural patterns may have a significant impact on the architecture. Finally, the change guide from the pattern describes how the change at the requirements level can be implemented in the architectural domain (by the architect), and/or other domains (e.g., implementation or deployment), using the architectural patterns that are present in the architecture.

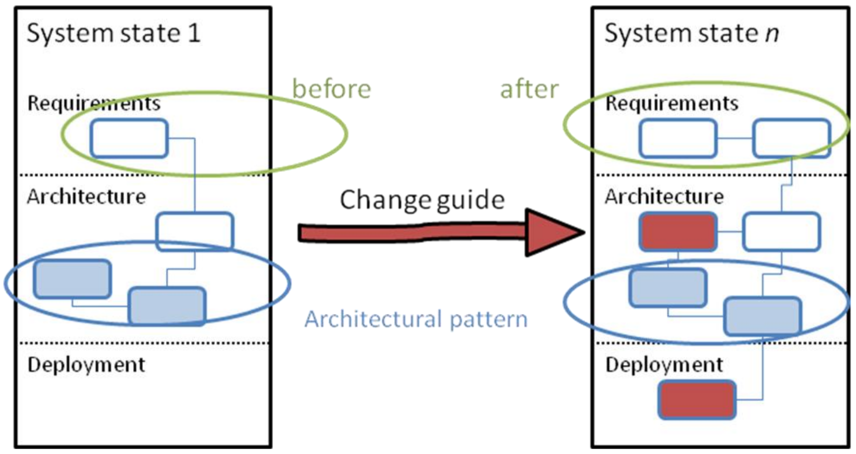


Figure 26 Relating change patterns to changing system state

Using the change pattern approach in the SecureChange framework requires a change pattern metamodel, 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.

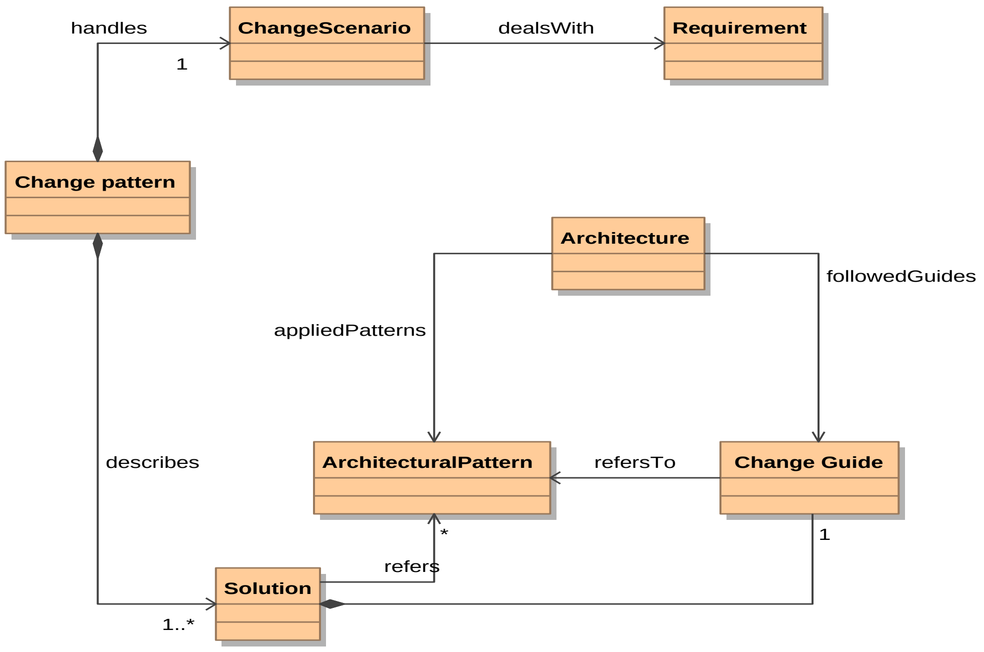


Figure 27 A change pattern meta model

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 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.

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	Over-protected	No	Nothing	Before	Followed	Overprotected
				Undo		Unfollowed	Matches
			Yes	Nothing	After	Followed	Matches
				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		Unfollowed	Risk

Table 4 Change patterns and state transitions

4.3.2 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 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.

4.4 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 deliverable 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[5]. 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.

4.4.1 Evolving trust scenarios

In the remainder of this section of the deliverable, 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 [34], adopted by the Secure Tropos methodology [35]. 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 [35] 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.

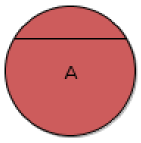
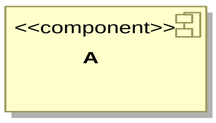
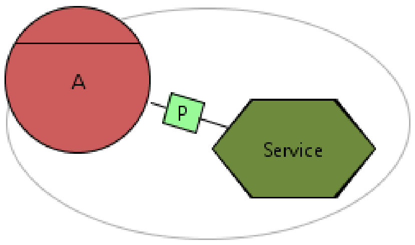
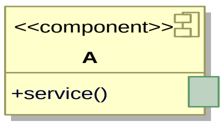
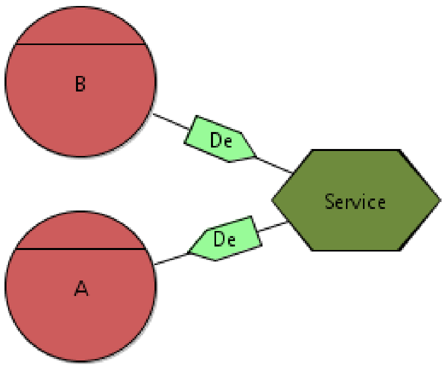
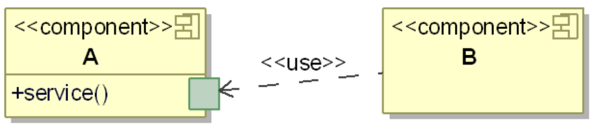
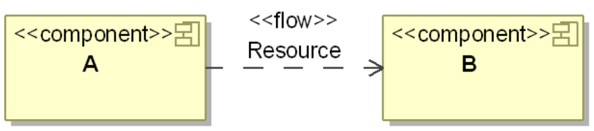
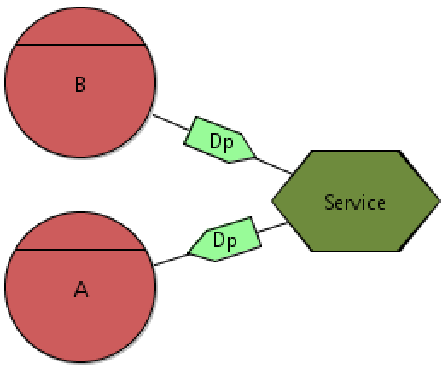
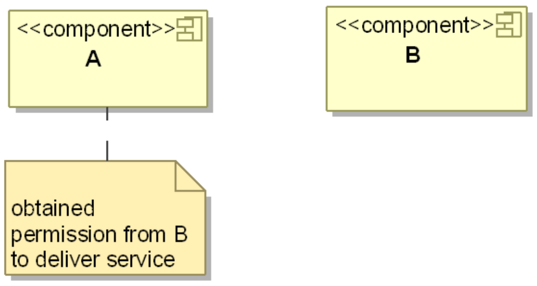
Finally, recall that both the concept of change patterns and the accompanying process are independent of the used requirements elicitation technique or model. Therefore, the elicitation technique that was simultaneously developed in Work Package 3 could be plugged in as well.

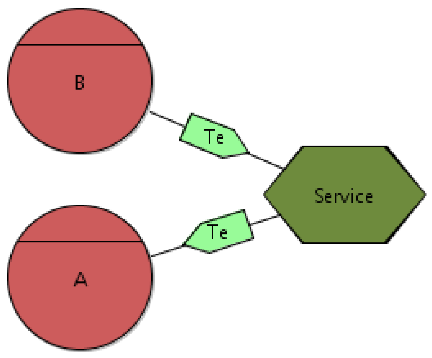
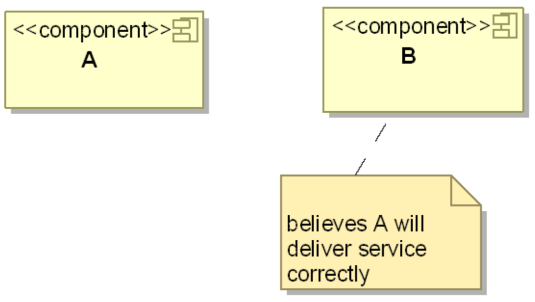
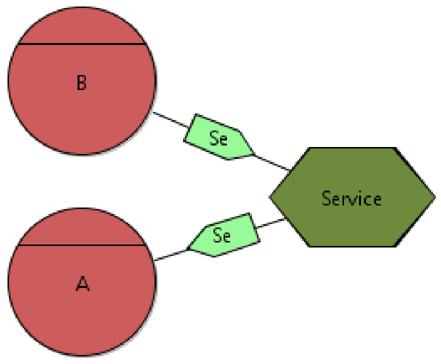
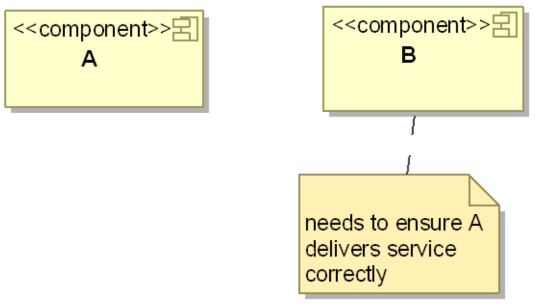
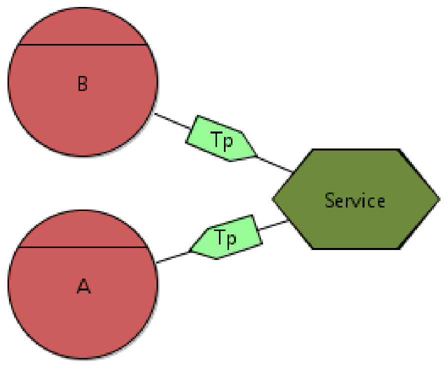
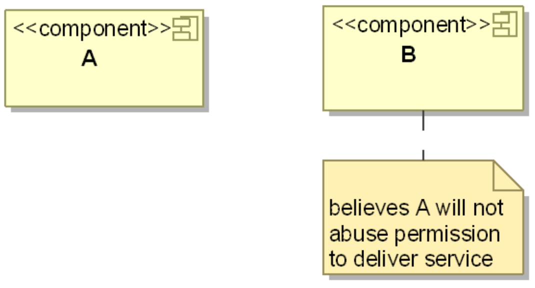
4.4.2 Mapping trust requirements to architecture

As discussed before, the catalogue needs to provide a mapping between SI* requirement models and component-oriented architectural models, which we will

[5] This is an illustration of Conway’s Law: “organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations”. (http://www.melconway.com/research/committees.html)

express in UML (version 2.0). This mapping is the same for all change patterns in the catalogue, and is depicted in Figure 28.

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

Trust of execution from B to A 	Not modelled explicitly  <<component>> A <<component>> B believes A will deliver service correctly
Distrust of execution from B to A 	Not modelled explicitly  <<component>> A <<component>> B needs to ensure A delivers service correctly
Trust of permission from B to A 	Not modelled explicitly  <<component>> A <<component>> B believes A will not abuse permission to deliver service

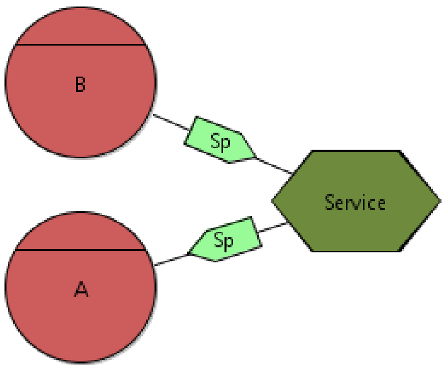
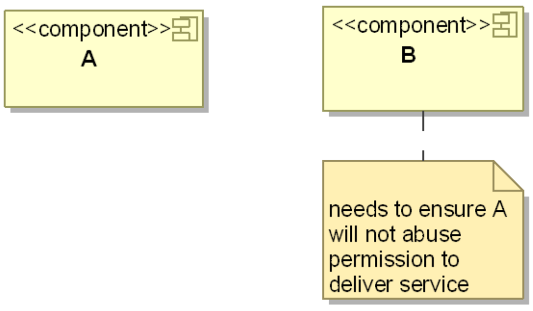
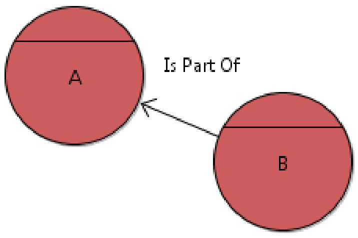
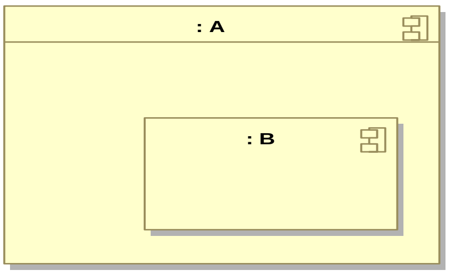
Distrust of permission from B to A 	Not modelled explicitly  <<component>> A <<component>> B needs to ensure A will not abuse permission to deliver service
Actor part of other actor 	Composite structure 

Figure 28 Mapping between SI* and UML

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).

4.5 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.

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 5.

Change scenario and solutions	Section	Page
Evolving trust of execution upon external actor Solution 1: Require commitment Solution 2: Use monitoring	4.5.1	66
Evolving trust of execution from external actor Solution 1: Provide commitment Solution 2: Enable monitoring	4.5.2	75
Evolving trust of permission upon external actor Solution 1: Apply least-privilege principle Solution 2: Attribute-based access control Solution 3: Use monitoring	4.5.3	82
Evolving trust of permission from external actor Solution 1: Request confirmation Solution 2: Enable monitoring	4.5.4	87
Delegate execution of a service to a trusted actor Solution: Encapsulate service	4.5.5	89
Delegate permission to a service to a trusted actor Solution: Flexible access control	4.5.6	92
Providing additional service with delegated execution Solution: Introduce bridge component	4.5.7	94
Providing additional service with delegated permission	4.5.8	96

Table 5 Change pattern catalogue overview

4.5.1 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[6]. This is shown in Figure 29.

[6] As a reminder: SI* represents trust using ‘Te’ and ‘Tp’ relations, for trust of execution and permission, respectively. Distrust is represented by ‘Se’ and ‘Sp’.

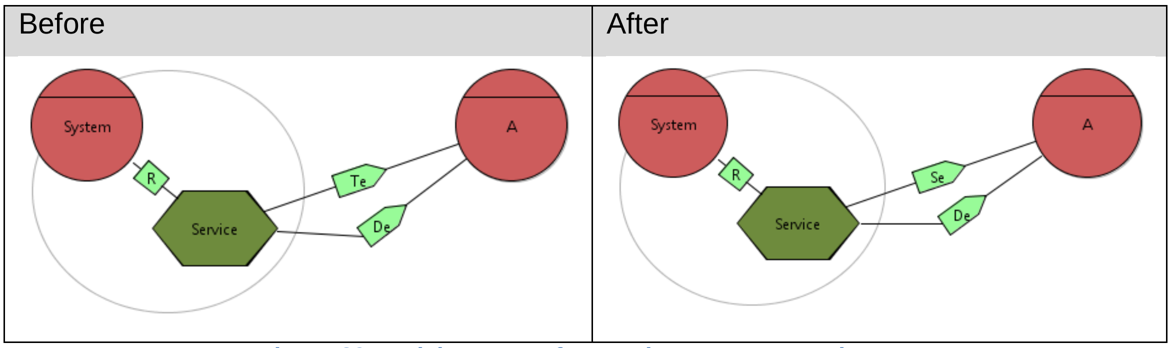


Figure 29 Evolving trust of execution upon external actor

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 28, to this example, we obtain the architectures in Figure 30 (before and after the trust relationship changes).

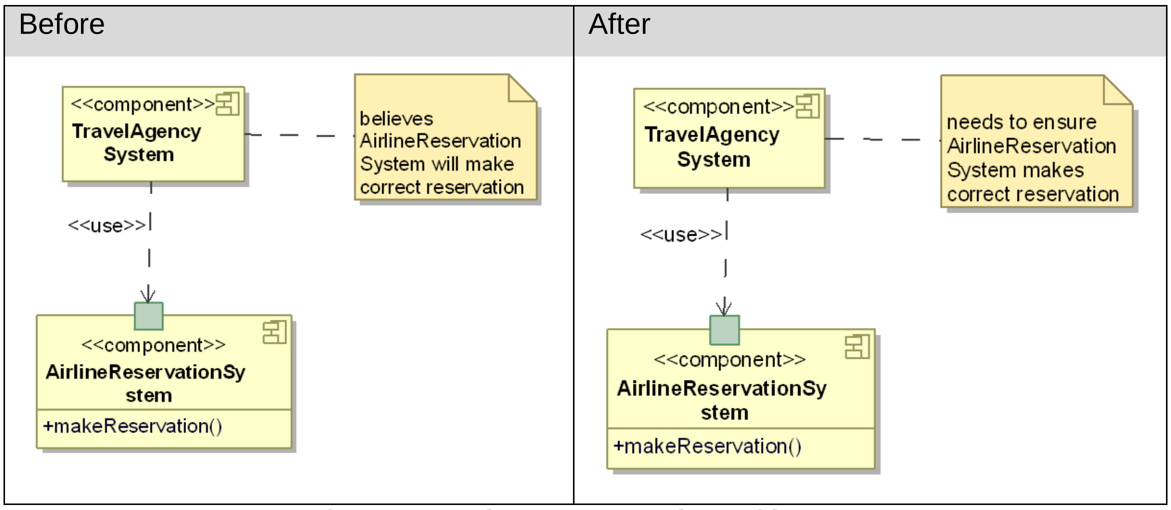


Figure 30 Requirements mapped to architecture

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.

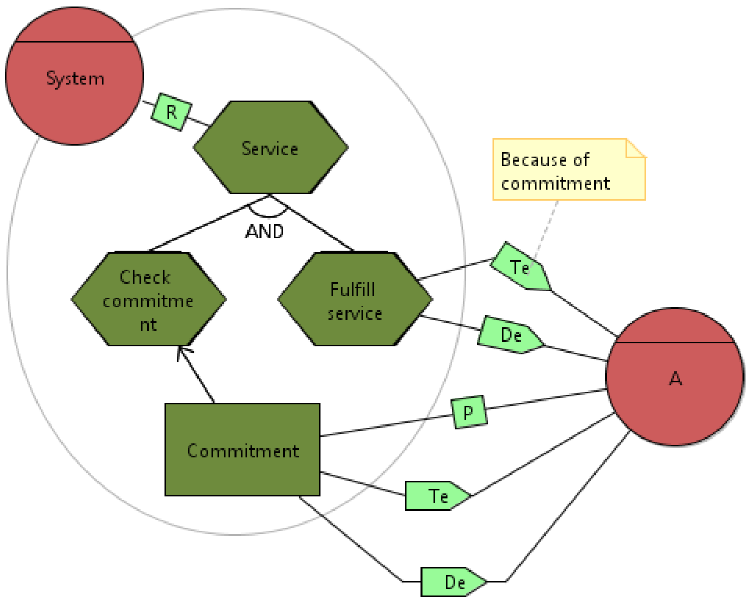
4.5.1.1 Solution 1: Require commitment

A first solution for the case above is presented as the 'non-repudiation pattern' in [36]. 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.

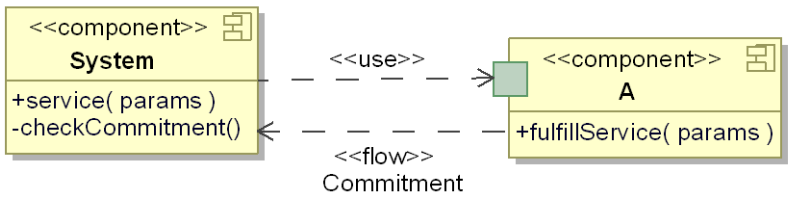
This solution strategy, expressed in SI* and mapped to the architecture, is shown in Figure 31.

Strategy



Strategy mapped to architecture

Structure



Behavior

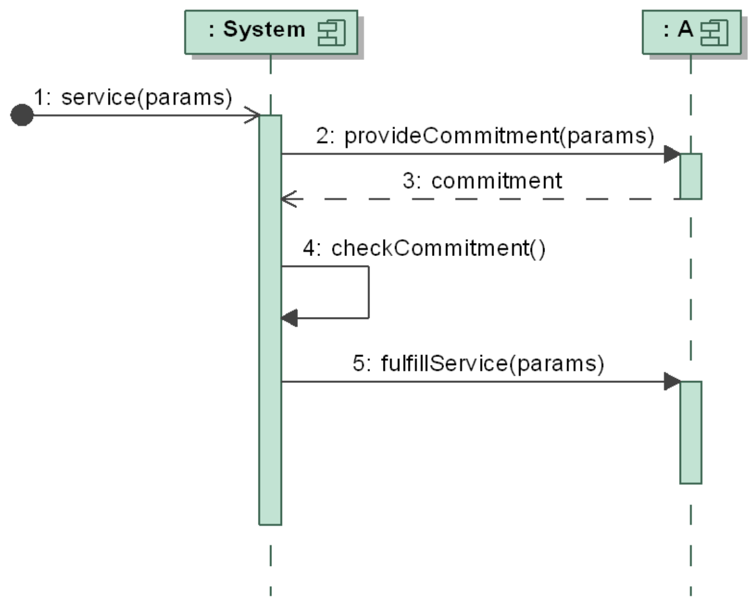


Figure 31 Require commitment

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

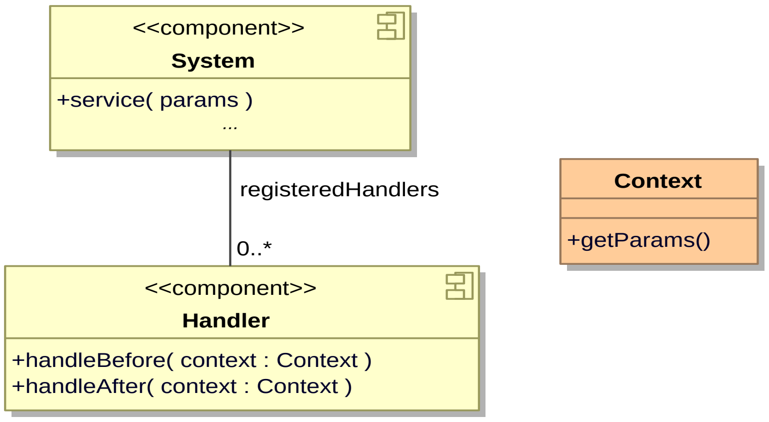
At the architectural level, it is clear that this solution requires the system to carry out additional actions and checks before and/or after one of its services is invoked. At a later point in time, these actions and checks might be removed or replaced. The architecture of the application therefore 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 32, 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.

