Chapter 6
Continuous Design Decision Support

Anja Kleebaum, Marco Konersmann, Michael Langhammer, Barbara Paech, Michael Goedicke, and Ralf Reussner

In this chapter, we elaborate on how design decisions are made, documented, and exploited during software evolution. We emphasise the importance of design decisions, in particular in the context of continuous software engineering. We detail the challenge of the *intrusiveness* of rational decision-making, documentation, and exploitation of design decisions and the challenge of ensuring *consistency* between design decisions and software artefacts.

The main contributions of this chapter are three approaches to a continuous design decision support: First, we present an approach that supports developers in design decision-making using a catalogue of design patterns. Second, we present an approach to support the awareness for documented design decisions by integrating the decision documentation with the underlying source code. Third, we present how short-cycled practices in continuous software engineering can be used to support the documentation and exploitation of design decisions.

A. Kleebaum · B. Paech (✉)
Universität Heidelberg, Mathematikon - Institut für Informatik, Heidelberg, Germany
e-mail: anja.kleebaum@informatik.uni-heidelberg.de; paech@informatik.uni-heidelberg.de

M. Konersmann
Institute for Software Technology, Research Group Software Engineering, Universität Koblenz-Landau, Koblenz, Germany
e-mail: konersmann@uni-koblenz.de

M. Goedicke
paluno – The Ruhr Institute for Software Technology, Specification of Software Systems, Universität Duisburg-Essen, Essen, Germany
e-mail: michael.goedicke@s3.uni-due.de

M. Langhammer · R. Reussner
Institute for Program Structures and Data Organisation, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: michael.langhammer@alumni.kit.edu; reussner@kit.edu

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_6

All three approaches contribute to the guiding themes *knowledge carrying software* and *methods and processes for evolution* of the priority program.

6.1 Introduction

Continuous Software Engineering (CSE) is a software engineering process in which developers continuously change the software while keeping it in a releasable state [KB17]. CSE means to develop, release, and learn from software in very short rapid cycles [Bos14]. It incorporates agile practices and involves activities such as continuous integration, delivery, and deployment [SAZ17, Joh+18b]. The emergence of CSE is driven by a growing need for flexibility and rapid adaption in the current software environment [FS17].

Software developers and architects continuously make design decisions while they develop and evolve software. They make decisions on the requirements to be addressed, the design artefacts (e.g. architectural components, packages, interfaces, classes, and methods) to be created or the design patterns to be applied. For example, it is a design decision to apply an adapter design pattern instead of changing an existing interface when adding new features to a software. The knowledge of the developers on the design decisions they make is called *decision knowledge*. In particular, decision knowledge comprises the knowledge about the problems, the decisions they address, solution approaches, their context, and rationale in terms of arguments, criteria, and the assessment of solution alternatives.

Decision knowledge should be communicated within a development team so that every developer knows and considers existing decisions [Bru+14]. When developers evolve software, it is important for them to reflect and build on former decisions. Otherwise, they might make inconsistent decisions and are likely to contribute to the erosion of the software architecture or introduce other quality problems [Cle+13, Cap+16]. Reflecting on former decisions is particularly important for long-living software systems where many decisions build on one another.

The documentation of decision knowledge is important for several reasons: First, many different developers might be involved at different times. Thus, they cannot communicate directly and rely on documented decision knowledge when they reflect on former decisions. That means that the documentation of decision knowledge is important to prevent knowledge vaporisation [Cap+16]. Decision knowledge vaporises quickly; that is, if developers do not capture decision knowledge immediately, it will never be captured and thus will not be available later [JB05]. Tacit decision knowledge (cf. Chap. 5) enlarges the risk of misunderstandings and errors during evolution or maintenance. Second, the documentation of decision knowledge makes the criteria for the design decisions explicit that might otherwise be overlooked. This promotes a more rational decision-making process. Third, documented decision knowledge is valuable to support future changes. It supports change impact analysis, requirement validation, and long-term maintenance and keeps developers informed about underlying architectural decisions [Cle+13].

While there is clearly a need for decision knowledge documentation, in practice this is often not performed [APM16]. In practice, decisions are mostly made and documented in a naturalistic way [ZCM07, Hes+16]. This means that only a part of the decision knowledge—often only the decision—is documented, which impairs the rational decision-making. Humans tend to overlook what is missing and are subject to cognitive biases [Raz+16]. Furthermore, if the arguments for the decision are not documented, other developers might not understand the decision or might not be convinced.

Recently, various techniques emerged that try to reconstruct decision knowledge by mining written text from informal sources such as chat messages, which is referred to as *extractive summarisation* [NHJ16]. These techniques are promising in identifying decision knowledge; however, the knowledge may be incomplete, outdated, or hard to access later. In other cases, the knowledge is not captured at all but only resides in the developers' heads as tacit knowledge. Researcher attempt to infer tacit knowledge by *abstractive summarisation* of software artefacts such as source code changes [Cor+14]. However, Robillard et al. confirm that it is unlikely to infer complex information such as rationale by mechanical extraction of facts from software artefacts [Rob+17]. Therefore, summarisation techniques only partially help to reconstruct decision knowledge in case they are applied retrospectively. Decision knowledge needs to be explicitly documented in order to preserve it. It is important to note that easy exploitation of the decision knowledge motivates developers to document it, as the developers themselves can profit from the documentation [BB08].

CSE provides many practices for a continuous change [KB17]. These can be used for a continuous design decision documentation. Our long-term vision is an *on-demand decision documentation* as part of the *on-demand developer documentation* suggested by Robillard et al. [Rob+17]. We envision that developers continuously capture and reflect decision knowledge during CSE. Our goal is to support developers in this continuous capture and reflection, in particular by performing rational decision-making. The following three developer tasks should be lightweight; that is, they should require as little effort as possible: *rational decision-making*, *documentation of decision knowledge*, and its *exploitation*.

6.1.1 Challenges for a Continuous Design Decision Support

Tool support to manage decision knowledge can be characterised by its intrusiveness in the software development process [Dut+06]. Tools that fit into the development context are less *intrusive* and will more likely be used [KCD09]. Such tools do not require additional effort (e.g. for installing or starting a separate tool) and are thus also lightweight. For example, a developer can capture the design decision for applying an adapter design pattern within a commit message instead of in a separate tool. Rational decision-making, documentation of decision knowledge, and its exploitation should be non-intrusive in the context of the CSE process. It is a challenge to minimise the intrusiveness of a continuous design decision support.

Challenge regarding intrusiveness: how to integrate rational design decision-making, documentation, and exploitation in software engineering practices

To exploit decision knowledge, it is important that the design decisions are *consistent* (a) with former design decisions and (b) with the artefacts, for example, with the requirements, architectural software design, and code. Consistency means that design decisions are documented, as well as linked to and realised, in the artefacts they relate to. For example, the design decision to apply the adapter design pattern should be linked to the code that implements the pattern. Then developers can reflect on this decision when they change the code. Developers need to reflect former decision knowledge during decision-making, so that the design decisions are consistent with each other. There are two types of former decision knowledge: First, general decision knowledge is documented in *external knowledge bases* (e.g. about design patterns). Second, new design decisions build on former decision knowledge *specific to the software development project*. Especially in long-living software systems, much decision knowledge accumulates. Documented decision knowledge might be invalidated during software evolution and needs to be updated. Not only decisions need to be consistent with each other. The decision knowledge also needs to be consistent with the artefacts. Moreover, the design artefacts, for example architectural software design and code, also need to be consistent with each other to ensure that the decisions are actually implemented. It is a challenge to document and maintain decision knowledge consistent with the other artefacts and with former decision knowledge.

Challenge regarding inconsistency: how to ensure consistency between decision knowledge and artefacts

6.1.2 Solution Approaches for Design Decision Challenges

In this chapter, we present approaches that address both challenges. The approaches try to find a balance between intrusiveness and consistency support. A more powerful support typically requires separate tools, which are more intrusive. Also, the approaches focus on different kinds of decision knowledge. The first approach promotes rational decision-making by providing software designers with a catalogue of questions that support them in choosing a design pattern. Thus, this approach focuses on *consistency with external decision knowledge, which is presented in a separate tool*. The second approach focuses on the *consistency among decisions within a project, architecture, and code*. It ensures that design decisions are documented and related to design and implementation artefacts. Thus, this approach improves the consistency relation between these artefacts. It incorporates the decision knowledge captured by the design pattern approach. The third approach provides *non-intrusive integration* of the documentation and exploitation support during CSE and *lightweight traceability for consistency*. During CSE, developers usually manage code and other development knowledge in a Version Control System (VCS) and issues in an Issue Tracking System (ITS) [Sai+17]. The third approach

integrates the documentation and exploitation of decision knowledge into practices relating to the VCS and ITS. Thus, it does not require a separate tool. It uses *extractive and abstractive summaries* to support the transition from naturalistic to rational decision knowledge documentation. Furthermore, it identifies relevant decision knowledge based on traceability links to support *consistent decision-making*. These approaches showcase different ways to support decision-making, documentation, and exploitation. The choice of one of them depends on the context.

6.1.3 Structure of This Chapter

This chapter is structured as follows: Sect. 6.2 sketches the Decision Documentation Model (DDM) as a foundation of this chapter. The DDM allows developers and architects to document decision knowledge incrementally and collaboratively. Section 6.3 presents the approach that supports the decision-making regarding design patterns using a pattern catalogue and documenting such decision knowledge. Section 6.4 presents the approach to support the documentation and consistency by integrating design decision models with program code. In Sect. 6.5, the approach focusing on short-cycled CSE practices is introduced. Section 6.6 presents related work. Section 6.7 discusses and concludes this chapter and provides an outlook. Section 6.8 provides references for further reading.

