How to Incorporate Security Requirements into the ArchWiz Tool

Caroline Uppsåll
cul05001@student.mdh.se
2010-01-26

Master Thesis, Computer Science, 30 credits
Mälardalens University
School of Innovation, Design and Engineering (IDT)
Performed at:
ABB Corporate Research
Västerås, Sweden

Supervisors:
ABB Corporate Research:
Pia Stoll, pia.stoll@se.abb.com
Mälardalens University:
Rikard Land, rikard.land@mdh.se
Abstract

This thesis have two tasks: one is to help in the development of the ArchWiz tool at ABB and the other is to find a way of how to formalise security related architectural responsibilities in a general way so that they can be incorporated in the tool. This thesis report focuses on the software quality attribute security.

The ArchWiz tool is developed at ABB Corporate Research in Västerås. The scope of the tool is to serve as a software architecture guidance and knowledge tool for all software architecture professionals within ABB. The ArchWiz tool is a way of helping the architects of ABB to match their product requirements with a list of general reusable requirements. The matched product requirements can then use the reusable requirement's architectural solutions, which are also presented in the tool. The tool focuses on usability, security and safety in this first version but it is constructed so that the user can add their own general requirements regarding any quality.

The architecture design phase in the development of a software system is a key part in the development process, it gives the first design decisions and gives information on if the system will have potential to meet its key requirements. Security is a software quality that has grown in importance for the architectural design of the system. There exist a number of potential threats and attacks that might breach the security of the software and these threats needs to be protected against.

The ArchWiz project closed in December 2009 and at that time the tool was not finished. A good foundation and a GUI framework for further implementations were developed but to get a fully functioning tool more implementations need to be made. Security is one of the quality attributes, which the ArchWiz is supposed to support. Suggestions on how to formalise security responsibilities in the tool have in this thesis been created and analysed. However, the suggestions need to be in incorporated in the tool and tested through users tests with the ABB architects. With the user tests as basis, the best suggestion can be selected.

Sammanfattning

Detta examensarbete innefattar två olika uppgifter: en är att hjälpa till med utvecklingen av verktyget ArchWiz på ABB, och den andra är att hitta ett generellt sätt att formalisera säkerhet på, så att det kan införivas i verktyget. Denna rapport fokuserar på säkerhetskraven i mjukvarusystem.

ArchWiz är ett verktyg utvecklat på ABB Corporate Research i Västerås. Syftet med verktyget är att fungera som guidning till mjukvaruarkitektiker och att vara ett kunskapsverktyg för alla som jobbar med arkitektur på ABB. ArchWiz är ett sätt att hjälpa ABB's arkitekter att matcha deras produkt-specifika krav med en lista av generella, återanvändbara krav. De matchade produkt-specifika kraven kan sen använda de återanvändbara och generella kravens lösningar, vilka är presenterade i verktyget.

Ett mjukvarusystems arkitektur är en viktig del i utvecklingsprocessen, den ger de första valen av design och information om huruvida systemet har potential att möte sina krav. Säkerhet är en växande kvalitet i mjukvara och då också i dess arkitektur. Det finns ett antal potentiella hot och attacker som möjligen kan överträda mjukvarans säkerhet. Systemet behöver skydd från dessa hot.

Preface

Acknowledgements

I would like to take the opportunity to thank some people for their help during this thesis job. Pia Stoll how is my supervisor at ABB CRC and also the ArchWiz project manager, for advice and assistance. And Anton Jansen who works at ABB CRC and was involved in the ArchWiz project, for all his help and support. I would also like to thank my supervisor at Mälardalens University, Rikard Land who helped with guidance and input during this period.

I would also like to give a thank you to my family for their support.
Table of Contents

Chapter 1 – Introduction................................................................................................................... 7
  1.1. Background........................................................................................................................... 7
  1.2. Purpose................................................................................................................................ 7
  1.3. Reader Instructions............................................................................................................... 8
Chapter 2 - Problem formulation.................................................................................................... 9
  2.1. The ArchWiz Project............................................................................................................ 9
    2.1.1 Changes along the way..................................................................................................... 13
  2.2. Problem analysis.................................................................................................................. 13
  2.3. Delimitation......................................................................................................................... 14
Chapter 3 – Method......................................................................................................................... 15
  3.1. Workflow.............................................................................................................................. 15
Chapter 4 - Related Work and Theory.......................................................................................... 17
  4.1 Software Architecture......................................................................................................... 17
    4.1.1. Influences to software architecture............................................................................... 17
    4.1.2. Why is software architecture important........................................................................ 18
    4.1.3 Architectural activities..................................................................................................... 18
    4.1.4. What makes a “good” architecture?............................................................................... 19
    4.1.5. Architectural Patterns, Reference Models, and Reference Architectures.................. 20
    4.1.6. Architectural structures and views................................................................................ 21
  4.2 Patterns................................................................................................................................ 23
    4.2.1 Why patterns?.................................................................................................................. 24
    4.2.2 What makes a pattern...................................................................................................... 24
    4.2.3 Categories of Patterns.................................................................................................... 26
    4.2.4 Relationships between patterns..................................................................................... 26
    4.2.5 Description...................................................................................................................... 27
    4.2.6 Patterns in Software Architecture.................................................................................. 28
  4.3 Usability-Supporting Architectural Patterns (USAP)............................................................. 29
    4.3.1. USAP project................................................................................................................. 29
    4.3.2. USAP Pattern Language.............................................................................................. 31
    4.3.3. The USAP taxonomy.................................................................................................... 32
    4.3.4. The USAP prototype..................................................................................................... 33
  4.4 Security................................................................................................................................. 34
    4.4.1 Security Concepts............................................................................................................ 36
    4.4.2 Risk management............................................................................................................ 38
    4.4.3 Security policy and mechanisms.................................................................................... 39
    4.4.4 The basic components.................................................................................................... 39
    4.4.5 Security tactics............................................................................................................... 41
    4.4.6 Threats........................................................................................................................... 43
    4.4.7 Kinds of security objectives........................................................................................... 45
    4.4.8 Guiding principles for software security....................................................................... 49
    4.4.9 Human issues.................................................................................................................. 51
  4.5 Security Patterns................................................................................................................... 52
    4.5.1 Authenticator pattern...................................................................................................... 53
    4.5.2 Authorization pattern...................................................................................................... 54
    4.5.3 Role-based access control pattern (RBAC)................................................................. 55
    4.5.4 Single access point pattern............................................................................................ 57
    4.5.5 Multilevel Security Pattern............................................................................................ 58
Chapter 5 – Results......................................................................................................................... 60
Index of Figures

Figure 1: System Process Transformation of the ArchWiz................................. 10
Figure 2: ArchWiz System Networking............................................................... 11
Figure 3: ArchWiz Primary Use Cases............................................................... 11
Figure 4: Make Architecture Decision.............................................................. 12
Figure 5: Navigate Knowledge Views............................................................... 12
Figure 6: State diagram of the ArchWiz tool.................................................... 13
Figure 7: Model of the workflow of the thesis work......................................... 16
Figure 8: The architect's influences. After [1].................................................. 18
Figure 9: Relationships between Architectural Pattern, Reference Model, Reference Architecture. After [1] ................................................................. 20
Figure 10: Architectural structures. After [1].................................................... 21
Figure 11: Summary of the three-part schema. After [3]................................... 25
Figure 12: Relationships between USAPs. After [19]........................................ 31
Figure 13: Screenshot of the USAP prototype web tool.................................... 33
Figure 14: USAP task flow model.................................................................... 34
Figure 15: Concepts that influence and are influenced by security requirements. After [26] ...... 38
Figure 16: Summary of the tactics for security. After [1]................................. 41
Figure 17: How some common threats relate to classes and basic components... 45
Figure 18: How patterns relate to accomplish access control. After [9]............. 52
Figure 19: Authenticator pattern. After [9]....................................................... 53
Figure 20: authorization pattern. After [44]...................................................... 54
Figure 21: Relations of authorization patterns. After [9].................................. 55
Figure 22: Basic RBAC Pattern. After [44]...................................................... 56
Figure 23: Single Access Point Pattern. After [9]............................................ 58
Figure 24: Multilevel security Pattern. After [9].............................................. 59
Figure 25: Welcome view of the ArchWiz tool................................................. 61

Index of Tables

Table 1: Architectural structures of a system.................................................... 22
Table 2: How some security patterns relate to the guiding principles for security. After [39] ...... 49
Table 3: Comparative model for security pattern and Security objectives.......... 71
Figure 26: Create new project window................................................................. 61
Figure 27: Open project window........................................................................ 62
Figure 28: The tool menu bar........................................................................... 62
Figure 29: The tree structure.............................................................................. 63
Figure 30: The tool on a text node..................................................................... 64
Figure 31: Select/Deselct nodes in the tree.......................................................... 65
Figure 32: Only selected nodes are shown in the tree......................................... 65
Figure 33: The tool on a decision node................................................................. 66
Figure 34: Splitted view..................................................................................... 67
Figure 35: Todo-list from the USAP prototype web tool........................................ 68
Figure 36: How different security aspects relate.................................................. 70
Figure 37: Security formalisation suggestion 1...................................................... 72
Figure 38: Modified version of suggestion 1........................................................ 73
Figure 39: Security formalisation suggestion 3...................................................... 74
Figure 40: Security formalisation suggestion 3...................................................... 75
Figure 41: Security formalisation suggestion 4...................................................... 76
Figure 42: Generalisation of suggestions 1-3....................................................... 78
Figure 43: A possible way to develop security formalisation suggestion 3.......... 81
Chapter 1 – Introduction

This report is a Master thesis made for the School of Innovation, Design and Engineering (IDT) at Mälardalens University. The thesis work was commissioned by ABB Corporate Research in Västerås.

1.1. Background

ABB AB is a global leader in power and automation technologies that enable utility and industry customers to improve their performance while lowering environmental impact. The ABB Group of companies operates in around 100 countries and employs about 120,000 people worldwide.

One division in the ABB Group is ABB Corporate Research Centers (ABB CRC) that are located in seven countries (China, Germany, India, Poland, Sweden, Switzerland and the United States). The ABB CRC in Sweden is located in Västerås and together with other R&D centers they develop technologies for future products and services for ABB’s core businesses. One of the areas ABB CRC focuses on is Software architecture and processes, that is the group within ABB CRC that this thesis work will be done in.

During 2007 to 2008, ABB and the Software Engineering Institute (SEI)/Carnegie Mellon University, Pittsburgh, USA conducted a ”Usability-Supporting Architecture Pattern” (USAP) project. The project resulted in contemporary research contribution to the field of usability and software architecture in the domain of Software Engineering. Usability-supporting architectural patterns (USAPs) where developed as a way to explicitly connect the need of architecturally-sensitive usability concerns to the design of software architecture. This work resulted in an activity taxonomy for USAPs based on software responsibilities and a web-based tool for evaluating an architecture with respect to the activity taxonomy. More about the USAP prototype tool and the research can be read in section 4.3.

ABB CRC decided to make a continuation of the USAP project called ”Architectural Wizard” (ArchWiz) project which is supposed to extend the software engineering methods and tool used by the USAP project to apply to the software quality attributes safety and security as well as usability. The ArchWiz tool take a slightly different approach than the USAP prototype did. The ArchWiz tool will allow the user to create their own quality attribute taxonomy, but will when started present the USAP responsibility taxonomy and one of the, in this thesis, proposed security taxonomies. The ArchWiz project is described in more detail in section 2.2.

1.2. Purpose

The goal of the ArchWiz project is to extend the software engineering methods used by the USAP project to apply to the qualities safety and security. The ArchWiz project should also extend the USAP prototype tool. The tool is to be extended in such a way that architects can contribute with architectural knowledge in the form of software requirements and refined requirements. The tool has it's own requirements for the architectural design of the ArchWiz tool. Some of the more important design requirements include requirements on distributed collaboration, portability, and usability. The presented architectural solution for ArchWiz and its implementation is required to meet these requirements and should be possible to realise within the project's budget.

To summarise, the USAP theory should be built upon in order to create hierarchies of reusable software quality requirements and reusable solutions for these requirements. These hierarchies should be incorporated in a tool to be used by the software architects and developers at ABB.
1.3. Reader Instructions

This thesis report discusses the area of software security in the architectural design phase of the software development process. Chapter 2 describes the thesis problem with its foundation in a previously done work on usability in software architecture described in section 4.3. A literature study of the field of software architecture is presented in section 4.1 and a literature study of the main area security is presented in section 4.4. Furthermore, in chapter 5 is the results of the thesis work presented, in section 5.1 is the implemented tool described and in section 5.2 is the results of a security formalisation presented. There still needs to be development work done on the tool for it to be fully functioning, the remaining work to be done is listed in section 6.2.1, as well as future work to be done on the security formalisation. My own thoughts and a conclusion can be read in chapter 6. The requirements list for the ArchWiz project at ABB CRC is displayed in appendix 1.
Chapter 2 - Problem formulation

The assignment was to research the area of security to come up with suggestions on how to formalise security responsibilities in a way so that the architect could consider the responsibility and make a decision whether the responsibility is relevant or not for the architecture design. The formalised security responsibilities should be incorporated in the ArchWiz tool. The quality of security needs to be formalised in a general way, i.e. the security responsibilities should be expressed as entities in a security domain model. How the security domain model should be formalised to suit the ArchWiz purpose is one of the challenges of this thesis. The architects at ABB can use the tool to build common architectural security solutions to common architectural security requirement into the systems they are designing. Another part of the assignment was also to help in the development of the new extended ArchWiz tool. For information about the ArchWiz project see section 2.2 and the results of the tool and the research can be found in sections 5.1 and 5.2.

2.1. The ArchWiz Project

The ArchWiz tool is developed at ABB Corporate Research in Västerås. The tool should help ABB software architects to map product requirements to a set of general requirements, i.e. the architectural responsibility. By mapping the product requirements to general requirements, recurring requirements can be identified and the general requirement's reusable architectural solution used. There are two basic approaches, a bottom-up or top-down approach. In the bottom-up approach, the architects collects requirements they have used for designing similar architectural solutions. Once the same set of requirements are encountered for similar architectural solutions, a pattern can be distilled that describes the set of requirements in a coherent way and provides a general solution for them. The top-down approach uses architectural patterns as a starting point and turns the responsibilities found in them into ABB specific responsibilities.

The ArchWiz project is a follow-up project from the ISSUE for FM project, run 2007-2008 by SECRC (ABB Corporate Research, Software Architecture) and Carnegie Mellon University/Software Engineering Institute, Pittsburgh, USA.

The scope and vision for ArchWiz is that the tool will serve as a software architecture guidance and knowledge tool for all software architecture professionals within ABB. The ArchWiz will contain architectural knowledge in the form of general software requirements for a number of software qualities, i.e. usability, security and safety.

All ABB software products that have quality concerns related to the quality requirements of ArchWiz such as safety, security or usability may benefit from using the ArchWiz tool. Either as an evaluation tool or as an architecture design aid tool. The ArchWiz tool should be used in conjunction with one of the following software development process phases:

- **Requirements eliciting phase** - The general quality requirements captured with the ArchWiz tool can be used as input to the first requirement eliciting phase.

- **Design phase** - After the high-level requirements are constructed there should be a break-down activity of the high-level requirements into finer requirements for specific part of the software. During this break-down process the ArchWiz might be used to make sure no important design detail is forgotten.

- **Evaluation phase** - The system is being evaluated for one of the qualities incorporated in the ArchWiz, e.g. usability, security or safety. The reason for evaluating a system might be that the system has experienced problems and the system's stakeholders want to evaluate the system for potential risks related to the problems. The ArchWiz tool might then be used as
an analysis tool for detecting if basic responsibilities related to the usability-, security, and safety scenarios have been considered.

Users should also be able to add and change general requirements in the tool. If for example the user wants to make the tool's requirements product specific, the users can deselect all general requirements not relevant for their product and save a product specific general requirements knowledge base. Users can also share sets of general requirements as a knowledge base and share this knowledge base with each other.

Abstractly seen the ArchWiz has a rather simple system process transformation. Figure 1 shows the system Process Transformation of the ArchWiz. The ArchWiz consists of three processes: Pattern mining, Requirements distillation, and Architectural evaluation. The main inputs are Pattern literature and Requirements coming from the Requirements Engineering process. In the bottom-up approach, the Requirements distillation process takes the requirements of a project and distils the reusable parts of them. The results of this process, i.e. reusable requirements, are stored in the knowledge base using the reusable requirements domain model. The reusable requirements are used for an Architectural evaluation of the Architecture design. In a similar way, the top-down approach uses concepts from pattern literature (e.g. responsibilities) using the Usability, Security, or Safety domain models to store the knowledge in the knowledge base, which in turn is used in the Architectural evaluation process. The Architecture evaluation results in a Todo list, which can be used in the Architectural Analysis and Synthesis processes to improve the Architecture design.