Architectural support pattern

Structure



Behavior

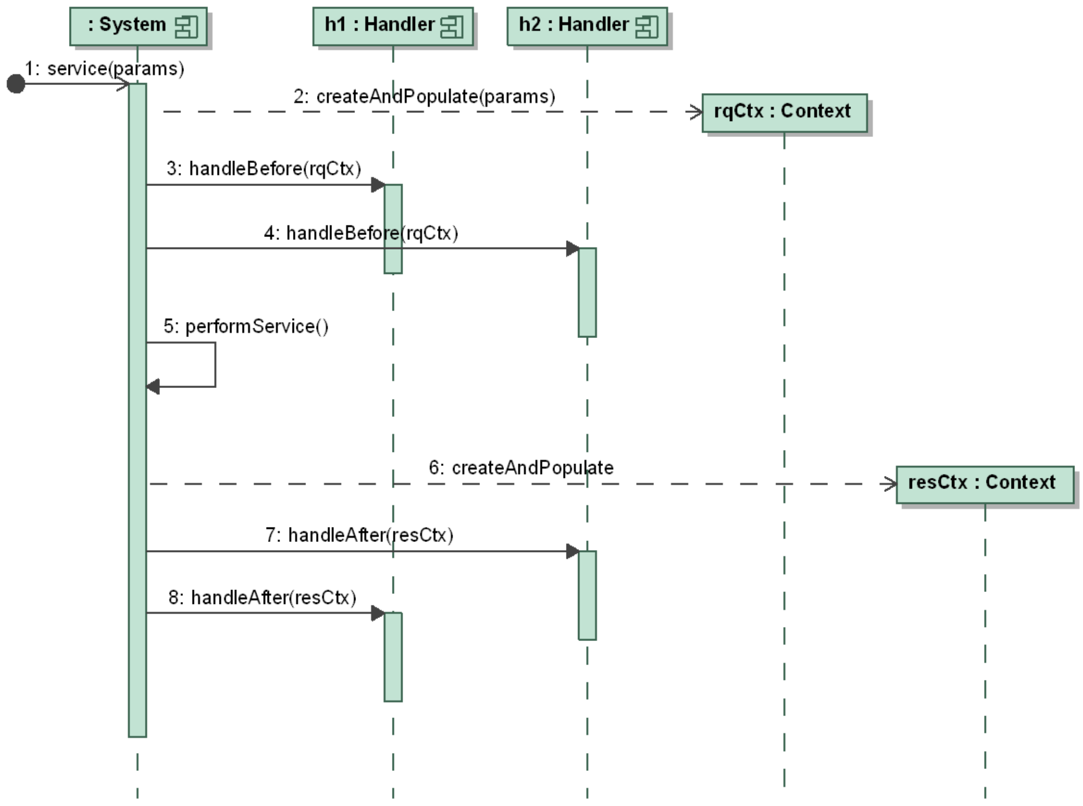


Figure 32 Handler architectural support pattern

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 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.

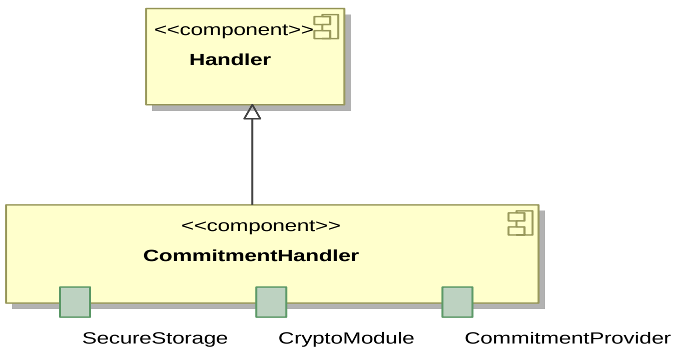
Without support from the underlying platform, the implementation has to be done manually for each operation. In this case, it is clear that aspect-oriented technologies can provide a significant benefit. They enable the modular interception technique that is needed for this pattern.

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 33). 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.

Structure



Behaviour

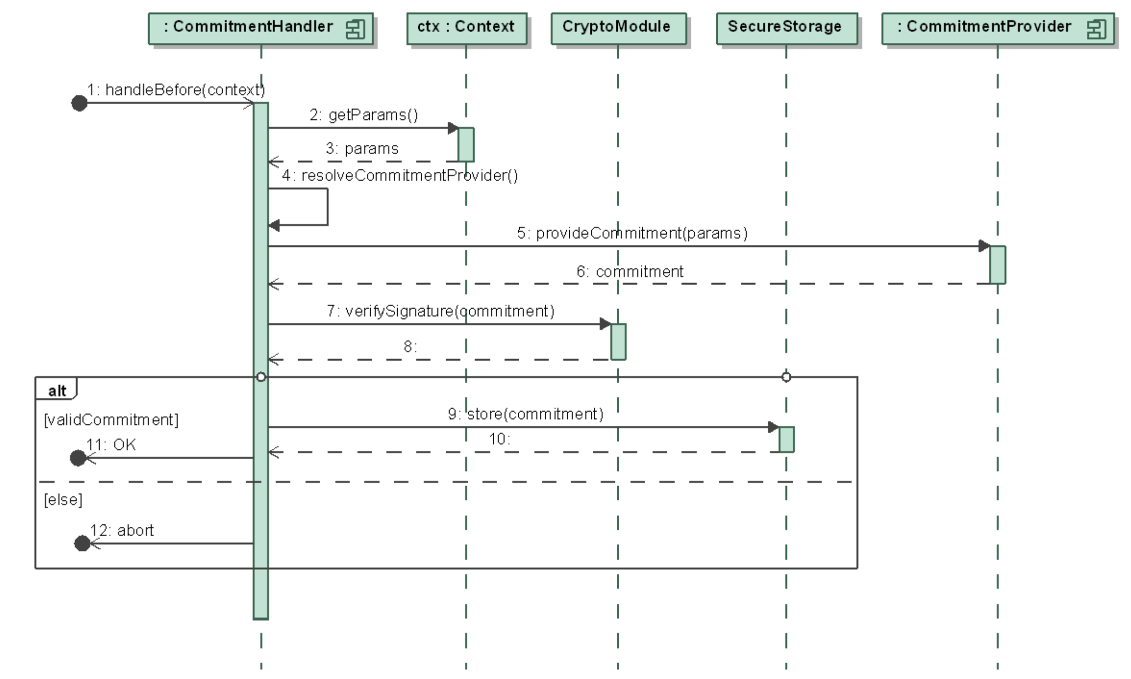


Figure 33 Applying the handler architectural support pattern

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 34.

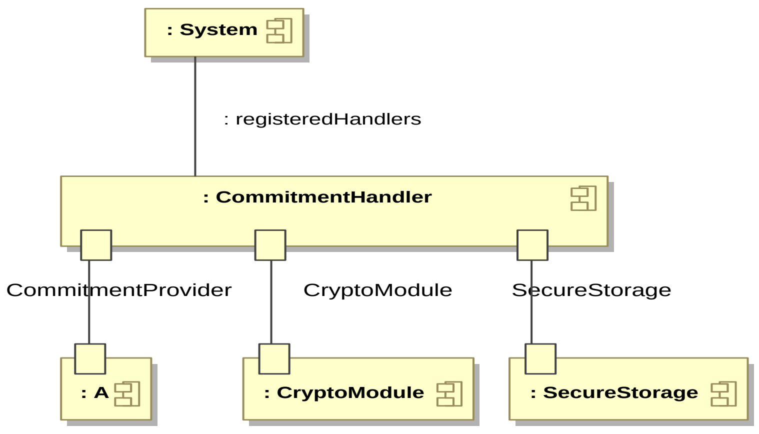


Figure 34 Adding a commitment handler

The commitment 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.

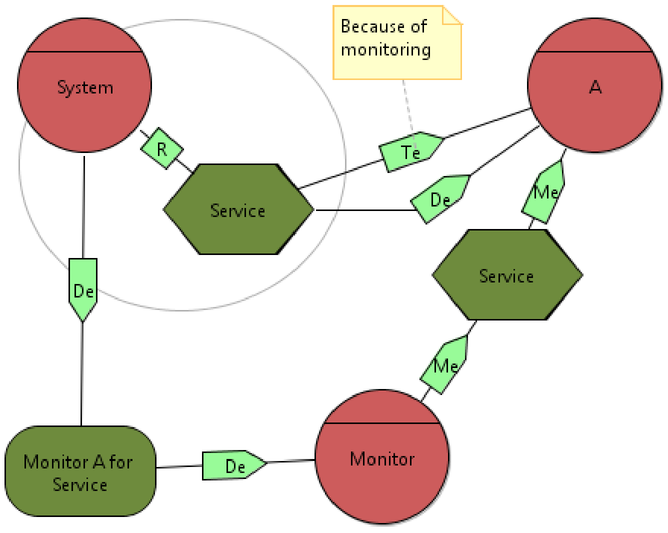
The addition of this handler ensures that the negative effects of the change in trust relationships are mitigated.

4.5.1.2 Solution 2: Use monitoring

A second solution in this case is the use of monitoring (Figure 35). 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.

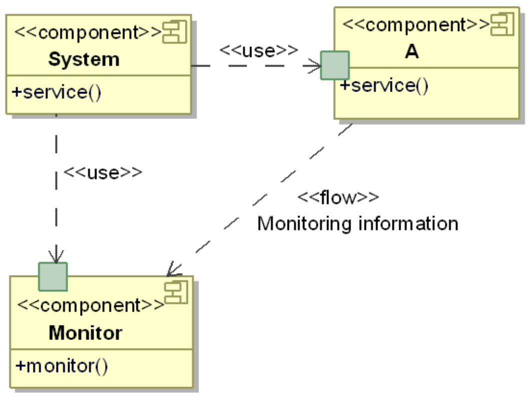
Monitoring can be performed using communicating software components, but could also be handled by intervening humans (e.g., using e-mails, telephone, letters, etc.). Therefore, the exact monitoring mechanism is not part of the solution. In general, however, monitoring an agent requires that some information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

Strategy



Strategy mapped to architecture

Structure



Behaviour

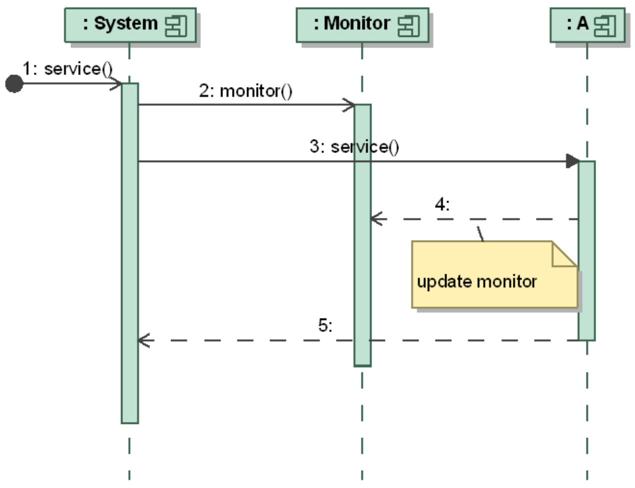


Figure 35 Use monitoring

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 4.5.1.1.

Change guidance

To implement this solution when the trust relationship changes, a monitor handler needs to be developed (see Figure 36). 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.

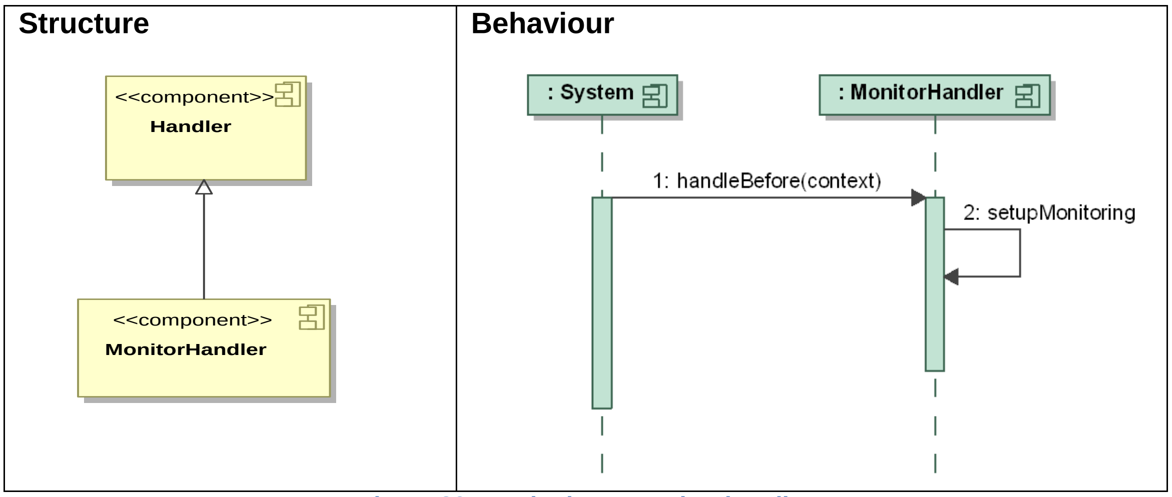


Figure 36 Developing a monitor handler

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 37.

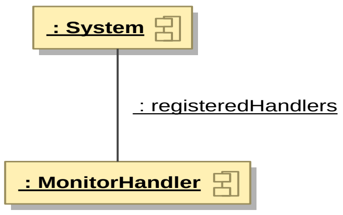


Figure 37 Register the monitor handler

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.

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

4.5.2 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 38.

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.

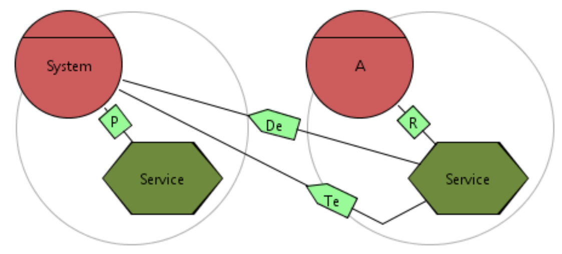
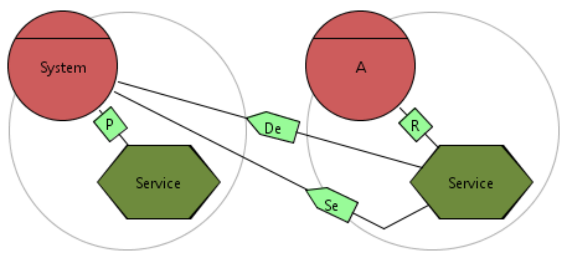
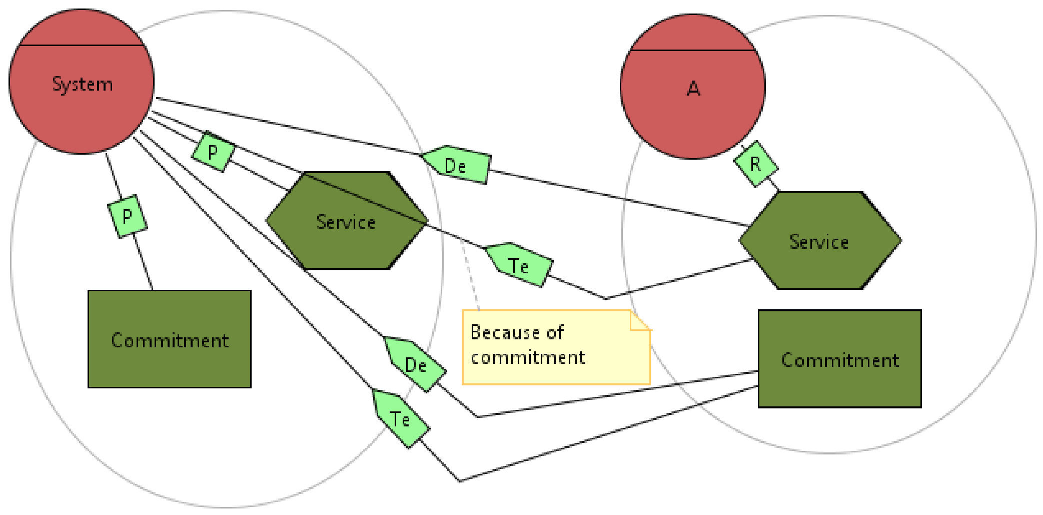
Before	After
	

Figure 38 Evolving trust of execution from external actor

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.

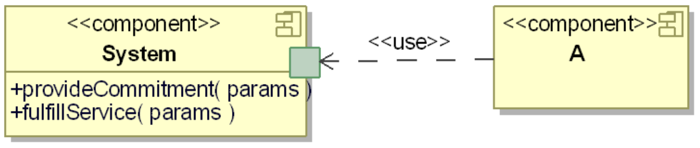
4.5.2.1 Solution 1: Provide commitment

The system can offer commitments for the services requested by other parties, as illustrated in Figure 39. 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.

Strategy
 System A P P De R Service Service Te Because of commitment Commitment Commitment De Te

Strategy mapped to architecture

Structure



Behavior

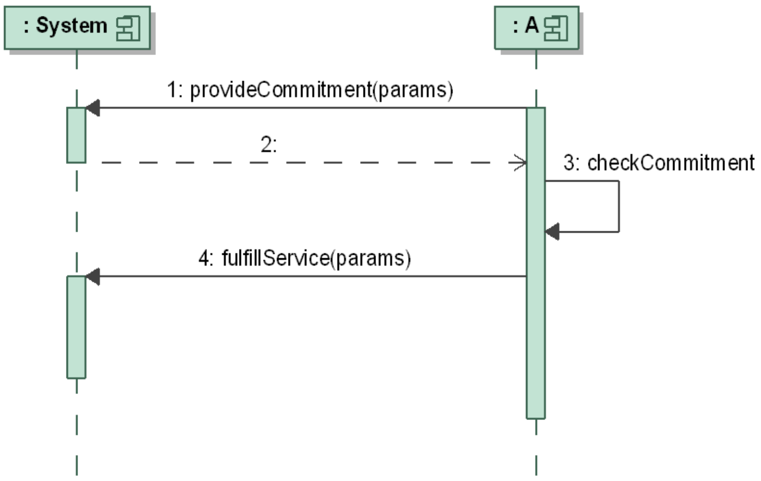


Figure 39 Providing a commitment

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 40.

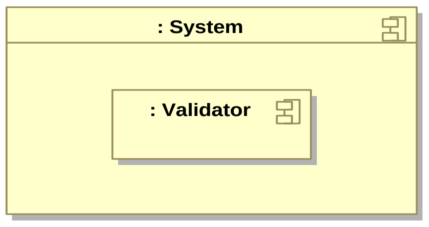
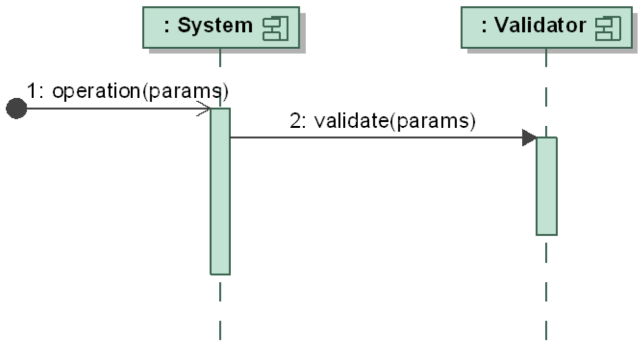
Architectural support pattern	
Structure 	**Behaviour**  : System : Validator 1: operation(params) 2: validate(params)

Figure 40 Validator architectural support pattern

Moreover, to digitally sign the commitment, the system should have access to a component offering cryptographic operations (Figure 41).

Structure

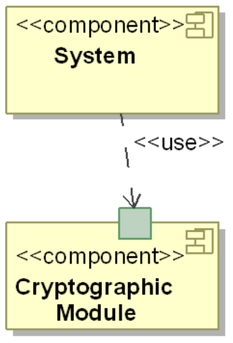
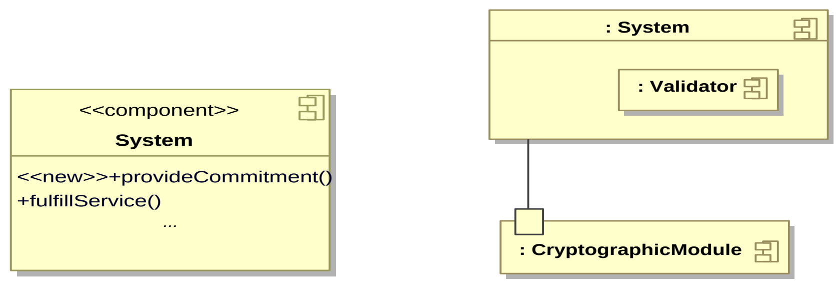


Figure 41 System accesses a cryptographic module

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 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 42.