6.2 Foundations

We represent decision knowledge based on the DDM by Hesse and Paech [HP13]. According to the DDM, decision knowledge is documented as *decision components*, which can be nested and refer to other knowledge. Figure 6.1 shows the key decision components of the DDM (depicted with yellow background), as well as additional decision components used in the pattern catalogue in Sect. 6.3 (depicted with white background). In Fig. 6.1, *decision component* is an abstract class that can only be instantiated through its subclasses. Related knowledge elements can be decision knowledge or software artefacts such as requirements, architectural design, code, and test cases. Decision components are the decision *problem* to be solved (issues or goals), *solution* (alternatives or claims), *context* information (assumptions, constraints, or implications), and *rationale* (arguments or assessments). The DDM subsumes decision elements used in other approaches [PDH14] but does not prescribe any components for decision documentation. Therefore, it supports incremental documentation of decisions and in particular both naturalistic and rational decision-making. Any part of the decision knowledge can be captured as soon as it is available. In addition, any number of stakeholders such as developers, architects, and requirement engineers can collaborate while documenting decisions. Each stakeholder contributes that part of the decision knowledge they know best. The requirement engineer can, for example, add constraints, which have to be reflected for a particular solution.

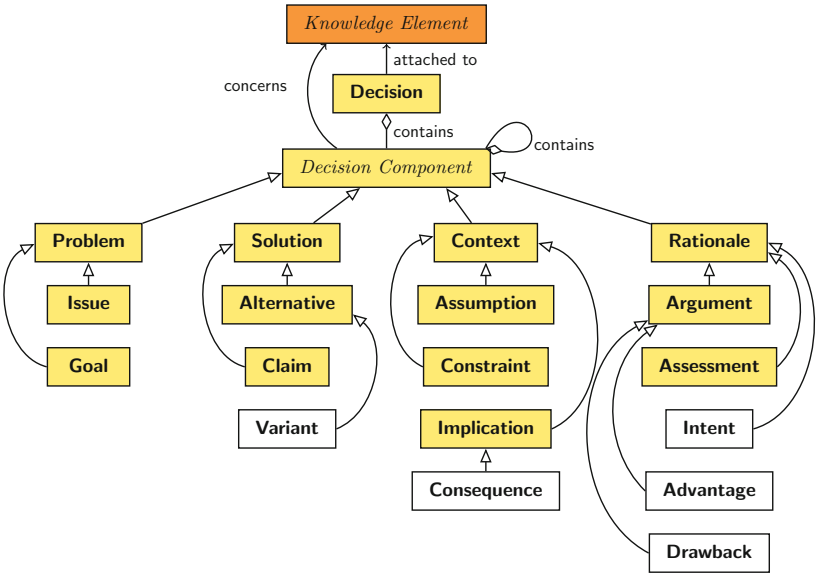

Fig. 6.1 Decision documentation model (DDM) adapted from [HP13]

The DDM has been applied in an empirical study on Firefox issue reports [Hes+16]. This showed that the DDM can adequately reflect the decision knowledge captured in issue trackers. The dominance of naturalistic decision-making in this study confirms the need for an incremental and collaborative decision documentation. In addition, the DDM has been applied in a case study on design session transcripts [HP16]. This confirmed that the DDM also adequately reflects decision-making in a team. In particular, the usage of the DDM made complex decision knowledge structures in the design sessions explicit.

6.3 Using a Design Pattern Catalogue to Make Design Decisions

In this section, we explain the *Architectural Modelling with Design Decision Documentation (AM3D)* [Dur14] approach that supports software architects and software developers in the process of decision-making. For this purpose, it uses a pre-defined pattern catalogue that contains patterns and pattern-specific questions as main artefacts. Software architects using the approach need to answer a set of questions to get the correct pattern that solves their current problem. Compared to the classical software architecture process, the advantage of this approach is that the design decisions become more rational and less naturalistic. Furthermore, the design decisions are documented and made explicit and thus are easier to understand by

other software architects and software developers. The AM3D approach supports rational decision-making consistent with former external decision knowledge. In the following, we describe the approach in detail and apply it to the Common Component Modeling Example (CoCoME) case study. The details of the pattern catalogue and the decision-making process are presented in the dissertation of Durdik [Dur14].

This section is structured as follows: Sect. 6.3.1 explains the motivation for using a pattern catalogue. In Sect. 6.3.2, we explain how a pattern catalogue can be used for decision-making. In Sect. 6.3.3, we show how the presented approach can be applied to our current example.

6.3.1 Motivation for Using a Pattern Catalogue

In the domain of software engineering, patterns are widely used to solve common problems. In the last decades, various pattern catalogues have been introduced, for example by Gamma et al. [Gam+95] and Buschmann et al. [Bus+96]. If software architects and software developers need to solve a specific problem, they can often use one of the already existing patterns. Choosing the correct pattern for a given problem, however, is not an easy task as there are many patterns solving similar problems. Another problem that arises using patterns is that they are often used wrongly. Hence, choosing the correct pattern and using it correctly is a difficult and error-prone task.

6.3.2 Decision-Making Process Using a Pattern Catalogue

In this section, we explain the decision-making process, which is used to choose the correct pattern. This includes the presentation of the pattern catalogue and the activities to make the design decisions explicit.

Most sources of patterns, such as [Gam+95], contain patterns in a free-text form. The advantage of these sources is that one can learn about patterns, their benefits, their usage, and others. However, their disadvantage is that the information is not structured, and it takes a lot of time to gain knowledge about patterns that can be used to solve a specific problem. Often, it is also unclear which pattern form shall be chosen to solve a given problem.

The design pattern catalogue of the AM3D approach aims to overcome these disadvantages. Its main purposes and goals that are relevant for the decision-making process are (1) to present structured information about patterns, (2) to allow for semi-automated documentation of the pattern usage, and (3) to support goal-oriented requirement engineering.

The three main information parts stored in the pattern catalogue are (1) general information about the pattern, (2) questions annotated to the pattern, and (3)

Table 6.1 Information about a pattern stored in the pattern catalogue [Dur14]

Category	Detailed attribute	Short description
General information	Name	A name for the pattern
	Type	A type for the pattern, for example object-oriented pattern or security pattern
	Category	The category of the pattern, usually described by pattern authors; for instance, *behavioural* patterns are a category by Gamma [Gam+95]
	Information source	The original source of the pattern
	ID	A unique identifier for the pattern
	Goal	A high-level description of the pattern's goal, respectively the problem that can be solved using the pattern
	Description	A brief description of the pattern, which is intended for users in order to understand the concept of the pattern
	Advantages	Advantages of the pattern, which come with the usage of this pattern
	Drawbacks	Highlighting problems/drawbacks of the pattern
	Keywords	Keywords to characterise the pattern
	Quality attributes	The pattern's impact on quality dimensions of the software system, for example performance increased/decreased
	Relationships	Relations to other patterns, divided into three dimensions: (1) recommended co-patterns, (2) similar patterns, and (3) excluded patterns
	Variants	Variants of the pattern
Question annotations	Goal	Questions on the goal of the user, that is whether the user likes to solve a problem in a specific way
	Intent	Questions on the intent of the user, that is whether the user intents to have a specific behaviour in a software system
	Consequence	Questions on consequences, that is whether some consequences are acceptable if a specific pattern is used

the structure of the implementation as a Unified Modeling Language (UML)-like diagram. Table 6.1 shows the details for the general information and the questions. The questions are divided into the following four categories: (1) questions regarding the goal of the pattern, (2) questions regarding the advantages of the pattern, (3) questions regarding the drawbacks of the pattern, and (4) questions regarding variants of the current pattern.

Table 6.2 Questions for the façade pattern [Dur14]

Type	Question
Goal	Would you like to provide a unified interface to a set of interfaces in a subsystem?
Intent	Would you like to minimise the communication and dependencies between subsystems?
	An additional functionality wrapped into the unified interface is not your intent? (otherwise → proxy)
	Is a stateless unified interface your intent? (otherwise → proxy)
	Is it desired that subsystem classes know nothing about the façade object(s)? (otherwise → mediator)
	A new interface for an object is not your intent? (otherwise → adapter)
Consequence	Is a potential performance bottleneck not an issue?

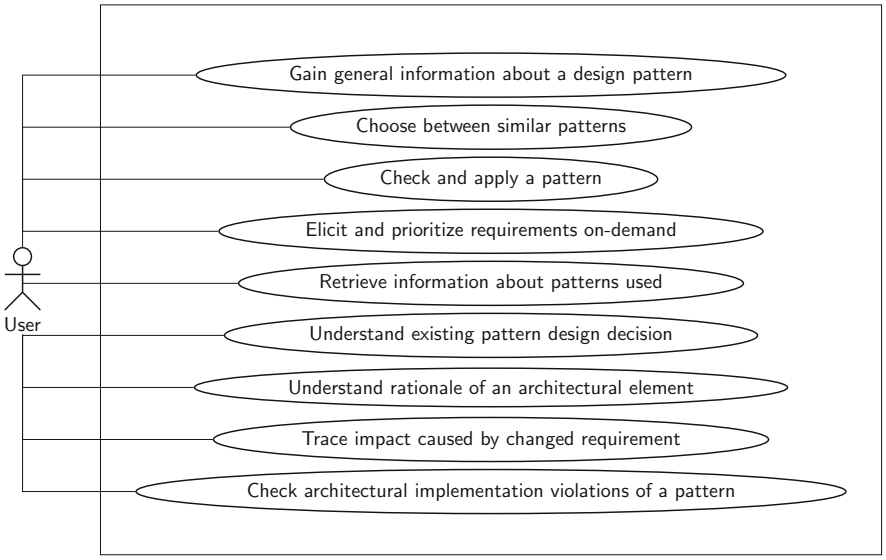

Fig. 6.2 Use cases for the pattern catalogue [Dur14]

As an example, questions for the façade pattern are shown in Table 6.2. The structured information about the patterns allows to ask structured questions to the users and to present appropriate patterns for the problem that the users want to solve.