The ArchWiz tool explicitly chooses to focus on a small part of the architecting process and not to try to capture it completely. Consequently, the ArchWiz need to cooperate with a large variety of tools to be used efficiently in the architecting process. Figure 2 shows an overview of these tools. The ArchWiz interacts with the following four types of tools.

- **Requirement tool** - they are used to create and maintain a description of the requirements for a system. Examples of tools are Door, Word, Rational RequisitePro.

- **Architecture description tools** - they are used to create and maintain the architecture description. Examples of tools are Word, Powerpoint, Visio, Whiteboard drawing and Enterprise Architect.

- **Collaboration tools** - they are used to share the created Todo-list in the architecture evaluation to stakeholders of interest. Examples of tools are Lotus Notes, Windows Share Folders and Sharepoint.
- **Project management tools** - they are used to plan the resulting actions from the Todo-list in the development process. Examples of tools are Microsoft Project.

Figure 3 presents the primary uses of the ArchWiz tool. The main actors of the system are software architects or designers, project leaders, quality experts and evaluators. The “Take architecture decision using reusable architecture artefacts” use case is elaborated in figure 4. Figure 5 elaborate on the use case “Access reusable architecture artefacts”.

Figure 2: ArchWiz System Networking.

Figure 3: ArchWiz Primary Use Cases.
Figure 6 shows a state diagram of how the ArchWiz tool is suppose to operate. When the tool is started the user are able to create a new project or to load an existing project. From the beginning of the ArchWiz project the idea was that the user then should be able to select the knowledge base he or she wanted to work with (e.g. usability, security, safety or reusable requirements). The user execute the evaluation project, can save/export the project, can import other knowledge bases, create a todo-list or shut down. It is also possible to edit the knowledge base, meaning that a user can change and add knowledge to the tool.
2.1.1 Changes along the way

The scope of the ArchWiz project was later changed a bit to be more of a requirements tool that only incorporate a general requirements knowledge basis where all software qualities can be represented. A knowledge base existing of reusable requirements for different qualities (the default qualities are usability, security and safety). The user can edit the content of the ArchWiz tool to better suit the project he or she is working on, a todo-list can be created with information about the decisions made for different requirements. The user can comment on each decision. The user should also still be able to import and export other versions of the knowledge base. A feature that was requested from architects at ABB was that the user should be able to split the screen in the tool so that he or she can take decisions on one requirement while viewing the history of decisions at the same time.

2.2. Problem analysis

The thesis work consists of two parts. One part is to help in the development of the new ArchWiz tool. It will probably not be possible to finish the tool but a good foundation should be made. The ones who work on developing the tool until the project ends in December 2009 is me (Caroline Uppsäll), Zafar Bahati (another thesis worker), Anton Jansen at ABB CRC that have some hours each week to help us and Pia Stoll who is the project manager as well as supervisor. The tool is supposed to be implemented in Java/Jigloo and the Eclipse environment which is both a new language and a new environment to learn. There also needs to be some graphical user interfaces implemented which also is new. Time needs to be taken into consideration for learning the language and work environment as well as figuring out how to develop the user interface. It will also take some time to learn the previous work done in the area e.g. the usability supporting architectural patterns.
The second part of the thesis is to come up with suggestions on the domain model for security that contains security entities upon which the ABB software architects can take architectural decisions whether to change the architecture or not to accommodate the security responsibility. To be able to do this a thorough study of the literature in the field needs to be done. With the knowledge gained through the literature together with the ways of using security requirements at ABB today and the way the ArchWiz tool is suppose to work some suggestions of formalisation needs to be addressed.

2.3. Delimitation

It was quite clear from the beginning of the ArchWiz project that the task was to big for the time and resources available. The goal was to get the concepts, the architectural design of the tool and some basic implementation in place by the middle of December 2009 when the project was supposed to be closed.

The delimitation on research was to focus only on the area of security and not to address other qualities. The formalisation is also supposed to be appropriate to the way the tool will be used. From the beginning of the project that was as an evaluation tool but later it was decided that the tool was suppose to be purely for system requirements. This affects the way security will be formalised.
Chapter 3 – Method

This part of the report presents the work flow of this thesis job. It describes which activities have been done in order to get to the results presented at the end of the report, chapter 5.

3.1. Workflow

There have been several different activities in this thesis job, how the workflow looked like are presented in figure 7 below. First it was needed to do research regarding the previous work done within the USAP project, since the ArchWiz is a continuation, and on the ArchWiz project which were the project this thesis was made in. To be able to get a deeper understanding of the software area concerned in the project there needed to be some read up on information regarding software architecture in general and also on architectural patterns that helps architects design different aspects of a software system.

When this knowledge was gained the actual work in the ArchWiz tool started with making iterative versions of mock-ups describing the GUI framework of the tool. These was changed multiple times with the help of the project members, the project owner, an expert in user interfaces and an architect that have worked with the USAP prototype tool described in section 4.3. The tools were during this work stripped of functionality. In the first versions of mock-ups there were quite many features that later were removed. The tool was at the first versions thought of as more of a general tool that it was suppose to be. The later mock-ups was more concerned with specific tasks.

After the mock-ups were made it was time to start learning the implementation environment and language. It was decided that the project was supposed to be implemented using the eclipse environment and the Java language, Jigloo was supposed to be used for the graphical elements. Since both the language and the environment were new, some time was needed to set aside to learn and explore them. When knowledge was gained in the language the project members sat down and discussed the design of the implementation. This was later changed when the focus of the tool was simplified but the diagrams made were still useful.

At this point the implementation of the tool needed to start, the tasks were divided and the implementation of the GUI framework was my tasks to implement. This continued the learning of Jigloo and it took some time before the first implemented version of the GUI framework was done. The time it took was to learn which widgets to use and in what ways, and also how to make the different widgets relate to each other. The first version was made from the mock-ups but the design still changed several times. The process of implementing the GUI framework was enlarged by the slight change of focus in the tool, that made the tool much simpler but the design still needed to be re-maid quite a lot. At the end of the project, in the middle of December 2009, a GUI framework was delivered.

Parallel with the work on the tool I also learned about security in software systems from a variety of books, articles and papers. The subjects were both security in general and security patterns that helps the design and implementation of different security features. After the knowledge was gained I started to work on different suggestions on how to formalise security requirements. These suggestions were sent to the project's owner for feedback. The resulting suggestions can be seen in section 5.2.

During the entire project, meeting were held at least once a week where the project members meet and discussed the progress in the project as well as divided the work to be done. These meetings were very helpful since it gave a good opportunity to discuss problems and ideas.
The original thought with the report was to write it during the whole thesis job period. This started out quite well with some literature studies but then the work of the design, implementation and security research took more and more time and the thesis report was postponed until the project was closed.

Figure 7: Model of the workflow of the thesis work.
Chapter 4 - Related Work and Theory

In this chapter is some literature studies of areas that the thesis concerns. The areas are Software architecture, architectural patterns, security and security patterns. Each subject is relevant background information to the ArchWiz work and security research. The rest of the report refers to these sections when discussing the different parts of these subjects.

4.1 Software Architecture

In developing software there are several different design aspects to consider. One of them is software architecture. But what is software architecture? It is the initial design process that identify the sub-systems of a system and also establishes a framework for how these sub-systems will communicate and be controlled. This initial design process is called architectural design and the output of this design process is a description of the software architecture. [1] and [6] gives the following definition of software architecture.

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them”. - [1] [6]

Architectural design is generally considered as a crucial step to cope with the difficulties of developing large-scale software systems. The software's architecture is a key artefact since it embodies the high-level structures and earliest design decisions. These decisions directly impact on the expected quality attributes and later design or implementation phase. The architecture covers the first design decisions to meet the essential quality requirements for the system [8].

The software architecture encompasses the structure of a larger software system; it is an abstract view of the system. The architecture is concerned with the top-level decomposition of the system into its main components [6]. It’s distilling away details of implementations, algorithms, and data representation and concentrates on the behaviour and interaction of “black-box” elements. The software architecture is the first steps in the design process towards a software system and represents a critical link between the design and requirements engineering process. The architectural design process is concerned with establishing a basic structural framework that identifies the major components of a system and the communication between these components.

4.1.1. Influences to software architecture

An architecture is the result of a set of business and technical decisions. The architect gets influenced in many different ways and those influences will affect the software architectural design. An architect designing a system for which the real-time deadlines is believed to be tight will make one set of design choices, the same architect designing a similar system in which the real-time deadlines are easily meet will make other design decisions. And if the system doesn’t have any real-time deadlines the architect will most likely make another set of decisions. The architectures will be influenced by the system stakeholders, the developing organisation, the background and experience of the architects and by the technical environment. The architect also affects the factors that influence them. Figure 8 shows the architect's influences and how the architect affects the factors that influence her [1] [6].
4.1.2. Why is software architecture important

According to [1] [2] there are three reasons for the importance of software architecture.

- **Communication among stakeholders** - The architecture is a high-level presentation of the system that most, if not all, stakeholders can use as a basis for understanding, negotiation, consensus and communication.

- **System analysis**. Architectural design decisions have a profound effect on whether the system can meet critical requirements such as performance, security, reliability and maintainability.

- **Large-scale reuse** - A system architecture model is compact and shows how a particular system is structured and how its elements work together. This model is transferable across systems. In particular, it can be applied to other systems that have similar quality attributes and functional requirements, and therefore it can promote large-scale reuse.

4.1.3 Architectural activities

So, what activities are involved in creating a software architecture, using that architecture to realise a design, and then implementing or managing the evolution of a system or application? These activities include the following [1]:

- **Creating the business case for the system** - Creating a business case is much more than just assessing the markets need for a system. It is an important step in creating and constraining any future requirements. Questions such as; how much should the system cost? What is its targeted marked? What is its targeted time to market? Will it need to interface with other systems? Etc. must involve the system’s architect. Although these questions cannot be decided only by the architect, he needs to be involved otherwise it may be impossible to achieve the business goals.

- **Understanding the requirements** - There are several different techniques that can be used in the process of eliciting requirements from the stakeholders and understanding them. For example, use cases, finite-state-machine models, formal specification languages and prototypes. The desired qualities of the system being constructed determine the shape of its architecture, regardless of the techniques used to elicit the requirements.

![Figure 8: The architect's influences. After [1].](image)
Creating or selecting the architecture - There might be several different architectural solutions to a system and the architect need to create and select which architectural solution best suits the problem.

Documenting and communicating the architecture - For the architecture to be able to be the backbone of the project's design in an efficient way it must be communicated and documented clearly. Developers must understand the work assignments it requires of them, the testers must understand the task structure and management must understand the scheduling implications it suggests, and so on.

Analysing or evaluating the architecture - In the development of the architectural design there will be different suggestions of how to build the system. The architects need to analyse all the different suggestions. Choosing between these competing designs in a rational way is one of the architect’s greatest challenges.

Implementing the system based on the architecture - This activity is concerned with keeping the development fateful to the structure and interaction protocols constrained by the architecture.

Ensuring that the implementation conforms to the architecture - When an architecture is created and used it goes into a maintenance phase. During this phase it is important to ensure that the actual architecture and its representation remain faithful to each other.

4.1.4. What makes a “good” architecture?

With the same technical requirements for a system, two different architects in different organisations will, because of their different influences, produce different architectures. How can we then know which architecture is the right one? There is no such thing as good or bad architectures; but they can be more or less fit for the system. There are a number of rules of thumb that should be followed when designing an architecture [1]. These rules of thumb are divided into process recommendations and structural recommendations.

The process recommendations are:

- The architecture should be the product of a single architect or a small group of architects with an identified leader.
- The architect should have the functional requirements for the system and a prioritised list of quality attributes that the architecture is expected to satisfy.
- The architecture should be well documented.
- The system’s stakeholders should be actively involved in the reviews of the architecture.
- The architecture should be analysed for applicable quantitative measures (e.g. maximum throughput) and formally evaluated for quality attributes before it is too late to make changes to it.
- The architecture should lend itself to incremental implementation.
- The architecture should result in a specific set of resource contention areas, the resolution of which is clearly specified, circulated and maintained.

The structural recommendations are:

- The architecture should feature well-defined modules whose functional responsibilities are allocated in the principles of information hiding and separation of concerns.
- Each module should have a well-defined interface that encapsulated changeable aspects.
- Quality attributes should be achieved using well-known architectural tactics.
The architecture should never depend on a particular version of a commercial product or tool.

Modules that produce data should be separated from modules that consume data, this increases modifiability.

The architecture for parallel-processing systems should feature well-defined processes or tasks that do not necessarily mirror the module decomposition structure.

Every task or process should be written in such a way that its assignment to a specific processor can be easily changed.

The architecture should feature a small number of simple interaction patterns. This means that the system should do the same thing in the same way throughout.

### 4.1.5. Architectural Patterns, Reference Models, and Reference Architectures

The first of starting points for an architecture leads through a number of intermediate stages to a full architecture with all the appropriate information about the system filled in. Each stage represents the outcome of a set of architectural decisions. Here is the definition of three of them [1] [5]. The relationships among them can be seen in figure 9.

**Architectural Patterns**

An architectural pattern is a description of elements and relations, together with a set of constraints on how they may be used. Pattern helps building the system on the experiences of skilled software engineers. Architectural patterns capture existing, well-proven experience in software development and help to promote good design practice. Every pattern deals with a specific and recurring problem in the design or implementation of a software system [3]. More about what a pattern is and how it can be used can be found in section 4.2.

**Reference Model**

A reference model is a division of functionality together with data flow between the pieces. A reference model is a standard decomposition of a known problem into parts that together solve the problem.

**Reference Architecture**

A reference architecture is a reference model mapped into software elements and the data flows between them. As stated above a reference model divides the functionality, the reference architecture is the mapping of that functionality onto a system decomposition.

Important to mention here is that reference models, architectural patterns and reference architectures are useful concepts that capture elements of an architecture, but they are not architectures. Each one of them is the outcome of early design decisions.

---

*Figure 9: Relationships between Architectural Pattern, Reference Model, Reference Architecture. After [1]*
4.1.6. Architectural structures and views

Today a software system is far too complex to be understood all at once. Therefore we restrict our
attention (at one time) to one or a small number of the software system’s structure. So, when a
system is discussed it is important to make clear which view we are taking in the architecture. A
view is a representation of a coherent set of architectural elements. It consists of a set of elements
and the relations among them. A structure is the set of elements itself, as they exist in software or
hardware.

Architectural structures can be divided into three groups (module structures, component-and-
connector structures and allocation structures [1] [5] [8]), these depend on the nature of the
elements they show. These three structures correspond to the three broad types of decisions that
architectural design involves:

➢ How is the system to be structured as a set of code units (modules)?
➢ How is the system to be structured as a set of elements that have runtime behaviour
  (components) and interactions (connectors)?
➢ How is the system to relate to non-software structures in its environment (i.e. CPUs, file
  systems, networks, development teams, etc.)?

The software structures that are described in this chapter are some of the most common and useful
structures [1], they are shown in figure 10.

![Figure 10: Architectural structures. After [1]](image)

**Module structures**

Here the elements are modules, which are units of implementation. They consider a code-based way
of considering the system. The modules assign areas of functional responsibility. The module
structures helps answering questions such as what is the primary functional responsibility assigned
to each module? What other software elements is a module allowed to use? And what other
software does it actually use? Module based structures include decomposition, uses, layered and
class.

**Component-and-connector structures**

Here the elements are runtime components and connectors. Component-and-connector structures
helps answering questions such as what are the major executing components and how do they
interact? What are the major shared data stores? Which parts of the system are replicated? How
does data progress through the system? Component-and-connector structures include process,
concurrency, shared data and client-server.
**Allocation structures**

The allocation structures show the relationship between the software elements and the elements in external environments in which the software is created and executed. These structures answer questions like what processor does each software element execute on? In what files is each element stored during development, testing and system building? And what is the assignment of software elements to development teams? Allocation structures include deployment, implementation and work assignment.

As mentioned these structures are only a few and there are many more out there. Of course an architect should not use all of them even though most of them will in fact exist in the system. Each of these structures provides a different perspective and design handle on a system, and each is valid and useful. But they are not independent; elements of one structure will be related to elements of another.

Table 1 shows a summary of the above structures. The table lists the relations in each structure as well as why is it useful.