Structure



Behaviour

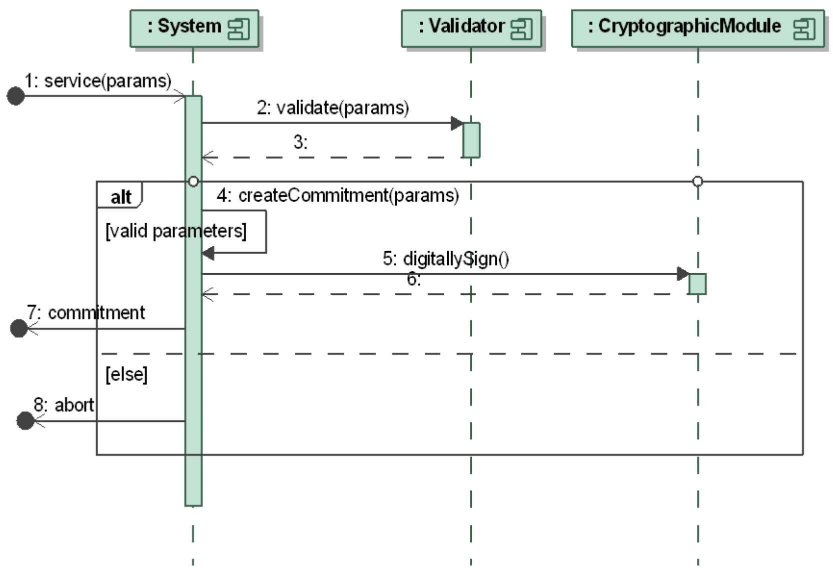
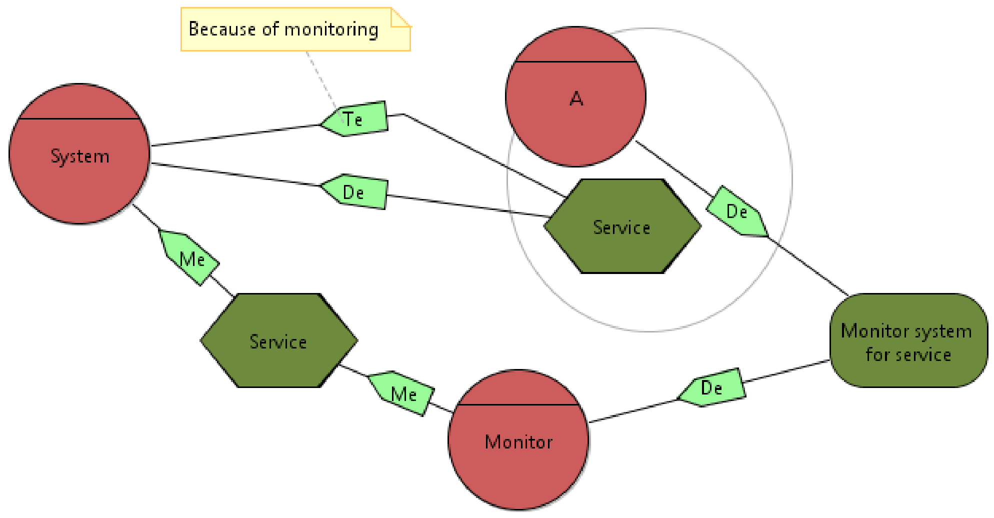


Figure 42 Applying the change guidance

4.5.2.2 Solution 2: Enable monitoring

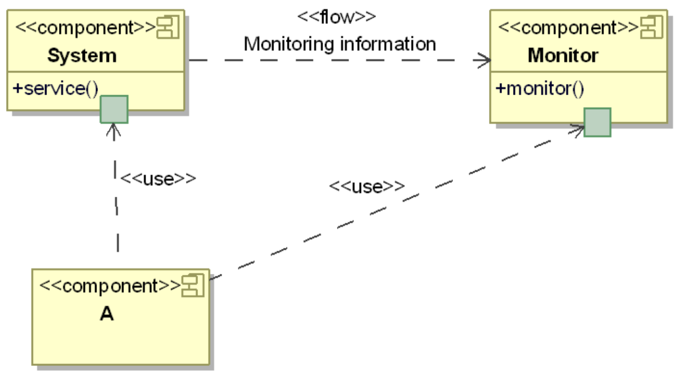
A second solution to restore the trust from the external party is to enable monitoring of the service execution (Figure 43). The external party delegates the task of monitoring the system to a monitor actor, which is then responsible for checking the fulfilment of the service by the system. Although 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).

Strategy



Strategy mapped to architecture

Structure



Behaviour

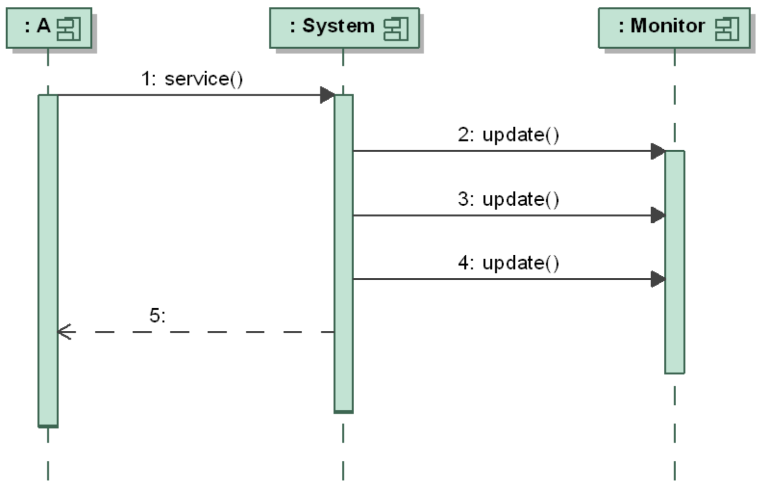


Figure 43 Enable monitoring

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 44), 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.

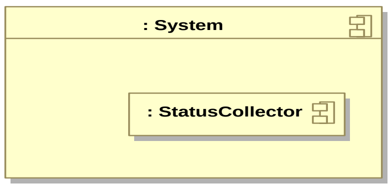
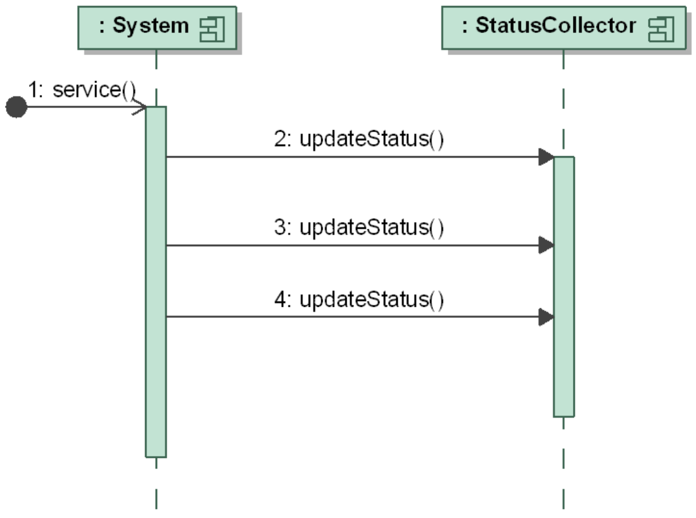
Architectural support pattern	
Structure	**Behaviour**
	 : System : StatusCollector 1: service() 2: updateStatus() 3: updateStatus() 4: updateStatus()

Figure 44 Status collector architectural support pattern

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 45). This could be achieved entirely using software, or the information could be made available to humans (e.g., displaying on a screen). Whatever the used mechanism, the information needs to be accessible to the outside of the system.

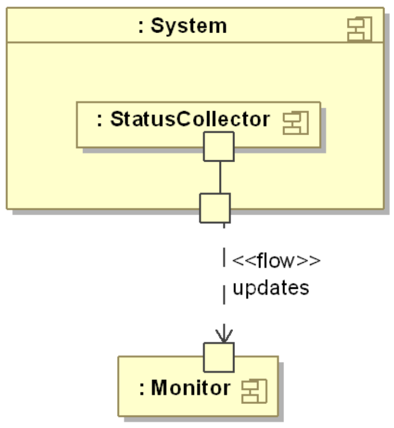


Figure 45 Enable monitoring using a status collector

4.5.3 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 46.

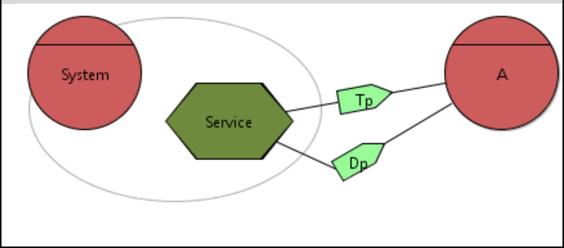
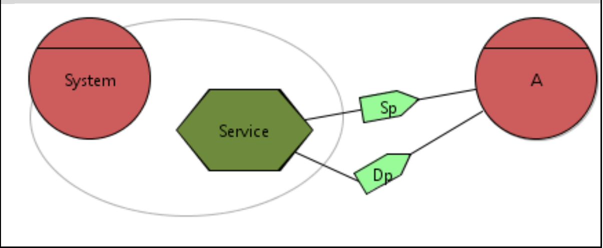
Before	After
	

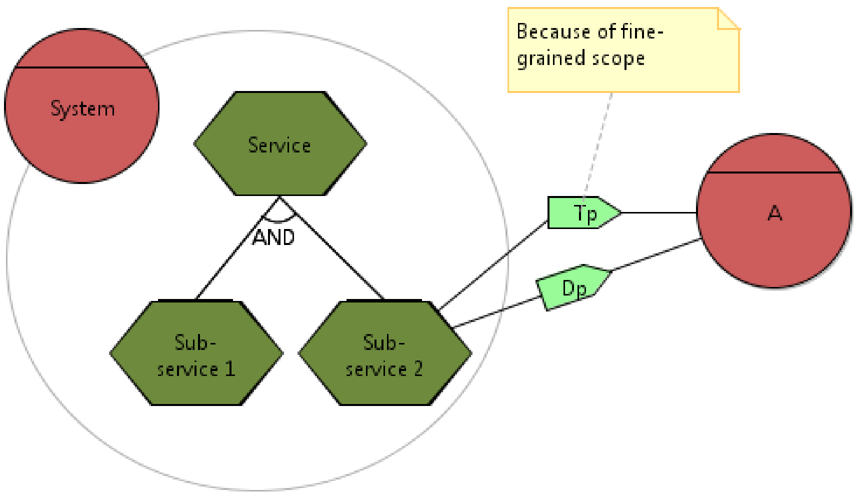
Figure 46 Evolving trust of permission upon external actor

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.

4.5.3.1 Solution 1: Apply least-privilege principle

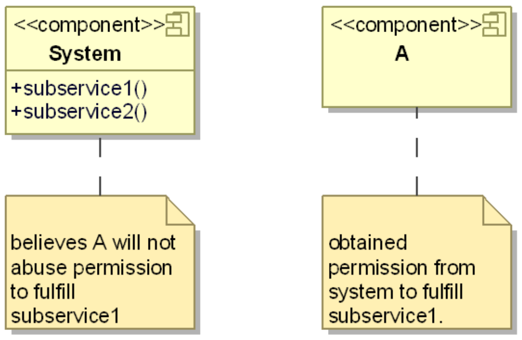
Instead of giving the external actor permission to the complete service, the permission can be made more fine-grained. If the service consists of (or can be split into) multiple sub-services, giving permission to execute a small subset of these may suffice; access to all other sub-services can be denied. This corresponds to applying the well-known principle of least privilege, and is shown in Figure 47.

Strategy



Strategy mapped to architecture

Structure


<<component>>
System
+subservice1()
+subservice2()
<<component>>
A
believes A will not abuse permission to fulfill subservice1
obtained permission from system to fulfill subservice1.

Behaviour

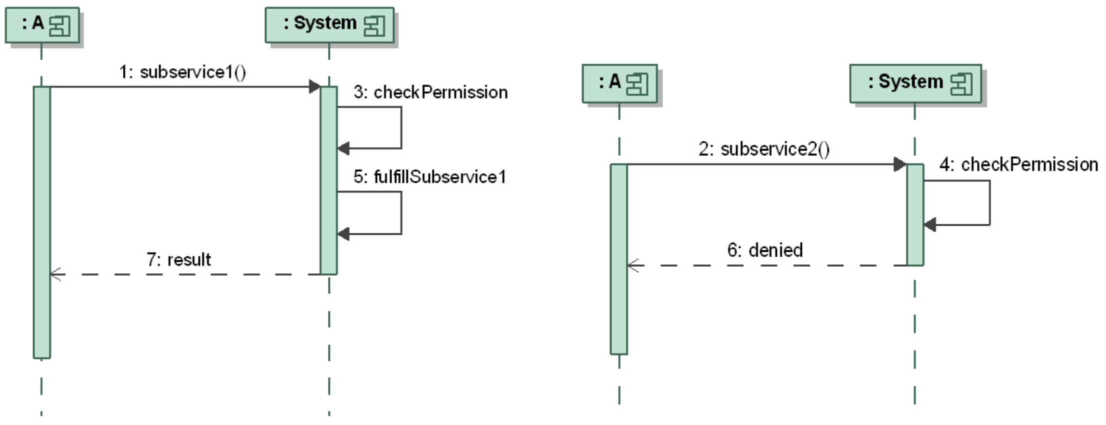


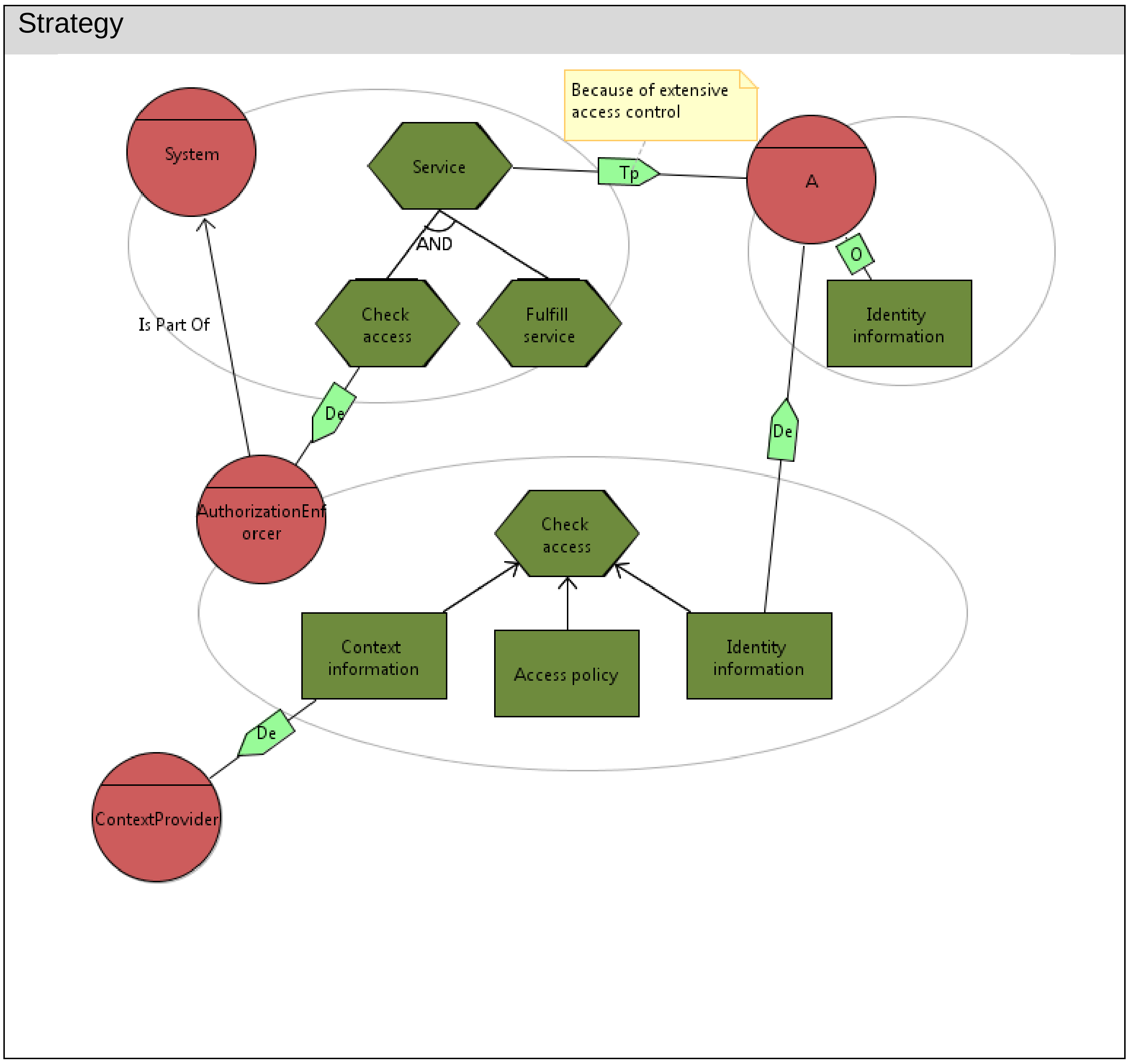
Figure 47 Apply least-privilege principle

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.

4.5.3.2 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 48). 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.).



Strategy mapped to architecture

Structure

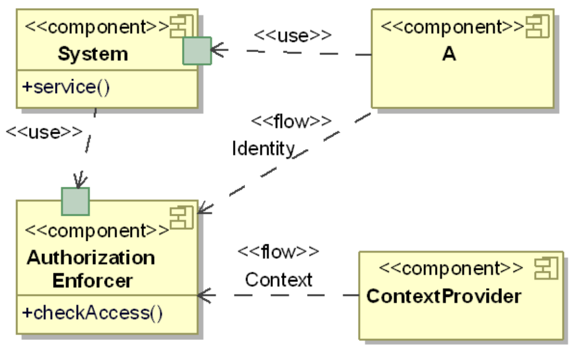


Figure 48 Attribute-based access control

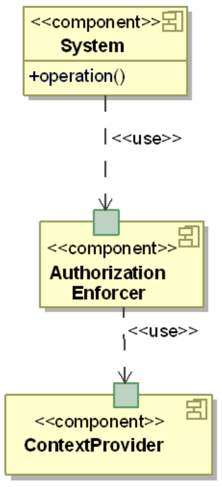
Example The nurse can only access the patient information of patients for who she is a designated nurse, and only within the nurse's working hours. A nurse has to identify herself before gaining access to patient information. The context information can be obtained from multiple systems: the designated 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.

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 49). The authorization enforcer will, before an operation is executed, check whether the subject that invoked the operation has permission to do this.

Architectural support pattern

Structure



Behaviour

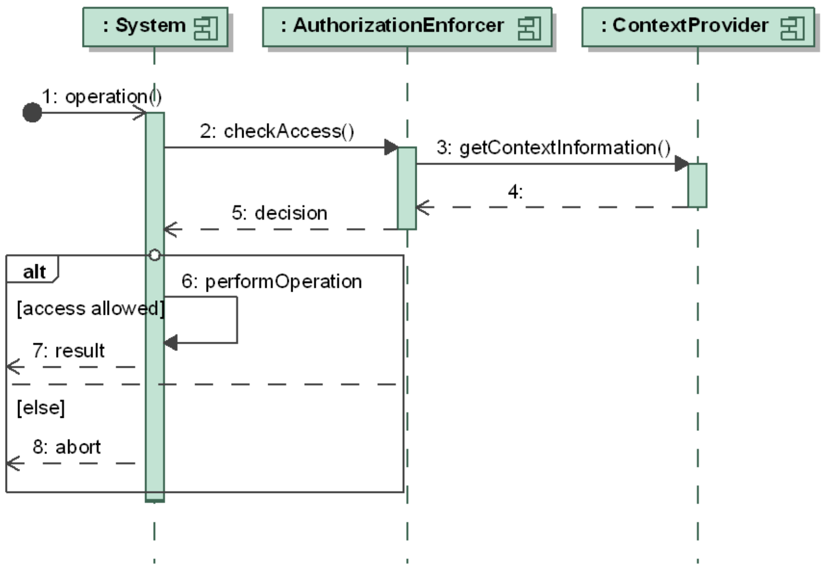


Figure 49 Authorization enforcer architectural support pattern

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 4.5.2.1. 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 50).

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.

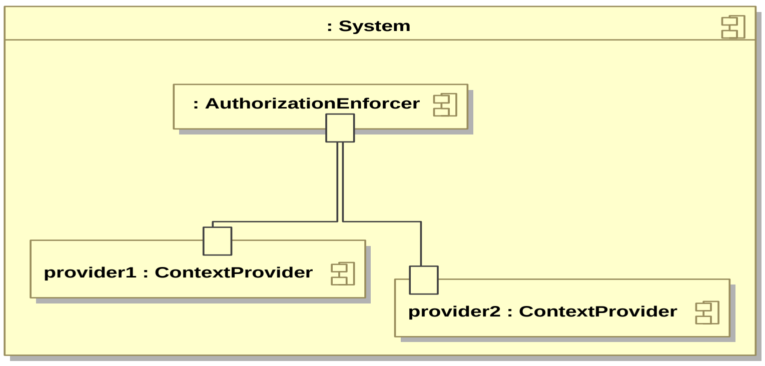


Figure 50 Applying the authorization enforcer support pattern

4.5.3.3 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 4.5.1.2.