The pattern catalogue can be used in multiple use cases during the development process. The use cases are shown in Fig. 6.2. In this chapter, we focus on the main use case *check and apply a pattern*, which involves making the design decision for a specific pattern and documenting this decision. The remaining use cases are explained in [Dur14]. Figure 6.3 shows the activity diagram of the use case. The first step is to analyse the problem based on the given requirements. The second step

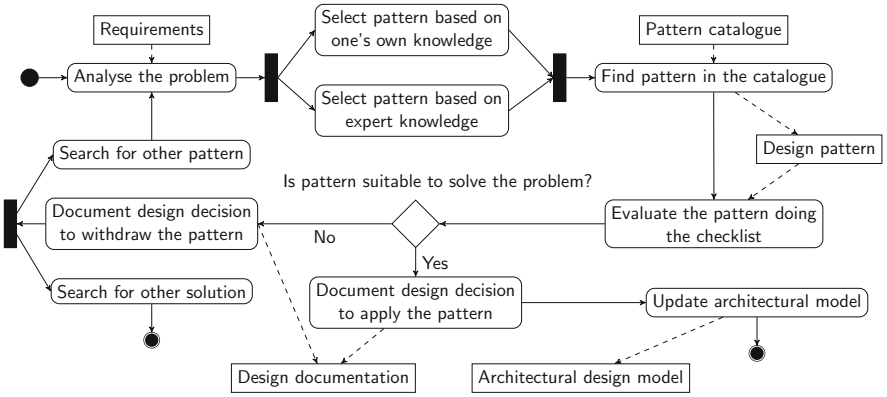

Fig. 6.3 Activity diagram of checking and applying a pattern [Dur14]

is to pre-choose a pattern based on one's own knowledge or based on an expert's knowledge. Next, the pattern catalogue is used to find and evaluate the pattern. The evaluation is done by using the checklist, which is attached to the pattern, that is the questions for the pattern are evaluated again to clarify whether the chosen pattern is a correct one for the given problem. If the pattern is suitable to solve the problem, the design decision has been made. The next steps are to document the decision and to update the architectural model with the newly chosen pattern. If the pattern is not suitable, the decision that the pattern has not been chosen and the reason why are documented. Then the iteration starts from the beginning by re-analysing the problem and looking for a different pattern. If, however, no pattern can be found that solves the problem, another solution needs to be found, for example clarifying the requirements.

In summary, the AM3D process guides users through the process of decision-making. It also stores the answers and the decision in a model, that is the decision knowledge is made explicit and is documented. The main advantages of using the catalogue and the structured process are as follows: (1) The rationale and other decision knowledge of the design decisions to apply a specific pattern is documented. (2) Through systematic pattern evaluation with the help of question annotations, software developers and software architects are supported in applying design patterns and design pattern variants correctly.

The AM3D approach has been evaluated in a controlled experiment with 20 students [Dur14]. During the evaluation, the technical questions concerning the patterns have been evaluated as well. For the evaluation, the approach was compared to a standard pattern catalogue. During the evaluation, the students had to face two scenarios: In the first scenario, a new design decision had to be made, whereas in the second scenario an existing decision had to be re-evaluated. The students who used the AM3D approach had better results in both scenarios. The results for the first scenario are statistically significant, while the results for the second scenario are not.

6.3.3 Application to the Case Study

In this section, we show an application of the AM3D approach to a CoCoME evolution scenario. In this scenario, the CoCoME sales system is extended by new payment possibilities. Up to now, customers could only pay via debit card. Payments are initiated by the CashDesk component. Currently, this component communicates with an external bank (`TrivialBankServer` component) via the `IBank` interface. The `IBank` interface defines the methods `validateCard` and `debitCard`. Figure 6.4 shows an excerpt of the CoCoME architecture as a Palladio Component Model (PCM) repository diagram [BKR09].

Requirements for modern payment possibilities such as PayPal and Bitcoins arise. The new payment possibilities are to be implemented, while the existing payment possibility using a bank server will still be supported. We focus on the latter case and assume that the decision process is executed using the AM3D approach.

In this scenario, the generic `IPayment` interface is introduced that defines the `authenticate` and `pay` methods. For using the existing component `TrivialBankServer` together with the new `IPayment` interface, the adapter pattern and the façade pattern are taken into account by software architects. Hence, they need to evaluate the two patterns using the design pattern catalogue. First, the façade pattern is evaluated. As we can see in Table 6.2, however, the first question for the façade pattern is answered with *no* because no unified interface to a set of interfaces needs to be provided. Thus, the architects know that the façade pattern is not the correct pattern in this case. As a next pattern, the adapter pattern is evaluated. Therefore, the questions in Table 6.3 are used. Even though the questions are technical and quite detailed, the evaluation showed that they can be answered correctly by intended users of the AM3D approach. As all the questions for the adapter pattern can be answered with yes, the architects know that they can use the adapter pattern for the implementation. The decision knowledge is illustrated

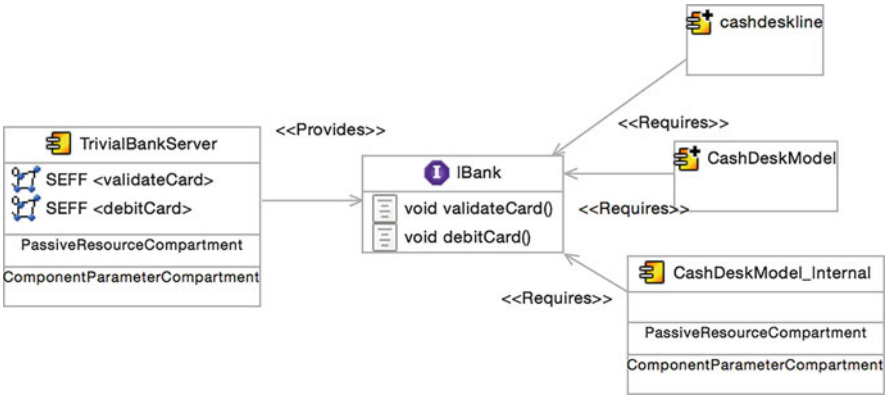

Fig. 6.4 An excerpt of the CoCoME architecture in PCM before the evolution scenario

Table 6.3 Questions for the adapter pattern

Type	Question
Goal	Would you like to convert an interface of a class (or an object) into another interface that clients expect?
Intent	Would you like to make interfaces of incompatible classes compatible? Would you like to change the interface of an existing object (a new interface design for an object)? (otherwise → proxy or decorator)
Consequence	Are you aware of the size of the code you have to write and maintain to adapt the class?

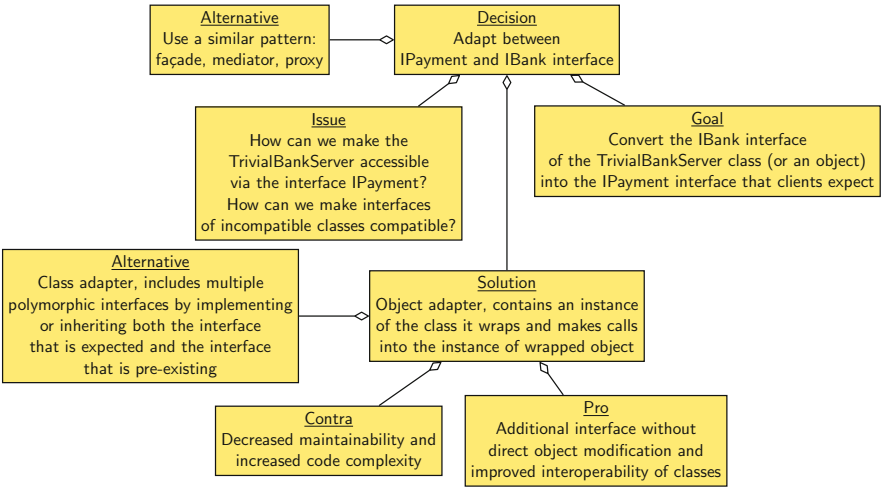

Fig. 6.5 Decision knowledge for the adapter pattern according to the DDM

in Fig. 6.5 (according to the DDM, cf. Sect. 6.2). From the pattern catalogue, they also get an example for the adapter pattern and adapt it to the CoCoME components and interfaces. They adapt the existing `IBank` interface using the `TrivialBankServerAdapter` component in order to make the component `TrivialBankServer` compatible with the new `IPayment` interface. Figure 6.6 shows the resulting architectural structure.