### Table 1: Architectural structures of a system.

<table>
<thead>
<tr>
<th>Software structures</th>
<th>Relations</th>
<th>Useful for</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Module structures</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decomposition</td>
<td>Is a submodel of; Shares secret with</td>
<td>Resource allocation and project structuring and planning; Information hiding, encapsulation; Configuration control</td>
</tr>
<tr>
<td>Uses</td>
<td>Requires the correct presence of</td>
<td>Engineering subsets; Engineering extensions</td>
</tr>
<tr>
<td>Layered</td>
<td>Requires the correct presence of; Uses the services of; Provides abstraction to</td>
<td>Incremental development; Implementing systems on top of “virtual machines” portability</td>
</tr>
<tr>
<td>Class</td>
<td>Is an instance of; Shares access methods of</td>
<td>In object-oriented design systems, producing rapid almost-alike implementations from a common template</td>
</tr>
<tr>
<td><strong>Component-and-connector structures</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>Runs concurrently with; May run concurrently with; Excludes; Precedes; etc.</td>
<td>Scheduling analysis; Performance analysis</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Runs on the same logical thread</td>
<td>Identifying locations where resource contentions exists, where threads may fork, join, be created or be killed</td>
</tr>
<tr>
<td>Shared data</td>
<td>Produces data; Consumes data</td>
<td>Performance; Data integrity; Modifiability</td>
</tr>
</tbody>
</table>
4.2 Patterns

The term patterns are frequently used when dealing with architecture and design. But what is a pattern and how is it used?

Patterns are a way to use the experiences of other developers and collect experiences of skilled software engineers. The patterns capture existing and well-proven experiences in software engineering and help to promote good design practice. Each pattern solves a specific problem that recurs in software development, both in the design and implementation phase. This report will focus on the patterns used in architectural design.

It is very unusual that experts working on a particular problem solve it by making an entirely new solution that stands all alone from existing ones. When they work they often recall a similar problem that they have already solved and uses that as reference when investigating the solution for the problem. Abstracting from these problem-solution pairs and filter out common factors leads to patterns. The term pattern has been defined as follows:

“Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.

As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.

As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.

The pattern is, in short, at the same time a thing which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing: both a description of a thing which is alive, and a description of the process which will generate that thing.” - Christopher Alexander [11].

A pattern can be characterised as:

“A solution to a problem that arises within a specific context”. - [9].

In [14] patterns are described as:

Descriptions of communicating objects and classes that are customised to solve a
With the use of patterns, solutions to specific problems get more available instead of just existing in the minds of experts.

### 4.2.1 Why patterns?

Typically a pattern describes several classes, components and details as well as the relationships among them and their responsibilities. All these elements together solve the problem at hand. This means that patterns identify and specify an abstraction on a higher level than single classes or components. If the names of patterns are carefully chosen they become part of a design language that is widespread. Patterns that have been established in the design language facilitate effective discussions of design problems and their solutions. The discussions get more effective since the need of explaining a solution to a particular problem in detail is no longer necessary. Instead a pattern name can be used and the way this pattern can be used in the architecture can be explained.

Patterns are also a means of documenting software architectures. They can describe the vision the architect has in mind when designing a software system. This helps others to avoid violating this vision when extending or modifying the original architecture or code. If the architect documents that a part of an architecture is made according to a specific pattern others know how to extend and modify that pattern.

Another benefit of using patterns is that they help building complex and various software architectures. As already mentioned each pattern provides a predefined set of components, roles and relationships. It can be used to specify certain aspects of concrete software structures. This means that patterns can be used as building-blocks when constructing more complex systems. Using the kind of solutions that patterns bring both speeds up the development of the design and increase the quality of it. This doesn't say that individual patterns will in all cases be better than other solutions but patterns can at least then be used to help evaluate and assess design alternatives. One thing to keep in mind is that patterns specify the basic structure of the solution to a problem and not a fully detailed solution. In other words, patterns don't provide complete solutions but they help solve problems.

Another benefit or patterns is that they support the construction of software with defined properties since they provide a skeleton of functional behaviour. Since every pattern describes a proven way to handle the problem it addresses there is no need to waste time inventing a new solution when encountering a design problem that is covered by a pattern. If the pattern is implemented correctly it can be relied on.

After this the following definition can be made:

“A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the way in which they collaborate.” - [3].

### 4.2.2 What makes a pattern

One could say that a pattern is built on a three-part schema:

- **Context** - a situation giving arise to a problem.
- **Problem** - the recurring problem arising in that context.
- **Solution** - a proven resolution of the problem.

This schema captures the essence of a pattern independent of its domain and it underlies many pattern descriptions. The above schema makes it easy to understand, share and discuss patterns.
Even though these three parts are closely coupled they have to be clarified in separation to understand the schema in detail.

**Context**
The context of a pattern might be quite general; it describes situations in which the problem occurs. An example is 'developing software with a human-computer interface'. The context can also tie different patterns together. A pattern can occur in the context of another pattern. It is really difficult to create a context for a pattern since it is fairly impossible to determine all situations in which a pattern can be applied.

**Problem**
This part of a pattern schema describes the repeated problem in the context. It begins with a general problem specification that captures the essence. This problem specification is often complemented with a set of forces that is any aspects of the problem that should be considered when solving it. Forces are the key to solving the problem. If the forces are balanced in a good way the solution to the problem will be better.

**Solution**
This part of the pattern schema shows how to solve the problem, or in other words how to balance the forces associated with the problem. This solution might not solve all the forces to the problem, which is important to beware of. Sometimes it focuses on particular forces and leaves others half or completely unresolved. In software architecture the solution includes two aspects. The first is that each pattern specifies a certain structure. This structure addresses the static aspect of the solution. A configuration of elements. It consists of both components and their elements. The second aspect is that every pattern specifies run-time behaviour. This behaviour addresses the dynamic aspects of the solution. This means questions like how do the participants of the pattern collaborate? How is work organised between them? And how do they communicate with each other?

A pattern provides a solution schema and not a fully-specified artefact. The solution in the pattern should be able to be reused in many implementations. But the essence still needs to be unchanged. When an architect chooses to apply a specific pattern in the design it means that the particular structure it provides should be included, but the architect needs to adjust the design and tailor it to the specific needs of the problem which is under design. It is unlikely that two implementations of the same patterns are exactly the same.

The three-part schema can be summarised in the following picture (figure 11) [3].

![Figure 11: Summary of the three-part schema. After [3]](image)
4.2.3 Categories of Patterns

Patterns cover a various range of scale and abstraction and they can be grouped into three categories of patterns.

- Architectural Patterns
- Design Patterns
- Idioms

This thesis report will focus on the architectural patterns but the other two categories will also be explained.

Architectural Patterns

The architectural patterns are used to describe how to solve some overall structuring principles according to which the architecture is built. A definition of architectural patterns is:

“An architectural pattern expresses a fundamental structural organisation schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organising the relationships between them.” - [3]

Architectural patterns specify the structural properties of an application and they are templates for concrete software architectures. The selection of an architectural pattern is therefore a fundamental design decision when developing a software system.

Design Patterns

The design patterns are used when describing the subsystems of a software architecture and the relationships between them. A definition of design patterns is:

“A design pattern provides a schema for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communication components that solves a general design problem within a particular context.” - [3]

These patterns are smaller in scale then the architectural patterns but they tend to be independent from programming languages. Many design patterns provide structures for decomposing more complex services or components.

Idioms

Idioms are on a lower level then design patterns. They deal with the implementation of certain design issues. A definition of idioms is:

“An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language”. - [3]

Most idioms are language specific and they represent the lowest level of patterns. They address the aspects of both design and implementation. Since the idioms are language specific the same idiom look different for different languages.

4.2.4 Relationships between patterns

Just because pattern focuses on providing a solution for resolving one specific problem doesn't mean they are independent from each other, there are many relationships among patterns. For
example the solutions from one pattern can often be implemented with help of other patterns. This kind of relationship between patterns means that each pattern depends on the smaller patterns it contains and on the larger patterns in which it is contained. Another kind of relationship between patterns is variations and combinations. The relationships between patterns are what make it possible to put them together to form a full architectural design for a system that is coherent and consistent. These relationships are very important because without them the patterns would only be able to solve isolated problems with no or very limited effect on the larger design [9].

When creating these relationships between patterns and putting them together, pattern languages are created. As already mentioned pattern supports the solutions of smaller development problems but they cannot themselves support the solution of an entire system. To support the development of a particular family of software systems or application frameworks a broader view must be taken on the patterns. The patterns need to be woven together in such a way that they together resolve the problems of the software development. The result of putting all these patterns together in a way so that they resolve the bigger problem is what pattern languages are. They are not languages in the sense that they are formal languages but they do provide a vocabulary for talking about certain problems [10]. Many patterns form a language, just as words must have grammatical and semantic relations to each other to form a useful language, patterns must have relations to each other to form a pattern language.

Christopher Alexander describes the concept of pattern languages as follows:

> The elements of a pattern language are patterns. There is a structure on the patterns, which describes how each pattern is itself a pattern of other smaller patterns. And there are also rules, embedded in the patterns, which describe the way that they can be created, and the way that they must be arranged with respect to other patterns.

> However, in this case, the patterns are both elements and rules, so rules and elements are indistinguishable. The patterns are elements. And each pattern is also a rule, which describes the possible arrangements of the elements-themselves again or other patterns.

- [12]

### 4.2.5 Description

If a pattern should be easy to understand and discuss it needs to be described in a good way. A good description of the pattern helps to understand the essence of it and also to implement it as well as consider the consequences of using it. There are different ways of describing patterns but it is important that they are described uniformly, if it is, it's easy to compare patterns with each other. The following is a uniform way to describe patterns that are frequently used, most patterns look something like this: Each pattern have a name, that is a descriptive name of the pattern and often also a short summary. Also Known As lists other names under which the same pattern is known. Example, this is a real-world example of how to use the pattern, it also expresses the need for the pattern. Context is as discussed above the situation in which the pattern may be applied. Problem is also discussed above and describes the problem the pattern addresses together with a listing of its forces. Solution this is as stated above the fundamental solution principle underlying the pattern. Structure is a detailed specification of the structural aspects of the pattern. Dynamics is typical scenario that describes the run-time behaviour of the pattern. Implementation, this is guidelines for how to implement the pattern. These implementation details needs to be looked at as suggestions and not rules. Example Resolved discusses the important aspects of resolving the example. These are aspects that are not considered in the solution, structure, dynamics and implementation sections. Variants gives a brief description of variants and specialisation of a pattern. Known Uses is example, taken from existing systems, of the use of the pattern. Consequences lists the benefits and possible liabilities of the pattern. And finally See Also that refers to other pattern that solve similar
problems. And also to pattern that help refine the described pattern [3] [9] [14]. All patterns don't follow this precise template but they tend to have a very similar description as this one.

4.2.6 Patterns in Software Architecture

Architectural patterns are a key concept in the field of software architecture and as already mentioned they are software patterns that offer a well-established solution to architectural problems in software engineering. An architectural pattern expresses a fundamental structural organisation schema for a software system that consists of subsystems, their responsibilities and interrelations. As also mentioned above, architectural patterns are larger in scale than design patterns. The architectural patterns facilitate communication between stakeholders through a common vocabulary. One of the most important aspects of architectural patterns is that they embody different quality attributes such as usability, security, safety, performance, maintainability etc. Different architectural patterns represent solutions to different problems in different qualities. In the early design phase a software architect makes a choice of which architectural patterns best provide the systems desired qualities [13].

Important to remember is that architectural patterns are not architectures as such, it is rather a concept that captures essential elements of a software architecture. Many different architectures may implement the same patterns therefore they share the same characteristics, but they do not solve the same problem.

Another point to be aware of is that there are two different schools of thoughts in the literature with respect to the nature of architectural patterns. One uses the term “architectural patterns” and the other uses the term “architectural style”. Both architectural patterns and architectural styles refers to recurring solutions to problems at the architectural design level, and they both provide a common vocabulary. They both also accept that patterns provide the means to reason for quality attributes and help to document the design decisions taken by the architect. Architectural patterns and architectural styles are complementary mechanisms for encapsulating design expertise. So while they are basically the same they do have some key differences. Styles provide a collection of building-block design elements, rules and constraints for composing the building blocks, and tools for analysing and manipulating designs created in the style. Styles generally provide guidance and analysis for building a broad class of architectures in a specific domain. Patterns focus on solving smaller, more specific problems within a given style, or perhaps multiple styles. Patterns do not need to be architectural. The differences are described more below [7] [13].

Architectural patterns

The architectural patterns perspective, patterns are considered as problem-solution pairs that occur in a given context. It does not only document how a solution solves the problem but also why, the rationale behind the solution. The documentation of a pattern gives much attention to the forces that shape the problem. As described in section 4.2.2 a description on an architectural pattern is based on the context-problem-solution schema and may be extended with further details. As also described above in, in this section, the patterns are meant to work together in pattern languages and have therefore numerous dependencies among each other. If a solution should qualify as a pattern it has to capture some common practices and the solution needs to be non-obvious.

Architectural styles

In the architectural styles perspective the problem or rationale behind it does not get much attention. Styles are described as components, connectors and issues related to control and data flow and the constraints upon them. Its attention is drawn to architectural configurations, semantics of the style and potential architectural analysis that can be performed in the system built on the style. In this way the styles become more concrete and focused. This leads to multiple variations of the same pattern in order to solve specialised design problems.
As discussed in the precious sections of this chapter patterns are the focus in this report. In theory an architectural pattern is on a higher level than design patterns but in reality it is quite hard to draw a line between the two. It is in many cases not clear when a pattern is “big” enough to be considered as architecture. So it is not uncommon that design patterns are referred to or used as architectural patterns. What is architecture and what is design is often decided upon the designers or architects point of view. Examples of some architectural patterns are [3]:

- **Layers** - Helps to structure applications that can be decomposed into groups of subtasks in which each group of subtask is at a particular level of abstraction. An example is the networking protocols.

- **Model-View-Controller** - Divides an interactive application into three components (model, view and controller). The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

- **Pipe and filer architecture** - Provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows building families of related systems.

- **Blackboard system** - Useful for problems for which no deterministic solution strategies are known. In Blackboard several specialised subsystems assemble their knowledge to build a possibly partial or approximate solution.

- **Client-Dispatcher-Server** - Introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.

### 4.3 Usability-Supporting Architectural Patterns (USAP)

There exist a number of important quality attributes that must be considered in the architecture of a system. Some of these qualities are availability, modifiability, performance, security, safety, testability and usability.

During 2007 and 2008 ABB CRC and the Software Engineering Institute (SEI)/Carnegie Mellon University in Pittsburgh, USA conducted a project called the USAP (Usability-Supporting Architectural Patterns) Project. The research in the USAP project was to find a way to formalise usability as an architectural concern. This formalisation was then supposed to help architects think about usability early in the design process. The project resulted in contemporary research that contributed to the field of usability and software architecture in the domain of Software Engineering. The project also resulted in a web-based tool at prototype state. The work with the USAP project and the results are described in this chapter.

#### 4.3.1. USAP project

There exist different techniques for qualities such as performance, reliability and maintainability among others when it comes to designing the architecture of a system. Usability is also a very important quality but for some reason it gets little attention in the architectural phase of the development process. Usability has been treated as a subquality of modifiability which means that it is treated in such a way that the usability issues that arise during testing can be handled with just some modifications. This is a problem, because many usability concerns reach deep into a systems architecture and can therefore not be treated by just separating the interface from the functionality. The effect of treating usability as a problem in modifiability simply postpones many of the usability
related requirements until the end of the development cycle where there might not be time or money to consider [15]. If the requirement of implementing some usability feature is discovered late in the development process the costs will be much bigger then if it was incorporated in the architectural design from the beginning. The consequence of this is that the system developed is less usable then it could be or more expensive in development then it could be if the usability concern was built into the architecture [16].

The risk of finding usability problems that requires architectural work late in the development cycle was not acceptable and ABB decided to use usability-supporting architectural patterns (USAPs) in collaboration with CMU. The decision to use USAPs was based on the fact that they use generic usability scenarios common in complex systems. From the scenarios they conduct generic software architecture responsibilities. With this approach ABB expected to address some of the major usability concerns in the design.

The main goal for ABB when applying the USAP technique was to incorporate usability support early in the design process in order to build in the support in the core architecture.

The question of the USAP project was: How can we best support usability early when the product prototypes cannot be user tested until years after the architecture design is to be completed [17].

The USAP project resulted in a new approach to provide usability knowledge to software architects early in the design process. A set of specific usability scenarios that have software architectural implications were identified. These scenarios cover much of what people typically mean by usability. To achieve the scenarios, architectural patterns were identified for each of the scenarios. The scenarios and patterns were then organised [18].