4.5.4 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 51.

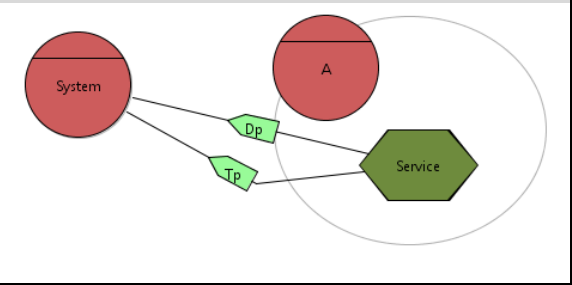
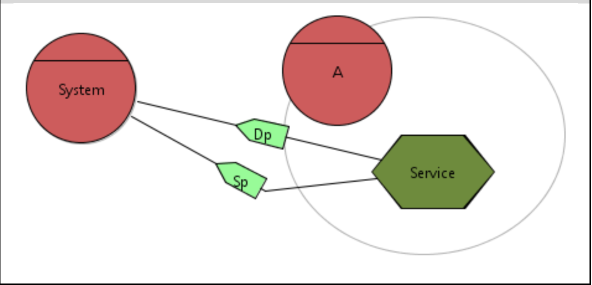
Before	After
	

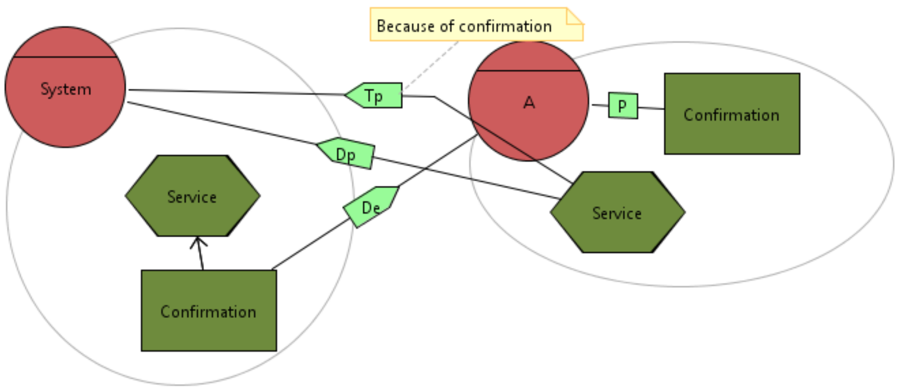
Figure 51 Evolving trust of permission from external actor

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.

4.5.4.1 Solution 1: Request confirmation

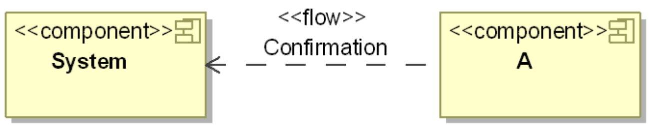
Each time the system performs a service, it needs to request a confirmation from the external party that the service may be fulfilled, as illustrated in Figure 52. This conformation can take various forms. For instance, it could be digitally signed evidence created by the external party whenever necessary. 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.

Strategy



Strategy mapped to architecture

Structure



Behaviour

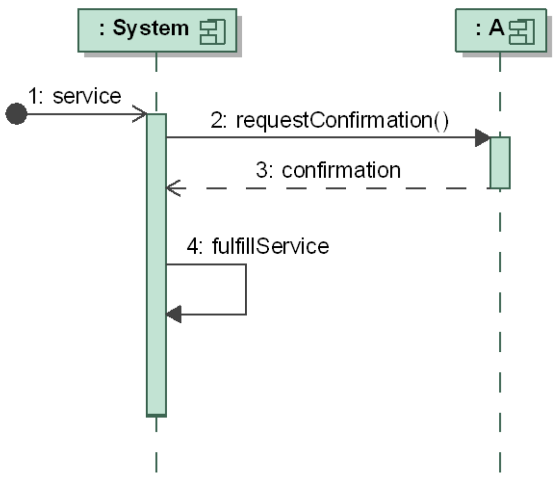


Figure 52 Request confirmation

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 4.5.1.1.

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 53). This confirmation handler will take care of obtaining and verifying the confirmation, before allowing the service to be fulfilled.

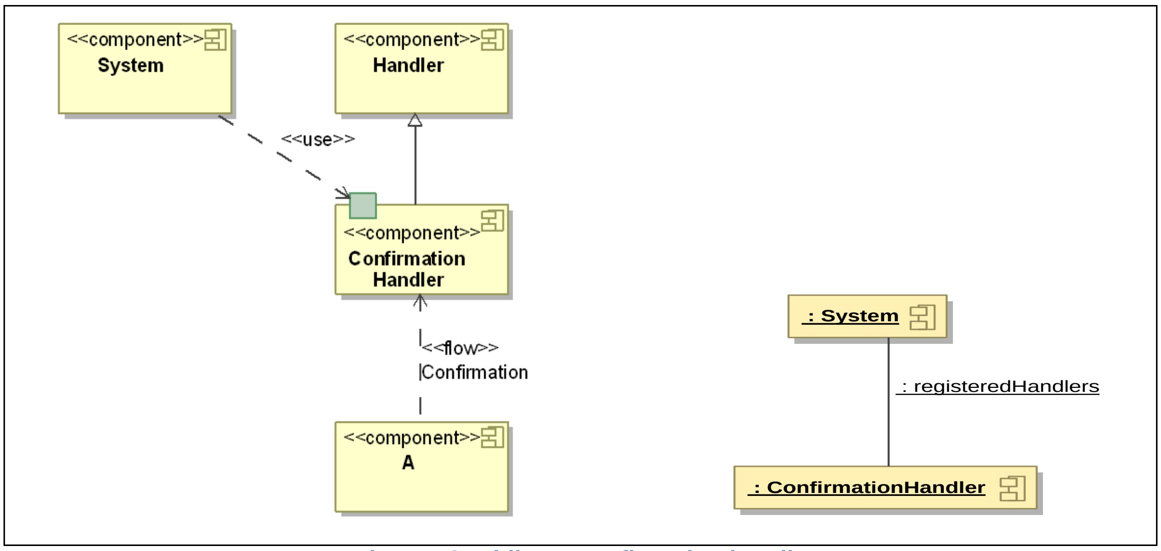


Figure 53 Adding a confirmation handler

4.5.4.2 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 4.5.2.2.

4.5.5 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 54. 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 4.5.1 needs to be considered together with this pattern.

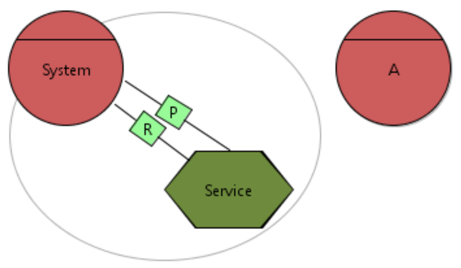
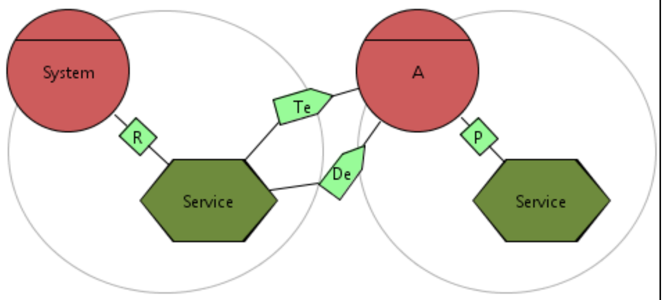
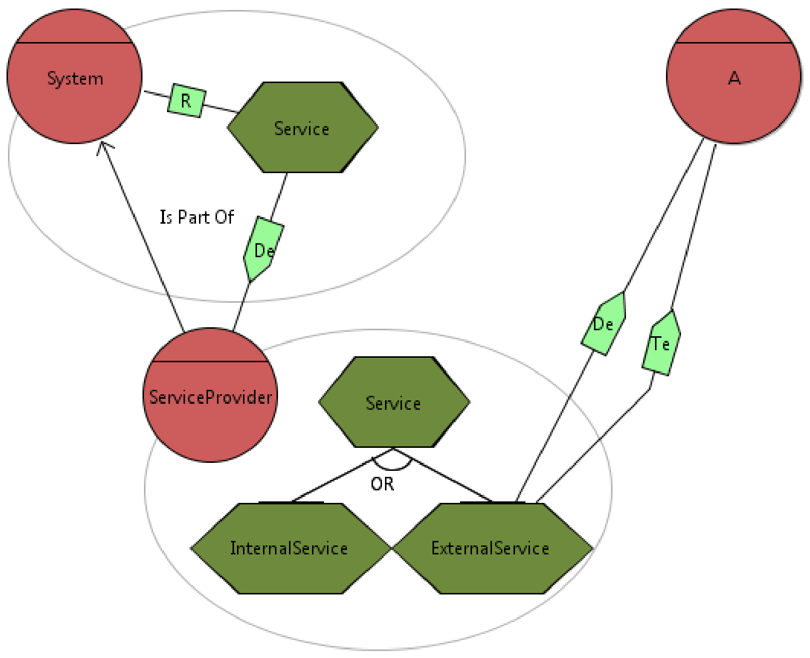
Before	After
	

Figure 54 Delegate execution of a service to a trusted actor

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.

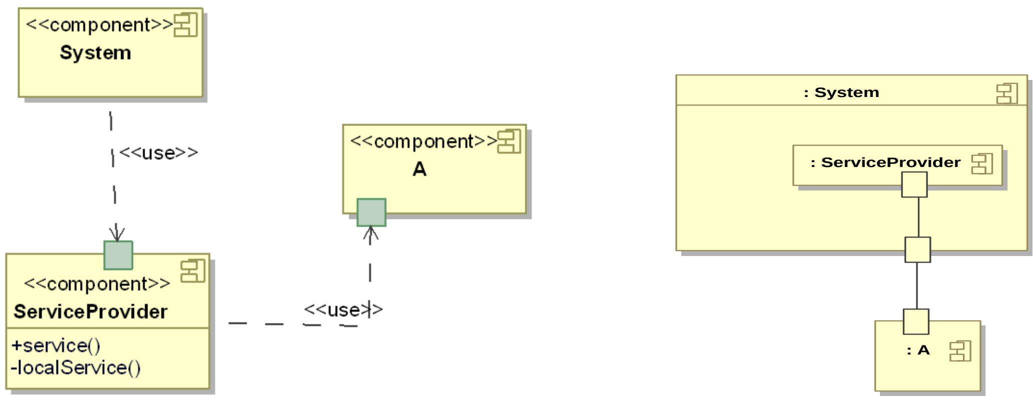
4.5.5.1 Solution: Encapsulate service

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 55. This is a well-known design solution for creating maintainable software solutions.

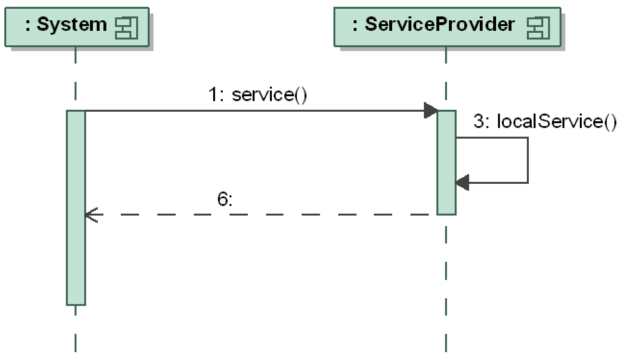
Strategy
 System R Service Is Part Of De A De Te ServiceProvider Service OR InternalService ExternalService

Strategy mapped to architecture

Structure



Behaviour



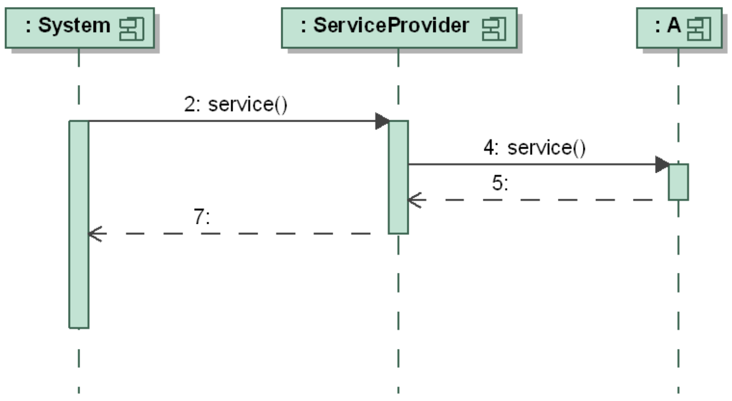


Figure 55 Encapsulate service

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.

4.5.6 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 56. 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 4.5.3 needs to be considered together with this pattern.

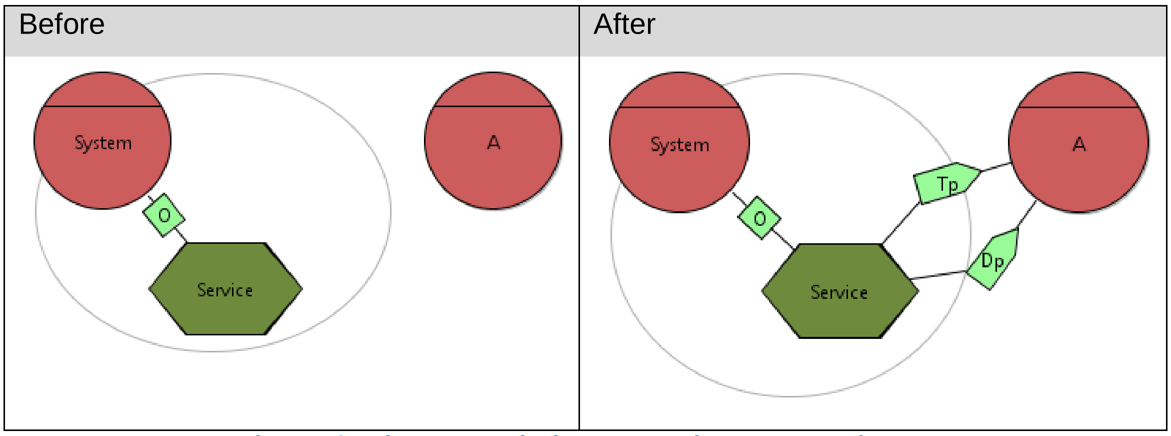


Figure 56 Delegate permission to a service to a trusted actor

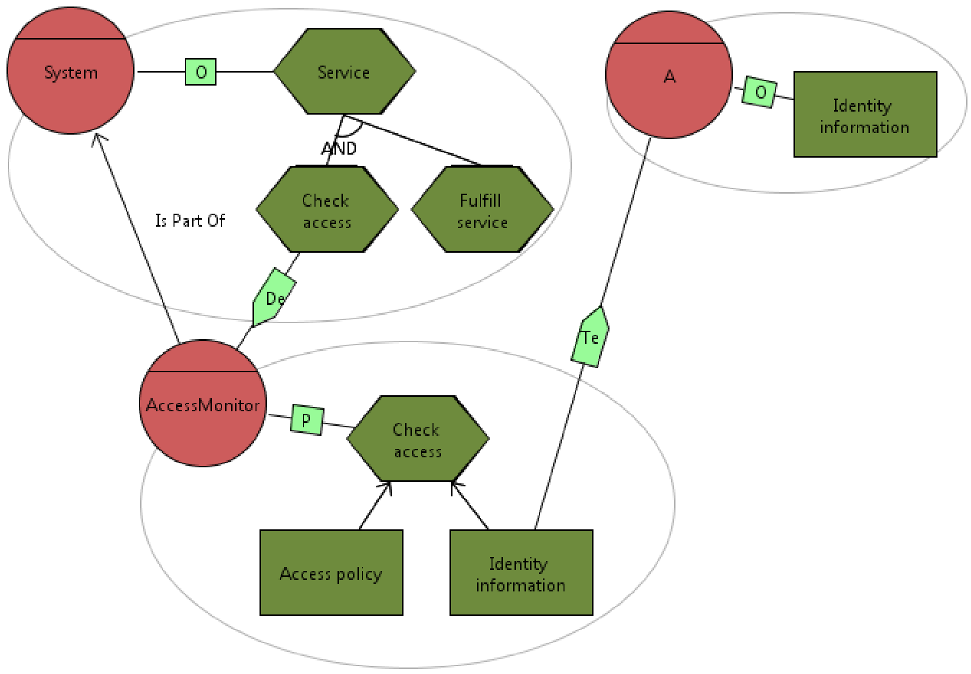
Example Suppose the system we are developing is a social network site. The system owns information about its users. Over time, the social network site wants to allow an advertising company to access this information, in order to deliver personalized advertisements. Of course, other external parties should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

Note that this scenario will often happen together with the scenario in Section 4.5.7.

4.5.6.1 Solution: Flexible access control

A solution for this evolution scenario is putting flexible access control in place (Figure 57). 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.

Strategy



Strategy mapped to architecture

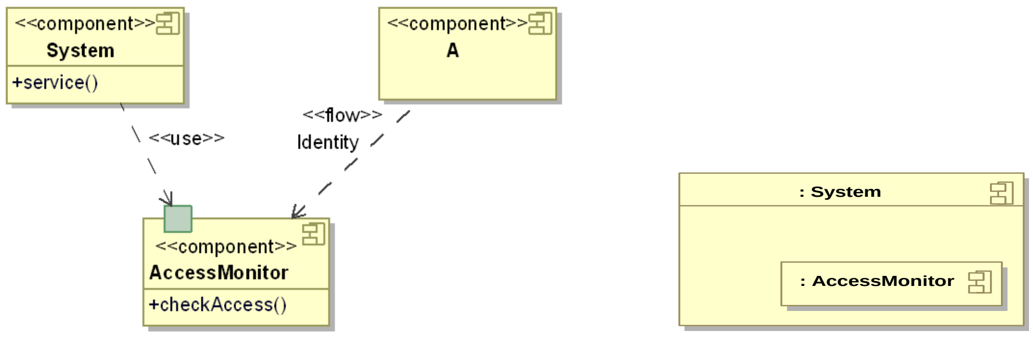


Figure 57 Flexible access control

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 4.5.3.2.

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.

4.5.7 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 58.

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 4.5.2 needs to be considered together with this pattern.

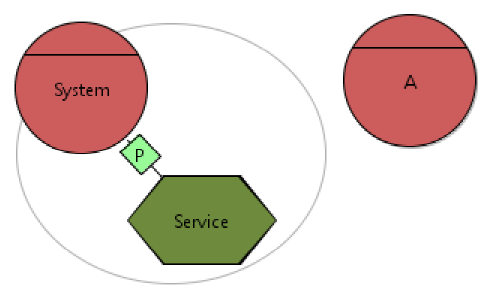
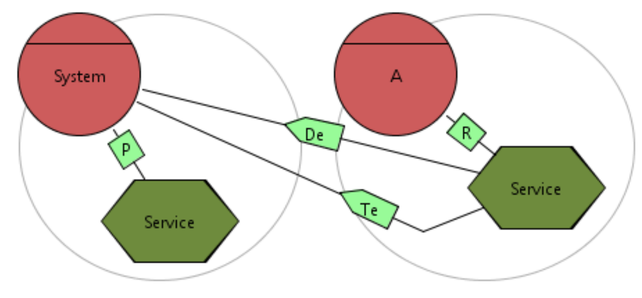
Before	After
	

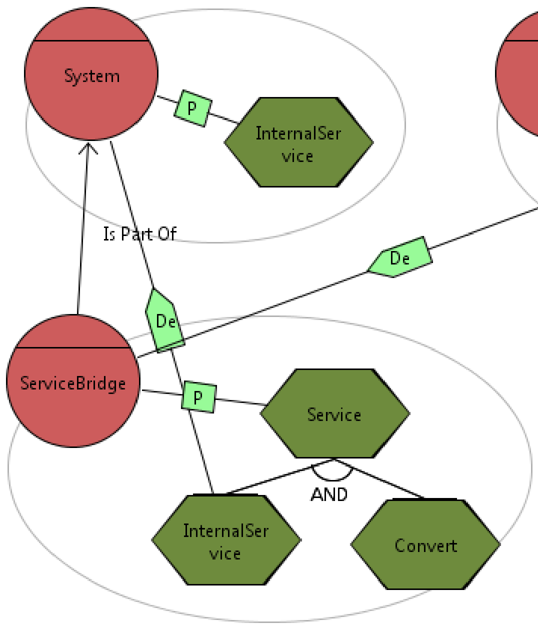
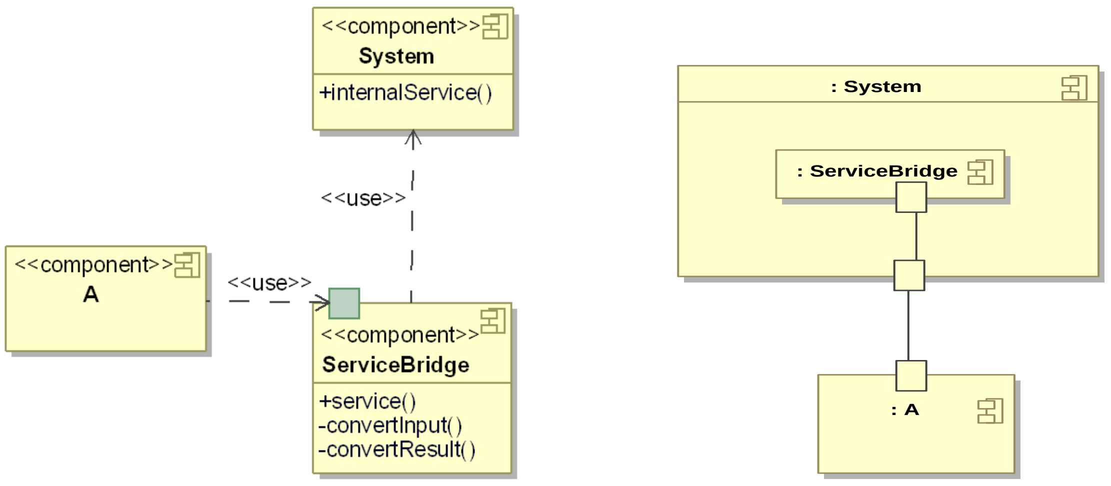
Figure 58 Providing additional service with delegated execution

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.