6.4 Integrating Design Decision Models with Program Code

During the evolution of software systems, documented design decisions are often not updated. The documented design decisions, design artefacts, and the program code are then no longer consistent. Even worse, the documented design decisions may be misleading, when they document a revised decision, and are neither updated

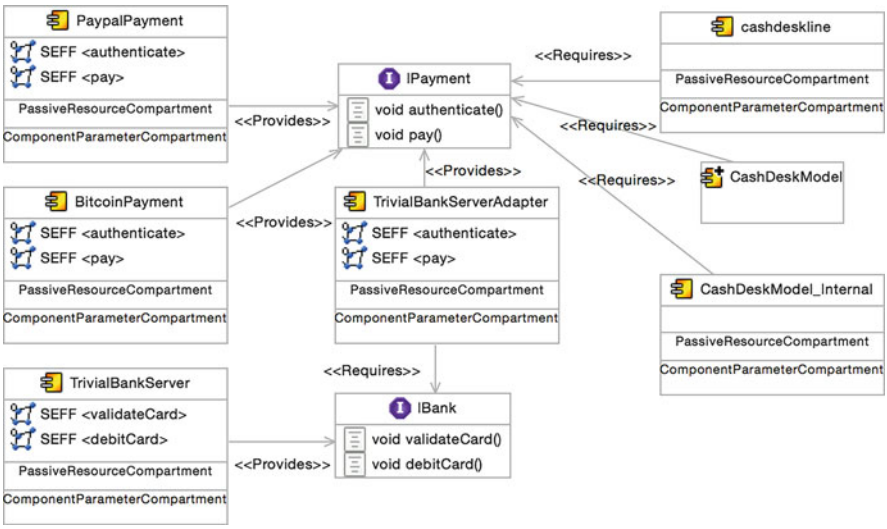

Fig. 6.6 An excerpt of the CoCoME architecture in PCM after the evolution scenario

nor marked as outdated. In this section, we describe an approach to integrate decision knowledge and software architecture information with program code. The tool *Codeling* [Kon18, Kon16] implements an approach for integrating model information with program code. Codeling is used to create bidirectional translations between program code and abstract models of that code. By documenting design decisions within the program code and relating them to architectural design artefacts, the documentation of design decisions is visible during the development and evolution of a system. The goal is to improve the documentation of decision knowledge, the consistency between software models and code, the evolvability, and the understandability of the software.

In Codeling, we create mappings between the concepts of architecture implementation languages and abstract software models. As an example, we define mappings between components defined in the UML and components defined with the Java programming language extended with a component framework. These are the artefacts to which decision knowledge is attached according to the DDM (Fig. 6.1). Therefore, Codeling can document design decisions that were made using the approach described in Sect. 6.3. We use these mappings to automatically propagate changes in the model or the program code to the other representation. As the mapping between these artefacts and program code is established with Codeling, it is possible to attach decision knowledge to these program code elements. When all modelled information has a representation in the program code, a separate model document is not necessary any more. It can be extracted from the program code using the defined mappings.

In Sect. 6.4.1, we briefly describe Codeling and its application to software architectures. Here, we address the challenge to ensure consistency between architectural

software design and code. Section 6.4.2 extends the approach with a notation for decision knowledge. Here, we extend the consistency relation between architectural design and code with design decision knowledge. We also address the challenge to integrate the documentation of decision knowledge into software engineering practices, especially into coding and modelling. Section 6.4.3 shows the application of Codeling on the current example of Sect. 6.3.3.

6.4.1 Integrating Architecture Models with Code

Specifications of software architectures can be seen as abstract views on relevant design decisions. The goals of architecture specifications are diverse, generally centering on the design, communication, or analysis of the subject of specification. A set of abstract concerns commonly agreed upon seems to exist for defining software architectures, as manifested by the standard ISO/IEC 42010 [ISO11b]. These include the general structure of a system, usually expressed in components, interfaces, and their interconnection. They are often accompanied by abstract behaviour descriptions or quality aspects. During software development, the architecture is realised in the software artefacts, including the program code, configuration, and the use of existing platforms. The goal of the implementation is an executable system. The implementation of software architecture is driven by industry standards and platforms that define standard elements such as components and interfaces. Languages for architecture specification and for architecture implementation have common concerns (see e.g. [MBG10]), typically at least the definition of components, interfaces, and their interconnections. However, they have different foci and include different types of architectural designs and different details added to the architectural description.

Codeling creates a systematic mapping between architecture specification model elements, relations, and attributes and their implementation based on standardised or project-specific architecture implementation languages. These mappings specifically define places where arbitrary other code can be added. This kind of mapping allows to extract architecture specification models from program code and to propagate changes in these models back to the code.

Codeling comprises three parts. Figure 6.7 sketches an overview of these parts and their relations. The figure describes artefacts of the approach with rounded boxes and translations between these artefacts with arrows. The parts are used to bidirectionally translate between program code and a specification model expressed in an architecture specification language. The parts are underlined in Fig. 6.7.

Intermediate Architecture Language The Intermediate Architecture Language (IAL) mediates between architecture implementation models and architecture specification models. The IAL is implemented with an Ecore-based [Ste+09] meta model. It has a small core with the common elements of architecture languages. The core is extended with profiles [Lan+12] to represent, for example different kinds of

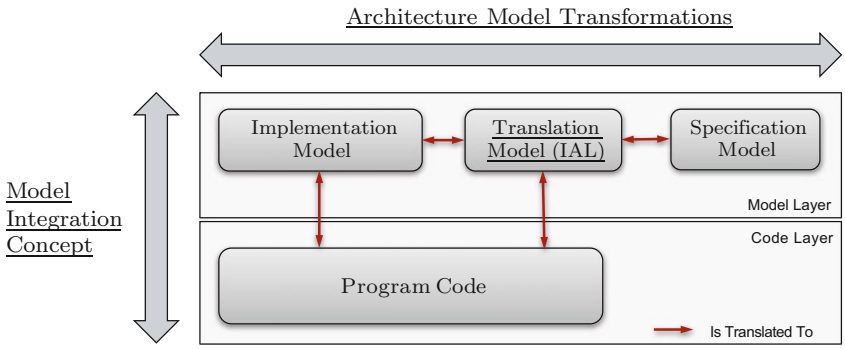

Fig. 6.7 The parts of *Codeling* for integrating architecture model information with program code

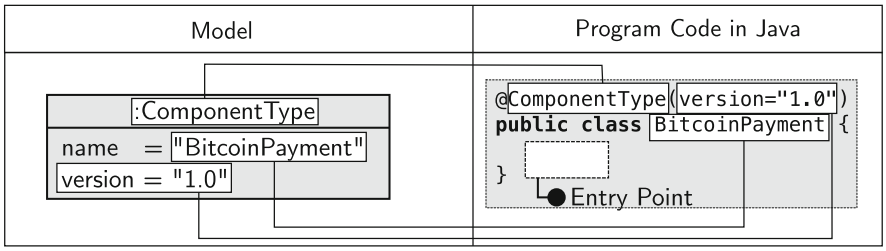

Fig. 6.8 An exemplary bidirectional model-to-code mapping from the MIC

interfaces, component hierarchies, or quality attributes. Models that are expressed with the IAL are called *translation models*.

Model Integration Concept The Model Integration Concept (MIC) describes bidirectional formal mappings between program code structures and an implementation model. The implementation model is a model representation of the architectural aspects of the code. For example, a Java type declaration with a specific annotation might represent a component type, and annotation parameters represent attributes of this component. Figure 6.8 gives an example of two combined mappings. A modelled component type is represented as a Java type declaration with the annotation *ComponentType*. The type's name is mapped to the component type's name. The modelled attribute *version* and the value are mapped to an annotation parameter assignment. Bidirectional model-to-code mappings in the MIC may include *entry points*. Within entry points, arbitrary other program code can be inserted.

In Codeling, the program code also contains information that is not part of an architecture implementation language but is only subject to a specification language. For example, many architecture implementation languages do not describe hierarchical architectures. The hierarchy information is added to the program code, for example using package structures. This information is forwarded directly to the

translation model using the MIC. The MIC implements bidirectional transformations. Therefore, changes in the model are propagated to code changes.

For Codeling, we have developed a set of translation templates between models and code. They generically describe how modelled objects, attributes, and references can be represented in program code, so that bidirectional translations can be implemented. Codeling consists of a tool to generate automated translations, by relating these templates to specific meta-model elements [Kon18]. The tool then generates translation classes in Java, which are executable within Codeling.

Architecture Model Transformations Bidirectional architecture model transformations translate between implementation models, translation models, and specification models. Architecture implementation models are translated into specification models. Changes to a specification model are propagated to the corresponding implementation model.

6.4.2 Design Decisions, Rationale, and Patterns in the IAL

Section 6.3 presents the specification language *AM3D* for design decisions and rationale applied to PCM diagrams. To integrate design decisions and rationale with Codeling, (a) the IAL must be able to handle this information. This makes design decision information available to Codeling. Then (b) transformations must be created between the AM3D and the IAL to make the information available to the existing tool environment of AM3D. Finally, (c) mappings must be created between the IAL and the program code.

The IAL can handle decision knowledge (a) via corresponding profiles. These are language extensions for expressing design decisions with rationale. Decisions can either be decisions for the existence or design of specific components or the decision for implementing a specific architectural pattern. Decisions are accompanied by rationale. The rationale can be expressed with informal text or by answering questions of a catalogue, as it is described in Sect. 6.3. We also added meta-model elements for describing instances of architecture patterns and the roles of components and connectors within them, as described in Sect. 6.3. We implemented transformations between the IAL and AM3D (b) with triple-graph grammars[1] (TGG) [Sch94b]. TGGs describe a bidirectional relationship between language elements. For example, they can be used to define that a decision element in the IAL corresponds to a decision element in AM3D. Automated synchronisation rules can be derived from these relationships.

Mappings between decision knowledge expressed in the IAL and Java program code (c) have to be designed in the context of the MIC. A simplified example for expressing the modelled decision knowledge in program code with the MIC is given

[1]The IAL meta model with the design decisions and pattern profiles and transformations between the IAL and AM3D are available under https://codeling.de.