A USAP is a pattern at the level of software architecture responsibilities. It provides instructions on how to achieve some specific usability scenarios. Some examples are: cancelling a long-running commands and supporting personalisation of the user interface. Here a distinction need to be maid, these are software architecture patterns and not usability patterns. The difference is that usability patterns describe user interface patterns such as an organisation's look and feel. Where on the other hand architecture patterns describe solutions to a specific problem in the software design. A USAP include six different types of information:

- A scenario that briefly describes the situation that the USAP is intended to solve.
- A description of the condition under which the USAP is relevant.
- A characterisation of benefits to the user from implementing the USAP.
- A description of the forces that impact the solution.
- An implementation-independent description of the solution, in other words responsibilities of the software.
- A sample solution using UML-style diagrams.

According to [17] the use of USAPs has, in laboratory experiments, been shown to significantly improve a software architecture design.

The researchers presented 19 usability scenarios that were possibly relevant to the domain. The entire list can be seen in [17]. The team at ABB prioritised the scenarios and made a decision to focus on two of them. The team also added an additional scenario. The three chosen scenarios were:

- User Profile
- Alarms, Events and Alerts
- Environment Configuration
4.3.2. USAP Pattern Language

The team made USAPs for each of these three scenarios but when considering the three USAPs together it was clear that there were a large amount of redundancy in the responsibilities. The responsibilities in the different USAPs were also grouped into similar categories. This led to the construction of a pattern language [19] that can be seen in figure 12. The pattern language defines relationships between USAPs, it also defines sets of responsibilities that can lead to code reuse.

Figure 12: Relationships between USAPs. After [19]

As can be seen in figure 12 the pattern language has two types of USAPs. “End-User USAPs” and “Activities” also referred to as Foundational USAPs. The End-User USAPs follow the six point structure above and their purpose (from a user’s point of view) can be expressed in a small scenario. The End-User USAPs have conditions under which they are relevant, they can express benefits for the user and they require fulfilment of software responsibilities in the architectural design. In the USAP prototype tool the End-User USAPs are “User profile”, “Alarms, Events and Alerts” and “Environment Configuration”. The Activities does not have the same structure, they act as frameworks to support the construction of End-User USAPs. An example is that all of the three End-User USAPs have an authoring part and an execution part. This means that they are specialisations of the Authoring Activity and the Execution with Authored Parameters Activity. The different Activities in the USAP prototype is Authoring, Execution with authored parameters, Logging and Authentication. The Activity also has relationships amongst each other. As shown in figure 12 there are three different kinds of relationships [19]:

- **Generalisation relationship** - this relationship say that the Activity is a generalisation of a part of the End-User USAP. The End-User USAP passes parameters to the Activity. The End-User USAP also defines the values of conditionals, if there are any in the responsibilities of the Activity.

- **Uses relationship** - This relationship also passes parameters but it is only between the Activities them self.

- **Depends-On relationship** - This relationship implies a temporal relationship. An example is that a system cannot execute with authored parameters if those parameters have not yet been authored.
4.3.3. The USAP taxonomy

Each of the Activities has a reasonable number of responsibilities and are also divided into activity tasks for ease of understanding. The tasks are smaller parts of the activity. The Activities are Authoring, Execution with authored parameters, Logging and Authorization. They are described with purpose and a justification.

**Authoring**

*Purpose:* The purpose of these responsibilities is to allow specification of the behaviour of the system in certain ways under certain circumstances.

*Justification:* Users want to control the behaviour of the system in certain ways under certain circumstances without having to set it up every time. The system needs a specification of parameters to determine its behaviour in these circumstances. Therefore the user must author a specification of parameters that will subsequently be used upon execution (Execution with authored parameters).

*Tasks:*
- Create a specification
- Save a specification
- Modify a specification
- Delete a specification
- Exit the authoring system

**Execution with authored parameters.**

*Purpose:* The purpose of these responsibilities is to allow a system to use a specification of parameters to determine its behaviour in the areas in which the parameters apply.

*Justification:* Users want to control the behaviour of the computer in certain ways under certain circumstances that they have previously specified (Authored).

*Tasks:*
- Access the appropriate specification
- Use specified parameters
- Additional responsibilities

**Logging**

*Purpose:* The purpose of these responsibilities is to retain and examine selected information generated during execution.

*Justification:* Some information known only during execution needs to be retained either for debugging or audit-trail purpose.

*Tasks:*
- Specify the items to be logged
- Log items during execution
- Post-processing

**Authorization**

*Purpose:* The purpose of these responsibilities is to identify and authenticate users (human or other systems) of the system.

*Justification:* Users must be authorised when security or personalisation is important.

*Tasks:*
- Authentication
- Permission
- Log of

Each responsibility is described in text. It has a rationale and implementation details also expressed in text format. The implementation details describe what to do with specific parts of the system to fulfil the requirement in the architecture. First the project team expressed the implementation detail with a UML solution but after a laboratory experiment [20] the UML solution was changed to a textural decision based upon that the architects partitioned in the experiment did not like the UML-style sample solution since they then felt pressured to use the specific pattern that the sample
solution was based upon. The textural implementation details now contain description of implementation details that express structural and behavioural parts of a solution. Each USAP is a pattern of responsibilities and each responsibility is a pattern of implementation suggestions. The responsibilities can be thought of as requirements on the architecture. The emphasis of the USAP prototype is on the responsibilities.

### 4.3.4. The USAP prototype

In the USAP prototype, for each responsibility the architect need to make a decision of whether the responsibility is not yet considered in the architectural design, whether the architectural design needs to be modified to address the responsibility, whether the architectural design already addresses the responsibility or whether the responsibility is not applicable to the system to be designed. The architect can also click a check box if the architect feels the need to discuss the status of the responsibility. A screenshot of the USAP prototype can be seen in figure 13. The decision needs to be done to the responsibility with respect to every scenario (End-User USAP) that are related to that Activity. The architect can then produce a todo-list and which is a list of the responsibilities that have the decision must modify architecture, not yet considered or must discuss.

![Figure 13: Screenshot of the USAP prototype web tool.](image)

1 - Activity, 2 - Activity task, 3 – Responsibility, 4 – Rational, 5 - Implementation details, 6 - Scenario, 7 - Decision.
The task flow of the USAP model is shown in figure 14.

![USAP task flow model](image)

**Figure 14: USAP task flow model.**

The project team expected the result of this lightweight reminder to increase the coverage of responsibilities, which is concerned with the quality of the architecture solution. A conclusion of the work [21] with the USAP project was that the USAP should be an efficient tool that allows the usability engineers and the software engineers to meet early and discuss the usability requirements and their impact on the software architecture. With well formulated and general responsibilities the project team feels that the USAP becomes reusable.

### 4.4 Security

This report has already discussed one system quality, usability. Another very important quality is security. So what is security and why is it so important? Security is a system attribute that reflects the ability of the system to protect itself from external attacks that may be accidental or deliberate. Since more and more systems are connected to the internet, security has grown more and more important in computer systems. Security is a measure of the system's ability to resist unauthorised usage while still providing its services to legitimate users [1].

> Security is the degree to which malicious harm to a valuable asset is prevented, reduced, and properly responded to. Security is thus the quality factor that signifies the degree to which valuable assets are protected from significant threats posed by malicious attackers. - [26]

As the use of computer systems and internet grows rapidly the need for developing secure software gets more and more important. An attempt to breach security is called an attack or a threat and it can take a number of different forms, for example unauthorised attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users. Without security precautions it is almost inevitable that attackers will compromise a networked system. They may steal confidential data or disrupt the services offered by the system. Therefore system security is an increasing and very important part of software engineering. The concern to security engineers is how to develop and maintain systems that can resist malicious [2]. More on threats are discussed in section 4.4.6.

In the broad sense security is the totality of all services and mechanisms that protect an enterprise. A basic aim of security is to isolate or restrict actors (human or other processes) from having unrestricted access to the resources. The isolations and restrictions are often...
provided through a variety of services and mechanisms. The field of security is very diverse, on one hand it involves protection of a system and on the other hand it involves decision-making processes that determine who may have access to certain system resources, where they may have access and when. There are different kinds of security [9]:

- **Procedural security measures** – these include administrative security or management constraints. They include operational, administrative, and accountability procedures.
- **Environmental or physical security measures** – these include all elements of personnel and physical security.
- **Personnel security** – this involves the policies and procedures required to establish authorization.
- **Physical security** – this protects all resources and assets from physical hazards.
- **Technical security measures** – these include communications, data, and automated information system security. The later include protecting the authenticity and integrity of message traffic, protection of all hardware, software, and firmware, and protection of the data handled by the system.

All systems might not need all of these different kinds of security but they often need at least some of them together.

**The four W’s**

Security flaws and attacks can exist across many application components in a system and this reinforces the importance of end-to-end security, and not just security for a single component. But the question is how should architects begin establishing end-to-end security? One way to start is by answering the four W’s [22].

- Which application are we protecting?
- Who are we protecting the application from?
- Where should we protect them?
- Why are we protecting them?

**Which application are we protecting?**

Some applications, such as business applications, require protection from unauthorised access and therefore use different levels of access control depending on how big the threat is determined to be. It is important to identify and determine which application resources that need security and access control.

**Who are we protecting the application from?**

Some applications include personal data or sensitive information, these systems needs to be protected from users other then the owners. In other worlds only authorised users should have access to the information. It is important for security architects and developers to consider the possibility of protecting these resources by categorising users by rights and privileges granted. Users can then be grouped based on their access rights. During the entire life-cycle of the system it is important to log, monitor and audit user access and application events.

**Where should we protect them?**

It is not sufficient for end-to-end security to know which applications and resources that need protection and from whom, where to protect them is the next question. The protection should address all the aspects of an application and its resources.
Why are we protecting them?

A motivation to make all of this security designing and implementation is that a fault in the system of for example a business application may cause great damage to the organisation, or to individual clients. Understanding what can happen from different security breaches will help the architects and developers to protect the application and resources properly.

4.4.1 Security Concepts

In the literature regarding security there exist a number of concepts that the reader needs to have knowledge about to be able to follow the discussions and descriptions about the subject. In this section are some of the most common concepts explained. The concepts presented are gathered from a number of books, articles and papers [2] [22] [24] [25] [26] [27] [28] [38].

- **Asset** - A system resource that has a value and has to be protected. Assets can be people, properties (e.g. data, hardware, software and facilities), and services.
- **Attack** - Also commonly called security breach. An attack is an attacker's unauthorised attempt to cause harm to an asset. It may be successful or unsuccessful. It is an exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause damage.
- **Attacker** - An attacker is an agent (e.g. humans, programs, processes, devices, or other systems) that causes an attack due to the desire to cause harm to an asset.
- **Control** - A control is a protective measure that reduces a system's vulnerability. One example of a control is encryption that reduced a vulnerability of a weak access control system.
- **Exposure** - This is the possible loss or harm that could result from a successful attack. This can be loss or damage to data or can be a loss of time and effort if recovery is necessary after a security breach.
- **Harm** - This is a negative impact of associated with an asset due to an attack.
- **Security** - Security is a quality attribute, it is the degree to which malicious harm to a valuable asset is prevented, reduced, and properly responded to. Security is thus the quality factor that signifies the degree to which valuable assets are protected from significant threats posed by malicious attackers.
- **Security attribute** - Also called properties or objectives are for example confidentiality, integrity and availability.
- **Security Components** - Security has three components, requirements, policy and mechanisms.
- **Security Goal** - A security goal is a quality goal that documents a target level of security or one of its subfactors.
- **Security Mechanisms** - Also called countermeasures. The security mechanisms or countermeasures is an architectural strategic decision that helps fulfil one or more security requirements and/or reduces one or more security vulnerabilities. The mechanisms enforce the policies, their goal is to ensure that the system never enters a disallowed state. The mechanisms answer the question “What tools, procedures, and other ways do you use to ensure that the policies are followed?” This concept is explained further in section 4.4.3.
- **Security Policy** - Also called trust model. This is a quality policy that mandates a system-specific quality criterion for security. The security policy is the basis for the security implementation of the system. This model identifies who will be trusted with what access to
which system resources. The security policy defines the meaning of security and it answers the question “What steps do you take to reach the goals (requirements)?”. This concept is explained further in section 4.4.3.

- **Security Requirements** - The security requirements are quality requirements that specify a required amount of security. They define security goals, they answer the question “What do you expect security to do for you?”.

- **Security Risk** - This is the potential risk of harm to an asset. It depends on the negative impact of the harm and on the likelihood of that the harm occurs. Risk analysis is the process of describing threats, their impacts and possible consequences as well as the probability and frequency of occurring.

- **Threat** - Circumstances that have potential to cause loss or harm. One can think of threats as system vulnerabilities that is subjected to an attack. It is a general condition, situation or state that may result in one or more related attacks.

- **Vulnerability** - Is a weakness in a computer based system that may be exploited to cause loss or harm.

These concepts have relationships to each other, some of these relationships are shown in figure 15 [26].
4.4.2 Risk management

Software security can be thought of as risk management. When we understand in which context a trade-off is being made we can start taking intelligent decisions about the software. These decisions will be influenced by business objectives and other central concerns [29]. Security risk management is important for effective security engineering and it is concerned with assessing the possible losses that might occur from attacks on assets in the system. Risk management also balance these losses against the costs of security procedures that might reduce these losses. In other world the management need to find a balance of what security features are economic to have in the system by looking at the cost of the potential loss and the cost of implementing the security feature. If an attack is unlikely, protecting against it will have lower priority than protection against an attack that

![Diagram of security and risk management concepts](image)

Figure 15: Concepts that influence and are influenced by security requirements. After [26]
is more likely to happen. But if the unlikely attack has much bigger consequences if it occurs then the more likely attack, a bigger effort will be put into protection against the unlikely attack. This is a question of trade-offs and balance, which risks are the management willing to take? This is thereby a business issue and not a technical issue, this means that the software engineers should not decide what controls should be included in a system. It is up to the management to decide if they want to take the cost of implementing the security or the potential cost of exposing the system. What the engineers do in this process is to provide technical information that helps the management in their decision. A very important input this decision process is the companies’ security policy. This applies to all systems and decides what should and what should not be allowed to happen. This is affected by the laws and customs that exist [2] [23].

A good security system strikes a balance between what is possible and what is acceptable through the risk management process. Risk management consists of risk assessment, risk reduction and risk acceptance. And there are four different ways of managing a risk. Accept the risk – the risk is too low and so costly to mitigate that it is worth accepting. Transfer the risk – Transfer the risk to somebody else via insurance, warning etc. Remove the risk – Remove the system component or feature associated with the risk if the feature is not worth the risk. Mitigate the risk – Reduce the risk with countermeasures [30].

### 4.4.3 Security policy and mechanisms

It is important to be aware of the difference between a security policy and a security mechanism.

> A security policy is a statement of what is, and what is not, allowed.

> A security mechanism is a method, tool, or procedure for enforcing a security policy. - [23]

Just because a mechanism is a tool, procedure or method does not mean that it needs to be technical, for example requiring proof of identity before changing a password. A policy often require some procedural mechanism that technology cannot enforce. A policy may be presented as a list of allowed and disallowed states, meaning secure and nonsecure states. They normally describe in plain text what a user, for example, are allowed to do in the system. The security mechanisms job is to enforce the policy and to ensure that the system never enters a state that is disallowed according to the policy. If the set of mechanisms correctly implements the security policy is a question of assurance. A security policy considers all relevant aspects of confidentiality, integrity and availability. When it comes to confidentiality, they identify those states in which information leaks to those not authorised to receive it. This should not only include the leakage of rights but also the illicit transmission of information. Regarding integrity a policy identifies authorised ways in which information may be altered and entities authorised to alter it. These kinds of policy describe the conditions and manner in which data can be altered. With regards to availability a security policy describes what services must be provided, it may present parameters within which the services will be accessible.

### 4.4.4 The basic components

Security is a composite of the attributes (also referred to as properties) of confidentiality, integrity and availability [24] [6]. Threats to the confidentiality of the system and its data means that information can be visible to people or programs that do not have authorised access to the information. Threats to the integrity of the system and its data can damage or corrupt the software or its data. Threats to the availability of the system and its data can restrict access to the software or its data for authorised users [2].
Confidentiality

Confidentiality is the concealment of information or resources, it is the concept of protecting sensitive data from being viewable by unauthorised entities. In many computer systems it is important to keep information secret, for example in fields such as government and industry. Access control mechanisms support confidentiality. A wide variety of data falls under the category of sensitive data. It can be data that is illegal to compromise such as patient's medical histories or a customer's credit card number. Another kind of sensitive data is data that may reveal too much information about the application. Securing the confidentiality of data reduces the probability of exposure if the application is under attack [22] [36].