4.5.7.1 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 59.

This solution is a widely-known and common technique to create evolvable systems.

Strategy
 System P InternalSer vice A Service Is Part Of De De ServiceBridge P Service AND InternalSer vice Convert
Strategy mapped to architecture
Structure  <<component>> System +internalService() <<use>> <<component>> A <<use>> <<component>> ServiceBridge +service() -convertInput() -convertResult() : System : ServiceBridge : A

Behaviour

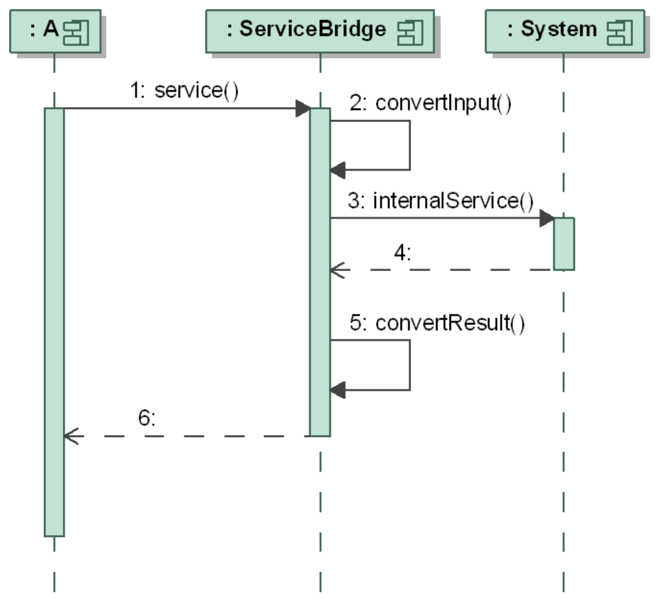


Figure 59 Introduce bridge component

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.

4.5.8 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 60.

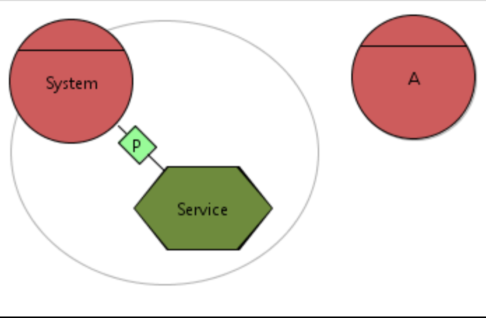
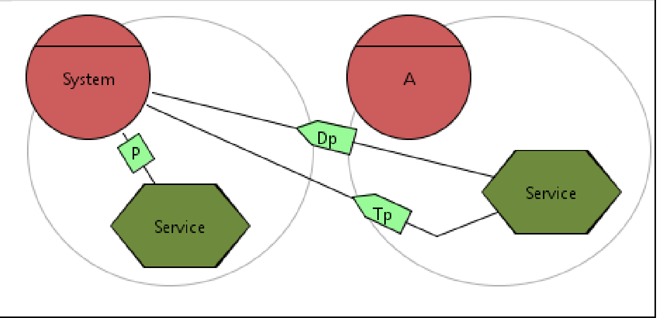
Before	After
	

Figure 60 Providing additional service with delegated permission

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. The transfer of ownership will manifest itself as an action outside the system (e.g., the signing of a contract).

This scenario may need to be considered together with the scenario described in Section 4.5.4, when the trust of permission from the external party may disappear over time.

4.6 Illustrations

4.6.1 Online shop

In this section, we will revisit the motivating example about the shop from the introduction. We will model the evolving trust of the scenario in SI* and show how an architecture can be designed so that it supports this evolution. The architecture is designed by applying the change patterns from the previous section. Then, it is shown how the architecture can deal with the evolution described in the example at the beginning of this section.

4.6.1.1 Initial situation

The initial situation is a situation in which all parties trust each other. More specifically, the client trusts the shop to a) correctly sell goods (e.g., ship the ordered goods when paid), and b) not to abuse submitted credit card information (e.g., charge more than agreed). The shop (maybe naively) trusts the client to provide a purchase acknowledge in case of a dispute, i.e., the shop trusts its clients never to repudiate a placed order. This situation is depicted in the SI* diagram in Figure 61.

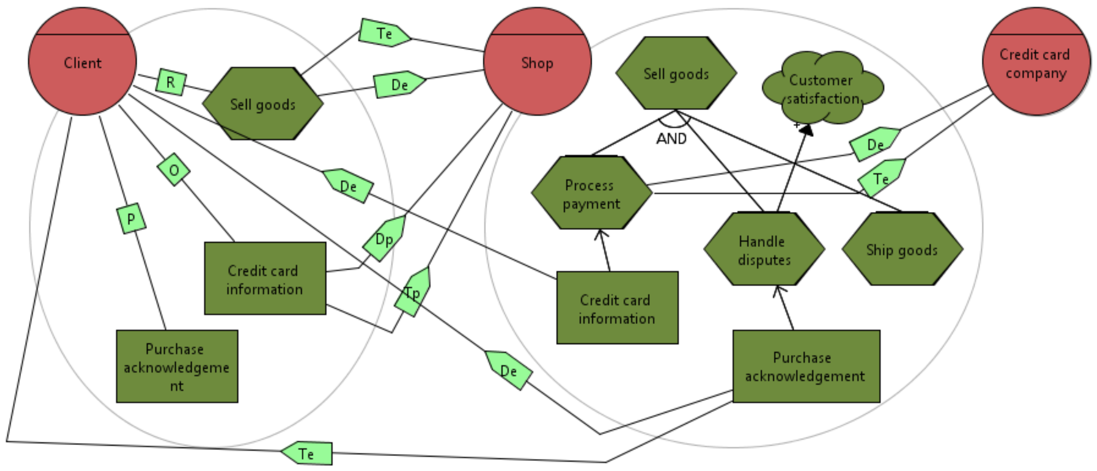


Figure 61 Simplified SI* diagram for online shop

We can create a corresponding architecture by applying the mappings from Section 4.4.2. The result is depicted in Figure 62.

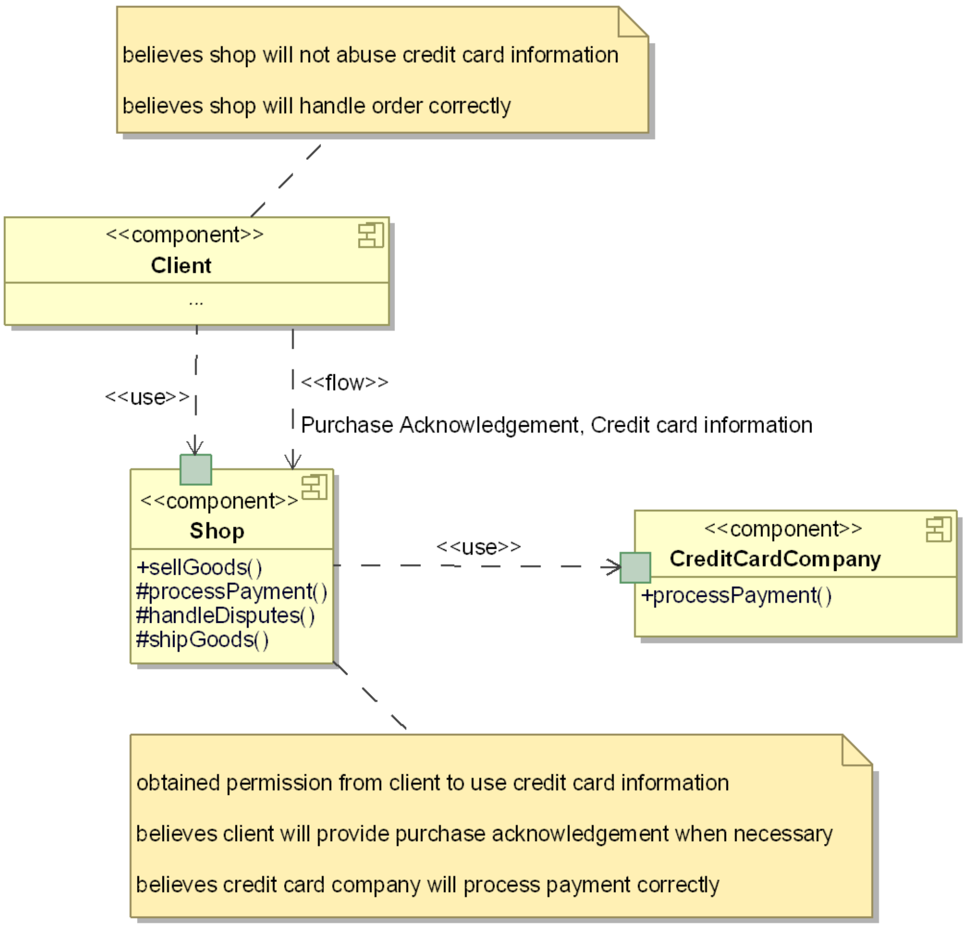

believes shop will not abuse credit card information
believes shop will handle order correctly
<<component>>
Client
...
<<use>>
<<flow>>
Purchase Acknowledgement, Credit card information
<<component>>
Shop
+sellGoods()
#processPayment()
#handleDisputes()
#shipGoods()
<<use>>
<<component>>
CreditCardCompany
+processPayment()
obtained permission from client to use credit card information
believes client will provide purchase acknowledgement when necessary
believes credit card company will process payment correctly

Figure 62 Corresponding architecture

4.6.1.2 Applying change patterns

To design the architecture for the shop we start from the initial SI* diagram in Figure 61. We will go through each change scenario from the catalogue and search for possible matches. Finding a match means that we have identified a possible future change in a trust relationship in the system. The stakeholders need to identify and select the likely and important matches, which the architect then needs to incorporate in the architecture, by using one of the solutions provided by the pattern catalogue.

The iteration over the cases and possible matches are presented in Table 6. The first row in the table is obtained as follows. The change pattern in Section 4.5.1 describes a situation in which trust 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 '×' were selected as likely and important scenarios by the stakeholders, and will need to be resolved in the architecture.

4.5.1 - 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
4.5.2 - 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
4.5.3 - Evolving trust of permission upon external actor	
	No matched identified

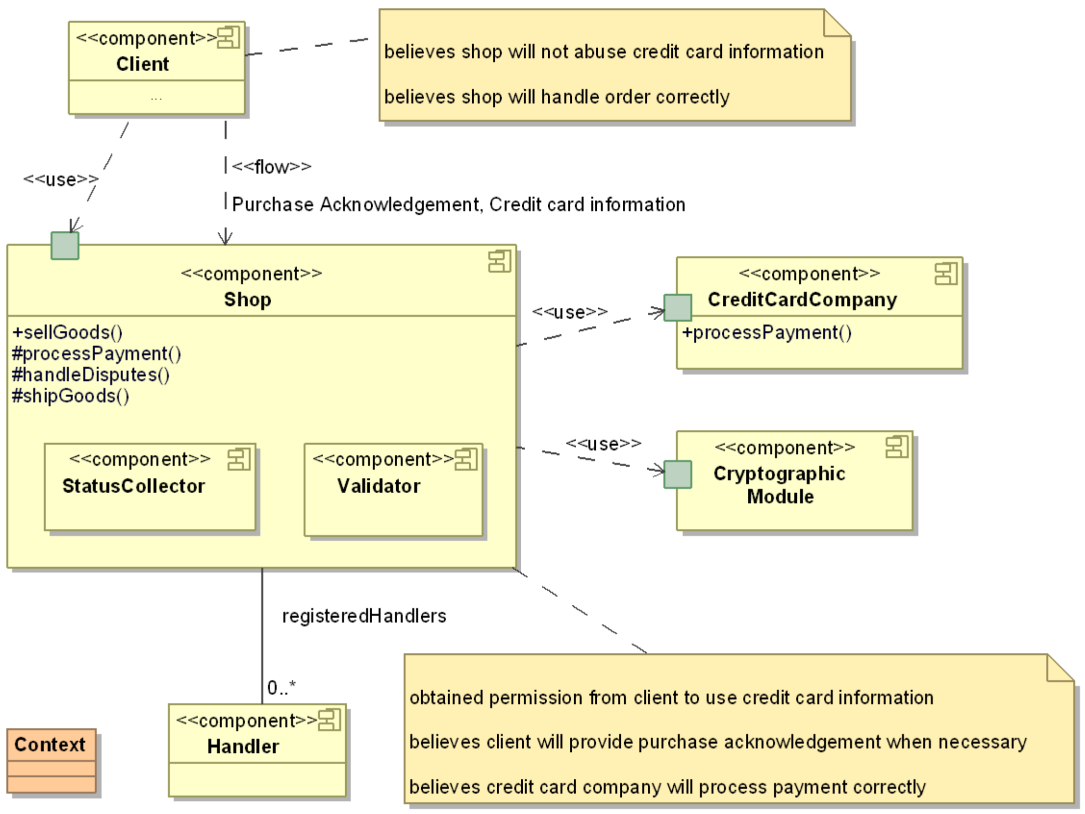
4.5.4 - Evolving trust of permission from external actor	
✕	**Current situation** Client trusts shop not to abuse credit card information **Possible change** Shop is no longer trusted with credit card information **Likelihood** High
4.5.5 - Delegate execution of a service to a trusted actor	
	Current situation Shop takes care of order shipment itself **Possible change** Shipping of orders is outsourced to dedicated shipping company **Likelihood** Low
4.5.6 - 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
4.5.7 - 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
4.5.8 - Providing additional service with delegated permission	
	No match identified

Table 6 Result of following the change pattern process

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 4.5.1.1).
- The second evolution scenario (where the shop is no longer trusted to correctly handle orders), the solution in Section 4.5.2.1 is chosen (providing a commitment).
- For the third evolution scenario, the solution based on monitoring (Section 4.5.4.2) 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 63 (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 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.



<<component>>
Client
...
believes shop will not abuse credit card information
believes shop will handle order correctly
<<use>>
<<flow>>
Purchase Acknowledgement, Credit card information
<<component>>
Shop
+sellGoods()
#processPayment()
#handleDisputes()
#shipGoods()
<<component>>
StatusCollector
<<component>>
Validator
<<use>>
<<component>>
CreditCardCompany
+processPayment()
<<use>>
<<component>>
Cryptographic Module
registeredHandlers
0..*
Context
<<component>>
Handler
obtained permission from client to use credit card information
believes client will provide purchase acknowledgement when necessary
believes credit card company will process payment correctly

Figure 63 Resulting architecture

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.

4.6.1.3 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 4.5.1 (Evolving trust of execution upon external actor), for which the first solution was chosen, the change guidance from Section 4.5.1.1 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 architecture, namely “Evolving trust of permission from external actor” (Section 4.5.4). Here, the architect has chosen the solution of enabling monitoring (Section 4.5.4.2). 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 4.5.2.1) 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.

4.6.2 HOMES

In the second year of the project, the HOMES case study will be used to validate the process of using change patterns to design an evolvable architecture. In particular, we will focus on the part of the HOMES case study that is concerned with the access control and policy enforcement for devices on the network. Especially the Home Gateway component will play an important role in this validation.

4.6.2.1 Challenges

When performing the validation on the HOMES case study, the following challenges are expected to be encountered.

1. The change pattern process assumes a trust model of the application under development is available. In the current case study, however, no explicit and complete list of trust assumptions is available. The validation exercise will thus need to provide a list of these trust assumptions, and can be used to assess the

feasibility of one of the requirement of the process (i.e., working with an explicit trust model) in an industrial context.

2. The outlined process requires the stakeholders of the system to decide on the likelihood, and eventually the inclusion of a solution in the architecture, of an evolution scenario. It is currently unclear in the case study in which evolution scenarios (related to trust assumptions) are likely to happen and should be foreseen in the architecture. The validation will assess how the outlined process performs with respect to identifying and classifying these evolution scenarios.
3. The presented catalogue of trust change patterns is not complete. Change scenarios may be missing, or solutions may not yet be described. Performing the validation on the case study will give an indication about the usability of the current catalogue, and may identify missing patterns and solutions.

4.6.2.2 Approach

For the validation, we plan to perform the following steps in close cooperation with the industry partner.

First, we model the situation of the case study as it is envisaged to be built, with no specific attention for evolving trust requirements. This model comprises both an architectural model in UML, and a corresponding trust model in SI*.

Then, the process as outlined in Section 4.3 needs to be executed. This activity will identify various possible change scenarios related to trust, which have to be assessed with respect to their likelihood and importance by the stakeholders from the case study.

Finally, the relevant change patterns need to be applied to the architecture. Also, we will need to verify with the industrial partner whether the resulting architecture is realistic and useful.

4.7 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.

5 Security as a service

In this Section we define an architectural blueprint that transposes the model of Software as a Service to the security domain and thereby realizes Security as a Service (SeAAS). The proposed architecture is robust in the sense that it is flexible enough to cope with a broad variety of changes and thus supports long-lived, evolving systems. Based on the architectural blueprint we define a reference architecture leveraging the principles of Service Oriented Architectures (SOA) and Web Services technologies.

In the context of the overall deliverable D 2.1, the proposed reference architecture represents a possible target infrastructure for the change-driven security engineering process as defined in Section 3.1, where architectural components and services map to the *System Operator*'s view. The architecture can also serve as a target infrastructure accommodating architectural solutions as proposed by the *Catalogue of Change Patterns* presented in Section 4.

We structured this section as follows. Section 5.1 opens with a brief problem statement linked to the current practice of enforcing security in service oriented systems exclusively at the service endpoint. Section 5.2 introduces a motivating use case from healthcare that will serve as a first running example for illustration purposes. The example will be replaced by a scenario from the HOMES case study in years two and three. In Section 5.3 we analyze the limitations of endpoint security and make the case for an architectural blueprint leveraging the paradigm of Security as a Service to support long-lived, evolving systems. We present the architectural blueprint for Security as a Service in Section 5.4 and elaborate a specific reference architecture based on the principles of SOA and Web services technologies in Section 5.5 followed by a brief discussion in Section 5.6. Section 5.7 introduces the HOMES case study. The case study will be used to validate the general idea of Security as a Service in context of the change-driven security engineering process as defined in Section 3.1 in the years two and three.

5.1 Introduction

Inter-organizational workflows spanning multiple domains of business partners involve the sharing of sensitive resources. The examples are numerous and ever-growing. In healthcare, a patient's electronic health-record stored with a hospital may be updated with a radiography produced by an external specialist, complemented with a diagnosis together with a regular update of the medication prescribed by the patient's practitioner. Or a company's financial statement may be forwarded to its auditors before being turned in as an electronic tax declaration with financial authorities (e.g., [53, 54]). Those large-scale software systems can be characterized as heterogeneous, distributed systems spanning across many enterprises under the control of as many "ownership domains". Often, they are realized based on the blueprint of Service Oriented Architecture (SOA). However, the decentralized security models and distributed infrastructures of SOA turn the enforcement of security requirements into a major challenge.

Technical interoperability was addressed first and with some success. To make sure companies were using “compatible” technology for cooperation with their peers, software engineers and architects could turn to the paradigm of SOA with its standardized technical underpinning, the stack of Web services standards and technologies. However, up until now, the standards only address basic security requirements (the triad of traditional information security, namely confidentiality, integrity and availability) and resolve issues at a low, technical level. This makes security engineering incredibly complex and – as a consequence – implementations error-prone.

According to current practice, security infrastructures enforce security exclusively at the service endpoint. They ignore the peculiarities of SOA’s decentralized peer-to-peer architecture which outmodes traditional security solutions and mechanisms, among them the concept of perimeter security and the centralized security models (cf to [69, 70] for interesting discussions). Besides placing a significant processing burden on service nodes, endpoint security renders maintenance and management of the distributed security infrastructures cumbersome, and impedes interoperability with external service providers and requesters. To meet these challenges, we propose a reference security architecture that transposes the model of Software as a Service to the security domain and thereby realizes Security as a Service (SeAAS). The proposed architecture goes beyond the mere bundling of security functionality within one security domain as it realizes complex security requirements for processes involving two or more domains.

The reference architecture complements the SECTET framework for model-driven security engineering [53]. The framework will serve as starting point for the elaboration of an approach for the high-level configuration of security-critical systems as planned in the description of work under T2.2 as part of the second year.

5.2 Motivating example

Our scenario draws the security requirements from use cases of the healthcare industry. They were elaborated in the context of national initiatives in Europe with the aim to realize the Electronic Health Record (EHR) [44, 51, 53]. We begin with a functional description in Section 5.2.1 and proceed to security requirements in Section 5.2.2.

5.2.1 The Electronic health record - a use case

Figure 64 shows the various stakeholders modeled as roles and their interactions with an EHR system modeled as message exchanges in a typical scenario. Security-relevant communication is indicated in red.

