Architecture-based Uncertainty Impact Analysis for Confidentiality

Master's Thesis of

Niko Benkler

at the Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Sebastian Hahner
Second advisor:	M.Sc. Maximilian Walter

August 23, 2021 – February 23, 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 23. February 2022

. . . .

(Niko Benkler)

Abstract

In times of highly interconnected systems, confidentiality becomes a crucial security quality attribute. As fixing confidentiality breaches become more costly the later they are found, software architects should address confidentiality early in the design time. During the architectural design process, software architects use Architectural Design Decisions (ADDs) to handle the degrees of freedom, i.e. uncertainty. However, ADDs are often subjected to assumptions, and unknown or imprecise information. Assumptions may turn out to be wrong and need to be revised. This re-introduces uncertainty. Thus, uncertainty at design time prevents from drawing precise conclusions about the confidentiality of the system. It is therefore necessary to assess their impact at architectural level before making a statement about confidentiality. So far, this assessment is manual and very tedious, and requires a great deal of knowledge about uncertainties, their characteristics and impact on the confidentiality of systems. Current approaches dealing with uncertainties in software systems do not consider them at design time, i.e. in software architectures, but often in other contexts such as self-adaptive systems.

To close this gap, we make the following contributions: First, we propose a novel uncertainty categorization approach to assess the impact of uncertainties in software architectures. Based on that, we provide an *uncertainty template* that enables software architects to structurally derive types of uncertainties and their impact on architectural element types for a domain of interest. Second, we provide an *Uncertainty Impact Analysis (UIA)* that enables software architects to specify which architectural elements are directly affected by uncertainties. Based on structural propagation rules, the analysis automatically derives further architectural elements which are potentially affected.

We evaluate the structural quality, applicability and the purpose of the *uncertainty template*. We show that underlying categories fulfil principles of good classifications, such as orthogonality, exhaustiveness and differentiability. We demonstrate its capability to derive uncertainty types, their impact and how it enables the reuse of knowledge and creation of awareness. Lastly, we illustrate the relevance of the uncertainty template by showing that it can classify uncertainties in software architectures more accurately and thus make more precise statements about their impact on the system compared to existing taxonomies. The UIA is evaluated with regard to usability, functionality, and accuracy. We show that the UIA increases the usability by reducing the required amount of expertise when dealing with uncertainties compared to a manual analysis. Using the large-scale open-source contract tracing application called Corona Warn App (CWA) as case study, we show that the UIA reduces the amount of elements to be examined by 85% compared to the amount of a manual analysis. We further illustrate how it enables architects to explicitly manage uncertainties during design time. Using the CWA case study, we show that the UIA achieves 100% recall of the actually affected elements, while maintaining 44%-91% precision.

Zusammenfassung

In Zeiten vernetzter Systeme ist Vertraulichkeit ein entscheidendes Sicherheitsqualitätsmerkmal. Da die Behebung von Vertraulichkeitsverletzungen kostspieliger wird, je später sie entdeckt werden, sollten Softwarearchitekten diese bereits in der Entwurfsphase berücksichtigen. Während des Architekturentwurfsprozesses treffen Architekten Entwurfsentscheidungen um Ungewissheit zu reduzieren. Allerdings unterliegen Entscheidungen oft Annahmen und unbekannten oder ungenauen Informationen. Annahmen können sich als falsch erweisen und müssen revidiert werden. Dies verursacht erneut Ungewissheit. Ungewissheiten zur Entwurfszeit machen genaue Schlussfolgerungen über die Vertraulichkeit des Systems daher unmöglich. Es ist also notwendig, ihre Auswirkungen auf Architekturebene zu bewerten, bevor eine Aussage über die Vertraulichkeit getroffen wird. Bisher ist diese Bewertung manuell und mühsam und erfordert ein großes Maß an Wissen. Derzeitige Ansätze berücksichtigen Ungewissheiten nicht zur Entwurfszeit, sprich in Softwarearchitekturen, sondern in anderen Bereichen wie z.B. bei selbst-adaptiven Systemen.

Diese Lücke wollen wir wie folgt schließen: Erstens stellen wir einen neuen Ansatz zur Kategorisierung von Ungewissheiten vor. Darauf aufbauend stellen wir eine *Ungewissheitsschablone* zur Verfügung, die es Architekten ermöglicht, Typen von Ungewissheiten und deren Auswirkungen auf Architekturelementtypen für eine Domäne strukturell abzuleiten. Zweitens stellen wir eine *Ungewissheits-Auswirkungs-Analyse* vor, die es Architekten ermöglicht zu spezifizieren, welche Elemente direkt von Ungewissheiten betroffen sind. Basierend auf strukturellen Ausbreitungsregeln leitet die Analyse automatisch weitere Elemente ab, die potenziell betroffen sein könnten.

Es wird die strukturelle Qualität, Anwendbarkeit und Zweck der Schablone evaluiert. Wir erläutern, dass die Kategorien Prinzipien wie Orthogonalität, Vollständigkeit und Unterscheidbarkeit erfüllen. Außerdem zeigen wir, dass sie dabei hilft Ungewissheitstypen und deren Auswirkungen abzuleiten, sowie Wiederverwendbarkeit und Bewusstsein schafft. Schließlich veranschaulichen wir die Relevanz der Schablone, indem wir zeigen, dass sie im Vergleich zu bestehenden Taxonomien Ungewissheiten in Softwarearchitekturen genauer klassifizieren und somit präzisere Aussagen über deren Auswirkungen machen kann. Die Analyse wird im Hinblick auf Benutzerfreundlichkeit, Funktionalität und Genauigkeit bewertet. Wir demonstrieren, dass die Analyse die Benutzerfreundlichkeit erhöht, indem sie die erforderliche Menge an Fachwissen beim Umgang mit Ungewissheiten im Vergleich zu einer manuellen Analyse reduziert. Anhand der Kontaktnachverfolgungs-Applikation *Corona-Warn-App* zeigen wir, dass die Analyse die Anzahl der zu untersuchenden Elemente im Vergleich zu einer manuellen Analyse um 85% reduziert. Darüber hinaus veranschaulichen wir, wie sie Architekten ermöglicht, Ungewissheiten während der Entwurfszeit explizit zu verwalten. Anhand der Fallstudie zeigen wir, dass die Analyse eine 100%ige Ausbeute bei einer Präzision von 44%-91% hat.

To Katrin, Sigrid and Nikolaus

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

In times of highly interconnected systems, confidentiality becomes a critical security quality attribute. The *International Organization for Standardization (ISO)* describes confidentiality as "property that information is not made available or disclosed to unauthorized individuals, entities, or processes" [35]. Compared to other quality attributes such as reliability and performance, confidentiality has legal and social implications [61]. Hence, software manufacturers must consider confidentiality at every stage of the development to ensure compliance [61]. Not only is concrete protection against security threats necessary, but also compliance with legal requirements, such as the European General Data Protection Regulation (GDPR). It demands, for instance, that storing and processing *EU* citizens' data only take place on servers located in the *EU* [28]. Further, confidentiality highly affects user acceptance. For instance, *Facebook* issued a new privacy policy for its messenger service *WhatsApp*, enabling them to collect metadata about sender, receiver or time a message was sent. After the announcement of the change, download numbers for Signal, a rival messenger service, have risen from 246.000 to 8.8 million in the following week [34].

According to Boehm and Basili [12], finding and fixing software issues, such as breaches in confidentiality, become more costly the later they are found. As finding confidentiality issues manually is a cumbersome task, software architects defined architectural confidentiality analyses based on the flow of data and the definition of data flow constraints within the software architecture to evaluate confidentiality in the design time [30, 61, 13].

During the architectural design process, various design issues, design alternatives or trade-offs for competing requirements need to be considered [45]. Those decisions are also known as Architectural Design Decisions (ADDs). Jansen and Bosch [37] introduce a new perspective on software architecture as being a "set of architectural design decisions" [37]. This view is widely accepted in the scientific community. With each ADD taken, the software architecture ideally becomes more accurate as more and more degrees of freedom (i.e. uncertainty) are removed. This phenomenon is described by the *cone of uncertainty*, first introduced by McConnell [46]. However, in the early design time, ADDs are often subjected to assumptions due to imprecise or unknown information. These assumptions may turn out to be wrong at a later stage, so that ADDs have to be revised. This re-introduces further uncertainty in the process of making rational ADDs and can thus enlarge the cone of uncertainty again [45]. As a consequence, the presence of uncertainty, especially in the early stages of the architectural design process, discourages drawing precise conclusions about the confidentiality of the overall system [30]. Further, some assumptions may not be resolvable during design time because the necessary information to resolve the accompanying uncertainty is not available until, for instance, the system gets deployed.

Therefore, architects need to analyse the impact these uncertainties have on other ADDs, or according to Jannsen's and Bosch's perspective [37], on which elements of the software architecture. We describe this effect as *propagation of uncertainty*, since elements of an architecture can be indirectly and therefore less obviously affected by uncertainty. Hence, a sound understanding of uncertainty and the propagation of uncertainty plays a crucial role when reasoning about the confidentiality of a system at design time. We therefore propose to consider uncertainty as first-class entity in the course of the architectural software development process but especially in the software architecture [26]. This makes it possible to not only represent uncertainty in the architectural model, but also to analyse its propagation effects and impact on confidentiality in a more structured way.

To do so, it is important to be able to identify and classify uncertainties in the architecture. A classification by categories helps to group uncertainties with a similar impact in order to better understand their characteristics and propagation effects. So far, interdisciplinary uncertainty taxonomies such as the one proposed by Walker et al. [73] have been adapted to better fit the needs of software architects [52, 14]. The majority of existing taxonomies are used to categorize uncertainties in Self-adaptive Systems (SASs) [21, 49], thus uncertainties at runtime. None of the available taxonomies are used to categorize uncertainties at architectural level. However, there are approaches that try to handle uncertainty in early software architectures by providing tool support such as the works proposed by Esfahani et al. [23] or Lytra and Zdun [45]. Those tools are mainly used to guide the architect through the decision making process. Here, uncertainty is limited to the impact of individual ADDs on quality properties, which cannot be precisely predicted but quantified using fuzzy mathematical methods [18]. By combining various ADDs, the tools are capable to reason about the impact on quality properties with fuzzy logic. One of the weaknesses of common tools is that they are not able to incorporate uncertainties into architectural models and other artefacts [44]. Further, the limitations are too narrow, as uncertainty does not only prevail due to unknown effects of ADDs.

Although a lot of research has been done in recent years to understand the impact of uncertainty in software development, there is still a "gap in the understanding of what constitutes uncertainty and the sources of uncertainty" [32] so that it is "essential to consolidate the knowledge on uncertainty and its sources in a standardised format" [32]. Further, to the best of our knowledge, current approaches do not investigate the impact of uncertainty on confidentiality at the architectural level, nor the general propagation of uncertainty in software architectures. This is what we aim to do in this thesis.

1.2. Motivating Example

Figure 1.1 illustrates a simple architectural diagram which shows two services Component A and Component B, as well as the deployment locations Cloud Server (EU), Cloud Server (Non-EU) and an On Premise Server. EU/Non-EU describes whether or not the location of the server is within the European Union. Further, it illustrates an activity diagram that refers to the behaviour of Component A. The dashed boxes represent uncertainties that are added to the architecture as a first-class entity. At the beginning of the architectural design process, there are various uncertainties (as indicated by the question marks). For instance,

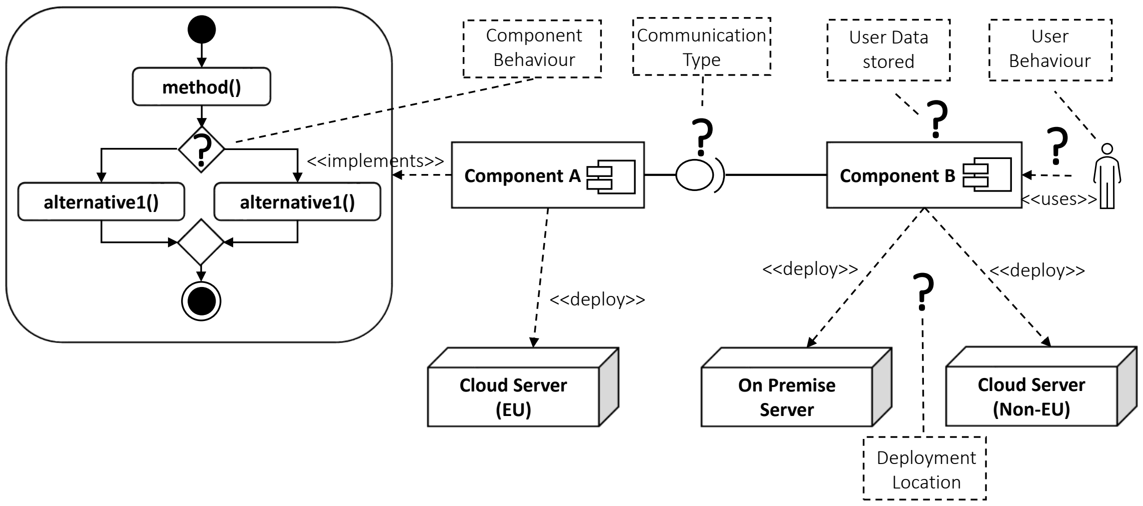


Figure 1.1.: Simple Architectural Diagram showing Uncertainty [30]

the communication type can be encrypted or unencrypted. Further, the services can be either deployed to an on premise server or somewhere in the cloud. Likewise, the internal behaviour of the components, as well as their handling of user data, may be uncertain. Another source of uncertainty is the user behaviour. Each of the uncertainties, as indicated in Figure 1.1, is assigned to an element of the software architecture, i.e. components, communication, resource environment, and usage behaviour.

Uncertainties that exist due to ADDs that have not yet been made can still be eliminated in the course of the design process. An example is the communication type to be used. Others, such as the uncertain behaviour of components, can also be eliminated through knowledge acquisition during the design process. Whether user-related data is stored should already be clarified in the requirements. If this information is missing, it can also be obtained during the design process. The deployment location, in turn, can be specified by software architects. However, the underlying uncertainty is not completely eliminated, as it is only clear at the deployment time. The uncertainty therefore persists, at least partially, in the form of an assumption. These can be revoked in the course of the design process and also afterwards. In contrast, user behaviour can only be described statistically or by empirical values. Uncertainty inevitably remains until the system is actually running, albeit with statistical means and design decisions in a reduced form. Although uncertainties can have an impact on various quality attributes of the system such as reliability or availability [26], we only focus on those with an impact on confidentiality. Therefore, most uncertainties concern user data and how or where it is stored, transferred and used.

For each of the uncertainties shown in Figure 1.1, we described where they have a direct impact, i.e. to which architectural elements they can be assigned. Further, we elaborated some characteristics with regard to resolution time and manageability. What is missing is another important characteristic of uncertainty: *propagation*. Uncertainty can be assigned to one element but have an impact on another ADD or, according to Jansen and Bosch [37], on other elements. We describe these effects as *uncertainty propagation*, because uncertainty propagates from one element of the architecture to another. In those cases, uncertainties have an indirect impact on architectural elements. For example, the deployment of a system has an impact on the choice of communication, as cloud deployment demands the usage of encrypted communication. If not, user-related data could be intercepted and the confidentiality of the system would no longer be guaranteed. Therefore, the uncertainty Deployment Location has an impact on the architecture element Communication. Furthermore, it is important to know whether a component processes user data or not, due to the GDPR [28]. Consequently, the uncertainty User Data Stored propagates to architectural elements describing the resource environment such as servers. This is only a small sample of possible propagation effects, but it demonstrates how important it is to take them into account. While this example is very small and easy to analyse manually, larger architectures won't. Here, it will be advantageous to use an automatic analysis that can determine the impact on confidentiality based on explicitly defined uncertainty types in an architecture.

1.3. Research Goals and Contributions

The goals of this thesis are twofold: First, we want to be able to analyse and assess the impact of uncertainties on the confidentiality at design time. Therefore, we analyse the relationship between ADDs and uncertainties with a foreseeable impact on confidentiality. To asses uncertainties and their impact, we propose new and enhanced categories to categorize uncertainties in software architectures. The categories are based on available uncertainty taxonomies and related literature. They differ from all previous ones because, on the one hand, they incorporate the evaluated and combined findings of a large number of existing taxonomies, and, on the other hand, they allow a categorisation of uncertainties in software architectures for the first time.

Based on the categories, we propose an *uncertainty template* that guides software architects through the process of identifying and analysing uncertainties in software architectures. Also, we provide an approach how software architects can use the uncertainty template to extract types of uncertainties and their impact on types of architectural elements based on existing ADDs. A template filled in by experienced architects creates awareness as less experience software architects can instantiate actual uncertainties in actual software architectures, and asses their impact based on the previously defined uncertainty types.

The second goal is to reduce manual effort and required expertise when analysing the impact and propagation of uncertainties in software architectures. To that end, we extend an available Architectural Description Language (ADL) called Palladio Component Model (PCM) [56] by the capability to model uncertainty as first-class entities in software architectures. Based on that extension, we create an Uncertainty Impact Analysis (UIA) which reduces the required expertise and manual effort when analysing the impact and propagation of uncertainties in software architectures. To achieve the goals, the following research questions (**RQs**) have to be answered in the course of this thesis:

RQ1: How to analyse the impact of uncertainty on the confidentiality of a system at design time?

RQ1.1: How to describe the relationship between uncertainty and confidentiality in software architectures?
Our goal is to elaborate the relationship between uncertainties and ADDs and how it is possible to derive uncertainties from them. The focus is on ADDs with a foreseeable impact on confidentiality.

RQ1.2: How to categorise uncertainties in software architectures with an impact on confidentiality?
To that end, we aim to derive categories based on existing uncertainty and design decision taxonomies. Also, we want to describe the process of finding those categories.

RQ1.3: How can the categorisation be illustrated?
Here, our goal is to provide a template which enable architects to categorize ADDs and their related uncertainties in a structured manner.

RQ1.4: How can the template be used to derive uncertainties and assess their impact on the software architecture?
To answer this question, we strive to describe a systematic approach on how to derive types of uncertainties and their possible impact on architectural elements based on existing ADD. Our goal is further to demonstrate how this approach can be used to annotate software architectures with actual uncertainties and how it is possible to derive and asses their impact.

RQ2: How to support the annotation and propagation of uncertainties in software architectures?

RQ2.1: How can uncertainty be represented in the software architecture?
Here, we will extend the Palladio Component Model (PCM) by uncertainty as first-class entity. Characteristics of this elements shall be inspired by the categories presented in *RQ 1.2*.

RQ2.2: How does uncertainty propagate in the architecture?
To answer this question, we plan to define and implement an Uncertainty Impact Analysis (UIA) based on the extended PCM.

1.4. Outline

This thesis is structured as follows: Chapter 2 provides the fundamentals for this thesis such as confidentiality, Component Based Software Engineering (CBSE) or the PCM. Chapter 3 presents the related work in the domain of uncertainty. In Chapter 4 we illustrate the relationship between uncertainties, ADDs and architectural elements which provides the basis for our contributions. In Chapter 5, we asses existing uncertainty- and architectural related taxonomies and provide our own categorization approach. Based on that, we present our *uncertainty template* in Chapter 6, including an approach to derive types of uncertainties and their impact based on existing ADDs. Chapter 7 presents our Uncertainty Impact Analysis. In Chapter 8, we evaluate our approach. We conclude this thesis with Chapter 9 and provide ideas for future work.

2. Foundations

This chapter provides the foundations for this thesis. Section 2.1 introduces the quality property *confidentiality* and illustrates its importance. Further, Section 2.2 defines *uncertainty* and illustrates its relationship to information. Section 2.3 introduces the Component Based Software Engineering (CBSE), which is the underlying concept of this thesis. Section 2.4 introduces the Eclipse Modelling Framework (EMF). Section 2.5 presents the used Architectural Description Language (ADL) - the Palladio Component Model (PCM). Lastly, Section 2.6 introduces a change propagation analysis framework.

2.1. Confidentiality

Confidentiality, together with integrity and availability, forms the so-called *CIA* triad. Together, the three quality attributes represent the cornerstone of *Information Security* [64]. *Integrity* refers to the accuracy and completeness of data whereas *availability* describes that data is available when needed [64]. The *International Organization for Standardization* (ISO) defines *confidentiality* as "property that information is not made available or disclosed to unauthorized individuals, entities, or processes" [35]. Confidentiality therefore concerns data that is to be protected from unauthorised access. Although data and information have an etymological difference ("information consists of processed data" [2]) we use both terms interchangeably in the course of this thesis. Furthermore, we restrict ourselves to confidential data. So when we talk about data in general, we always refer to confidential data. Otherwise, the term "public data" will be used if public, i.e. non-confidential, data is meant. There are various examples of confidential data, first and foremost user-related data, such as social security numbers or passwords. Another example is *GPS* data, from which user-related data can be derived. In addition, non-personal data can also be confidential, such as secret keys for the creation of certificates.

According to Boehm and Basili [12], finding and fixing software issues, such as breaches in confidentiality, become more costly the later they are found. Confidentiality must therefore be respected early in the software development process, such as in the software architecture design time. To that end, software architects defined architectural confidentiality analyses based on the flow of data and the definition of data flow constraint within the software architecture description. Using access control policies, such analyses enable to evaluate confidentiality in the early software architecture design time [30, 61, 13]. However, uncertainties in the early architecture make it difficult for these analyses to draw clear conclusions [28, 22].

2.2. Uncertainty

Uncertainty in general can be described as "any deviation from the unachievable ideal of completely deterministic knowledge of the relevant system" [73]. This definition also emphasizes the fact that uncertainty can't typically be eliminated completely, or at least not in a practically meaningful way [21]. In order to make uncertainty manageable and better understandable, researchers created taxonomies [73, 52, 14], and identified sources of uncertainty [21, 51]. There is a general consensus that uncertainty is divided into two basic types: *epistemic* uncertainty and *aleatory* uncertainty [73, 44, 52, 14]. Epistemic uncertainty describes the lack of knowledge and can be mitigated by gathering more information. However, new information can reduce uncertainty but also lead to an increase, as new knowledge can reveal previously unknown uncertainties [73]. In contrast, aleatory uncertainty is caused by the inherent variability of the nature or due to random events [28]. Consequently, this type of uncertainty cannot be reduced by acquiring more information. Some authors map aleatory uncertainty to irreducibility and epistemic uncertainty to reducibility [72]. A different view is provided by Esfahani and Malek [21] who disagree as "both the reducible and irreducible uncertainties can have aleatory and epistemic components" [21]. They argue that *aleatory* and *epistemic* represent the essence of uncertainty whereas *reducibility* and *irreducibility* refer to the managerial aspect. We share this view, because although aleatory uncertainty is caused by randomness, it often can be modelled using statistical methods. An example is the modelling of user behaviour which is aleatory by nature but is often modelled via statistical means [56]. As this makes uncertainty more manageable, it reduces uncertainty to a certain extent. Nevertheless, statistics can only make predictions. Thus, by definition, there can be no certainty about the actual outcome. To that end, Esfahani and Malek [21] propose a concept called *the spectrum of uncertainty*, which illustrates the relationship between information and uncertainty. This concept is shown in Figure 2.1.

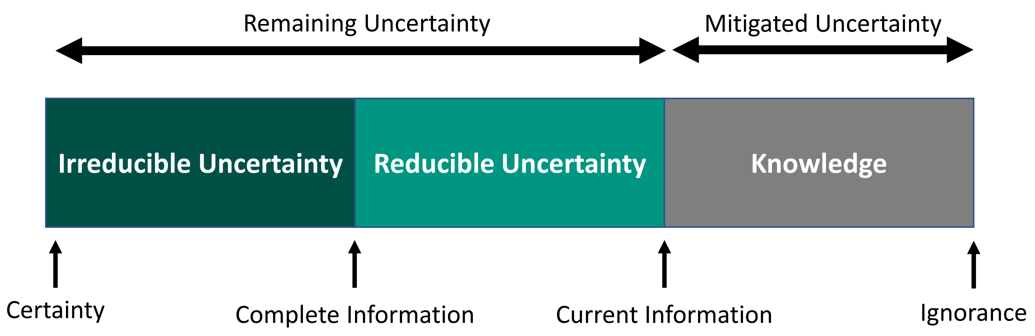


Figure 2.1.: Spectrum of Uncertainty [21]

The concept illustrates the spectrum of uncertainty by using a linear scale of available information with the idealistic *certainty* at one end, where all information is available, and total *ignorance* at the other, where no information is available. Two arrows indicate the flexible states *current information* and *complete information* along the knowledge scale.

This produces a spectrum with the three areas *irreducible uncertainty*, *reducible uncertainty* and *knowledge*, i.e. no uncertainty. First, *knowledge* indicates the amount of information already acquired, and therefore the mitigated uncertainty. *Reducible uncertainty* represents the area between the *current information* and the *complete information* as this type of uncertainty can be reduced by acquiring more information. Therefore, *complete information* represents the maximum threshold to which *current information* can move. *Irreducible uncertainty* represents the range between *complete information* and *certainty*, as no further information is capable to reduce this type of uncertainty. It should be noted that the ranges can be of different sizes and even zero. This is the case if for instance all reducible uncertainty is actually reduced so that *current information* and *complete information* fall to the same point. Similarly, it is possible that the system to be described is not affected by irreducible uncertainty. In this case, *complete information* and certainty are identical. Ideally, *certainty*, *complete information* and *current information* are then at the same point, so that only knowledge prevails. However, this should be rather theoretically possible as already the presence of human user input is a source of irreducible uncertainty. More generally, any form of environment or context can create irreducible uncertainty. Other sources of uncertainty are the environment, available resources, organizational structures, missing requirements or unexplored design alternatives [31, 44].

2.3. Component-based Software Engineering

Component Based Software Engineering (CBSE) is a software programming paradigm where functionality is bundled as software packages - the so called *components*. Architectures which are developed according to this principle are called CBSAs. A comprehensive introduction into CBSE is provided by Szyperski et al. [67]. They describe three key characteristic of components. First, components are units of independent deployment. Second, components are units of third-party composition. Third, components have no (externally) observable state. To be composable, each component has *provided* and *required* interfaces, which together form the *component signature*. *Provided interfaces* describe the functionality that a component is able to provide, whereas *required interfaces* describe which other components, or specifically which specific interfaces are needed to function. Software

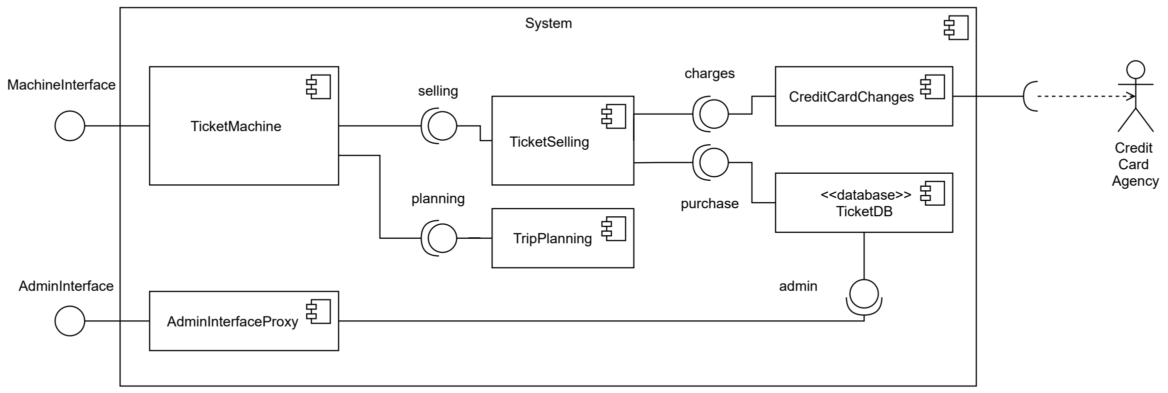


Figure 2.2.: CBSA Example - Ticket Machine

architects use basic components to create composite components, thus building a hierarchy of components [56]. The topmost component can thus be described as the *system*, which in turn has interfaces to interact with. Figure 2.2 shows a simple example of software components that together represent a system. In addition to components and interfaces, CBSE is characterised by the ability to represent other concepts as first class entity, such as the ones illustrated in Table 2.1. The supported concepts depend on the ADL used, so please note that Table 2.1 only provides an excerpt of possible concepts supported by the PCM. Other ADL may support other concepts, or use a different naming.

Concept	**Description**
Basic Component	Basic entity that bundles functionality
Composite Component	Composition of several components
Component Repository	Collection of available components
Interface (provided / required)	Means by which components connect
Assembly Context	Provide context to instantiate components
Assembly	Set of assembly contexts
Connector	Link between interfaces
Resource Container	Set of processing resources for hosted components
Link	Connects resources containers
Usage Behaviour	Representation of the system's usage

Table 2.1.: First Class Entities in the PCM (Excerpt)

CBSE has several advantages over object-oriented or procedural development approaches, such as separation of concerns, the encapsulation of functionality, reusability and facilitated evolution [56]. Further, components can easily be encapsulated as services, which enables their applicability in Service-oriented Architectures (SOAs) and the Microservice architecture [9]. However, CBSE also has its disadvantages, since components are usually considered as black boxes, which means that software architects only know the *component signatures* in the first place, but have no knowledge about the internals. But if these are needed, for example, to carry out confidentiality analyses, this view complicates the situation. In general, CBSE is the fundamental concept for a variety of successful component models such as the *Enterprise JavaBeans* model[1] or the *.NET* model[2].

2.4. Eclipse Modeling Framework

The Eclipse Modelling Framework (EMF)[3] is an Eclipse extension for modelling and data integration [15]. EMF provides various concepts and tools to define and extend meta-models based on the meta-meta model *Ecore*. *Ecore* is an Essential MOF (EMOF) compatible

[1]https://docs.oracle.com/javaee/5/tutorial/doc/bnbls.html
[2]https://dotnet.microsoft.com/
[3]`https://www.eclipse.org/modeling/emf/`

meta-meta model, which is an implementation of a subset of Meta-Object Facility (MOF). The EMF environment provides various tools to support the creation of Eclipse plug-ins for graphical model editors or automatic code generation. The exchange format is XML Metadata Interchange (XMI)[4], so that instances of the *Ecore* model, such as the PCM model are persisted in XMI format.

2.5. Palladio Component Model

The Palladio Component Model (PCM) is a domain-specific meta-model to describe CBSEs. The *Palladio Workbench* is a platform that provides tools and concepts to model and analyse instances of the PCM model with regard to quality characteristics [55]. Although its main purpose is the prediction of performance characteristics, various add-ons have been proposed to e.g. analyse scalability characteristics of SASs at design-time [7], or to provide maintainability predictions on architectural level [57]. An overview of the PCM approach is provided in Figure 2.3.

The PCM is divided into four sub-models, which represent different views on the software architecture: the *Component Specifications* also know as *Component Model* [55], the *Assembly Model*, the *Allocation Model* and the *Usage Model* [56]. The *Component Model* specifies the components which included a model of the component behaviour. Further, it defines required and provided interfaces for each component. Provided interfaces are the services a component can offer to other components whereas required interfaces specify which external services the component needs, in order to fulfil its purpose [55]. The *Assembly Model* is used to explicitly describe the wiring of the components, i.e. the structure of the system. The *Allocation Model* describes the allocation of components and connections to physical resources. The *Usage Model* specifies the workload induced by

[4]`https://www.omg.org/spec/XMI/2.5.1/About-XMI/`

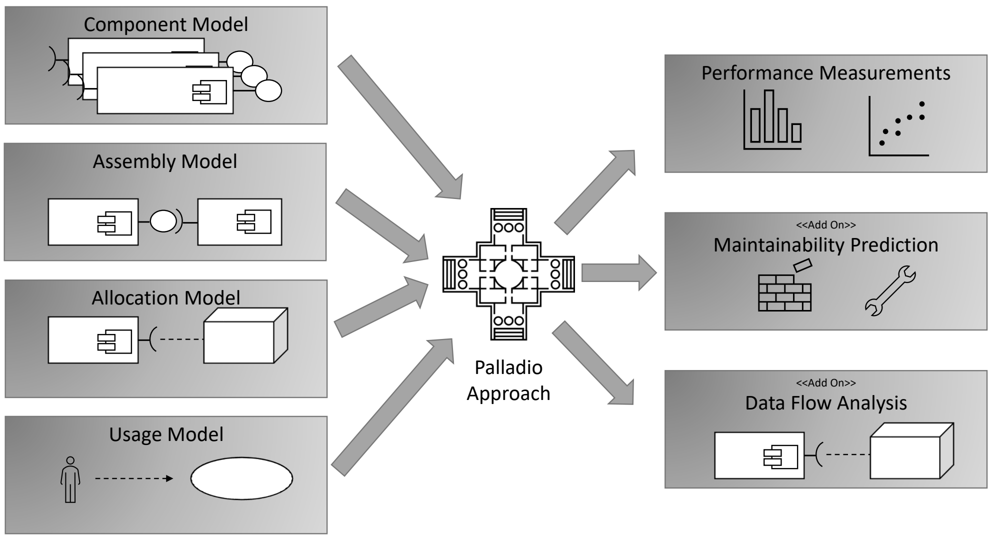


Figure 2.3.: The Palladio Approach - Overview [55]

the system's end user [55]. Reussner et al. [56] provide a sound overview regarding the technical aspects of the PCM.

Figure 2.4 shows the classification of the PCM in the four modelling layers as proposed by the MOF standard of the Object Management Group (OMG)[5]. The model layers are numbered from M0 to M3. Between the layers there is a one-to-many relationship, i.e. a model of a higher layer can represent several models of a lower layer. Models at layer M0 are called *Originals*. In our case, *Originals* are the software systems. M1 models are referred to as *Models* and describe, for instance, the system's architecture. The PCM model is located at M2, the meta-model layer. As the PCM is a domain-specific meta-model, it is limited to the description of architectures in the domain of CBSE. However, the PCM itself is based on a M3 layer model called *Ecore* which is a meta-meta model with self-describing capabilities. *Ecore* comes with the Eclipse Modelling Framework as elaborated in the previous section.

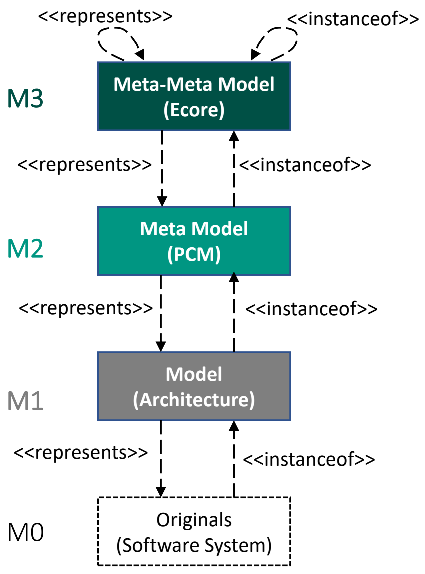


Figure 2.4.: Meta-Modelling - Basic Concepts [15]

2.6. KAMP Approach

The Karlsruhe Architectural Maintainability Prediction (KAMP) framework is a change propagation analysis tool that analyses the effects of change requests due to change propagation in software systems on the architectural level. The analysis is conducted from both, the technical and the organizational perspective and concerns the whole software life-cycle [57]. Change requests can be implemented in various ways resulting in different effort when it comes to the adaption of code and other related artefacts such as tests, deployment or the architecture itself [57]. This is closely related to the overall maintainability of the system as a low effort indicates high maintainability [66].

[5]https://www.omg.org/mof/

Change propagation analyses differentiate between three sets of changing elements, which is illustrated in Figure 2.5. The *change set* is composed of the elements that are directly affected by the initial changes. In contrast, the *affected set* consists of the elements that are actually affected by changes due to change propagation effects. However, the exact determination of this set is impossible for greater systems, because the problem can be reduced to the undecidable halting problem [1]. Therefore, change propagation analysis, such as KAMP, always provide an over-estimation of possibly affected set - the *impact set.* As shown in Figure 2.5, the *change set* is a subset of the *affected set* which in turn is a subset of the *impact set.*

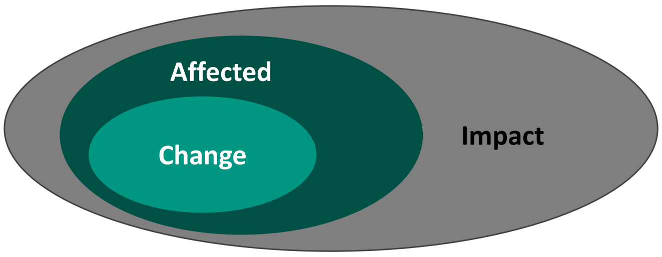


Figure 2.5.: Change Propagation Analysis - Overview [58]

The KAMP framework extends the Palladio approach and has been instantiated for various domains, such as Information Systems (IS), referred to as *KAMP4IS* and Business Processes (BP), called *KAMP4BP* [58]. Each implementation defines different types of possible (initial) change requests as well as various propagation rules. The overall core process, however, is consistent and is divided into three phases: the *preparation phase*, the *impact phase* (sometimes also referred to as analysis phase) and the *postanalyze phase* [57, 66, 58]. Rostami et al. [57] provide a formalisation of the process. In the preparation phase, software architects first model the corresponding software architecture using the *Palladio Workbench.* Architects can then annotate the software architectures with additional context information such as test cases, build configurations, or object-relational database mappings [58]. Finally, the initial change requests are modelled. For each change request, software architects specify which architectural elements have to be initially modified. The input thus consists of an annotated architecture model, as well as the specification of the change requests. The defined propagation rules are iteratively applied in the second phase. Thus, starting from the initial change requests, KAMP calculates elements that may be affected by this change. Besides the propagation based on the structure of the software architecture, KAMP also provides the possibility to define other propagation types. For instance, it is possible to determine the impact on organisational tasks, such as test cases to be changed, build configurations and other affected artefacts [57]. Thereby, it is possible to refine the initial structural propagation by defining further rules, so that the result of the propagation is more fine-grained [58]. In the *postanalyze phase*, the results of the propagation are processed. Based on the *base architecture*, which is the initial annotated architecture, and the *target architecture*, which represents the architecture after propagation, KAMP creates a task list that contains all the necessary tasks to implement change requests.

3. Related Work

This chapter outlines related work and their contribution to the proposed thesis. We conducted an extensive literature review with regard to existing works of uncertainty in software architectures, particularly in relation to the impact on the confidentiality of a system. The findings are summarized in the following sections, which are loosely categorized according to their content.
The focus of our work is to analyse i) where uncertainty emerges in the software architecture ii) how uncertainty can be represented as first-class entity in software architectures iii) how uncertainty propagates in software architectures and thus impacts the overall confidentiality. The literature review reveals that current approaches do not adequately cover these points. It also shows that the focus of current research is on the runtime analysis of uncertainty. So far, there are papers that deal with uncertainty at design time, but none of them deal with uncertainty explicitly in software architectures. This is the gap we want to close with this thesis.

3.1. Literature Review

The findings presented in this chapter are the result of an literature review which was conducted using the digital libraries IEEE [1], ACM [2] and SpringerLink[3]. The web search engine Google Scholar [4] provided further approaches and general information.

["Uncertainty Impact Analysis"]
OR
["Software Architecture" AND "Confidentiality" AND "Uncertainty"]
OR
["Software Architecture" AND ["Confidentiality" OR "Uncertainty"]]
OR
["Architectural Design Decision" AND "Confidentiality"]
OR
["Software Design Decision" AND "Impact on Confidentiality"]

Apart from the search string "Uncertainty Impact Analysis", each query resulted in at least one related work. More related work was identified by analysing outgoing and ingoing references, as well as provided by the supervisors of this thesis.

[1]http://ieeexplore.ieee.org
[2]http://portal.acm.org
[3]http://www.springerlink.com
[4]http://scholar.google.com

3.2. Findings

Confidentiality Analyses: Design-time confidentiality analyses are not new to scientific research [61, 62, 13, 30]. Although all of the approaches found do not take uncertainties explicitly into account, they are nevertheless the basis of this thesis as they might be extended by uncertainty characteristics. Seifermann et al. [62] propose a data-driven analysis to cope with confidentiality at architectural level. The approach integrates Data Flow Diagrams (DFDs) in the PCM to specify confidentiality constraints on architectural level. Boltz et al. [13] present an analysis process to find confidentiality issues on the architectural level for industrial Internet of Things (IoT) systems. Although this approach provides a solid confidentiality analysis for the design time, it does not take into account potential uncertainties in the analysed software architecture and their impact on the accuracy of the analysis. Hahner et al. [30] provide an approach to model data flow constraint for design-time confidentiality analyses based on Seifermann et al. [61]. Therefore, they specified a Domain-specific Language (DSL) using architectural terminology, so that software architects can specify data flow constraints on the architectural level. The results of the analysis are presented at the architecture level so that the architecture domain does not have to be left. Again, this approach does not consider uncertainty at all.

Walter et al. [74] elaborate that dynamic changes in production environments are inevitable, but need to be managed to preserve confidentiality. According to their definition, "a dynamic change can be every context change during runtime, which is detectable during runtime and foreseeable during design-time" [74]. Although they consider changes during runtime, the definition does not cover uncertainty comprehensively, as changes need to be foreseeable during design-time, which is certainly not always the case.

Uncertainty: Garlan [26] justifies why uncertainty should be considered as *first-class entity* in design, implementation and deployment of software systems. So far, uncertainty in software systems is mainly considered by implementing self-adaption capabilities [49, 52, 21, 47]. This is the case, if software systems are capable to adapt their behaviour dynamically at run-time to handle different situations in which components fail, unforeseen changes in environment or unexpected user behaviour occur [14]. Self-adaptive Systems (SASs) monitor the current state of the system and apply adaption strategies to control the system behaviour. An example is the *RAINBOW* approach presented by Cheng and Garlan [19]. Further, Esfahani and Malek [21] identify and classify sources of uncertainties and their impact in SASs. The work aims to demonstrate the importance of uncertainty in SASs and explains their most important characteristics. Musil et al. [49] provide patterns on how to handle uncertainty in SASs. The Systematic Literature Research (SLR) by Hezavehi et al. [31] indicates that most of the available work deals with uncertainty treatment at runtime using SASs). They also point out that there is a gap in handling uncertainty at runtime. This is one of the gaps we want to close in this thesis as we focus on the investigation of uncertainty at design-time. However, related work dealing with uncertainties in SASs influences our ideas and concepts, as these provide valuable content. Other domains are Cyber-Pyhsical Systems (CPSs), the modelling of uncertain behaviours [48] or the representation of uncertainty in development processes [38].

The recently published work of Hezavehi et al. [32] and Troya et al. [72] again emphasize the actuality of uncertainty in software systems. In [32], the authors identify key findings based on the current perception of the research community on uncertainty. The work revealed that there is a "lack of systematic approached for managing uncertainty" [32]. In general, the study revealed that consensus in the community only exists with regard to primary uncertainty characteristics such as nature or the necessity to address uncertainty at design time and runtime. There is no consensus, for example, on whether SASs alone are sufficient to manage uncertainty in software systems. Whereas Hezavehi et al. [32] provide a recent overview from the community perspective, Troya et al. [72] present an extensive SLR on current uncertainty representations in software models. By doing that, the authors provide an overview on available approaches dealing with uncertainty and existing notations and formalisms to represent it. Further, they propose a categorization into different types. Especially, this work elaborates that "software models are still falling short for explicitly representing uncertainty and effectively dealing with it" [72].

One of the goals of this thesis is the categorization of ADDs based on their uncertainty character. Therefore, uncertainty-related categories are necessary. We extract possible categories and ideas for new categories from available uncertainty taxonomies, which are presented in the following. Walker et al. [73] define a "conceptual basis for the systematic treatment of uncertainty in model-based decision support activities" [73]. Further, the work provides our definition of uncertainty as being "any deviation from the unachievable ideal of completely deterministic knowledge of the relevant system" [73]. As main contribution, the authors propose a taxonomy based on the three dimensions *location*, *level* and *nature*. The taxonomy is supposed to be applicable to all model-based decision support. It further is the basis for other taxonomies such as the one proposes by Perez-Palacin and Mirandola [52] and Bures et al. [14].

Perez-Palacin and Mirandola [52] condense and enhance the work of Walker et al. [73] to create an uncertainty taxonomy which is more suitable for modeling software systems. The taxonomy consists of three dimensions, namely *location*, *nature* and *level*. Whereas *nature* and *level* are very similar to the same-named dimensions in [73], *location* is slightly adjusted to better fit the needs of software modelling. It describes uncertainties concerning the completeness and the accuracy of models, as well as the input parameters used to calibrate the models. An analogy to *Model-Driven Engineering* highlights the main focus of the taxonomy, which is the classification of uncertainties in software models.

Bures et al. [14] enhance the uncertainty taxonomy proposed by Perez-Palacin and Mirandola [52] to "better fit the needs of uncertainty in access control in Industry 4.0 Systems". In addition to *nature* and *level*, the authors propose the category *source of uncertainty*. In terms of content, this category largely corresponds to *location* as proposed by Perez-Palacin and Mirandola [52] and Walker et al. [73]. Further, they identified patterns per type of uncertainty that serve as adaption strategies for access control rules, when unexpected situations (induced by uncertainty) occur in the system. Yet, the taxonomy is elaborated to classify uncertainties in use-cases and scenarios in Industry 4.0 systems, where the dynamic adaption of access control rules is required. Although we require a taxonomy for a different purpose, this work serves as a basis and a source of ideas.