There are a number of mechanisms for preserving confidentiality, one of them is cryptography. Other system-dependent mechanisms can prevent processes from accessing information illegally. However, data protected by these controls alone can be read when the controls fail or are bypassed. Even though they might protect the secrecy of data more completely then cryptography, if they fail the data becomes visible.

Confidentiality does not always apply to the data itself, sometimes it applies to the existence of the data. Sometimes knowing that some information is there is more revealing then actually knowing what the information is. Another aspect of confidentiality is resource hiding. Sites, for example, often wish to conceal their configuration as well as what systems they are using. Organisations may for example not wish others to know about specific equipment.

All the mechanisms that enforce confidentiality require supporting services from the system. The assumption is that the security services can rely on the kernel, and other agents, to supply correct data. This means that assumptions and trust underlie the confidentiality mechanisms.

Integrity

Integrity is the degree of trustworthiness of data or resources, it is often expressed in terms of preventing improper or unauthorised change. It is the concept of ensuring that data has not been altered by an unknown entity during its transit or storage. It is for example possible to modify an email containing sensitive data before it reaches the receiver. Checking the data integrity ensures that data has not been compromised [22] [36].

Integrity include data integrity, being the content of the information. And origin integrity, being the source of data, this is often called authentication. The source of the information may bear on its accuracy and creditability and on the trust that people place in the information.

Integrity mechanisms fall into two classes [23]: prevention mechanism and detection mechanisms. The prevention mechanisms job is to maintain the integrity of the data by blocking any unauthorised attempts to change the data or any attempts to change the data in unauthorised ways. Good authentication and access control will generally stop break ins from the outside. Preventing breakins from the inside on the other hand requires other kind of controls. Detection mechanism do not try to prevent violations (that's the preventive mechanisms job), they simply report that the data have been tampered with and are no longer trustworthy. These kinds of mechanisms may analyse system events such as system or user actions and thereby detect problems. A more common kind of detection mechanism is to analyse the data itself to see if the required or expected constraints still holds.

There is quite a difference between working with integrity and confidentiality. With confidentiality, the data i ether compromised or not. Integrity includes both the correctness and the trustworthiness of the data. Things that effect the integrity of data is the origin of the data, how well the data was protected before it arrived and how well the data is protected where it is now. One thing that makes evaluating the integrity very difficult is that it relies on assumptions about the origin of the data and about trust in that source.
Availability

Availability refers to the ability to use the information of resources desired, this is an important aspect of reliability. An unavailable system is just as good as no system at all, because none can use it. The question to be raised here is how availability is related to security? The aspect of availability that is related to security is that someone may deliberately try to arrange so that data or services gets unavailable, by for example, changing the access rights [23] [36].

Denial of service attacks (more discussed in section 4.4.6) are attempts to block availability and they can be the most difficult to detect, because a deliberate attempt to make a resource unavailable may simply look like or be an atypical event, and in some environments they may not even seem that atypical.

Designing a system for availability involves thinking about planned and unplanned events that can occur due to the nature of the system itself. When designing for availability it is also important to think about the environment in which the system exists. This has become more and more important as the systems gets more public [25].

4.4.5 Security tactics

There exist three distinct tactics for security (also called goals or approaches) prevention or resistance, detection and recovery. All of these three categories are important and the strategies may be used together or separately. Depending on the status of the different states in the security policy, prevention, detection and recovery are very helpful to think about. One way of putting these tactics into a more familiar analogy is to say that putting a look on your door is a form of prevention or resisting an attack. Having an alarm or a motion sensor inside of the house is a form of detecting an attack. And having insurance on the house is a form of recovering from the attack. A summary of the tactics can be seen in figure 16 [1].

![Figure 16: Summary of the tactics for security. After [1]](image)

Resisting attacks (prevention)

Resisting an attack or prevention means that an attack will fail. This typically involves implementation of mechanisms that users cannot override and that are trusted to be implemented in a correct way. This means that the attacker should not be able to defeat the mechanism by changing it. Preventative mechanisms often interfere with the use of the system, sometimes so much that they hinder normal use. But some simpler preventive mechanisms have become widely acceptable, for
example the use of passwords [23].

Some preventive mechanisms are [1] [31]:

- **Authenticate users** - This ensures that a user or process is actually who it claims to be. Ways to implement this is by using password, one-time passwords, digital certificates and biometric identification to name a few.
- **Authorise users** - This ensures that an authenticated user has the rights to access and modify either data or services. Classes or users can be defined by user groups, user roles, or by lists of individuals.
- **Maintain data confidentiality** - As discussed in section 4.4.4, data should be protected from unauthorised access. This kind of confidentiality is often achieved by using some sort of encryption to the data and to the communication links.
- **Maintain integrity** - As discussed in section 4.4.4, data should be delivered as intended. It can have redundant information encoding in it, such as checksums or hash results.
- **Limit exposure** - Attacks typically depend on exploiting a single weakness to attack all data and services on a host. This can be protected against by for example design the allocation of services to hosts so that limited services are available on each host.

**Detecting attacks**

If a system is attractive enough to an attacker, it will most possibly occur some kind of security breach. This means that an important concern to security is to be able to detect these security breaches and also recover from them (recovering from attacks is described later in this section) [25]. The detection tactic is most useful when an attack cannot be prevented, but it can also indicate how effective a preventive measure is. Detective mechanisms accept that an attack will occur, its goal is to determine that the attack is on its way or that it has already occurred, and of course to report this. The detection of an attack is often thought of as intrusion detection [1]. The attack may be monitored to gather information about its nature and results. Mostly detection mechanisms monitor different aspects of the system, looking for actions or information that indicates an attack. A good example of this is a mechanism that gives a warning when a user enters an incorrect password three times. Depending on the system, the login may still continue but in those cases an error message is registered in the system log reports. A big drawback of the detection mechanisms is that they do not prevent compromise of parts of the system [23].

**Recovering from attacks**

Recovery together with detection is unlikely to be entirely technical, it involve people and processes as well as the technology required to spot a security breach and make an appropriate reaction to it [25]. Recovery has two different forms, the first is to stop an attack and to assess and repair any damage caused by the attack, and the second is that the system continues to function correctly while an attack is underway [23].

The second kind of recovery is quite difficult to implement because of the complexity of computer systems. It draws on techniques of fault tolerance as well as techniques of security, it is typically used in safety-critical systems. It differs from the first kind of recovery in a very important way, in the second kind of recovery the system will never function incorrectly. But to do so the system may disable nonessential functionality. This type of recovery is often implemented in a weaker form, the system detects incorrect behaviour automatically and then tries to correct the error.

Recovery also involves identification and fixing of the vulnerabilities used by the attacker to enter the system, this is important since the attacker may return and try again. In most, if not all cases of attacks, the system's functionality is inhibited by the attack. By the definition, recovery needs to resume the system to correct operation.
4.4.6 Threats

The policy discusses in section 4.4.3 defines the security constraints the system requires, the threats the system faces are the possible ways the security constraints might be breached by an attacker [25].

Most system requirements are based on higher level goals, they are stated in terms of what must happen. Security requirements on the other hand are driven by security threats, they are expressed in terms of what must not be allowed to happen. Identifying threats helps develop realistic and meaningful security requirements [32] [30]. Security requirements should be based on an analysis of the assets and services that should be protected, and on an analysis of the security threats from which these assets and services should be protected. The relationship between assets that are vulnerable to threats, which in turn necessitate security requirements that requires mechanisms that counter the threats to security and thereby also protect the assets and services are shown in figure 17 [33]. So, to be able to identify and develop realistic and meaningful security requirements it is a very good idea to start with identifying the threats to the system. This is very important because if the security requirements are faulty, the definition of security for that system is faulty, and therefore the system cannot be secure. Properly identifying the threats and an appropriate set of countermeasures reduces the ability for attackers to misuse the system [30]. As explained, a threat is a potential violation of security and the violation need not actually occur for there to be a threat. The fact that a violation might occur means that those actions that could cause it to occur must be guarded against. Those actions are called attacks and the one who performs them are called attackers (could be people, processes and other systems to name some) [23].

Figure 17: Security Threats, Requirements and Mechanisms. After [33]

Threats are often grouped into different classes. The four broad classes are: disclosure – unauthorised access to information, deception – acceptance of false data, disruption – interruption or prevention of correct operation and the fourth is usurpation – unauthorised control of some part of the system. Below are some important threats to system security [23] [34] [35]. In figure 18 is a diagram of how the different threats below are connected to these different classes and also to the three basic components, confidentiality, integrity and availability.

Snooping

This is a kind of disclosure, it is the unauthorised interception of information. It is passive, meaning that some entity is listening to or reading communication or information. It could also mean that the entity is browsing through files of system information. Confidentiality countermeasures counter this form of attack.

Modification or alteration

This is an unauthorised change of information and it covers three classes of threats, deception, disruption and usurpation. Deception might be the goal of the treat since some entities rely on the modified data to determine which action to take, or in which incorrect information is accepted as correct and released. The threat of disruption and usurpation arise if the modified data controls the operation of the system. Modification attacks are active since it changes information. Integrity
services counter these kinds of threats.

**Masquerading or spoofing**

This is a threat where one entity does an impersonation of another entity and it is a form of both deception and usurpation. It lure a victim into believing that the entity with which it is communication is a different entity. Masquerading or spoofing is primarily deception it could also be usurpation since it is used to usurp control of a system by an attacker impersonating an authorised manager or controller. Integrity services can counter these threats, but in this context they are called authentication services instead of integrity services.

But all forms of masquerading or spoofing should not be disallowed. Delegation occurs when one entity authorises another entity to perform functions on its behalf. The difference between delegations of masquerading is important to keep in mind. Delegation should in many systems be allowed. In delegation all parties are aware of the delegation but in masquerading no other parties except from the attacker are aware of the attack taking place. Masquerading is a violation of security but delegation is not.

**Repudiation of origin**

Repudiation of origin is a false denial that an entity sent or created something, and it is a form of deception. An example. Suppose a customer sends a letter to a vendor agreeing to pay a large amount of money for a product. The vendor ships the product and then demands payment. The customer denies having ordered the product and by law is therefore entitled to keep the unsolicited shipment without payment. The customer has repudiated the origin of the letter. The attack succeeded if the vendor cannot prove that the letter came from the customer. Integrity mechanisms deal with this kind of threats. A variant of repudiation of origin is denial by a user that he created specific information or entities such as files.

**Denial of receipt**

This threat is a false denial that an entity received some information or message. Denial of receipts is a form of deception. An example related to the one for repudiation of origin could be: A customer orders an expensive product, but the vendor demands payment before shipment. The customer pays, and the vendor ships the product. The customer then asks the vendor when he will receive the product. If the customer has already received the product the question constitutes a denial of receipt attack. Then vendor can defend against this attack only by providing that the customer did, despite his denials, receive the product. Integrity and availability mechanisms guard against these kind of attacks.

**Delay**

Delay means a temporary inhibition of a service and is a form of usurpation but it can also play a supporting role in deception. Typically, delivery of a message or service require some time, if an attacker can force the delivery to take longer time, the attacker has successfully delayed delivery. It is a form of usurpation because it requires manipulation of system control structures (such as network or server components). If an entity is waiting for an authorization message that is delayed, it might query a second server for the authorization. Even though the attacker might not be able to masquerade as the primary server it might be possible for it to masquerade as the second server. Availability mechanisms can counter this threat.

**Denial of service**

Denial of service means a long-term inhibition of service and is a form of usurpation, although it is often used with other mechanisms to deceive. In a denial of service attack the attacker prevents a server to perform or provide a service. It may occur at the source by preventing the server from obtaining the resources needed to perform its function, at the destination by blocking the communication from the server, or along the intermediate path by discarding messages from either
the client, the server or both. A denial of service attack is the same as a delay attack that is infinite and availability mechanisms counter this threat to. Both delay and denial of services can be both an attempt to breach the security system and a product of environmental characteristics (and not actions of an attacker).

4.4.7 Kinds of security objectives

There exist a number of different security objectives. Some objectives of security requirements that fall into these groups are: Ensure that users and client applications are identified and that their identities are properly verified. Ensure that users and client applications can only access data and services for which they have been properly authorised. Detect attempted intrusions by unauthorised persons and client applications. Ensure that communications and data are not intentionally corrupted. Ensure that parties to interactions with the application or components cannot later repudiate those interactions. Ensure that confidential communications and data are kept private. Enable security personnel to audit the status and usage of the security mechanisms. Ensure that applications and centers survive attack, possibly in degraded mode. Ensure that centers and their components and personnel are protected against destruction, damage or theft. Ensure that system maintenance does not unintentionally disrupt the security mechanisms of the application, component or center [32]. Below are some security objectives that correspond to the requirements above [22] [26] [29] [32] [35]. Some of them have been touched upon and discussed in previous sections such as the basic components in section 4.4.4. Be aware that this kind of requirement should not be specified in terms of the types of security architecture mechanisms that are typically used to implement them.

![Figure 17: How some common threats relate to classes and basic components.](image-url)
Identification
This is security requirements that specify to which extent a business, application, component, or center shall identify its externals, for example human actors and external applications, before interacting with them. These kinds of security requirements are often insufficient on their own and they need to have some authentication requirement as well. They also need to be considered with requirements for privacy.

Some examples of identification requirements are:
- The application shall identify all of its client applications and human users before allowing them to use its capabilities.
- The application shall not require an individual user to identify him or her multiple times during the same session.

Authentication
This is a security requirement that specifies the extent to which a business, application component or center shall verify the identity of its externals before interacting with them. The typical objective of authentication requirements is to ensure that externals are actually who they claim to be, this avoids compromising security to impostors. These requirements depend on the identification requirements. If identification requirements are specified, then authentication requirements should also be specified. The authentication requirements are necessary for authorization requirements. Some mechanisms used to implement authentication requirements also implement identification requirements. Identification and authentication are very close and are therefore sometimes grouped together.

Examples of authentication requirements are:
- The application shall verify the identity of all its users before allowing them to use its capabilities.
- The application shall verify the identity of all its users before allowing them to update their user information.

Authorization
This security requirements specify the access and usage privileges of authenticated users and applications. The objective of this kind of requirements are typically to ensure that specific authenticated externals can access specific application or components capabilities or information if and only if they have been explicitly authorised to do so. This prevents unauthorised users from obtaining access to inappropriate or confidential data and from requesting the performance of inappropriate or restricted services. Authorization requirements depend on authentication requirements and thereby also on identification requirements. Authorization can be granted to individual persons or applications and/or to groups of related persons or applications.

Examples of authorization requirements are:
- The application shall allow each user to obtain access to all of his/her own personal account information.
- The application shall not allow users to obtain access to any account information of any other user.

Identification, authentication and authorization requirements together are called access control requirements.

Immunity
This is security requirements that specifies the extent to which an application or components shall
protect itself from infection by unauthorised and undesirable programs such as viruses, worms and Trojan horses. This is a form of software integrity and the typical objectives are to prevent any undesirable programs from destroying or damaging data and applications.

Examples of immunity requirements are:

- The application shall protect itself from infection by scanning all entered or downloaded data and software for known computer viruses, worms, Trojan horses and other similar harmful programs.
- The application shall disinfect any file found to contain a harmful program if disinfection is possible.
- The application shall notify the security administrator and the associated user if it detects a harmful program.

**Integrity**

This is a security requirement that specifies the extent to which an application or component shall ensure that its data and communication are not intentionally corrupted via unauthorised modification or deletion. The objectives are typically to ensure that communications and data can be trusted.

Examples of integrity requirements are:

- The application shall prevent the unauthorised corruption of data collected from users.
- The application shall prevent the unauthorised corruption of all communications passing through networks that are external to any protected data center.

**Intrusion detection**

This is any security requirement that specifies the extent to which an application or component shall detect and record attempted access or modification by unauthorised users. The objective of this kind of requirement is to detect unauthorised users and programs that are attempting to access the application or component, to record information about the unauthorised access attempts, and to notify security personnel so that they can handle the intrusions in a proper way. These kinds of requirements depend on access control requirements.

Examples of intrusion detection requirements are:

- The application shall detect and record all attempted accesses that fail identification, authentication or authorization requirements.
- The application shall daily notify the data center security office of all failed attempted accesses during the previous 24 hours.

**Nonrepudiation**

This kind of security requirements specify the extent to which a business, application, or component shall prevent a party to one of its interactions from denying having participated in all or part of the interaction. Nonrepudiation requirements deal with ensuring that adequate, temper-proof records are kept.