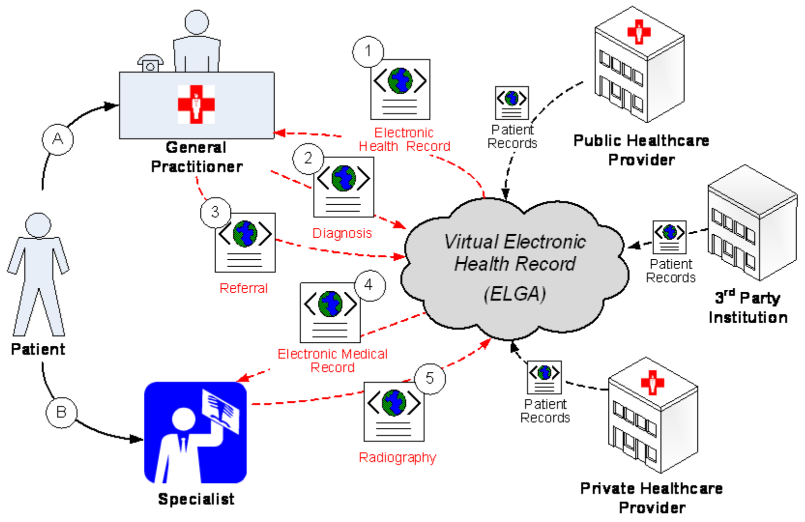


Figure 64: Roles and Message Exchanges in a Distributed Healthcare Scenario

Markus Maier (role Patient) goes to see Dr. David Daum, his family doctor (role General Practitioner) for his yearly medical check-up. In step 1, Dr. Daum accesses Markus Maier's Electronic Health Record over the centralized EHR service (ELGA2). In its essence, an EHR represents a consolidated virtual medical record assembled from information distributed across various healthcare providers, which produced clinical information during past consultations and treatments. In Markus Maier's case these were the *City Hospital* and the *City Sanatorium* as Public- and Private Healthcare Provider respectively and the *City Health Insurance* as a 3rd Party Institution. After a first examination, Dr. Daum decides to refer Markus Maier to the radiologist Dr. Rudolf (role Specialist). He does so by issuing an electronic Referral which updates the EHR (step 3). In consultation with his patient, Dr. Rudolf accesses Markus Maier's EHR (step 4), and updates his EHR with the produced Radiography (step 5). Afterwards, Markus Maier may have to submit to a couple of further checks (not shown here), e.g., have blood samples checked by a medical laboratory, and submit to a stress electrocardiogram with an internal specialist etc., before he pays his final visit to Dr. Daum for a discussion of the medical statement. On this occasion, Dr. Daum updates Markus Maier's EHR with the Medical Statement.

5.2.2 Security requirements

Based on that common scenario, we can identify a broad array of security requirements. Sections 5.4 and 5.5 discuss in depth how our architecture realizes these requirements. Details are illustrated by taking Non-repudiation as an example for a complex security requirement.

Authentication (And Identity Management). The EHR infrastructure facilitates the identification, registration and authentication of professional users – (be they humans or services) – based on digital certificates and public key technology. Users access medical information in the EHR based on role credentials issued by ELGA Certification Authorities. The credentials are valid across the security domains of involved stakeholders in the overall scenario. Nevertheless, a single health organization is very likely to manage identities within its own security domain running a "local" certification or registration authority. The local identities of users and applications which interoperate with the EHR system are mapped to "global" identities managed by the

ELGA Certification Authorities. For example, although Dr. Daum may have authenticated himself to his local application, he would have to authenticate himself a second time with his global credentials to the EHR system to access his patient's records via ELGA.

Authorization. Role credentials define the permissions to access health records. Realizing the principle of Least Privilege, users are given those privileges necessary to perform their job as specified by system roles (e.g., Specialist, Healthcare Provider etc.). This entails the necessity for a fine grained protection of the resource. For example, the role Pharmacist only needs access to those parts of the EHR containing the prescription of medication. In the EHR System, there is a default Permission-Role assignment that may be overwritten by the record owner. Although a user holding the role General Practitioner may be given the most comprehensive access to his patient's medical records (if he is the primary care physician), a Patient may confine his privileges. Markus Maier may not want his father-in-law working as a psychiatrist in the City Hospital to see medical records about a psychotherapeutic treatment he had to undergo a couple of years ago due to a mental problem. So he could define a *Negative Access Permission* to these records for his father-in-law. Other complex authorization policies that come into play in standard use cases are the *Delegation of Rights*, *Four-Eyes-Access-Control*, *Break-Glass-Policy*, and *Dynamic Access Control*. A comprehensive treatment on the modeling and enforcement of complex access control policies in healthcare with the SECTET framework, can be found in [51] and [53].

Non-repudiation. This requirement aims at preventing parties in a communication from falsely denying having taken part in that communication. In our context, this is a security requirement whose enforcement is typically transparent to users [39]. It comes in two flavors. *Non-repudiation of Reception* requires the addressee to return a proof of receipt (e.g., a signed message carrying a time-stamp) to the sender to be kept in case dispute resolution is needed. In our scenario, Dr. Daum and Dr. Rudolf will both get a proof of receipt from the EHR system after having updated Markus Maier's EHR with the produced documents and artefacts (Referral, Radiography, and Medical Statement). Complementarily, *Non-repudiation of Origin* requires the sender to produce a proof of submission and make it accessible to the receiver. The EHR system will log the updates to Markus Maier's EHR. The security infrastructure takes care of producing and consuming the messages and initiating logging activities.

Security Compliance and Governance. Security compliance aims at the detection of deviation from allowed behavior, specified interaction patterns, or message structures. In the current state of the SECTET framework, we view security compliance in its narrowest sense. It defines the adherence of messages to predefined structures or interaction patterns based on supported security infrastructures and mechanisms (e.g., type of tokens, encryption, signature algorithms, request-reply, one way etc.). It is enforced by the security infrastructure and is offered as a service to local applications and users.

5.3 Security as a service - making the case

In this subsection, we motivate the need for *Security as a Service* (SeAAS) as a paradigm for security architectures and present related work as we pursue our line of argument.

5.3.1 Limitations of endpoint security

According to current best practice, Web service security is mostly enforced at the physical node providing the service or the component proxying the service (henceforth, we will call the service and / or its proxy component *service endpoint*). In a typical Web services based request, the service endpoint applies basic cryptographic processing to inbound and outbound messages leveraging XML based standards [57, 42, 47]. It extracts and validates tokens of incoming messages, decrypts encrypted parts, validates signatures etc. Outbound messages are processed and their structure extended so to comply with policy requirements as imposed by the service endpoint's communication peer.

Traditional endpoint security falls short on two fundamental issues of large-scale business solutions. The first issue is related to the complexity of security engineering. Web services standards and technologies are constantly evolving. New security standards are added to the stack of Web services standards to cover new requirements and use cases (c.f. [58] for an overview). This fast moving target is a challenge to security experts and software engineers alike. Traditional methods of software engineering can hardly cope with the plethora of standards combined with the complexity of security solutions needed for the realization of enterprise-wide and inter-organizational business solutions. This is often considered to be a major obstacle to the rapid adoption of Web services as a reference platform to large-scale solutions. Another issue is related to the consistent enforcement of security policies in enterprise-level solutions. These environments are characterized as large-scale distributed architectures with thousands of services deployed on hundreds of endpoints and possibly as many internal and external consumers. Nevertheless, any access decision has to be attributable to security policies that meet the obligations imposed by laws and regulations like the Sarbanes-Oxley Act and complying corporate governance policies. This necessitates policy management concepts and security mechanisms that guarantee consistent enforcement of security policies in distributed, heterogeneous environments.

5.3.2 Declarative security

The concept of declarative security was a first step to cope with the issues exposed in the previous section. It addresses three challenges. *1. Development.* Security concerns are separated out of actual application and service development. The burden of security enforcement is shifted from service developers and service requesters to security experts who codify security requirements into policies based on rules. This simplifies the realization of security-critical use cases. XACML is an example standard for declarative access control [75]. It proposes a declarative access control policy language implemented in XML and an architectural blueprint for the communication between infrastructural components. *2. Interoperability.* Security requirements on message structure and syntax are codified as rules in the machine readable XML standard WS-Security Policy [41] and advertised to potential service requesters. This improves interoperability of security solutions that cross organizational boundaries. *3. Policy Management.* Security policies expressed in declarative statements can be checked for consistency and – once consolidated – distributed to the application which is meant to enforce them. This fosters the consistent application of policies across all solutions in an enterprise.

Even with declarative security, the realization of security-critical inter-organizational scenarios still faces two major hurdles. For one, with enforcement left to the endpoint, security solutions are scattered over the service landscape. This means that in order to keep up to date with evolving technologies and changing security requirements that demand new functionality, security engineers have to propagate the changes to every single endpoint – a very inefficient form of reusability. Secondly, the state of the art in Web service and SOA security technology indicates that for the moment only very basic security requirements like those based on the application of cryptographic operations are realized. This is actually the main criticism advanced by industries that are primarily concerned about complex security requirements in an inter-organizational setting like in healthcare and e-government. Here, use cases have to accommodate security requirements derived from complex industry regulations and laws (cf. Section 5.2.2).

Existing standards and specifications do not address these security requirements at all. The reason is, that their realization would overstrain the capacities of a single endpoint, either in terms of the complexity of the underlying security concept (e.g., the protocols of non-repudiation), or in terms of the processing power (e.g., evaluation of log-files for security monitoring), or in terms of functionality (e.g., basically stateless service endpoints are not supposed to have all information necessary to infer unusual user behavior for fraud detection). A very technical account on how to realize declarative security that covers basic security for Web services (authentication, confidentiality, integrity) in an SOA is given in [61].

5.3.3 The Enterprise Service Bus

These practical problems (processing burden, complexity of security) and conceptual issues (statelessness of services) suggest the outsourcing of security tasks to an architectural component with the needed capabilities. A very promising approach is put forward by the paradigm of SOA: the *Enterprise Service Bus* is the technical backbone of a SOA landscape. This centralized communication infrastructure is responsible to provide interoperability between heterogeneous systems. This means connecting them in a loosely coupled way (independent of technical protocol details), mapping data-types, transforming formats and guaranteeing transparent routing dealing with technical aspects, such as load balancing and failover. It is considered to be the ideal candidate to offer value-added services such as security, monitoring and debugging [59].

Up until now, security has only been integrated at a very basic level. For example, [61] gives a detailed technical account on how to secure SOAs, but only covers authentication, authorization, confidentiality and integrity. The focus is set on a centralized setting confined to a single security domain (as opposed to the decentralized setting of inter-organizational scenarios presented in the next section). [54, 53] give a detailed account on issues related to the realization of security-critical decentralized SOA.

[71] describes how to realize the concept of a centralized communication infrastructure – the ESB – with open source software in all details. Security is only covered at a very basic level. In [51] the authors move a step further and discuss security as an infrastructure service in the context of an Enterprise Service Bus (ESB) and other patterns for the deployment of an SOA-security infrastructure. Nevertheless their

solution only covers the standards basic security services (e.g., authentication, key management etc.).

5.3.4 The SECTET framework for model driven security

Model Driven Security is an engineering paradigm that specializes Model Driven Software Engineering towards information security. It pursues two objectives: first, the integration of security aspects at an early stage of the engineering process and second, to shift the burden of security implementation from the software engineer to the security engineer. The term Model Driven Security was coined in [43]. The paper describes a software development process that supports the integration of access control requirements into system models. The models form the input for the generation of .net and J2EE security infrastructures. [60] Presents a framework for the formal verification of basic security requirements for security protocols based on UML models. Focusing on Service Oriented Architectures, in [48] the authors propose a framework for the platform-independent configuration of security infrastructure with authentication information.

Pursuing a much broader goal, the SECTET [53] framework supports business partners during the development and distributed management of decentralized peer-to-peer scenarios. Primarily developed for the realization of decentralized, security critical collaboration across domain boundaries – so-called inter-organizational workflows, it realizes a domain architecture aiming at the correct technical implementation of domain-level security requirements. It consists of three core components:

1. **Security Modeling.** The modeling component supports the collaborative specification of a scenario at the abstract level in a platform independent context. The component implements an intuitive domain specific language, which is rendered in a visual language based on UML2 for various modeling tools. The modeling occurs at a level of abstraction appropriate to bridge the gap between domain experts on one side and engineers on the other side, roles chiefly involved in two different phases of the engineering process – the requirements engineering and the design phase respectively.
2. **Code Generation & Model Transformation.** Model information is translated it into platform independent models (PIM) based on security patterns and protocols enforcing security requirements. The PIMs are refined into platform specific models of various granularities until they can be mapped into configuration code for the components of the target architecture. The layered approach is detailed in [66].
3. **Web services Based Reference Architecture.** The architecture specifies a Web services based target runtime environment for local executable workflows and back-end services at the partner node. The workflow and security components implement a set of workflow and security technologies based on XML- and Web services technology.

The SECTET reference architecture as presented in [53] enforces security mostly at the service endpoint. As already exposed, the approach exhibits significant limitations, especially when it comes to the realization of the complex security requirements. In the present contribution we propose an alternative blueprint realizing *Security as a Service*.

Nevertheless, the framework will serve as starting point for the elaboration of an approach for the high-level configuration of security-critical systems as planned in the description of work under T2.2 as part of the second year.

5.3.5 Security as a service

Security as a Service was introduced a couple of years ago in a series of publications. [69] first advocated the transition from traditional perimeter security to the concept of Service Oriented Security – a framework for risk analysis and management focusing on assets of a decentralized (service based) software architecture. Security is realized through decoupled, composeable services. The contribution focuses on identity and risk management, omitting complex security requirements. [70] discusses how the field information security failed to resolve key challenges of SOA security and arguments very much in line with our strategy that workable solutions have to move beyond traditional information security which only considers the CIA traid. Other publications on various aspects of security services in SOA that we discuss in this deliverable are [52, 49, 68].

We define Security as a Service (SeAAS) as the delivery of security functionality over infrastructure components in a service-oriented manner. For SOA, this means that security services are accessed through common Web services technologies and standards. Our definition thus goes beyond the common understanding which confines SeAAS to the practice of delivering traditional security application functionality (e.g., anti-virus software, anti-spyware, etc.) on-demand over the Internet (e.g., [63, 67, 72]). We identified the following security services (in order of increasing complexity):

Cryptographic and message processing services ensure basic confidentiality and integrity e.g., en-/ decryption of XML documents, signature validation etc.

1. **Security inter-operability services** facilitate interoperability of security mechanisms with external partners e.g., mapping of a user credentials to a kerberos token Services can be provisioned by an internal or an external security token service.
2. **Authentication** is a basic service necessary to all other requirements. Local or external service requesters are identified and authenticated relying on local identity stores and / or external identity providers.
3. **Authorization services** provide access control to resources. Authorization policies can be very complex. We cover static and dynamic role-based access control, four-eyes principle, negative access permissions, delegation, and break-glass policies.
4. **Security compliance services** check inbound messages in inter-domain communication for compliance with stated security requirements, e.g., valid and complete messages, presence of tokens, format, etc.
5. **Protocol based security services** are statefull services executed between two or more partners. A very prominent example is non-repudiation of sending or receiving in inter-domain communication.
6. **Security monitoring & auditing services** facilitate business- or application level security requirements, e.g., fraud and intrusion detection.

To cope with all classes, the proposed architecture goes beyond the mere bundling of security functionality within one security domain. For example, the execution of protocol based security (e.g., non-repudiation) realizes complex security requirements for processes involving two or more domains.

It is noteworthy, that in some cases, especially for the sake of security interoperability and efficient manageability, endpoints already rely on centralized services. We identified the following cases. *1. Authentication*. To authenticate a service requester, an endpoint commonly relies on a centralized identity store for identity management in its own security domain. In [49], the author identifies dedicated security services for advanced authentication and authorization requirements. *2. Interoperability*. In some cases, an endpoint may issue a request for a token mapping with a trusted 3rd party security token service. Here, interoperability seems to make the case for a "service-ization" of security. It is a way to cope with the heterogeneity of distributed, inter-organizational processes with different infrastructure owners organized into separated security domains. *3. Authorization*. Within a security domain, authorization is enforced at the endpoint, but relies on a central policy decision point for decisions on access requests. The XACML dataflow model [75] defines a reference architecture for the enforcement of access control in a service based environment within a security domain.

5.4 Architectural blueprint for security as a service

5.4.1 Architecture

Figure 65 shows the conceptual architecture for the proposed SeAAS approach (EI Pattern names in italics). The upper part shows the *ELGA Healthcare Services Architecture*. *Service Endpoints* provide business functionality. ELGA offers a number of healthcare services, such as access/update a patient's EHR, add Radiography to EHR etc. The service endpoints are decoupled from the security and messaging components. Inbound and outbound messages are delivered over the ESB. A business message contains service requests and responses whereas a security message contains security protocol data.

An ESB handles internal communication among the various components of a domain and external communication with business and security components of other domains. It intercepts inbound requests and forwards them to the *SeAAS Engine* for security evaluation. The SeAAS Engine is the central part of the SeAAS Component. To evaluate security, it retrieves the applicable security policy from *Policy Repository*. The security policy defines the security requirements for a particular request. The SeAAS Engine parses the policy, retrieves the security requirements and decides which security services will be needed to fulfill those requirements. It composes a security process to call those services in an appropriate order. For simplicity, our prototype currently uses a static process with a pre-defined order of execution of security services. For example, Authentication, Authorization and Non-repudiation Services are executed in following order: *Non-repudiation → Authentication → Authorization*.

The SeAAS Component offers security functionality as a set of *Security Services* implemented as *Security Components*. *Primitive Security Services* consist of Encryption, Signature and Time-stamping services. All other services (e.g., Authentication, Authorization, etc.) are considered *Advanced Services*. They leverage primitive security services. For example, the Non- repudiation service uses them to encrypt, sign and timestamp evidence. Primitive security services need keys/certificates, which are stored in the *Key Repository*. One key feature of the SeAAS architecture is the realization of security through decoupled components so to attain technology and language independence. All components can be implemented in any language and/or technology without any inter-dependence. Communication is fully message-oriented and is carried out over an ESB. Our prototype is based on Apache ServiceMix [40] – an open source ESB.

5.4.2 The SeAAS component

Deployed within a security domain, the *SeAAS Component* consists of a number of security services. Depending upon the requirements of the domain, new security services can be deployed during runtime. The *Policy Repository* (PR) and the *PKI Repository* offer supporting services. The PR holds the policies which specify security requirements, whereas PKI Repository is a local store for keys and certificates.

1. **Authentication.** The authentication Service provides intra- and inter-domain authentication. In case of an internal request, the authentication service validates the user’s local identity and sends the signed authentication decision to the endpoint. For a request from an outside domain, the authentication service first resolves the identity of the external user: it contacts the external identity provider (e.g., a *Security Token Service* (STS)) using *WS Interface*. After the STS validates the user, the authentication service creates a security context. This provides the functionality for *Identity Federation*.
2. **Authorization.** The Authorization Service verifies permissions assigned to users. They are defined in the policies stored in *Policy Repository*. Based on the policy, the service takes a decision and sends the result to the service endpoint for enforcement.
3. **Non-repudiation.** This service executes an out-of-band non-repudiation protocol between requester and the endpoint and stores evidence for dispute resolution (Section 0 is dedicated to a detailed discussion of non-repudiation).

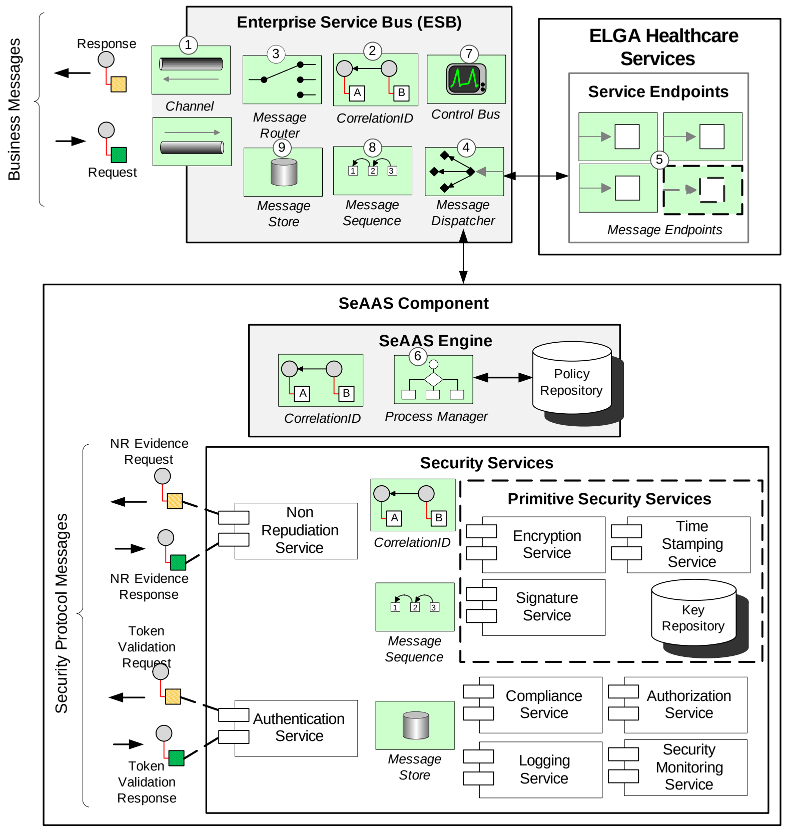


Figure 65: Conceptual SeAAS Architecture

4. **Security Compliance.** This service verifies the compliance of an inbound message with the security policy of the target service endpoint. The security policy of service endpoint defines the supported security mechanisms such as types of tokens, encryption and signature algorithms, message parts to be protected etc. The authentication service depends upon the evaluation performed by the compliance service. If a request is compliant, then the authentication service proceeds with token validation.