Hezavehi et al. [31] provide a taxonomy of uncertainty in SASs, based on a profound SLR. Given the nature of the studied system, the authors focus on uncertainty due to the "dynamicity and unpredictability of a variety of factors existing in software systems". However, their taxonomy is influenced by a number of other works such as [52, 26, 21, 73, 54], so that it includes important uncertainty-related categories including valuable options for each category. Furthermore, the study shows that of the 51 papers examined, only two explore uncertainty from the design time perspective, another 13 from both design and runtime, and 36 focus only on runtime. Although it's main purpose is the classification of uncertainties in SASs, we use it as basis for our classification process.

Ramirez et al. [54] propose a taxonomy to classify sources of uncertainties at requirements, design and execution time. The authors present a template to describe uncertainties and facilitate organization, including fields such as name, classification, context, impact or related sources. However, we do not want to classify sources of uncertainties but uncertainties themselves. Nevertheless, especially their taxonomy of uncertainty sources at the design level provides valuable information for our classification process. Cámara et al. [16] summarize the works proposed in [31, 54, 22] to provide both, a taxonomy for uncertainties and sources of uncertainties. They explain the sources with numerous examples, which are very helpful as a source of inspiration. Further, the authors provide multiple theories and techniques for uncertainty management, but unfortunately for SASs only. Given examples for management theories are fuzzy sets, possibility theory, probability theory or game theory. For each theory, they provide a pointer to an available technique implementing this theory. Lastly, the authors describe uncertainty representation techniques and self-adaptive models such as the *MAPE-K loop* and the *RAINBOW* framework.

Troya et al. [72] provide an uncertainty taxonomy based on an extensive SLR on uncertainty in software models. Their taxonomy is different to the one proposed by [73, 52, 31, 14]. Yet, the authors draw parallels between their proposed categories and those available. The authors introduce several types of uncertainty, such as *measurement uncertainty*, *design uncertainty* or *behaviour uncertainty*. For each, the authors also identify notations and formalisms to represent the various types of uncertainty. Due to the relevance and the scientific basis (SLR based on 123 paper) this work is another inspiration for our thesis.

Uncertainty management requires not only to detect uncertainties but to create processes and tools to mitigate uncertainties. Several authors propose various mitigation strategies for different categories of uncertainties. For instance, Perez-Palacin and Mirandola [52] provide mitigation strategies according to the *location*, whereas [32, 54, 24] provide a categorisation according to the *emerging time* of the uncertainties. Further, Cámara et al. [16] provide several theory-based techniques such as fuzzy sets, probability or game theory. Nevertheless, uncertainty mitigation is out of the scope of this thesis.

Architectural Design Decisions: Our goal is to use ADDs as the functional instrument to analyse the impact of uncertainty on the confidentiality of the system at design time. Therefore, a sound understanding is required. Bhat et al. [11] present a SLR with more than 200 papers from the last four decades focusing on the topic of architectural decision making, and especially a sound understanding of ADDs. Another SLR about design decision documentation is provided by Alexeeva et al. [3], which includes 96 papers from 2004 to 2015. The authors classify current approaches by the dimensions *goal*, *formalisation*, *context*, *extension*, *tool-support* and *evaluation*. Altogether, this indicates the importance of this topic in scientific research. Kruchten [43] presents an ontology for architectural design decision, and proposes to make them a first-class entity. The author identifies three general types of design decisions: Existence decisions, property decisions and executive decisions. For each of them, he provides further options and examples. Also, he identifies and describes possible relationships between design decisions. We use this ontology as basis for the categorization of ADDs.

The work of Jansen and Bosch [37] is an often-cited paper providing the definition of an software architecture as "the composition of a set of architectural design decisions". Further, it provides a comprehensive definition of an architectural design decision (ADD), including a well elaborated explanation of its elements. We use it as reference for our understanding of an ADD. Also, Jansen and Bosch [37] propose a model for ADDs as foundation for an approach called *Archium*, which enables to define and maintain the relationships between ADDs throughout the entire life-cycle of software systems.

Gerdes et al. [27] provide a tool called *Decision Buddy* to support the design decision process. Instead of uncertainty, they consider design constraints to be the most important factor in the design process. Consequently, uncertainty is not considered to any significant extent. Koziolek [41] presents a tool called *PerOpteryx*. It explores various design alternatives for several design decisions, called degrees of freedom, to obtain the most suitable candidates with regard to quality attributes such as cost or performance. Although *degrees of freedom* can, to some exten,t be described as uncertainty in relation to the design alternative to be chosen, an explicit modelling of uncertainties is not intended. In particular, the tool does take into account uncertainties that cannot be resolved by choosing a design alternative.

Ntentos et al. [50] elaborate a reusable ADD meta model and instantiate it for various design decisions in the Microservice domain. To that end, the authors identify model elements such as *category*, *decision*, *domain class*, *pattern* or *practice*. For each *category*, various coarse-grained design alternatives are represented by *decision* elements. Fine-grained decisions are represented by the elements *pattern* and *practice*. Each decision affects one ore multiple *domain classes*. Although the authors affirm that their model reduces the effort required to reduce the uncertainty in the design process, they do not consider uncertainty as first-class entity.

Uncertainty in Software Architectures: Architectural decision making problems under the influence of uncertainty can be investigated from both design-time and run-time [23]. The latter perspective is mostly covered by SASs, as described previously. For the design-time perspective, tools have been developed to guide the architect through the decision making process. For instance, Esfahani et al. [23, 22] present *GuideArch*, a tool to handle uncertainty in early software architectures. The authors "accept uncertainty as a natural component of architecting a software system" but the scope of uncertainty in this work is limited to "not knowing the exact impact of architectural alternatives on properties of interest". With *GuideArch*, software architects can incrementally make and refine ADDs to explore possible design alternatives. For each alternative, the tool calculates the expected quality properties based on fuzzy values. Features such as defining priorities, dependencies, conflicts and constraints between ADDs enable a realistic representation of the decision making process. Although this tool is able to guide an architect during the decision making process, it is not able to satisfy our definition of uncertainty, which goes beyond not knowing the exact impact of architectural alternatives on quality properties. Another tool to support the documentation and decision making process of recurring ADDs under uncertainty is presented by Lytra and Zdun [45]. Using fuzzy logic, expert architects can specify the expected, but still uncertain impact of each design alternative on quality attributes. This is stored as fuzzy models in a model repository. User architects can specify requirements, based on which an inference system infers appropriate design decision. Again, this tool only aims to guide the decision making process instead of analysing the impact of uncertainty on quality properties such as confidentiality.

Famelis and Chechik [24] present a tool-supported approach called *Design-Time Uncertainty Management (DeTUM)*. As a foundation, they introduce the *DeTUM* model which describes the stages of design-time uncertainty *articulation*, *deferral* and *resolution*. In the first stage, uncertainty is accepted and introduced. During the deferral stage, uncertainty remains stable whereas it is resolved in the resolution stage. Gathering more information enables to proceed to the second or the last stage. However, the authors emphasise that this can also uncover new uncertainties. In general, the approach enables software architect to identify and manage uncertainty during the design process, in order to be able to make decisions under uncertainty. The authors assume that for each decision, all alternatives must be known in beforehand. Uncertainty is therefore only taken into account to the extent that, at a given point in the design process, it is unclear which alternative is the best.

The position paper by Lupafya [44] is closest to our research questions. The author aims to develop a "conceptual framework that allows architects and developers to consider uncertainty in the context of software architectures". Second, he aims to realize "the concepts in the form of a workbench of tools for managing uncertainty at the architectural level." As a first step, the author provides a categorisation of *lack of knowledge* in software development, where his definition of uncertainty is a sub-category of involuntary, incomplete knowledge. He further clearly differentiates between uncertainty, inconsistency, inaccuracy and the absence of knowledge. Although this work has a large intersection with our work, (especially in the area of a conceptional framework for uncertainties), the author has not published any further progress so far.

4. Uncertainties in Software Architectures

In this chapter, we elaborate the relationship between uncertainty and confidentiality in software architectures (**RQ1.1**). In Section 4.1, we introduce our view on software architectures, Architectural Design Decisions (ADDs), and their role in the design process. This includes the presentation of the *Cone of Uncertainty*. Section 4.2 presents the general relationship between ADDs, *architecture elements*, *uncertainty* and *confidentiality* and how it is possible to derive uncertainties from existing ADD. We focus on ADDs with a foreseeable impact on confidentiality. In Section 4.3 we illustrate how to extract uncertainties and their impact from existing ADDs. This is complemented by an exemplary application to a collection of ADDs. This chapter is concluded by Section 4.4, which summarizes the main contributions with regard to **RQ1.1**.

4.1. Software Architecture, ADDs and the Cone of Uncertainty

Jansen and Bosch [37] describe software architectures as "the composition of a set of architectural design decisions" [37] (ADDs). Further, they provide a definition of ADD as being "a description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture" [37]. Based on that definition, the authors elaborate the importance of considering ADDs as first class entity in the software architecture. We adopt this view, also due to the general acceptance in the research community. In the course of the design process, software architects take many ADDs. Each decision taken thus contributes to the construction of the overall software architecture, or as per the view of Jansen and Bosch [37], the aggregate of all decisions *is* the architecture.

At the beginning of the design process, the degrees of freedom, i.e. the uncertainties, are at a very high level. This phenomenon is metaphorically described by the *cone of uncertainty* [46], and is represented in Figure 4.1. The original concept of the *cone of uncertainty* refers to the holistic software development process, including the requirements phase up to the completion of the software [46, p.31f]. To that end, we focus on the design process only. Nevertheless, the influence of uncertainty extends beyond the design process. For this reason, Figure 4.1 indicates that uncertainty is not completely eliminated at the end. In general, the cone illustrates that there is a lot of uncertainty at the beginning, since only the initial concept is made and no further design decisions have yet been made. In the course of the design process, architects acquire further knowledge and make a variety of decisions. Both are closely intertwined and ideally reduce the cone. This effect is visualised by the narrowing cone. In early design time, ADDs are often subjected to assumptions due to imprecise or unknown information. Since ADDs are not to be considered in isolation

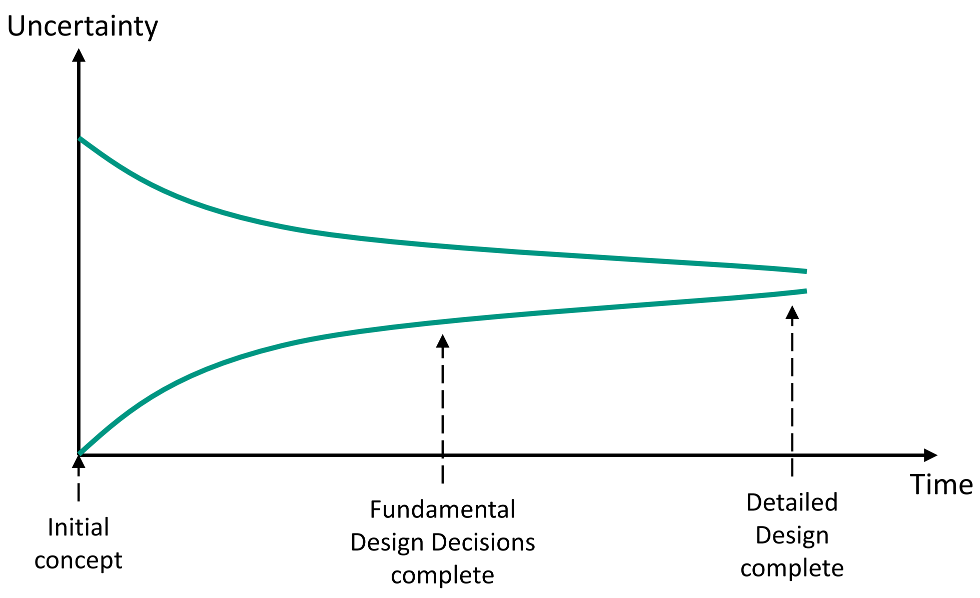


Figure 4.1.: The idealized Cone of Uncertainty [46]

but depend on others [17, 60, 27], assumptions often need to be made, in order to make progress in the design process. These assumptions may turn out to be wrong at a later stage, so that ADDs have to be reversed. This introduces further uncertainty and can thus enlarge the *cone of uncertainty* again [45].

4.2. ADDs, Architecture Elements and Uncertainties

The relationship between uncertainty, ADDs and architectural elements is illustrated in Figure 4.2. *Uncertainty*, ADD and *Architecture Element* are elements we consider as first-class entity. The other keywords are merely constructs that exist for a better understanding of the relation, but which are not modelled. The focus is therefore on the relationships between the key elements (indicated by the solid lines). The other relationships (dashed lines) are not considered in more detail. Whereas the impact of ADDs on confidentiality is usually known in advance, the impact of uncertainty on confidentiality is what we aim to analyse in this thesis. This is what the question mark is supposed to express. We identified the following relationships between *uncertainty*, *architecture element* and ADD:

ADDs are taken to reduce degrees of freedom within an architecture, i.e. to reduce the *cone of uncertainty*. That is the trivial part of the relationship because every decision that has not yet been taken induces uncertainty, or the other way around, each ADD resolves the underlying uncertainty in its choice. For instance, uncertainties about the kind of communication is resolved by taking the ADD that specifies the communication. Dealing with it is therefore relatively uncomplicated, as such uncertainties will be resolved by the end of the design time. Nevertheless, it is important to identify those uncertainties as they indicate which ADDs have not yet been made.

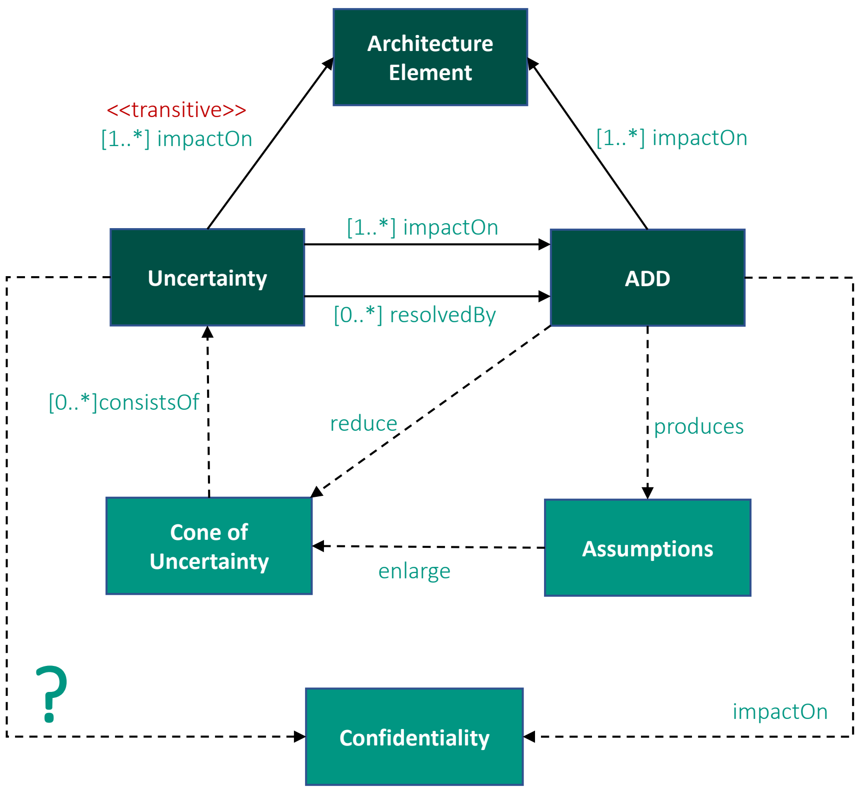


Figure 4.2.: The Relationship between ADDs, Uncertainty and Confidentiality

However, not all uncertainties can be resolved by taking an ADD (see multiplicity of the *resolvedBy* relation). These uncertainties are either epistemic or aleatory in nature (cf. Section 2.2), hence uncertainty prevails due to missing knowledge or due to the variability of the problem which needs to be described. In both cases, uncertainties have an impact on ADDs as software architects are forced to make assumptions if they take ADDs under uncertainty. These assumptions may turn out to be wrong at a later stage, so that ADDs have to be reversed. This introduces further uncertainty and can thus enlarge the *cone of uncertainty* again. Although some of these uncertainties can be eliminated during design time, others cannot. For example, uncertainty about the behaviour of an external component can be eliminated by knowledge acquisition during design time, whereas uncertainty about user behaviour can only be eliminated at runtime. However, both uncertainties have an impact on ADDs such as the choice of a component or the modelling of user interfaces which must be decided at design time. So there are two basic sources of uncertainties in the architectural design process: First, the lack of information, including ADDs that have not yet been made, missing requirements and missing knowledge in general. Second, assumptions that have to be made when making ADDs.

Yet the overarching goal is to assign uncertainties to concrete architectural elements. According to Jansen and Bosch [37], we can extract these relationships via the ADDs, since each ADD has an impact on elements of the architecture. Thus, we can transitively determine on which architectural elements an uncertainty has an influence. As motivated in Section 1.2, there are two types of impact: *direct* and *indirect*. First, uncertainties are assignable to concrete architecture elements of a certain type. This is what we describe as *direct* impact. Second, uncertainties propagate through the architecture and have a (less obvious) impact on other architectural elements. This is what we describe as *indirect* impact or *uncertainty propagation*.

4.3. Extracting Uncertainties

In the following, we discuss how uncertainties and their impact can be extracted from existing ADDs. In particular, we address the challenges and the knowledge required to do this, and suggest what knowledge can be reused. Since software architectures can be viewed as a "set of architectural design decisions" [37], we can identify uncertainties in architectures from existing ADDs. This allows for a more structured approach when extracting uncertainties. ADDs are therefore only a means to an end, on the one hand to extract uncertainties in general, and on the other hand to assign their impact and propagation effects in the architecture to concrete elements.

According to the previous section, each ADD that is not yet taken can be seen as uncertainty. This kind of uncertainty is resolvable by the respective ADD. As ADDs affect certain elements of the architecture, these uncertainties have a direct impact on those very elements. Therefore, software architects have to analyse the ADDs with regard to the elements they affect. This information is reusable for CBSAs because the underlying principles are the same for each architecture instance. ADDs are extracted from existing literature or software architecture design documents in order to examine where and how uncertainty affects them. However, it is not part of this thesis to provide a comprehensive guide on how to extract ADDs. This is covered by other scientific work such as [3].

Further, ADDs often depend on each other [17, 60, 27]. Thus, as soon as an ADD *A* depends on another ADD *B* that has not yet been taken, the associated uncertainty of the ADD *B* also affects the elements of ADD *A*. Consequently, this uncertainty has an indirect impact on the elements of ADD *A*. This information is also widely reusable, as dependencies between ADD are not domain specific. Uncertainties that do not correspond to an ADD that has not yet been made become apparent in the course of the design process by making assumptions. Every assumption exists because of an uncertainty that could not been eliminated at the time of the decision. Therefore, these uncertainties have a direct impact on the very elements that are in turn affected by the associated ADD. Especially the knowledge about possible assumptions is not trivial knowledge. However, these kinds of uncertainties need to be identified, as they are not necessarily apparent when making decisions. Especially here, it is of great advantage that this knowledge is collected and reused in a structured way.

Example: In the following, we show exemplarily how it is possible to identify uncertainties, determine to which architectural elements they can be assigned (i.e. where they have an direct impact) and specify where they have an indirect impact (via propagation). The identification of potential ADDs, possible design alternatives, the relationships between ADDs and especially the required assumptions are based on the expertise of the architect.

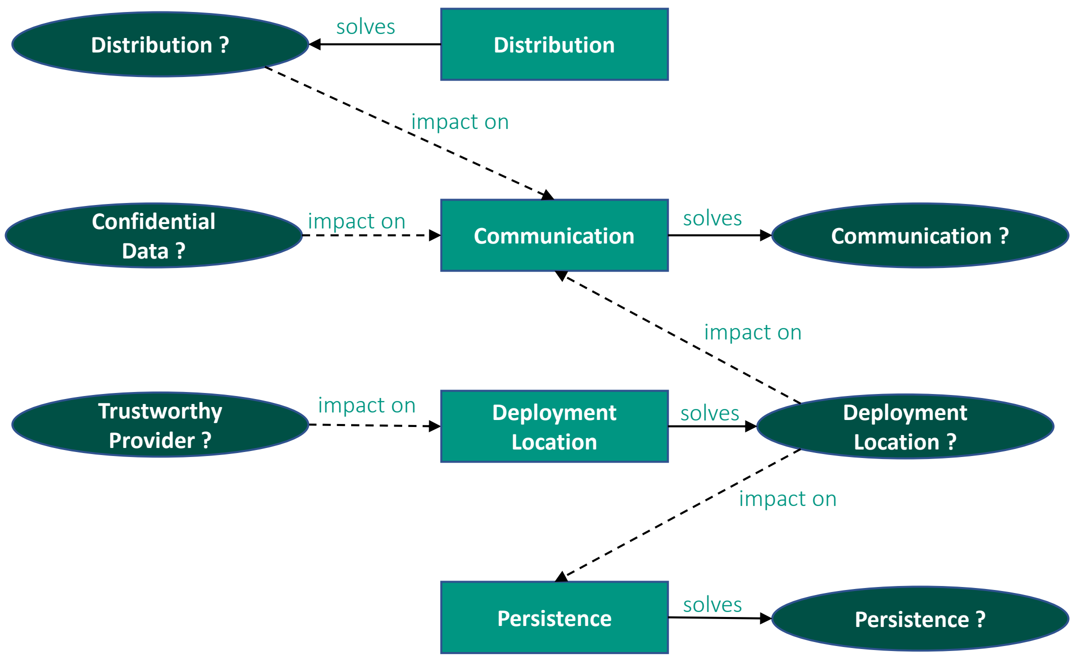


Figure 4.3.: Extract Uncertainties from ADDs - Example

Figure 4.3 illustrates four ADDs (the angular shapes) and six uncertainties (the elliptical shapes), including their relationships. The following ADDs are identified: Distribution decides whether a system is centralized or decentralised. Communication determines whether the communication between software components is encrypted or unencrypted. Deployment Location defines where the components are to be deployed, which can be either on-premise or using a external cloud provider. Persistence specifies whether data has to be stored encrypted or not. Each ADD affects certain element in CBSAs: Distribution affects the overall system structure. Communication affects the interfaces which are considered as first-class entities in CBSE. Deployment Location affects the resources such as servers, whereas Persistence affects the database components. Again, this collection of ADDs, including their design alternatives are for illustrative purpose only and do not represent a comprehensive system at all. When it comes to uncertainty extraction, software architects can first identify the rather trivial uncertainties which are the ones corresponding to each ADD. Therefore, the uncertainties Distribution, Communication, Deployment Location and Persistence are identified (elliptical shapes), including the *solves* relationships with the respective ADDs (angular shapes). As previously described, these uncertainties can be assigned to instances of the respective architecture element types. For example, the uncertainty Communication can be assigned to any communication element in an architecture where the type of communication is still uncertain.

In a second step, software architects identify which ADDs depend on each other. For instance, encrypted communication is necessary as soon as the system is designed in a decentralised manner. Consequently, uncertainty distribution also impacts the Communication (i.e. indirect impact). Similarly, when deploying in the cloud, it is necessary for services to communicate in encrypted form. In addition, the data must be stored in encrypted form. This is not the case with on-premise deployments. Hence, the uncertainty Deployment Location has an impact on the communication and the way data is persisted. The information gathered in this way thus shows which other elements can be indirectly influenced by uncertainties.

Lastly, software architects make assumptions when taking ADDs. At this point we assume for the time being that architects do not yet have any knowledge, so that they have to make assumptions about everything. For instance, communication can be unencrypted if the transmitted data is not confidential, regardless of the deployment location and distribution. The information whether or not transmitted data is confidential is not yet available. Of course, it can be obtained when the requirements have been specified. Nevertheless, this way of thinking helps us to identify further uncertainties and, above all, to investigate where these have an impact. Another example is the trustworthiness of a provider. If cloud deployment is chosen, architects must trust their handling of confidential data. This means, for example, that they do not give any third party access to the system. However, one can only assume that providers are trustworthy. This uncertainty remains for the future. Nevertheless, it can already be identified and analysed at the design time. For each uncertainty, the directly affected architectural elements are identified. Consequently, uncertainties due to assumptions can be assigned to the elements that are affected by the respective ADD.

Altogether, several steps are necessary. First, one needs to identify the ADDs and analyse which architectural elements they affect. Further, it is necessary to investigate the dependencies between ADDs in order to extract the indirect impact of uncertainties on architectural elements. Finally, software architects need to consider what assumptions they have made, explicitly or implicitly, whose discarding has an impact on the architecture. This step-by-step approach makes it possible to define a process with which software architects can derive uncertainties and their effects on architectural elements in a structured way. This is one of the goals of this thesis

4.4. Conclusion

In this chapter, we presented the relationship between uncertainty, ADDs, *architecture elements* and confidentiality (**RQ1.1**). In doing so, we have shown how taking ADDs can both reduce and increase the *cone of uncertainty*. Lastly, we discussed how software architects can use existing ADDs to derive uncertainties and their impact, which is exemplarily applied to a collection of ADD. In particular, we highlighted that a structured approach would be beneficial in this matter.

5. Categorizing Architecture-related Uncertainties

In this chapter, we present how to categorize uncertainties in software architectures with an impact on confidentiality **(RQ1.2)**. As described in Chapter 4, ADDs are a means to an end to extract uncertainties and to investigate their propagation effects in software architectures. Therefore, it is necessary to categorise not only the uncertainties, but also the ADDs in the context of which they occur. Based on existing taxonomies and other related work (cf. Chapter 3), we collect and assess available categories for both ADDs and uncertainties. Generally, such taxonomies are elaborated for a specific purpose. Therefore, it is necessary to assess and adapt them for our purpose, which is the categorization of uncertainties in CBSAs with an impact on the confidentiality of the overall system. So far, existing taxonomies and other related work do not aim to categorize uncertainties in software architectures, which is why a categorization template is necessary for this purpose. We first present and assess available ADD- and uncertainty related categories in Section 5.1. To finally answer **RQ1.2**, we propose our selection of ADD- and uncertainty related categories in Section 5.2.

5.1. Assessment of Available Categories

In Chapter 3, we presented related work with regard to ADDs and uncertainties in software systems. For both cases, we extract possible categories as follows: First, we collect and assess available taxonomies, as they provide categories explicitly. In a next step, we analyse related work with regard to similar categories, even if the authors do not explicitly employ the terms *category, dimension, type, option* or something similar. This helps us to substantiate already identified categories with further sources, as well as to extract further categories that have not yet been identified. If authors use different terms to describe the same category, we align the vocabulary to avoid confusion. Each category is further specialized and provides several options. A standardisation of the terms *category* and *option* should serve the purpose of better understanding. Therefore, we use only the terms *category* and *option*, regardless whether the authors use terms like *dimension* [73] or *characteristic* [21]. When assessing taxonomies and other sources of possible categories, it is absolutely necessary to consider the purpose and the scope of the respective work.

In the following, we provide a brief overview of the available sources and their scope: Both, Kruchten [43], and Jansen and Bosch [37] provide taxonomies to classify ADDs in software systems which is exactly what we intend to do. Walker et al. [73] lay the foundation for model-based uncertainty taxonomies and provide an interdisciplinary framework for systematic uncertainty analysis. The second taxonomy is presented by

Perez-Palacin and Mirandola [52]. This work is based in the taxonomy proposed by Walker et al. [73] but focuses more on describing uncertainties in the modeling of SASs. This taxonomy focuses on uncertainties in models with regard to structure and composition. As software architecture is usually described as model, this taxonomy provides valuable input for our classification. Esfahani and Malek [21] does not provide an uncertainty taxonomy but enumerates further possibilities for uncertainty categories in SASs. They propose the concept of the spectrum of uncertainty based on knowledge which we adapted in Section 2.2. Hezavehi et al. [31] present a taxonomy for uncertainties in SASs, that is based on a SLR and that includes the works presented previously. Bures et al. [14] adapt the taxonomy of Perez-Palacin and Mirandola [52] to be able to categorize uncertainty in access control in Industry 4.0 systems. The proximity of this work to ours allows for a high degree of reusability of proposed categories and options, although not on a one-to-one basis.

In another work, Hezavehi et al. [32] provide an overview of uncertainty in SASs from a community perspective. In contrast, Troya et al. [72] provide a recent SLR about uncertainty representation in software models. Both works are highly interesting since their actuality illustrates the current state of the art. The works of Ramirez et al. [54], Cámara et al. [16], and Famelis and Chechik [24] do not provide fully elaborated categories in form of a taxonomy, but they provide valuable input for the categorization process.

Current taxonomies and most of the related work with regard to uncertainty in software systems therefore mostly cover the area of SASs. Therefore, we cannot reuse proposed categories one by one as we want to describe and classify uncertainties in CBSAs. Categories created exclusively for SASs are omitted for the sake of clarity. Nevertheless, each work is to be assessed with regard to the suitability for our purpose. In the following, we provide a detailed description of available categories for both, categorizing ADDs (Section 5.1.1) and uncertainties (Section 5.1.2).

5.1.1. ADD-related Categories

In this section, we present available categories and options to categorize ADDs with regard to their architectural characteristics and impact on confidentiality. Table 5.1 provides an overview of the extracted categories including their options. The categories are mainly extracted from Jansen and Bosch [37] and Kruchten [43].

Category ADD Class categorizes ADDs in four major classes: *existence decisions*, *non-existence decisions*, *property decisions* and *executive decisions* [43]. *Existence decisions* are further refined into *structural decisions* and *behavioural decisions. Structural decisions* refer to the creation of elements such as subsystems, layers, partitions or components whereas *behavioural decisions* describe how those elements interact. Kruchten [43] refers to them as one class, although both classes represent two different concepts in CBSAs. In contrast, *Non-existence decisions* only express the opposite of the aforementioned *existence decisions*. In both cases, such decisions usually refer to specific elements. On the contrary, *property decisions* often cannot be related to specific element as they often express cross-cutting concerns. To that end, Kruchten [43] differentiates between positively expressed decisions,

Architectural Category	**Description**	**Options**	**Descriptions**	**Source**
ADD Class	Categorization of ADDs into classes	Existence decisions	Expresses the existence of elements and artefacts in the system (structural decisions) and how they interact (behavioural decisions)	[43]
		Non-existence decisions	Ban-Decisions: Opposite of existence decisions	[43]
		Property Decisions	Describe ADDs defining rules or constraints	[43, 37]
		Executive decisions	Affect matters of finance, methods, education, training, organization, technology and tools	[43]
Attributes	Relevant attributes for each ADD	Epitome	Short textual description of the ADD	[43]
		Rational	Justification of ADD	[43, 37]
		Scope	Scope in time, organization or in design	[43]
		State	Current status of the ADD	[43]
		History	Author and date when ADD was taken	[43]
		Cost	Estimated costs	[43]
		Risk	Risk due to uncertainty in problem domain or novelty of solution domain	[43]
		Additional Require-ments	Requirements that need to be adressed by additional ADDs	[37]
Relationship	Possible relationships between ADDs	Enables	Decision B enables A (but does not take it)	[43, 37]
		...	...	[43]

Table 5.1.: Overview of available architectural-related Categories including their Options

such as rules or guidelines, and negatively expressed decisions, such as constraints. Finally, *executive decisions* affect the business environment, development process, education and training, the organizations, and the choice of technology and tools. Kruchten [43] cites as a major difference to the previous classes that such decisions do not concern specific elements. As an example, he presents the technology decision that "The system is developed using J2EE" [43]. However, in times of Microservices, it is arguable whether or not such decisions can or cannot refer to, for instance, specific components [9]. Overall, this category is highly suitable for our own categorization.

Category Attributes are used to describe ADDs more precisely [43]. The options *epitome*, *rational*, *scope*, *state* and *history* are more suitable to describe ADDs in a management process which is not what we aim to do. Options *cost* and *risk* illustrate the need to choose ADDs conscientiously, as otherwise high economic damage can occur. Also, Kruchten [43] identifies various *relationships* between ADDs, which can be considered as stand-alone category. The following options are available: *constraints*, *excludes*, *enables*, *subsumes*, *conflicts with*, *overrides*, *compromise*, *is bound to*, *is an alternative to*, *is related to*. For the sake of clarity, the relations are not described in detail. Nevertheless, this category shows that ADDs must always be analysed in relation to each other. Hence, the definition of a category for relationships is inevitable. Jansen and Bosch [37] consider software architecture as set of ADDs making those decisions a first-class entity. They also identify options to describe ADDs. As with Kruchten [43], these include *rational* and *design constraints*. Further, they identify the option *additional requirements* which are requirements that need to be satisfied by other ADDs.

Category Relationship describes various relationships between ADDs. Both Kruchten [43], and Jansen and Bosch [37] present several possible relationships such as *enables, subsumes, override* or *conflicts with.*

5.1.2. Uncertainty-related Categories

This section provides an overview of available categories to describe characteristics of uncertainties from various point of views. Some approaches provide elaborated uncertainty taxonomies [73, 52, 14, 31], others enumerate possible categories in a less structured manner [21, 32, 72, 24]. We standardise the terminology as described previously. For each category/option, we argue whether or not it is suitable for our purpose or how it can inspire our categorization process. Table 5.2 provides an overview of the available categories.

Category Location is a widely recognised category for uncertainty and is described by [14, 31, 73, 52, 72]. Each of them provide a slightly different definition of this category, depending on the scope of the respective work. *Location* is either defined as category that describes where uncertainty manifests itself within the model [73, 52, 72] or within the entire system [31, 14]. The differences really become noticeable in the options which are presented in Table 5.3.

In the following, we describe all options in detail with regard to the different views of the respective authors. Several authors identify the *context* as a possible location for uncertainties: According to Walker et al. [73] the *context* represents the boundaries of the system to be modelled. This includes "uncertainty about the external economic, environmental, political, social, and technological situation that forms the context for the problem being examined" [73]. Perez-Palacin and Mirandola [52] refer to context uncertainty when there is uncertainty about the information to be modelled. In contrast, Hezavehi et al. [32] describe this option as the execution context, including humans that

Uncertainty Category	**Description**	**Source**
Location	Describes where uncertainty manifests itself within the system	[14, 31, 73, 52, 72]
Level	Describes how much is known about the uncertainty on a scale from deterministic knowledge to total ignorance	[73, 31, 52, 14]
Nature	Describes the character of uncertainty	[73, 52, 31, 14, 44, 72]
Manageability	Describes the manageability character of uncertainty with regard to its reducibility	[21, 73]
Emerging Time	Describes when uncertainty is acknowledged during the cycle of the system	[31, 54, 32]
Impact on non-functional Properties	Describes how uncertainty affects non-functional properties the system	[32]
Relationship	Relationship between uncertainties	[54, 32]
Source	Potential sources of uncertainty	[52, 21, 31, 44]

Table 5.2.: Overview of available uncertainty-related Categories

interact with or affect the system. Finally, Bures et al. [14] define uncertainty in the context as any uncertainty with regard to the system's context and the system's input data. Each of the definitions addresses several fundamentally different concepts. Overall, the definition of the category *context* is very overloaded as it includes several conceptionally different locations of uncertainties, as for instance uncertainties due to unknown system boundaries, unknown environmental circumstances, uncertainties in the system to be modelled as well as the uncertain behaviour of external actors [72].

Another possible location for uncertainty is described as *model structural.* Again, various different views exist. Walker et al. [73] mention uncertainty in the structure of the model itself, in the behaviour of the system and relationships between inputs and variables. The definition of Perez-Palacin and Mirandola [52] is mainly adopted from Walker et al. [73] and describes uncertainties in the accuracy of the structure of the model, i.e the elements and relationships that exist in a model. According to Hezavehi et al. [31], this location represents uncertainties due to various conceptional models, which represent the same system differently. Bures et al. [14] describe *model structural* as uncertainty within the design of the system, i.e. uncertainty related to the occurrence of components and their wiring. Lastly, Troya et al. [72] refer to uncertainties with regard to the existence of entities in a model, as well as possible design alternatives. Again, the explanations include several concepts of CBSAs such as input variables, the structure of a system, the behaviour of entities and their relationship. Each of these concepts represent valid locations of uncertainties, but they should be clearly structured and considered separately from each other.

Category: Location	
Option	**Descriptions**
Context	- Refers to the boundaries of the system to be modelled incl. external economic, environmental, political, social, and technological situation [73] - Refers to the boundaries of the meta-model [52] - Refers to system context and the system's input data [14] - Refers to execution context and humans interacting with it [31]
Model Structural	- Refers to the structure of the system, its behaviour and the relationships between elements [73] - Refers to the model elements and their relation [52] - Refers to the existence of entities [72] - Refers to possible design alternatives [72] - Refers to various conceptional models representing the same system differently [31] - Uncertainty about system components, their wiring and hardware resources [14]
Model Technical	- Uncertainty induced by software and hardware errors [73]
Inputs	- Uncertainty due to uncertain input types and values [73] - Uncertainty due to the actual value of inputs [52] - Uncertainties due to measurement deviations [72]
Model Parameters	- Uncertainty in data for model parameter calibration [73, 52]
Model outcome uncertainty	- Accumulated uncertainty induced by the uncertainties of all other locations [73]
System Behaviour	- Uncertainties in the actual behaviour of the system [14] - Uncertainties in the actual behaviour of the system including possible actions, the underlying motivation and point in time, and the actual parameters or the actions [72]
Belief Uncertainty	- Uncertainty about statements about the system or the environment [72]

Table 5.3.: Overview of available Options for the Uncertainty Category *Location*

Model Technical is based on [73] and describes uncertainty induced by software errors due to software bugs or design errors in algorithms, and hardware errors due to failures in technical equipment. It is questionable whether software errors could not be assigned directly to system components and hardware errors to the system environment, i.e. whether it is necessary to consider these locations separately.

Walker et al. [73], Troya et al. [72] and Perez-Palacin and Mirandola [52] identify the *Inputs* of the system as separate location where uncertainty can occur. According to Walker et al. [73], uncertainty prevails not only in the type of the inputs that cause system changes, but also in their possible values. Perez-Palacin and Mirandola [52] only mention the actual value of possible inputs. In contrast, Troya et al. [72], define this option as uncertainty due to measurement deviations. Whereas the first two definitions represent a useful conceptional location of uncertainties, the last rather describes a potential source of uncertainties.

Both, Walker et al. [73] and Perez-Palacin and Mirandola [52] identify *model parameters* as another possible location of uncertainties. In both cases, parameter uncertainty describes uncertainties in methods and data for model parameter calibration, which is not required for our approach. *Model outcome uncertainty* is the accumulated uncertainty induced by the uncertainties of all other categories, which propagate through the model and reflect uncertainties in model analyses outcome [73]. In a sense, this type of uncertainty is the one we try to analyse with our approach. Hence, it is not part of the categorization process, but rather the overall result.

Uncertainties in the *system behaviour* is another location which is already partly covered by the *model structural* definition of Walker et al. [73]. Yet, uncertainties in the *system behaviour* is explicitly defined by Bures et al. [14] and Troya et al. [72]. Bures et al. [14] provide a rather general definition of this location as they describe it as "uncertainty in the actual behaviour of the system" [14]. Troya et al. [72] expand this by uncertainties in the actual behaviour of the system including possible actions, the underlying motivation and point in time, and the actual parameters or the actions. Clearly, the definition of Troya et al. [72] also includes uncertainties in input parameters, which we consider as separate option. Uncertainties in system behaviour should therefore be explicitly considered in our categorisation process

Belief uncertainty is referred to as "second-order uncertainty" [72] and describes uncertainty about statements about the system or the environment made by belief agents. This uncertainty is not specifically assignable to the system, but rather to belief agents such as software architects. It is therefore rather less important for our categorisation.

Category Level describes how much is known about the uncertainty on a scale from deterministic knowledge to total ignorance. Although this category is defined in the same way by [73, 31, 52, 14], the underlying idea is very different. Whereas Walker et al. [73] and Hezavehi et al. [31] describe the different levels by how much knowledge lacks to achieve deterministic knowledge, Perez-Palacin and Mirandola [52] and Bures et al. [14] describe the levels by orders of ignorance. Table 5.4 illustrates this by separating the options into two areas, as indicated by the bold line. In the following, both concepts are explained in more detail.

Walker et al. [73] employ the terms *statistical uncertainty*, *scenario uncertainty*, *recognised ignorance* and *total ignorance. Statistical uncertainty* is any uncertainty that can be described with statistical means. In other words, one is aware of the uncertainty and capable to describe it adequately using statistical expression (e.g. failure rate of a CPU). *Scenario uncertainty* refers to plausible descriptions on how a system might develop in the future. The difference between statistical and scenario uncertainty is that the former can be expressed using stochastic terms whereas the latter involves choosing from a range of discrete possibilities, because too little information is available to adequately describe the outcomes statistically

Category: Level	
Option	**Descriptions**
Statistical Uncertainty	Any uncertainty that can be described with statistical means [73, 31]
Scenario Uncertainty	Refers to available descriptions (=scenarios) on how a system might develop in the future [73, 31]
Recognized Ignorance	Uncertainty is accepted but no mitigation strategy exist, i.e. cannot be described using statistical means or scenarios. Further divided in reducible and irreducible ignorance. [73]
Total Ignorance	One does not even now what is unknown [73]
0^{th} Order	Lack of uncertainty, i.e. knowledge [52, 14]
1^{st} Order	Lack of knowledge but awareness of uncertainty, i.e. known uncertainty [52, 14]
2^{nd} Order	Lack of knowledge and lack of awareness of uncertainty [52, 14], i.e. unknown uncertainty
3^{th} Order	Lack of process to find out what is unknown [52, 14]
4^{th} Order	Uncertainty about orders of uncertainty, i.e. meta-uncertainty [52]

Table 5.4.: Overview of available Options for the Uncertainty Category *Level*

The next level is *recognised ignorance* where uncertainty is accepted but cannot be described using statistical means or scenarios. The authors further divide *recognised ignorance* in *reducible ignorance* and *irreducible ignorance*, where *reducible ignorance* can be resolved by conducting further research and *irreducible ignorance* cannot. However, it is arguable if reducibility and irreducibility only applies to *recognized ignorance.* For instance, one can argue that user behaviour is irreducible in its nature but can be described using statistical means. This uncertainty characteristic therefore needs further attention and will be discussed in the course of this thesis.

Lastly, *total ignorance* refers to "not even know what we do not know" [73], hence the architects is not aware of uncertainties at this level. Especially the last level does not apply for our approach, as we only consider uncertainties that we know and in particular we know the location of. As already said, this classification is based on the fact that the gradual descent of levels requires a higher amount of knowledge to get closer to ideal determinism. More precisely, uncertainty that can be described by statistical means is closer to determinism than uncertainty that can only be handled by creating potential future scenarios, and so on. Hezavehi et al. [31] adopt only *statistical uncertainty* and *scenario uncertainty*, and neglect the remaining levels. Here it is questionable whether each uncertainty can be described by statistical means or by scenarios, which is why we consider the option *recognized ignorance* to be particularly important.

In contrast, Perez-Palacin and Mirandola [52] describe five levels of ignorance along the spectrum between deterministic knowledge (zeroth order) and total ignorance (fourth order) which is based on the work of Armour [4]. The *zeroth order* represents the lack of uncertainty, i.e. the idealistic presence of absolute knowledge. The *first order* of uncertainty describes that a subject lacks knowledge but is aware of that, which the authors refer to as known uncertainty. The *second order* of uncertainty describes that a subject lacks knowledge and awareness, which is described as unknown uncertainty. In this case, a subject does not know that a specific uncertainty exists but has adequate means to detect such uncertainties eventually. The *third order* represents the lack of process to move unknown uncertainty to known, which describe the circumstances that no means exist to detect such uncertainties. The *fourth order* represents the meta uncertainty and describes uncertainty about the order of uncertainty. Bures et al. [14] adopt this taxonomy except that the fourth level (meta-uncertainty) is removed due to impracticality.

For our approach, second, third and especially the fourth order do not provide any valuable benefit, as we can only analyse the impact of uncertainties that were identified beforehand, i.e. uncertainties at the first order. For uncertainties of the other levels, approaches exist that deal with the detection of previously unknown uncertainties. In the case of SASs, for example, monitoring is a suitable means of detecting unknown uncertainties in behaviour [26]. However, detecting unknown uncertainties is out of the scope of this thesis. Therefore, the categorisations proposed by [52, 14] is only partially suitable for our use case.

When comparing the two different approaches to describe the level of uncertainty, the following mapping can be applied: *statistical uncertainty, scenario uncertainty* an *recognized ignorance* can be mapped to the *first order* of uncertainty (i.e. known uncertainty) whereas *total ignorance* can be mapped to the *second order.* In a sense, the categorisation proposed by Walker et al. [73] is a more fine grained view on known uncertainties (*first order*), which is what we want to classify.

Category Nature is addressed by almost all papers that deal with uncertainty. As described in Section 2.2, nature refers to the essence of uncertainty and is divided into *epistemic* and *aleatory* [73, 52, 31, 14, 44, 72] (see Table 5.5). Epistemic uncertainty is caused by the imperfection or lack of knowledge. Therefore, it can be mitigated by gathering more information. In contrast, aleatory uncertainty is caused by the inherent variability of the nature or due to random events [28]. Consequently, this type of uncertainty cannot be reduced by acquiring more information. Some authors use the term *variability* instead of *aleatory* [73]. However, this is an interchangeable term in the literature for the same term [31]. *Nature* is the only category that is widely accepted and always defined similarly.