Nonrepudiation requirement typically involve the storage of a significant amount of information about each interaction including the authenticated identity of all parties involved in the transaction, data and time that the interaction was sent, received and acknowledged and significant information that is passed during the interaction.

**Privacy**

This kind of security requirements specifies the extent to which a business, application, component
or center shall keep its sensitive data and communications private from unauthorised users and programs. The typical requirements of privacy requirements are to ensure that unauthorised users and programs do not gain access to sensitive data and communications, to provide access to data and communications on a need to know basis, and to minimise potential loss or user confidence and legal liabilities. There are three major kinds of privacy, anonymity, communication privacy and data storage privacy. Privacy requirement should clearly identify their scope and they must be considered with audibility, identification and nonrepudiation requirements. They require users to be identified and information about their interactions to be stored.

Examples of anonymity, communication privacy and data storage privacy are:

- Anonymity – The application shall not store any personal information about the users.
- Communication privacy – The application shall not allow unauthorised individuals or programs access to any communications.
- Data storage privacy – The application shall not allow unauthorised individuals or programs access to any stored data.

Auditing

These kind of requirements specify the extent to which a business, application, component or center shall enable security personnel to audit the status and use of its security mechanisms. The objectives are to ensure that the application or component collects, analyses and reports information about the status of its security mechanisms and about the use of security mechanisms like access and modifications by security personnel. These requirements can duplicate with the intrusion detection requirements.

Examples of audit requirements are:

- The application shall collect, organise, summarise and regularly report the status or its security mechanisms including identification, authentication and authorization, immunity, privacy and intrusion detection.

Survivability

These kind of security requirements specifies the extent to which an application or center shall survive the intentional loss or destruction of a component. Objectives are to ensure that an application or center either fails gracefully or else continues to function, perhaps in a degraded mode, even though some components are damaged.

Examples of survivability requirements are:

- The application shall not have a single point of failure.
- The application shall continue to function even if a component is destroyed.

Physical protection

These kind of security requirements specify the extent to which an application or center shall protect itself from physical assault. Objectives are to ensure that an application or center are protected against the physical damage, destruction, theft or replacement of hardware, software or personnel components due to vandalism, sabotage or terror. Physical protection requirements are related to survivability requirement.

Examples of physical protection requirements are:

- The data center shall protect its hardware components from physical damage, destruction, theft or surreptitious replacement.
- The data center shall protect its personnel from death, injury and kidnapping.
Maintenance
These kind of security requirements specify the extent to which an application, component or center shall prevent authorised modifications from accidentally defeating its security mechanisms. The typical objective is to maintain the levels of security specified in the security requirements during the usage phase.

Examples of maintenance security requirements are:

- The application shall not violate its security requirements as a result of the upgrading of a data, hardware or software component.
- The application shall not violate its security requirements as a result of the replacement or a data, hardware or software component.

4.4.8 Guiding principles for software security
In this section, ten principles for building secure software are presented. The goal of the list is to highlight the most important objectives you should keep in mind when designing and building a secure system. It is important to note that this list does not cover every possible flaw and no list of this kind is ever perfect but it is a very good start. These guiding principles are widely accepted and commonly applied and are considered important to establishing security within a system, these ten are some of the more important ones. They are each described below [2] [25] [29] [39].

1. Secure the weakest link.
2. Practice defence in depth.
3. Fail securely.
4. Follow the principle of least privilege.
5. Compartmentalise.
6. Keep it simple.
7. Promote privacy.
8. Remember that hiding secrets is hard.
9. Be reluctant to trust.
10. Use your community resources.

How some security patterns relate to these guiding principles is presented in table 2. Some of the patterns in the table are presented in section 4.5 and the one not presented in this thesis report can be read about in the references listed in the same section.

Table 2: How some security patterns relate to the guiding principles for security. After [39]
Secure the weakest link.

A chain is always as strong as its weakest link. It is just the same with security, the security of a system is only as strong as its weakest element. Therefore, an important step in understanding how secure your system really is, is to understand what the weakest security link is. The weakest link could be technological, an unsecured network link for example, procedural, allowing easy access to a data center, or human, for example people who write down their login information. Attackers will attack the weakest link in the system because that is the element that is easiest to break.

It's not always the software that holds the weakest link, it could also be the surrounding infrastructure. For example social engineering, an attack in which the attacker uses social manipulation in some way to break into a system.

Identifying and securing the weakest link in the system could be an infinite loop since a system can never cover all different kind of security features. There needs to be a stopping point somewhere. The loop can stop for example when the system reaches an acceptable level of security risk.

Practice defence in depth.

Medieval castles had moats, drawbridges and strong walls. Banks have alarms, vaults, security guards, surveillance systems and multiple locks in important doors. What do the medieval castles and banks have in common? They both practice defence in depth since they do not rely on just a single security measure but instead have several different ones. So why should our computer systems have only one kind of security? A series of defences provide a greater level of security than a single technique could. Defence in depth is extra important in computer security since the security techniques used may themselves have hidden flaws, and they are also sensible to human and procedural failures. If one layer of defence turns out to be inadequate, another layer of defence hopefully prevents a full breach. Defence in depth is especially powerful when each layer works together with the others.

Fail securely.

That a system fails should be planned for, because it is unavoidable. When many systems fail they do so in an unsecured way. In these kinds of systems the attacker can just cause the right kind of failure or wait for it to happen. Then they can exploit the system as they wish. An example of when not failing secure is an audit trail that suspend auditing if the audit logs run out of space. System failures occur occasionally so the system should be able to fail in a secure way.

Follow the principle of least privilege.

The system should always grant users the smallest set of privileges they require to still be able to perform their tasks, the privileges can also be changed over time. This means that only the minimum access necessary to perform an operation should be granted, and that access should be granted only for the minimum amount of time necessary.

Compartmentalise.

The basic idea behind compartmentalisation is to minimise the amount of damage that can be done to a system by breaking up the system into as few units as possible while still isolating code that has security privileges. This principle must be used in moderation. If each little bit of functionality is separated, then the system will become impossible to maintain.

Keep it simple.

Security professionals often say that complexity is the enemy of security. Complexity increase the risk of problems. Complexity in a system is difficult to deal with and makes it very difficult to analyse the system to assess its security, which makes problems more difficult to understand and rectify when they are found. Systems with stringent security requirements need to be simple enough to make it possible for them to be secured and verified. The most obvious implication is that
software design and implementation should be as straightforward as possible.

**Promote privacy.**

Many users consider privacy a security concern. One of the things that privacy most often trades off against is usability. For example, most systems are better off forgetting about credit card numbers as soon as they are used. That way, even if the web server is compromised there is not anything interesting stored there long term. Users hate that from a usability perspective since they have to type in their credit card information each time they want to use it online.

User privacy is not the only kind of privacy. Malicious hackers tend to launch attacks based on information easily collected from a target system. Services running on a target machine tend to give out big amounts of information about themselves that can help the attacker figure out how to break in. There is no reason to give out any more information than necessary.

**Remember that hiding secrets is hard.**

Security is often about keeping secrets. Users don't want their personal data leaked and keys must be kept secret to avoid eavesdropping and tampering for example. These kind of security requirements are often high on the requirements list but they turn out to be far more difficult to meet than the average user may suspect. There are many different kind of attacks that needs to be considered and the attacker must be thought of as very intelligent. The potential for serious misuse and abuse in software is a very real risk.

**Be reluctant to trust.**

People commonly hide secrets in client code, assuming those secrets will be safe. The problem with putting secrets in client code is that talented end users will be able to abuse the client and steal its secrets. Instead of making assumptions that need to hold, you should be reluctant to extend trust. Servers should be designed not to trust clients, and vice versa, because both clients and servers get hacked. A reluctance to trust can help with compartmentalisation. Sometimes it is prudent not to trust even yourself. It is all too easy to be shortsighted when it comes to your own ideas and your own code.

**Use your community resources.**

If a technique have been used several times without failure it is a good idea to use that technique in some way in your own solution, even though it is not a good idea to follow everyone else blindly. This principle only applies if you have reason to believe that the community is doing its part to promote the security of the components you want to use.

**4.4.9 Human issues**

The heart of any security system is people. This is very true in computer security since it deals mainly with technological controls that can usually be bypassed by human interventions. For example, when a system authenticates a user it might ask for a password of some sort. If the correct password is inserted the access is granted. But even if the user told (in some way) someone else his or her password then the second person can masquerade as the user and enter the password, the access will still be granted.

People who have some motive to attack an organisation and are not authorised to use the organisations systems are called outsiders and they are a serious threat. But a far more dangerous threat comes from disgruntled employees and other outsiders who are authorised to use the computers. Experts agree on this. This is because insiders typically know the organisation of the company's systems and what procedures the operators and users follow and often know enough passwords to bypass many security controls that otherwise would detect an attack made by an outsider. The problems of insiders that misuse authorised privileges are very hard to solve.
Untrained personnel also pose a threat to the security since they might not know all the security procedures that the personnel need to do. Lack of training need not be in the technical arena, but many successful break-ins have arisen from the art of social engineering. The social engineering attacks are remarkably successful and often devastating.

These kinds of human issues that are threats to the security of a system is very hard to defend against and are not something that can be solved with technical solutions. It is therefore not the focus of this report [23] [37].

4.5 Security Patterns

As described in section 4.2, patterns describe recurring solutions to common problems in a given context and system of forces. There exist a number of security patterns that helps solving different security issues.

A security pattern describes a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution. - [37]

Security patterns are proposed as means of bridging the gap between developers and security experts [44]. This section of the report presents very shortly some of the many security patterns that exist. Further information about the presented patterns and information about other patterns can be seen in [22] [38] [39] [40] [41] [42] [43] [44] [45]. Presented in the patterns below (section 4.5.1-4.5.5) is the pattern context, the problem, the forces, the solution, consequences and related patterns where relevant. The patterns presented here are the Authenticator pattern, the authorization pattern, the Role-Based Access Control pattern, the Single Access Point pattern and the Multilevel Security pattern.

Access control is an important part of security in software and there are a number of patterns for different aspects of access control available. How some of them relate together can be seen in figure 19 [42], and also in figure 22. This gives an overview of how different patterns can be used together to achieve a specific purpose, it also shows that just one pattern cannot do the entire job itself, a pattern solves just a small specific problem or even part of a problem and need to be used together with other patterns. Not all of the patterns used in these figures are presented in this thesis report, the ones that are not chosen to be presented here can be read in the references listed above.

![Figure 18: How patterns relate to accomplish access control. After [9]](image-url)
4.5.1 Authenticator pattern

Context
Systems authenticate users when they first log in, and maybe again when they access specific resources. The system controls the creation of a session in response to the request by a subject (typically a user). The authenticated user is then allowed to access resources according to their rights. Sensitive resource access may require additional process authentication.

Problem
A malicious attacker could try to impersonate a legitimate user to gain access to his or her resources. This could be particularly serious if the impersonated user has a high level of privilege. How can we prevent impostors from accessing our system?

Forces
- There is a variety of users that may require different ways to authenticate them. We need to be able to handle all this variety, or we risk security exposures.
- We need to authenticate users in a reliable way. This means a robust protocol and a way to protect the results of authentication. Otherwise, users may skip authentication or illegally modify its results, exposing the system to security violations.
- There are trade-offs between security and cost – more secure system are usually more expensive.
- If authentication needs to be performed frequently, performance may become an issue.

Solution
Use a Single Access Point (section 4.5.4) to receive the interactions of a Subject, typically a user, request access to system resources. The Authenticator receives this request and applies a protocol using some Authentication information. If the authentication is successful, the Authenticator creates a Proof of Identity, which can be explicit, for example a token, or implicit. The relationships between the classes are shown in figure 20.

Consequences
The advantages of this pattern include:
- Depending on the protocol and the authentication information used, we can handle any types of users and we can authenticate them in diverse ways.
- Since the authentication information is separated, we can store it in a protected area to which

Figure 19: Authenticator pattern. After [9]
all subjects may have read-only access at most.

- We can use a variety of algorithms and protocols of different strength for authentication. The selection depends on the security and cost trade-off.
- Authentication can be performed in centralised or distributed environments.

Possible disadvantages include:
- The authentication process takes some time.
- The general complexity and cost of the system increases with the level of security.

**Related patterns**
Single-Sign-On pattern is a variant of the authenticator pattern.

### 4.5.2 Authorization pattern

**Context**
Any computational environment where there are active entities that request resources whose access must be controlled.

**Problem**
How to describe allowable types of accesses (authorizations) by active computational entities (subjects) to passive resources (protection objects).

**Forces**
- The authorization structure must be independent of the type of resources, e.g. it should describe access by users to conceptual entities, access by programs to operating system resources, etc., in a uniform way.
- Predicates or guards may restrict the use of the authorization according to specific conditions.
- Some of the authorizations may be delegated by their holders to other subjects.

**Solution**
Represent the elements of an authorization rule as classes and associations. Class Subject describes the active entities, while class ProtectionObject describes the requested resources. An authorization rule is defined by an association between these two classes. An association class, Right, includes the type of access allowed (read, write, etc.), a predicate that must be true for the authorization to hold, and a copy flag that can be true or false indicating if the right can be transferred or not. An operation check_rights can be used in the subject or object to check the validity of a request. Figure 21 shows these elements together.

*Figure 20: authorization pattern. After [44]*
Consequences
The advantages of this pattern include:

- The pattern applies to any type of resource. Subjects can be executing processes, users, roles, user groups. Protection objects can be transactions, memory areas, I/O devices, files, or other OS resources. Access types can be read, write, execute, or methods in high-level objects.
- The predicates in the rules are a general representation of any conditions that may restrict the application of a rule.
- The copy flag in the rule controls transfer of rights. However, some systems do not allow transfer of rights, i.e., their copy flags are always false.
- Some systems separate administrative authorizations from user authorizations for further security.
- The request may not need to specify the exact object in the rule, this object may be implied by an existing protected object. Subjects and access types may also be implied. This improves flexibility at the cost of some extra processing time.

Related patterns
The role-based access control pattern (section 4.5.3) is a specialisation of this pattern.

Figure 22 shown how different security patterns for authorization can work together to manage the authorization part of access control.

4.5.3 Role-based access control pattern (RBAC)
Context
Most institutions have a variety of job functions that require different skills and responsibilities. For security reasons users should get rights based on their job functions. This corresponds to the application of the need-to-know principle, a fundamental security policy. Job functions can be interpreted as roles that people play in performing their duties. In particular, web-based systems have a variety of users: company employees, customers, partners, search engines, etc.

Problem
How to assign rights to users according to their roles in an institution.
Forces

- People in institutions have different needs for access to information, according to their functions.
- We want to help the institution to define precise access rights for its members according to a need-to-know policy.
- Granting rights to individual users would require storing many authorization rules and it would also be hard for administrators to keep track of these rules.
- Users may have more than one role and we may want to enforce policies such as separation of duty, where a user cannot be in two specific roles in the same session.
- We may need to have hierarchies of roles, with inheritance of rights.
- A role may be assigned to individual users or to groups of users.

Solution

Extend the idea of the authorization pattern interpreting roles as subjects. A basic model of Role-Based Access Control (RBAC) is shown in figure 23. Classes User and Role describe the registered users and the predefined roles, respectively. Users are assigned to roles, roles are given rights according to their functions. The association class Right defines the access types that a user within a role is authorised to apply to the protection object. In fact, the combination Role, ProtectionObject, and Right is an instance of the authorization pattern. Accordingly, the predicate indicates content dependent restrictions that can be used to select specific objects.

![Figure 22: Basic RBAC Pattern. After [44]](image)

Consequences

The advantages of this pattern include:

- It allows administrators to reduce the complexity of security, there are much more users than roles.
- Institution policies about job functions can be reflected directly in the definition of roles and the assignment of user to roles.
- Roles can be structured for further flexibility and reduction of rules.
- Users can activate more than one session at a time for functional flexibility.
- We can add UML constraints to indicate that some roles cannot be used in the same session or given to the same user.
- Groups of users can be used as role members, thus further reducing the number of authorization rules and the number of role assignment.
Possible disadvantages include:

- Additional conceptual complexity (new concept or roles, assignments to multiple roles etc.).

**Related patterns**

A simpler version is the authorization pattern. Other related patterns are the Role pattern and the Abstract Session pattern. It is also possible to specialise this pattern more, one way is to include the authorization pattern and the Composite pattern in the RBAC pattern.

### 4.5.4 Single access point pattern

**Context**

Access needs to be provided to a system for external clients. It needs to be ensured that the system is not misused or damaged by such clients.