5. **Security Monitoring**. This service monitors significant security events generated by the security services of the SeAAS Component. For instance, the compliance service reports a security event, if a message does not meet an endpoint's security policy. The Non-repudiation service notifies a protocol failure, when the external endpoint does not follow the Non-repudiation protocol. The monitoring service of a domain's SeAAS component forwards these events to a central service accessible to all domains. The purpose of monitoring security centrally is to receive the security events from different domains and notify responsible and affected endpoints.

6. **Logging.** This service logs notifications sent by endpoints related to various business requests, responses, errors and exceptions. Externalizing security functionality as a set of services significantly reduces endpoint complexity. Moreover, the composition of security services as SeAAS components facilitates the deployment and the configuration of existing and new security components at deployment time and even during runtime.

5.4.3 Enterprise Integration Patterns

Patterns provide sound solutions to commonly known problems. In today's business world application integration is more complex, as the systems are loosely-coupled and use heterogeneous technologies. Message-oriented integration (MOI) aims at achieving integration among heterogeneous applications based on Enterprise Integration (EI) patterns. Figure 65 shows the EI patterns used by components instead of language-specific modules to realize message-oriented communication. EI patterns will help the security developer implement proposed SeAAS architecture, irrespective of what tools, technologies and languages she is using. A full catalogue of EI patterns is presented in [55]. As discussed below, we used some of those patterns, which are appropriate for designing the SeAAS components (EI patterns are circle-numbered in Figure 65 as well as in the text below).

The ESB uses *Channels (1)* to send/receive business and security protocol messages. As the integration of components in SeAAS is message-oriented, there should be certain mechanisms to relate the incoming and outgoing messages at any component. The *CorrelationID (2)* is used for matching requests and responses by the business and security components. Every message that enters and leaves the boundaries of a domain or a component in the domain is assigned a unique CorrelationID. The global correlationID for a domain is assigned by the ESB, whereas the local correlationIDs are assigned by components such as SeAAS Engine and Security Services. The *Message Router (3)* pattern is used for routing, so that the ESB sends the messages to appropriate destinations. A *Message Dispatcher (4)* consumes messages from Message Router and distributes them to their destinations. ESB uses this pattern to dispatch (business/security) messages to SeAAS Engine, Security Services, Service Endpoints and external domains. The service endpoints use *Message Endpoints (5)* pattern to indicate a client of messaging system i.e. ESB to send/receive messages. The *Process Manager (6)* pattern is used to model the SeAAS Engine for security process composition. The *Control Bus (7)* pattern indicates that the ESB sends logging and security events to the Logging and Security Monitoring components, which monitor failures, exceptions and security violations. The order of the messages is important, when the security services send and receive security protocol messages. The *Message Sequence (8)* pattern is used by security services to maintain the required order of security protocol messages. The ESB uses a separate Channel to store a copy of the messages into *Message Store (9)* to analyze the message before it delivers it to the target destination. The non-repudiation service uses this pattern to store the signed messages in a local persistent database. Similarly, Logging and Security Monitoring services store event notifications associated to certain message for security analysis.

5.4.4 Realizing complex security requirements with SeAAS

Complex security requirements are realized through advanced security services. Here, we illustrate the working of one of those services taking *Non-repudiation* as an example. There is much research related to the design of non-repudiation protocols [50, 65, 62]. Most is focused to the achievement desired properties like *Fairness* and *Timeliness*. Another issue extensively covered in research is concerned with the design of protocols with or without a Trusted Third Party (TTP) [64]. The protocols achieve non-repudiation among two or more protocol participants. A non-repudiation protocol is a cryptographic protocol that provides irrefutable evidence to its participants. A protocol is called *Fair*, if it provides the originator and the recipient of the message, with some evidence after completion of the protocol, without giving a participant an advantage over the other at any stage. Our design of NR protocol is based on ZG's protocol to achieve non-repudiation properties of fairness and timeliness [50].

The basic assumption is that a service endpoint handles both the business messages and protocol messages (i.e. keys, evidences etc). As already mentioned, our prime objective is to free the service endpoints from performing security related tasks. Here, we will apply the same principle to design and implement a fair non-repudiation protocol.

Although in [39] a first step has been achieved by integrating non-repudiation communication into Web service communication, there exists only a logical separation between non-repudiation messages and business messages. In this section we explain how these messages can be separated from business messages, by executing an out-of-band non-repudiation protocol. The proposed protocol does not only separate the business and security messages, but also maintains the desired properties of non-repudiation i.e. fairness and timeliness. In the next section, we will illustrate, how we design the non-repudiation protocol in the SeAAS architecture.

5.4.5 Enforcing a fair non-repudiation protocol in the SeAAS architecture

There are two different approaches to execute a NR protocol. The protocol is either enforced by the service endpoint [78] – in which case the service endpoint has to handle the security protocol messages in addition to business messages – or the service endpoint delegates the responsibility of executing the protocol to a dedicated non-repudiation service. The SeAAS architecture leverages the second approach.

In this approach, the protocol executes out-of-band between two dedicated non-repudiation services. The result is then communicated to the service endpoints, as shown in Figure 68. The NR services in domains 1 and 2 execute the protocol on behalf of the GP's *Client Application* (used by GP) and the ELGA service endpoint. Both delegate the security task to NR services through their respective SeAAS Engines. The detailed message communication to achieve fair non-repudiation in SeAAS architecture is shown as a UML Sequence Diagram in Figure 66. It shows the inter-domain communication between two non-repudiation services of the domains 1 and 2 and intra-domain communication among various components of each domain. The protocol is based on the ZG's protocol, which ensures *Fairness* and *Timeliness*.

The request to access the medical service is sent by a GP, through a *Client Application*. The ESB in domain1 intercepts the request and forwards it to the service endpoint of domain 2. The ESB in domain2 receives the request and routes it to the SeAAS Engine for security evaluation. The SeAAS Engine retrieves the policy that applies to the request from the Policy Repository and assigns the task to NR Service. Further security communication will take place among the non-repudiation services of two domains based on the detailed non-repudiation policy (an example is given in Section 6). Non-repudiation is achieved by exchanging the evidences of messages sent and received: NRService@Domain2 requests NRService@Domain1 for the evidence of the service request sent by GP (Mess. 5). NRService@Domain1 retrieves the request details from ESB through SeAAS Engine (Mess. 6-9), signs the message and sends the signed message NRO1 to NRService@Domain2 as evidence. This message (Mess. 10) consists of a URI's of NRServices@Domain1 & 2 (represented by letters A & B respectively), timestamp (i.e. T), Label for any other information (i.e. L) and NRO1(i.e. evidence). Note, unlike ZG's fair Non-repudiation Protocol, we don't send the service request in this message. Because, a service request is a business message, which has already been sent to service endpoint before beginning the NR protocol (Mess. 2). However, the encrypted message C requires a key i.e. K, and so far, the NRService@Domain1 has not sent that key to NRService@Domain2 for decrypting the request message.

To continue the protocol, NRService@Domain2 stores the evidence and sends a signed acknowledgment to NRService@Domain1. This is shown in Message 11 as Non- repudiation of Receipt (NRR1). At this moment, both NR services have the first part of evidence. The second part of the evidence will be signed by the TTP. Therefore, in Message 12, NRService@Domain1 sends the key K to TTP. TTP publishes the key and the second piece of evidence i.e. NROR2. Both the NR services retrieve this piece of evidence after time T. This completes the fair non-repudiation protocol between the NR services. After successful completion, the NRService@Domain2 sends the decrypted message and protocol completion notification to the service endpoint through SeAAS Engine (Messages 16-18). The service endpoint then sends the response to the client application. Thus, the protocol has executed out-of-band and the service endpoints were never involved. With this, we have not only separated the security from the business components in the architecture, but also the security communication from business communication.

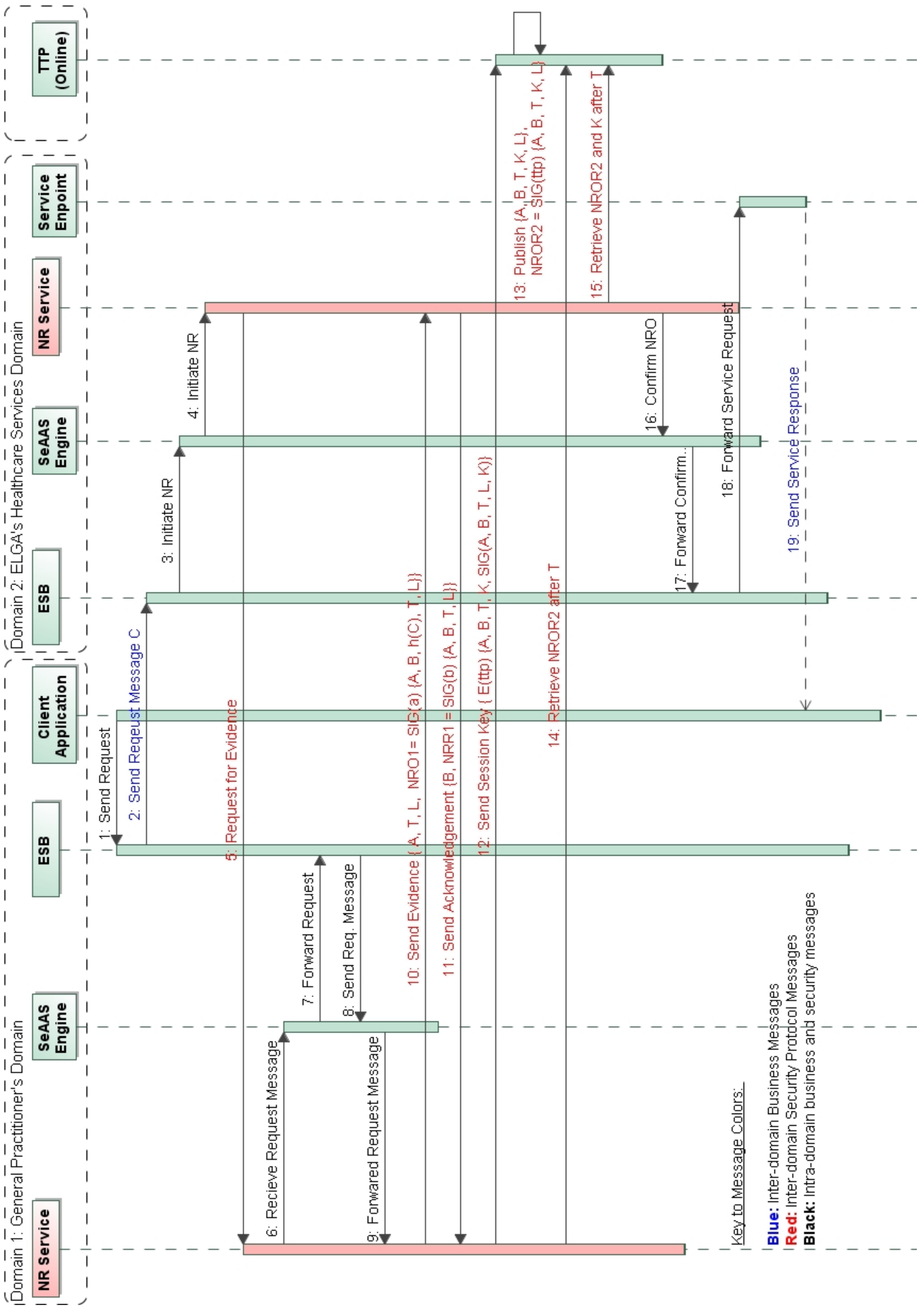
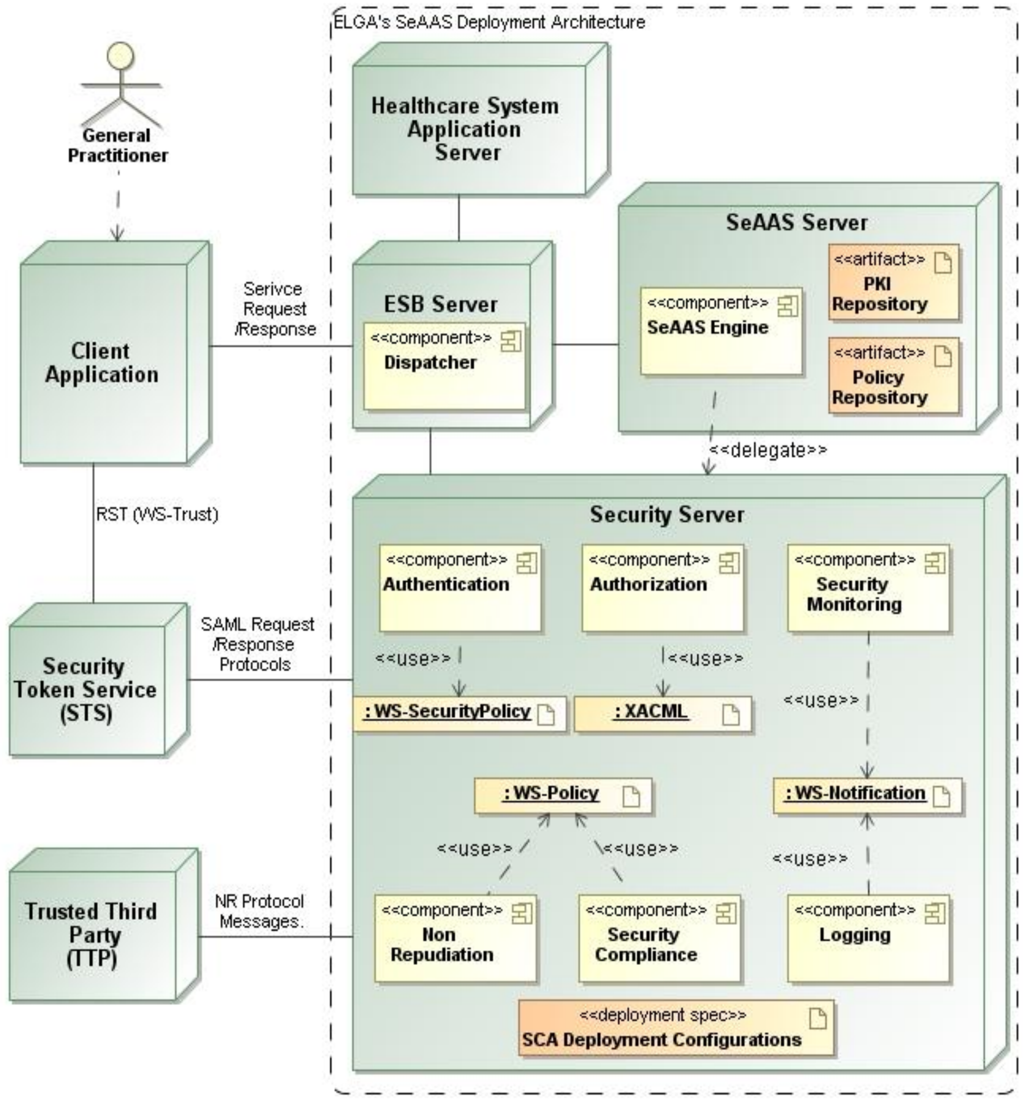

Domain 1: General Practitioner's Domain
NR Service
SeAAS Engine
ESB
Client Application
Domain 2: ELGA's Healthcare Services Domain
ESB
SeAAS Engine
NR Service
Service Enpoint
TTP (Online)
1: Send Request
2: Send Reqeust Message C
3: Initiate NR
4: Initiate NR
5: Request for Evidence
6: Recieve Request Message
7: Forward Request
8: Send Req. Message
9: Forwared Request Message
10: Send Evidence {A, T, L, NRO1= SIG(a) {A, B, h(C), T, L}}
11: Send Acknowledgement {B, NRR1 = SIG(b) {A, B, T, L}}
12: Send Session Key { E(ttp) {A, B, T, K, SIG(A, B, T, L, K)}
13: Publish {A, B, T, K, L}, NROR2 = SIG(ttp) {A, B, T, K, L}
14: Retrieve NROR2 after T
15: Retrieve NROR2 and K after T
16: Confirm NRO
17: Forward Confirm..
18: Forward Service Request
19: Send Service Response
Key to Message Colors:
Blue: Inter-domain Business Messages
Red: Inter-domain Security Protcol Messages
Black: Intra-domain business and security messages

Figure 66: Inter- and Intra- domain communication for Fair NR Protocol in SeAAS Architecture

5.5 Reference architecture



ELGA's SeAAS Deployment Architecture
General Practitioner
Healthcare System Application Server
SeAAS Server
<<artifact>> PKI Repository
<<artifact>> Policy Repository
<<component>> SeAAS Engine
Client Application
Serivce Request /Response
ESB Server
<<component>> Dispatcher
<<delegate>>
RST (WS-Trust)
Security Server
<<component>> Authentication
<<component>> Authorization
<<component>> Security Monitoring
Security Token Service (STS)
SAML Request /Response Protocols
<<use>>
: WS-SecurityPolicy
: XACML
: WS-Policy
: WS-Notification
Trusted Third Party (TTP)
NR Protocol Messages.
<<component>> Non Repudiation
<<component>> Security Compliance
<<component>> Logging
<<deployment spec>> SCA Deployment Configurations

Figure 67: Reference Architecture

The *Reference Architecture* (RA) is shown in Figure 67 as an UML deployment diagram. It shows components deployed; their relationships and Web services standards used. A service requester uses a *Client Application* to access the healthcare services offered by ELGA. The healthcare services are deployed at *Healthcare Systems Application Server*. The *ESB Server* provides a message-oriented middleware, which uses a *Dispatcher* component for inter- and intra-domain communication. The *SeAAS Server* hosts a *SeAAS Engine* component, which evaluates security based on the policy retrieved from the *Policy Repository*. The SeAAS Engine delegates the security task to security components4, which are deployed at the *Security Server*. The Security Server deploys a number of security components i.e. Authentication, Non -Repudiation, Authorization etc. The security components are configured at deployment time based on Service Component Architecture (discussed below) and the configurations are stored in a *SCA deployment*

Configurations file. The security components use different Web services security standards for performing security tasks.

Policy assertions for functional and non-functional requirements are defined with WS-Policy [77]. For example, the Non-repudiation component specifies the specific policy describes supported protocols, types and contents of the evidence and cryptographic methods required for message protection.

Figure 67 shows an example non-repudiation policy. It describes policy requirements as WS-Policy Assertions. The *Evidence Type* assertion defines that *DigitalSignature* on the message is required as an evidence. *Elements of Evidence* assertion defines that what should be the contents of a non-repudiation evidence. This implies that an evidence should contain the *Message With Token, MessageTimeStamp, URI's of EvidenceOriginator/EvidenceRecipient* and *EvidenceExpiry*. The *ProtocolType* assertion defines that a *Fair Non-repudiation Protocol* is required which involves an *Online TTP*, defined in the *TTPRole* assertion.