Category: Nature	
Option	**Descriptions**
Aleatory	Caused by the inherent variability of the nature or due to random events [73, 52, 31, 14, 44, 72]
Epistemic	Caused by the imperfection or lack of knowledge [73, 52, 31, 14, 44, 72]

Table 5.5.: Overview of available Options for the Uncertainty Category *Nature*

Category Manageability refers to the management of uncertainty which is proposed by Esfahani and Malek [21], and consists of the options *reducible* and *irreducible* (see Table 5.6). The first option describes uncertainty that can be reduced (i.e. managed) by acquiring additional knowledge. Option two expresses that if something is fundamentally unknowable, the associated uncertainty is irreducible. Although we already justified in Section 2.2 that aleatory uncertainty is irreducible in its nature, and epistemic is reducible, we consider this category to be important as it describes a different view on uncertainties. However, it is questionable whether uncertainty can always be binarily assigned to one of the two options, or whether there should not be others. Walker et al. [73] use the terms *reducible* and *irreducible* to divide their uncertainty level *recognized ignorance* (cf. category *Level*). This demonstrates that a certain connection exists between this category, and the categories *nature* and *level*. Although it is clear that aleatory uncertainty can never be completely reduced, it can, be at least partially reduced by describing the uncertainty with statical means if possible.

Category: Manageability	
Option	**Descriptions**
Reducible	Uncertainty can be reduced by acquiring more knowledge
Irreducible	Uncertainty cannot be reduced if something is inherently unknowable

Table 5.6.: Overview of available Options for the Uncertainty Category *Reducibility*

Category Emerging Time is a category that describes when uncertainty is acknowledged or appears during the cycle of a system. It is describes by several works such as [54, 72, 31, 32]. The options corresponds to known software development phases, also specified by the Rational Unified Process (RUP) [42]. Some of the available options are presented in Table 5.7. Although we consider uncertainties at architectural level, thus at the design time, it will be necessary to address other phases as well. Hence, this category provides valuable input for our categorization process.

Category: Emerging Time	
Option	**Descriptions**
Requirements Time	Uncertainty arises during requirements time [54, 72]
Design Time	Uncertainty arises during design time [54, 32, 72, 31]
Verification	Uncertainty arises during the verification of the models [72]
Testing	Uncertainty arises during testing [32, 72]
Implementation	Uncertainty arises during implementation of the system [72]
Run Time	Uncertainty arises during run time [54, 32, 72, 31, 52]

Table 5.7.: Overview of available Options for the Uncertainty Category *Emerging Time*

Category Impact describes the way uncertainties affect non- functional properties of a given system. To that end, Hezavehi et al. [32] provide the options *performance*, *resource consumption* and *safety* (see Table 5.8). As already mentioned in Chapter 1, current approaches do not investigate the impact of uncertainty on the non-functional property confidentiality which is what we aim to do in this thesis. Nevertheless, this proposed category indicates the importance to consider the impact of uncertainties on non-functional properties.

Category: Impact	
Option	**Descriptions**
Performance	Impact of uncertainty on the performance of a system [32]
Resource Consumption	Impact of uncertainty on the performance of a system [32]
Safety	Impact of uncertainty on the safety of a system[32]

Table 5.8.: Overview of available Options for the Uncertainty Category *Impact*

Category Relationship describes how uncertainties are related to each other. Ramirez et al. [54] propose a taxonomy of sources of uncertainties and how they are related to each other. This realisation contributes significantly to the fact that we cannot consider uncertainties in isolation, but must always see them in relation to each other. As presented in Table 5.9, Ramirez et al. [54] define the two options *directed* and *related*. The first option describes a directed relationship between two uncertainties such as an uncertainty causing another uncertainty. The latter describes that two uncertainties are somehow related to each other, without further specifying the characteristics of the relationship. For our categorization process, it is therefore necessary to elaborate and define potential relationships between uncertainties.

Category: Relationship	
Option	**Descriptions**
Directed	Directed relationship between uncertainties [54]
Related	Unspecified relationship between uncertainties [54]

Table 5.9.: Overview of available Options for the Uncertainty Category *Relationship*

Category Source is another frequently mentioned category of uncertainties. Table 5.10 provides an incomplete overview of available sources, such as uncertainty due to *abstraction* or due to *changing resources*. A comprehensive overview of available sources is provided by Cámara et al. [16]. However, we consider sources of uncertainties as entities that we want to categorise with our categories. Therefore, the sources of uncertainties are rather to be considered as the input of our categorization process and not as category.

Category: Source	
Option	**Descriptions**
Abstraction	Uncertainty due to omitting details for simplicity reasons [21]
Incompleteness	Uncertainty due to (known) missing elements of the model [52]
Human in the loop	Uncertainty due to the unpredictability of humans [21]
Changing resources	Uncertainty due to the dynamicity of the system's resources
Sensing	Uncertainty due to inaccurate or broken sensor values
...	...

Table 5.10.: Overview of available Options for the Uncertainty Category *Source* (Excerpt)

Conclusion: The literature search shows that some categories occur more often (cf. *nature*, *location*, *level*) and some do not (cf. *emerging time*). This is due to the fact that most of the works build on each other, but adapt the proposed categories for their needs. It is always important to note for which purpose an uncertainty categorisation is made. Examples are more general categorisations [73], categorisations for SASs [51] or for access control [14]. Further, the definitions of the categories and especially those of the specified options vary considerably in some cases. For this reason, it is important that we specify the categories and options of our approach precisely, in order to avoid confusion with existing categories of other approaches.

5.2. Proposed Categories

In the following, we present the ADD- and uncertainty related categories. For each category, we provide the following information: First, a detailed definition of category and options, including examples where possible. Second, the foundations on which the categories/options are based. Third, the expected added benefit of the categories and their options. The first part contributes substantially to the understanding of the respective category/option. The second part aims to substantiate the right of the category/options to exist. The third part explains how users can process and use the information provided by the category/option. In the end, the proposed categories should help to assess uncertainties and their impact on confidentiality in CBSAs.

5.2.1. ADD-related Categories

In this section, we present our selection of categories and options used to categorize ADDs in CBSAs. An overview is provided in Table 5.11. In the following, each ADD-related category is explained in detail.

Architectural Category	**Description**	**Options**	**Descriptions**
ADD Class	Categorization of ADDs into classes	Structural Decision	Expresses the existence of elements and artefacts in the system
		Behavioural Decision	Expresses how elements interact
		Property Decision	Describes ADDs defining rules or constraints
		Executive decision	Affects matters of finance, methods, education, training, organization, technology and tools
Amount of Alternatives	Extensibility of known sets of design alternatives	Closed Set	Final set of alternatives
		Open Set	More alternatives are added in the future
Probability of Revisability	Expresses whether it is more or less likely that a taken ADD will be revised in the future.	Likely	Higher probability that a decision will be revised later
		Unlikely	Probability that decision will be revised later is low
Possibility of Revisability	Expresses whether it is actually possible to revise an ADD after it was taken	Yes	ADD can be revised in the future
		No	ADD can't be (practically) revised in the future
Costs of Revision	Expresses the expected costs of a potential revision	High	Expected costs to revise an ADD are high
		Low	Expected costs to revise an ADD are high

Table 5.11.: Overview of the proposed ADD-related Categories including their Options

Category ADD Class enables to categorize ADDs into four major classes, where each option corresponds to a respective class. ADDs can be categorized as *structural decisions*, *behavioural decisions*, *property decisions* and *executive decisions*. *Structural decisions* express the existence (or if expressed negatively the non-existence) of elements of CBSAs. This includes the creation of subsystems, layers, partitions, components and other first-class entities such as interfaces. *Behavioural decisions* concern the relationship between elements, i.e. how elements interact and communicate with each other. *Property decisions* express design guidelines and rules (if expressed positively), and constraints (if expressed negatively). The main difference between *structural* and *behavioural decisions* on the one hand, and *property decisions* on the other, is that the latter cannot be related to specific types of elements, as it affects several element types and therefore has a rather overarching influence. Finally, *executive decisions* affect matters of finance, methods within the development process, education and training of people, overall organizational decisions, and especially technology and tools.

This category and options are mainly adopted from the taxonomy proposed by Kruchten [43]. Only the former category *existence decisions* is divided into the two categories *structural decisions*, *behavioural decisions*, which are two different concepts in CBSAs according to Reussner et al. [56]. The former *non-existence decisions* are part of the respective *structural* and *behavioural decisions*, since the categorisation would be unnecessarily inflated. It has to be noted that this categorisation does not always have to be unambiguous. For instance given the ADD: *'Components must communicate via facades'* which, on the one hand, leads to the creation of the structural elements Facade and, on the other hand, describes how components communicate with each other. Thus, this ADD can be categorised as both, *structural* and *behavioural* decision. Although this ADD affects the communication of components in general, it is not categorised as *property decision*, as it affects specific types of elements.

With this category, architects should be able to better understand the basic idea of an ADD, in order to better assess possible effects on confidentiality. For example, when a *behavioural decision* is made, an architect should investigate whether the communication/interaction needs special protection in terms of confidentiality breaches.

Category Amount of Alternative is based on the fact that ADDs come with an amount of possible alternatives from which architects need to select one at a certain point during the design time. The basic idea of this category refers to the extensibility of known sets of design alternatives [41]. *closed set* refers to the ADDs where all possible alternatives are known at the time of the decision. For instance, when deciding on certain design or communication patterns, architects choose from a well-known and closed set of existing patterns. Another example is the selection of access control mechanisms, where architects choose one of the available mechanisms such as Attribute-based Access Control (ABAC) or Role-based Access Control (RBAC). One can argue that new patterns or mechanisms, and consequently further design alternatives will be developed in the future, but by the time the architect makes a decision each currently available design alternative is known. *Open set* refers to the situation where some alternatives are known at the time an architects makes a decision, but other alternatives are likely to emerge in the course of the software development process, e.g. through the acquisition of more knowledge. For example, architects have to select a component for a specific purpose from a set of previously available components. However, it is likely that more alternative components will become available, as architects gain more knowledge about other software vendors offering such components. The main difference is that with *closed set* it is assumed that all design alternatives are known, whereas with *open set* it is deliberately said that further possibilities will be added. Although we speak of sets, we do not restrict ourselves to finite or infinite sets in the mathematical sense. For example, although the alternatives for calibrating a sensor value are potentially infinite (since real numbers could be used), in our sense they are known beforehand and thus can be described as *closed set*. The edge case of a decision without previously known alternatives can be assigned to the *open set*, since further alternatives must be added in the course of the software design.

The idea of extending known set of design alternatives is based on the work of Koziolek [41]. In addition, this category is inspired by Esfahani et al. [22] who describe that a software architecture "results from selecting a viable alternative for each and every decision". According to the authors, each alternative can have a different impact on the software architecture's non-functional properties. In doing so, they explain that it is not always possible to precisely describe the impact of known alternatives on these properties, which induces uncertainty. We transfer this concept to uncertainties in the design process, i.e. that not only the possible impacts of the alternatives are unknown, but also the possible alternatives themselves.

With regard to the classification of an ADD, its underlying uncertainty and the impact on confidentiality, this category should help as follows: For each alternative of a *closed set*, the impact on confidentiality can be assessed or at least estimated [22]. Subsequently, the best alternative can be chosen and thus the underlying uncertainty can be at eliminated. In the case of an *open set*, it is still possible to compare the currently available alternatives and select one of them. However, the underlying uncertainty cannot be completely eliminated, as possible further alternatives can lead to the revision of the ADD. In general, it can be assumed that ADDs with a *closed set* are more manageable than those with an *open set.*

Category Probability of Revisability defines whether it is more or less likely that a taken ADD will be revised in the future. The options *likely* and *unlikely* describe the directions of the continuous spectrum of the possibility of a revision. *Likely* is intended to suggest that an ADD is likely to be changed whereas *unlikely* indicates, that a revision will probably not be made. For instance, fundamental decisions about the system structure, the distribution or authentication are less likely to be revised. In contrast, decisions regarding the system environment such as the deployment location are more likely to be revised, due to unpredictable changes in the environment.

This category is inspired by architectural change impact analyses such as the one proposed by Rostami et al. [57]. ADDs made during the design process can be revised due to, for example, incorrect assumptions or changed requirements [57]. Nevertheless, decisions have to be made to drive the design process ahead, even if this is based on assumptions. At this point, we do not consider a division into discrete value ranges or percentage points to be purposeful. Such a determination is often very dependent on the application area and can often only be determined very imprecisely.

For software architects, it would be helpful to know which decisions are more or less likely of being revised. Decisions that probably will not be revised in the future can be taken without further consideration. However, if a revision is likely, architects should look more closely at the impact of possible revisions.

Category Possibility of Revisability is based on the fact that in some cases ADDs cannot be revised at all, or not with reasonable effort and cost. This circumstance is expressed by this category with the respective options *yes* and *no* for revisability and non-revisability. For instance, decisions about the fundamental structure of the system cannot be revised without reasonable effort and costs. It can be argued that in theory any decision can be revised by rewriting the entire software. However, the aim of this category is to define whether an ADD can be practically revised or not.

Again, this category is inspired by architectural change impact analyses and should be seen in combination with the category *probability of revisability*. For software architects, information about the possibility of revisability is important as decisions that cannot be revised must be treated with caution. If such a decision is influenced by too many uncertainties due to assumptions at any point in the design process, it should not be made yet. Beforehand, as many uncertainties as possible should be eliminated through, for example, further knowledge acquisition.

Category Costs of Revision describes the required costs to redesign parts of the software architecture due to a revised ADD. But some decisions are "cheaper" than others when it comes to their revision. Therefore, we introduce this category with the option *high* and *low* to express the expected costs of a potential revision. For instance, adding new access rights will probably be cheaper than changing a component, as components need to be implemented or purchased, whereas new access rights can be added by a system administrator in a few minutes.

Likewise, this category is based on the insights of the two previous categories. The rationale and benefits for software architects are similar. ADDs involving high costs of change should be decided later when uncertainty is present. In other words, uncertainties that could lead to high costs when the accompanying ADD is revised, should be given higher priority. A quantification by means of specific amounts of money or hours of work is also not expedient here, as a precise statement on this is part of an entire branch of research [57].

5.2.2. Uncertainty-related Categories

This sections presents the categories/options to categorize uncertainties in CBSAs. They are explained in detail in the course of this section. An overview is provided in Table 5.12.

Uncertainty Category	Description	Options	Descriptions
Location	Describes where uncertainty manifests itself within the architecture	System Structure	Refers to the structure of components, their wiring and communication
		System Behaviour	Refers to the behaviour of the systems and its components
		System Environment	Refers to the system's context, including hardware resources and the external situation
		Input	Refers to the inputs provided by external actors

Type	Describes how much is known about the uncertainty	Statistical Uncertainty	Information available to describe uncertainty with statistical means
		Scenario Uncertainty	Known descriptions available on how a system might develop in the future
		Recognized Ignorance	Uncertainty accepted but no mitigation strategy exists, i.e. it cannot be described using statistical means or scenarios
Nature	Describes the essence of uncertainty	Aleatory	Caused by the inherent variability of the nature or due to random events
		Epistemic	Caused by the imperfection or lack of knowledge
Architectural Element Type	Architectural elements to which an uncertainty can be assigned	Commu-nication	Uncertainty assignable to the communication
		Hardware Resource	Uncertainty assignable to hardware resources
		Component	Uncertainty assignable to components
		Interface	Uncertainty assignable to interfaces
		Usage Behaviour	Uncertainty assignable to elements describing the behaviour
Manageability	Describes if more knowledge or other appropriate means can reduce the uncertainty	Fully Reducible	Uncertainty fully reducible
		Partial Reducible	Uncertainty at least partially reducible
		Irreducible	Uncertainty won't be reduced
Impact on Confidentiality	Describes the impact on confidentiality if uncertainty persists	Direct	Direct impact on confidentiality
		Indirect	Impact, only in conjunction with other factors
		None	No impact on confidentiality

Severity of the Impact	Describes the severity of the impact on confidentiality if uncertainty persists	High	Loss of confidentiality expected
		Low	Confidentiality cannot be guaranteed any more
		None	No loss of confidentiality expected
Resolvable by ADD	Describes whether uncertainty can (partially) be resolved by taking an ADD	Yes (+ reference)	Uncertainty can be resolved by taking an ADD
		No	Uncertainty is not resolvable by taking an ADD
Resolution Time	Describes time at which uncertainty is resolved at the latest	Requirements Time	As soon as requirements are clarified
		Design Time	As soon as design is finished
		Deployment Time	As soon as system is deployed for the first time
		Runtime	As soon as knowledge gained from ongoing operations is available
		Never	No resolution possible
Root Cause	Describes the root cause for the uncertainty	Assumption	Uncertainty due to assumption
		Missing Information	Uncertainty due to missing knowledge

Table 5.12.: Overview of proposed uncertainty-related Categories including their Options

Category Location describes where uncertainty manifests itself within the architecture. Here, we propose the following options: *system structure*, *system behaviour*, *system environment* and *input*. To create the *system structure*, software architects compose components via their provided and required interfaces. This process is referred to as wiring and determines which instantiated components communicate with each other.

Option *system structure* thus describes uncertainties due to the design of the system including the choice of components, their composition into composite components, their wiring via provided and required interfaces, the structure of the interfaces, as well as the data that is provided and required. As the composition of components can be continued up to system level (the system is considered as the top composite component), *system structure* also covers uncertainties with regard to the system interfaces. In the early stages of system design, for example, it is still uncertain whether the software architects choose an internally developed component, or another component that is conceptually the same but provided by a third-party vendor. In that case, the internal structure and the interface design might be unknown. However, this option only refers to the structure and not to the behaviour of components. This is covered by the option described hereafter.

System behaviour refers to uncertainties due to the actual behaviour of the system, respectively its components. This includes uncertainties about data that is processed and especially stored by components. The internal behaviour of components may be unknown if, for example, it has not yet been defined, or components are only available as black boxes. In addition, this option includes uncertainties due to software errors that may cause the system to behave differently than expected.

System environment describes uncertainties in the actual context where the system is executed. This includes in particular uncertainties about hardware resources and the allocation of system components to those resources. However, this does not include the structure of these resources, such as their communication. Further, it includes uncertainties due to external economic, environmental, political, social, and the technological situation. Examples are uncertainties due to volatile deployment locations, changing cloud providers or other unexpected technological changes.

Input describes uncertainties due to the type of inputs and their actual values provided by external actors. Uncertainties arise from the unpredictability of humans or from a lack of knowledge about external systems that interact with the system under consideration. While system behaviour describes uncertainties within the system, input describes uncertainties outside the system.

Category *location* is based on available locations as presented in Section 5.1.2. Each option discussed is based on the findings of previous uncertainty taxonomies, as well as other related work. However, we adapted the available options to better fit the needs of CBSAs, and to align the locations with the models of the PCM (see Section 2.5). *System structure* is mainly adopted from option *model structural* as presented by [73, 52, 31, 72, 14] and *model technical* [73]. However, we explicitly exclude uncertainties regarding the behaviour of the system [73] and the hardware resources [73, 14]. The focus is on the existence of entities [72], their relation [52] and possible design alternatives on system level [31, 72]. This separation is in line with the PCM [56], as our definition of *system structure* roughly corresponds to the *Assembly Model.* In this way, we hope to achieve a better mapping from possible locations of uncertainties to the widely used and often validated software architecture views of the PCM.

System behaviour describes the behavioural-related parts of *model structural* as described by Walker et al. [73], but is mainly based on the definition provided by Bures et al. [14]. Again, we do no consider the *system behaviour* as part of the *system structure*, as they describe two fundamentally different concepts of CBSAs. As with the previous category, this option can be roughly mapped to the *Component Model* of the PCM.

System environment is based on the respective definition of Bures et al. [14] but also includes further context-related locations as proposed by Walker et al. [73] and Hezavehi et al. [31]. Nevertheless, the focus is on the execution context and excludes uncertainties due to humans as added by [31, 14]. With regard to the PCM, the *system environment* can be mapped to the specification of the hardware resources which are used in the *Allocation Model.* Due to its focus, the PCM concentrates on the definition of the resource environment only. Nevertheless, we consider uncertainties due to external economic, environmental, political, social and technical situations as part of the context which is in line with the findings as presented in Section 5.1.2. Therefore, those aspects are added to our definition of the *system environment.*

Input combines the user related parts of the option *context* as proposed by [73, 14, 31], as well as the option *input* as defined by [73, 52, 72]. In the PCM, the user input is modelled in the *Usage Model.* The majority of the locations identified in Section 5.1.2 can be found in our category. Nevertheless, we adapt most of them to better fit the needs of locating uncertainties in CBSAs with a focus on PCM. Only options *model outcome uncertainty* and *belief uncertainty* are removed completely. Regarding *model outcome uncertainty*, we already explained that this type of uncertainty is the one we try to analyse with our approach and should be considered as the overall result. *Belief uncertainty* is excluded, as it rather describes the uncertainties within the software architects' statements. This type of uncertainty is what we try to reduce by providing a comprehensive description of each category and location in the course of this thesis.

In general, software architects should be able to represent uncertainties as first-class entities within software architectures. This category provides several possible locations that mostly map to specific PCM models. That enables software architects to categorise given uncertainties and already roughly assign them to the respective PCM models, thus to locations within software architectures. In consequence, they can better specify where which uncertainty has an impact, which is one of the goals of this thesis.

Category Type illustrates how much knowledge lacks to achieve the ideal, but usually unachievable deterministic knowledge. We therefore employ a gradual scale, the extremes of which are the unattainable determinism at one end, and complete ignorance at the other. The three options *statistical uncertainty*, *scenario uncertainty* and *recognised ignorance* divide that scale into three areas where *statistical uncertainty* is closer to determinism than *scenario uncertainty*, which is in turn closer to determinism than *recognised ignorance*. *Statistical uncertainty* is any uncertainty that can be adequately described with statistical means such as the failure rate of hardware devices or the expected number of user accesses. *Scenario uncertainty* refers to any uncertainty that can be described by providing scenarios such as one of the possible access control methods or known deployment locations. In those cases, there is not enough information available to describe the uncertainty with statistical means, but only to create possible future scenarios about how the uncertainty could be resolved. *Recognised ignorance* describes the circumstance that uncertainty is accepted but can be neither described with statistical means, nor with possible scenarios. A reason for this is, for example, that too little knowledge is available to form suitable scenarios. For instance, software architects still have little knowledge about the internal behaviour of available components, so that they cannot create valid scenarios for comparison purposes. In summary, describing uncertainty by statistical means requires more knowledge about the prevailing uncertainty compared to creating potential scenarios. Likewise, being able to create potential future scenarios is closer to determinism than just being aware that uncertainty prevails.

This category is based on the *level* definition of Walker et al. [73]. As already explained, *statistical uncertainty*, *scenario uncertainty* and *recognized ignorance* map to the *known uncertainties*, i.e. the first level of the orders of ignorance according to Perez-Palacin and Mirandola [52]. Thus they represent a more fine grained view on *known uncertainties*, which is exactly what we want to analyse in more detail. Anything beyond the first level, i.e. achieving awareness and providing processes to achieve awareness, is not part of this thesis. This is due to the simple fact that once we use our template to categorise uncertainties, we are already at the first level because we know them. Hence, our use case does not require the remaining levels proposed by Perez-Palacin and Mirandola [52]. Although Walker et al. [73] use the term "level" for this category, we refrain from using this term to not be confused with the widely accepted *level* definition of Perez-Palacin and Mirandola [52]. Further, we only use a subset of the levels defined by Walker et al. [73]. In particular, we omit the option *total ignorance* for the same reason we omit the remaining levels of Perez-Palacin and Mirandola [52]. Moreover, Walker et al. [73] further divide *recognized ignorance* into *reducible ignorance* and *irreducible ignorance*. In our case, the reducibility character of uncertainties is covered by a separate category which is in line with the findings provided by [21].

A classification of existing uncertainties into these three options can help software architects as follows: *Statistical uncertainties* can be handled well due to statistical significance. Software architects can statistically describe which occurrence is most likely. In terms of *scenario uncertainty*, software architects could simulate existing future outcomes. Finally, for *recognized ignorance* architects are at least aware that uncertainty prevails, so that they can use adequate means to reduce uncertainty in a focused way.

Category Nature refers to the essence of uncertainty and is divided into *epistemic* and *aleatory*. Epistemic uncertainty is caused by the imperfection or lack of knowledge. Therefore, more knowledge can (but not necessarily) lead to the resolution of uncertainty. If, for example, the internal behaviour of a component is unknown, this uncertainty can be mitigated by gathering information about its internals. In contrast, aleatory uncertainty is caused by the inherent variability of the nature or due to random events. For instance, the failure of components, although statistically describable, is absolutely random and therefore aleatory. Consequently, aleatory uncertainty cannot be reduced by acquiring more information.

As indicated in Section 5.1.2, *Nature* is widely accepted by current researchers [73, 52, 31, 14, 44, 72]. We also share this view on the uncertainties, which is why we adopt the category without any changes. For software architects, differentiating between aleatory and epistemic uncertainty is useful as it allows to decide whether the acquisition of further knowledge is appropriate.

Category Architectural Element Type describes to which architectural element types uncertainties can be assigned. The available options are *communication*, *hardware resource*, *component*, *interface* and *usage behaviour*. Due to cross-cutting concerns, uncertainties may affect multiple element types which is why this category is not exclusive.

The idea of this category is based on the principle of CBSAs, where each of the options refer to first-class entities, so that it is possible to exactly specify to which elements uncertainties can be assigned. ADLs generally allow a more precise description of software architectures, which is why it is possible to specify exactly where uncertainties can manifest themselves in the architecture. In CBSAs not only *components* and *hardware resources* are first-class entities, but also *interfaces* and the *communication* [56]. Further, ADLs such as the PCM also provide the possibility to model the *usage behaviour*, which enables the representation of usage behaviour as first-class entity, too. Therefore, uncertainties due to usage behaviour can be assigned to the respective architectural elements that represent the usage behaviour. The difference to category *location* is the degree of abstraction. While category *Location* conceptually describes where uncertainty manifests itself, this category describes the specific elements to which uncertainties can be assigned. For example, uncertainty about the internal structure and behaviour of a component represents two different concepts according to category *location*. However, from a purely technical point of view, both are assigned to a specific component. In order to conceptually understand and specifically define where uncertainty manifests itself, both categories are necessary.

With regard to the analysis of the impact of uncertainties, this category enables a mapping of uncertainties to architectural elements. With this information, software architects can understand where uncertainties may have an potential impact within the software architecture.

Category Manageability refers to the possible management of uncertainty. To manage uncertainty, software architects can either acquire more knowledge or apply other methods, such as statistics or fuzzy methods [22]. As options, we propose *fully reducible*, *partial reducible* and *irreducible*. *Fully reducible* describes uncertainty that can be reduced, i.e. eliminated, by acquiring additional knowledge or by applying other methods. *Irreducible* describes that if something is fundamentally unknowable and cannot be managed with adequate methods, the associated uncertainty is irreducible. In this case, no further knowledge and no method is able to reduce the uncertainty and thus manage it. It should be noted that more (and potentially infinite) knowledge does not necessarily lead to the complete elimination of uncertainty. Moreover, circumstances that are described by statistical means are always only an expected approximation of the facts to be described. In those cases, uncertainty is neither fully, nor irreducible. This is what we refer to as *partial reducible*. On the one hand, for example, uncertainty about authentication mechanisms to be used can be fully reduced if the associated ADD is taken. On the other hand, the number of expected user accesses can be described by studies and statistics and therefore made manageable. However, the associated uncertainty is not fully reducible due to the unpredictability of human behaviour, and is therefore only partially reducible. The uncertainty of whether a system can handle all possible inputs, for example, is irreducible, since no knowledge and no procedure can check an infinite input space. Even a supposedly high test coverage cannot reduce this due to the inherent nature of the problem.

Manageability in terms of *reducibility* and *irreducibility* is proposed by [21, 73]. Reducibility, however, does not describe whether the uncertainty can be reduced completely or only partially. Yet, we consider this information very important for a more accurate assessment of the manageability of uncertainties. Further, the authors only define the extent to which knowledge can contribute to reducing uncertainties or not. Other methods, such as the description of uncertain circumstances through statistics, are not taken into account. However, statistical descriptions of inherently uncertain things are often used, especially in computer science, for example when modelling user behaviour, failure rates of hardware components or in queuing theory. Those uncertainties are still inherently uncertain, but from a management perspective they are easier to handle than uncertainties that are not describable at all.

Knowing to what extent uncertainties can be made manageable is essential for software architects. This information is crucial for uncertainty mitigation, as it allows to identify whether or not an existing uncertainty can be at least partially reduced. But irreducible uncertainties should by no means be neglected. This information is also valuable, as the impact of such uncertainties must be closely observed.

Category Impact On Confidentiality is used to asses whether the persistence of uncertainty hast either *direct*, *indirect* or no (=*none*) impact on confidentiality. The first option applies, if uncertainty has *direct* impact on the confidentiality, regardless of the presence and/or absence of other uncertainties and already taken ADDs. Uncertainty about whether confidential data is being transmitted, for example, has a direct impact on the confidentiality of a system. Without this knowledge, no valid statement can be made. We speak of indirect impact, if it depends on other factors whether an uncertainty has an impact on the confidentiality of a system or not. Such factors are missing knowledge or ADDs that have not yet been made, basically other uncertainties. Looking, for example, at the uncertainty about the deployment location. There is only an impact on confidentiality when user-related data is processed or stored there [28]. The uncertainty about the deployment location is thus linked to the uncertainty about the processing of user-related data. In the case of *none*, an uncertainty has no effect on the confidentiality of the system, whether it is resolved or not. Other quality attributes are not in the scope of this thesis.

The impact of uncertainty on non-functional properties such as performance, resource consumption and safety is motivated by Ramirez et al. [54]. This motivates that it is necessary to assess what impact uncertainties might have on the confidentiality of a system, should they prevail. Further, the distinction between direct and indirect is based on our realisation that many statements about a possible impact are often context-sensitive. Although this distinction is not based on any work found, we consider it important with regard to the benefits for software architects.

The persistence of uncertainties has implications for the predictive validity of a system's confidentiality. Software architects therefore need to know whether a specific uncertainty has or has no impact on the confidentiality. However, research shows that the impact of an uncertainty can often only be determined as a function of other factors (i.e. with *indirect* impact). Yet, knowing which uncertainties have *direct* impact is essential as these are the uncertainties that architects need to consider in any case.

Category Severity of the Impact refers to the significance of the impact on confidentiality, if uncertainty persists. To that end, we introduce the options *high*, *low* and *none* to provide a basic quantification scheme. *High* indicates that a loss in confidentiality is expected, if the uncertainty prevails. For instance, not knowing the internals of a component probably results in a loss of confidentiality. As described previously, this depends on the information whether or not a component receives user-related data in the first place. This category always refers to the worst case so that it is not necessary to include dependencies on other uncertainties. *Low* refers to uncertainties where the expected loss in confidentiality is small. Nevertheless, confidentiality cannot be guaranteed any more. *None* refers to uncertainties that do not have any impact on the confidentiality of the system.

The basis of this category is the Common Vulnerability Scoring System (CVSS) [20] which introduces impact metrics to assess the severity of software vulnerabilities, including a metric to measure the impact on confidentiality. This metric also suggests the three options *high*, *low* and *none*. Yet, their definitions need to be adapted to our needs.

The goal is to eliminate uncertainty completely. However, software architects should be able to assess which uncertainties have a high or low impact. This allows a targeted allocation of the available but limited resources when mitigating these uncertainties. This category therefore primarily serves software architects as a ranking, i.e. to assess which uncertainties should be dealt with first.

Category Resolvable by ADD expresses if the respective uncertainty can be (at least partially) eliminated by taking an ADD or not. Therefore, we propose the option *yes* and *no* where *yes* provides a reference to a description of the respective ADD. As many possibilities exist to describe and document ADD, we do not further specify how this reference must look like. Examples of *yes* and *no* are as follows: The uncertainty about the *communication type* is removed by the choice of the ADD that determines the communication type. Hence, the uncertainty is resolvable by a specific ADD. In contrast, the uncertainty about the trustworthiness of a provider cannot be resolved by an ADD. The statement whether an uncertainty can be completely or only partially eliminated by an ADD is outsourced to category *manageability*.

This category is based on the architectural-related category *relationship*, as presented in Section 5.1.1 which states that ADDs cannot be considered in isolation, but always in relation to each other [43, 17, 60, 27]. Since uncertainties can represent ADDs that have not yet been taken (c.f. Chapter 4), the concept of *relationship* can be applied to the relationship between uncertainties and ADDs. In this case, the specific type of relationship is referred to as *resolvable by*.

In relation to an uncertainty impact analysis, this category contains valuable information. It clearly shows what action a software architect must take to resolve an uncertainty, specifically what ADD to take. Non-resolvable uncertainties are marked, so that software architects can use appropriate resources in a targeted manner to reduce an uncertainty as far as possible.

Category Resolution Time refers to the time in the software development process when uncertainty is resolved at the latest. To that end, we propose the following options: *requirements time*, *design time*, *deployment time*, *runtime* and *never*. *Requirements time* describes that uncertainties are resolved at the latest when the requirements are clarified. For instance, the uncertainty whether or not data is confidential is to be defined during the requirements time. *Design time* refers to a resolution as soon as the design is finished. Main representatives are uncertainties that can be resolved by taking a respective ADD, such as the uncertainty about the used database system. *Deployment time* refers to the time when the system is deployed for the first time. A good example is the uncertainty about the deployment location. Although being able to specify the deployment location earlier, the point in time when this uncertainty is resolved at latest, is the *deployment time*. In fact, a decision made beforehand only relates to assumptions that architects have made. In contrast, uncertainties that require knowledge gained from ongoing operations are categorized with *runtime*. This could be uncertainties about the actual behaviour of humans. *Never* refers to uncertainties that won't be resolved. An example of this is the trustworthiness of an external provider, which can never be fully confirmed despite agreements.

Resolution time is mainly inspired by the available category *emerging time* which is proposed by [54, 72, 31, 52]. However, the basic idea of *emerging time* is to specify when uncertainty is acknowledged or appears. In this thesis, though, we aim to investigate the impact of uncertainties at architectural level, thus during design time. Therefore, we cannot investigate uncertainties that are acknowledged afterwards. We are interested in the uncertainties that are known at design time, but which we are well aware might not be resolved at design time. The reason why we do not define the earliest possible resolution time is that otherwise all uncertainties could be resolve by specifying a requirement during *requirements time*.

For software architects, this category is important because it makes statements about when they can "forget" about uncertainties. Uncertainties that are resolved at the latest at requirements and design time must be taken into account during the design process. Uncertainties that could only be resolved at a later time must be managed by architects by other means, such as introducing self-adaptive capabilities.

Category Root Cause describes the two possible root causes for uncertainty from the design time perspective. Either, uncertainty prevails due to *assumptions* or due to *missing information. Assumptions* describe everything we have to predict about the future in order to make ADDs. One example is the uncertainty about the suitability of access control mechanisms. It is possible to decide on a mechanism, but its suitability can only be assumed. In contrast, *missing information* refers to everything we should know by now but do not know yet. Uncertainty about the behaviour of a component, for example, is based on knowledge that does not yet exist.

This category is based on Figure 4.2 which describes the relationship of ADDs, assumptions and uncertainties: On the one hand, missing information leads to not being able to make a decision. A decision that has not yet been taken implies uncertainty. On the other hand, it is sometimes necessary to make assumptions in order to take an ADD. This induces further uncertainty, as assumptions may turn out to be wrong. Both, missing information and assumptions cause uncertainty. Yet this category is not to be interchanged with the category *nature* and *manageability*. In particular, uncertainties categorised as *missing information* are not necessarily *epistemic* or *reducible*. For example, not knowing the natural variability of a *GPS* sensor is missing information of an aleatory nature which can only partially reduced.

This category provides another view on uncertainty which has more of a temporal character, and is detached from *nature* and *manageability*. In particular, it helps software architects to understand how uncertainties and ADDs are related. *Assumptions* tend to have an impact in the future, as assumptions cannot be reduced before they are disproved. In contrast, *missing information* tends to result from the past. As already mentioned, the point of view is always design time.

5.3. Conclusion

In this chapter, we assessed existing ADD- and uncertainty-related categories from available taxonomies and other literature. The identification of implicitly suggested categories is based on our ability to examine continuous text for possible categories. Despite this limitation, we extracted three ADD-related categories including 13 options, and eight uncertainty-related categories including 37 options. For each category and option, we assessed their (un-)suitability for our categorization approach. Based on that, we proposed our selection of ADD- and uncertainty related categories (**RQ1.2**). To that end, we provided a detailed definition of both, categories and options including some examples. In addition, we have outlined the foundations of each category and option, as well as their expected benefit when categorizing uncertainties in software architectures.

6. Uncertainty Template and Application

This chapter presents our *uncertainty template* which enables software-architects to derive and categorize types of uncertainties based on existing ADDs, and to investigate their possible propagation effects within Component Based Software Architectures (CBSAs). It is based on the findings of Chapter 4 and Chapter 5, namely the relationship between ADDs and uncertainties, and the categories to categorize them. Unlike other works in this research area such as Perez-Palacin and Mirandola [52] and Hezavehi et al. [32], we speak of a template rather than a taxonomy. The reason for this is that a taxonomy aims at the systematic and holistic classification of a domain of interest, which is not our goal. We rather want to provide a *tool* with which software architects can better derive and and asses the potential impact of uncertainties in software architectures.

In addition, we illustrate how the template enables software architects to systematically identify types of uncertainties based on existing ADDs. Also, we demonstrate how software architects can systematically derive on which ADDs, and consequently on which types of architectural elements the uncertainties can have an impact. It has to be noted that a filled-in template is not bound to a specific architecture, hence can be reused for several software architectures within one domain of interest. In addition, we demonstrate how software architects can use the filled template to instantiate actual uncertainties, i.e. to annotate a specific software architecture with uncertainties. We further propose our concept of the propagation of uncertainties and illustrate how the filled-in template supports the analysis of uncertainty propagation. Lastly, we illustrate how software architects could use the template to assess the severity of impact of actual uncertainties within CBSAs.

The structure of the chapter is as follows: Section 6.1 provides a running example of a CBSA, as well as a collection of ADDs with which the structure and usage of the *uncertainty template* are illustrated. **RQ1.3** asks how the categories from the previous chapter can be illustrated. For this, we propose our *uncertainty template* whose overall structure is outlined in Section 6.2. Section 6.3 presents an approach on how to systematically derive possible uncertainty types and their potential impact on architectural element types. In this section, we also apply the approach for illustration purposes. Section 6.4 explains how software architects can use the *uncertainty template* to instantiate actual uncertainties and assign them to elements of an existing software architecture. Again, the approach is illustrated using the running example mentioned before. Section 6.5 introduces the concept of *uncertainty propagation.* Section 6.6 illustrates how combinations of statements can help to assess the severity of the impact of uncertainties more precisely. Altogether, this contributes to the derivation and assessment of uncertainties and their impact in software architectures (**RQ1.4**). The chapter is concluded by Section 6.7.

6.1. Running Example and Collection of ADDs

Figure 6.1 displays a small, illustrative CBSA to demonstrate our proposed approach. It shows two services Component A and Component B. Component A has a provided interface which is required by the requiring interface of Component B. The communication between both interfaces is represented as first-class entity. Further, Component A is deployed on Server 1, whereas Component B is deployed on Server 2. Lastly, a human shaped pictogram indicates that external actors interact with the system.

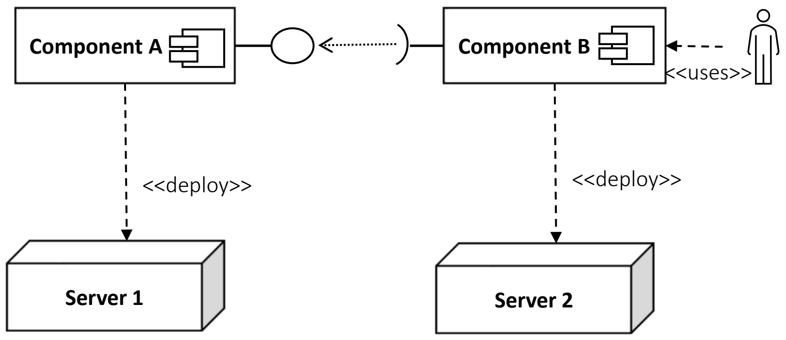


Figure 6.1.: Software Architecture - Running Example

In previous works, software architects could have identified the following ADDs. It should be noted that the ADDs presented have no claim to completeness or correctness, but are for presentation purposes only. We claim that the presented ADDs are generally applicable and not bound to the architecture illustrated in Figure 6.1.

Deployment Location (A1) Software components can be deployed to various deployment locations, such as on-premise and cloud servers. The type and actual location of the server (i.e. country, continent) can be relevant, as data protection regulations might prohibit the processing or storage of confidential data for certain locations.

Communication Type (A2) Software components need to communicate to each other when connected via provided and required interfaces to offer services. Thereby, communication can be encrypted using various different encryption mechanisms. Encryption is necessary when confidential data, such as user relevant data, is transmitted and when the communication is via publicly accessible networks, i.e. the internet.

Persistence of User Data (A3) Components offer services for which they require to process user data. In some cases, however, user data needs to be persisted which is also subjected to data protection regulations.

Choice of Component (A4) Software architects might develop components in house or buy them from third party vendors. Therefore, they need to choose from various different components. As users may be entering private data, such as credit card details, it can be crucial to know how the components function internally to determine how they process this data.

6.2. Structure of the Uncertainty Template

In Chapter 4, we elaborate the following relationships between ADDs, architecture elements and uncertainties: Uncertainties are either resolved by, and/or have an impact on ADDs. Moreover, ADDs have an impact on architectural element types. In this way, the impact of uncertainties on architectural elements and thus the overall software architecture can be determined transitively by using the respective ADDs as a means to an end. This relationship is once again depicted in Figure 6.2, which is a simplification of Figure 4.2. It shows the (*uncertainty template*-) relevant elements only, namely *architecture element*, *uncertainty* and ADD. Further, the multiplicities are slightly adjusted compared to Figure 4.2 due to practical reasons which is further explained in the course of this section. Whereas the overall structure of the *uncertainty template* is based on the previously described relationships, the actual possible values, i.e. the categories and their options, result from the proposed uncertainty and ADD-related categories defined in Chapter 5.

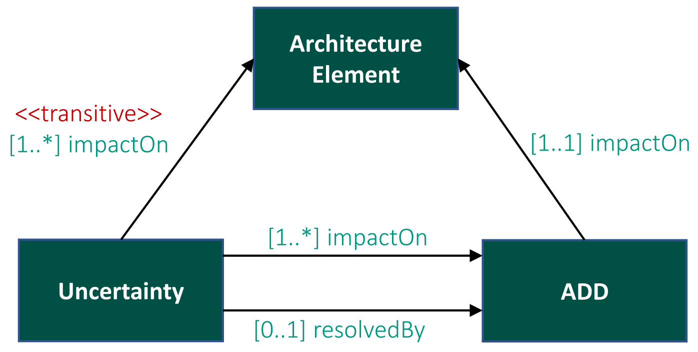


Figure 6.2.: The Relationship between ADDs and Uncertainty (based on Figure 4.2)

The structure of the *uncertainty template* is outlined by Table 6.1. It already includes some exemplary ADDs (ADD1 - ADD3) and uncertainties (U1 - U5). The two main headers *ADD-related Categories* and *Uncertainty-related categories* coarsely divide the *uncertainty template* in two areas, where the first area is for categorising the respective ADDs, and the second area is for categorising the respective uncertainty types. For the sake of clarity, we omit most of the categories. The entire uncertainty template is available in our data set [10]. The ADD-related categories correspond to the ones proposed in Section 5.2.1, whereas the uncertainty-related categories correspond to the ones proposed in Section 5.2.2. Each identified ADD exists once and has a name (column ADD), an unique identifier (column ID) and is categorised according to the ADD-related categories (column ADD Class to Cost of Revision). One to several uncertainty types can be assigned to each ADD, so that the resulting cell structure always indicates which uncertainty types are related to which ADDs. We refer to a row representing one ADD and the uncertainty types related to that ADD as *multi row*. As explained in Chapter 4, each ADD not yet taken induces uncertainty. For this reason and as an invariant, the first uncertainty type in a multi row always corresponds to the respective ADD, representing the *resolvedBy* relationship. The remaining uncertainties in a multi row indicate the uncertainty-related *impactOn* relationship, as those uncertainty types might have an impact on the respective ADD.

				ADD-related Categories			*Uncertainty-related Categories*					
ADD	**ID**		**Uncertainty Type**	**ADD Class**	**...**	**Costs of Revision**	**Location**	**...**	**Architectural Element Type**	**...**	**Resolvable by ADD**	**Root Cause**
ADD 1	A1	U1	A1 decided?	Structural	...	High	System Structure	...	Component	...	Yes (A1)	Missing Information
ADD 2	A2	U2	A2 decided?	Behavioural	...	Low	System Behaviour	...	Hardware Resource	...	Yes (A2)	Assumption
		U1	A1 decided?				System Structure	...	Component	...	Yes (A1)	Missing Information
		U4	Uncert. 4				Input	...	Usage Behaviour	...	No	Assumption
		...	...				...	...	...	...	...	...
ADD 3	A3	U3	A3 decided?	Executive	...	High		...	Communication	...	Yes (A3)	Missing Information
		U5	Uncert. 5					...	Component	...	No	Assumption
...	...	...	...	...	...	...	...	...	...	...	...	...