**Problem**

Whenever a system is used by an external client such as a user, the system's integrity is in danger. Often such systems require some security property, like protection from misuse or damage. One means is to check every interaction with an external client to determine whether it is authorised. When the system has a non-trivial inner structure and consists of multiple parts or subsystems, an external interaction of the system can result in many different interactions of the client with the individual parts of the system. Checking each of these sub-interactions is required to protect all the parts, and thus the whole system. First, implementing all these checks can be a burden: second, if the same information has to be presented over and over, these checks can hinder performance and annoy a user: third, assessing the correct implementation of the overall security policy is hard, because of its complexity.

**Forces**

- You need to provide access to a system to make it usable.
- In a complex interconnected world, no system is an island.
- Most systems exhibit a non-trivial structure and are constructed from sub-systems that also need protection.
- Many entry points to a system reduce security, because the additional complexity makes it easier to bypass controls.
- Multiple entry points can have duplicate code for the same kind of checking.
- Repeated checks annoy clients or slow down the system.
- Uniform access to a system can lower its usability if different situations really require different means of access.
- Uniform access to a system is easier to control.

**Solution**

Define a single access point for clients using the system. At this access point you can check the legitimacy of the client according to your defined policy. Once clients passed the access point, they are free to use the system from that point on. Protect the rest of the system's boundary, so that no circumvention of the single access point is possible. Make the single access point prominent, so that it is easy to find and absolutely obvious where to enter the system. If auditing is required, the single access point can record which clients entered the system and when. It might also record the termination of a client's use of the system. This pattern applies to many levels of abstraction and technology. It further might apply within a more complex system to the system itself as its subsystems, which in turn can have additional single access points. The single access point pattern applies.
can be represented with the UML diagram in figure 24.

![Figure 23: Single Access Point Pattern. After [9]](image)

**Consequences**

The advantages of this pattern include:

- It provides a single place to go for entering the system, a clearly defined entrance for users of the system, and a single place to set up the system or application properly.
- It provides a single place to guard your system: you only need to trust your gate guards at the single access point within your system. However, applying defence in depth might be required to improve security further.
- The inner structure of system is simpler, because repeated authorization checks are avoided. The system trusts the single access point.
- No redundant authorization checks are required: once the access point is passed, the system trusts the client.
- It applies to many levels of abstraction.

Possible disadvantages include:

- Having just a single access point may make the system cumbersome to use, or even completely unusable.
- You need to trust the guard of the single access point.
- The single access point need to check the client on entrance more thoroughly than is required in the concrete situation, thereby annoying the client or slowing down entrance unacceptable.
- In a complex system, several single access points might be required for subsystems.
- The single access point might become a single point of failure. If the single access point breaks down, the system might become unusable, or its security compromised.

**4.5.5 Multilevel Security Pattern**

**Context**

In some environments data and documents may have critical value and their disclosure could bring serious problems.

**Problem**

How can you control access in an environment with sensitive documents so as to prevent leakage of information?
Forces

- We need to protect the confidentiality and integrity of data based on its sensitivity.
- Users have to be allowed to read documents based on their rank or position in the organisation.
- There should be a way to increase or decrease the ability of users to read documents and the sensitivity of the documents. Otherwise, people promoted to higher positions for example could not read sensitive documents, and we would end up with a proliferation of sensitive and obsolete documents.

Solution

Assign classifications as clearances to users and classifications as sensitivity levels to data. Separate different organisational units into categories. Classifications could for example include levels such as top secret, secret and so on. Figure 25 shows the basic structure of this pattern. The User Classification and Classification classes define the active entities and the objects of access, respectively. Both classifications may include categories and levels. Trusted Processes are allowed to assign users and data to classifications, as defined by the Assignment() class.

Consequences

The advantages of this pattern include:

- The classification of users and data is relatively simple and can follow organisation policies.
- This model can be proven to be secure under certain assumption.
- The pattern is useful to isolate processes and execution domains.

Possible disadvantages include:

- Implementations should use labels in data to indicate their classification. This assures security. If this is not done the general degree of security is reduced.
- We need trusted programs to assign users and data to classifications.
- Data should be able to be structured into hierarchical sensitivity levels and users should be able to be structured into clearances. This is usually hard, or even impossible, in commercial environments.
- Covert channels may break the assumed security.
Chapter 5 – Results

In this chapter of the report the results of the thesis work is presented. The implemented GUI framework for the ArchWiz tool is presented in section 5.1. The security formalisation suggestions are presented and discussed in section 5.2.

5.1 ArchWiz GUI

The time for the ArchWiz project was too short and too few resources was working on the project to be able to finish it completely. The tool is about halfway finished at the point when the project was closed in December 2009. What is finished is a good foundation for future implementations on the tool if the ABB management decides to carry the ArchWiz through. As mentioned earlier the people working on the project was I (Caroline Uppsåll), another thesis student named Zafar Bahati, Anton Jansen who works at ABB CRC and our project manager and supervisor Pia Stoll. The GUI implementation and also some design was decided to be my assignment for the implementation part of the project and therefore that is the part of the ArchWiz tool that is presented in this report.

Tests of the GUI framework have been done in a tool called CogTool developed at Carnegie Mellon University, Pittsburg, USA and it is a general purpose UI prototyping tool with a difference, it automatically evaluates your design with a predictive human performance model. Using the CogTool you can calculate how long it takes for an average user to click through the design. This gives an estimate of how much more efficient the ArchWiz tool design is compared to the USAP prototype tool described in chapter 4.3. The tests done in CogTool as well as the result are presented in section 5.1.1.

To decide a default size, some layout issues and text types in the ArchWiz tool a guide for how to design webpages at ABB have been used where appropriate.

In figure 26 is a screenshot of the welcome screen in the tool. This is the first view the user is presented with when opening ArchWiz. Here the menu bar is visible (presented in figure 29). The user can open an existing project through “File” in the menu bar, by clicking the “Open Project” button or by selecting a recent project in the “Recent Projects” area to the right. The user can also create a new project, either through “File” in the menu bar or by clicking the “Create New Project” button.
In figure 27 is the pop-up dialogue window that is displayed when the user chooses to create a new project. Here the user enters information about the project, author name, project name and where to store the new project.

In figure 28 is a screenshot of how it looks like when the user chooses to open an existing project by clicking the “Open Project” button or by the ”File” option in the menu bar. The user navigates to the preferred project file in the file explorer and opens it. The selected project file will be opened and the user can continue working on his/her project.
Figure 29 displays the menu bar with its different menu items. The menu bar consists of “File”, “Project”, “View” and a question mark symbol. Under “File” the user can create a new project (figure 27), open an existing project (figure 28), save the project, reload the project, print a todo-list and view the project’s properties. The todo-list is explained with figure 36. The menu items under “Project” is documentation, version control if several users are working on the same project version control is needed as well as import and export options so that users can import and export different versions of the tool. Under “View” the user can split the screen into two, if not already splitted, and if the view is splitted already he or she can go back to a non-splitted mode by clicking “Undo Split”. The question mark symbol is the help symbol. If the user clicks it a pop-up dialogue window will be displayed with information about how to use the ArchWiz tool.

Figure 27: Open project window.

The tool is based on a tree structure with all general requirements. Figure 30 shows the tree structure placed on the left hand side of the tool. In the default version is the three qualities (usability, security and safety) the node-level directly under general requirements. Each of the qualities holds its own formalisation of requirements. Usability holds the USAP model but without the scenarios since the tool is suppose to be requirement specific. Security holds the preferred suggestion of security formalisation and Safety will hold the formalisation Zafar Bahati comes up with since that is his quality to formalise.
There are two different kinds of nodes in the tree, text nodes that only hold descriptive text and decision nodes where the user can take a decision. Text nodes are described in figure 31 and decision nodes are described in figure 34. The user can add a new node of preferred kind by right clicking the parent node of the new node and selecting the node type in the pop-up menu. In front of each node is a check box, by using these the user can select and deselect requirements according to the project they are working on. In front of the nodes there may also be a symbol (green check mark, red cross or black question mark) depending on which decision have been made, these are described further in figure 34. How the selection of requirements work is described in figure 32 and 33. If the tree grows outside of the tree area scroll bars will be visible.

This is one of the two views that can be displayed when a project is running depending on the node type. Figure 31 show the view for a text node. A text node is for example Authoring (Usability) as shown in the figure. A text node can display for example a descriptive text about the node, a purpose and justification as for Authoring in the figure or some other text about the node.

It has the menu bar at the top and the tree at the left hand side as described earlier. At the right area of the screen is a text area that holds the text. Here the user can edit, by just placing the marker in the text area, if he/she wishes. If the text gets longer than the text box can display the scroll bar can
be used.

Under the tree is a button “Only show product specific requirements”, it is described in figure 32. Under the text area is two button “Previous” and “Next” that can be used to navigate through the requirements, in addition to the tree, and a progress bar that shows the progress of requirements the user have taken decisions on, relative to the total amount of requirements selected for the project.

The tool have a default size of 1024x768 px, if the window gets bigger all the different fields in it will grow relative to each other, this means that the layout will still be the same even if the window is larger. If the window gets smaller though scroll bars both on the bottom and to the right will appear, a minimum size is decided since the text, trees and buttons will be too small otherwise.

![Figure 30: The tool on a text node.](image-url)

Purpose:
The purpose of these responsibilities is to allow specification of the behavior of the system in certain ways under certain circumstances.

Justification:
Users want to control the behavior of the system in certain ways under certain circumstances without having to set it up every time. The system needs a specification of parameters to determine its behavior in these circumstances. Therefore the user must author a specification of parameters that will subsequently be used upon execution (see the responsibilities under Execution with Authored Parameters)
As mentioned earlier in this section the user can select and deselect requirements depending on the project he or she is working on. This is done with the help of the check boxes in front of the tree nodes in the tree as shown in figure 32. All the requirements are selected by default, the user can deselect requirements that do not suit the project by de-checking the check box in front of it. When only the wanted requirements' check boxes are selected the user needs to press the “Only show product specific requirements” button. This action will update the project's knowledge base. An updated knowledge base depending on the selections in figure 32 are shown in figure 33, the tree shows that the project have been updated.

Figure 31: Select/Deselect nodes in the tree.

Figure 32: Only selected nodes are shown in the tree.
Figure 34 shows a screenshot of the second view of the tool. The view that is displayed when a decision node is selected in the tree or by the “Previous” and “Next” buttons. This view shows, just as the other view for a text node, the menu bar at the top, the tree at the left hand side, the “Only show project specific requirements” button, the “Previous” and “Next” buttons as well as the progress bar. What is specific for this view is that it has a text area that describes the requirement and some implementation details, rationales, notes etc. It has an area for making decisions under the text box with the different decisions that can be made for the requirement (Not yet considered, Must modify architecture, Architecture already addresses this and Not applicable), there is also a check box “Discuss status of requirement” that the user can check if he or she feels the need to discuss the requirement with someone else or come back to it later. There is also a comment box beside the decision buttons where the user can write his/her own comments about the requirement. If the user wants to edit the node he or she can just simply place the marker in the text box and edit the text in it. If the text gets longer than the text area a scroll bar can be used to view all the text.

Depending on what decision has been made a symbol is displayed in front of the node in the tree:

- Not yet considered – default.
- Must modify architecture – displays the black question mark in the tree
- Architecture already addresses this – displays the green check mark in the tree.
- Not applicable – displays the red cross in the tree.

Is is so that the user can have an overview of the kinds of decisions he/she have made on the different requirements.

Just as the text node view this view can also be made bigger and still remain its layout relatively and displays scroll bars if it gets smaller than the default size.
One of the options in the menu bar at the top was to split the view in two. How that looks is displayed in figure 34. This makes it possible for the user to display two different nodes at the same time. It is for example useful if the user wants to make a decision on one node and at the same time look through the history of decisions he or she have made. It is possible to drag the divider to make one side of the screen larger or smaller than the other.

The user can, by selecting the print option under “File” produce a todo-list that contains the requirements where the decision is “Must modify architecture” or “Discuss status of requirement”. It also contains the comments made on the requirements. This will be a simple HTML file that opens in the web browser. A todo-list design for the ArchWiz tool have not been made but it will probably look similar to the one generated in the USAP prototype web tool. A screenshot of the todo-list from the USAP prototype web tool is shows in figure 36.

This is the GUI framework as it looks at the end of the project. Some changes still need to be done to it if the development of the tool continues. Those are described in section 6.2, Future work.
Many iterations of designs have been made before this version was implemented. Both GUI mock-ups on paper and real frameworks made in Java/Jigloo in Eclipse have been made. Since the focus of the tool has been slightly changed throughout the project some features belonging to the first versions of the tool have been removed which have been quite time consuming. The final version of the framework is still a good GUI for further development to build on.

5.1.1 Result of the CogTool test:

The main test was:

1. Show the Welcome screen (figure 26).
2. The user presses the create new project button (figure 26).
3. A pop-up window is displayed (figure 27).
4. The user enters author name (figure 27).
5. The user enters project name (figure 27).
6. The user selects the directory (figure 27).
7. The user clicks the save button (figure 27).
8. The tool is started with default content.
9. The user deselects the requirements that are not product specific (figure 32).
10. The user presses the “only show product specific requirements button”.
11. The tree will display only the selected requirements (figure 33).
12. The user selects File -> Print (figure 29).
13. The user is presented with a todo-list (figure 36).

This sequence took 36,671 seconds in the CogTool. The time it takes to change view was estimated
and the time does only encounter the time it takes to click this sequence, it does not take into consideration how long it takes for the user to read the text and make a decision/write comments. It also has a standard time of how long it takes a user to find and click a button for example. To know this a test with real users need to be done, this is unfortunately not possible at the point of the projects closure since the tool is not complete to such an extent to have a proper test.

There were also some smaller tests made in CogTool:

- How long it takes to click a decision (1,052sec)
- How long it takes to check the discuss status button (1,243sec)
- How long it takes to write a longer comment (41,484sec)
- How long it takes to click the next button and navigate to the next requirement (2,457sec)

The test made with real users was concerning 40 requirements. To be able to make a comparison between the USAP prototype web tool and the ArchWiz tool the same amount of requirements are used in this CogTool test.

Just clicking through 40 requirements and take decision on all of them, checking the “discuss status” check box (on half of the requirements we take decisions on) and writing a comment (also on half of the requirements we take decisions on) took 1038,851sec this is 17,214min.

The tests done in the USAP prototype tool with Force Measurements resulted in a time around 6 hours (360min) to go through all the 40 requirements and take decisions on each requirement for each scenario.

Since 17,214min is just clicking through the ArchWiz tool, we calculate how long a test person may take on reading and thinking by using the results in the test for the USAP prototype tool:

We take the 360min-17,214min = 342,786min

Because we don’t have scenarios any more in the ArchWiz tool we take 342,786min/3scenarios = 114,262min

Then we can subtract 30min for the time it took to understand the USAP model 114,262min-30min = 84,262. This is how long it took the test persons to read the instructions and decide on a decision for every requirement.

If we apply the same 84,262 min of thinking and reading to the new tool the total time will be:

17,214min+84,262min = 101,476min

The time it takes to make decisions on 40 requirements (and make comment and check the “discuss status” check box on half of them) in the ArchWiz tool is by using the CogTool and some estimations 101,476 minutes, compared to the 360 minutes it took for the USAP prototype web tool and the same amount of requirements.

### 5.2 Security formalisation suggestions

Help with the implementation of the ArchWiz tool was one of the tasks for the thesis job, the other task was to suggest a way of how to formalise security requirements in a general way. After reading a variety of books, articles and papers regarding the field of security different suggestions of formalisation was created. The suggestions were sent to the project owner for feedback and updated with the feedback in mind. The suggestions are presented in section 5.2.1-5.2.4 and analysed in section 5.2.5. Today ABB have a design handbook where users can add and edit requirements of different qualities. The content of some of the examples of the different suggestion below come from that handbook.
Before the different suggestions were put together a diagram of how the basic different concepts relate, most of them described in section 4.4.1, were drawn to get an overview of the security field. The diagram is presented in figure 37.

The existing patterns used to solve different aspects of security problems have been useful in the development of these suggestions and should be used when making implementation details for the refined requirements, where appropriate of course. The security patterns are well known and accepted, functioning solutions to common security problems and should therefore be used to achieve secure software. To be able to map which security patterns are appropriate using for which refined requirements table 3 shows a comparative model of how some security patterns relate to some of the security objectives. The table presented below are most appropriate for security formalisation suggestion 1 since it is concerned with the security objectives.