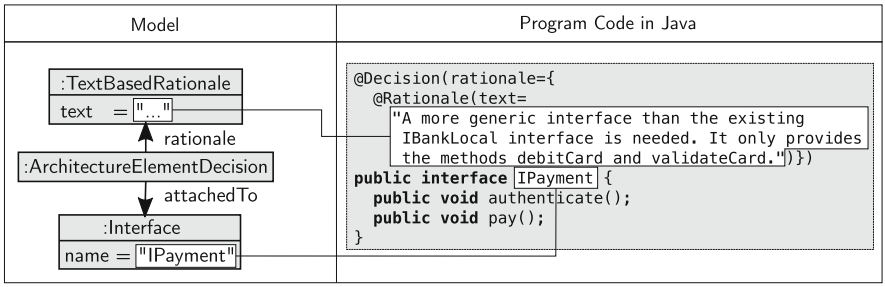

Fig. 6.9 A bidirectional example model-to-code mapping of a decision for an architectural element with text-based rationale

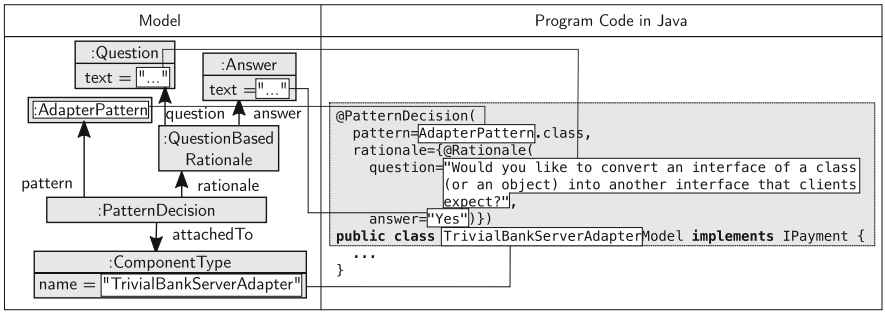

Fig. 6.10 A bidirectional example model-to-code mapping of a pattern decision with question-based rationale

in Fig. 6.9. The figure shows an interface. The modelled interface is represented with an interface definition in Java. The name of the Java interface is mapped to the value of the attribute *name*. A decision with a text-based rationale is attached to the interface. The attached decision is defined as an annotation attached to the Java interface. The rationale is an annotation parameter. The model instantiation of the rationale is a *TextBasedRationale* with a text that contains the actual, informal rationale. The code equivalent is an instantiation of the annotation `Rationale` with the parameter `text` with the respective content.

Figure 6.10 shows an exemplary mapping between a question-based decision for an architectural pattern and a respective code representation. The model shows a pattern decision attached to a component type. The pattern decision relates to an instance of the adapter pattern. The rationale is based on the answering of questions in a catalogue. The component type is represented as a Java type declaration with the name of the component type and the suffix *Model*. The pattern decision is represented as an annotation attached to that type. It has two annotation parameters: The pattern references the type `AdapterPattern`. This type is defined in a library. It represents the corresponding pattern. This mechanism allows for type-safe references because the referenceable types need to implement a specific interface.

In this example, the rationale is represented by the parameters `question` and `answer` of the annotation `Rationale`.

In four case studies, Codeling has shown its applicability and usefulness for improving the consistency between architecture models and code, and the understandability and evolvability of software architectures [Kon18, Chapter 10]. In these case studies, Codeling has been used to extract software architecture models from code, propagate changes in architecture models to the code, and to migrate between architecture languages. Besides architectural structure, the integrated information in these case studies also include performance annotations on operations. They indicate the expected performance of an operation for simulation purposes. This is comparable to design decisions as they are presented in this section. In this section, design decisions are also attached to structural elements and have no operational semantics for the software. Therefore, the approach presented here can document design decisions integrated with the program code and improve the understandability and the evolvability of the software architecture, including the design decisions.

6.4.3 Application to the Case Study

In the context of the case study used in this chapter, Codeling is used to create a PCM view upon the CoCoME architecture with AM3D extensions. Figure 6.4 in Sect. 6.3.3 shows an excerpt of the PCM repository as it is extracted with Codeling. The full repository diagram is shown in Fig. 12.4 on page 350. Table 6.4 gives an overview of the mapping between the CoCoME code, the corresponding architecture implementation language, and PCM meta-model elements. The table contains

Table 6.4 Overview of the mapping between PCM meta-model elements, CoCoME meta-model elements, and program code structures

PCM meta-model element	CoCoME meta-model element	Program code structures
Basic component with the name “Model”	“Model” component	Type declaration with the name “Model”
Basic component with the name “Console”	“Console” component	Type declaration with the name “Console”
Basic component with the name “Server”	“Server” component	Type declaration with the name “Server”
Composite component	Component with children	Package declaration with package or type declarations as subcomponents
Operation provided role	Provided interface	Implemented interface
Operation required role	Required interface	Interface instance given to type via constructor

the mappings relevant for adding design decisions and rationale to component and pattern decisions.

First, we developed a meta model for describing the CoCoME architecture. This was necessary because the original CoCoME implementation does not follow any standard for implementing components but uses a custom style for describing architectural elements and their interconnections using plain Java. For example, it defines three different types of components: *model* components, *console* components, and *server* components. Instances of these component types are implemented using Java type declarations with names that end with that specific suffix. The different component types indicate different roles of the corresponding components within the program. Second, we implemented bidirectional model-to-code transformations between the CoCoME program code and the newly created meta model for the CoCoME architecture.

Next, we created mappings between the CoCoME architecture meta model and the IAL using triple-graph grammars. Design decisions and their rationale are information that can be attached to their corresponding code elements. Figures 6.9 and 6.10 show examples of this set of transformations between models and code. In the CoCoME example, a new interface `IPayment` is introduced because the existing `IBank` interface did not provide the necessary operations. This decision is attached to the new Java interface in Fig. 6.9. The informal text of the text-based rationale is added as annotation member value. Another change in the CoCoME example is the introduction of an adapter, following the adapter pattern, to make the TrivialBankServer accessible via the interface `IPayment`. Figure 6.10 shows the integration of a pattern decision with a question-based rationale. The listing shows how the pattern decision is documented with annotations in the Java code. The implementation of the pattern is not shown in this figure, for readability reasons. Documented design decisions have no operational semantics, which means that it is not necessary to evaluate them at run time. A pattern decision references a pattern in an annotation parameter. Here, only the decision is defined. The actual implementation of the pattern is not evaluated with this mapping. However, such mappings can be created with the MIC. For example, such translations would ensure that a component type, which has the role of an adapter in an adapter pattern, implements the respective interface and has a reference to the adaptee. The actual behaviour of the adapter can then be implemented in entry points of the code representation.

The model-to-code translations and model-to-model translations have to be defined by a developer. Codeling contains tools to support the definition of bidirectional model-to-code transformations with templates and a code generator. Once defined, the automated translations can be used with Codeling to create an architecture model of the CoCoME code with decision knowledge in PCM with AM3D extensions. Changes in the model are automatically propagated to the program code.

In summary, Codeling addresses the challenge of the consistency between architectural knowledge and the program code. Besides other information, this architectural knowledge includes architectural structure, design decisions, and

architectural patterns. The approach also addresses the challenge to integrate the documentation of decision knowledge into software engineering practices, especially into coding and modelling. The main advantage is that design decision models are documented at the code level, so that the decisions are available to developers and are included in the VCS.

6.5 Continuous Management of Decision Knowledge

The approach presented in this section integrates the documentation and exploitation of design decisions into the work of the developers, in particular the usage of VCS and ITS. We refer to it as Continuous Management of Design Decisions (ConDec).

We address both challenges in this section. In Sect. 6.5.1, we detail the relevant knowledge elements of the DDM introduced in Sect. 6.2. Section 6.5.2 presents the main ideas on how to use short-cycled CSE practices to trigger developers so they would document and exploit decision knowledge. Section 6.5.3 describes the application to the case study.

6.5.1 Integrating Design Decisions into CSE

The knowledge meta model is shown in Fig. 6.11. Software artefacts contain knowledge that we classify into system and project knowledge [PDH14]. *System knowledge* concerns the software itself (e.g. code, requirements, design, test cases), whereas the knowledge about its development and evolution is summarised under the term *project knowledge*. *Decision knowledge* can relate to both knowledge types.

In CSE, features are more prominent than components [Bos14]. Thus, we focus on features and code as essential system knowledge elements in CSE. Features represent both functional and non-functional requirements. Features can be split into

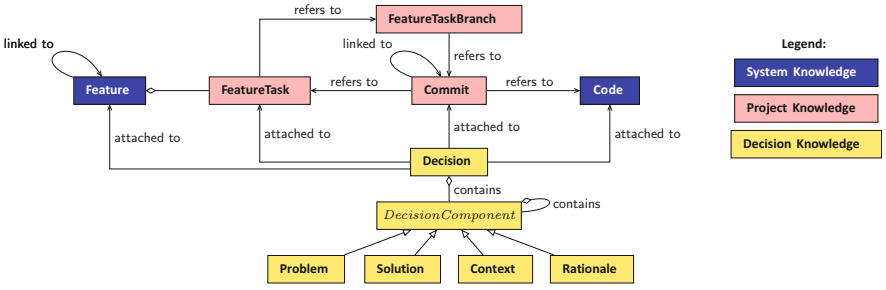

Fig. 6.11 Relationship between features, tasks to implement the feature (feature task), code, commits, and decision knowledge

sub-features or grouped into bigger features. We refer to the tasks that developers fulfil to implement a feature as *feature tasks*. Short-lived branches can be used to encapsulate the actual development work [Kru+14]. We refer to these branches as *feature task branches*. A feature task branch comprises one or more commits that refer to code. When a feature task branch is merged into another branch, a merge commit is created. The difference between merge commits and normal commits is that the merge commit has two parent commits [CS14]. Feature tasks, feature task branches, and commits are types of project knowledge. We use the DDM explained in Sect. 6.2 to represent the decision knowledge.