Table 6.1.: Exemplary Version of the Uncertainty Template (Excerpt)

As previously mentioned, we slightly adjust the multiplicities in Figure 6.2 compared to its initial Figure 4.2. More precisely, the *impactOn* relationship between ADDs and *Architecture Element* is changed from *one-to-many* to *one-to-one*, and the *resolvedBy* relationship between uncertainty and ADD from *none-to-many* to *none-to-one*. Both changes serve exclusively to improve the clarity of the *uncertainty template* and have no effect on its expressiveness. The first change is based on the idea that an ADD represented by a multi row should only have an effect on a single architecture element. This makes it more obvious on which architectural element type the respective ADD, but also the associated uncertainty types have an impact. However, if an ADD actually has an effect on several architectural element types, this can be simulated simply by inserting them several times into the template by using different identifiers. This even gives us the advantage of a more fine-granular view of a potential impact. The latter adjustment is based on the same reasoning, as it should always be clear which ADD resolves an uncertainty type. In the case an uncertainty type can be resolved by several ADDs, it is again possible to insert it several times by using different identifiers.

In the schematic *uncertainty template* (see Table 6.1), U1 is resolved by ADD1, U2 by ADD2 and U3 by ADD3. Further, U1 has an impact on ADD2. Besides that, the uncertainty types U4 and U5 are not resolvable by any ADD but have an impact on ADD2 and ADD3 respectively. Uncertainty types, unlike ADDs, can therefore occur more than once. Nevertheless, an uncertainty type that appears several times is always uniquely identified (column ID), has the same name (column Uncertainty Type) and categorized in exactly the same way (columns Location to Root Cause). The architectural-related *impactOn* relationship (between an ADD and the architectural element) is implicitly represented by the uncertainty-related category *Architectural Element Type* of each first uncertainty type of a multi row. In Table 6.1, for example, ADD1 has an effect on components, which is why option *Component* is selected for U1. Although this presentation induces information duplication (an uncertainty type can occur several times), it has the advantage that it is evident for each ADD which uncertainty types have an influence on it and how they are categorised. Furthermore, the implicitly defined *impactOn* relationships seem complicated at a first glance. However, we explain in the next section why this is useful and how it helps to systematically derive uncertainty types and their possible impact on architectural element types.

6.3. Uncertainty Type Derivation

In this section, we provide a systematic approach on how software architects can derive uncertainty types and their potential impact on software architectures from existing ADDs. The assessment of the severity of the impact of actual uncertainties is not part of this section but is covered in the course of this chapter. In general, uncertainty types can be identified in other ways, too. For instance, experienced software architects can collect uncertainties in workshops or from existing software projects. The approach using the *uncertainty template* is therefore only one possibility. Further, it has to be said that collecting potential ADDs is not part of this thesis, as this is already covered by numerous other related works such as [63, 3]. We present our uncertainty type derivation approach in Section 6.3.1 and conduct it in Section 6.3.2.

6.3.1. Uncertainty Type Derivation Approach

Figure 6.3 illustrates our approach to systematically extract uncertainty types from existing ADDs, enhance the template by uncertainty types which are not resolvable by any ADD, and transitively derive the possible impact of uncertainty types on other architectural element types. It is mainly inspired by the process to extract uncertainties in Section 4.3. In the following we briefly explain each of the steps:

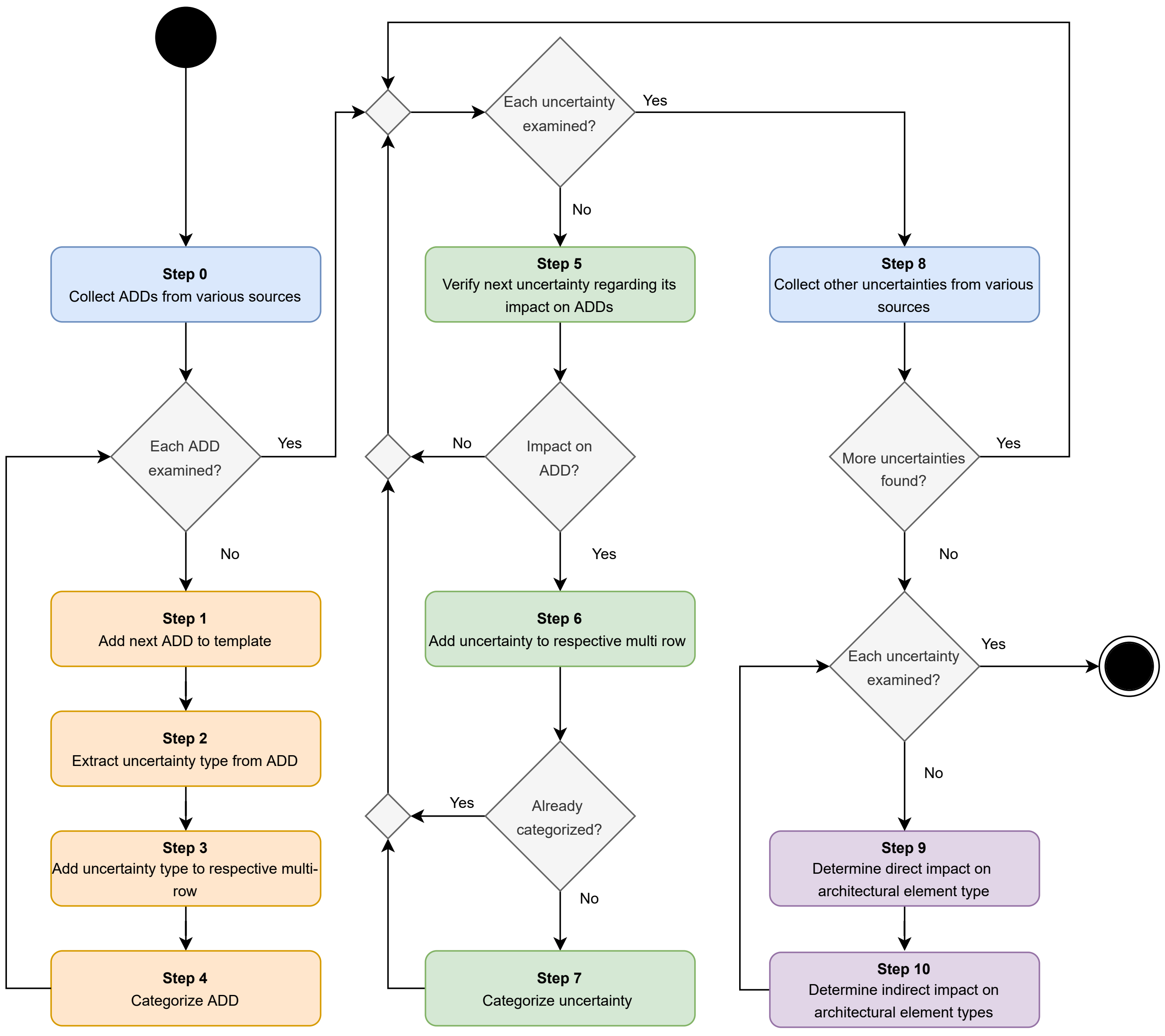

Step 0
Collect ADDs from various sources
Each ADD examined?
Yes
No
Step 1
Add next ADD to template
Step 2
Extract uncertainty type from ADD
Step 3
Add uncertainty type to respective multi-row
Step 4
Categorize ADD
Each uncertainty examined?
Yes
No
Step 5
Verify next uncertainty regarding its impact on ADDs
Impact on ADD?
No
Yes
Step 6
Add uncertainty to respective multi row
Already categorized?
Yes
No
Step 7
Categorize uncertainty
Step 8
Collect other uncertainties from various sources
More uncertainties found?
Yes
No
Each uncertainty examined?
Yes
No
Step 9
Determine direct impact on architectural element type
Step 10
Determine indirect impact on architectural element types

Figure 6.3.: Activity Diagram describing the Uncertainty Type Derivation Approach

In *Step 0*, ADDs are to be collected from various sources. As previously described, this should be rather considered as preliminary work and is therefore not considered to be part of the actual approach. For each of the collected/available ADDs, the subsequent steps are performed. Given a certain collection of ADDs, software architects start with *Step 1*. First, software architects need to add an ADD to the template. Therefore they need to create a multi row and assign a unique identifier, for instance a consecutive number preceded by the letter *A*. As described in Chapter 4, each ADD can be considered as uncertainty if not yet decided. Therefore, software architects can extract an uncertainty type from an ADD by simply re-formulating it as a question (*Step 2*). In *Step 3*, the extracted uncertainty type needs to be added to the first row of the multi row, represented by the respective ADD. Similar to the previous step, software architects need to assign an unique identifier, for instance a consecutive number preceded by the letter *U*. In *Step 4*, the ADDs need to be categorized according to the definition of the architectural-related categories (cf. Section 5.2.1). As indicated by the colour coding, *Step 1* to *Step 4* are repeated for each ADD. Consequently, each ADD has been "transformed" into an uncertainty. In the following, those uncertainties are to be examined.

In *Step 5*, software architects identify for each added uncertainty type on which other ADDs they could have an impact. For the uncertainty types that correspond to the respective ADDs (result of *Step 2*), this question is linked back to the relationship or dependency between ADDs. This question corresponds to a separate field of research and has already been addressed among others by [50, 27, 17, 3, 45]. If an uncertainty type to be verified results from another source (c.f. *Step 8*), software architects must also investigate on which existing ADDs each uncertainty type can have an impact. An impact exists, for example, if the actual choice of an ADD depends on another still uncertain ADD. Accordingly, the ADD can be made either when the uncertainty is resolved or on the assumption of how the uncertainty might be resolved. If an uncertainty type has an impact on an ADD, it must be added to the associated multi row (*Step 6*). If not, *Step 6* and *Step 7* are skipped and one must proceed with the next uncertainty, if available. Again, an uncertainty type that is added several times (i.e. an uncertainty type that affects several ADDs) has the same identifier. If the uncertainty has not yet been categorised, the software architects must do that in *Step 7*. Otherwise, this step can be skipped because the information already exists and only needs to be copied. In case the uncertainty corresponds to an ADD (result of *Step 2*), the two uncertainty-related categories *Architectural Element Type* and *Resolvable by ADD* are already defined. In the first case, the selected option corresponds to the type of architectural elements on which the ADD has an impact. The latter is only a reference to the identifier of the accompanying ADD. As described in Chapter 4, there are also uncertainty types which, on the one hand, cannot be resolved by taking an ADD, but which have an impact on it. *Step 5* to *Step 7* need to be executed for each of them.

In *Step 8* further uncertainties must be collected once for a domain of interest. Similar to *Step 0*, this task requires software architects to extract other uncertainties from sources such as previous (similar) software projects, design decision documents or workshops. Finding assumptions that have been made during the design process or missing information can help identify uncertainties. After collecting and classifying ADDs and uncertainty types, it is now to conclude on which (further) architectural element types uncertainties can have an impact.

In Section 4.3, we differentiate between *direct* and *indirect* impact. Uncertainties have a direct impact on an architectural element, if they can be assigned to it. If the uncertainty assigned to one element has an impact on another, we speak of indirect impact. This information is implicitly provided on type level by a filled-in template and can be extracted as follows: The direct impact can be determined immediately via the category *Architectural Element Type* (*Step 9*). To determine possible indirect impacts of an uncertainty type, we must identify in which multi rows it occurs (Step 10). For each multi row, the affected architecture element type is given by the first uncertainty type, as it represents the accompanying ADD. Uncertainty types in multi rows can therefore have an indirect impact on the element types specified by the first row. Of course, it is possible that both types are the same, i.e. the directly affected element type is the same as the indirectly affected element type. In these cases, we do not speak of propagation, as the uncertainty only affects one element type.

A major advantage of using our structured *uncertainty template* is the fact that a filled-in *uncertainty template* can be relevant to multiple software architectures within a domain of interest as it is defined on type level. That means, uncertainty types can be relevant for several architectures within the domain of interest. Furthermore, the template is iteratively extendable. For example, more ADD can be added while repeating *Step 1* to *Step 4* as indicated by the colour coding. The same applies to uncertainty types when executing *Step 8* (repeat *Step 5* to *Step 7* afterwards). In addition, it is possible to add not yet identified relationships between ADDs (i.e. between ADDs and uncertainty types) subsequently by repeating *Step 5* and *Step 6*.

Nevertheless, the approach consists of various manual steps, such as *Step 0*, *Step 5* and *Step 8*. They require a non-negligible amount of manual work, knowledge and expertise. However, *Step 0* is not part of the approach and *Step 5* is at least partially covered by related approaches [50, 27, 17, 3, 45]. A key factor for the success of the approach is *Step 8* because to the best of our knowledge, there is neither a collection of uncertainties (or uncertainty types) at architectural level, on which software architects can rely, nor an existing process to extract uncertainty types systematically. But as described before, uncertainties can be derived from assumptions and missing information in the design process, so software architectures are not completely without a suitable mean.

6.3.2. Uncertainty Type Derivation Approach - Example

We now conduct our *uncertainty derivation approach* from Section 6.3.1 based on the collection of ADD from Section 6.1. This exemplary application of the uncertainty type derivation approach serves only to illustrate the process and has no claim to completeness, nor to correctness of the content. Table 6.2 shows the completely filled *uncertainty template*, which we extend by line numbers for explanatory purposes (lines 1-10).

Step 0 is already performed in Section 6.1 resulting in the ADDs A1 to A4. In this case, the architectural model of a CBSA and our expertise leads to the collection of ADDs. Now, each ADD has to be added to the template (*Step 1*). Therefore, four multi rows are to be created to which the respective ADDs are inserted. In Table 6.2, those are lines one to two, three to five, six to seven, and eight to ten. Of course, the actual size of the multi rows is not known before hand. Identifiers and names correspond to the collection of ADDs.

					ADD-related Categories	*Uncertainty-related Categories*
#	**ADD**	**ID**		**Uncertainty Type**	**...**	**Architectural Element Type**
1	Deployment Location	A1	U1	Where is deployed?	...	Hardware Resource
2			U3	Is user data stored?		Component
3	Commu-nication Type	A2	U2	How is communicated?	...	Communication
4			U1	Where is deployed		Hardware Resource
5			U5	Is confidential data transmitted?		Interface
6	Persistence of User Data	A3	U3	Is user data stored?	...	Component
7			U1	Where is deployed?		Hardware Resource
8	Choice of Component	A4	U4	Which component is chosen?	...	Component
9			U6	What is the user behaviour		Usage Behaviour
10			U7	What is the behaviour of the component?		Component

Table 6.2.: Uncertainty Template - Running Example (Excerpt)

In the following (*Step 2* and *Step 3*), each ADD is *transformed* into the respective uncertainty type and added to the first row of the accompanying multi row. This produces the following uncertainty types: *"Where is deployed?"* (U1) results from *Deployment Location* (A1) and is inserted in line one. *"How is communicated?"* (U2) results from *Communication Type* (A2) and is inserted in line three. *"Is user data stored?"* (U3) results from *Persistence of User Data* (A3) and is inserted in line six. *"Which component is chosen?"* (U4) results from *Choice of Component* (A4) and is inserted in line eight. In the subsequent *Step 4*, the ADDs are classified according to the categories. The ADD-related categories are omitted for the sake of clarity.

Step 5, *Step 6* and *Step 7* are executed twice: First for the uncertainties that correspond to the ADDs and that result from *Step 2*. Second, for the uncertainties found in *Step 8*. In the following, we describe how software architects can perform those steps based on the uncertainties resulting from *Step 2*. The required information for *Step 5* and *Step 6* is based on the description of the ADDs A1 - A4 in Section 6.1. This is an example of how software architects could extract dependencies between ADDs from other sources, such as their description. For instance, the GDPR demands that user data must be stored within the *EU* [30]. The deployment location can therefore depend on whether user data is stored or not. The persistence of user data must be prohibited if there is an illegal deployment location. This bi-directional dependency leads to the creation of line two and line seven, i.e. the insertion of U3 in the multi row belonging to A1, and the insertion of U1 in the multi row belonging to A3. Further, specific deployment locations, such as deployment in a public cloud, demand a specific encryption type, hence this has an impact on the communication type. This leads to the insertion of U1 to A2 (line four). The fact that there is no bi-directional connection between A1 and A2 is a decision made for this scenario and is the responsibility of the architect, in this case us. Accordingly, we have decided that the communication type has no influence on the deployment location, but must be oriented towards it. The extent to which this decision is right or wrong is not important. Much more important is the circumstance that it is possible to correct this decision later if it turns out to be wrong. Afterwards, software architects have to categorize the added uncertainties. In the example, most of the uncertainty-related categories are omitted for the sake of clarity, and because the selected options only become necessary for the assessment of the severity of the impact. It is only category *architectural element type* that is relevant at this point. For each related ADD, it has to be decided which architectural element it affects. *Deployment Location* (*A1*), for instance, affects hardware resources. Therefore, the accompanying uncertainty type U1 affects the architectural element type *Hardware Resource*. This procedure is repeated for A2 to A4.

In *Step 8*, further uncertainty types are to be identified, i.e. those that do not originate from an untaken ADD. As previously described, *Step 5*, *Step 6* and *Step 7* are to be repeated for those newly identified uncertainties. Each uncertainty type is categorized and added to the respective multi row, hence to the ADD on which it has an impact. To that end, the textual description of the ADDs A1 - A4 in Section 6.1 serve as a source of information. For the communication type (A2), it is explained that communication encryption is only relevant if, for instance, the providing interface provides confidential data. This leads to line five, i.e. to the insertion of *"Is confidential data transmitted?"* (U5) to A2. According to the description, it should be possible to assign these uncertainty types to interfaces, which is why this is selected for category *Architectural Element Type*. At this point, one can argue whether U5 can and should be formulated as ADD, too. Although we have decided against this for this example, it can be corrected retrospectively. This further shows that the approach can be executed iteratively and the content of the template can thus be further improved. When choosing an appropriate component for a specific use case (A4), it can be necessary to know both, the behaviour of the users interacting with that component and the internal behaviour of the component itself. This leads to the insertion of *"What is the user behaviour?"* (U6) and *"What is the behaviour of the component?"* (U7) to lines nine and ten, respectively. As described in Section 2.5 and Section 5.2.2, the usage behaviour is considered as first class entity in CBSA, which is why *U6* is to be assigned to the architectural element type *usage behaviour*. U7 is to be assigned to components. At this point, a limitation of the approach becomes apparent, as the filled *uncertainty template* does not reveal the dependencies between the uncertainties within a multi row. This, however, has an impact on the order of how uncertainties might have to be resolved. With regard to the communication type (A2), one can argue that it is more important to first identify whether or not confidential data is transmitted (U5) before one determines the deployment location (U1), otherwise the deployment location will to some extent dictate the behaviour of an interfaces, and its component respectively.

In *Step 9* and *Step 10*, we determine the conclusive *impactOn* relationship between each uncertainty type and architectural element types which consists of the conjunction of directly and indirectly affected elements. *Step 0* to *Step 7* enable to immediately identify the direct impact of uncertainty types on architectural elements, as this information is provided by category *architectural element type*. For instance, U1 directly affects hardware resources. In addition, U1 affects A2 and A3 which are characterized by U2 and U3. As U2 affects the communication, and U3 affects components, we can summarize that U1 has impact on hardware resources, the communication and components. This needs to be done for U2 to U7 in the same way.

6.4. Uncertainty Annotation

In the previous section, we explained how to derive uncertainty types and how to determine to which types of architectural elements they can be assigned. This information is available in the form of a completed *uncertainty template* like the one we provided in Table 6.2. However, one of the goals of our thesis is to annotate existing architectures with actual uncertainties. In the following, we therefore use the example from Figure 6.1 to demonstrate how the *uncertainty template* helps to instantiate uncertainties and assign them to the correct elements of an architecture. We now distinguish between uncertainty and uncertainty type, and between architecture element and architecture element type, since we are now looking at a concrete architecture. Figure 6.4 exemplifies how software architects can annotate an architecture.

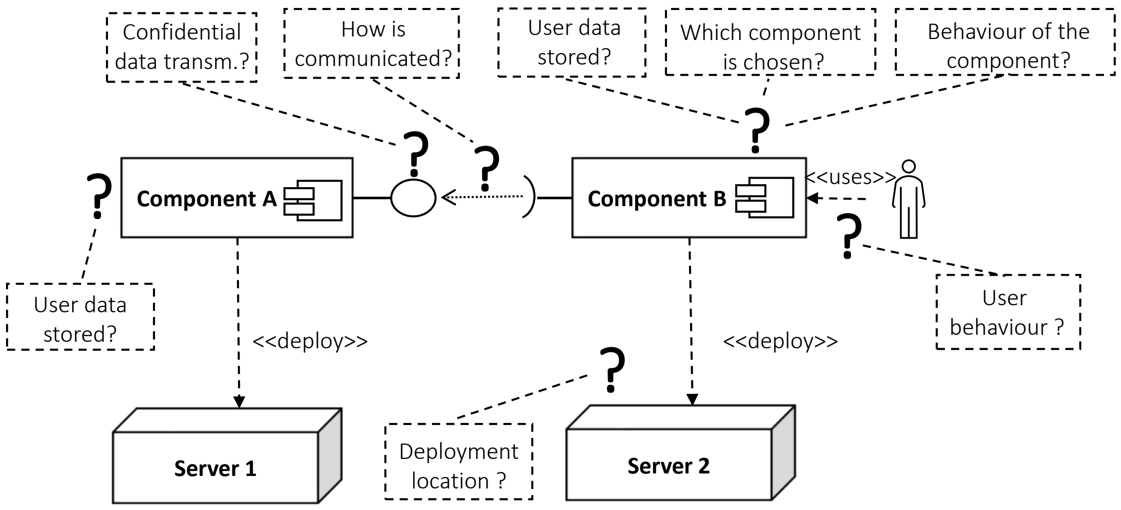


Figure 6.4.: Uncertainty Annotation - Running Example

The uncertainty types U1 - U7 are instantiated and assigned to concrete architectural elements as depicted in Figure 6.4. Each uncertainty corresponds to a dashed box, whereas the line indicated to which element it is actually assigned. Of course, it is possible that uncertainties are instantiated several times. An example is *"Is user data stored?"* (U3), which is instantiated once for Component A and again for Component B. Moreover, it can be expressed that an uncertainty only applies to certain elements. *"Where is deployed?"* (U1), for example, is only instantiated once and assigned to Server 2 which means, that the deployment location for Server 1 is certain, according to the software architect. Where which uncertainties are actually assigned is up to the software architect using the template, as this depends on his or her concrete architecture.

6.5. Uncertainty Propagation

One goal of this thesis is to assess the impact of uncertainties in software architectures. However, before the impact can be assessed, it must first be determined which elements could potentially be affected by which uncertainties. The direct impact is already covered by annotating software architecture with uncertainties, hence by the previous section. In the following, we provide the terminology and the process to determine indirectly affected elements.

6.5.1. Terminology

We motivated our research by elaborating the importance of the analysis of uncertainty propagation in Chapter 1 and Chapter 4. As described in those chapters, uncertainties either directly or indirectly affect architectural elements. For the latter, we introduce the term *uncertainty propagation*. The idea is based on the similarity to *change propagation analyses* such as the KAMP approach (cf. Section 2.6). Based on *change propagation analyses*, we define the terms *uncertainty set*, *affected set* and *impact set* as illustrated in Figure 6.5. The *uncertainty set* refers to a set of architectural elements of a specific architecture which are affected by instantiated uncertainties, or in other words, all elements on which uncertainties have a direct impact. The *impact set* is a result of the usage of our *uncertainty template* which provides an overestimation of the architectural elements that might be affected. In other words, all elements on which uncertainties have a direct impact, plus the ones on which they might have an indirect impact. In contrast, the *affected set* states which architectural elements are actually affected, either directly or indirectly. As with *change impact analyses*, it is not the aim (and practically not possible) to determine the affected set precisely [58]. Our approach is only to provide a "good" overestimation which includes each element that is actually affected, and not too many mistakenly affected elements. In

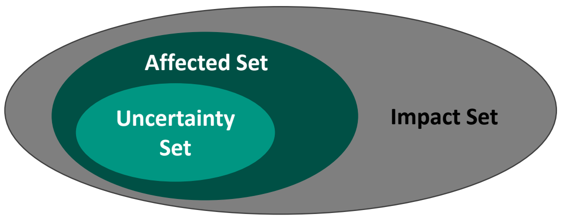


Figure 6.5.: Uncertainty Propagation - Terminology

the following, we explain how to derive an *impact set*, based on instantiated uncertainties in a software architecture, and a filled *uncertainty template*.

6.5.2. Determining the Impact Set

The starting point is a filled *uncertainty template* with which software architects already annotated a software architecture. The process is identical either for one single uncertainty or for a collection of uncertainties, irrespective of their type. But for the sake of simplicity, we describe the process for one uncertainty only. By definition, the *impact set* consists of the architectural element that is directly affected by an uncertainty plus the ones that might be affected. The first part is trivial, since it corresponds exactly to the element at which the uncertainty is annotated. The second part is based on the information provided by *Step 5.2* of our *Uncertainty Type Derivation Process* (cf. Section 6.3). It specifies the architectural element types on which a respective uncertainty type might haven an impact. Then, only the architectural elements that correspond to these type(s) have to be included in the *impact set*. Again, it should be noted that this procedure provides an overestimation of the affected set, as every element of a given type is included, regardless of whether the uncertainty actually has an impact on that element or not. Nevertheless, it is an improvement on the initial situation, as without the template potentially all elements of an architecture would have to be examined.

To improve the impact set, i.e. to sort out some of the wrongly classified elements, the software architect can analyse the original *impact set* manually. For instance, software architects can use contextual information to subsequently discard the uncertainties that have no impact at all. This could be, for example, information about the data flow, requirements or ADDs that have already been made. Another possibility could be to define structural propagation rules which determine only those elements that are "reachable" by an uncertainty. Those rules are based on the structural properties of software architectures. Either way, each improvement could potentially decrease the impact set and thus the number of elements to be examined.

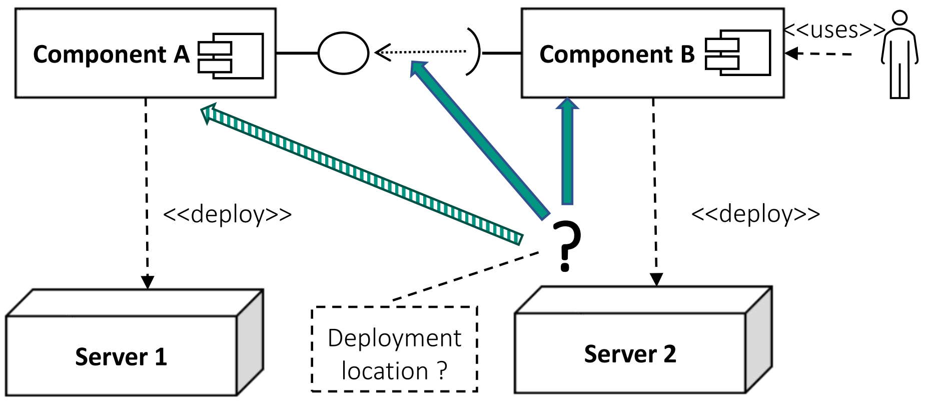


Figure 6.6.: Uncertainty Propagation - Running Example

Example: For the sake of simplicity, we use the example from the previous section but consider only one uncertainty. The example (enhanced by the propagation) is illustrated by Figure 6.6. Potential propagation effects (i.e. the indirect impact of uncertainties) are exemplified by the three green arrows starting at the uncertainty of type *"Where is deployed?"* (U1), which is assigned to Server 2. According to the filled *uncertainty template*, uncertainties of type U1 can impact the architectural element types *component* and *communication*. With that information, we know that this uncertainty can only affect Component A, Component B and the communication entity between them. These three elements thus form the initial *impact set*. In particular, neither the two interfaces, nor Server 1 need to be examined afterwards. The *impact set* can now be improved manually. For example, we assume that instances of U1 only affect components and their directly reachable elements deployed on the hardware resource to which the uncertainty is annotated. This exemplary structural propagation rule removes Component A. Further, we know (due to a previously made fictive design decision) that the communication between those two components uses encryption mechanisms. Therefore, the manually cleaned *impacts set* finally consists only of Component B. Using structural propagation rules, as well as context information enables to reduce the *impact set* to one element only.

6.6. Uncertainty Impact Assessment

Besides the identification, annotation and propagation of uncertainties, **RQ1.4** also asks about the assessment of uncertainties. Therefore, we demonstrate in this section how software architects could use the *uncertainty template* and its categories to asses annotated and propagated uncertainties regarding their impact on confidentiality. It should be added that this is explicitly not about assessing whether or not an uncertainty actually has an impact. Rather, the focus is on assessing the severity of a potential impact. The basis is the completion of the processes as described in Section 6.3 - Section 6.5. In those sections, most of the ADD- and *uncertainty* related categories are not of high importance, as they are only necessary to annotate uncertainties and determine their propagation effects. This is different when it comes to the assessment of the severity of the impact of uncertainties.

In Section 5.2.1 and Section 5.2.2, we elaborated for each category independently how it can contribute to the assessment of the impact of uncertainties. In the course of this section, we exemplify how the combination of certain categories can help to assess the impact even more precisely. For each combination of categories, we present one or several choices of options. Software architects can use these suggestions when they try to assess the impact of an actual uncertainty whose type is categorized with exactly the same options. In addition to the general assessment of the impact of uncertainty (especially regarding their impact on the confidentiality of a system), the categories could also be used for prioritisation purposes. The combination of certain categories could therefore contribute to answer the following question: *"Which uncertainty should be considered next?"*.

At this point, we would like to emphasise that the proposed combinations do not claim to be exhaustive and only indicate how software architects could use the template for assessment purposes. Moreover, it is not feasible to fully explore all combinations of categories due to the large number of options available (15 categories, 42 options). We therefore only show a selection of combinations that have proved relevant or interesting in the course of our research. They are based on our profound knowledge of uncertainties, but do not arise from a systematic study. We exemplarily applied these guidelines on instantiated uncertainties in Section A.1.

Probability, Possibility and Cost of Revision: The three architectural-related categories *Probability of Revisability* (*likely, unlikely*), *Possibility of Revisability* (*yes, no*) and *Costs of Revision* (*high, low*) are strongly interrelated. These categories are relevant for each first uncertainty type of a multi row of the template, as they correspond to the associated ADD. For example, if it is possible (*yes*), *likely* and at the same time expensive (*high* costs) to revoke an ADD, the associated uncertainty should be treated with caution. *High* costs, on the other hand, could be neglected, if it is *unlikely* that an ADD will be revoked. The combination *likely*, *no* and *high/low* initially appears contradictory. This combination could occur, for example, if ADDs are initially considered *likely* to be revisable due to many uncertainties, but cannot be revised due to their fundamental impact on the design of the system. In such a scenario, the other uncertainties should be dealt with before the ADD-related uncertainty is resolved.

Probability of Revisability and Root Cause: The architectural-related category *Probability of Revisability* (*likely, unlikely*) and the uncertainty-related category *Root Cause* (*missing Information, assumption*) can be combined as follows: ADDs are more *likely* to be revised when they are based on several *assumptions*, i.e. on uncertainties that cannot be resolved by acquiring *missing information*. This is because the assumptions may turn out to be wrong, so that the ADD that depends on them must be revised. To prevent this, all assumptions should be evaluated in detail before an ADD is made, especially if the ADD is already *likely* to be revised. This can be transferred to the uncertainty representing the respective ADD, as each affecting uncertainty whose *Root Type* is *assumption* should be considered first.

Impact on Confidentiality and Severity of the Impact: The combination of the categories *Impact on Confidentiality* (*indirect, direct*) and *Severity of the Impact* (*high, low*) is probably the most obvious one when it comes to the assessment of the impact of uncertainties on the confidentiality of the system. If uncertainties with *direct* and *high* impact can be resolved during the design process, software architects should do so. Otherwise, such uncertainties should be carefully tracked.

Manageability, Resolution Time and Root Cause: The categories *Manageability* (*fully reducible, partial reducible, irreducible*), *Resolution Time* (*requirements time, design time, deployment time, run time, never*) and *Root Cause* (*missing information, assumptions*) could be combined to asses whether uncertainties can and should be resolved during design time and, if resolved (partially), how much attention is still needed afterwards. If an uncertainty is *fully reducible* during *requirements* or *design time* and based on a *missing information*, the impact on the confidentiality can probably be neglected, as software architects should be able to resolve it. In case it is only *partial reducible* (same options for the remaining categories), software architects should be aware that an impact might not be completely impossible. If an uncertainty is *partially reducible* to the extent that an *assumption* is made, software architects should apply appropriate means to be able to check the *assumption* at the given resolution time (*deployment* or *run time*). Uncertainties categorized in this way could presumably have an impact on confidentiality, if the assumptions turn out to be wrong. Nevertheless, not all combinations are likely to make sense, as they seem to be contradictory. For instance, if an uncertainty can *never* be resolved, it is probably also *irreducible* and vice versa. Other combinations with one of the two options do not seem logical. However, each of the three categories aims to provide a different perspective for assessing the impact of uncertainty, so illogical combinations do not necessarily mean that the categories and options are ill-defined.

6.7. Conclusion

In this chapter, we presented our *uncertainty template*, a tool with which software architects can structurally identify types of uncertainties and their possible propagation effects based on a previously identified set of ADDs. In addition to the general structure (Section 6.2), we outlined a process for how software architects can use the template to derive uncertainty types in Section 6.3. Further, we presented in Section 6.4 how software architects can use a filled template to annotate software architectures with instantiated uncertainties. In Section 6.5, we introduced the terminology for the propagation of uncertainties, which is based on the idea of *change impact analyses*. In doing so, we have shown the extent to which software architects can both determine and reduce the so-called uncertainty *impact set*. Finally, we illustrated in Section 6.6 how software architects can use the template to assess uncertainties and their impact.

The *uncertainty template* has several advantages. First and foremost, the filled-in *uncertainty template* can be reused for other (similar) architectures which enables a reuse of knowledge. Also, it provides the possibility to systematically derive and structurally depict types of uncertainties. It is further easy to extend, so that it is initially possible to deal with incomplete knowledge about possible uncertainties. The *uncertainty template* can also create awareness, as the person using a filled template may not necessarily be the one who had it completed. In other words, more experienced software architects have the knowledge to complete the *uncertainty template* for a certain domain, for instance, for software architectures in the area of IS, whereas less experienced software architects use it to identify and instantiate possible uncertainties in their actual software architecture. This means that users of a completed template no longer have to find out for themselves which types of uncertainties exist and which architectural elements they affect. Propagation also potentially reduces the number of architectural elements to be investigated, as one only has to consider a maximum of those elements that correspond to the respective element type on which an uncertainty type has an effect. Without this information (i.e. without the template, the type derivation and the propagation process), it would be necessary to examine all elements for a possible impact for each uncertainty respectively.

In the course of this chapter, we also sketched some limitations and drawbacks. Due to structural constraints, it was necessary to adjust the relationships (i.e. the multiplicities) between uncertainties, ADDs and architectural elements with respect to our results from

Chapter 4. To that end, we explained why these changes have no restriction on the expressiveness of the template, as the original multiplicities can be simulated by information duplication. Our process is based on the availability of a collection of confidentiality-relevant ADDs which could be seen as a drawback. Yet, this is not part of this thesis, as we consider these to be provided by different sources. Furthermore, two non-trivial manual steps have to be conducted to fill the template: First, it has to be manually identified which uncertainty types might affect which ADDs. Second, additional uncertainties are to be identified manually based on other sources of information. Again, let us point out that these steps are only necessary to fill the template initially. In the case of subsequent (multiple) use, these steps are omitted. Using some sample combinations of categories, we have shown how the template can provide insights into the impact of an uncertainty. Although the illustrated implications between the different options chosen are interesting, an all-quantified statement is probably impossible due to the amount of possible domains of interests, eventual uncertainties and software architectures. In general, we assume that the actual impact is always dependent on the system under consideration. Another drawback to this point is the lack of tool support for the annotation and automatic propagation of uncertainties for software architectures. To the best of our knowledge, there is no tool that explicitly enables to annotate software architectures with uncertainty as a first class entity. Manual propagations based on (structural) propagation rules can be tedious, especially for larger architectures. In order to support the annotation of software architectures, as well as to automate propagation, we present our Uncertainty Impact Analysis (UIA) in the following chapter.

7. Uncertainty Impact Analysis

This chapter elaborates how to support the annotation and propagation of uncertainties in software architectures (**RQ2**). To that end, we extends the Palladio Component Model (PCM) by the capability to represent uncertainties in software architectures as first-class entity (**RQ2.1**). Based on that, we present our Uncertainty Impact Analysis (UIA), which supports the annotation and automates the propagation of uncertainties in software architectures (**RQ2.2**). Figure 7.1 illustrates the concept of the UIA, including the integration of the enhanced PCM.

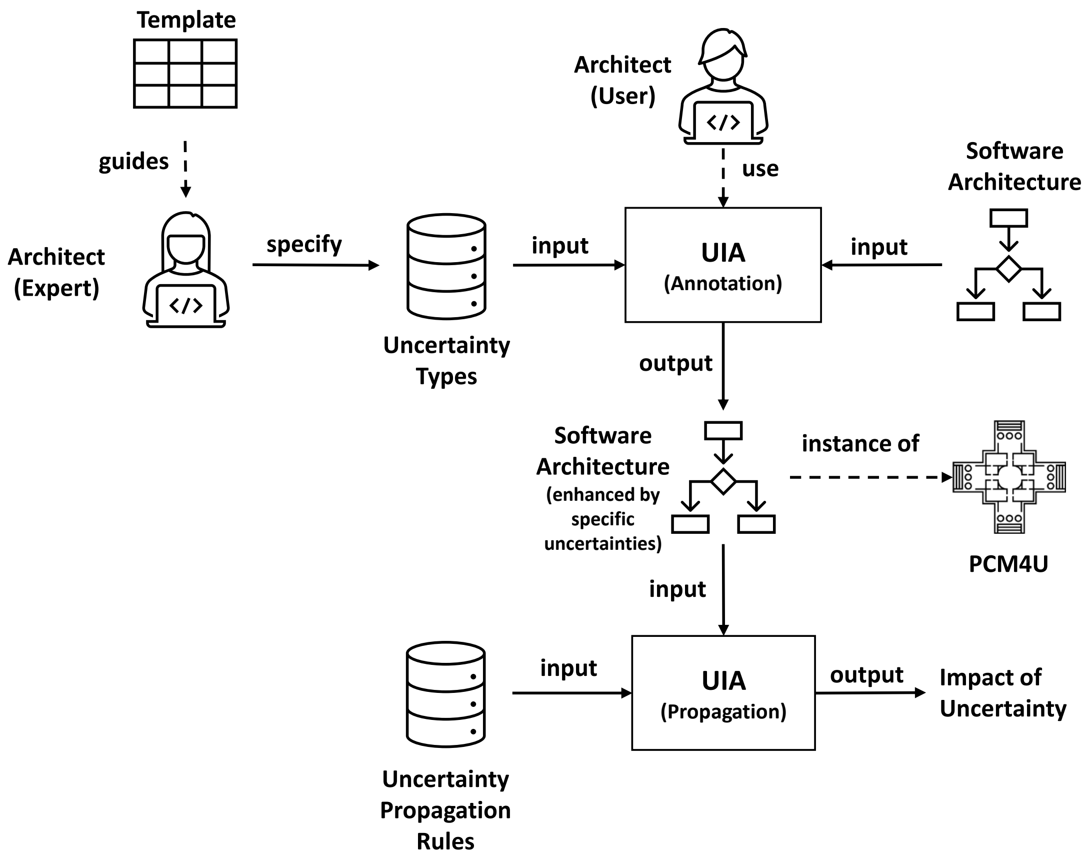


Figure 7.1.: Overview of the Approach

As described in the previous chapter, we differentiate between types of uncertainties and instantiated uncertainties. Uncertainty types are elaborated by more experienced software architects, for a specific domain of interest, and by using our *uncertainty template*. Less experienced software architects can use those uncertainty types to annotate their CBSA. Further, our uncertainty template enables to derive the direct and indirect impact of uncertainties on other architectural elements which we refer to as *uncertainty propagation*. So far, this process is conducted manually without tool support. This gap is to be filled by our proposed analysis. To differentiate between experienced and less experienced software architects, we distinguish between two stakeholder roles: *Expert Architects* and *User Architects*. Both roles are illustrate by Figure 7.1, including their tasks as well as inputs and outputs.

Expert architects specify types of uncertainty by using our *uncertainty template*. A collection of uncertainty types serves as input for the UIA. *User architects* use the analysis to annotate their existing architecture by actual uncertainties. Based on a selected uncertainty type, the analysis is designed to help in finding all elements of an architecture on which an uncertainty can be annotated. Based on pre-defined uncertainty propagation rules, the analysis automatically calculates the impact of the uncertainties, i.e. it determines to which other elements the instantiated uncertainties could potentially be propagated.

The structure of this chapter is as follows: Section 7.1 introduces implementation aspects, including the meta modelling and the class structure of the UIA. In Section 7.2, we present the specification of uncertainty types, as well as the supported types of architectural elements. Section 7.3 briefly introduces how *user architects* can use the analysis to annotate their architectures with uncertainties. Afterwards, Section 7.4 provides a profound explanation of the actual propagation mechanisms including the presentation of two exemplary algorithms. In Section 7.5, we illustrate how the UIA enables *user architects* to assess uncertainties and their impact. In Section 7.6 we differentiate our analysis from the existing KAMP approach. We discuss limitations and possible improvements of the UIA in Section 7.7. This chapter is concluded by Section 7.8.

7.1. Overview

Technically, our approach is to extend the Palladio Component Model (PCM) (cf. Section 2.5), which is the ADL we use to model CBSAs. The PCM is based on the Eclipse Modelling Framework (EMF) (cf. Section 2.4) and is defined on M2 level. To extend the PCM by the capability of annotating software architectures with uncertainties, one has several possibilities, such as extending the PCM meta-model itself, stereotyping or referencing. We decided to apply the last one as we consider it to be the most stable and the most straight forward solution. In order to extend the PCM, we create several models which reference each other, and in particular, the PCM meta-model itself.

Figure 7.2 illustrates the different models and the model hierarchy which we created to implement the UIA. Model ADD describes the Architectural Design Decisions (ADDs), including a set of attributes which correspond to the ADD-related categories of the *uncertainty template*. Instances therefore represent collections of instantiated (i.e. identified) ADDs. Model UncertaintyType represents the uncertainty types, including a set of attributes which correspond to the uncertainty-related categories of the *uncertainty template*. Similar to ADDs, instances of this model represent a collection of identified uncertainty types. Model UncertaintyTemplate references both, *UncertaintyType* and ADD and represents our uncertainty template. Each of the models are to be created by *expert architects*. Model Uncertainty describes the uncertainty-related model elemtents. In particular, this includes

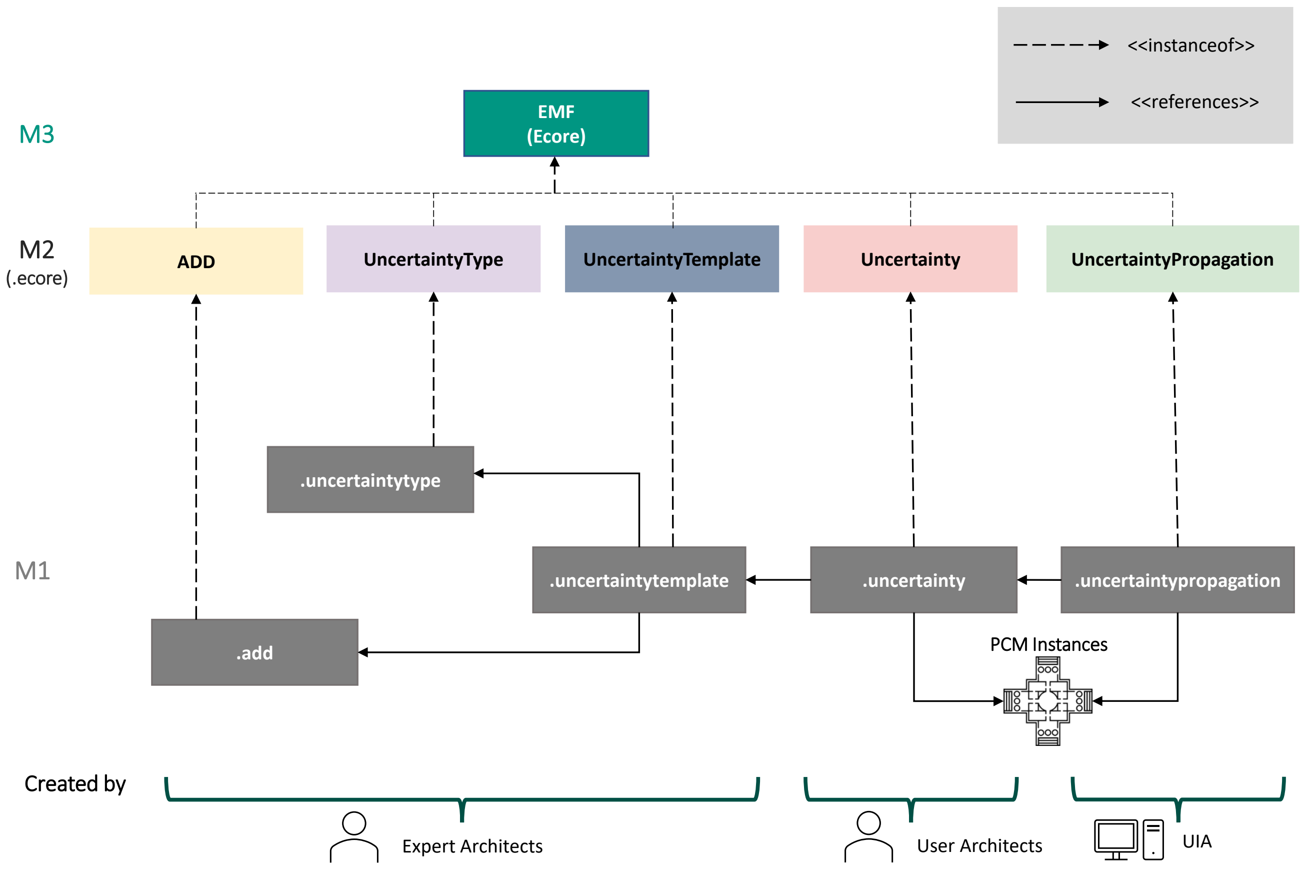


Figure 7.2.: Uncertainty Impact Analysis - Models and Model Hierarchy

the relationships to uncertainty types, ADDs and architectural elements. As illustrated in Figure 7.2, instantiated uncertainties always reference the uncertainty template which they are based on, as well as the architectural model (i.e. instances of PCM models) which they reference. Lastly, model UncertaintyPropagation describes the model elements required for the uncertainty propagation. Instances are created automatically by the UIA. Conceptionally, we have multiple modelling levels, as *expert architects* instantiate models which are then used by *user architects*. Similarly, *user architects* define models which in turn are used by the UIA to produce the actual propagation result. However, EMF limits the available levels to M2 and M1 only. Therefore, we technically realize multiple levels by referencing, which is a common approach when extending available meta-models such as the PCM. In particular, defining one single uncertainty meta-model would be impracticable at this point as various roles instantiate different model elements at different points in time.

