Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 536–557
The Journal of Systems and Software
www.elsevier.com/locate/jss

Documenting after the fact: Recovering architectural design decisions

Anton Jansen [a,*], Jan Bosch [b], Paris Avgeriou [a]

[a] Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700AV Groningen, The Netherlands
[b] Intuit, 2632 Marine Way, Mountain View, CA 94043, USA

Received 6 July 2006; received in revised form 31 July 2007; accepted 6 August 2007
Available online 29 August 2007

Abstract

Software architecture documentation helps people in understanding the software architecture of a system. In practice, software architectures are often documented after the fact, i.e. they are maintained or created after most of the design decisions have been made and implemented. To keep the architecture documentation up-to-date an architect needs to recover and describe these decisions.

This paper presents ADDRA, an approach an architect can use for recovering architectural design decisions after the fact. ADDRA uses architectural deltas to provide the architect with clues about these design decisions. This allows the architect to systematically recover and document relevant architectural design decisions. The recovered architectural design decisions improve the documentation of the architecture, which increases traceability, communication, and general understanding of a system.

Keywords: Architectural design decisions; Software architecture recovery

1. Introduction

Software architectures represent the design of a software system and the decomposition of a system into its main components (Bass et al., 2003; Perry and Wolf, 1992; Shaw and Garlan, 1996). Architectural design decisions underly the software architecture (Kruchten, 2004; Kruchten et al., 2006; Tyree and Akerman, 2005; Bosch, 2004). As such, software architectures can be seen as the result of a set of architectural design decisions (Jansen and Bosch, 2005; Bosch, 2004).

Software architectures are typically described in one or more software architecture documents. Architecture documentation approaches provide guidelines on which aspects of the architecture should be documented and how this can be achieved (Hofmeister et al., 2000; Clements et al., 2002; Kruchten, 1995). However, these approaches document only partially what an architecture is, as they lack rationale, rules, constraints, and a clear relationship to the requirements (van der Ven et al., 2006; Tyree and Akerman, 2005). This information is valued by practitioners (Tang et al., 2005) and helps in future design decision making (Falessi et al., 2006). Consequently, not only the architecture should be documented, but also its underlying design decisions.

To document architectural design decisions, two problems need to be addressed. The first problem is that the current notion of an architectural design decision is rather vague. It is unclear what is part of an architectural design decision and what is not. Consequently, this vague notion leads to problems in documenting architectural design decisions, as it is unclear what should be documented.

A second problem is that, in practice, software architecture documentation is often not well maintained or not created at all (Visconti and Cook, 2004). Reasons for this practice are the perceived benefits of documenting the architecture or lack thereof, and time pressure (Conklin and Yakemovic, 1991). In this practice, two distinct cases can be discerned.

In the first case, the time pressure to deliver the software is perceived to be so great that documenting is perceived as overhead. Consequently, no effort is spent to document the

* Corresponding author. Tel.: +31 50 363 3968; fax: +31 50 363 3800.
E-mail addresses: anton@cs.rug.nl (A. Jansen), jan_bosch@intuit.com (J. Bosch), paris@cs.rug.nl (P. Avgeriou).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.08.025

architecture. If an organization cares for the quality of its software and documentation, then it usually reserves some cleanup time after a deadline to update the documentation accordingly.

In the second case, an organization does not create or maintain architecture documentation. The knowledge of the architecture resides in the head of the architect. The lack of documentation then becomes an issue when the architect is no longer available, e.g. has moved to another project or company. The tacit knowledge (Nonaka and Takeuchi, 1995) of the organization is no longer sufficient to (completely) understand the system. To prevent this situation from happening, the architect is usually given the task to document the architecture before he or she becomes unavailable.

In both cases, the architect tries to guess the architectural design decisions *after* they were made. Therefore, these decisions are not readily available to the architect. Consequently, there is a need to recover them. The key issue this paper addresses is how an architect in these cases can systematically recover architectural design decisions.

The main contribution of this paper is a recovery approach, which systematically recovers architectural design decisions. The approach uses a template based on a conceptual model to describe the recovered architectural design decisions.

The rest of this paper is organized as follows; Section 2 introduces the notion of architectural design decisions and its conceptual model. Sections 3 and 4 present an approach for recovering architectural design decisions. In Section 5, this approach is validated with a case study. The approach is evaluated in Section 6. Section 7 presents related work and the paper concludes in Section 8 with conclusions and future work.

2. Architectural design decisions

2.1. Introduction

A notion is needed of what architectural design decisions are before they can be recovered. This section introduces our notion of architectural design decisions and presents a process and conceptual model describing them. The process model describes architecting from an architectural design decision perspective, which describes the context in which architectural design decisions are created. Complementary to this is the conceptual model, which describes the concepts that make up these decisions. In the next section, the conceptual model is used to underpin our recovery approach for these decisions.