We assume that tracing between features, feature tasks, commits, and code, as well as decision knowledge, is possible (cf. the relationships in Fig. 6.11). Tracing can be accomplished either using (a) textual annotations such as decision annotations [Hes+15] or task identifiers in the commit messages, (b) distinctly documented trace links (e.g. within a table), and (c) trace retrieval techniques [Cle+13]. A tracing possibility is the prerequisite for developers to consider and ensure the consistency of decision knowledge and artefacts. Tracing enables developers to simultaneously reflect decision knowledge and artefacts. Developers can explore code and decision knowledge that evolved during the implementation of a feature. Likewise, developers can see decision knowledge and features relevant to a certain piece of code.

Evidently, there are other CSE artefacts that can contain relevant knowledge, for example user feedback, pull requests, or chat messages. We consider the artefacts in Fig. 6.11 as the minimal set of CSE knowledge artefacts.

In the following, the implementation of this meta model is introduced: Feature tasks are often called *tickets* and managed in an ITS [Sai+17]. We store both feature tasks and features in the ITS, whereas code and commits are stored in a VCS. In the ITS, developers can create distinct decision knowledge elements linked to the respective features and feature tasks. In the VCS, developers textually capture decision knowledge in commit messages and code. We encourage developers to mark it as such knowledge using *decision annotations* (cf. Sect. 6.5.3, Listing 6.1), as suggested by Hesse et al. [Hes+15]. The identifier of the feature task is added to the commit message. This satisfies the finding by Codoban et al. [Cod+15] that a good commit message expresses the rationale of the change and provides a link to requirements. Therefore, we use decision annotations, feature task identifiers in the commit messages, and distinctly documented trace links to establish tracing.

A first tool to capture these kinds of decision structures is the tool DecDoc, which is based on the DDM and allows to document design decisions collaboratively and incrementally [HKR16]. DecDoc supports the capturing of distinct decision knowledge elements, as well as implementation decisions, as annotations in the code. The DDM and DecDoc were evaluated by a retrospective analysis of decision-making processes of professional software designers [HP16]. The evaluation showed that it is feasible to document complex decision knowledge in DecDoc from collaborative and incremental decision-making processes [HKR16]. In order to be less intrusive, we now develop the ConDec tool support, which directly integrates into the ITS (JIRA) and VCS (Git) [Kle+18b]. ConDec comprises the features of DecDoc and

more features, such as the capture of decision knowledge when committing code as part of the commit message.

6.5.2 *Decision Knowledge Triggers*

CSE involves implementing and delivering many small increments. Practices advancing these increments are ideal to integrate *decision knowledge triggers*, that is techniques that trigger developers to capture and use decision knowledge. They are ideal because they are regularly performed by developers. Furthermore, they comprise practices that indicate that developers either *start* or *finish* work (Table 6.5). Practices that indicates *start* are to open a feature task and to create a feature task branch. Practices that indicate *finish* are to commit code, merge a feature task branch, or close a feature task. Before performing a *finish* practice, developers might have made important decisions. Thus, when developers perform a *finish* practice, we want to trigger them to explicitly capture decision knowledge. When developers perform a *start* practice, we want to trigger them to use existing decision knowledge to make sure they *consider consistency between old and new decisions*.

Figure 6.12 shows a state diagram of decision knowledge in CSE. The labels of the transitions indicate the type of CSE practice (*start* or *finish*) that developers perform. The *start* transitions always involve that developers make decisions. In

Table 6.5 CSE practices to trigger developers to document and exploit decision knowledge

Tool	CSE practice	Type
ITS	Start feature task	*start*
	Close feature task	*finish*
VCS	Create feature task branch	*start*
	Commit code	*finish*
	Merge branches	*finish*

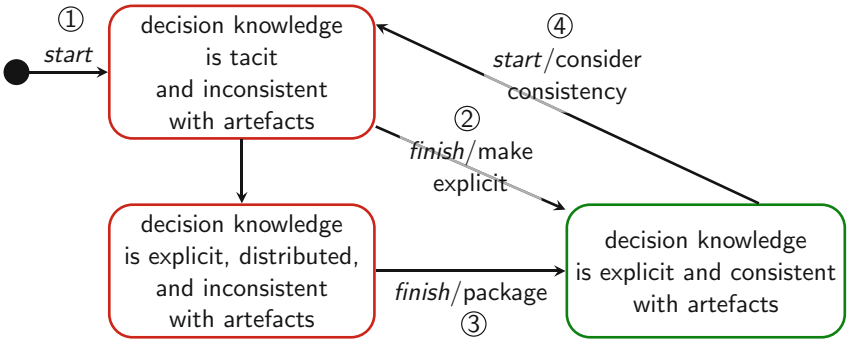

Fig. 6.12 State diagram of decision knowledge and artefacts. The state on the lower right side is the preferred state

addition, the tasks on the right side of the labels (*make explicit*, *package*, and *consider consistency*) need to be performed by developers for certain transitions. The ConDec approach supports these tasks: The integration of tool support into short-cycled start and finish practices *triggers* developers to explicitly capture decision knowledge consistent with artefacts and exploit it afterwards.

At the beginning of the work, decision knowledge is often *tacit* in the head of a few developers (Fig. 6.12-①). If decisions are not tacit, they are often discussed informally and captured partly and in a distributed manner, such as in issue comments [Hes+16], commit messages, pull requests [Bru+14], wikis, emails, chat messages [Alk+17a], or Internet relay chat channels [Alk+18]. We refer to this decision knowledge as *distributed knowledge*. This knowledge is hard to access later and might even be outdated. Therefore, we consider tacit and distributed knowledge as inconsistent with artefacts (cf. Fig. 6.12, left). Decision knowledge and artefacts become inconsistent as soon as they are created or changed. Transitions between consistent and inconsistent states are frequently recurring during CSE, while some artefacts are in a consistent and others in an inconsistent state at the same time. We describe the techniques behind the decision knowledge triggers in the following.

Making Tacit Decisions Explicit

Many decisions remain tacit, that is they are not captured anywhere but are already incorporated in the software. We present developers with *abstractive summaries* of changes to software artefacts when they perform a finish practice (Fig. 6.12-②). By presenting *abstractive summaries*, we want to trigger developers to make tacit decisions explicit, that is to reconstruct decision knowledge. This approach builds on the summarisation of source code changes, as suggested by Cortés-Coy et al. [Cor+14]. Tool support extracts change sets by comparing the code before and after the change. These change sets are the basis for generating an abstractive summary.

Packaging Distributed Decision Knowledge

Developers are presented with relevant distributed decision knowledge when they finish an implementation, as indicated through a finish practice (Fig. 6.12-③). They can check whether the decision knowledge really reflects the changes made. Thereby, we want to trigger them to package the most important decisions and to link them to the corresponding feature, feature task, or commits.

We present relevant distributed decision knowledge as *extractive summaries* using two techniques: (1) Developers can explicitly mark decision knowledge using decision annotations, as presented by Hesse et al. for code [Hes+15] and Alkadhi et al. for chat messages [Alk+17b]. Similarly, they are enabled to apply such decision annotations in other CSE artefacts, for example in comments to feature tasks, pull requests, or wiki pages. (2) We mine the unstructured distributed decision

knowledge by machine learning techniques similar to Rastkar and Murphy [RM13], Rogers et al. [Rog+14], Bhat et al. [Bha+17], and Alkadhi et al. [Alk+17a, Alk+18]. All of these techniques require a gold standard to train a supervised classifier. It needs to be investigated to which extent such gold standards can be generalised to identify decision knowledge from different types of CSE artefacts.

Criteria for relevance for inclusion in extractive summaries could be a direct reference (e.g. decisions captured in the code to be committed) or an indirect reference (e.g. decisions mentioned in a recent chat message or feature task comment by the developer).

Considering Consistency Between Decisions

To ensure consistency between decisions, we focus on practices that indicate that a decision is to be taken (Fig. 6.12-④). One example is when a developer sets the status of a feature task from *open* to *in progress*.

By presenting relevant decision and system knowledge, we want to trigger the developers to take previous decisions into account when working on the new feature task. This supports developers during the implementation of features. Criteria of relevance are derived from the trace links in Fig. 6.11. For example, relevant decision knowledge and code are those from other feature tasks that are related to the same feature.

6.5.3 Application to the Case Study

In the following, we use the CoCoME evolution scenario described in Sect. 6.3.3. In this scenario, the CoCoME sales system is extended with new payment possibilities. That is, one feature should enable the CoCoME customer to pay via Bitcoins and another feature to pay via PayPal. First, the requirement engineer (product owner) creates a feature task to implement the Bitcoin payment feature. The feature task is assigned to developers, who set the status from *open* to *in progress* and create a feature task branch to work on this feature task. Thus, they perform a start practice, as indicated in Fig. 6.12-①. The developers collaboratively discuss the design. One developer suggests extending the `IBank` interface with new payment methods, while another developer states that there are bank regulations that forbid to easily change that interface. The developers decide that a new `IPayment` interface could be added. Thus, the developers create the `IPayment` interface that contains the `authenticate` and `pay` methods.