Figure 7.3 illustrates a strongly simplified class diagram of the UIA. The affiliation of the entities to the models is highlighted with colours. The division according to the creators is further indicated by the dotted lines. Again, the ADD- and UncertaintyType- related entities are part of the UncerteinatyTemplate model, which is why they belong to the same part. For the sake of simplicity, we omit most of the enums that represent the architectural and uncertainty-related categories. Enum ArchitecturalElementTypes represents the types of

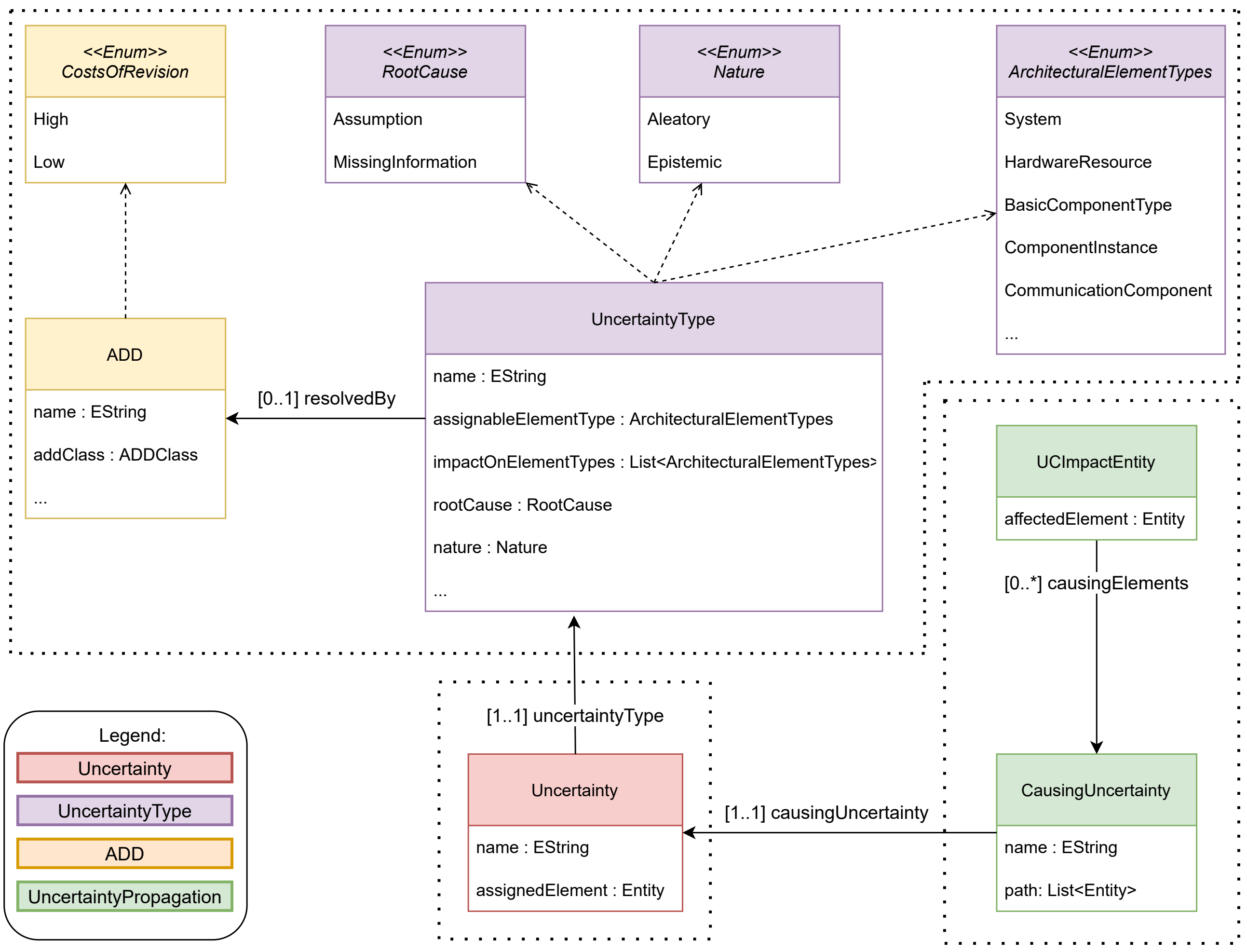


Figure 7.3.: Uncertainty Impact Analysis - Class Diagram

architectural elements that are supported by the UIA, i.e. to which uncertainties can be assigned. CausingUncertainty and UCImpactEntity are propagation-specific entities used by the UIA when calculating the propagation effects. In the following sections, we explain each part of the UIA in more detail, including how software architects can use the analysis.

7.2. Specification of Uncertainty Type Models

In Chapter 6, we introduced our uncertainty template which enables software architects to derive uncertainty types based on existing ADDs. We consider this task to be executed by *expert architects* as it requires manual effort and a non-negligible amount of expertise. Once this process is completed, a collection of categorised ADDs and uncertainty types is available. Of course, it is not required to use the uncertainty template to identify possibles types, but nevertheless recommended. The architects' task is now to provide this information in the form of an ADD and UncertaintyType model (and finally combined as UncertaintyTemplate model). Here, the EMF framework already provides tool support in form of tree editors which is why we do not implement this functionality. For each ADD- and uncertainty related category of the uncertainty template, we provide a respective enum that contains the relevant options. *Expert architects* therefore only need to create both models and add instances of type *ADD* and *UncertaintyType*, including the desired options. Yet this happens with a single exception: For category *Architectural Element Type*, the uncertainty template supports general concepts of CBSE only, such as *interfaces* or *components*, and is not bound to a specific ADL. However, the PCM allows for a more specific, i.e. a more fine-grained collection of types of architecture elements to which uncertainties could be assigned or propagated. In order to exploit this possibility, we therefore support more element types as specified in the uncertainty template.

Architectural Element Type	**Palladio Concept**	**Description**
System	System	Refers to the whole system
Hardware Resource	Resource Container	Refers to hardware resources such as servers
Basic Component Type	Basic Component	Refers to components in repository (not instantiated)
Component Instance	Assembly Context	Refers to instantiated top level components
Basic Component Behaviour	RDSEFF	Refers to inner component behaviour descriptions
Communication Component	Assembly Connector	Communication between all top level components
Communication Resource	Linking Resource	Describes communication between hardware resources
System Interface	Role	Refers to the system's provided and required interface
Component Interface Instance	Assembly Context & Role	Refers to interfaces of instantiated top level components
Component Interface Type	Role	Refers to interfaces of component types (not instantiated basic components)
Usage Behaviour	Entry Level System Call	Refers to the actual usage of the system by external actors

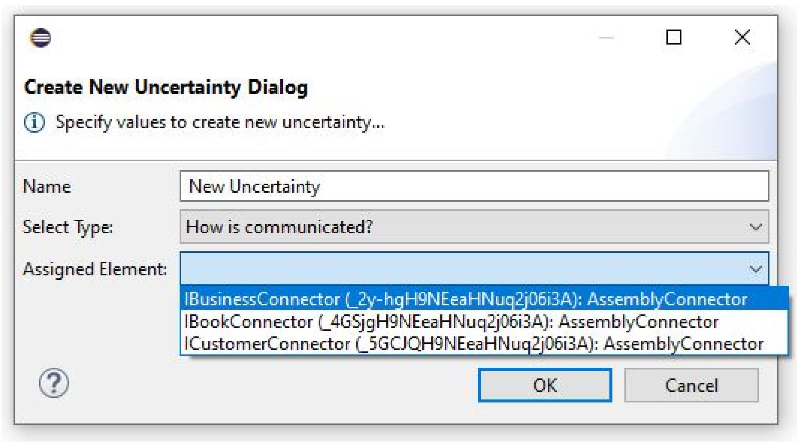
Table 7.1.: Supported Architectural Element Types and Mapping to PCM Concepts

Table 7.1 illustrates the architectural element types which are currently supported by the UIA, including a mapping to available PCM concepts. Be aware that this is only the current state as further (more specific) element types might be supported in the future. The PCM defines various architectural elements to which uncertainties can and will be assigned, but their technical names do not always fit our purpose. In the course of the implementation, it became apparent that the technical PCM names are often not meaningful enough. For instance, the technical Element Assembly Connector describes the communication between components in software architecture. A more suitable name in this case is therefore Communication Component. But more important, the PCM uses the same technical elements to describe syntactically identical, but semantically different elements. The technical element Role for instance, refers to interfaces in general. As we want to distinguish between interfaces on component and system level, as well as between interfaces of instantiated components and component types, it is not possible to use the technical names. In case of Component Interface Instance, we have to combine two PCM concepts to identify the desired architectural element as no suitable concept is available for mapping. Although we decided against specifying the *System* option when defining the template, the analysis offers this as a fallback mechanism. This is only for practical reasons, as a referenceable architectural element type should be defined for each uncertainty type.

Overall, the collection of the presented architectural element types provides the following two advantages: First, an alignment between the concepts of our uncertainty template and the PCM. Second, a more fine-grained view of architectural elements in CBSAs that might be affected by uncertainties. The definition of more specific architectural element types requires the *expert architect* to decide which option to choose. In some cases, the decision is trivial as there is an one-to-one mapping between the architectural element types specified in the uncertainty template and the one supported by the UIA. For instance, Hardware Resource or Usage Behaviour are identical. With regard to interfaces, *expert architects* have to decide whether the derived uncertainty types should be assignable to system interfaces, component interfaces of instantiated components or to component interfaces of uninstantiated components. The same applies for components. All in all, it enables the architect to specify even more precisely to which architectural elements uncertainties can be annotated and to which architectural element types they might propagate.

7.3. Uncertainty Annotation

In order to be able to annotate software architectures with uncertainties, *user architects* require two types of models: First, an instance of an *UncertaintyTemplate* model that specifies uncertainty types for their domain of interest. Second, an existing PCM-based software architecture, i.e. instances of a *Component Model*, *Assembly Model*, *Allocation Model*, and *Usage Model* (cf. Section 2.5). To actually instantiate an uncertainty and assign it to a specific element, a *user architect* selects one of the available uncertainty types. Each uncertainty type defines to which architectural element type it can be assigned. As soon as a type is selected, the UIA automatically proposes each element to which the uncertainty can be assigned. This is illustrated by Figure 7.4 which is a screenshot of the actual UIA tool. This step is repeated for each uncertainty a *user architect* wants to create. Each instantiated uncertainty instance is added to an Uncertainty model instance. This model can be saved and reloaded to either display existing uncertainties, add new ones or delete resolved uncertainties. In doing so, *user architects* can manage their uncertainty annotations throughout the software design process and beyond.



Create New Uncertainty Dialog

Specify values to create new uncertainty...

Name: New Uncertainty

Select Type: How is communicated?

Assigned Element:

IBusinessConnector (_2y-hgH9NEeaHNuq2j06i3A): AssemblyConnector
IBookConnector (_4GSjgH9NEeaHNuq2j06i3A): AssemblyConnector
ICustomerConnector (_5GCJQH9NEeaHNuq2j06i3A): AssemblyConnector

Figure 7.4.: Create New Uncertainty Dialogue - Screenshot

7.4. Uncertainty Propagation

In Section 6.5, we introduced the terminology of *uncertainty propagation*, including the definition of the *uncertainty set*, the *affected set* and the *impact set*. Thereby, we also explain how the uncertainty template can be used to derive the *impact set* and why doing this on type level produces a vast overestimation of the actual *affected set*. Manual post-analyses could reduce this overestimations, but they are tedious and require expertise. Yet, this is precisely what is supposed to be avoided by using the analysis. Therefore we propose structural propagation rules which determine only those elements that are "reachable" by an uncertainty. This possibility is implemented and is explained in more detail hereafter. Note that this section requires considerable prior knowledge of the structure of the PCM. More detailed information can be found here [56]. In the following, we first provide general information with regard to the uncertainty propagation rules, followed by a brief description of the technical realisation and some examples of existing algorithms.

7.4.1. Uncertainty Propagation Algorithms

The uncertainty propagation rules are algorithms which traverse PCM-based architectures to identify reachable architecture elements. The starting element is defined by the uncertainty which is to be propagated, i.e. the element to which the uncertainty is assigned. When traversing the architecture, the algorithms collect the elements that are both, structurally reachable and whose types correspond to one of those defined by the respective uncertainty type (defined via impactOnElementTypes). Note that this list can be empty, indicating that no propagation is to be conducted. So far, ten different architectural element types are identified (cf. Section 7.2). As each combination of start and end element type theoretically needs to be supported, a total of 100 algorithms is required. The current prototypical implementation comprises 13 algorithms, which we consider to be the most relevant ones.

7.4.2. Technical Realisation

Our propagation is technically realised by using the KAMP framework (cf. Section 2.6) due to the similarity of change propagation analysis and uncertainty propagation analysis. As illustrated in Figure 7.3, UCImpactEntity and CausingUncertainty are the two relevant classes that are required for the propagation. CausingUncertainty wraps the actual uncertainties to be propagated, but adds a list of PCM entities describing the propagation path. This is possible as each Palladio concept (cf. Table 7.1) inherits from Entity. For each architectural element (i.e. Entity) that is affected by one or several uncertainties, an instance of UCImpactEntity is created. Given a collection of uncertainties to be propagated as input, the UIA automatically creates an UCImpactEntity, where the affected element corresponds to the element directly affected by the uncertainty, and the list of causing elements contains one CausingUncertainty that wraps the actual uncertainty.

The UIA is able to automatically choose which algorithms to execute, as each uncertainty has an uncertainty type which specifies on which architectural element types this uncertainty might have an impact. So starting element type and zero to several indirectly affected element types together provide the required information to choose the specific propagation algorithms. For each indirectly affected element (i.e. the result of the propagation), the UIA either creates a new UCImpactEntity and adds a CausingUncertainty containing the causing uncertainty, or if the element is already affected by another uncertainty, it simply adds a CausingUncertainty to the list. If there are multiple paths from a start element to an end element, then multiple CausingUncertainty instances are created that wrap the same uncertainty but have different paths.

The output therefore consists of two sets of UCImapctEntity instances: First, the entities that reflect the directly affected architectural elements (via uncertainty annotation). Second, the entities that describe the architectural element which are indirectly affected by one or several uncertainties (via uncertainty propagation). Consequently, the unprocessed output is ordered according to affected architectural elements. This form of presentation is conditioned by the use of the KAMP framework. But we need a different form of representation, i.e. a sorting according to uncertainties and the elements affected by them. Therefore, the UIA automatically transforms the output by iterating over the collection of affected entities in order to derive for each uncertainty the architectural elements it affects, including the paths. Again, if there are multiple paths between the same start and end element, the affected element is included twice. In the following, we sketch two uncertainty propagation algorithms.

7.4.3. Uncertainty Propagation Algorithm - Examples

We provide a textual explanation of some of the algorithms. For the sake of simplicity, we decide against the illustration via (pseudo-) code as some of the algorithms work from top to bottom (i.e. beginning at the starting element), others from bottom to top (i.e. beginning with all potential ending elements) and yet others starting somewhere in the middle due to technical peculiarities of the PCM. Further, the (pseudo-) code notation gets even more complicated as recursion is required frequently. It further has to be noted that the algorithms are very simplified and only sketch the actual way of traversing the PCM models. A detailed representation has no added value at this point, but can be looked up using the well-documented code under [10].

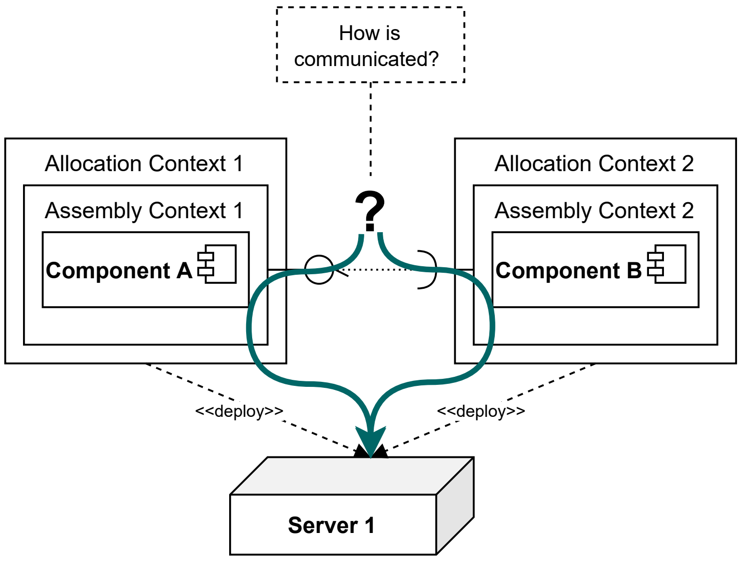


Figure 7.5.: Propagation from Communication Component to Hardware Resources

Figure 7.5 shows a strongly simplified architecture diagram which combines some PCM models for illustration purposes. Component A and *Component B* are instantiated in a separate assembly context, which are both allocated on the same server. To realize the allocation, the PCM uses the concept of allocation context. The uncertainty type *How is communicated?* is assignable to architectural elements of type Communication Component and has an impact on Hardware Resources. Given this uncertainty type, the *user architect* instantiates this type by annotating the communication component that connects both, *Assembly Context 1* and Assembly Context 2. The algorithm to propagate uncertainties from communication components to hardware resources is as follows:

1. Extract providing and requiring interface connected by Communication Compoent
2. Do for both interfaces
    2.1. Extract Assembly Context
    2.2. Extract Allocation Context
    2.3. Extract Hardware Resource
    2.4. Create UCImpactEntity for Hardware Resource (if not yet created)
    2.5. Add CausingUncertainty, including the path

With regard to Figure 7.5, the UIA creates two CausingUncertainty instances, both containing the same uncertainty but a different path, and which are both added to the same UCImpactEntity at Server 1. Propagation is now complete and the output is available to the user.

7.4.3.2. From System Interface to Component Interface Type

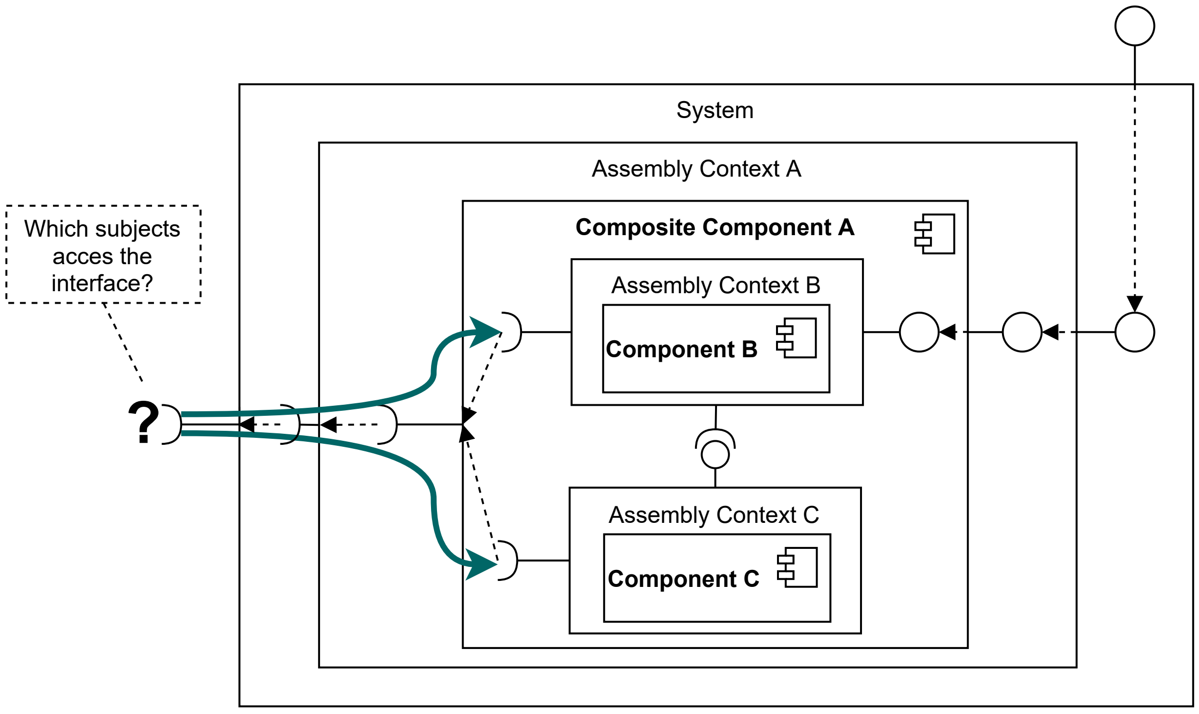


Figure 7.6.: Propagation from System Interface to Component Interface Type

Again, Figure 7.6 shows a strongly simplified architecture diagram. Component B and Component C are encapsulated by Composite Component A. Delegation connectors connect the interfaces from the basic components, to the composite component to the system interface. Further, it illustrates the assembly contexts that encapsulate Component B and Component C to build Composite Component A, as well as the assembly context that encapsulates (i.e. instantiated) Composite Component A. The uncertainty type *Which subjects access the interface?* is assignable to architectural elements of type System Interface and has an impact on Component Interfaces Type. Given this uncertainty type, the *user architect* instantiates this type by annotating the required system interface. The algorithm to propagate uncertainties from system interfaces to hardware resources is as follows:

1. Extract Delegation Connector which connects System Interface and AssemblyContext
2. Extract Assembly Context
3. Do if encapsulated component is Composite Component
    3.1. Extract Delegation Connectors which connect respective Composite Component Interface with encapsulated *Assembly Contexts*

4. Repeat previous step until encapsulated component is Basic Component
5. Do for each found Basic Component
    5.1. Extract respective Interface Type of component
    5.2. Create UCImpactEntity for Component Interface Type (if not yet created)
    5.3. Add CausingUncertainty, including the path

With regard to Figure 7.6, the UIA creates two CausingUncertainty instances, both containing the same uncertainty but a different path. Each of them is added to a different UCImpactEntity (one refers required interface of Component B, the other one refers the required interface of Component C).

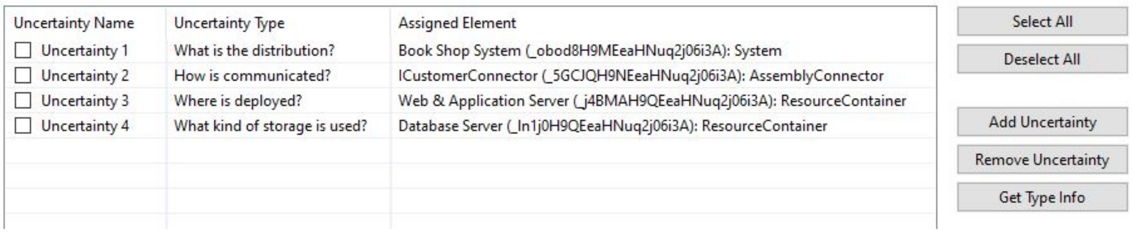
7.4.4. Termination

A common problem in traversing models, especially in change impact analysis, is the termination of algorithms [58]. However, the UIA distinguishes between input and output sets, whereby the algorithms process all elements of the input set one after the other and store the results in the output set. In particular, this means that the input set is not changed during propagation and the uncertainty propagation analysis terminates as soon as each uncertainty (i.e. the input) is propagated. In contrast to the KAMP approach, propagation is only carried out in one step. The further use of the output as input for a next propagation step is currently not planned, but could be done in the future. Nevertheless, it is possible that architecture models have (illegal) cyclical dependencies, so that the traversal may not terminate. Since checking the models for correctness is beyond the scope of this thesis, we assume correctly modelled architectures, especially without cycles. From these aspects it can be concluded that propagation always terminates in correct architecture models.

7.5. Uncertainty Assessment

When assessing uncertainties, a distinction must be made between two areas. On the one hand, determining the impact of uncertainty and, on the other hand, assessing the impact. The UIA supports the software architect in both.

In determining the impact, the UIA helps as follows: First, it enables the management of uncertainties in general, i.e. *user architects* can annotate their architectures and store/reload/enhance this information. As shown in Figure 7.7, the UIA visualizes the instantiated uncertainties, including their uncertainty type and the architectural element to which they are assigned. Second, it automatically calculates an improved *impact set* based on predefined propagation rules, i.e. it determines the impact of uncertainties with regard to the architectural elements which they might affect. Although it is still necessary to manually assess whether an impact actually exists at this point, this is limited to the elements of the *impact set*. Since we assume that the impact set contains only a part of all possible architectural elements, the UIA helps insofar as it can presumably reduce the set of elements to be assessed. This manual assessment can be based on contextual information such as the data flow, requirements, information about exchanged data or decisions already made, as described in Section 6.5.



Uncertainty Name	Uncertainty Type	Assigned Element
☐ Uncertainty 1	What is the distribution?	Book Shop System (_obod8H9MEeaHNuq2j06i3A): System
☐ Uncertainty 2	How is communicated?	ICustomerConnector (_5GCJQH9NEeaHNuq2j06i3A): AssemblyConnector
☐ Uncertainty 3	Where is deployed?	Web & Application Server (_j4BMAH9QEeaHNuq2j06i3A): ResourceContainer
☐ Uncertainty 4	What kind of storage is used?	Database Server (_In1j0H9QEeaHNuq2j06i3A): ResourceContainer

Figure 7.7.: Uncertainty Management - Screenshot

In assessing the impact, the UIA helps as follows: As described in Section 6.6, the categories of the uncertainty types can be used to asses (i.e. to compare) uncertainties at type level. Therefore, the analysis helps by visualizing the type-related information. More precisely, for each uncertainty type, *user architects* can retrieve the selected options for the architectural- and uncertainty related categories. Based on that, *user architects* can manually compare existing uncertainties based on the guidelines provided in Section 6.6. Similarly, the analysis visualises the propagation results. As indicated, counting metrics such as *Number of Affected Elements* can help to assess the impact of uncertainties. Therefore, we exemplarily implement a functionality that sorts propagated uncertainties by the amount of elements they affect. It must be emphasised again that the current state of the UIA is to be considered a proof-of-concept implementation and does not claim to be a mature tool.

7.6. Differentiation from *KAMP*

In the previous sections and chapters, we describe the similarity to the KAMP approach which is used to calculate the impact of changes by propagation change requests. Although the concept of propagating via architectural models, specifically *Palladio* models, is not new, we differ from KAMP in that we provide the possibility to propagate uncertainties. This includes modelling uncertainties as a first class entity, the knowledge of what propagation algorithms should look like, and the interpretation of the results. All in all, we use the basic idea of KAMP in a different context, especially with the aim of enabling non-experts to take on expert tasks, which is the annotation, propagation and assessment of uncertainties.

7.7. Limitations and Improvements

The functional scope of the UIA is limited. In particular, not all the necessary propagation algorithms are implemented yet. Further, subsequent analysis of whether an impact is present or not must still be carried out manually, although presumably reduced to a smaller amount of potential impacts. In order to decrease the *impact set* further, we propose the integration of design time data flow analyses such as the one proposed by Seifermann et al. [62]. Thereby, uncertainties could be propagated along identified data flows in addition to the structural propagation. This could make it possible to sort out wrongly affected elements, i.e. to decrease the *impact set*. Another limitation is that the assessment of the uncertainty impact is still to be conducted manually. However, these limitation is less serious, as it could be remedied by further implementation work, as demonstrated by the exemplary implemented functionality to count the amount of affected elements. An imminent limitation is that the improvement of the impact set through the use of the structural propagation algorithms may remove architectural elements that are actually affected. For instance, Figure 7.8 illustrates two components which are deployed on different servers. Further, the uncertainty type *Is user data processed?* is instantiated and annotated to Component B. *Expert architects* identified that this type has potential impact on Hardware Resources, so in this case on Server 1 and Server 2. Due to the structural propagation algorithms, the UIA would currently only identify Server 2 as indirectly affected element. However, it is conceivable that Component B exchanges confidential data with Component A, which in turn stores it on Server 1. Consequently, this uncertainty also affects Server 1. So in this particular case, an element that is actually affected is not included in the *impact set* when using the UIA. In order to overcome these limitations, we propose again the integration of design time data flow analyses in order to find further affected elements.

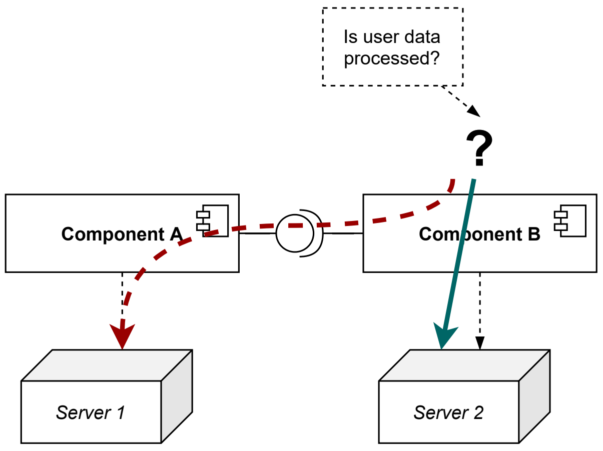


Figure 7.8.: Limitation of the Structural Limitation: Green Path is found whereas the red Path is not.

7.8. Conclusion

In this chapter, we extended the PCM by the capability to represent uncertainty as first-class entity (**RQ2.1**). Based on that, we presented our Uncertainty Impact Analysis (UIA) that enables software architects to manage uncertainties as follows: First, it enables less experienced architects to annotate their architecture with uncertainties based on pre-defined uncertainty types. Second, it automatically propagates instantiated uncertainties (**RQ2.2**). Third, it helps to assess the impact of uncertainties. We introduced the stakeholders roles of the *expert architect* and the *user architect* to differentiate between experienced and less experienced architects.

Beside an overview of the technical implementation, we explained how *expert architects* can prepare their knowledge about uncertainty types and their propagation effects in order to make it accessible for the *user architects.* Further, we explained how *user architects* can use the UIA to annotate software architectures with uncertainties, and how the analysis supports the uncertainty management process. In addition, we described how the UIA uses structural propagation algorithms to determine the *impact set* for given uncertainties. In doing so, we explicitly addressed the schematic structure of two algorithms. Lastly, we discussed the termination of the algorithms, the differentiation to the KAMP approach, as well as current limitations and possible improvements.

8. Evaluation

In this chapter, we evaluate our approach. The evaluation is twofold. In the first part, we evaluate our *uncertainty template* including the proposed ADD- and uncertainty-related categories. In the second part, we evaluate our Uncertainty Impact Analysis (UIA). To structure our evaluation process and to minimize the risk of "collecting unrelated, meaningless data" [6], we use a Goal-Question-Metric (GQM) plan [6]. The approach is based on the assumption that specifying the goals first contributes to evaluating in a "purposeful way" [5]. To that end, Basili et al. [5] propose a framework that consists of three levels: First, goals need to be defined on the conceptual level. Thereupon, questions are defined on the operational level to "characterize the object of measurement" [5]. Finally, metrics are derived to (quantitatively) answer the questions, and to evaluate the accomplishment of the predefined goals. The structure of this chapter is based on Hahner [29].

Section 8.1 presents the goals and questions. The evaluation data is outlined in Section 8.2. We present the evaluation design in Section 8.3, which is used to answer the questions. In Section 8.4 we present and discuss the evaluation results which is followed by the discussion of the threats to validity in Section 8.5. Section 8.6 outlines the assumptions and limitations of our approach. The evaluation results are summarized in Section 8.7. This chapter is concluded by Section 8.8 which provides information on the availability of the evaluation data in order to reproduce the evaluation.

8.1. Goals and Questions

The aim of this chapter is to evaluate whether the research questions (**RQs**) from Section 1.3 have been answered satisfactory. In **RQ1**, we ask about the impact of uncertainties on the confidentiality of a system at design time, hence the impact on software architectures. This includes the definition of categories to categorise types of uncertainties and a template to illustrate the categorisation. Further, **RQ1** also asks about how the template can be used to derive and asses the impact of uncertainty types on software architectures. In **RQ2**, we ask about how to support the annotation and propagation of uncertainties in software architectures. We derive the following evaluation goals based on the research questions:

G1 **Structural Quality**: The *uncertainty template* shall provide a high structural quality regarding the categories and the options.

G2 **Applicability**: The *uncertainty template* shall enable the derivation of uncertainty types and their impact on software architectures based on existing ADDs.

G3 **Purpose**: The *uncertainty template* creates awareness and enables the reuse of knowledge while being more precise than existing approaches.

G4 **Usability**: The Uncertainty Impact Analysis (UIA) simplifies the annotation and propagation of uncertainties by reducing the required amount of expertise and manual effort.

G5 **Functionality**: The UIA is capable to support the uncertainty annotation and propagation.

G6 **Accuracy**: The UIA finds the *affected set*, albeit over-approximated.

8.1.1. G1 - Structural Quality

G1 evaluates the structural quality of the template, including the categories and their options. Bedford [8] provides a collection of principles that can be used to evaluate classification schemes, such as the categories we provide within this thesis. In general, a high structural quality promotes better comprehension, prevents misunderstandings and reduces the risk of incorrect categorisation. To that end, the authors stress the importance that "no two categories should overlap or should have exactly the same scope and boundaries" [8]. We refer to that as *orthogonality* (**Q1.1**). This concept allows an individual and independent classification of an uncertainty according to each category at a given point in time. According to Bedford [8], a classification schema is usually defined as hierarchy from top to bottom. Sub-categories (i.e. our options) need to be defined "in the light of its parent" [8], so that the intention of the options for each category increases while their extension decreases [8]. Fernandez and Eastman [25] refer to that as *generalization* and *specialization* between different *levels of abstraction* (**Q1.2**). Options at the same level of abstraction enable the holistic division of a respective category which promotes to evaluate their exhaustiveness (**Q1.3**) and their differentiation (**Q1.4**). On the one hand, the options of a category should be exhaustive enough that relevant uncertainties and ADDs can be classified with at least one. On the other hand, they should be differentiable enough so that each uncertainty and each ADD can be classified with only one option at most. This supports the classification process as it reduces ambiguity. To evaluate **G1**, we define the following questions:

Q1.1 Are the categories orthogonal, in terms of their statement of impact?

Q1.2 Are the options defined at the same level of abstraction?

Q1.3 Can uncertainties and ADDs be described with at least one option for each category?

Q1.4 Can uncertainties and ADDs be described with at most one option for each category?

8.1.2. G2 - Applicability

G2 evaluates the applicability of the uncertainty template. We discuss its ability to derive and categorize types of uncertainties from existing ADDs (**Q2.1**) as well as its ability to derive their impact on software architectures (**Q2.2**). Otherwise, the template would not contribute to the overall objective of the thesis. To evaluate **G2**, we define the following questions:

Q2.1 Can the uncertainty template be used to derive and categorize uncertainty types based on existing ADDs?

Q2.2 Can the uncertainty template be used to systematically derive the impact of uncertainty types on software architectures?

8.1.3. G3 - Purpose

Having shown that the template is structurally sound (**G1**) and also practically applicable (**G2**), we now evaluate in **G3** that it has a purpose. According to Bedford [8], evaluating the purpose of a contribution is one of the main goals of an evaluation. This is in line with the general view that (scientific) contributions should always serve a purpose. Without a purpose, our *uncertainty template* does not differentiate from existing approaches. Specifically, we demonstrate that the template increases reusability of knowledge and also creates awareness (**Q3.1**). Reusability increases the benefit, as knowledge once collected is made more easily accessible to others. Awareness, on the other hand, enables people to become aware of situations that would otherwise remain hidden from them. The evaluation of the purpose is to show in particular why related approaches are not sufficient for our specific use case and why our approach is more precise (**Q3.2**). To evaluate **G3**, we define the following questions:

Q3.1 Are uncertainty types once identified relevant for multiple scenarios within the same domain?

Q3.2 Is the classification of uncertainties at the architectural level more precise with our template than with existing approaches?

8.1.4. G4 - Usability

Goal **G4** evaluates the usability of the UIA. First, we consider the knowledge software architects require to use our analysis (**Q4.1**). Here, we distinguish between required domain concepts necessary to conduct a manual analysis, an analysis from an expert's point of view and from a user's point of view. Next, we evaluate to what extent the UIA can reduce the necessary effort in determining the impact of uncertainties (**Q4.2**). If this is not sufficiently reduced, the additional effort caused by the use of UIA could compromise the usability. To evaluate **G4**, we define the following questions:

Q4.1 How much and which knowledge is required to use the UIA in order to annotate software architecture with uncertainties and to determine their impact?

Q4.2 Can the UIA reduce the set of model elements that must necessarily be considered when analysing the propagation effects of uncertainties?

8.1.5. G5 - Functionality

G5 evaluates the functionality of the Uncertainty Impact Analysis. In doing so, we evaluate whether it can support the annotation of uncertainty, as well as the propagation of uncertainty (**Q5.1**). Both capabilities are key characteristics of our approach. Consequently, they should be supported by our analysis as otherwise, the analysis would not contribute to the overall goal of this thesis. To evaluate **G5**, we define the following question:

Q5.1 Can the UIA support uncertainty annotation and uncertainty propagation?

8.1.6. G6 - Accuracy

In **G6**, we evaluate the quality of the propagation results, i.e. the accuracy of the UIA. To do this, we examine the precision and completeness of the *impact set* calculated by the UIA (**Q6.1**). A precise estimate is to be preferred to an excessive overestimation. Nevertheless, it is more important that the calculated *impact set* is complete, i.e. it includes the elements which are actually affected. More technically, the *affected set* should be a *true* subset of the calculated *impact set* so that software architects can retrospectively extract all the elements actually affected from the set of calculated elements. To evaluate **G6**, we define the following question:

Q6.1 How precise and comprehensive is the uncertainty propagation?

8.2. Evaluation Data

This section presents the evaluation data that is used throughout this evaluation.

8.2.1. Uncertainty Template - Reference Set

Throughout this thesis, we collect a variety of uncertainties which we iteratively use to verify the suitability of the current state of our *uncertainty template*. We refer to this collection as *reference set*. More specifically, we use the reference set to iteratively test the suitability of the defined categories and their options. We iteratively apply our *Uncertainty Type Derivation Approach* (cf. Section 6.3.1) on sources such as [56, 31, 21, 28] to collect ADDs and other uncertainties with a foreseeable impact on the confidentiality. This iterative process allows us to both, incrementally define new categories as well as to improve and refine existing ones. Apart from the limitation to CBSA with a focus on confidentiality, the reference set is not designed for any specific domain of interest. Therefore, the uncertainties found could potentially be relevant across domains, which we believe they are. Since the reference set is used to create and refine the categories, it is only of limited use for evaluating the categories. Table A.1 sketches the reference set by using the *uncertainty template* notation we propose in Chapter 6. In total, the reference set consists of 12 ADDs (RA1 - RA12) and 24 uncertainties (RU1 - RU24).

8.2.2. Case Study: Corona Warn App

The Corona Warn App (CWA) serves as case study for the evaluation whenever applicable [40]. It is characterised by the following qualities:

1. The CWA is designed according to the CBSE paradigm, which is the underlying principle of our approach
2. The CWA is an open source project. Design documents are freely accessible [68].
3. The CWA is a real system in operation which was releases at June 16, 2020 [33] and as of January 12, 2022 has been downloaded by 40.2 million users [39].
4. The aim of the project is to preserve confidentiality by applying high data protection standards.
5. It was already utilised as case study in a related work, including a detailed review of all design documents [71].

Due to these characteristics, this case study is particularly suitable for our evaluation, because i) The component-based approach allows us to model the software architecture with our ADL - the Palladio Component Model ii) The available documentation [68] and the previous work [71] can serve as sources for our *Uncertainty Type Derivation Approach*, i.e. to collect ADDs and uncertainties iii) It is a contact tracing app which has high confidentiality requirements, so confidentiality is in the focus which is in line with our focus.

8.2.2.1. Software Architecture of the CWA

Unfortunately, there is no available software architecture model of the CWA that shows all the necessary components together and at the required level of abstraction. Furthermore, modelling with the PCM requires the definition of several views (cf. Section 2.5) which is not in line with the design documents provided by the CWA documentation [68]. Consequently, it was necessary to review the available design documents in order to extract the necessary information to create a *Palladio-based* version of the CWA. The modelling follows the *General Model Theory* principles *mapping*, *abstraction* and *pragmatism* proposed by Stachowiak [65]. In particular, the parts of the architecture that we do not consider relevant are not modelled, i.e. we *abstract* from the architecture of the CWA. The architecture model is illustrated by Figure A.1.

8.2.2.2. Uncertainty Template based on CWA

Table A.2 sketches the identified types of uncertainties which are based on the CWA case study. It uses the notation we proposed in Chapter 6, i.e. the *uncertainty template.* Although the filled *uncertainty template* presented in Table A.2 is based on CWA-related sources only, we assume that it is suitable for similar CBSA, specifically in the domain of Contact Tracing Systems (CTS). As the CWA focuses on maintaining confidentiality, the template should be relevant for other systems with the same focus. In total, the template consists of 22 ADDs (CA1 - CA22) and 28 uncertainties (CU1 - CU28).

8.2.2.3. Collections of Instantiated Uncertainties

The CWA is a real-world case study, which has already passed the design time to a large extent. Only minor features are still being added. Since CU1 - CU28 are based on decisions made and other problems solved, the CWA architecture is free of instances of these types. Nevertheless, for evaluation we need to instantiate uncertainties based on the types from Table A.2 and assign them to concrete elements of the CWA architecture illustrated by Figure A.1. In other words, we deliberately bring uncertainty back into the architecture in order to investigate its possible impact. In the following, we briefly present two collections of instantiated uncertainties used in the course of this evaluation. Both collections are created using our UIA, thus there are available as instantiated Uncertainty models in our data set [10].

Collection 1: The first collection is an unsorted list of one instantiated uncertainty per type. So, based on the types CU1 - CU28 we instantiate the uncertainties ICU1 - ICU28, where ICUx is based on the type CUx. The elements affected are randomly selected and only match the expected type. Furthermore, the instantiations do not follow any principle and do not correspond to any realistic use case. This collection is therefore probably unsuitable for analysing the impact of uncertainties. Nevertheless,it instantiates each of the 28 available uncertainty types so that it can be used to illustrate which types of uncertainties can be instantiated and propagated. This is not the case with *Collection 2* as presented in the following.

Collection 2: The second collection is divided into four scenarios, with each scenario corresponding to what we consider a more realistic use case of how uncertainties might exist. Combined, the four scenarios instantiate all types that can also be propagated using the current state of the UIA. In particular, no instances of types CU1, CU4, CU6, CU12, CU22 and CU26 are included, as the propagation rules for these are currently still missing. This limitation of the evaluation data is unfortunately necessary, as we are not able to analyse uncertainties that cannot be propagated. Since the scenarios cover the entire CWA-based template except for the excluded types, they are at least not less meaningful than randomly instantiated uncertainties. The basic idea, however, is that scenarios can help to make the evaluation more understandable, as they provide a certain structure. Therefore, we assume the scenarios to be more suitable to evaluate our analysis.