In this paper, the used definition of an architectural design decisions is taken from van der Ven et al. (2006). It is defined as a specialization of the Merriam Webster (Webster, 2006) definition of a decision, i.e. 'a report of a conclusion':

> A description of the choice and considered solutions that (partially) realize one or more requirements. Solutions consist of a set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements.

An architectural design decision is therefore the result of a decision process that takes place while architecting. Fig. 1 illustrates this process. It presents the context in which architectural design decisions are made. Furthermore, it visualizes the close interaction between architecting and requirements engineering, as every activity can lead to the *Requirements engineering* activity. This is due to new insights acquired in the architecting activities, which lead to a better understanding of the problem domain. The architecting process consists of the following activities:

(0) *Requirements engineering*. Although the requirements engineering activity is not part of the architecting process, it closely interacts. Most importantly, it fuels the architecting process with different issues (e.g. requirements, drivers, decision topics, risks, and concerns) from the problem space. These elements form the main input for the activity of scoping the problem space.
(1) *Scope problem space*. Based on the issues at play in the problems space the architect makes a scoping (and thereby a prioritization) of these issues and distills it into a concrete problem. To put the problem in perspective, a motivation and cause of the problem is described as well. This scoping is needed, as the problem space is usually so big that an architect is unable to address all the issues at once.
(2) *Propose solutions*. The existing architecture description and the problem of the previous step form the starting point from which the architect tries to come up with one or more solutions that might (partially) address the problem.
(3) *Choose solution*. The architect makes a choice between the proposed solutions, which can entail making one or more trade-offs.
(4) *Modify & describe architecture*. Once a solution is chosen, the architecture description has to be modified to reflect the new status.

The architecting process presented here is a specialization of the generalized model of architecting by Hofmeister et al. (2005a). Their model discerns three main activities in architecting: architectural analysis, synthesis, and evaluation. From an architectural design decisions perspective, architectural analysis is the *scoping of the problem* (activity 1), architectural synthesis is *proposing solutions* (activity 2), and architectural evaluation is both *choosing a solution* and *describing the architecture* (activities 3 + 4).

2.2. A conceptual model

To refine and more formally define the definition of architectural design decisions, this section presents a con-

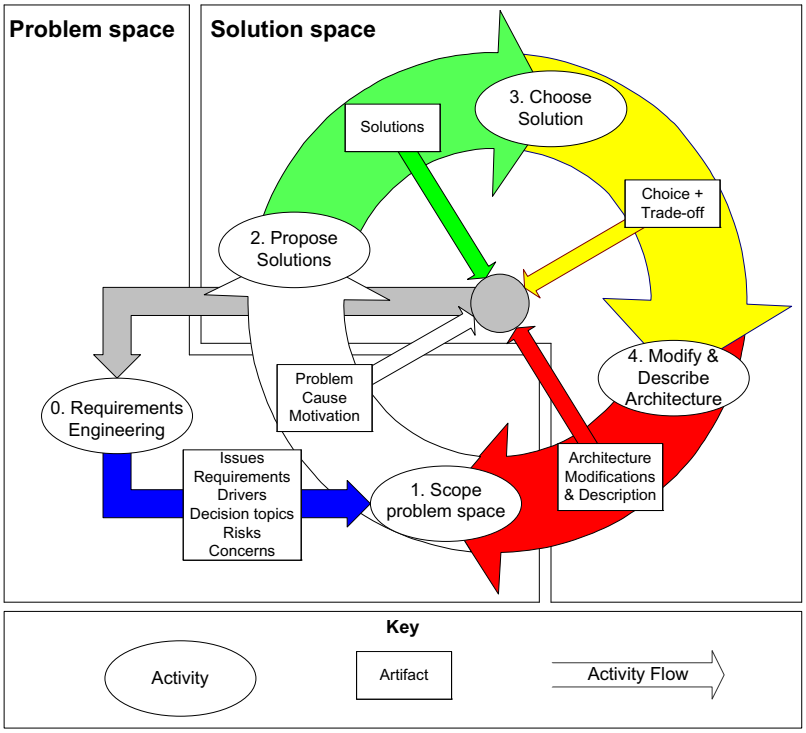

Fig. 1. The architecting process from an architectural design decision perspective.

ceptual model. The model describes the elements of architectural design decisions and their relationships in more detail. The presented model has been briefly presented before in earlier work (van der Ven et al., 2006; Jansen and Bosch, 2005) for forward engineering purposes. However, in this paper the conceptual model is used for recovery purposes and presented in more detail than before. The conceptual model is used later in this paper to define the elements of an architectural design decision that should be recovered.