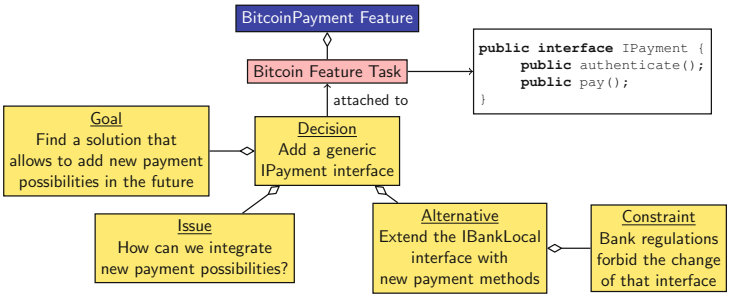

Fig. 6.13 Decision for adding the IPayment interface and related knowledge

Scenario for the Explicit Documentation of Decision Knowledge

The developers explicitly document decision knowledge consistent with artefacts. One possibility to document decision knowledge is that the developers write it in the code using decision annotations (Listing 6.1). Similarly, the developers can document the decision knowledge in the commit message or in the ITS. Consequently, the decision knowledge is consistent with the feature, feature task, and code, as depicted in Fig. 6.13. The knowledge can be accessed from each of these artefacts. For this purpose, it does not make any difference whether the developers document the decision knowledge in the VCS or ITS.

Scenario for Making Tacit Decisions Explicit

Imagine the developers did not document the decision *Add a generic IPayment interface* in the decision annotations (Listing 6.1). However, the decision knowledge resides tacitly in the head of the developers. When the developers commit the code changes, they perform a finish practice (Fig. 6.12-②). Since the code change

Listing 6.1 Example for using decision annotations during implementation

```
1 /* @Decision Add a generic IPayment interface
2  * @Issue How can we integrate new payment possibilities?
3  * @Goal Find a solution that allows to add new payment
       possibilities in the future
4  * @Alternative Extend the IBank interface with new payment
       methods
5  * @Constraint Bank regulations forbid the change of that
       interface */
6 public interface IPayment {
7     public authenticate();
8     public pay();
9 }
```

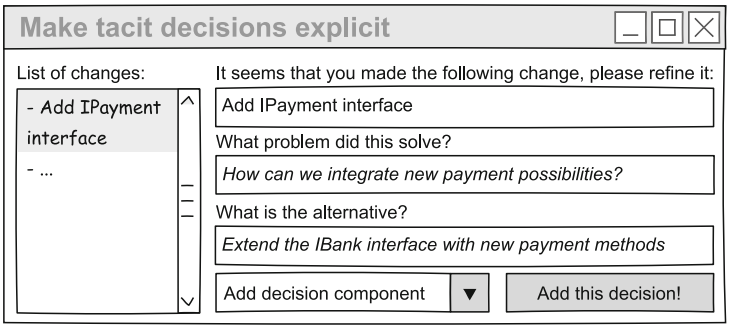

Fig. 6.14 A summary of changes illustrated as a sketch. The italic text is manually added by the developers

contains the addition of a new interface, the summary *Add IPayment interface* is suggested to them (Fig. 6.14). The developers approve that the summary of the change belongs to an important decision and reconstruct additional information on the decision problem (issue) and its alternatives. In particular, this supports developers to reflect about naturalistic decisions.

Scenario for Packaging Distributed Decision Knowledge

Imagine the developers did not document the decision knowledge as depicted in Listing 6.1 but discussed it in a written form, for example in the comments to the feature task, chat messages, Internet relay chats, or pull request for the feature task branch. Developers perform a finish practice when they close the respective feature task (Fig. 6.12-③). Tool support extracts relevant distributed decision knowledge from the original source (i.e. comments to the feature task, chat messages, Internet relay chats, or pull request). For example, the distributed decision knowledge is expected to be relevant when it was recently mentioned by the same developers. Further, the decision knowledge is classified by a machine learning approach and presented to developers, as shown Fig. 6.13. Since the developers discussed the addition of an IPayment interface, the decision *Add a generic IPayment interface* is suggested to them. Developers acknowledge that *Add a generic IPayment interface* is a decision they made and that the related decision knowledge is correct. The decision knowledge is stored inside of the ITS and gets linked to the feature task (Fig. 6.13).

Scenario for Considering Consistency Between Decisions

Imagine that the implementation of the Bitcoin payment feature task was finished and the decision knowledge is documented, as shown in Fig. 6.13. The feature

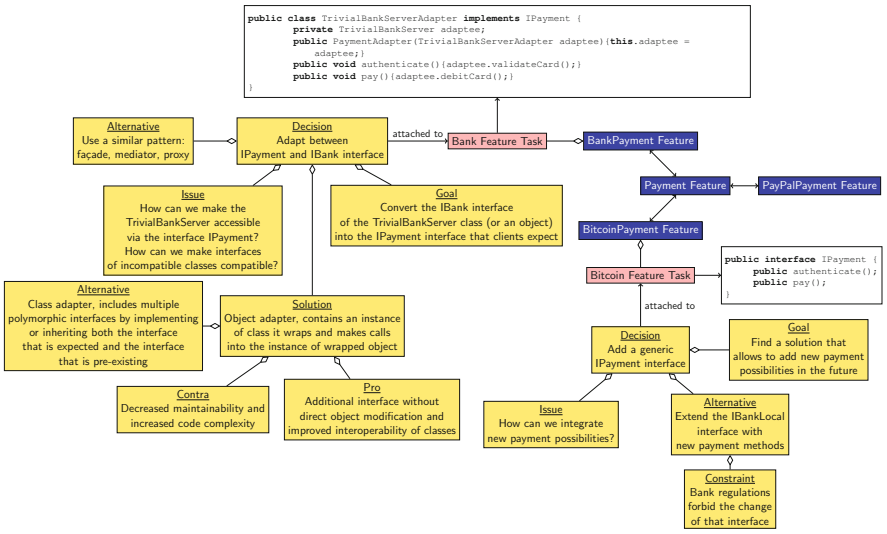

Fig. 6.15 Important decision knowledge related to the implementation of the payment feature

task to implement the PayPal payment feature is assigned to other developers. Figure 6.15 shows how decision knowledge is visualised in the context of related artefacts. When the developers set the status of the new feature task to implement the PayPal payment feature from *open* to *in progress*, they perform a start practice (Fig. 6.12-④). Since this feature is linked to the Bitcoin payment feature (Fig. 6.15), the code of the `IPayment` interface, as well as the decision knowledge *Add a generic IPayment interface*, is presented to the developers. Thus, they will learn about the integration of new payment possibilities and make decisions consistent with this previous one. The decision knowledge for the usage of the adapter pattern (cf. Sect. 6.3.3) can also be accessed.

6.6 Related Work

In this section, we discuss related work regarding the challenges of ensuring the consistency and minimising the intrusiveness of the design decision documentation.

6.6.1 Documentation Consistent with External Decision Knowledge

The following section summarises related work for the AM3D approach by Durdik [Dur14]. One of the most related approaches to the AM3D approach is Software

Engineering Using Rationale (SEURAT) by [BB08] including its extension presented by Wang and Burge [WB10]. It supports architects and developers by finding a pattern for a given problem. The design rationales and the design decisions during the decision-making are stored. Furthermore, SEURAT includes questions that have to be answered during the decision-making process in order to find the correct pattern. The purpose of these questions, however, is different from that of the AM3D approach. In SEURAT, the questions are used to find information, which is required before the decision can be made. They also specify which source of information is used to answer them.

Zimmermann et al. [Zim+08, Zim11] introduce a decision framework. The approach is based on reusable architectural decision models. The goal of the approach is to support developers and architects during the decision-making process, in particular during the phases decision identification, decision-making, and decision enforcement. The main focus of the approach by Zimmermann et al., however, is on the reuse of decisions and decision-related information itself, while AM3D focuses on the reuse of solutions.

The AM3D approach is not an expert system approach (see Table 3.1 in Durdik [Dur14]). The main difference between an expert system and the AM3D approach is that AM3D goes beyond a typical expert system as it helps users not only by finding a suitable solution but also by evaluating the solution, comparing it with other solutions, and documenting the found solution together with its decision rationales. However, Durdik [Dur14] pointed out that some expert system approaches, such as Garbe et al. [Gar+06], are also related to the AM3D approach as they use questions in order to choose a software pattern for a given problem.

6.6.2 Documentation of Decision Knowledge Consistent with Architecture and Code

There is also related work regarding the relationship between models and code. The field of model/code co-evolution describes how models and code can evolve together. Work in this area usually focuses on one specific type of model. For example, Langhammer [Lan17] describes an approach for the co-evolution of Palladio architecture models and Java program code. Langhammer describes rules that preserve a consistent relationship between the architecture model and the program code during changes on either side. The Codeling approach, presented in Sect. 6.4, instead allows for co-evolution between arbitrary object-oriented program code and model languages, as long as the latter can be represented with a specific subset of the Ecore meta model [Kon18].

Approaches for the co-evolution of models and code often do not consider the evolution of the underlying languages. Rocco et al. [Roc+14] explicitly describe language evolution as an aspect of model/code co-evolution. When a system is modelled using meta models and a corresponding code is generated, the evolution of

the meta model is a challenge. Such changes can break the code generators. This is a case of model/code co-evolution: The meta model can be regarded as model, and the code generator can be regarded as code in the context of model/code co-evolution. The authors propose a co-evolution approach where model changes are propagated via well-defined transformations, which operate on the code and take the model difference as input. This approach can be used to handle architecture language evolution regarding model editors but not regarding the code that implements a system's architecture.