The scenarios are as follows: *Scenario 1* affects mostly the CoronaWarnAppServer, including its hardware resources, communication components and interfaces. It includes the uncertainties S1_1 - S1_7. The basic idea is that one component is still subjected to uncertainties whereas the remaining components are not. This is the case, for example, when one component is still in planning / development while the remaining components are already implemented. *Scenario 2* refers to uncertainties with regard to deployment locations, the kind of storage used, and the data to be stored. It includes the uncertainties S2_1 - S2_7. In this scenario, the majority of the design decisions are already taken but the resource environment is still subjected to uncertainty. *Scenario 3* refers to uncertainties with regard to user behaviour, input data and user authentication. It includes the uncertainties S3_1 - S3_8. The idea here is that the interfaces of the system boundary are not fully defined. *Scenario 4* contains uncertainties that can be assigned to the system rather than to concrete elements. It includes the uncertainties S4_1 - S4_4. The purpose of this scenario is to analyse uncertainties that could have a system-wide impact.

8.2.3. Uncertainty Types of Related Approaches

We collect types of uncertainties from related approaches, including how they are categorised. The aim is to categorise these uncertainties in the course of the evaluation with our approach. Yet, our focus is exclusively on the classification of uncertainties at design time, in particular those with an impact on the software architecture. Uncertainties specifically for other areas, such as Self-adaptive System (SAS), are out of scope. Therefore, we filter the collected ones based on this criterion. Table A.3 illustrates a collection of uncertainties based on the work of Perez-Palacin and Mirandola [52].

Unfortunately, it is not possible for us to extract further collections from other works, as either no specific types of uncertainties are explained and categorised, or they cannot be used for our purpose. For instance, Hezavehi et al. [31] propose a taxonomy including categories such as *location, nature, spectrum, emerging time* and *source*. Furthermore, the authors name some exemplary uncertainties, which they unfortunately only classify according to the category *source*. The other categories are ignored. Cámara et al. [16] use the taxonomy provided by [31] to categorize an exemplary collection of uncertainty types. This collection consists only of uncertainties specifically for SAS, such as *"What is the range of a possible sensor value?"* or *"What is the response time of the sensor?"*. Ramirez et al. [54] propose a taxonomy based on the category *mitigation techniques* and on the three different levels *requirements time*, *design time* and *runtime*. However, the uncertainties mentioned as examples are rather process-related and cannot be transferred to software architectures in a reasonable way. For instance, the authors propose uncertainties such as *"Are there any missing or ambiguous requirements?"*, *"Are there unexplored design alternatives?"* or *"Is there a lack of numerical precision in a measurement?"*.

8.3. Evaluation Design

In this section, we explain how we intend to answer the previously defined questions. We discuss the evaluation for each goal separately. Wherever possible, we use the CWA case study and additional information from the past section to carry out the evaluation.

8.3.1. Preface

The original GQM approach focuses on defining metrics that can be used to quantitatively answer the questions and thus demonstrate the fulfilment of the initial goals. However, it is not always possible to define valid metrics, due to several reasons. First, the evaluation of classification schemes (such as our *uncertainty template*) is classically argumentative in nature. Apart from conducting user studies, such schemes are often evaluated in terms of whether generally applicable principles have been adhered to [8]. Furthermore, the uniqueness of our approach and the lack of comparable evaluation data makes it difficult to define valid quantitative metrics. For this reason, the following evaluation is often based on argumentation. It is backed up wherever possible with references to related literature and to the data we collected in Section 8.2.

8.3.2. G1 - Structural Quality

The first evaluation goal is the investigation of the structural quality of the *uncertainty template*. In Chapter 5 we defined various categories and options to categorize ADDs and uncertainties. We evaluate these both argumentatively and by applying them to a case study. In both cases, we use a selection of principles to evaluate classification schemas provided by Bedford [8].

8.3.2.1. Q1.1 - Orthogonality

This question aims to evaluate the orthogonality of the categories in terms of their statement of impact. By definition, orthogonality means that no two categories overlap so that the *message* of one category cannot be generated by the concatenation of (several) others. Since it is not practical to examine all possible combinations of categories, we proceed as follows: First, we justify why we can make a division into two independent, non-overlapping areas. Each category is an exclusive part of one of these two areas. This reduces the set of potentially orthogonal categories to be examined in advance, as this subdivision is intended to prevent overlaps of categories from one of these areas at a time.

As experts, we know best which categories could be orthogonal to each other. Therefore, we can enumerate those categories that could potentially be orthogonal but which are not (with justification) and which are (with explanation why this is not critical). Wherever necessary, we reference categorised uncertainties and ADDs from both, our reference set (see Table A.1) and the CWA case study (see Table A.2). Orthogonality can be recognised, for example, by the fact that two options of a category always occur together. Here again, the available evaluation data can help us identify or dismiss potentially orthogonal categories. Strictly speaking, the result is not universally valid since we do not compare all possible combinations of categories.

8.3.2.2. Q1.2 - Level of Abstraction

In question **Q1.2**, we evaluate the level of abstraction of the respective options. A definition at the same level of abstraction improves understanding and in particular promotes answers to the next two questions. Similar to the previous question, we evaluate this argumentatively. Therefore, we assess each category independently, i.e. we enumerate those categories whose options are not defined on the same level of abstraction. In those cases, we argumentatively explain why this is not critical. Wherever possible, we refer to related literature to back up our statements.

8.3.2.3. Q1.3 - Exhaustiveness

Question **Q1.3** evaluates the exhaustiveness of the respective categories which is another quality principles for good categorization schemas [8]. In other words, we evaluate if each uncertainty and ADD can be described with at least one option. What is important here is that we focus on exhaustiveness with regard to uncertainties in software architectures. In particular, the options do not strive for general exhaustiveness, i.e. being able to classify uncertainties in other domains such as SASs.

To evaluate the exhaustiveness, we proceed as follows: Similar to the previous question, we discuss argumentatively for each category separately why the options are likely to be exhaustive. Thereby, we refer to the evaluation data to support our reasoning. More specifically, we highlight those categories where it is not always possible to select a suitable option. In order to make a quantifiable assessment, we present for each category how many uncertainties could be classified with at least one option. Here, we refer to the uncertainties from our reference set (see Table A.1) and the CWA case study (see Table A.2). At this point we would like to emphasise that we do not want to evaluate the content of the two filled *uncertainty templates*, but the categories and their options. In particular, we do not want to show whether the identified uncertainties and ADDs are correctly categorised.

8.3.2.4. Q1.4 - Differentiability

In Question **Q1.4** we discuss the differentiability of the options of the respective categories which is another structural quality of classifications according to Bedford [8]. Whereas the exhaustiveness (**Q1.3**) aims to asses whether uncertainties and ADD can be classified with at least one of the available options, differentiability refers to the classification with at most one of the available options. This questions evaluates two sub-aspects: Firstly, the options should not overlap (i.e. orthogonality at option level) which means that options and their definitions are clearly delineated. Secondly, it has to be evaluated to what extent several options could be considered when categorising uncertainties. Altogether, we evaluate whether or not it is clear which option to choose for a certain category.

While the evaluation of exhaustiveness is largely based on objective, logical reasoning, this does not apply to differentiability. This is because reasoning about an unambiguous classification is very subjective, i.e. one person might categorise one entity with a different option than another. In order to obtain meaningful evaluation results, we proceed as follows: For each of the categories we argue, if possible, why a unique classification is probably achievable. If necessary, we identify ambiguously interpretable categories to the best of our knowledge and belief. Here, we refer to the evaluation data whenever possible. In order to make a quantifiable assessment, we present for each category how many uncertainties could be clearly classified and how many are subjected to ambiguity. Here, we refer to the uncertainties from our reference set (see Table A.1) and the CWA case study (see Table A.2). As with **Q1.3**, the aim is to evaluate the categories and their options, not to evaluate the content of the two completed templates.

8.3.3. G2 - Applicability

The second goal of this thesis is to demonstrate the applicability of our *uncertainty template*. First, we discuss its ability to derive and categorize uncertainty types based on existing ADDs. Second, we evaluate its ability to systematically derive the impact of uncertainty types on software architectures. In doing so, we follow the approach that we defined in

Section 6.3.1 and illustrated in Figure 6.3. We therefore demonstrate the applicability by applying it to the CWA case study. The ADDs, that serve as the basis for the approach, are derived from a collection of confidentiality requirements presented by [71]. In his work, Tobias [71, 70] structurally examines the design documents of the CWA [68] in order to extract the confidentiality requirements defined therein. In addition, we examine the CWA design documents [68] by ourselves to gather more design decisions and other uncertainties.

8.3.3.1. Q2.1 - Uncertainty Type Derivation and Categorization

In question **Q2.1**, we evaluate the ability of the template to enable the derivation and categorization of uncertainties based on existing ADDs. To that end, we exemplarily apply *Step 1* to *Step 8* of the approach defined in Section 6.3.1 to the CWA case study. This includes to derive the impact on relationships between the evaluated ADDs as well as the extracted uncertainties.

8.3.3.2. Q2.2 - Uncertainty Type Impact Derivation

Question **Q2.2** evaluates the ability of the template to enable the derivation of the impact of uncertainty types on the software architecture, i.e. their direct and indirect impact on architectural elements. To that end, we exemplarily apply *Step 9* and *Step 10* of the approach defined in Section 6.3.1 to the CWA case study.

8.3.4. G3 - Purpose

The previous goals aim to discuss the structural quality of the template as well as its applicability. Now, we evaluate to what extent the template also has a purpose, as this is one of the main objectives of evaluations in general. The focus is on evaluating to what extend the *uncertainty template* is able to increase the reuse of knowledge and whether or not it can create awareness. We also want to evaluate whether or not it enables a more precise classification of uncertainties at the architecture level compared to existing approaches that deal with uncertainties.

8.3.4.1. Q3.1 - Reuse of Knowledge and Creation of Awareness

Question **Q3.1** asks for the relevance of uncertainty types for multiple scenarios. In particular, it is to be evaluated to what extent once identified types of uncertainty promote the reusability of knowledge and also create awareness. This question shall be discussed once argumentatively and once by comparing our self-defined reference set to the one we extracted based on the CWA documentation [68]. The reference set is illustrated in Table A.1 whereas the CWA-based template is illustrated by Table A.2. Although the filling of the reference set was completed before starting on the CWA-based template, we claim that both templates are independent of each other. To be precise, we tried to fill the CWA-based template as far as possible without the previously collected knowledge and only based on the existing sources [68, 71]. In order to be able to compare the two, it was necessary to adjust the terminology in some cases. It was ensured that neither the content nor the meaning was changed. Only an alignment of the names has taken place. For instance, *"Data to be stored"* and *"Data Persistence"* is aligned to *Persistence of Data (What)* (RA4, CA2).

8.3.4.2. Q3.2 - Relevance

This question evaluates the purpose and relevance of the template in general. More specifically, we discuss the benefit of our template in terms of its ability to do something that existing (related) approaches are not able to do - the classification of uncertainties at architectural level, i.e. precisely in CBSAs. To do so, we proceed as follows. First, we argumentatively discuss the relevance by comparing our approach with available related approaches. In particular, we refer to the suitability of existing categories. For this purpose, we discuss how many of our categories could be adopted unchanged, how many had to be adapted for our purposes and how many are entirely new. A large number of adapted, or completely new categories allow conclusions to be drawn about the unsuitability of existing approaches. To achieve a fair comparison with other approaches that deal with uncertainties, we only consider our uncertainty-related categories as presented in Section 5.2.2.

In addition, we also compare uncertainties categorised with our template with existing approaches. For this we apply existing classification approaches to the uncertainty types which are based on the CWA case study. We consider the approaches of Perez-Palacin and Mirandola [51], Hezavehi et al. [31] and Bures et al. [14] to be the most advanced and most relevant approaches in this field of research which is why we take these into account. Thereby, we discuss to what extent our template classifies uncertainty types more precisely than existing approaches. We exemplarily discuss this based on a selection of uncertainty types, but also provide an overview with regard to the entire CWA-based uncertainty template.

Following the other direction, we use the uncertainties extracted (and categorised) by Perez-Palacin and Mirandola [52] (see Table A.3) and categorize them with our approach. Again, we discuss to what extent our template classifies them more precisely than the one proposed by [52]. In both cases, the comparison of our uncertainty template with existing approaches is not a comprehensive review. First, we only compare based on a selection of uncertainty types which we do not consider to be exhaustive in the first place. Further, a fair comparison is not always possible because existing approaches do have another purpose, i.e. they aim to categorize uncertainties in different domains such as SAS [52, 31] or for access control [14].

8.3.5. G4 - Usability

We start the evaluation of the UIA with regard to its usability. To that end, we discuss the required knowledge that is necessary to analyse the impact of uncertainties on the architecture, from their extraction up to their propagation and assessment. We further compare the amount of architecture elements that must be considered when examining the propagation of uncertainties with both, a manual approach and our approach.

8.3.5.1. Q4.1 - Required Knowledge

Question **Q4.1** asks about the required knowledge to analyse uncertainties and their propagation effects using the UIA compared to a manual uncertainty analysis. As it is hard to quantify knowledge, we focus on *which* knowledge is required and discuss the amount qualitatively [29]. Therefore, we gather different concepts and tasks which need to be understood and executed when analysing uncertainties and their impact. Ideally, using the UIA reduces the required knowledge, as fewer tasks need to be performed and thus understood. For this reason, the concepts presented not only describe areas of knowledge, but also tasks to be carried out. We differentiate between the manual uncertainty impact analysis and the use of our analysis. The latter is further divided into the view of *expert architects* and *user architects*, since *expert architects* perform tasks which are required for *user architects* to be completed. Although *expert architects* do not necessarily use the UIA, but rather provide the input, we decide to include their required knowledge and tasks to achieve a fair comparison. As *manual analysis* we therefore refer to the procedure of manually extracting types of uncertainties, deriving their impact, as well as their instantiation, propagation and assessment, i.e. without the help of the uncertainty template, the type derivation approach and the UIA. Based on the knowledge about uncertainty type derivation, uncertainty propagation and assessment gathered in Chapter 6, and the required knowledge to use the UIA in Chapter 7, we gather the following domain concepts and tasks:

C1 *Uncertainties in CBSAs*, i.e. understanding basic principles of uncertainty, such as the *cone of uncertainty*.

C2 *Palladio*, i.e. knowing how to use the PCM to model software architectures.

C3 *Uncertainty Type Extraction*, i.e. collecting potential types of uncertainties which are relevant for a certain domain of interest.

C4 *Uncertainty Type Categorization*, i.e. categorizing identified types of uncertainties according to our categorization approach.

C5 *Uncertainty Type Impact Derivation*, i.e. identifying on which types of architectural elements uncertainty types can have an impact.

C6 *Uncertainty Template Model Creation*, i.e. transferring the knowledge into the UIA

C7 *Uncertainty Impact Analysis (UIA)*, i.e. knowing how to use the UIA.

C8 *Uncertainty Annotation*, i.e. annotating software architectures with uncertainties.

C9 *Uncertainty Propagation*, i.e. determine architectural elements where uncertainties might have an impact.

C10 *Uncertainty Impact Validation*, i.e. using context information to determine if impact actually exists

C11 *Uncertainty Impact Assessment*, i.e. assessing the impact of uncertainties with regard to the confidentiality.

8.3.5.2. Q4.2 - Amount of Elements

This question discusses the amount of architectural elements that must necessarily be considered when examining the propagation effect of uncertainties. Without our approach, each architectural element of a software architecture must be considered as it might be potentially affected by an uncertainty. In contrast, the *uncertainty template* and UIA provide a collection of potentially affected architectural elements which we refer to as *impact set*. Based on Rostami et al. [58], we define the following two metrics:

$$r_{template} = \frac{ImpactSet_{template}}{n} \qquad (8.1)$$

$$r_{uia} = \frac{ImpactSet_{uia}}{n} \qquad (8.2)$$

Metric 8.1 is defined as the ratio, $r_{template}$, of the amount of potentially affected elements to the number of all elements n where $ImpactSet_{template}$ corresponds to the amount of elements of the *impact set* calculated using the *uncertainty template* only. Similarly, metric 8.2 describes the ratio, r_{uia}, of the amount of the potentially affected elements according to the UIA to all possible elements. Elements that can be reached via several paths are only listed once in the impact set.

The interpretation of the metrics is as follows: The minimum value is $\frac{x}{n}$ where x corresponds to the amount of propagated uncertainties. This indicates that no propagation occurs and no further elements have to be analysed. The maximum value is $\frac{n}{n} = 1$, indicating that each element of a software architecture needs to be analysed. Any value less than one indicates that fewer elements need to be examined after the propagation is conducted. In order to obtain valid evaluation results, the produced *impact sets* must be comprehensive, i.e. the actually affected elements should be part of the respective *impact sets*. Otherwise, the ratio could be low, falsely suggesting that only a small number of elements need to be considered. Evaluating the comprehensiveness, i.e. the recall, is covered by **Q6.1**. Consequently, when assessing the evaluating results of this question, it is necessary to include the results of question **Q6.1**.

The evaluation is performed using the scenarios presented in Section 8.2. For each scenario, we calculate the two metrics independently in order to evaluate to what extend using our approach reduces the amount of architectural elements to be analysed. These scenarios consist only of an exemplary selection of uncertainties. It is therefore not a comprehensive review, as we do not consider our selection to be comprehensive in the first place. Further, the amount of uncertainties per scenario influence the results so that no generalization is possible here. The aim of this question is rather to show whether our approach is suitable for reducing the number of elements, and not by how much.

8.3.6. G5 - Functionality

Besides the usability, we also aim to evaluate the functionality of the UIA. To this end, we discuss whether the UIA is capable to support the annotation and propagation of uncertainties in software architectures.

8.3.6.1. Q5.1 - Uncertainty Annotation and Propagation

This question evaluates whether the functionality of the UIA is sufficient to annotate and propagate uncertainties in software architectures. A prerequisite for the uncertainty annotation is the representation of uncertainties in software architectures. *"Represent"* in this context means the following: Uncertainties are to be modelled as first class entities. This includes, in particular, the possibility of providing them with characteristics such as a certain type or the architectural element they affect. This enables to attach uncertainties to the elements where they have a direct impact, which we refer to as uncertainty annotation. For this purpose, common concepts of CBSE are to be supported, so that different architecture element types can be annotated.

Uncertainty propagation refers to the core capability of (automatically) calculating the architectural elements on which instantiated uncertainties have a indirect impact. As our approach uses structural propagation rules, each combination of starting and ending architectural element type should be supported, otherwise uncertainties may not be annotated correctly resulting in faulty propagation results.

Since there is no comparable tool and the literature often only provides coarse-grained functional requirements (e.g. there is a "lack of systematic approaches for managing uncertainty" [32]), it is not possible to check our approach against an eligible list of requirements. We therefore evaluate the functionality quantitatively based on our CWA case study. To that end, we use the uncertainty types CU1-CU28 and their instances ICU1-ICU28 to evaluate whether or not the uncertainties can be annotated to the element to which they actually belong. Further, we evaluate if the uncertainty analysis is capable of propagating the instantiated uncertainties.

8.3.7. G6 - Accuracy

The last evaluation goal evaluates the accuracy of the UIA with regard to the precision and completeness of the propagation results (**Q6.1**), usually referred to as *precision and recall*. We conduct the evaluation using the CWA case study. In particular, we calculate the *precision and recall* for each of the four scenarios.

8.3.7.1. Q6.1 - Precision and Recall

Question **Q6.1** evaluates the accuracy of the UIA from two perspective: the functional accuracy and the accuracy after incorporating context-related information. First, we discuss whether or not the UIA identifies the architectural elements it is supposed to find. We refer to that as functional accuracy, as we evaluate the functional correctness of the propagation algorithms. In particular, it is not a question of whether the elements found are actually affected by uncertainty, but whether they have been correctly identified as potentially affected. Therefore, we first manually identify which architectural elements are structurally affected by uncertainty propagation according to the underlying structural propagation rules which form the basis for the implemented algorithms. Then, we conduct the automatic propagation using the UIA and compare the results to the manual analysis. We classify the results of the UIA for the uncertainties of each scenario as follows: *true-*

positive (structurally affected element included), *true-negative* (structurally not affected element not included), *false-positive* (structurally not affected element included) and *false-negative* (structurally affected element not included). Based on that, we derive the metrics of binary classifications [53]:

$$Precision = \frac{\textit{true-positive}}{\textit{true-positive} + \textit{false-positive}} \qquad Recall = \frac{\textit{true-positive}}{\textit{true-positive} + \textit{false-negative}}$$

In a second step, we discuss the accuracy of the propagation results after incorporating context-related information. For each scenario we proceed as follows: We manually assess the elements that are actually affected by the uncertainties by using context information such as design decisions, requirements or information about the data flow. Thus, we determine the *affected set* manually. Afterwards, we conduct and compare the UIA results, i.e. the *impact set* with the previously (manually) defined *affected set*. Again, we apply the *precision and recall* metric for each scenario as follows: *true-positive* (element in *affected set* and in *impact set*), *true-negative* (element not in *affected set* and not in *impact set*), *false-positive* (element not in *affected set* but in *impact set* and *false-negative* (element in *affected set* but not in *impact set*). We briefly explain for each element the reason for its classification.

Please note that both perspectives represent fundamentally different aspects of accuracy and, above all, are differently "difficult" to determine. The first perspective is tedious to determine manually, but it is structured and unambiguous. One simply needs to apply the structural propagation rules manually to examine which elements the UIA should identify. The second perspective is quite the opposite: Here it is necessary to manually determine for each uncertainty which elements it actually affects. This is not based on structured rules, but on knowledge and experience and is above all characterised by subjectivity. Hence, different experts could therefore obtain diverging results given the same initial situation. However, as our case study is sufficiently small, and as we have the expertise to analyse uncertainties, we are confident that we can accomplish this task satisfactorily.

8.4. Results and Discussion

In this section, we show and discuss the results of our evaluation in the order of the GQM plan we elaborated in the previous sections.

8.4.1. G1 - Structural Quality

G1 evaluates the structural quality of the *uncertainty template*. Therefore, question **Q1.1** evaluates the orthogonality of the categories, while question **Q1.2** evaluates the level of abstraction of the options. Further, the options are evaluated with regard to their exhaustiveness (**Q1.3**) and differentiation (**Q1.4**).

8.4.1.1. Q1.1 - Orthogonality

This question is twofold. We begin by arguing why we can consider the architectural-related categories and the uncertainty-related categories separately. This allows us to focus on potential orthogonality only within the two areas. The argumentation at this point is almost trivial, as the respective categories have a different purpose. In the first case, we categorize ADDs, whereas in the second case we categorize uncertainties. However, this argument is further supported by the sources of the respective categories as described in Section 5.2.1 and Section 5.2.2. The sources of the ADD-related categories and the sources of the uncertainty-related categories have no intersection.

Within the architectural-related categories, the categories *Probability of Revisability*, *Possibility of Revisability* and *Costs of Revision* all refer to the revisability of an ADD. While the latter is clearly different from the first two, the first two initially appear partially orthogonal. One might think that the probability is always *unlikely* when something cannot be revised. However, ADD (CA3) indicates the opposite. The point here is the location of persistence, which can change more likely in cloud-based systems such as the CWA, but in principle cannot be changed once data has been (incorrectly) persisted somewhere. We therefore conclude that the architectural-related categories are not orthogonal.

Within the uncertainty-related categories, several pairs seem suspicious with regard to orthogonality: First, *Location* and *Architectural Element Type* both express locality characteristics. It is also noticeable that the following applies for each uncertainty from our evaluation data: If the option *System Environment* is selected for *Location*, the *Architecture Element Type* always corresponds to *Hardware Resource*. For the remaining options in these two categories, there are no other noticeable findings. Even if both categories do not seem to be fully orthogonal to each other, this is not a big issue in our humble opinion. The reason for this is the definition of both categories: While category *Location* is based on view level, category *Architectural Element Type* is on element level within the respective views. Since elements such as interfaces can occur in several views, it is reasonable to distinguish between them. Also, categories *Type* and *Nature* have similarities, as aleatory events can often be described by statistical means. However, CU6, RU10 and RU14 show that this is not always the case. Moreover, both categories are based on work that also lists them (albeit modified) as distinct categories, too [14, 52].

Another conspicuous finding emerges from the examination of categories *Impact on Confidentiality* and *Severity of the Impact*. The evaluation data reveals that a *direct* impact always implies a *high* severity. However, this circumstance is hardly surprising, as we suppose that uncertainties with a direct impact automatically have a high severity. However, for the indirect impact, the choice between *high* and *low* is more than necessary, as the evaluation data show. It is not necessary to assume that this makes it impossible to distinguish between the two categories with regard to their core message. For this reason, partial orthogonality can be accepted for these two categories.

Finally, the evaluation data suggest a further correlation between categories *Root Cause* and *Resolution Time*. If *missing information* is selected, the *Resolution Time* is almost always either *design time* or the temporally preceding *requirements time*. *Assumption* always indicate *deployment time*, *runtime* or *never*. These implications do not apply only to the two uncertainties CU10 and RU19. Although the evaluation data convey a certain orthogonality, we would like to point out that both categories look at two completely different aspects. *Resolution Time* is used to describe the temporal character of uncertainties. *Root Cause* is defined from within the design time (which is an option of *Root Cause*), but describes the overall reason why uncertainty exists. To the best of our knowledge, there are no further categories that seem to be orthogonal.

8.4.1.2. Q1.2 - Level of Abstraction

In this question we deal with the abstraction level of the options. For this purpose, we list hereafter the categories whose options are not clearly defined at the same level of abstraction. Binary classifications such as high/low, yes/no or likely/unlikely are trivially defined at the same level of abstraction. In the following, we therefore only consider non-binary classifications. For the evaluation, we proceed in the order in which categories are presented in Chapter 5.

We start by evaluating the architectural-related categories: The options for category *ADD Class* are based on the taxonomy provided by Kruchten [43]. The author provides another level of abstraction as he combines option *structural decision* and *behavioural decision* under *existence decision*. According to the author, the remaining options *property decision* and *executive decision* are located on the same level as *existence decision*. We have not followed this division of levels, as in our opinion structure and behaviour are two fundamentally different concepts that cannot be assigned to one sub-category. This is in line with Reussner et al. [56]. Since each of the options describes a basic ADD class, we locate them at the same level of abstraction. Category *Amount of Alternatives* is composed of *closed set* and *open set* which both describe the characteristics of a collection of alternatives. Hence, both the options are on the same level of abstraction.

We continue with the evaluation of the uncertainty-related categories: The options for category *Location* correspond to the views defined by the PCM [56]. Therefore, we expect them to be on the same level of abstraction. *Type* is used to differentiate between different types of *Level 1* uncertainties, i.e. *known uncertainties* as defined by Perez-Palacin and Mirandola [52] and Bures et al. [14]. Each option characterizes a different type of uncertainty within the first level. Therefore, we consider them on the same level of abstraction.

Nature is a well established category and defined by various related approaches such as [73, 52, 31, 14, 44, 72]. They all agree that the two options *aleatory* and *epistemic* divide nature in those two areas. *Architectural Element Type* describes possible first-class entities of CBSAs. Solely option *usage behaviour* suggests that is not on the same level as the remaining options. However, as described by Reussner et al. [56], CBSE enables software architects to describe the *usage behaviour* as first class entity in software architectures, which shifts this option to the same level of abstraction as the others.

Category *Manageability* differentiates between *fully reducible*, *partial reducible* and *irreducible*. Since all the options similarly refer to the manageability aspect of uncertainties, it can be claimed that the level of abstraction is the same. Category *Impact on Confidentiality* consists of *none*, *direct* and *indirect* impact. *Direct* and *indirect* represent the existence of an impact whereas *none* represents the absent of an impact. Consequently, this category violates the principle of same level of abstraction. Nevertheless, our research showed that it is necessary to differentiate between *direct* and *indirect* impact, but also to provide the possibility to express that their is no impact. Another solution would be to define two categories: one category indicating the binary decision between impact and no impact (yes/no), and another indicating the type (direct/indirect). However, this complicates the categorisation process, as for no impact the choice of type makes no sense. In order to avoid this problem and at the same time fulfil the previously identified requirements, we have decided to deliberately break the principle of the same level of abstraction at this point.

Regarding category *Resolution Time*, the options *requirements time*, *design time*, *deployment time* and *runtime* refer to potentially finite time spans, whereas in the case of *never* it is questionable whether it is a time span in the true sense of the word. Here we slightly break up the concept of the same level of abstraction for practical reasons. With option *never*, we had to provide an unusual time span which was necessary to characterize the resolution time of some uncertainties, such as CU23, CU27, RU14 and RU17. Nevertheless, the remaining options correspond to common terms of the software development process [36]. Thus, their meaning is widely known and accepted so that we do not assume that the additional option causes misunderstanding.

Root Cause differentiates between *missing information* and *assumption*. Both options are defined from the perspective of the design time and have a rather temporal character to describe the reason why uncertainty prevails. When categorising the uncertainties of the CWA, we observed that neither category is more specific or generalisable than the other. This indicates that both are on the same level of abstraction. The remaining categories are binary, as already explained at the beginning, which is why an evaluation at these points is trivial. Overall, we can satisfactorily conclude that the respective options of the categories (with some less serious exceptions) are defined at the same level of abstraction.

8.4.1.3. Q1.3 - Exhaustiveness

The evaluation for this question is twofold: First, we argumentatively discuss why the options for each category are exhaustive, in terms of the capability to categorize uncertainties in software architecture with, at least, one of the provided options. Second, we demonstrate the exhaustiveness based on our *reference set* and the CWA case study.

We start by evaluating the architectural-related categories: Category *ADD Class* is based on the widely accepted taxonomy of Kruchten [43]. We derive its exhaustiveness based on its recognition and our evaluation data [10]. Category *Amount of Alternatives* consists of *open set* and *closed set*. Possible design alternatives are either fully known by the time software architects make the decision (=*closed set*) or not yet fully known (=*open set*). This binary-like definition leads us to conclude that ADDs must fall into one of the two sets and that the two options are therefore exhaustive. *Probability of Revisability* divides the probability scale holistically in two areas and can therefore be considered as exhaustive. *Possibility of Revisability* is a binary classifier, hence the options are exhaustive per definition. *Costs of Revision* aims to avoid precise quantifications in terms of specific amounts of money or necessary working hours. It is apparent that *high* and *low* are sufficient to say whether a revision is rather expensive or rather cheap. A precise quantification is not possible at this point, as costs can be assessed differently from system to system [57]. The need for a more fine-grained division has not emerged, at least with regard to the examined case study.

We continue with the evaluation of the uncertainty-related categories: *Locations* is based on the four different views on software architectures as defined by Reussner et al. [56]. The division into these views has been evaluated several times, which is why we conclude that uncertainties in software architectures are assignable to at least one of these views. Category *Type* describes uncertainties on the 1st level, i.e. known uncertainties. Option *recognized ignorance* act as kind of fallback, as uncertainties are neither describable with statistical means, not as by developing scenarios. Consequently, we conclude that uncertainties can always be categorized using on of the given options. The exhaustive categorization of the nature in either *aleatory* of *epistemic* is widely accepted by the scientific community [73, 52, 31, 14, 44, 72]. We see no reason to question this. For category *Architectural Element Type* the evaluation data demonstrate that the available options are not exhaustive. For instance, RU1, RU8, CU1, CU23, CU17, CU18 indicate that a specific option is missing - *the system*. Each of the mentioned uncertainties have in common that they are to be assigned to the system, instead of a specific element type. In a previous version of our *uncertainty template*, this option was part of category *Architectural Element Type*. However, we decided to remove it as it violates the level of abstraction (cf. **Q1.2**). Further, the system would include all the other options which could lead to misunderstanding when categorizing uncertainties. For practical reasons, we circumvent this limitation in our UIA by allowing uncertainties to be assigned to the system itself.

Category *Manageability* differentiates between *fully reducible*, *partial reducible* and *irreducible*. Since these options completely divide the theoretical scale of reducibility, any uncertainty should fall into at least one of these three areas. The evaluation data show nothing to the contrary. Uncertainties can either have *direct*, *indirect* or *no Impact on Confidentiality*. As described in the previous section, *direct* and *indirect* represent the existence of an impact whereas *none* represents the absence of an impact. Logically, this is a binary choice and thus exhaustive per definition. The same reasoning applies for categories *Severity of the Impact* and *Resolvable by ADD*. Category *Resolution Time* divides the software development process into several phases. Yet, well known and accepted phases such as *Validation*, *Testing* or *Maintenance* [36] are not part of this category. This is because, for example, testing extends over several already defined phases. To avoid confusion, we decided not to include any further options as in our opinion, the options provided are sufficient to classify uncertainties in software architectures in terms of their resolution time. The evaluation data also indicate that no further option is needed. The exhaustiveness of category *Root Cause* is based on our findings of Chapter 4, where we already discussed why uncertainties in software architectures are caused by either *assumptions* or *missing information*.

In order to consolidate our argumentative evaluation with quantitative results, we analyse for each category how many uncertainties could be classified with at least one option. The *reference set* (see Table A.1) consists of 12 ADDs and 24 uncertainties. Consequently 12/12 and 24/24 indicates the optimum, meaning that the options for the respective category were exhaustive enough to classify the respective ADD or uncertainty. With regard to the CWA (see Table A.2), the optimum values are 22/22 and 28/28, respectively. Table 8.1 reflects the evaluation results we previously collected argumentatively. For each category, the options are sufficiently exhaustive so that uncertainties and ADDs can be categorized with at least one option. The only outlier is category *Architectural Element Type*, for which we already provided an extensive explanation. Finally, we can conclude that the options are sufficiently exhaustive.

Category	**Reference Set**	CWA
ADD Class	12/12	22/22
Amount of Alternatives	12/12	22/22
Propability of Revisability	12/12	22/22
Posibility of Revisability	12/12	22/22
Costs of Revision	12/12	22/22
Location	24/24	28/28
Type	24/24	28/28
Nature	24/24	28/28
Architectural Element Type	**22/24**	**24/28**
Manageability	24/24	28/28
Impact on Confidentiality	24/24	28/28
Severity of the Impact	24/24	28/28
Resolvable by ADD	24/24	28/28
Resolution Time	24/24	28/28
Root Cause	24/24	28/28

Table 8.1.: Exhaustiveness of the Options sorted by Category and Case Study (Interesting Cases highlighted in Colour)

8.4.1.4. Q1.4 - Differentiability

The evaluation for this question is twofold. First, we argumentatively discuss the differentiability of the options for each category. In other words, we explain why a clear choice of one of the available options is probably achievable. Second, we demonstrate the differentiability based on our *reference set* and the CWA case study.

We start by evaluating the architectural-related categories: Category *ADD Class* is based on the taxonomy presented by Kruchten [43]. Although the taxonomy defining this category is well established, we consider the options to be ambiguous. The reason is how you want to classify ADD. Although the definitions of the different classes are rather unambiguous, the usage is not. For instance *"Data Minimization"*(CA9) and *"Logging"*(CA14) have both, a *behavioural* character and a *property* character. CA9 can describe the behaviour of the system, i.e. that data minimization is actually executed, as well as the constraint (=*property*), that data minimization has to be carried out. The same applies for CA14, where the ADD either refers to how logging is performed or that it is required that logging is performed. Either way, each possibility is sound. On the other hand, it is precisely this ambiguity that allows us to clarify the actual meaning of the respective ADD. With regard to CA9, for instance, we decided to categorize it as *behavioural decision* indicating that we refer to the way data minimization is performed where for CA14, we categorized it as *property decision*, indicating that logging is a constraint that need to be fulfilled. So we conclude that the ambiguity in this case actually brings advantage in terms of clarifying the intention.

The remaining architectural categories are, in our opinion, not affected by ambiguity as the respective options are considered to be binary. Again, we do not want to evaluate the content of the templates, but the categories and options. Of course, it is not always clear whether, for example, the costs are *high* or *low*. But that is not the point of this evaluation question. Rather, it is clear what is meant by *high* and *low*, so that architects only need to be able to determine whether the expected costs of the ADD are either *high* or *low* when filling in the template. In general, the choice of the respective option is of course highly dependent on the domain of interest and thus also on the systems to be described. For the remaining categories, there is no ADD in the evaluation data where we notice any issue regarding the selection of the appropriate option.

We continue with the evaluation of the uncertainty-related categories: Similar to category *ADD Class*, determining the *Location* of uncertainties can sometimes be ambiguous due to a similar reason. Depending on how one interprets an uncertainty, it can be assigned to a different location. For instance, *"Which authentication mechanisms are used?"* (`CU16`) and *"What kind of access control is used?"* (`RU6`) both are uncertainties that could be located within the *system behaviour* as well as within the *system environment*. In both cases, the uncertainties could be due to not knowing how authentication and access control are actually implemented (i.e. *system behaviour*) versus not knowing which technological choice has been made (i.e. *system context*). For `RU6`, this could mean that either it is uncertain how RBAC is implemented or whether RBAC or ABAC is chosen. Similar to category *ADD Class*, this ambiguity enables to clarify the actual meaning of the respective uncertainty. Since the options themselves correspond to the established views of the PCM, we claim that the definitions do not overlap.

The usage of category *Type* could be affected by ambiguity if software architects can describe the uncertainty with either statistical means or by defining scenarios. However, our precise definitions of those two options indicate that *statistical uncertainty* is closer to determinism as *scenario uncertainty*. Thus, software architects should always use statistical means if possible so that the ambiguity does not occur. This enables to clearly categorize each of the collected uncertainties in our evaluation data.

Category *nature* is widely used and therefore unambiguous with regard to the definition of *aleatory* and *epistemic*. However, many authors claim that "it is not always easy to clearly distinguish between these categories of uncertainty" [73], as it "may depend on the point of view" [52]. However, we did not encounter any cases in our evaluation data where we found it difficult to clearly identify an option. This is in line with the insights of Esfahani and Malek [21], who differentiate between the theoretical and philosophical debates, and the practical application. In Section 6.2, we argue why uncertainties are likely to (directly) affect multiple *Architectural Element Types* which would introduce ambiguity. Due to our decision to include such uncertainties several times in the template, this form of ambiguity is structurally eliminated. Each option corresponds to a well-defined type of CBSAs, so that confusion can be ruled out at this point.

Category *Manageability* differentiates between *fully reducible*, *partial reducible* and *irreducible*. Since these options completely divide the theoretical scale of reducibility in three, non overlapping areas we consider this category to be unambiguous. Further, evaluation data demonstrates that this also applies for the actual usage of this category as in no point of time, we had trouble to identify which kind of reducibility applies for an uncertainty. *Impact on Confidentiality* is a two-step binary decision. First, software architects must determine whether an uncertainty has an impact on confidentiality or not. In case they identify a potential impact, they must further asses whether this impact is *direct* or *indirect*. Due to the definition, an overlap of the options is therefore impossible. However, the definitions of *direct* and *indirect* could be misunderstood so that is might not always be clear which option to choose.

Severity of the Impact is subjected to the point of view of software architects, especially to the domain of interest for which they complete the template. While the binary decision allows for an unambiguous decision, the actual severity may depend on the software architect's perspective. However, we did not encounter any cases in our evaluation data where we found it difficult to clearly identify an option. *Resolvable by ADD* is a rather structural defined category whose choice is clearly determined by the structure of the template.

Resolution Time is unambiguous with regard to the definitions of the options as they correspond to generally accepted, clearly delineated phases of the software development process. However, it might not always be clear by which phase an uncertainty will be resolved at the latest. We encountered this problem several times in the evaluation data. For instance, *"Are the access control rules correct?"* (RU18) usually resolves at the latest during runtime, as soon as the rules are executed in the running system. However, one could argue differently here, since a rule may never be executed and thus its correctness can never be checked. Further, *"Correct design for secure architecture?"* (CU23) could theoretically never be resolved. Nevertheless, in the case of CU23, we have chosen *design time* as we consider the architecture to be secure, when it follows the guidelines of secure software architectures, for instance by using reference architectures or frameworks. In our opinion, this specific uncertainty is resolved at latest by the time the design is finished. This example again indicates that the usage of the template depends on the point of view, which in this case is ours. The *Root Cause* is another binary decision that is to be made from the design time perspective. Consequently, there is no overlap in the definition. Choosing either *assumption* or *missing information* again is subjected to the point of view. As we already explained several times that we consider uncertainties from the point of view of the design time, the choice of the appropriate option should be unambiguous.

In order to illustrate our argumentative evaluation with quantitative results, we show in Table 8.1 for each category how many uncertainties could be classified successfully or ambiguously. The *reference set* (see Table A.1) consists of 12 ADDs and 24 uncertainties. Consequently 12/12 and 24/24 indicates the optimum, meaning that the options for the respective category were exhaustive enough to classify the respective ADD or uncertainty. With regard to the CWA (see Table A.2), the optimum values are 22/22 and 28/28, respectively. An exception is category *Architectural Element Type*, as we identified two respectively four uncertainties in the previous question (**Q1.3**), for which no meaningful classification is possible. Therefore, neither successful nor ambiguous classification is possible so that the optimum values are adapted.

Table 8.2 reflects the evaluation results we previously collected argumentatively. As described in the introduction of **Q1.4**, the evaluation of the differentiability consists of two aspects - i) non-overlapping definitions and ii) unambiguous categorization. With regard to the first aspect, we conclude that the definitions allow a sufficiently clear distinction between the different options within each category. Although this is mainly based on our humble opinion, we have adequately discussed this aspect for each category. In relation to the second aspect, we have identified the categories where we either had difficulties in making an unambiguous classification ourselves, or where we assume, based on our knowledge, that there could be ambiguous classification possibilities. As indicated by Table 8.2, those categories are *ADD Class*, *Location* and *Resolution Time*. Nevertheless, we conclude for each category that the options are sufficiently differentiable so that uncertainties and ADDs can usually be categorized with only one reasonable option. A more detailed description of the ambiguously classified uncertainties can be found in the evaluation data [10].

Category	**Reference Set**		**Corona Warn App**	
	Successful	**Ambiguous**	**Successful**	**Ambiguous**
ADD Class	9/12	**3/12**	14/22	**8/22**
Amount of Alternatives	12/12	0/12	22/22	0/22
Propability of Revisability	12/12	0/12	22/22	0/22
Posibility of Revisability	12/12	0/12	22/22	0/22
Costs of Revision	12/12	0/12	22/22	0/22
Location	20/24	**4/24**	24/28	**4/28**
Type	24/24	0/24	28/28	0/28
Nature	24/24	0/24	28/28	0/28
Architectural Element Type	22/22	0/22	24/24	0/24
Manageability	24/24	0/24	28/28	0/28
Impact on Confidentiality	24/24	0/24	28/28	0/28
Severity of the Impact	24/24	0/24	28/28	0/28
Resolvable by ADD	24/24	0/24	28/28	0/28
Resolution Time	22/24	**2/24**	26/28	**2/28**
Root Cause	24/24	0/24	28/28	0/28

Table 8.2.: Differentiability of the Options sorted by Category and Case Study (Interesting Cases highlighted in Colour)

8.4.2. G2 - Applicability

This goal is evaluated by applying the *uncertainty type derivation approach* as presented in Section 6.3.1 to the CWA case study. *Step 1* to *Step 8* are executed in order to derive and categorize tpye osf uncertainties (**Q2.1**) based on a collection of ADDs. *Step 9* and *Step 10* are executed to determine the direct and indirect impact of the uncertainty types on architectural elements (**Q2.2**). Due to the large amount of ADDs collected in a preliminary work, we only show the steps based on selected uncertainty types. The complete results are available in the form of a filled *uncertainty template* under [10]. During the evaluation of **G2**, we do not aim to evaluate the content of the *uncertainty template*, i.e. whether or not ADDs, uncertainty types and their relationships are identified and categorized correctly. The focus is to demonstrate that by using our *Uncertainty Type Derivation Approach*, the *uncertainty template* supports the extraction of uncertainty types and their impact on software architectures.

8.4.2.1. Q2.1 - Uncertainty Type Derivation and Categorization

This question discusses the ability of the template to derive and categorize uncertainties based on existing ADDs. To that end, we exemplarily apply *Step 1* to *Step 8* of the approach defined in Section 6.3.1 to the CWA case study. Table A.2 presents the result of the steps in an abbreviated form, as the categories are omitted for illustration purpose. The entire filled-in template is available under [10]. We present the steps exemplarily for *"Persistence of Data (What)"*(CA2) and *"Choice of DB System"*(CA15).

In *Step 1*, CA2 is to be added to to template. Therefore, we create a multi row with with only one line for the time being (more lines are to be added in following cycles) and insert name (*"Persistence of Data (What)"*) and *id* CA2. For *Step 2* We extract the uncertainty type *"What data is persisted?"* by simply reformulating CA2 as question. We further define an unique *id*, e.g. CU2. The uncertainty type CU2 is added to the first row in the multi row of CA2 by inserting name and id (*Step 3*). In *Step 4*, CA2 is to be categorized according to the definitions provided by Section 5.2.1. *Step 1* to *Step 4* are repeated to add CA15 and to extract uncertainty type *"What kind of storage is used?"* (CU15) to the template. For illustration purpose, the first loop terminates as we do not examine any further ADD. In the following, the currently added uncertainty types are examined.

In *Step 5*, it is now to validate whether CU2 has an impact on any other ADD that has been added to the template. We identify that it has an impact on CA15, as the kind of storage might depend on the kind of data that needs to be persisted. During *Step 6*, CU2 is added to the multi column representing CA15. In *Step 7*, CU2 is not yet categorized. Therefore, it has to be categorized according to the definitions provided by Section 5.2.2. Be aware that CU2 already occurs in two multi columns for structural reasons. However, the classification has to be carried out only once. *Step 5* is repeated for CU15 but no impact is identified. Consequently, the second loop terminates and we continue with the next step.