When it comes to threats, for spoofing attacks the Authenticator pattern can be used. For tampering attacks the Check Point pattern and Single Access Point pattern can be used. Information disclosure threats can be protected from with the help of Authorization patterns. These are examples and there are many other patterns that can be used for different threats.

![Diagram](image-url)
5.2.1 Security formalisation suggestion 1

Suggestion 1 of a security formalisation is the one most alike the design handbook. This formalisation is based on security objectives and mechanisms. The highest level of requirements is the objectives such as authentication, authorization, confidentiality, integrity etc., these are explained in section 4.4.7. Each security objective can have a number of mechanisms that helps with implementing the objective and each mechanism in turn holds a number of refined requirements that should cover all aspects of the mechanism that needs to be considered. The architect will have to decide which objective is appropriate for the project and which mechanism/s that are to be suitable. Then the architect will take a decision (not yet considered, must modify architecture, architect already addresses this and not applicable) on each of the refined requirements. A diagram of this suggestion is shown in figure 38.
Patterns can be used to describe the mechanisms that are appropriate for the goal and they can be broken down to smaller parts to describe a general way of implementing the refined requirement. Each node of this hierarchy will need an explanatory text and the refined requirement nodes might also have rationale, implementation details, notes etc. where they are needed. The user should also be able to make comments on each refined requirement.

Below is an example of how the hierarchy can look like:

**Example:**

- **AU-Authentication (Security objective)**
  
  *Description:* Authentication requirements specify the extent to which an application or component shall verify the identity of its externals (e.g. Human actors and external applications, components) before interacting with them. The typical objectives of an authentication requirement are to ensure that externals are actually who or what they claim to be and thereby to avoid compromising security to an imposter.

  - **AU1 – Password Based (Mechanism)**
    - **Refined Requirements**
      - A1.1 There should be a field to enter user id, if not otherwise known in the system.
      - A1.2 There should be a field to enter password.
      - A1.3 Password entry should be masked to avoid that the entered password can be observed.
      - A1.4 There should be an easily accessible user interface to close the authentication session (log out).
      - A1.5 There should be a display of log-in status during session.
    
    etc.

  - **AU2 – Certificate Based (Mechanism)**
    - **Refined Requirements**
      - AU2.1 Refined requirement for certificate based authentication.
      - AU2.2 Refined requirement for certificate based authentication.

  etc.

- **AUTH – Authorization (Security objective)**
  
  *Description:* Authorization requirements specify the access and usage privileges of authenticated users and client applications.
AUTH1 – Role Based (Mechanism)

AUTH2 – Refined Requirements

AUTH1.1 Users should only be able to edit data they have permission to edit.

AUTH2.2 Only administrators should be able to change users privileges within the system etc.

A modification to this suggestion can be to add another level of abstraction between the security objective and the mechanisms that holds goals, a diagram of this can be seen in figure 39.

![Diagram](image)

Figure 38: Modified version of suggestion 1.

An example of this is that confidentiality is the security objective and that two possible goals of confidentiality are Goal1: Ensure confidentiality of data stored in database and Goal2: Ensure confidentiality of data communicated oven LAN. These two goals of confidentiality will require different mechanisms.

5.2.2 Security formalisation suggestion 2

This suggestion is based on different threat types. Threats are a natural way of thinking of security and as described in section 4.4.6, security requirements are driven by security threats. Most requirements are stated in terms of what must happen but security requirements are often specified in terms of what must not be allowed to happen. It is in this context natural to work with threats and countermeasures and those are therefore the basis of this suggestion. There exist a number of different kinds of threats to security, and those threats have countermeasures. The refined requirements are in this suggestion on the level of countermeasures. Each countermeasure consists of a number of refined requirements that needs to be considered to help designing the countermeasure. The architects make a selection of which threats the system is suppose handle and then which countermeasure to use. The architects then takes decisions for (not yet considered, must
modify architecture, architect already addresses this and not applicable) each refined requirement. A diagram of this security formalisation suggestion is presented in figure 40. The nodes needs to have explanatory text and the refined requirements should have for example rationale, implementation details and notes as appropriate. The user should also be able to make comments on each refined requirement.

Below is an example of how this hierarchy can look like:

**Example:**

- **SUI – Spoofing user identity (Threat type)**
  
  **Description:** Spoofing is attempting to gain access to a system by using a false identity. This can be accomplished using stolen user credentials or a false IP address. After the attacker successfully gains access to a legitimate user or host, elevation of privileges or abuse using authorization can begin.

- **SUI1 – Authentication (Countermeasure)**
  
  **Description:** Authentication requirements specifies the extent to which an application or component shall verify the identity of its externals before interacting with them. The typical objectives of authentication is to ensure that externals are actually who or what they claim to be and thereby avoid compromising security to an impostor.

  - **Refined Requirements**
    
    - **SUI1.1** There must be a way to enter a user identity, if not otherwise known in the system.
    - **SUI1.2** If an authentication method involving password is chosen, there must be a way to enter a password.
    - **SUI1.3** If an authentication method involving password is chosen, the password entry should be masked to avoid that the entered password can be observed.
    - **SUI1.4** There should be an easily accessible user interface to close the authentication session (log out) etc.

- **SUI2 - Storage (Countermeasure)**
  
  **Description:** Information regarding authentication needs to be stored.

  - **Refined Requirements**
    
    - **SUI2.1** There should be a way of logging of session establishment (user id,
log-in time/date, log-out date) and failure to secure storage.

**SUI2.2** If an authentication method involving password is chosen, the password must be stored in a secure way (for example, not in plaintext).

**SUI2.3** If an authentication method involving password is chosen, don’t store/cache password in client.

e tc.

### 5.2.3 Security formalisation suggestion 3

This is another suggestion based on threats. This one is based on treat types and tactics. Tactics are prevention, detection and recovering which are described in section 4.4.5. For each treat type there will be refined requirements about preventing a specific threat type, detecting a specific threat type and recovering from that specific threat type. It is quite useful to think about security in the terms of preventing, detecting and recovering from attacks. The architect makes a selection of which threats the system is suppose to handle. And he or she will then take decisions (not yet considered, must modify architecture, architect already addresses this and not applicable) on the refined requirement for each tactic appropriate. A diagram of this suggestion is presented in figure 41. Such as in the previous two examples the nodes in this hierarchy also needs explanatory text and the refined requirements should have implementation details, rationale, notes etc. where they are needed. The user should also be able to make comments on each refined requirement.

![Diagram of security formalisation suggestion 3](image)

**Figure 40: Security formalisation suggestion 3.**

Below is an example of how this hierarchy can look like:

**Example:**

- **SUI – Spoofing user identity (Threat type)**
  
  *Description:* Spoofing is attempting to gain access to a system by using a false identity. This can be accomplished using stolen user credentials or a false IP address. After the attacker successfully gains access to a legitimate user or host, elevation of privileges or abuse using authorization can begin.

- **SUI.P – Prevention (Tactic)**
  
  **Refined Requirements**
  
  - **SUI.P1** There needs to be secure way of enter the system.
  - **SUI.P2** If passwords are used the password needs to have a minimum level of security.

- **SUI.P2** Refined requirement for detecting spoofing user identity.
etc.

- **SULD – Detection (Tactic)**
  - **Refined Requirements**
    - **SULD1** The actions in the system needs to be monitored.
    - **SULD2** If the wrong password is inserted three times, there should be a security alert, but the login should continue.
    - etc.

- **SULR – Recovery (Tactic)**
  - **Refined Requirements**
    - **SULR1** The system needs to report potential masquerading to a security log.
    - **SULR2** The actions done while masquerading need to be reported.
    - etc.

### 5.2.4 Security formalisation suggestion 4

Suggestion 4 of how to formalise security is different from the previous three suggestions since it is based on use cases and different scenarios that can happen in a system. There are different Use Cases, like Access Control and Integrity and each of these have different use case paths. Each path consists of preconditions, the path and postconditions. The architect will follow the path and take a decision on whether or not the system has the same preconditions. This suggestion is different from the previous ones since it is much more like a story of what the user, misuse and system does in different scenarios and what should be allowed and not allowed to happen in those scenarios. Figure 42 shows a diagram of this suggestion. The user should be able to make comments associated with each decision.

![Figure 41: Security formalisation suggestion 4.](image)

Below is an example of how this hierarchy can look like:

**Example:**

- **Access Control (Use Case)**
  - **Attempted spoofing using Valid User Identity (Use Case Path)**
    - **Security Threat:** The system authenticates and authorizes the misuser as if the misuser were a valid user.

  **Preconditions:**
  1. *The misuser has a valid means of user identification*
  2. *The misuser has an invalid means of user identification*

  **Use Case Path**
  1. *The system shall request the misusers means of identification and*
Authentication

2. The misuser provides a valid means of user identity but an invalid means of user authentication.
3. The system shall misidentify the misuser as a valid user.
4. The system shall not authenticate and authorize the misuser.
5. The system shall reject the misuse by cancelling the transaction.

Postconditions

1. The system shall not have allowed the misuse to steal the user's means of authentication.
2. The system shall not have authenticated the misuse as a valid user.
3. The system shall not have authorised the misuse to perform any transaction that requires authentication.
4. The system shall have recorded the access control failure.

Another use case path under Access Control could be Attempted Identity and Authentication Theft. Some examples of other use cases with some of their possible use case paths could be:

Integrity use case with use case paths like System Data Protected, System Data Corrupted, System Message Integrity, Denial of Service Attack. And Privacy use case with use case paths like Data Privacy.

5.2.5 Analysis of security formalisation suggestions

The different suggestions of how to formalise security above serve different purposes. The suggestions best suited for the ArchWiz tool as it turned out to be e.i. a pure requirements tool, is suggestion 1 that deals with security objectives and mechanisms, suggestion 2 that deals with threat types and countermeasures and suggestion 3 that deals with threat types and tactics. The one preferred by the project owner of the ArchWiz project were suggestion 1 since it is the most aligned with the idea of requirement specifications. Suggestion 2 is very similar to suggestion 1 but it guides the user into thinking about security in terms of threats. The third suggestion is the one that best correspond with the literature about security but it might be more suitable for a threat modelling tool or perhaps the original idea of the purpose of ArchWiz as more of an evaluation tool.

The diagrams of security formalisation suggestion 1-3 have the same visualisation but different content this means that one UML diagram can be implemented in the tool and that implementation supports several different models. This makes it possible for the user to modify the knowledge base in the ArchWiz tool so that it best suits his or her way of thinking about security. It also makes it easier to test the different suggestions to figure out which one is the most suitable to be the default one in the tool. The general UML will have different levels of requirements, if they are made in the way figure 43 shows it is also possible to have the modification of suggestion 1 that is proposed. This model can be called a flexible multi-hierarchical information model. With such a model it is also possible to make a collaboration between the different suggestion. One possible collaboration would be to continue working with tactics and apply those to different security objectives who have refined requirements.
Suggestion 4 is more different than the other three since it focuses on use cases and different scenarios and might better suit a different purpose tool than a pure requirement tool. It leads to far away from the idea and usage semantics of a requirements refinement tool.

Figure 42: Generalisation of suggestions 1-3.
Chapter 6 - Summary and Conclusions

The architecture to a software system is a key part in the development process, it gives the first design decisions and gives information on if the system will have potential to meet its key requirements.

Security is a growing quality to software and then also to the architectural design of the system. There exist a number of potential threats and attacks that might breach the security of the software and these threats needs to be protected against. Having architects incorporate security mechanisms and how to manage security breaches in the architectural design decisions will probably make the system securer than it would be otherwise. Security requirements should be driven by security threats and they are expressed in terms of what must not be allowed to happen, in contradiction to most system requirements that are based on higher level goals and stated in terms of what must happen. Security requirements should be based on an analysis of the assets and services that should be protected, and on an analysis of the security threats from which these assets and services should be protected. This means that security perhaps should be treated in a slightly different way than requirements regarding other qualities.

The ArchWiz tool is a way of helping the architects of ABB to incorporate a bigger number of requirements regarding important qualities, such as usability, security and safety, in their architecture. But to get the tool to be as effective as possible to this concern it needs to incorporate a good formalisation of these qualities that are as default general to all kinds of systems. The user can then specify it to the particular system he or she is working on.

To help architects develop good and manageable designs there exist a big amount of patterns that manage a big variety of problems in different areas. For the field of usability the USAP project team created such patterns. For the field of security many patterns already exist and these are very helpful as guidance of how to develop secure software. These patterns are well known and accepted and should be used in the design and creation of different security aspects, therefore the security patterns have been quite helpful in the development of the suggestions on how to formalise security in the ArchWiz tool.

6.1 Personal Reflections

Working on this thesis have given me a lot of knowledge, not only in the field of architecture, patterns and security, implementing using Eclipse, Java and Jigloo and designing and implementing user interfaces but also how it is to work in a real project in a big company. It has been very interesting to be a part the whole development of a project, different aspects of design, research and implementation. An important insight as I see it is that it's very hard to estimate how long different tasks and activities will take when there are more people to please than the examiner and there are many different forces that affect the progress of the work.

The scope of the ArchWiz tool have during the projects life time changed from being more of an evaluation tool that was quite general, to become a much more requirements specific tool that only focused on one very specific area. Tools should not be made as “tools of the world”, meaning they should not do everything. To be manageable, both in development, maintenance and usage, they need to have a specific task and mission. This was a lesson learned from the ArchWiz project. The first versions of the tool were created to be much more general than what was possible. Once the scope was narrowed down the problem got much clearer and the development got much more efficient.

One thing that was very interesting for me when working in a project at a company was the rocky road the project was driving on. The scope changed and the directions felt different depending on
which way the development was taking. There were many ideas to please and also some difficulties in understanding each other’s ideas and thoughts. This took quite some time and effort but is a thing that needs to be calculated with when working in a project, but perhaps quite not as much as it was in the ArchWiz project. It was quite stressing sometimes but also educational to be a part of it.

With regards to security there exists a lot of literature but one thing that struck me when reading about the quality attribute is that there tend not to be a common language for talking about security matters. Some concepts are the same in most literature, such as asset, threat and attacker. Other things, such as the tactics and basic components for example are discussed in most literature but the concept of them are named different things. This could in some cases make different aspects of security quite confusing.

There are many aspects to security in software and therefore also many different ways that these aspects can be formalised in a tool. Which formalisation are most efficient is up to the business using it and in some cases it may even be up to the person using them, cause even though the aspects of security are the same in most literature, it tends to be a bit more personal how different people think about the subject.

6.2 Future Work

6.2.1 ArchWiz tool
As already mentioned the ArchWiz tool was not finished by the time of the project's closure. The implementation was done halfway and still need work to be a finished, working tool. Whether or not the development of the tool will be continued is up to the project's management to decide. In order to get a fully functioning tool as the project intended the following features needs to be implemented. The import/export functionality needs to be implemented. The todo-list needs to be generated based on the user’s decisions, the check boxes functionality needs to be implemented, the functionality for the decision and comment field needs to be implemented. The trees should be populated with the project file, the properties view in the File menu item should be implemented, functionality for the Next and Previous buttons need to be implemented, the user should be able to add and delete nodes in the tree, the status icons in front of the tree nodes should change depending on the decisions made, version control needs to be managed, progress bar functionality should be implemented as well as the functionality for the help-button. The nodes need to be tagged in somewhat based on if they are decision nodes or text nodes.

6.2.2 Security formalisation
With regards to how security can be formalised in the ArchWiz tool, suggestions have been constructed. Suggestion 3 that is based on threats and tactics could perhaps be developed to use tactics as scenarios. Depending on how the refined requirements look like it might be possible to have one refined requirement that architects take decision on regarding each of the tactics. This suggestion is shown in figure 44.
Whether or not this approach is possible depends on how the refined requirements will look like. To see if this formalisation is appropriate or not, the tool needs to be tested with the formalisation implemented, which have not been done since the tool is not finished to that extent.

The other suggestions presented in section 5.2 are also in need to be tested if some of them are supposed to be used as a standard knowledge base. What can be interesting to test regarding these suggestions are whether the user prefers to look at security as a pure requirements as in suggestion 1, or as requirements based on threats to the system as in suggestion 2 and 3. Suggestion 4 is interesting to test to see if the idea of following a path could trigger the thinking of security attributes in the system.

The number of existing security patterns are big and it would be a good idea to classify more patterns with the security objectives and also with the biggest threats to software if any of the suggestions using threats will be used as a knowledge base.

\[\text{Figure 43: A possible way to develop security formalisation suggestion 3.}\]
Chapter 7 – References


[21] Stoll, P., Alfredsson, F., Lövemark, S. *Usability Supporting Architectural Patterns for