The synchronisation between models and between models and code is the focus of the research in (in)consistency management [Fel+15]. These approaches assume that two views upon a shared body of information overlap. When one view is changed in the overlapping part, these changes should be propagated to the other view. Consistency management deals with methods and tools to re-establish synchronisation. Existing consistency management approaches focus on coarse-grained program code structures, such as code files or classes and relate them to model elements. Konersmann [Kon18] argues that a more fine-grained abstraction level is necessary and implements such consistency relationships in Codeling. Vitruv [KBL13] is a more general approach to keep different views consistent. It bases on coupling EMOF-specified meta models. For the coupling of the Palladio meta model for architectural specification with Java, see the PhD thesis of Langhammer [Lan17].

In 1995, Murphy et al. [MNS95] presented an approach to bridge the gap between program code elements and higher-level software models. In their approach, a mapping is created between higher-level model elements and program code elements. The approach of Murphy et al. is limited to mappings between model elements and program code files, neglecting the structures within the code files. Approaches need to address structures within the code files to add decision knowledge to specific architecture elements in the code.

6.6.3 Non-intrusive Documentation of Decision Knowledge

A documentation technique is lightweight if developers require only little effort to document knowledge. In addition, a non-intrusive documentation technique enables developers to document knowledge in a lightweight way as part of their development practices. In the following, we discuss both lightweight and non-intrusive techniques.

There are several models to represent decision knowledge, for example *Question, Options, Criteria* by MacLean et al. [Mac+96] and the *Decision Representation Language* by Lee [Lee91]. In this chapter, we use the DDM to represent decision knowledge (cf. Sect. 6.2). The main difference in comparison to former models is that developers can explicitly model context knowledge in a fine-grained way and that all components of a decision can be nested and refined. The *collaborative and incremental nature of the DDM* allows for a flexible documentation of decision

knowledge in contrast to filling out static text templates. The DDM is suitable to represent decision knowledge from informal and thus lightweight decision-making processes [HP16, Hes+16].

Hesse et al. [HKR16] investigate whether other approaches (implemented in tools) allow to document decision knowledge in a collaborative and incremental way. They identified that Archie [Cle+13] and SEURAT [BB08] are most similar to the tool DecDoc. The main differences are that SEURAT does not support naturalistic decision-making and Archie does not support shared documentation. Alexeeva et al. provide a literature overview of 56 decision documentation approaches [APM16]. They identified that the approaches are concerned with the following goals: documentation, consistency, evolution, extraction, impact analysis, reuse, sharing, traceability, and visualisation. Twelve of the approaches have the goal of enabling architecture consistency or compliance checks and thus address the consistency challenge of this chapter. However, the usage of these existing approaches requires developers to perform additional steps. Instead, the ConDec approach (Sect. 6.5) is integrated into developers' daily practices, such as committing code. In this regard, the ConDec approach is less intrusive.

A lightweight approach to document decision knowledge are *decision annotations*. Decision annotations enable developers to classify information as decision knowledge. Hesse et al. [Hes+15] use decision annotations to capture decision knowledge in code. Alkadhi et al. present an approach to capture decision knowledge in chat messages using such annotations [Alk+17b]. The ConDec approach also uses decision annotations in commit messages and issue comments. The importance of rationale in commit messages is confirmed by Codoban et al. [Cod+15]. They criticise that commit messages as often being non-informative. Our approach combines annotations to important artefacts like code or commit messages with explicit decision models, as the former eases the capture and the latter eases the understanding of decisions.

Perhaps the most lightweight approach to capture decision knowledge is using informal, non-structured *natural language*. Recently, various approaches emerged that try to automatically identify and extract decision knowledge captured in non-structured natural language. For this purpose, they use machine learning techniques. Alkadhi et al. show how to automatically identify decision knowledge in chat messages [Alk+17a] and Internet relay chat channels [Alk+18]. Rogers et al. mine decision knowledge from bug reports [Rog+14], whereas Bhat et al. focus on issue comments in general [Bha+17]. The ConDec approach allows for the informal documentation of decision knowledge and integrates mining techniques into the daily work of the developers instead of applying them retrospectively. As part of their future work, Rogers et al. [Rog+14] and Bhat et al. [Bha+17] state that they are planning to integrate mining features into existing knowledge management tools. The ConDec approach picks up this idea, as described in previous sections.

Saito et al. [Sai+17] and Rastkar and Murphy [RM13] also exploit *knowledge documented in artefacts of the ITS and VCS* . Similar to them, the ConDec approach also uses commits to link code in the VCS to tasks in the ITS. The meta model in Fig. 6.11 makes these relationships explicit and shows how decision knowledge

refers to these artefacts. Saito et al. [Sai+17] developed an approach to retrospectively link commits to tasks. After applying their approach, they found that still 20% of the tasks were not documented in the ITS as issues but directly communicated to developers. The ConDec approach does not address such undocumented tasks but supports developers to make tacit decision knowledge explicit. In the approach by Rastkar and Murphy [RM13], extractive summaries of issues that relate to a certain piece of code are presented to the developers. The summaries are supposed to provide developers with the rationale for code changes. Unlike Rastkar and Murphy, the ConDec approach creates summaries during finish practices in order to trigger developers to document important decision knowledge.

6.7 Conclusion

This chapter presented three approaches regarding the elicitation, documentation, and exploitation of design decisions in the context of CSE and long-living, evolving software systems. These approaches focused more on either the challenges of intrusiveness or consistency.

The AM3D approach supports architects and developers in making rational design decisions *consistent with external decision knowledge, which is presented in a separate tool*. In addition, the Codeling and ConDec approaches focus on ensuring *consistency among decisions within a project, architecture, and code*. While the AM3D approach leaves open where to document the knowledge, the Codeling and the ConDec approaches use annotations. In the ConDec approach, *lightweight traceability* is established, whereas the Codeling approach uses *transformations (formal mappings) between architecture and code*. These transformations are more powerful than traceability links since transformations can be used to create decision models that are interrelated with architecture models and the corresponding code. Hence, changes in the models can be propagated to the code. However, transformations are more intrusive than traceability links because they require extra notations. In addition to using annotations, the ConDec approach also captures decision knowledge in commit messages and in the ITS. Further, the ConDec approach uses short-cycled CSE practices to support developers in documenting and exploiting decision knowledge. The presentation of decision knowledge supports developers in making consistent design decisions and design decisions consistent with the software artefacts. In particular, ConDec also needs to find a balance between (a) the extent to which it can support developers in documenting decision knowledge consistent with former decisions and artefacts and (b) the intrusiveness of the presentation of knowledge. Thus, there is a trade-off between lightweight capturing or having powerful consistency checks that need to be considered when setting up a software development project.

The presented approaches are a first step towards extending CSE with a *continuous management of decision knowledge*. The following enhancements are desirable. Durdik [Dur14] pointed out future work for the AM3D approach. For

instance, the AM3D approach can be extended to support behavioural models; that is, behavioural information contained in design patterns can be supported. Currently, the AM3D approach only supports component-like models. In Codeling, the information about pattern instantiations is integrated with program code. This integration only contains the decision and the name of the instantiated pattern. In the future, the implementation should be generated accordingly to actually implement the pattern, where possible. The ConDec approach is implemented in tools [Kle+18b]. We will evaluate the tool support during CSE projects that are part of a practical course at university. We will assess to which extent decision knowledge triggers support developers during CSE. In particular, we will investigate which knowledge is worth capturing. Furthermore, we will clarify how to maintain the knowledge in order to keep it useful and how to access the relevant parts of knowledge.

6.8 Further Reading

Using a Design Pattern Catalogue to Make Design Decisions The main idea and details about the AM3D approach are presented in the dissertation of Durdik [Dur14]. Durdik and Reussner [DR13] explain the rationale for using design patterns and pattern documentation. The ADVERT approach, which uses AM3D for design decision-making, is explained in [Kon+13].

Integrating Design Decision Models with Program Code The integration of architecture models with code is subject to the work by Konersmann [Kon18]. It is based on the idea of embedded models by Balz [Bal11]. The tools for creating and executing translations between architecture-related program code and models are available on https://codeling.de. Konersmann et al. describe variants of this approach, for example for integrating deployment model information [KH16] or behaviour models [KG15] with program code, and the use of integrated model information for locating and understanding errors [Kon14].

Continuous Management of Decision Knowledge The integration of project and system knowledge, in particular the joint management of decisions and work items, is thoroughly discussed by Paech et al. [PDH14]. The DDM was first introduced in [HP13]. Hesse et al. performed several studies that demonstrated the feasibility of the DDM to represent complex decision knowledge. In [Hes+14], they use the DDM to document decisions that address security requirements. In [HP16], they investigated the decision-making process during design sessions. In [Hes+16], they empirically investigated informal decision knowledge from the ITS of the Firefox open-source project. They found that the documented knowledge mostly concerned the decision context and that naturalistic decision-making is dominant over rational decision-making for both bug reports and feature requests. Hesse et al. describe their implementation of the DDM in [HKR16, Hes+15].

The ConDec approach is described by Kleebaum et al. [Kle+18a, Kle+18b]. Johanssen et al. in particular address the visualisation of decision knowledge in relation to usage knowledge [Joh+17b]. Tool support for the documentation and exploitation of decision knowledge in the ITS and VCS is available on https://github.com/cures-hub.