Having extracted each uncertainty type from the existing ADDs, as well having determined their impact on others, it is know to identify further uncertainty types from other sources (*Step 8*). More specific, types of uncertainties that do not correspond to an ADD and therefore have not been identified in the previous steps. For instance, the CWA documentation [68] states that "The Corona-Warn-App server does not store or process any confidential information requiring trust or secrecy" [68]. Although *"Persistence of Data (What)"*(CA2) already covers parts of that information, we decided to add "Is the data to be persisted confidential?"(CU24) as additional uncertainty which might influence other design decisions. For each uncertainty type identified in this step, *Step 5* to *Step 7* are to be repeated. In our case, CU24 has an impact on CA2 (*Step 5*) which is why we add it to the respective multi row (*Step 6*). Finally, CU24 has to be categorized. As this is only an excerpt of the actual process, we provide a short summary on the identified ADDs, uncertainty types and additional impact on relationships between the types of uncertainties and ADDs.

- ADDs classified: **22**
- Uncertainty types derived: **22**
- Additional uncertainty types: **6**
- Additional impact relationships derived: **35**

Based on the 22 ADDs, 22 uncertainty types could logically be derived. After the examination of [71] and [68], six more uncertainty types are identified. Of course, these absolute values do not provide any information about the suitability of our approach, as more detailed investigations of the sources could further increase these figures. More striking, however, are the 35 additionally derived impact on relationships (*Step 6*). In other words, in 35 cases, we are able to identify that an uncertainty type has an impact on the decision of another ADD. This information is used in the following to identify not only the direct (and trivial) impact of uncertainty types on architectural elements, but also their indirect impact.

8.4.2.2. Q2.2 - Uncertainty Type Impact Derivation

Question **Q2.2** evaluates the ability of the template to derive the impact of uncertainty types on architectural element types, i.e. on software architectures. We exemplarily apply *Step 9* and *Step 10* of the approach defined in Section 6.3.1 to the CWA case study (cf. Section 8.2). Although these steps are carried out for all the uncertainty types collected so far, we only present them for the following ones: *"What data is persisted?"* (CU2), *"What kind of storage is used?"* (CU15) and "Is the data to be persisted confidential?" (CU24).

				...	*Uncertainty related Categories*
ADD	**ID**		**Uncertainty Type**	...	**Architectural Element Type**
Persistence of Data (What)	CA2	CU2	What data is persisted?	...	Component
		CU24	Is the data to be persisted confidential?	...	Hardware Resource
Choice of DB System	CA15	CU15	What kind of storage is used?	...	Component
		CU2	What data is persisted?	...	Component

Table 8.3.: Collection of Uncertainty Types to be examined for direct and indirect Impact

Table 8.3 highlights the uncertainty types under examination which are based on the result of the previous question. For the three uncertainty types CU2, CU15 and CU24 we iteratively determine on which architectural element types they might have a direct (*Step 9*) and/or indirect impact (*Step 10*). Starting with CU2, we can determine its direct impact on element type Component by simply extracting the information provided by category *Architectural Element Type*. It further has a potential indirect impact on element type Hardware Resoruce as it occurs in the multi row of CA15 (resolved by CU15) which affects hardware resources. CU15 directly affects element type Hardware Resource. Any further indirectly affected elements types cannot be identified as it does not occur in any other multi column of Table 8.3. In the case of CU24, the directly and indirectly affected element types are equal, indicating that no propagation occurs. As this is only an excerpt of the actual process, we provide a short summary on the identified direct and indirect *impactOn* relationships:

- Uncertainty types examined: 24
- Direct *impactOn* relationships: 24
- Indirect impactOn relationships (same element type): 21
- Indirect impactOn relationships (different element types): 19

Of the 28 uncertainty types identified in the previous question, four (CU1, CU17, CU18, CU23) cannot not be used, as they do not specify category *Architectural Element Type* (cf. **Q1.3**) which is indispensable to derive the direct and indirect impact. Based on the 24 remaining uncertainty types, 24 direct *impactOn* relationships can be derived. Further, in 21 cases we elaborate that the directly affected element type matches the indirectly affected element type so that no propagation occurs. Again, these absolute values do not provide any information about the suitability as a larger amount of previously identified uncertainty types can increase those values. Nevertheless, it is striking that with our structured approach it is possible to uncover 19 cases where uncertainty types might have an indirect impact on other architectural element types, hence propagate trough the software architecture. This suggests that the *uncertainty template* not only helps to determine direct impact, but also to extract non-trivial propagation effects. With this information, it is possible to assess the impact of uncertainty types more accurately, as software architects now have an understanding of the other element types that uncertainty types may affect.

8.4.3. G3 - Purpose

G3 evaluates the purpose of the *uncertainty template*. Therefore, **Q3.1** evaluates to what extend it is able to provide reuse of knowledge and to what extend it can also create awareness. In **Q3.2** we compare our approach with existing approaches to demonstrate that we provide a more precise classification of uncertainty types at the architectural level.

8.4.3.1. Q3.1 - Reuse of Knowledge and Awareness

We evaluate the capability of the uncertainty template to provide reuse of knowledge and to create awareness in two ways. First, argumentatively and second, by comparing our reference set and the one we completed based on the CWA. Software architects use the uncertainty template to classify types of uncertainties and collect them in a structured way. This includes in particular the manual and cumbersome determination of the impact on other ADDs, hence on other types of architectural elements. Once classified, the uncertainty types can be reused across several software architectures within the domain of interest for which the template is created, i.e. filled in. Further, using our uncertainty template creates awareness with regard to uncertainties in the following two scenarios: First, software architects can use existing and already completed uncertainty templates as source of information for theirs. In doing so, they might identify types of uncertainties which they have not yet considered to be important. Be aware that uncertainties, like ADDs, can be domain specific, as well as relevant for several domains, or also generally applicable. Therefore, software architects still have to assess whether or not an uncertainty type is relevant for their domain.

The second scenario refers to the instantiation of uncertainty types. Most importantly, we believe that less experienced software architects can use pre-filled templates to annotate their software architectures with concrete uncertainties. In particular, they do not need to know which types of uncertainties exist, nor which architectural elements they might affect directly as well as indirectly. The categories also allow inexperienced software architects to evaluate their instantiated uncertainties using the guidelines we defined in Section 6.6. In summary, the concept of the *uncertainty template* enables reuse of knowledge and creates awareness when creating and specifically when using it.

To evaluate **Q3.1** quantitatively, we compare our reference set (see Table A.1) to the uncertainty template based on the CWA (see Table A.2). More specifically, we analyse the amount of uncertainty types shared across both templates. Note that we created both uncertainty templates consecutively, but independent of each other. This enables us to determine the overhead that could be avoided, if the reference set were used as knowledge base for the creation of the CWA-based template. Table 8.4 illustrates the uncertainty types that are shared across both templates. It shows that from the 22 uncertainty types identified while creating the reference set, 12 are also part of the CWA-based uncertainty template. Further, Table 8.4 reveals that 16 of the 32 *impactOn* relationships are within this collection of 12 uncertainty types. Thus, if one had taken the reference set as a basis, the reuse of knowledge would have been possible for 12 uncertainty types including 16 *impactOn* relationships. Not to be neglected is the necessary effort to check the uncertainties for suitability. Yet, we claim that this is significantly less in relation to the effort required to categorise uncertainty types and identify their impacts. Although these absolute values do not allow any general statements, they nevertheless show that the template enables knowledge to be reused, as any value greater than zero implies that knowledge is reused.

In addition to that, we analyse the remaining uncertainty types of the reference set with regard to their relevance for the CWA-based uncertainty template. So we determine those uncertainty types that could not be identified from the available sources [68, 72] but might relevant for CBSA in the domain of CTS. Due to the more general scope of the reference set,

Uncertainty Type	**ID (Reference Set)**	**ID (CWA)**
What is the distribution?	RU1	CU1
How is communicated?	RU2	CU12
Where is deployed?	RU3	CU20
What data is persisted?	RU4	CU2
How is data persisted?	RU5	CU4
Which authentication mechanisms are used?	RU6	CU16
Which component is chosen?	RU8	CU18
What data is provided?	RU9	CU9
What data is entered?	RU10	CU6
Is the data anonymized?	RU11	CU22
What is the structure of the interface?	RU12	CU8
Is the data to be persisted confidential?	RU16	CU24
Summary:	Matching: 12 / 24	
***impactOn*-Relationships:**	16/30	

Table 8.4.: Shared Uncertainty Types

the remaining uncertainties are potentially equally relevant for the CWA-based uncertainty template. Table 8.5 shows the remaining uncertainties with regard to their relevance for the domain of CTS. In no case have we found any argument why the uncertainty type should not be relevant. Consequently, there are 12 more uncertainty types and another

Uncertainty Type	**ID (Reference Set)**	**Relevant for CWA**
What kind of access control is used?	RU7	✓
Is confidential data transmitted?	RU13	✓
Is the deployment provider trustworthy?	RU14	✓
Is user data stored/processed?	RU15	✓
Can access control cover all matters?	RU17	✓
Are the access control rules correct?	RU18	✓
What is the behaviour of the component?	RU19	✓
Is the component currently compromised?	RU20	✓
Is the component's provider trustworthy?	RU21	✓
Which subjects access the interface?	RU22	✓
Is the data entered correct?	RU23	✓
What is the user behaviour?	RU24	✓
Summary:	Relevant: 12 / 12	
***impactOn*-Relationships:**	14/30	

Table 8.5.: Remaining Uncertainty Types

14 *impactOn*-relationships which are potentially relevant for the CWA-based uncertainty template but which were not identified based on the the available sources when creating it. Hence, the usage of the reference set as additional source of information even creates awareness. Altogether, the entire reference set could potentially be integrated into the uncertainty template for *contact tracing systems*, enabling reuse of knowledge and the creation of awareness. Although it is not possible to evaluate this in the time available, the reference set could serve as a general starting point for more specific uncertainty templates, where *"specific"* refers to the focus on particular domains of CBSA-based software systems.

8.4.3.2. Q3.2 - Relevance

This question evaluates the purpose and the relevance of the uncertainty template and the categories by comparing it to related approaches. We discuss why our categories enable a more precise classification of uncertainties on architectural level. When speaking about architectural level, we refer to Component Based Software Engineering (CBSE) as described in Section 2.3. So, in particular, we are discussing about uncertainties in Component Based Software Architectures (CBSAs) and not software architectures in general.

To answer question **Q3.2** we proceed as follows: First we discuss the relevance of our approach argumentatively by highlighting the differences to related approaches. In particular, we address the suitability of our categories compared to categories of other approaches. At this point, we give an overview to the extent to which we have adopted or modified categories from other approaches. Finally, we first categorise uncertainties already classified by us with the categories of other approaches and second, we categorise uncertainties categorised with other approaches with our approach. In both cases, we discuss why our approach is more precise with regard to classification of uncertainties at architectural level than existing approaches. *Precise* in this case means that our classification allows more detailed statements regarding the impact of uncertainties on CBSAs, with a focus on confidentiality.

In Chapter 3 we provide a detailed overview of related approaches that deal with uncertainties in various domains. Many of the available approaches consider uncertainties in the domain of SASs [52, 26, 19, 31, 32] but also in other domains such as CPSs [75], object behaviour modeling [48] or specifically for access control [14]. Although some approaches allow the classification of uncertainties in models [51] or at design time [22], all of them lack the ability to explicitly refer to entities such as interfaces, communication or components, but this is exactly what CBSA is all about. It can therefore be assumed that none of the available approaches are capable of classifying uncertainties at the architectural level sufficiently. This is exactly the gap we are addressing with the provision of our template, i.e. our proposed categories.

In order to show quantitatively that existing approaches are probably not suitable to classify uncertainties at the architectural level, Table 8.6 illustrates to what extent we adopted, adapted or discarded existing categories and their options which are explained in detail throughout Section 5.1.2. Be aware that those categories are not to be interchanged with the ones we propose. Even if names are matching, we usually adapt the definitions for our purposes.

	Categories			*Options*		
Name	**Adopted**	**Adapted**	**Discarded**	**Adopted**	**Adapted**	**Discarded**
Location		✗		0/8	5/8	3/8
Level		✗		3/9	0/9	6/9
Nature	✗			2/2	0/2	0/2
Manageability		✗		2/2	0/2	0/2
Emerging Time		✗		0/6	3/6	3/6
Impact		✗		0/3	0/3	3/3
Relationship			✗	0/2	0/2	2/2
Source			✗	0/5	0/5	5/5*
Summary	1/8	5/8	2/8	7/37	8/37	22/37

Table 8.6.: Existing Categories/Options which are adopted, adapted or discarded. (*) indicates that more Options are available in the Literature.

From the eight categories we extract from existing literature literature, only category *Nature* is adopted completely, i.e. definition and options remain unchanged. In five cases, we adapt parts of the category, so either its underlying idea or some of the options. Category *Manageability*, for instance, is adapted to the extend that we reuse its definition as well as its options. In this case, however, we add another option which is why this category is classified as *adapted*. For category *Impact*, we discard the available options but adopt the underling idea of categorizing the impact on quality attributes. Three categories are fully discarded. Altogether, out of 37 different options only seven were adopted, eight had to be adapted and 22 more are discarded. Regarding category *Sources* all available options are not extracted in the first place, as they were too many. The set of discarded options would therefore even larger if all of them were included. Overall, Table 8.6 indicates that existing approaches (i.e. their categories and options) are probably not suitable for our need, as to a large extent many of them are either adapted or discarded completely.

In the following, we further illustrate this statement from the point of view of our proposed uncertainty categories. More specifically, Table 8.7 shows to what extend the categories and options we propose are adopted, adapted or newly created. From the ten categories we propose, only *Nature* is fully adopted. Yet this category is known and relevant for uncertainties in general, i.e. also outside our field of research [73]. Five categories are based on existing categories. Nevertheless, in each of these cases it was necessary to adapt or extend the definitions and especially the options for our purposes. More striking is the fact that four new categories are identified that we consider to be relevant to classify uncertainties at architectural level. This picture also emerges from the evaluation of the options. Here, it is possible to adopt six out of 32 options. As the majority of the options is either adapted or completely new, this overview confirms the necessity for a novel approach that fits the need of classifying uncertainties at architectural level. We claim that our approach exactly closes this gap.

	Categories			*Options*		
Name	**Adopted**	**Adapted**	**New**	**Adopted**	**Adapted**	**New**
Location		✗		0/4	4/4	0/4
Type		✗		2/3	0/3	1/3
Nature	✗			2/2	0/2	0/2
Architectural Element Type			✗	0/5	0/5	5/5
Manageability		✗		2/3	0/3	1/3
Impact on Confidentiality		✗		0/3	0/3	3/3
Severity of the Impact			✗	0/3	0/3	3/3
Resolvable by ADD			✗	0/2	0/2	2/2
Resolution Time		✗		0/5	3/5	2/5
Root Cause			✗	0/2	0/2	2/2
Summary	1/10	5/10	4/10	6/32	7/32	19/32

Table 8.7.: Proposed Categories/Options which are adopted, adapted or new

To further evaluate whether or not existing approach are capable of classifying uncertainties, we apply the approaches of Perez-Palacin and Mirandola [51], Hezavehi et al. [31] and Bures et al. [14] to the CWA case study, i.e. to the uncertainty types illustrated in Table A.2. We first discuss this in terms of the following collection of uncertainty types: *"What data is persisted?"* (CU2), *"What data is entered?"* (CU6), *"How is communicated"* (CU12), *"Which authentication mechanisms are used?"* (CU16), *"Is the data to be persisted confidential?"* (CU24). Another view regarding the entire collection (CU1 - CU28) is provided in the course of this section by creating equivalency classes of uncertainty types. For each approach, we discuss to what extend it enables precise statement about the classified uncertainty types.

ID	**Location**	**Nature**	**Level**
CU2	—	epistemic	1st order
CU6	Input Parameters	aleatory	1st order
CU12	Model Structural	epistemic	1st order
CU16	—	epistemic	1st order
CU24	—	epistemic	1st order

Table 8.8.: Categorization according to Perez-Palacin and Mirandola [52]

Table 8.8 illustrates the categorization of the selection of uncertainty types with the approach of Perez-Palacin and Mirandola [52]. In two out of five cases it is not possible to determine a valid location, as we do not consider any of the available options to be applicable. Furthermore, it is noticeable that all uncertainties are assigned to the first order (i.e. *known uncertainties*), which is only logical since they are known. Otherwise they would not be part of the table. So this indicates that categories *Location* and *Level* are probably not suitable for the categorization of uncertainties at architectural level. As our approach also includes category *Nature* (and many others) we conclude that the approach presented by Perez-Palacin and Mirandola [52] does not enable to make precise statements.

ID	Location	Nature	Spectrum	Emerging Time	Sources
CU2	—	epistemic	Scenario Uncertainty	Design Time	Incompleteness
CU6	Environment	aleatory	Statistical Uncertainty	Design Time	Human in the Loop
CU12	Model	epistemic	Scenario Uncertainty	Design Time	Incompleteness
CU16	Model	epistemic	Scenario Uncertainty	Design Time	Incompleteness
CU24	—	epistemic	Scenario Uncertainty	Design Time	Incompleteness

Table 8.9.: Categorization according to Mahdavi et al. [31]

Table 8.9 illustrates the categorization of the selection of uncertainty types with the approach of Hezavehi et al. [31]. At first glance, the approach seems to be suitable for our purpose, as all but two cells are filled. Only in two cases is it not possible, in our opinion, to categorise the uncertainties on the basis of the available options. What is particularly striking, however, is the (almost) identical classification of `CU2`, `CU12`, `CU16` and `CU24`. If these uncertainties are to be compared with each other based on this classification, there is no meaningful statement possible. Our classification, on the other hand, allows a much more precise statement in this respect as we provide various other categories.

ID	Source	Nature	Level
CU2	System Behaviour	epistemic	2nd level
CU6	System Environment	aleatory	2nd level
CU12	System Behaviour	epistemic	2nd level
CU16	System Behaviour	epistemic	2nd level
CU24	System Environment	epistemic	2nd level

Table 8.10.: Categorization according to Bures et al. [14]

Table 8.10 illustrates the categorization of the selection of uncertainty types with the approach of Bures et al. [14]. Compared to the previous two approaches, this one is still the most applicable in our opinion. Nevertheless, category *Level* is similar to the on proposed by Perez-Palacin and Mirandola [52], hence it offers no added value in our humble opinion.

In all three cases, we claim that the approaches do not allow the precision that we achieve with our approach. For example, none of the approaches provides the possibility to define the specific location within CBSAs, i.e. the type of architectural element uncertainties can be assigned to. Further, the approaches are not capable (also because they were not developed for this purpose) to make any statements about the impact of uncertainties on the confidentiality of the system.

In the following we extend this discussion to all uncertainty types from the CWA-based uncertainty template (CU1 - CU28). After showing that the other approaches do not allow a precise distinction between types of uncertainty, we now want to clarify this information using equivalence classes. Uncertainty types are in a class whenever the options for all categories are exactly the same. If no option can be defined for a category, it is considered to be *"unknown"* which we consider as fall back option for evaluation purposes. Given the classification in Table 8.9, for instance, the following three equivalence classes can be identified: {CU2, CU24}, {CU6} and {CU12, CU16}. With a small number of equivalence classes, many uncertainty types fall into the same class, which means that they cannot be distinguished from each other at the category level. An evaluation or comparison of the uncertainty types is therefore not possible, at least within an equivalence class. A larger amount of equivalence classes thus means that the uncertainty types are largely categorised differently from each other which enables differentiation at category level.

Approach	**Perez**	**Mahdavi**	**Bures**	**Our Approach**
# Equivalency Classes	3	8	5	21

Table 8.11.: Comparison of the Amount of Equivalency Classes

Table 8.11 illustrates the amount of equivalency classes we identify for the 28 uncertainties (CU1 - CU28). Using the approach of Perez-Palacin and Mirandola [52] to categorize these uncertainties, we only identified three equivalency classes which means that each uncertainty type falls in either on of them. Consequently, in many cases there is no assessment possibility with regard to a possible comparability. For the approaches of Hezavehi et al. [31] (eight classes) and Bures et al. [14] (five classes), the same results apply. Our approach, on the other hand, produces 21 equivalency classes, including 16 which only consists of one uncertainty type. Consequently, most of the uncertainty types can be compared to each other by at least one differently categorized option. This highly indicates that our approach produces more usable results than the others. Yet, it is conceivable that in some cases one would want to group types of uncertainties together. At this point, a larger set of equivalence classes seems to be a disadvantage. In this case, we propose to use a threshold for classification into equivalence classes. For example, two types could end up in the same class if they are classified differently in up to 3 categories

To further support our findings, we show the other direction in the following. Specifically, we compare uncertainties identified and categorised by Perez-Palacin and Mirandola [52] by categorising them with our approach. Thereby, we discuss how our approach is able to provide more precise statement compared to the existing classification. The classification of the uncertainties PU1 - *PU6* according to Perez-Palacin and Mirandola [52] is illustrated in Table A.3. It has to be noted that these uncertainties stem from a work that has its focus on uncertainties in SASs. Therefore, we do not expect that each uncertainty type can be categorized sufficiently. Table A.4 shows the classification of PU1 - PU6 with our approach. We successfully conducted the classification, apart from PU1 and *PU6* were it is not possible to choose an appropriate option for both, category *Location* and category *Architectural Element Time*. However, it is striking that *Location* in particular is one of the categories that we adapt from Perez-Palacin and Mirandola [52]. We reason that PU1 and *PU6* are uncertainties specific to the modelling of architectures and not uncertainties in architectures themselves. Since the approach of Perez-Palacin and Mirandola [52] is specifically designed for uncertainties in modelling, but ours is not, we claim that the inability of our approach in these two individual cases should not be generalised.

On the contrary, the successful classification of the remaining uncertainties with our approach allows for more precise statements. For instance, based on the guideline presented in Section 6.6 (especially the combination of categories *Manageability, Resolution Time* and *Root Cause*) we can compare the uncertainties and conclude that PU3 is an uncertainty type that is easy to handle and therefore, if instantiated, should be considered first. Further, *PU4* potentially has a severe impact on the confidentiality and, if instantiated, must be resolved as soon as possible. Those exemplary statements are not readily possible with the approach by Perez-Palacin and Mirandola [52].

8.4.4. G4 - Usability

G4 evaluates the usability of the UIA with regard to the necessary knowledge (**Q4.1**) and the amount of model elements that must be considered (**Q4.2**) when using our analysis.

8.4.4.1. Q4.1 - Required Knowledge

We evaluate the required knowledge by comparing which concepts need to be understood and which tasks need to be performed when analysing the impact of uncertainty manually, and by using our approach. Table 8.12 provides an overview of the concepts **C1 - C11** and illustrates which are required for the manual analysis, from the *expert architect's* point of view, and from the *user architect's* point of view.

Basic knowledge about uncertainties in CBSA (**C1**) is considered to be relevant for each of the views. The differences can only be observed in the more specific knowledge areas. The PCM modelling tool (**C2**) is the underlying approach that we use to model software architectures. As described in the course of this section, a manual analysis requires a sound understanding of the *Palladio* concepts, including the structure and concepts of the different sub-models. As *User architects* only annotate existing architectures, they solely require basic knowledge regarding the elements to which they can assign uncertainties. *Expert architects* do not need to know about the internals of the PCM at any point, as we abstract from this by specifying an independent set of architectural element types. The mapping between those types and concepts of the PCM is implicitly provided by the UIA.

C3 to **C6** are not to be executed by *user architects* which is why they do not need do understand the underlying concepts. With regard to **C3**, *expert architects* are supported by our *uncertainty type derivation approach*. Nonetheless, identifying the impact of uncertainty

#	Manual Analysis	Expert Architect	User Architect
C1	**Yes**, because this is the underlying concept of the entire approach	**Yes**, because this is the underlying concept of the entire approach	**Yes**, because this is the underlying concept of the entire approach
C2	**Yes**, profound knowledge required throughout the entire process	**No**, because architectural element types are independent of the PCM	**Yes**, but only basic knowledge about the structure
C3	**Yes**, profound knowlege required because of the inexistance of uncertainty type collections.	**Yes**, but supported by the uncertainty template	**No**, because this not their responsibility
C4	**No**, because this concept only exists when using the uncertainty template	**Yes**, but only executed once for domain of interest	**No**, because this is not their responsibility
C5	**Yes**, profound knowledge required because this is a non-trivial task.	**Yes**, but supported by the uncertainty type derivation approach	**No**, because this is not their responsibility
C6	**No**, because this concept only exists when using the UIA	**Yes**, but mainly corresponds to the categories of the uncertainty template.	**No**, because this is not their responsibility
C7	**No**, because this concept only exists when using the UIA	**No**, because this is not their responsibility	**Yes**, knowledge required to use the analysis.
C8	**Yes**, and has to be manually conducted.	**No**, because this is not their responsibility	**Yes**, but supported by analysis
C9	**Yes**, profound knowledge required and has to be manually conducted.	**No**, because this is not their responsibility	**No**, because this is executed by analysis
C10	**Yes**, analysis of all elements found in C9	**No**, because this is not their responsibility	**Yes**, analysis of all elements found in C9
C11	**Yes**, and lack of basis for assessment.	**No**, because this is not their responsibility	**Yes**, but guided by analysis and guidelines.

Table 8.12.: Required Knowledge to analyse Impact of Uncertainties manually, from the Perspective of *Expert Architects*, and from the Perspective of *User Architect*

types on types of architectural elements is not considered to be trivial. Yet, we claim that our approach reduces the amount of required knowledge compared to a manual analysis which is carried out without any structure or guidance. This has already been sufficiently discussed in **Q3.1**.

The categorization of the extracted uncertainty types (**C4**) is only to be conducted by *expert architects* as this concept only applies for our approach. Nevertheless, this is to be done only once for uncertainty types within a domain of interest. Furthermore, this task, which can initially be classified as additional concept, contributes considerably to the reduction of the necessary knowledge in the course of the analysis, i.e. for the assessment of the impact. By following our uncertainty type derivation approach, deriving the impact of uncertainty types (**C5**) is structurally guided by the *uncertainty template*. Although it requires a non-negligible effort to identify the impact in the first place, we claim that the structured approach again reduces the amount of required knowledge compared to a manual analysis.

Transferring the gathered uncertainty type information (**C6**) into to UIA is to be done by *expert architects only*. Besides choosing the more specific *architectural element type*, this step does not require any particular knowledge. **C7** to **C11** are not to be executed by *expert architects* as they only provide the input for the actual usage of the UIA. Although software architects must first become familiar with our analysis (**C7**), we assume that this additional effort is not in proportion to that of a manual analysis. However, we cannot prove this assumption without extensive user studies. *User architects* need to annotate their software architectures with uncertainties (**C8**). Whereas the UIA automatically proposes for a selected uncertainty type to which architectural element it can be assigned, a manual analysis requires architects to do that manually.

C9 is executed automatically by the analysis based on the uncertainty types as specified by the *expert architect*, the instantiated and annotated uncertainties, and the pre-defined structural propagation rules. In a manual analysis, we see this as the most critical point, since manual propagation is very time-consuming and requires detailed knowledge about the structure of the PCM models as well as knowledge of how uncertainties can potentially propagate in the architecture. This is where the strength of our approach becomes particularly apparent.

For the identified architectural elements that are potentially affected, both the person conducting the manual analysis as well as the *user architect* need to manually verify whether or not the impact actually exists (**C10**). This is necessary because this decision depends on context-related information, such already taken ADDs. At this point we would like to emphasise that the aim of this thesis is not to develop an *uncertainty assessment analysis* that incorporates the context, but an *uncertainty impact analysis* that guides the architect when assessing the impact. Nevertheless, this task depends on the amount of elements identified in **C9**. While an experienced architect might identify fewer potentially affected elements in a manual analysis than the UIA, this is likely to depend on his/her expertise. Yet it is precisely the latter that we want to reduce with our approach.

The assessment of the impact of the uncertainties (**C11**) is guided by using the categorises and the guidelines we propose in Section 6.6. This is not the case when it comes to a manual analysis. At this point, the additional effort of categorizing the types of uncertainties pays off. Further, we already exemplarily implemented how the UIA can be enhanced to guide the assessment of propagated uncertainties based on metrics.

This discussion yields several results. Quantitatively, the manual analysis requires 8 concepts that need to be understood or tasks that need to be executed. In contrast, *user architect's* and *expert architect's* view together even require 10 concepts or tasks.

Seen in this perspective, our approach requires more knowledge. However, there is a serious difference when one compares both perspectives qualitatively: the manual analysis requires a high level of necessary knowledge and effort from start to finish. We consider **C9** in particular to be decisive here, as less experienced architects probably do not identify at least all the elements that are actually affected. In other words, their *impact set* may not include the entire *affected set*, because this requires expertise that we assume less experienced software architects do not have. As this step is the basis for the validation (**C10**) and assessment (**C11**) of the impact of uncertainties, it is therefore probably not possible for less experienced architects to manually analyse uncertainties. Beside the required amount of knowledge, the discussion further highlights the amount of manual effort when analysing uncertainties manually. Moreover, the manual approach must be repeated for each architecture under consideration.

In contrast, our approach clearly separates the necessary knowledge and required effort: Experienced software architects (i.e. *expert architects*) still require a non negligible amount of knowledge to extract types of uncertainties and their impact on architectural element types, but this preparatory work is only required once within a domain of interest. In particular, this enables less experiences software architects (i.e. *user architects*) to annotate and analyse uncertainties and their impact for the first time without profound knowledge. *User architects* only need to understand six concepts compared to eight concepts when using a manual analysis. Further, any of the six *user architect*-related concepts are either basic domain knowledge (**C1, C2**) or in some ways guided by the analysis (**C7, C8, C10, C11**). This indicates that using our approach reduces the necessary knowledge from the *user architect's* point of view, therefore enables less experienced architects to analyse the impact of uncertainties in software architectures.

8.4.4.2. Q4.2 - Amount of Elements

Question **Q4.2** evaluates the amount of elements that need to be considered when analysing the propagation effects of uncertainties. Therefore, we analyse the four scenarios presented in Section 8.2. Specifically, we analyse the amount of elements affected by the uncertainties of the respective scenarios. We distinguish between the the *impact set* calculated by using the *uncertainty template* only, and the one calculated by the UIA.

	Uncertainties	$ImpactSet_{template}$	$r_{template}$	$ImpactSet_{uia}$	r_{uia}
Scenario 1	7	28	0.2523	17	0.1532
Scenario 2	7	28	0.2523	16	0.1441
Scenario 3	8	34	0.3063	11	0.0991
Scenario 4	4	9	0.0811	9	0.0811

Table 8.13.: Amount of elements to be analysed using the template and the *UIA*

Table 8.13 illustrate the results of our evaluation. Altogether, the architecture under consideration consists of 111 elements. In a manual analysis, software architects would therefore, in the worst case, have to look at each of the 111 elements for each instantiated uncertainty, i.e. analyse whether there is an impact or not. Based on the defined metrics, Table 8.13 shows the results of the evaluation both in absolute terms ($ImpactSet_{template}$, $ImpactSet_{uia}$) and relative to the total number of architectural elements ($r_{template}$, r_{uia}). Nevertheless, those results still depend on the comprehensiveness of the *impact sets*, which is evaluated by **Q6.1**.

Under this constraint, the results are satisfactory, as the remaining amount of elements to be examined using both approaches is a maximum of $\sim$ 30% of the total number of elements. It is also noticeable that the UIA consistently delivers at least the same, but often better results than using the *uncertainty template* only. As expected, the results are dependent on the uncertainties to be propagated, which is why *scenario 4* produces the smallest values. It can be assumed that as the number of uncertainties per scenario increases, the r-value also increases. Nevertheless, the results highly indicate two things: First, the UIA improves the overestimation compared to using the *uncertainty template* only. Consequently, the definition of structural propagation rules seems to be a valid measure to reduce the *impact set*. Second, our approach reduces the amount of elements software architects still need to analyse when using our approach. At this point, it should be emphasised that these evaluation results have no claim to general validity, as the situation is only shown for an exemplary set of instantiated uncertainties on the basis of a simplified architecture model. Whether our approach thus consistently calculates this small amount of elements to be investigated (compare to the entire set of architectural element) must therefore be confirmed in further case studies.

8.4.5. G5 - Functionality

G5 evaluates whether the UIA is capable to annotate and propagate uncertainties in software architectures (**Q5.1**).

8.4.5.1. Q5.1 - Uncertainty Annotation and Propagation

To answer question **Q5.1**, we discuss to what extend the UIA is capable to support the annotation and propagation of uncertainties. The representation of uncertainty as first class entity in software architectures is a prerequisite for uncertainty annotation. In Section 7.1, we illustrated this based on the PCM. Table 7.1 provides an overview of the currently supported architectural element types for both, uncertainty annotation and propagation. In our humble opinion, this collection covers the most relevant concepts of CBSE. As the PCM is only limited to the description of architectures in the domain of CBSE, we assume that the concept of representing uncertainties as first-class entity can be transferred to other ADLs in the same domain.

In the following, we refer to the uncertainties ICU1 - ICU28 as introduced in Section 8.2. We assess how many of the instantiated uncertainties can successfully be annotated and propagated. "Successful" in refers to whether an uncertainty can be assigned to the architectural element where it also has a direct impact, as well as whether the propagation can be carried out. We derived the following numbers:

- Successfully annotated: 28 of 28
- Successfully propagated: 22 of 28

For none of the 28 uncertainties it was necessary to define further architecture element types, i.e. we were able to assign each uncertainty to the element we believe it affects. However, in six cases (ICU1, ICU4, ICU6, ICU13, ICU22, ICU26) it was not possible to propagate uncertainties due to the lack of propagation algorithms in our current prototypical implementation. As already explained, we do not assume that the definition of these algorithms differs from the others in terms of effort, so they can simply be added. This limitation therefore only refers to the prototypical implementation, and not to the concept of our analysis. Therefore, we conclude that the analysis is capable of annotating and propagating uncertainties. The results are provided in detail in our data set [10].

8.4.6. G6 - Accuracy

The last goal evaluates the accuracy of the UIA regarding the *precision and recall* of the propagation results.

8.4.6.1. Q6.1 - Precision and Recall

We distinguish between the functional accuracy and the accuracy using context-related information. Whereas the first perspective evaluates to what extend the UIA finds the elements it is supposed to to, the latter evaluates to what extend it identifies the elements which are actually affected by uncertainties. In both cases, we use the scenarios introduced in Section 8.2. In each of the four scenarios, we have intentionally added back uncertainty and analyse how it affects the architecture. So, while for the first perspective we look exclusively at uncertainties in isolation, for the second we aggregate information about certainties and uncertainties of all kinds. For both views, however, we assume that we have correctly determined the impact at the type level. For the evaluation, this means that the input in the form of the UncertaintyType model is considered correct. If an element is not found, this is either because the implementation is faulty (first perspective) or because additional, context-relevant information would be necessary to find it (second perspective).

	$Precision_{functional}$	$Recall_{functional}$	$Precision_{context}$	$Recall_{context}$
Scenario 1	1.0000	1.0000	0.7647	1.0000
Scenario 2	1.0000	1.0000	0.6250	1.0000
Scenario 3	1.0000	1.0000	0.9091	1.0000
Scenario 4	1.0000	1.0000	0.4444	1.0000

Table 8.14.: Precision and Recall of the calculated *Impact Set*

Table 8.14 illustrates the results of the evaluation. As it turns out, we get precision and recall values of 100% for all four scenarios when determining the functional accuracy. These perfect values indicate that our implementation is at least functionally correct and enables the identification of potentially affected elements more easily. However, it is important to determine how much the UIA overestimates and, in particular, whether it finds at least all elements that are actually affected. The first aspect is indicated by a high precision value, whereas the second is indicated by a high recall value. As illustrated by Table 8.14, the precision varies between $\sim 44\%$ and $\sim 90\%$, depending on the scenario.

With regard to *scenario 1*, the UIA identifies some architectural elements of type *communication component* which are not affected because the data transmitted is either publicly available or the decision to use encryption is already taken. The same applies for *scenario 2*. In both cases, the precision is acceptable. The precision of *scenario 3* is over 0.90, which means that almost all identified elements are actually affected. In *scenario 4*, the precision is only just under 0.45, so that less than every second element is actually affected. This poor value could be due to the fact that this scenario unfortunately consists of four uncertainties, of which only one is to be propagated at all. The other three have no indirect impact according to the underlying UncertaintyType model. Although the available data suggest that the UIA may overestimate in some cases, we still perceive the results to be satisfactory. In particular, for exactly these four scenarios, we do not find any other elements that are affected but could not be found, which leads to a recall of 100%.

All in all, the results indicate the following: Our UIA seems to be functionally correct. It also produces sufficiently accurate results in the form of *impact sets*. Consequently, software architects no longer have to examine the entire architecture, but are shown the elements they need to analyse in more detail. We claim that this task can be carried out by less experienced architects, as the problem is reduced from the analysis of the entire architecture to a few elements. Our analysis thus makes it possible for less experienced architects not only to determine the impact of uncertainties, but also to asses them. Nevertheless, this needs to be evaluated by a comprehensive user study.

8.5. Threats to Validity

In the following, we discuss the threats to validity for our evaluation. Inspired by the work of Hahner [29], we also use the schema by Runeson and Höst [59] and discuss internal, external and construct validity, and reliability separately. Validity in general refers to the "trustworthiness of the results, to what extent the results are true and not biased by the researchers' subjective point of view" [59].

Internal validity describes the extent to which the assessment allows for unfounded results, usually due to systematic errors and bias [9], as well as additional interferences due to overlooked or ignored influences [29]. The *uncertainty template* is mostly evaluated argumentatively, which can be biased by subjectivity. Regarding the evaluation of the exhaustiveness of the categories (**Q1.3**), we assume that our knowledge gathered during the elaboration of this thesis is sufficient to choose the right option for each uncertainty and ADD to be classified.

The evaluation of the differentiability of the categories (**Q1.4**) is based on our perception of unambiguous categorization. Although we have tried to discuss as self-critically and objectively as possible, bias cannot be ruled out. To further reduce this threat we base our reasoning wherever possible on our *reference set* and the uncertainty types based on the CWA case study. Since we extracted the latter ourselves, it cannot be ruled out that we unintentionally extracted only those that provide satisfactory results.

When evaluating the reuse of knowledge and the creation of awareness (**Q3.1**), two threats occur. First, it is possible that the knowledge gathered by filling the *reference set* influenced the process of filling the CWA-based uncertainty template. This would bias the evaluation results, as the templates to be compared should be independent of each other. To mitigate this threat, we tried to fill the CWA-based template as far as possible without the previously collected knowledge and only based on the existing sources [68, 71]. The second threat is that we have aligned the terminology of both templates, possibly changing uncertainty types not only syntactically but also semantically. We may have accidentality adjusted two previously different uncertainties so that they were identical after the alignment.

In **Q4.1**, we defined concepts to evaluate the required knowledge when using our UIA. Those concepts are based on our point of view, so the evaluation could be biased here as well. For instance, it is possible that we missed essential concepts that are additionally required when using our UIA. In **Q5.1**, we evaluated the functionality by counting the amount of successfully annotated and propagated uncertainties. It is conceivable that some of the annotated uncertainties should be assigned to other element types that we do not know yet. Another threat occurs whenever an existing architecture is modelled with a different ADL, since modeling intentionally and unintentionally abstracts from the actual architecture. This is also the case with the PCM-based architecture of the CWA which is used to evaluate **Q4.2** and **Q6.1**. Other software architects might have interpreted the available information differently, so that the architecture model would be different and consequently produces other results.

We see the biggest threat to internal validity in determining the accuracy of the UIA (**Q6.1**), since this is based on uncertainties we instantiated and grouped into scenarios ourselves. Here, it is conceivable that we have annotated uncertainties on exactly the architecture elements for which the results are good. In addition it is possible that we have unintentionally sorted out elements that are actually affected, even though the necessary information is available, or that we lack knowledge that would be necessary to identify further elements. Both cannot be excluded, so that a recall of less than 100% is possible.

The *external validity* refers to the generalizability of the evaluation results with regard to other contexts [9]. The first threat occurs as we use the *reference set* to discuss the structural quality (**Q1.1 - Q1.4**) of the *uncertainty template*. It is of limited use, because it was exploited for the definition of the categories. Nevertheless, we assume that the reference set is representative, since it is based on several sources and has been extended by further uncertainties even after the category definition process was completed. To further mitigate this threat, we additionally used a second template to evaluate those question, which is based on the real-world case study Corona Warn App (CWA).

In **Q2.1** and **Q2.2** we illustrated the applicability of our *uncertainty type derivation approach* by exemplarily applying it to the case study. Usually, applicability is evaluated by conducting user studies from which we had to refrain due to time constraints. In **Q4.1** we try to quantify knowledge, which is a difficult task in general [29]. Again, we make statements that need to be verified in a user study. In particular, this concerns the statement that our analysis is probably easy to understand and use.

Further, the results of the accuracy of the analysis (**Q6.1**) cannot be generalised, because we do not consider the scenarios to be complete, and we may lack information to determine actual false negatives. Furthermore, manual analysis depends on the experience of the person performing the analysis. Although we have accumulated profound knowledge in the course of the elaboration of this thesis, it is therefore possible that other architects obtain different and possibly more accurate results.

The biggest threat to *external validity* is the exclusive use of the CWA case study. Due to the large number of evaluation questions, we were not able to include further case studies. So, strictly speaking, our evaluation results are only valid for this specific case study in the domain of CTS. However, we assume the CWA case study to be sufficiently suitable and comprehensive for the evaluation, as already discussed in Section 8.2. The generalizability must therefore be validated using further case studies from other domains.

Construct validity describes the suitability of our questions and metrics to contribute to the overreaching goals. To mitigate this threat, we elaborated a comprehensive Goal-Question-Metric plan. Even if individual questions or metrics prove to be unsuitable, this should be compensated for by their number. Nevertheless, it turned out that some aspects cannot be fully evaluated without an additional user study which could not be carried out due to time constraints.

Lastly *reliability* refers to the repeatability of the evaluation so that other researchers can understand and repeat our evaluation. This makes it possible to verify and, in the best case, disprove that the evaluation results are dependent on the researcher carrying out the evaluation. To this end, before discussing the results in Section 8.4, we have presented the evaluation design in Section 8.3. Further, we publish the evaluation data, including the relevant architecture models, uncertainty-related models, and a prototypical implementation of the UIA [10] .

8.6. Assumptions and Limitations

In this section, we discuss assumptions and limitations that we made during the design and the implementation of our *uncertainty template* and the UIA. The first assumption concerns the *uncertainty impact derivation approach*. Software architects either have the ability to collect ADDs or a collection is provided by another source. The second assumption concerns the correctness of the input models for the propagation, in particular the absence of cycles.

The first limitation is related to the categories that form the basis of the *uncertainty template*. Their purpose is limited to the categorization of uncertainties at design time in CBSA and with a potential impact on the confidentiality of the overall system. The second limitation is due to the usage of the PCM as ADL, as it does not support some CBSA concepts out of the box. This limitation is somewhat mitigated by the definition of further architectural elements. However, we consider the current list to be incomplete because, for example, Inner Component Instances are not yet supported, i.e. no uncertainties can be annotated to them. Also, it is not the goal of this thesis to provide a mature tool but rather prototypical proof-of-concept implementation. A further limitation therefore relates to the lack of some propagation algorithms, which means that not every uncertainty can be propagated (cf. Section 7.7).

8.7. Summary

We close the discussion by summarizing the evaluation results briefly. We evaluated the structural quality (**G1**), the applicability (**G2**) and the purpose (**G3**) of the uncertainty template, including the architectural- and uncertainty related categories, and the proposed uncertainty impact derivation approach. We further evaluated the usability (**G4**), the functionality (**G5**) and the accuracy (**G6**) of our Uncertainty Impact Analysis (UIA).

Goal **G1** indicated a good structural quality of the proposed architectural- and uncertainty related categories which form the basis for the uncertainty template. In question **Q1.1**, we analysed the orthogonality of the categories with regard to their statement of impact. We have listed some category pairs that are potentially orthogonal to each other, but explained why this does not decrease the structural quality. In question **Q1.2**, we discussed the level of abstraction within the categories, i.e. the level of abstraction of the respective options. With one exception, we have concluded that the options for each category are defined at the same level. We demonstrated the exhaustiveness of the options **Q1.3** using a real-world case study with sufficient results. In question **Q1.4**, we discussed the differentiability of the options. Although for three categories it was not always clear which option to choose, we do not assume that the options as a whole are ambiguously defined. We conducted the evaluation of **G1** on an argumentative basis and by applying it to a case study. Altogether, we concluded that the *uncertainty template* has an adequate structural quality. Nevertheless, further case studies are necessary to discuss whether the few weaknesses are structural in nature or relate only to the specific case study.

With **G2**, we evaluated the applicability of the uncertainty template. In particular we demonstrated in **Q2.1** its suitability to derive uncertainty types from ADDs and in **Q2.2** its ability to structurally derive the direct and indirect impact on architectural element types. Although we were not able to conduct a user study due to time constraints, we demonstrated its applicability based on a real-world case study. In particular, we have shown that in relation to the number of ADDs and uncertainties investigated, we were able to uncover several non-trivial indirect relationships. The evaluation also revealed that in a few cases it was not possible to determine the impact because some uncertainty types could not initially be assigned to an architectural element. Overall, the evaluation results are promising for this one case study. Again, the evaluation has to be reproduced with other case studies.