Security requirements of a service endpoint are defined as security assertions embedded into **WS-Policy** assertions **WS-SecurityPolicy** [76]. We use this standard to write the security policy of an endpoint, which defines supported type of bindings, tokens, encryption/signature algorithms. Two of the security components (security compliance and authentication) deployed at the Security Server use the WS-SecurityPolicy standard. The security compliance service checks the service request's compliance with the security policy. After the check, the authentication service proceeds for token validation for which it sends the requester's credentials to the Security Token Service (STS).

```
<wsp:Policy wsu:id="NR_policy">
  <wsp:ExactlyOne>
    <wsp:All>

<!-- Evidence Type Assertion: -->
 <nrp:EvidenceType>
   <wsp:Policy>
     <nrp:DigitalSignature/>
    </wsp:Policy>
</nrp:EvidenceType>

<!---Elements of Evidence Assertion-->
<nrp:EvidenceElements>
  <wsp:Policy>
     <nrp:MessageWithToken/>
     <nrp:MessageTimeStamp/>
    <nrp:EvidenceOrginator/>
    <nrp:EvidenceRecipient/>
    </nrp:EvidenceExpiry>
  </wsp:Policy>
</nrp:EvidenceElements>
. . . .
```

```
 . . . .
<!-- NR Protocol Type Assertion-->
<nrp:NRProtocolType>
  <wsp:Policy>
   <nrp:FairNRProtocol/>
  </wsp:Policy>
</nrp:NRProtocolType>

<!-- TTP Role for NR Assertion-->
<nrp:TTPRole>
  <wsp:Policy>
      <nrp:onlineTTP>
      <nrp:TTP_uri uri="ttp01.org"/>
    </wsp:Policy>
  </nrp:TTPRole>

    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 68: Example Non-Repudiation Policy (WS-Policy)

The **Security Assertion Markup Language (SAML)** is used to exchange security information between security domains [73]. In the RA, two components of Security Server i.e. authentication and Authorization components, use SAML standard. The authentication component creates authentication request/response based on SAML protocols. Using these protocols, the *Security Token Service* (STS) validates the tokens and sends a signed authentication SAML assertion to the authentication service, which forwards them to the service endpoint through SeAAS Engine. The Authorization component uses SAML in a similar manner. It makes authorization decision according to service requester's authorization policy and sends the decision as SAML authorization assertions to the service endpoint for enforcement.

The Extensible Access Control Markup Language (XACML) is a standard for authorization policies [75]. We use it to define permissions of a service requester. The *Policy Decision Point* (PDP) of the authorization service, makes decisions based on the permissions assigned to the roles (e.g. practitioner), defined as XACML rules.

WS-Trust provides interfaces for token issuance and validation [74]. A service consumer can get security tokens from an STS. The authentication component uses the WS-trust interface to get token validation decision by STS.

We use **WS-Notification** to send event notifications to Logging and Security Monitoring components [37]. Logging notifications carry information pertaining to the service requests and responses, whereas security alerts are notifications for security monitoring.

The **Service Component Architecture (SCA)** model is used for composition of security services performed by SeAAS Engine [38]. Security components are integrated to a *Security Composite*, which realizes a set of security requirements. SCA composite is written in the XML-based *Service Composition Definition Language* (SCDL).

The component-based architecture facilitates language- and technology-independence, reusability, and improves extensibility and maintainability. We use SCA properties for *Deployment-time Configuration* of security components based on security policy of a domain. These configurations are stored in *Deployment Specifications* as an *SCA Deployment Configurations* file, as shown in Figure 67.

The **Lightweight Directory Access Protocol (LDAP)** is used for directory access to retrieve certificates, policies and related information [56]. The policy repository is used for storage and retrieval of security policies, Authorization policies, and component policies like e.g. Non-repudiation used by SeAAS Engine and the Security services. The PKI Repository holds certificates and keys of service endpoints.

5.6 Discussion

In Section 5 we presented an architecture based on the paradigm Security as a Service. We motivated our approach with a discussion of the many limitations of endpoint security, the current practice in SOA security to enforce security with the endpoints. By default, the engineering intuition seems to impose a turning away from the concept of centralization entailing the threat of a single-point-of-failure or unbearable communication overhead.

But we showed how concepts of SOA, like declarative security, the Enterprise Service Bus and Model Driven Security, an advanced method of software engineering can open a new venue to the efficient realization of security critical, inter-organizational processes. The reference architecture is able to cope with the complex security requirements imposed by use cases from industries that have to deal with security-critical processes spanning multiple security domains.

The objection that SeAAS creates a single point of failure or a bottleneck can be countered by balancing the workload over replicated service components – an inherent advantage of SOA-style architectures. Taking communication overhead into account it does not always make sense to outsource all security functionality to a SeAAS provider. Tasks like basic XML processing can be left with the endpoint. It is up to the architect to decide upon the degree of service centralization. A hybrid approach distributes the tasks between endpoints and the SeAAS according to specific need. We currently envisage the extension of our architecture to support the flexible, run-time adjustment of the degree of centralization. Security services are registered with the SeAAS engine and advertised to potential consumers. Authorized requesters would access them as needed.

5.7 Case Study

The general idea of an architectural blueprint based on security services will be validated using a scenario derived from the description of the HOMES case study as described in Deliverable D1.1. - Description of the Scenarios and their Requirements.

The main idea of a reference architecture based on the paradigm *Security as a Service* is the support of a broad variety of architectural alternatives. Each alternative enforces one or more security requirements based on various participants, protocols and technologies – with a constellation that may possibly even change over time. This demands a highly flexible architecture.

The scenario should validate the claim that a security infrastructure

1. designed according to the blueprint "*Security as a Service*",
2. complemented by a methodology supporting model driven configuration,
3. and embedded in a change driven process for security-engineering

facilitates efficient and effective security engineering as well as management of evolving systems.

5.7.1 Scenario Description

This is a sketch of a scenario that will be elaborated in depth with the industry partner providing the HOMES case study.

We consider a third party service provider offering information services (e.g., an airline keeping passengers informed about flight details). An interested party (e.g., a passenger) can subscribe to various notification services. Communication basically runs over a subscriber's domotic network - called Home Network - which provides multiple channels for message forwarding (e.g., pager, mobile, blackberry, laptop etc). In such a scenario, connectivity to notification services is established through the deployment of one or more software bundles on the Home Network's Home Gateway. The latter identifies the subscriber, receives and forwards messages.

In case of security-critical transactions or communication additional security services may be needed to enforce security requirements or policies. For example, to make sure that neither the airline nor the passenger can deny having sent or received messages, the Home Network needs to provide a non-repudiation service).

5.7.2 Challenges

5.7.2.1 Architecture

Concerning the architecture, the key challenge is linked to the many alternatives on how a security service like non-repudiation may be realized (e.g., protocol type, software platform, algorithms etc.). In this respect, the proposed architecture should provide the highest degree of flexibility, thereby catering to various evolution paths.

The proposed solution consists in a robust reference architecture for an open, evolving service-oriented system that can cope with change in its many facets. The scenario will illustrate

1. How the architecture copes with changing security policies. To meet this challenge we will introduce concepts that support the high-level (platform-independent) configuration of security-critical systems.
2. How the architecture enforces a broad set of evolving security requirements. The key to this challenge will be the architecture's modular (or service based) design which will support the continuous evolution of security services.

5.7.2.2 Process

A second major challenge is related to the management of security-services in an evolving system and/or context.

In this context, the architecture will serve as a target system infrastructure for the change-driven process for security engineering which supports the collaboration and cooperation among different stakeholders with their specific views on the target system infrastructure as defined in Section 3.1.

The scenario will illustrate how the collaboration and cooperation among different Stakeholders is supported by a tool-based framework, taking into account the different perspectives and responsibilities of stakeholders.

5.7.3 Outlook

The following activities will be pursued in Year 2.

1. The architectural blueprint for the home network will be mapped to a suitable reference architecture that may accommodate a variety of alternative or complementary technical platform (e.g., OSGI, Linux services, etc.).
2. Security services will be identified and implemented at one or more appropriate layers (e.g., OSGI, JVM, User Space, Kernel, etc.).
3. Concepts to support the high-level configuration and administration of security services in the reference architecture will be elaborated.
4. The engineering, deployment and management of the various security services will be analyzed in context of the change-driven security engineering process as defined in Section 3.1.

6 Conclusion

In the preceding sections we have presented the results of WP2 of the first year focusing on the aspects of security engineering processes and security architectures for evolving systems.

The **SecureChange security engineering process i**s the first security engineering process which is fully driven by change events and change propagation. This novel process provides generic mechanisms for change propagation across stakeholders´ views.

The process is complemented by a framework and a catalogue of **change patterns**. Change patterns provide guidance for architectural change within the process, assisting the designer to develop an architecture that is resistant against certain foreseen evolutions of the requirements and assumptions.

Finally, at the architectural level we addressed the question of how a robust security architecture can be designed in order to support a broad range of changes. Our **Security as a Service Approach** (SEASS) applies well established mechanisms of functional architectures to the security domain, like the separation of abstraction layers, model-based configuration and orchestration of services.

The results of our research efforts have been published in three international publications (one further publication is submitted).

For the second year our activities will go in the following directions.

Concerning the SecureChange process we will develop a first prototype for tool support based on the experiences of the preliminary study (Section 3.3). The SecureChange Engine will at least support the concepts of change events, change propagation and the management of model states and model element dependencies. In addition we will develop an integrated view of the process as an umbrella all work packages of the project. This will be complemented by a case study in the ATM context.

Concerning change patterns and the SEASS approach, the activities in the second year will have a focus on validation based on the HOMES case study.

7 Glossary

An **architectural support pattern** is a software pattern referenced by a **change pattern** that needs to be instantiated in a software architecture to prepare this architecture for future evolution.

An **artefact** distinguishes on an abstract level the different models and concepts which are used by the different Work Packages.

Change is described by a Change Trigger.

A **change event** is a general trigger of change which is derived from a set of change scenarios.

A **change line model** represents relationships between several Changes included in one Change Line and Transitions which describes a set of transformation rules between several changes

The **change guidance** is described by a **change pattern**, and outlines the steps that an architect of a system should follow to update the software system when a **change scenario** manifests itself.

A **change pattern** is a combination of a possible **change scenario**, **architectural support patterns** and **change guidance** that can be used to prepare an architecture for future evolution.

A c**hange request** is a general description of some change in the system.

The **change request model** traces changes inside the Static Model.

A **change scenario** is the description of a change in requirements of assumptions, that can be accommodated by using a change pattern.

A **change transition** is a description of all the differences from one change to another.

A **change trigger** expresses the rationale of Change and activates a Change Request.

The **Common System View** represents the conceptual underpinning for the security management process. Its elements are the conceptual units subject to change.

Declarative security, technique of configuring security components through machine readable policies.

Dependencies describe the associations between various model elements allowing change to percolate through the Model Layers.

A **domain** is any subset of a conception (being a set of elements) of the universe that is conceived of as being some 'part' or 'aspect' of the universe.

DSML stands for Domain Specific Modeling Language.

Electronic Health Record, European initiative aiming at realising the infrastructure for nation-wide centralized repositories of electronic patient records.

ELGA (Elektronische Gesundheitsakte), German acronym for „electronic health record“, stands for the consortium driving the realization of an HER in Austria.

Endpoint security, method of enforcing security in Web services based environments directly at the node hosting the service as opposed to a centralized service as in Security as a Service.

Enterprise Service Bus, technical backbone of a SOA landscape. This centralized communication infrastructure is responsible to provide interoperability between heterogeneous systems**.**

Meta-Model Plugins extend the Functional System Meta Model with specific concepts that cover different aspects of the System (e.g. Security Plugin, Verficiation Plugin).

Model Element States reflect the milestones in the lifecycle of the modelled artefacts and are used to reflect relevant changes.

A **Model Layer** is comprised of a set of model elements types which capturing different levels of abstraction or degrees of granularity.

The **Functional System Meta Model** defines functional system concepts like business processes, information objects, roles, components and their relationships.

Model-driven security is an engineering paradigm that specializes Model Driven Software Engineering towards information security.

Reference architecture, specific technical platform or infrastructure realised according to an architectural paradigm (e.g., Security as a Service).

Robustness, property of a component, system or architecture referring to the ability to easily accommodate changes. A robust architecture thus supports long-lived, evolving systems.

SECTET, a framework for model driven security engineering in SOA.

Security as a Service (SeAAS) stands for an architectural paradigm. It defines an architectural blueprint that transposes the model of Software as a Service to the security domain.

The **security micro process** is executed by each of the stakeholders within her specific domains and consists of a set of activities result in a set of newly created or updated security artefacts.

Service-oriented architecture (SOA), architectural paradigm for distributed systems relying on loosely coupled software components called services. SOA is based on a set of flexible design principles.

The **Static Model** is a set of interrelated models and an instantiation of a Functional System Model which is extended with different Meta-Model Plugins.

A **View** consists of a selected set of model elements (together with selected interdependencies) and corresponds to the usual notion of a representation of a system from the perspective of a related set of concerns.

8 Bibliography

[1] Bart De Win, Riccardo Scandariato, Koen Buyens, Johan Grégoire, Wouter Joosen, On the secure software development process: CLASP, SDL and Touchpoints compared, Information and software technology, volume 51, issue 7, pages 1152-1171, July 2009.

[2] OWASP, Comprehensive, lightweight application security process, http://www.owasp.org, 2006.

[3] M. Howard, S. Lipner, The Security Development Lifecycle (SDL): A Process for Developing Demonstrably More Secure Software, Microsoft Press, 2006.

[4] G. McGraw, Software Security: Building Security, Addison Wesley, 2006.

[5] Perry, D.E. and Wolf, A.L. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes. 1992, Vol. 17, 4.

[6] Bass, L., Clements, P. and Kazman, R. Software architecture in practice. s.l. : Addison-Wesley Professional, 2003.

[7] ISO/IEC 42010:2007 - IEEE Std 1471-2000. Systems and software engineering - Recommended practice for architectural description of software-intensive systems. 2007.

[8] Yoshioka, N. and Washizaki, H. and Maruyama, K., A survey on security patterns. In Progress in Informatics, Vol 5, 2008.

[9] Munawar Hafiz, Paul Adamczyk, Ralph E. Johnson, "Organizing Security Patterns," IEEE Software, vol. 24, no. 4, pp. 52-60, July/August, 2007.

[10] Gross, D. and Yu, E., From non-functional requirements to design through patterns. Requirements Engineering, Springer, Vol. 6, Number 1, 2001.

[11] Weiss, M., Modelling security patterns using NFR analysis. In: Integrating security and software engineering: advances and future visions (eds: Mouratidis, H. and Giorgini, P.). Idea Group Publishing, 2006.

[12] Axel van Lamsweerde. From System Goals to Software Architecture In Formal Methods for Software Architectures, M. Bernardo & P. Inverardi (eds), LNCS 2804, Springer-Verlag, 2003, 25-43

[13] Manadhata, P. and Wing, J.M., An attack surface metric. 2008.

[14] Koen Buyens, Bart De Win, Wouter Joosen, Resolving least privilege violations in software architectures, Proceedings of the 31st International Conference on Software Engineering Workshops, Vancouver, 16-24 May

[15] Ren, J. and Taylor, R. A secure software architecture description language. Workshop on Software Security Assurance Tools, Techniques, and Metrics, 2005.

[16] Jürjens, J., Secure systems development with UML. Springer Verlag, 2005.

[17] Lodderstedt, T. and Basin, D. and Doser, J. and others, SecureUML: A UML-based modeling language for model-driven security. Lecture notes in computer science, Springer, 2002.

[18] Shawn Hernan and Scott Lambert and Tomasz Ostwald and Adam Shostack, Uncover Security Design Flaws Using The STRIDE Approach. MSDN Magazine, November 2006. http://msdn.microsoft.com/en-us/magazine/cc163519.aspx

[19] L. Dai, K. Cooper. A Survey of Modelling and Analysis Approaches for Architecting Secure Software Systems. International Journal of Network Security, 2007.

[20] C. Rahn, “Lufthansa computer-server glitch hampers flights (update2),” September 2009, accessed: 2009-10-15. [Online]. Available: http://www.bloomberg.com/apps/news? pid=20601100\&sid=avUsNKymzMPk

[21] B. Y. Y. Nhlabatsi, A.; Nuseibeh, “Security requirements engineering for evolving software systems: a survey,” The Open University, Department of Computing, Technical Report 2009-05, 2009.

[22] R. Breu, “Ten principles for living models — a manifesto of change–driven software engineering,” accepted for 4th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2010).

[23] F. Innerhofer-Oberperfler and R. Breu, “Using an enterprise architecture for it risk management,” in ISSA06: Proc. Information Security South Africa Conference, 2006.

[24] F. Innerhofer-Oberperfler, D. Bachlechner, R. Maier, V. Hahn, M. Weitlaner, and R. Breu, “Information Security Management: A Collaborative Approach,” in In Proc. STM 2009: 5th International Workshop on Security and Trust Management (STM 2009) in conjunction with ESORICS 2009, September 2009.

[25] M. Hafner and R. Breu, Security Engineering for Serviceoriented Architectures. Springer, October 2008.

[26] “Deliverable 5.2: Documentation of forecasts of future evolvement,” SecureChange (EU ICT-FET-231101), Unpublished Draft Report ICT-FET- 231101 D5.2, 2009.

[27] C. J. Alberts and A. J. Dorofee, Managing information security risks: the OCTAVE approach. Pearson Education, 2002.

[28] T. I. S. Organization, “Introduction to iso 27005 (iso27005),” 2008.

[29] IEEE Architecture Working Group. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471-2000, IEEE. (2000).

[30] Normand, V., Felix, E., Jitia, C. A DSML for security analysis, IST MODELPLEX project restricted deliverable 3.3.g. (2009).

[31] “Deliverable 4.1: Security modeling notation for evolving systems,” SecureChange (EU ICT-FET-231101), Unpublished Draft Report ICT-FET- 231101 D4.1, 2009.

[32] “Deliverable 3.2: Methodology for evolutionary Requirements,” SecureChange (EU ICT-FET-231101), Unpublished Draft Report ICT-FET- 231101 D3.2, 2009.

[33] IBM Rational DOORS Homepage, http://www-01.ibm.com/software/awdtools/doors/

[34] Giorgini, P., Massacci, F. and Zannone, N. Security and trust requirements engineering. Lecture notes in computer science. 2005, Vol. 3655.

[35] Compagna, L., et al. How to integrate legal requirements into a requirements engineering methodology for the development of security and privacy patterns. Artificial Intelligence and Law. 2009, Vol. 17, 1.

[36] Massacci, F, Mylopoulos, J and Zannone, N. An ontology for secure socio-technical systems. Handbook of Ontologies for Business Interaction. 2007.

[37] OASIS Web Services Notification (WSN) Specifications, 2006. http://docs.oasisopen.org/wsn.

[38] Introducing SCA, 2007. http://www.davidchappell.com/.

[39] B. Agreiter, M. Hafner, and R. Breu. A fair Non-repudiation service in a web services peerto-peer environment. *Computer Standards & Interfaces*, 30(6):372–378, 2008.

[40] Apache-ServiceMix. Open Source ESB. http://servicemix.apache.org/.

[41] S. Bajaj. Web Services Policy 1.2 - Framework (WS-Policy) W3C Member Submission 25 April 2006. Technical report, W3C, 2006.

[42] M. Bartel, J. Boyer, and B. Fox. XML-Signature Syntax and Processing, W3C Recommendation 12 February 2002. Technical report, W3C, 2002.

[43] D. Basin, J. Doser, and L. Torsten. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[44] M. Y. Becker and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *Comp. Sec. Foundations Workshop, 2004. Proc. 17th IEEE*, 2004.

[45] M. Donner. From the editors: Whose data are these, anyway? *IEEE Security and Privacy*, 2(3):5–6, 2004.

[46] Arge Elga. Arbeitsgemeinschaft elektronische gesundheitsakte. http://www.arge-elga.at/.

[47] B. Atkinson et al. Web services security (ws-security) - version 1.0. Specification, IBM Corp., Mircosoft Corp., VeriSign, Inc., 2002.

[48] F. Satoh et al. Adding authentication to model driven security. In *ICWS '06: Proc. of the IEEE Intern. Conf. on Web Services*, Washington, DC, USA, 2006. IEEE Computer Society.

[49] J. Lopez et. al. Specification and design of advanced authentication authorization services. *Computer Standards and Interfaces*, 27(5):467–478, 2005.

[50] J. Zhou et al. Evolution of Fair Non-repudiation with TTP. In *ACISP '99: Proc. of the 4th Australasian Conf. on Information Security and Privacy*, London, UK, 1999. Springer.

[51] M. Alam et al. Modeling and Enforcing Advanced Access Control Policies in Healthcare Systems with SECTET. In *MOTHIS '07: MODELS 2007*, Nashville, USA, 2007.

[52] M. Hondo H. Hinton and B. Hutchison. Security patterns within a service-oriented architecture, Nov. 2005. http://www.ibm.com/websphere/developer/services/.

[53] M. Hafner and R. Breu. *Security Engineering for Service-oriented Architectures*. Springer, October 2008.

[54] M. Hafner, R. Breu, B. Agreiter, and A. Nowak. Sectet: an extensible framework for the realization of secure inter-organizational workflows. *Internet Research*, 16(5):491–506, 2006.

[55] Gregor Hohpe and BobbyWoolf. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[56] IETF. The LDAP Technical Specification, 2006. http://tools.ietf.org/html/rfc4510. 21. T. Imamura. XML Encryption Syntax and Processing, W3C Recommendation 10 December 2002. Technical report, W3C, 2002.

[57] T. Imamura. XML Encryption Syntax and Processing, W3C Recommendation, 10 December 2002. Technical report, W3C, 2002.

[58] innoQ. Web services standards overview, Nov. 2006, http://www.innoq.com/resources/wsstandards-poster/ and http://www.innoq.com/soa/ws-standards/.

[59] N. M. Josuttis. *SOA in Practice: The Art of Distributed System Design (Theory and Practice)*. O'Reilly Media, Inc., August 2007.

[60] J. Juerjens. *Secure Systems Development with UML*. SpringerVerlag, 2004.

[61] R. Kanneganti and P. Chodavarapu. *SOA Security in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[62] S. Kremer, O. Markowitch, and J. Zhou. An Intensive Survey of Fair Non-Repudiation Protocols. *Computer Communications*, 25:1606–1621, 2002.

[63] MacAfee. Security as a service, 2008. http://www.mcafee.com/us/local content/solution briefs/sb saas 0709.pdf.

[64] O. Markowitch and S. Kremer. An optimistic non-repudiation protocol with transparent trusted third party. In *ISC '01: Proc. of the 4th Int. Conf. on Information Security*, London, UK, 2001. Springer.

[65] O. Markowitch and Y. Roggeman. Probabilistic non-repudiation without trusted third party. In *2nd Conf. on Security in Communication Network*, 1999.

[66] M. Memon, M. Hafner, and R. Breu. SECTISSIMO: A Platform-Independent Framework for Security Services. In *ModSec '08: MODELS 2008*, Toulouse, France, 2008.

[67] Microsoft. Windows live onecare, 2006. http://onecare.live.com/standard/enus/3/default.htm.

[68] Oracle. Service-Oriented Security: An Application-Centric Look at Identity Management, 2008. http://www.oracle.com/.

[69] G. Peterson. Service oriented security architecture, 2005. http://www.arctecgroup.net/ISB1009GP.pdf.

[70] G. Peterson. Service-oriented security indications for use. *IEEE Security and Privacy*, 7(2):91–93, 2009.

[71] T. Rademakers and J. Dirksen. *Open-Source ESBs in Action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[72] Symantec. Symantec names genesis norton 360, 2006. http://www.symantec.com/about/news/release/article.jsp?prid=2006053101.

[73] OASIS TC. Security Assertion Markup Language (SAML), 2005. http://www.oasisopen.org.

[74] OASIS TC. WS-Trust Specifications, 2005. http://docs.oasis-open.org/.

[75] OASIS TC. Extensible Access Control Markup Language (XACML), 2006. http://www.oasis-open.org.

[76] OASIS TC. WS-SecurityPolicy, 2007. http://docs.oasis-open.org/.

[77] W3C. Web Services Policy 1.2 - Framework , 2006. http://www.w3.org/Submission/WSPolicy.

[78] Reto Zimmermann. Design and Prototypical Implementation of a Non-Repudiation System for Mobile Grid Services. http://www.ifi.uzh.ch/archive/mastertheses/.