Goal **G3** evaluated the purpose and the relevance of the uncertainty template. In particular, we demonstrated in **Q3.1** how the uncertainty template enables reuse of knowledge and even creates awareness. Furthermore, in **Q3.2**, we illustrated the relevance of our approach by showing that it can classify uncertainties in software architectures more accurately and thus make more precise statements about their impact on the system.

With **G4**, we illustrated how the UIA reduces the required amount of knowledge when analysing the impact of uncertainties compared to a manual, unstructured approach. Although the discussion in **Q4.1** showed that the necessary knowledge is not negligible, at least for experts doing the preliminary work, it nevertheless showed that for the first time less experienced users can analyse uncertainties and their effects without sound knowledge regarding uncertainties and their propagation effects. Furthermore, we have shown in **Q4.2** that by using the UIA, the proportion of architectural elements to be investigated in a specific case study is reduced by 85%. The limitation of the validity of the results due to a potentially incomplete *impact set* is removed by the results of **Q6.1**, as we achieved a recall of 100% for the *impact set* under investigation. Although these results still need to be confirmed by further case studies, they indicate a high level of usability in terms of a significant reduction in the necessary work.

Goal **G5** evaluated the functionality of the UIA. We demonstrated in **Q5.1** that the UIA is capable to annotate and propagate uncertainties in software architectures. We demonstrated this for a specific case study in the domain of CTS. Unfortunately, the current prototypical implementation lacks propagation algorithms, so that some uncertainties cannot be propagated. However, these can presumably be implemented with little effort. Altogether, we concluded that the UIA is conceptually capable to annotate and propagate uncertainties in software architectures.

With **G6**, we evaluated the accuracy of the propagation results. Therefore, we discussed in **Q6.1** the functional accuracy and compare the *impact set* with a manually defined *affected set*. With regard to the functional accuracy, the evaluation results are highly satisfactory due to precision and recall values of 100% Therefore, we assume our analysis to be at least functional correct. The comparison of the *impact set* and the *affected set* is based on four scenarios that consist of several uncertainties. For these scenarios, we obtained precision values between $\sim$ 44% and $\sim$ 90%, but a recall of 100%. Despite overestimation due to some *false-positives*, the UIA found at least all the architectural elements actually affected for these four scenarios. However, further case studies are necessary to verify these results in other domains and for other systems.

8.8. Data Availability

The evaluation data is publicly available [10]. This includes the various uncertainty- and PCM- related models, as well as the processed and detailed results of the respective evaluation findings. To reproduce our evaluation, we also include the current prototypical implementation, including an installation guide and sample data.

9. Conclusion

To conclude this thesis, we summarize our approach and findings in Section 9.1. In Section 9.2, we provide an outlook for future work. Lastly, we express our acknowledgements in Section 9.3.

9.1. Summary

The goals of this thesis were twofold. The first goal was to provide an *uncertainty template* that guides software architects through the process of identifying and analysing uncertainties in CBSAs. The second goal was to provide support for uncertainty annotation and propagation to reduce manual effort and required expertise when analysing the impact of uncertainties. Our research questions are divided along these contributions.

Our *uncertainty template* consists of categories to categorize ADDs and uncertainties in CBSAs. The categories are inspired by existing architectural- and uncertainty related taxonomies. We have discussed and analysed existing categories in detail to assess whether they can be discarded, adapted or adopted. In most cases, we had to adapt or discard the categories for our purposes, as existing approaches do not intend to categorise uncertainties at design time. The result is a template consisting of 15 categories and 44 options. Based on that template, we defined a structured *uncertainty type derivation approach* to extract types of uncertainties and their impact on architectural element types based on existing ADDs. To that end, we defined the concept of *uncertainty propagation*, which occurs if directly annotated uncertainties indirectly affect other elements in the software architecture. So the approach enables software architects to iteratively extract several uncertainty types and their impact within a domain of interest. Once extracted, those types are potentially relevant for several architectures within that domain. Hence, the approach enables the reusability and accessibility of knowledge about uncertainty types and their impact on types of architectural elements.

At instance level, this can be used to determine all potentially affected architectural elements of a concrete architecture for actual uncertainties. Based on the idea of *change impact analyses*, we defined the concepts of the *uncertainty set*, the *affected set* and the *impact set* where the *impact set* is an overestimation of the actually affected architectural elements. Thus, it includes the directly affected architectural elements (via annotation) and the potentially indirectly affected architectural elements (via propagation). The results of the *uncertainty type derivation approach* therefore enable less experienced software architects to identify the *impact set* without profound knowledge. More specifically, software architects instantiate types of uncertainties and extract the potentially affected elements based on the previously defined type-level impacts. However, this needs to be done manually, i.e. without tool support.

The second contribution is therefore an Uncertainty Impact Analysis (UIA) which supports the annotation and propagation of uncertainties in software architectures. To that end, we extended an Architectural Description Language (ADL) called Palladio Component Model (PCM) by the capability to express uncertainties as first-class entity. More experienced architects which we refer to as *expert architects* use the *uncertainty type derivation approach* to identify uncertainty types for a domain of interest. This information, which can be reused within a domain, serves as input for the UIA. Based on a selected uncertainty type, the UIA automatically proposes each element to which the uncertainty can be assigned. In addition to that, we elaborated structural propagation rules which aim to improve the overestimation, i.e. to reduce the *impact set.* The analysis helps insofar as the manually tedious structural uncertainty propagation is automated. By separating the identification of uncertainty types and the instantiation for a concrete architecture, the analysis allows less experienced architects, which we refer to as *user architects*, to perform the analysis of the impact of uncertainties without profound knowledge on existing types of uncertainties, their characteristics, and especially knowledge on how they might propagate through the architecture and affect other architectural element. The assessment of the impact is further supported by guidelines based on the combination of categories and counting metrics.

We evaluated the *uncertainty template* including the categories and the *uncertainty type derivation approach* with regard to structural quality, applicability and purpose. Regarding the structural quality, we demonstrated the compliance with classification principles such as orthogonality, level of abstraction, exhaustiveness and differentiability. We further illustrated the applicability of the *uncertainty type derivation approach* based on a current real-world case study. Regarding the purpose, we evaluated the relevance of our approach and how it enables reuse of knowledge and the creation of awareness.

We further evaluated the UIA with regard to usability, functionality and accuracy. Concerning the usability, we compared the required knowledge of a manual uncertainty impact analysis with the required knowledge using our analysis. Although the preliminary work to identify the uncertainty types still requires a non negligible amount of expertise, the analysis reduces the required knowledge from the *user architect's* point of view. Regarding the functionality, our evaluation revealed that the current state of the analysis lacks some uncertainty propagation algorithms to successfully propagate all possible types of uncertainties. However, this gap can be closed with additional implementation effort. The overall functionality of our prototypical implementation is satisfactory. Lastly, we evaluated the accuracy UIA based on previously defined scenarios that consist of instantiated uncertainties. For each scenario, we manually determined the *affected set* and compared it to the *impact set* calculated by the UIA. The results show a recall of 100% while maintaining a precision between $\sim$ 44% and $\sim$ 90% depending on the scenario. These results are satisfactory because the analysis for the scenarios at least automatically identifies all the elements actually affected while being sufficiently precise.

The thesis aimed to provide an architecture-based uncertainty impact analysis for confidentiality to make uncertainties manageable already at design time. For this purpose, we have developed a template and an analysis to analyse uncertainties at both type and instance level. In particular, this separation allows architects for the first time to analyse and evaluate the impact of uncertainties without profound knowledge.

9.2. Future Work

In addition to structural propagation rules, it would be conceivable to integrate data flow information in order to further improve the overestimation, i.e. the *impact set*, in a second propagation step. To this end, it is necessary to examine the extent to which the data flow has an impact on the propagation of uncertainties. We further recommend the extension of the supported architecture element types.

The prototypical implementation presented in this thesis is operational in its current state. Nevertheless, there are a some places that could be improved or extended in the future. The UIA could be extended by automatically checking the architecture models. This allows the assumption of correct models to be discarded. The current state of the UIA exemplarily provides the possibility to sort uncertainties regarding the amount of elements on which they have an impact. Comparable metrics and the combination of the categories of the uncertainty types should be elaborated to generate recommendations for action. This would allow the tool to automatically determine which uncertainties need to be resolved next. The current implementation should be complemented by the remaining structural propagation algorithms to allow propagation of all representable uncertainties. This is labour-intensive, but probably easy to implement.

Also, we propose to create and validate a reference set consisting of uncertainty types that are relevant across domains. This could serve as a starting point for an *expert architect* in filling a domain-specific uncertainty template. Lastly, the quality of the *uncertainty template* and the UIA could be enhanced by executing a user study. Possible, still unknown shortcomings could be discovered and eliminated in this way. In particular, this will be useful to provide further empirical results on the suitability for supporting the analysis of the impact of uncertainties at architectural level.

9.3. Acknowledgements

I want to thank my advisors Sebastian Hahner and Maximilian Walter for the very enlightening meetings and their continuous support. This has sustainably improved the quality of this thesis. Further, I also want to thank Dr.-Ing. Erik Burger for his advice on meta-modeling and Sandro Koch for his introduction on change impact analyses.

Bibliography

[1] Aynur Abdurazik. *Coupling-based analysis of object-oriented software.* Vol. 68. 03. Citeseer, 2007.

[2] Russell L Ackoff. "From data to wisdom". In: *Journal of applied systems analysis* 16.1 (1989), pp. 3–9.

[3] Zoya Alexeeva, Diego Perez-Palacin, and Raffaela Mirandola. "Design decision documentation: A literature overview". In: *European Conference on Software Architecture.* Springer. 2016, pp. 84–101.

[4] Phillip G. Armour. "The Five Orders of Ignorance". In: *Commun. ACM* 43.10 (Oct. 2000), pp. 17–20. ISSN: 0001-0782. DOI: 10.1145/352183.352194. URL: https://doi.org/10.1145/352183.352194.

[5] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. "The goal question metric approach". In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[6] Victor R Basili and David M Weiss. "A methodology for collecting valid software engineering data". In: *IEEE Transactions on software engineering* 6 (1984), pp. 728–738.

[7] Matthias Becker, Markus Luckey, and Steffen Becker. "Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time". In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures.* QoSA '13. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2013, pp. 43–52. ISBN: 9781450321266. DOI: 10.1145/2465478.2465489. URL: https://doi.org/10.1145/2465478.2465489.

[8] Denise Bedford. "Evaluating classification schema and classification decisions". In: *Bulletin of the American Society for Information Science and Technology* 39.2 (2013), pp. 13–21.

[9] Niko Benkler. "An Approach for Identifying Microservices using Clustering on Control Flow and Data Flow". Bachelor's Thesis. Karlsruher Institut für Technologie (KIT), 2019. URL: http://dx.doi.org/10.13140/RG.2.2.24676.86403 (visited on 08/23/2021).

[10] Niko Benkler. *Data Set for Master Thesis "Architecture-based Uncertainty Impact Analysis for Confidentiality".* URL: https://doi.org/10.5281/zenodo.6202288 (visited on 02/23/2022).

[11] Manoj Bhat et al. "The Evolution of Architectural Decision Making as a Key Focus Area of Software Architecture Research: A Semi-Systematic Literature Study". In: *2020 IEEE International Conference on Software Architecture (ICSA).* 2020, pp. 69–80. DOI: 10.1109/ICSA47634.2020.00015.

[12] Barry Boehm and Victor R. Basili. “Software Defect Reduction Top 10 List”. In: *Computer* 34.1 (Jan. 2001), pp. 135–137. ISSN: 0018-9162. DOI: `10.1109/2.962984`. URL: `https://doi.org/10.1109/2.962984`.

[13] Nicolas Boltz, Maximilian Walter, and Robert Heinrich. “Context-Based Confidentiality Analysis for Industrial IoT”. In: Portoroz, Slovenia. Portoroz, Slovenia: IEEE, 2020, pp. 589–596. ISBN: 978-1-7281-9533-9. DOI: `10.1109/SEAA51224.2020.00096`.

[14] Tomas Bures et al. “Capturing dynamicity and uncertainty in security and trust via situational patterns”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2020, pp. 295–310.

[15] Erik Burger. *MDSD WS2021 VL02*. URL: `https://svnserver.informatik.kit.edu/i43/svn/%20lehre/WS2021_Vorlesung_MDSD/MDSD_WS_2021_VL02/%20VL_MDSD_WS1920_VL02.tex` (visited on 07/26/2021).

[16] J. Cámara et al. “Uncertainty in Self-Adaptive Systems: Categories, Management, and Perspectives”. In: 2017.

[17] Carlos Carrillo and Rafael Capilla. “Ripple Effect to Evaluate the Impact of Changes in Architectural Design Decisions”. In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. ECSA ’18. Madrid, Spain: Association for Computing Machinery, 2018. ISBN: 9781450364836. DOI: `10.1145/3241403.3241446`. URL: `https://doi.org/10.1145/3241403.3241446`.

[18] Shu-Jen Chen and Ching-Lai Hwang. “Fuzzy multiple attribute decision making methods”. In: *Fuzzy multiple attribute decision making* (1992), pp. 289–486.

[19] Shang-wen Cheng and David Garlan. “Handling uncertainty in autonomic systems”. In: *In International Workshop on Living with Uncertainties*. 2007.

[20] *CVSS v3.1 Specification Document*. URL: `https://www.first.org/cvss/v3.1/specification-document#2-3-Impact-Metrics` (visited on 07/14/2021).

[21] Naeem Esfahani and Sam Malek. “Uncertainty in Self-Adaptive Software Systems”. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 214–238. ISBN: 978-3-642-35813-5. DOI: `10.1007/978-3-642-35813-5_9`. URL: `https://doi.org/10.1007/978-3-642-35813-5_9`.

[22] Naeem Esfahani, Sam Malek, and Kaveh Razavi. “GuideArch: Guiding the Exploration of Architectural Solution Space under Uncertainty”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 43–52. ISBN: 9781467330763.

[23] Naeem Esfahani, Kaveh Razavi, and Sam Malek. “Dealing with Uncertainty in Early Software Architecture”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. Cary, North Carolina: Association for Computing Machinery, 2012. ISBN: 9781450316149. DOI: `10.1145/2393596.2393621`. URL: `https://doi.org/10.1145/2393596.2393621`.

[24] Michalis Famelis and Marsha Chechik. “Managing Design-Time Uncertainty”. In: MODELS ’17. Austin, Texas: IEEE Press, 2017, p. 179. ISBN: 9781538634929. DOI: 10.1109/MODELS.2017.24. URL: https://doi.org/10.1109/MODELS.2017.24.

[25] Marta J Fernandez and Caroline M Eastman. “Basic taxonomic structures and levels of abstraction”. In: *Advances in Classification Research Online* 1.1 (1990), pp. 59–70.

[26] David Garlan. “Software Engineering in an Uncertain World”. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research.* FoSER ’10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 125–128. ISBN: 9781450304276. DOI: 10.1145/1882362.1882389. URL: https://doi.org/10.1145/1882362.1882389.

[27] Sebastian Gerdes, Mohamed Soliman, and Matthias Riebisch. “Decision buddy: tool support for constraint-based design decisions during system evolution”. In: *2015 1st International Workshop on Future of Software Architecture Design Assistants (FoSADA).* 2015, pp. 1–6.

[28] Sebastian Hahner. “Dealing with Uncertainty in Architectural Confidentiality Analysis”. In: *Proceedings of the Software Engineering 2021 Satellite Events.* GI Software Engineering Tagung. SE 2021 (Online, Feb. 22–26, 2021). 46.23.03; LK 01. Gesellschaft für Informatik e.V. (GI), 2021, pp. 1–6.

[29] Sebastian Hahner. “Domain-specific Language for Data-driven Design Time Analyses and Result Mappings for Logic Programs”. MA thesis. Karlsruher Institut für Technologie (KIT), 2020. 138 pp. DOI: 10.5445/IR/1000123271.

[30] Sebastian Hahner et al. “Modeling Data Flow Constraints for Design-Time Confidentiality Analyses”. In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C).* IEEE. 2021, pp. 15–21.

[31] Sara M. Hezavehi, Danny Weyns, and Paris Avgeriou. “A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements”. In: *Managing Trade-Offs in Adaptable Software Architectures.* Elsevier, 2017, pp. 45–77.

[32] Sara M. Hezavehi et al. “Uncertainty in Self-Adaptive Systems: A Research Community Perspective”. In: *ACM Trans. Auton. Adapt. Syst.* 15.4 (Dec. 2021). ISSN: 1556-4665. DOI: 10.1145/3487921. URL: https://doi.org/10.1145/3487921.

[33] Presse- und Informationsamt der Bundesregierung (BPA). *Veröffentlichung der Corona Warn App.* URL: https://www.bundesregierung.de/breg-de/themen/coronavirus/veroeffentlichung-der-corona-warn-app-1760892 (visited on 01/12/2022).

[34] *Is it time to leave WhatsApp – and is Signal the answer?* Jan. 2021. URL: https://www.theguardian.com/technology/2021/jan/24/is-it-time-to-leave-whatsapp-and-is-signal-the-answer (visited on 07/26/2021).

[35] *Information technology — Security techniques — Information security management systems — Overview and vocabulary.* Tech. rep. Geneva, CH: International Organization for Standardization, Feb. 2018.

[36] Ivar Jacobson and Stefan Bylund. *The road to the unified software development process.* Vol. 18. Cambridge University Press, 2000.

[37] Anton Jansen and Jan Bosch. "Software architecture as a set of architectural design decisions". In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05).* IEEE. 2005, pp. 109–120.

[38] Xiao Jing et al. "Modeling Uncertain and Imprecise Information in Process Modeling with UML". In: Jan. 2008, pp. 237–240.

[39] SAP Deutschland SE & Co. KG. *Corona Warn App: Kennzahlen.* URL: https://www.coronawarn.app/de/analysis/ (visited on 01/12/2022).

[40] SAP Deutschland SE & Co. KG. *Official Website of the Corona Warn App.* URL: https://www.coronawarn.app/en/ (visited on 01/12/2022).

[41] Anne Koziolek. "Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes". PhD thesis. 2011. DOI: 10.5445/IR/1000024955.

[42] Per Kroll and Philippe Kruchten. *The rational unified process made easy: a practitioner's guide to the RUP.* Addison-Wesley Professional, 2003.

[43] Philippe Kruchten. "An Ontology of Architectural Design Decisions in Software-Intensive Systems". In: *2nd Groningen Workshop on Software Variability* (Jan. 2004).

[44] Chawanangwa Lupafya. "A Framework for Managing Uncertainty in Software Architecture". In: *Proceedings of the 13th European Conference on Software Architecture - Volume 2.* ECSA '19. Paris, France: Association for Computing Machinery, 2019, pp. 71–74. ISBN: 9781450371421. DOI: 10.1145/3344948.3344954. URL: https://doi.org/10.1145/3344948.3344954.

[45] Ioanna Lytra and Uwe Zdun. "Supporting Architectural Decision Making for Systems-of-Systems Design under Uncertainty". In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems.* SESoS '13. Montpellier, France: Association for Computing Machinery, 2013, pp. 43–46. ISBN: 9781450320481. DOI: 10.1145/2489850.2489859. URL: https://doi.org/10.1145/2489850.2489859.

[46] Steve McConnell. *Software Project Survival Guide.* USA: Microsoft Press, 1997. ISBN: 1572316217.

[47] Gabriel A Moreno et al. "Proactive self-adaptation under uncertainty: a probabilistic model checking approach". In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering.* 2015, pp. 1–12.

[48] Paula Muñoz, Priyanka Karkhanis, and Mark Brand. "Modeling Objects with Uncertain Behaviors." In: *The Journal of Object Technology* 20 (Jan. 2021), 8:1. DOI: 10.5381/jot.2021.20.3.a8.

[49] Angelika Musil et al. “Patterns for Self-Adaptation in Cyber-Physical Systems”. In: *Multi-Disciplinary Engineering for Cyber-Physical Production Systems: Data Models and Software Solutions for Handling Complex Engineering Projects*. Ed. by Stefan Biffl, Arndt Lüder, and Detlef Gerhard. Cham: Springer International Publishing, 2017, pp. 331–368. ISBN: 978-3-319-56345-9. DOI: 10.1007/978-3-319-56345-9_13. URL: https://doi.org/10.1007/978-3-319-56345-9_13.

[50] Evangelos Ntentos et al. “Supporting architectural decision making on data management in microservice architectures”. In: *European Conference on Software Architecture*. Springer. 2019, pp. 20–36.

[51] Diego Perez-Palacin and Raffaela Mirandola. “Dealing with Uncertainties in the Performance Modelling of Software Systems”. In: *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA ’14. Marcq-en-Bareul, France: Association for Computing Machinery, 2014, pp. 33–42. ISBN: 9781450325769. DOI: 10.1145/2602576.2602582. URL: https://doi.org/10.1145/2602576.2602582.

[52] Diego Perez-Palacin and Raffaela Mirandola. “Uncertainties in the Modeling of Self-Adaptive Systems: A Taxonomy and an Example of Availability Evaluation”. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE ’14. Dublin, Ireland: Association for Computing Machinery, 2014, pp. 3–14. ISBN: 9781450327336. DOI: 10.1145/2568088.2568095. URL: https://doi.org/10.1145/2568088.2568095.

[53] David Powers and Ailab. “Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation”. In: *J. Mach. Learn. Technol* 2 (Jan. 2011), pp. 2229–3981. DOI: 10.9735/2229-3981.

[54] Andres J. Ramirez, Adam C. Jensen, and Betty H. C. Cheng. “A taxonomy of uncertainty for dynamically adaptive systems”. In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2012, pp. 99–108. DOI: 10.1109/SEAMS.2012.6224396.

[55] Christoph Rathfelder and Benjamin Klatt. “Palladio Workbench: A Quality-Prediction Tool for Component-Based Architectures”. In: *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*. 2011, pp. 347–350. DOI: 10.1109/WICSA.2011.55.

[56] Ralf Reussner et al. *The Palladio Component Model*. Tech. rep. 14. Karlsruher Institut für Technologie (KIT), 2011. 193 pp. DOI: 10.5445/IR/1000022503.

[57] Kiana Rostami et al. “Architecture-Based Assessment and Planning of Change Requests”. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA ’15. Montréal, QC, Canada: Association for Computing Machinery, 2015, pp. 21–30. ISBN: 9781450334709. DOI: 10.1145/2737182.2737198. URL: https://doi.org/10.1145/2737182.2737198.

[58] Kiana Rostami et al. “Architecture-based change impact analysis in information systems and business processes”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE. 2017, pp. 179–188.

[59] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14 (2008), pp. 131–164.

[60] Carlos Carrillo Sanchez, Rafael Capilla, and Miroslaw Staron. "Estimating the Complexity of Architectural Design Decision Networks". In: *IEEE Access* 8 (2020), pp. 168558–168575. DOI: `10.1109/ACCESS.2020.3023608`.

[61] Stephan Seifermann, Robert Heinrich, and Ralf Reussner. "Data-Driven Software Architecture for Analyzing Confidentiality". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019, pp. 1–10. DOI: `10.1109/ICSA.2019.00009`.

[62] Stephan Seifermann et al. "Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams". In: *J. Syst. Softw.* 184.C (Feb. 2022). ISSN: 0164-1212. DOI: `10.1016/j.jss.2021.111138`. URL: `https://doi.org/10.1016/j.jss.2021.111138`.

[63] Mojtaba Shahin, Peng Liang, and Mohammad Reza Khayyambashi. "Architectural design decision: Existing models and tools". In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*. 2009, pp. 293–296. DOI: `10.1109/WICSA.2009.5290823`.

[64] Michael G Solomon and Mike Chapple. *Information security illuminated*. Jones & Bartlett Publishers, 2004.

[65] Herbert Stachowiak. *Allgemeine modelltheorie*. Springer, 1973, pp. 131–132.

[66] Johannes Stammel and Ralf Reussner. "Kamp: Karlsruhe architectural maintainability prediction". In: *Proceedings of the 1. Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2):" Design for Future-Langlebige Softwaresysteme*. 2009, pp. 87–98.

[67] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[68] SAP Deutschland SE & Deutsche Telekom. *Corona Warn App: Documentation*. URL: `https://github.com/corona-warn-app/cwa-documentation` (visited on 01/12/2022).

[69] SAP Deutschland SE & Deutsche Telekom. *Corona Warn App: Repositories*. URL: `https://github.com/corona-warn-app` (visited on 01/12/2022).

[70] Michael Tobias. *Datensatz zur Bachelor-Arbeit "Evaluierung architektureller Datenflussanalyse mittels einer Fallstudie anhand der Corona-Warn-App"*. URL: `https://doi.org/10.5281/zenodo.4631213` (visited on 01/10/2022).

[71] Michael Tobias. "Evaluierung architektureller Datenflussanalyse mittels einer Fallstudie anhand der Corona-Warn-App". Abschlussarbeit - Bachelor. Karlsruher Institut für Technologie (KIT), 2021. 147 pp.

[72] Javier Troya et al. "Uncertainty representation in software models: a survey". In: *Software and Systems Modeling* 20 (Aug. 2021). DOI: `10.1007/s10270-020-00842-1`.

[73] Warren Walker et al. "Defining Uncertainty: A Conceptual Basis for Uncertainty Management in Model-Based Decision Support". In: *Integrated Assessment* 4 (Mar. 2003). DOI: `10.1076/iaij.4.1.5.16466`.

[74] Maximilian Walter, Stephan Seifermann, and Robert Heinrich. “A Taxonomy of Dynamic Changes Affecting Confidentiality”. In: *11th Workshop Design For Future – Langlebige Softwaresysteme*. 11. Workshop "Design For Future - Langlebige Softwaresysteme" (DFF 2020) - 22. Workshop "Software-Reeingineering and Evolution". WSRE 2020 (Online, Sept. 16–18, 2020). 2020.

[75] Man Zhang et al. “Uncertainty-wise cyber-physical system test modeling”. In: *Software & Systems Modeling* 18.2 (2019), pp. 1379–1418.

A. Appendix

A.1. Uncertainty Assessment - Example

In Section 6.6 we proposed how it is possible to assess uncertainties at category level, i.e. how it is possible to compare two uncertainties of different uncertainty types based on selected options for the respective categories. Further, we presented how using the UIA can contribute to the assessment of instantiated uncertainties. We illustrate this assessment process based on the uncertainties ICU1 - ICU28, which are instantiated and assigned to elements of the CWA case study (cf. Section 8.2). First, we asses the uncertainties ICU1 - ICU28 based on their types, respectively by comparing the options of each category. In addition, we assess their impact by using the results of the UIA.

The uncertainties, the template, architecture models and uncertainty-related models are available in our data set [10]. The goal is to exemplarily illustrate how to asses which uncertainties should to be given higher priority, because they might have a more severe impact on the confidentiality than others.

We start by combining the options for categories *Impact on Confidentiality* and *Severity of the Impact* of the respective types of uncertainties CU1-CU28. The following uncertainties correspond to types which have both, *direct* and *high* impact on confidentiality: ICU3, ICU20, ICU22, ICU23, ICU24, ICU26. In a next step we combine the categories *Manageability*, *Resolution Time* and *Root Cause* to further compare the selected uncertainties. Whereas ICU3, ICU20, ICU23, ICU26 are either irreducible or partial reducible, and resolved during deployment time, runtime or never, ICU22 and *ICU24* are fully resolvable during design time due to missing information. Those two uncertainties are therefore candidates which could and should be resolved next. The remaining uncertainties are not resolvable during design time but have a direct, and high impact on the confidentiality. Consequently, they should be carefully tracked.

In addition to the assessment of uncertainties at category level, the UIA provides the possibility to prioritize instantiated uncertainties based on counting metrics such as the number of affected elements, or the amount of instantiated uncertainties per type. The prototypical implementation already provides the possibility to sort the instantiated uncertainties according to the number of elements they affect. In doing so, the UIA (automatically) determines the first three positions: ICU17 (6 elements), ICU20 (5 elements) and ICU3 (5 elements). In combination with the findings of the category-based assessment, it is thus particularly striking that ICU20 is a candidate that should be prioritised higher, not only at type level but also at instance level.

A.2. Evaluation Data

This section provided data such as tables and illustrations that we use throughout our evaluation.

A.2.1. Reference Set

Table A.1 illustrates our reference set in a highly abbreviated version. Due to space constraints, the following information is omitted: i) the architectural-related categories, ii) the uncertainty-related categories apart from category *Architectural Element Type*, iii) names and affected architectural element of duplicate uncertainty types. The completely filled uncertainty template is available in our data set [10]. To ensure distinct identification throughout the evaluation, the ADDs and uncertainty types are numbered consecutively. The following rules are applied: Each id is preceded by the letter R for ***R**eference Set*, followed by A for ADD and U for uncertainty, respectively. Each uncertainty that represents an ADD has the same number, i.e. *RUx* = *RAx*. Further uncertainties are numbered consecutively, starting at the next available number. In total, the reference set consists of 12 ADDs (RA1 - RA12) and 24 uncertainties (RU1 - RU24).

				...	*Uncertainty related Categories*
ADD	**ID**		**Uncertainty Type**	...	**Architectural Element Type**
Distribution (System)	RA1	RU1	What is the distribution?	...	n/a
Communication Encryption	RA2	RU2	How is communicated?	...	Communication
		RU3	...	...	...
		RU13	Is confidential data transmitted?	...	Interface
		RU5	...	...	...
		RU1	...	...	...
Deployment Location	RA3	RU3	Where is deployed?	...	Hardware Resource
		RU14	Is the deployment provider trustworthy?	...	Hardware Resource
		RU15	Is user data stored/processed?	...	Component
		RU2	...	...	...

Persistence of Data (What)	RA4	RU4	What data is persisted?	...	Component
		RU16	Is the data to be persisted confidential?	...	Component
		RU5	...	...	...
		RU10	...	...	...
		RU11	...	...	...
Persistence of Data (How)	RA5	RU5	How is data persisted?	...	Component
		RU3	...	...	
		RU4	...	...	...
Authentication	RA6	RU6	Which authentication mechanisms are chosen?	...	Interface
Authorization (AC)	RA7	RU7	What kind of access control is used?	...	Interface
		RU17	Can access control cover all matters?	...	Interface
		RU18	Are the access control rules correct?	...	Interface
Choice of Component	RA8	RU8	Which component is chosen?	...	n/a
		RU19	What is the behavior of the component?	...	Component
		RU20	Is the component currently compromised?	...	Component
		RU21	Is the component's provider trustworthy?	...	Component
Interface Access (Output)	RA9	RU9	What data is provided?	...	Interface
		RU22	Which subjects access the interface?	...	Interface
		RU7	...	...	...
		RU12	...	...	...
		RU6	...	...	...
		RU11	...	...	...

Interface Access (Input)	RA10	RU10	What data is entered?	...	Interface
		RU22	...	...	...
		RU7	...	...	...
		RU23	Is the data entered correct?	...	Usage Behaviour
		RU11	...	...	...
		RU12	...	...	...
		RU6	...	...	...
		RU24	What is the user behavior?	...	Usage Behaviour
Anonymization of data	RA11	RU11	Is the data anonymized?	...	Component
Interface Design	RA12	RU12	What is the structure of the interface?	...	Interface

Table A.1.: Uncertainty Template - Reference Set (excerpt)

A.2.2. Corona Warn App

This section provides evaluation data with regard to the CWA case study.

A.2.2.1. CWA Software Architecture

In Figure A.1, we show a simplified architecture model of the CWA, which combines the PCM-based *Component Model* and the *Assembly Model* for illustration purposes. It illustrates the previously mentioned components, as well as the composition of some components. Furthermore, it shows the "*wiring*" of the components which forms the overall structure, i.e the *assembly*. Among others, the allocation of components to hardware resources is not shown. However, it can be anticipated that each top level component is allocated to a different server. Further information such as description of hardware resources, data types, detailed interface definitions or usage models can be found at [10]. The *GitHub* project available under [69] contains all the repositories, including requirements, design documents, source code and other material. A detailed review of all the materials is already carried out by Tobias [71, 70].
We created the CWA architecture based on the following repositories:

- cwa-documentation
  High level overview to create the overall *assembly model* and *allocation model*. Scenarios to derive *usage model*. Derivation of component CondentDeliveryNetwork.

- cwa-server
  Definition of composite component CoronaWarnAppServer.

- cwa-testresult-server
  Definition component TestResultServer.

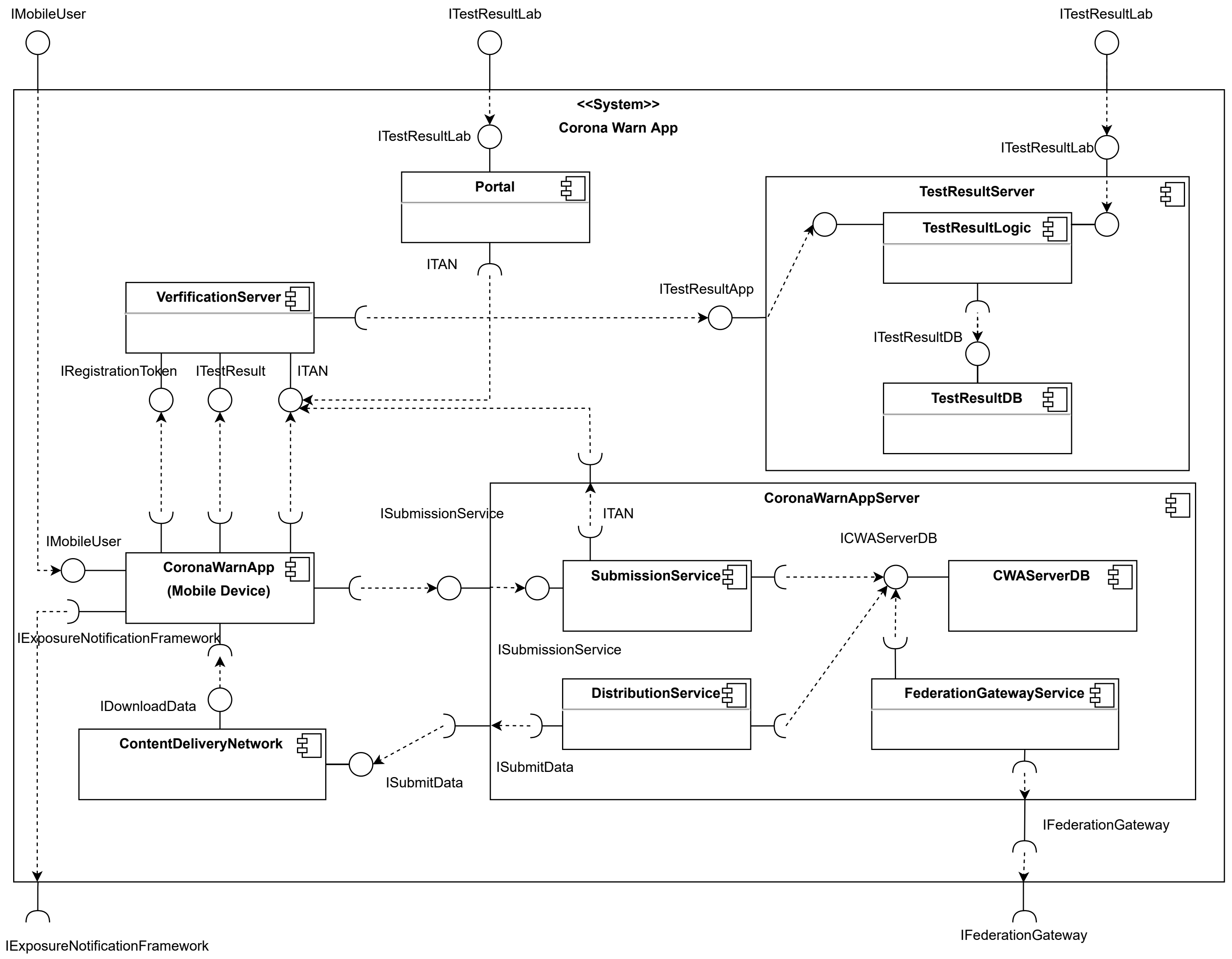


Figure A.1.: Architecture Diagram CWA

- cwa-app-android
  Definition of component CoronaWarnApp (Mobile Device).
- cwa-verfification-server
  Definition of component VerificationServer.
- cwa-verfification-portal
  Definition of component Portal.

A.2.2.2. Collection of ADDs

As indicated in Section 6.3.1, extracting ADDs from existing sources (i.e. *Step 0*) is not considered to be part of the approach. Nevertheless, for the sake of completeness, we show with some examples how we have used the work of Tobias [71, 70] and the CWA design documents to derive possible ADDs. Altogether, we were able to extract 22 ADDs - CA1 - CA22.

- *"Neither location data, nor the rolling proximity identifiers [...] are ever stored centrally."* [68]
  - The requirement indicates that it has to be decided what data shall be persisted. From this, we were able to extract the ADD *"Persistence of Data (What)"*(CA2).
- *"[...] the information that is securely stored on the end device [...]"* [68]
  - The information indicates that it has to be decided how data shall be persisted. From this, we were able to extract the ADD *"Persistence of Data (How)"*(CA4).
- *"Diagnosis keys are automatically deleted by the Corona Warning App [...]" [71]*
  - The requirement indicates that it has to be decided how long data should be stored. From this, we were able to extract the ADD *"Duration of Data Storage"*(CA10).
- *"Each country has its own separate database, which contains the keys from infected individuals. " [68]*
  - This information provided two different ADDs: *"Data Separation"*(CA13) and *"Choice of DB System"*(CA15).

A.2.2.3. CWA-based Uncertainty Template

Table A.2 illustrates the uncertainty template which we created based on the CWA documentation described previously. Due to space constraints, the following information is omitted: i) the architectural-related categories, ii) the uncertainty-related categories apart from category *Architectural Element Type*, iii) names and affected architectural element of duplicate uncertainty types. The completely filled uncertainty template is available in our data set [10]. To ensure distinct identification throughout the evaluation, the ADDs and uncertainty types are numbered consecutively. The following rules are applied: Each id is preceded by the letter C for ***C**orona Warn App*, followed by A for ADD and U for uncertainty, respectively. Each uncertainty that represents an ADD has the same number, i.e. *CUx* = *CAx*. Further uncertainties are numbered consecutively, starting at the next available number. In total, the template consists of 22 ADDs (CA1 - CA22) and 28 uncertainties (CU1 - CU28).

				...	*Uncertainty related Categories*
ADD	**ID**		**Uncertainty Type**	...	**Architectural Element Type**
Distribution (System)	CA1	CU1	What is the distribution?	...	n/a
		CU23	Correct design for secure architecture?	...	n/a
Persistence of Data (What)	CA2	CU2	What data is persisted?	...	Component
		CU3	...	...	...
		CU4	...	...	...
		CU6	...	...	...
		CU24	Is the data to be persisted confidential?	...	Component
Persistence of Data (Where)	CA3	CU3	Where is data persisted?	...	Hardware Resource
		CU2	...	...	...
		CU4	...	...	...
Persistence of Data (How)	CA4	CU4	How is data persisted?	...	...
		CU2	...	...	...
		CU3	...	...	...
		CU20	...	...	...
		CU24	Is the data to be persisted confidential?	...	Component
Data Processing (Where)	CA5	CU5	...	...	...
		CU10	...	...	...
Interface Access (Input)	CA6	CU6	What data is entered?	...	Interface
		CU8	...	...	...
		CU11	...	...	...

Interface Access (Output)	CA7	CU7	What data is provided?	...	...
		CU8	...	...	...
		CU11	...	...	...
		CU16	...	...	...
		CU22	...	...	...
Interface Design	CA8	CU8	What is the structure of the interface?	...	Interface
		CU6	...	...	...
		CU7	...	...	...
		CU17	...	...	...
Data Minimization	CA9	CU9	How is meta-data handled?	...	Component
		CU10	...	...	...
Duration of Data Storage	CA10	CU10	How long is data being stored?	...	Component
		CU6	...	...	...
User Identification	CA11	CU11	How is user identification performed?	...	Interface
Communication Encryption	CA12	CU12	How is communicated?	...	Communication
		CU1	...	...	...
		CU3	...	...	...
		CU4	...	...	...
		CU5	...	...	...
		CU20	...	...	...
		CU25	Is communication intercepted?	...	Communication
		CU26	Is data manipulated?	...	Communication
Data Separation	CA13	CU13	Is data separation performed?	...	Component
		CU1	...	...	...
		CU12	...	...	...
Logging	CA14	CU14	What data is logged?	...	Component
		U3	...	...	

Choice DB system	CA15	CU15	What kind of storage is used?	...	Hardware Resource
		CU2	...	...	
		CU3	...	...	...
Authentication	... CA16	CU16	Which authentication mechanisms are used?	...	Interface
		CU26	Is data manipulated?	...	Interface
Choice UI Pattern	CA17	CU17	Which UI pattern is used?	...	n/a
Choice of Component	CA18	CU18	Which component is chosen?	...	n/a
		CU27	Is the component insecure due to software bugs?		Component
Input Validation	... CA19	CU19	Is input validated?		Interface
		CU6	...	...	...
		CU26	...		
		CU28	Are SQL Injection performed?		Usage Behaviour
Deployment Location	CA20	CU20	Where is deployed?		Hardware Resource
		CU2	...		
		CU12	...		
Virtualization	CA21	CU21	Is virtualization used?		Hardware Resource
		CU20	...	...	...
Anonymization of data	CA22	CU22	Is the data anonymised?		Component
		CU6	...	...	...

Table A.2.: Uncertainty Template - Corona Warn App (excerpt)

A.2.3. Collection of Uncertainty Types from Related Approaches

Table A.3 illustrates a collection of uncertainties based on the work of Perez-Palacin and Mirandola [52]. This work summarizes uncertainties elaborated by [21, 26, 19] and categorizes them according to their proposed taxonomy. The taxonomy consists of the following categories: *Location* (context, structural, input parameters), *Level* (1st order - 4th order) and *Nature* (epistemic, aleatory). A detailed explanation of those categories and options is already provided in Section 5.1.2. We omit several uncertainties such as *"Is the* *monitoring cycle subjected to model drift?"* or *"Does noise exist in the sensor values?"* as they are specific for SAS or hardware.

Uncertainty Type	ID	Location	Nature	Level
Does the model make simplifying assumptions?	PU1	Structural / Context	Epistemic	1 - 4
What is the behaviour of humans?	PU2	Context	Epistemic / Aleatory	1 - 4
How is the interaction between systems?	PU3	Context / Structural	Epistemic	1 - 4
What is the execution context?	PU4	Context / Structural / Input Parameter	Epistemic	1 - 4
How do machine learning algorithms behave?	PU5	Structural / Input Parameter	Epistemic / Aleatory	1 - 4
How is the system evolving?	PU6	Structural / Input Parameter	Epistemic / Aleatory	1 - 4

Table A.3.: Collection and Categorization of Uncertainties according to Perez-Palacin and Mirandola [52]

A.3. Evaluation Results

Table A.4 illustrates the categorization of the uncertainty types PU1 - PU1 (see Table A.3) with our approach which is used to answer evaluation question **Q3.2**.

Uncertainty Type	ID	Location	Type	Nature	Arch. Element Type	Manage-ability	Impact on Conf.	Severity of the Impact	Resolv. by ADD	Resol. Time	Root Cause
Does the model make simplifying assumptions?	PU1	—	Scenario Uncer-tainty	Epistemic	—	Fully Re-ducible	Indirect	Low	No	Design Time	Missing Informa-tion
What is the behaviour of humans?	PU2	Input	Scenario Uncer-tainty	Aleatory	Usage Behaviour	Partial Re-ducible	Indirect	Low	No	Runtime	Assumption
How is the interaction between systems?	PU3	System Behaviour	Scenario Uncer-tainty	Epistemic	Commu-nication	Fully Re-ducible	Indirect	Low	Yes	Design Time	Missing Informa-tion
What is the execution context?	PU4	System Environ-ment	Scenario Uncer-tainty	Epistemic	Hardware Resource	Partial Re-ducible	Direct	High	Yes	Deploy-ment Time	Assumption
How do machine learning algorithms behave?	PU5	System Behaviour	Scenario Uncer-tainty	Epistemic	Component	Partial Re-ducible	Indirect	Low	No	Runtime	Assumption
How is the system evolving?	PU6	—	Scenario Uncer-tainty	Epistemic	—	Partial Re-ducible	Indirect	Low	No	Runtime	Assumption

Table A.4.: Classification of existing Uncertainty Types with our Approach

A.4. Acronyms

ADD Architectural Design Decision

PCM Palladio Component Model

ADL Architectural Description Language

GDPR European General Data Protection Regulation

EMF Eclipse Modelling Framework

MOF Meta-Object Facility

OMG Object Management Group

CBSE Component Based Software Engineering

CBSA Component Based Software Architecture

EMOF Essential MOF

API Application Programming Interface

XML Extensible Markup Language

XMI XML Metadata Interchange

SLR Systematic Literature Research

RBAC Role-based Access Control

ABAC Attribute-based Access Control

DSL Domain-specific Language

UIA Uncertainty Impact Analysis

SDQ Software Design and Quality

RUP Rational Unified Process

CVSS Common Vulnerability Scoring System

IoT Internet of Things

DDSA Data-Drive Software Architecture

DFD Data Flow Diagram

SAS Self-adaptive System

SOA Service-oriented Architecture

KAMP	Karlsruhe Architectural Maintainability Prediction
IS	Information Systems
BP	Business Processes
GQM	Goal-Question-Metric
CWA	Corona Warn App
CTS	Contact Tracing Systems