Fig. 2 presents the conceptual model for architectural design decisions. For each concept of an architectural design decision, the corresponding activity of Fig. 1 is noted. At the heart of the model is the *problem* element, which together with the *motivation* and *cause* elements describes the *problem*. A *motivation* describes why the *problem* is relevant. The *cause* describes the causes of this problem. Solving the *problem* is the goal the *architectural design decision* wants to achieve. The *solution*s element contains the solutions that have been proposed to solve the problem at hand. A *choice* is made, which solution should be used, resulting in an *architectural modification* of the architecture.

Most of the model elements are made in a specific *context*. Apart from the *issue* concept, all the *context* concepts (i.e. concern, system, environment architecture, architecture description) and the relationships among them come from the IEEE 1471 standard for describing software architectures (IEEE/ANSI, 2000). An extension is made with the new concept of an *issue*, which is a generalization of the *concern* element. This new concept is needed, as a *concern* is traceable to one or more stakeholders. For an *issue*, this does not have to be the case, as it might directly come out of the *environment*. For example, an issue might be a problematic technical constraint coming forth from an earlier design decision.

An architectural design decisions is related to the *context* in three different ways. First, the *problem* element is the scoping of various *issues* of the problem space. Second, an architectural design decision modifies with an *architectural modification* the *architecture*, which will lead to an update of the *architecture description*. Third, an architectural design decision is part of the architecture, as this is a set of design decisions (Jansen and Bosch, 2005).

In the remainder of this section, the concepts that make up an architectural design decision are explained in more detail. Following is a list of these concepts and their relationships:

- *Problem*. A design decision is made to solve a certain problem. For example, the problem can be how specific requirements can be met or how the design can improve on some quality aspects.
- *Motivation*. This element contains the rationale for why the problem needs to be solved. This element therefore describes the rationale behind the scoping of the problem, as it explains the importance (and thereby the prioritization) of the problem. Usually, this comes from the requirements stated for the system. Together with the *problem* element this element determines the architecture significant requirements (ASR). As a requirement is per definition architectural significant if it is addressed by an architectural design decision.

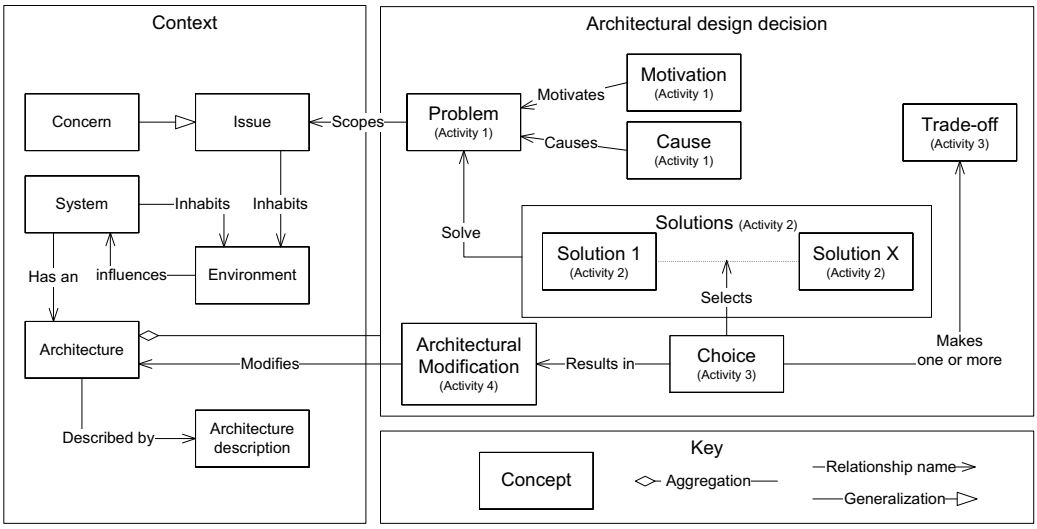

Fig. 2. Conceptual model for an architectural design decision.

- *Cause*. Often multiple causes for a *problem* do exist, this element describes them. This knowledge is important, as it decreases the chance that *solutions* are proposed, which are inadequate to solve the *problem*. Causes can include technical limitations, changed requirements, limitations imposed by previous design decisions, symptoms of other problems, etc.
- *Current architecture*. This element describes the architecture upon which the architectural design decision is made, i.e. the architecture before being modified by the decision. (see Fig. 2). The element is described by referring to the appropriate architecture description of the current version of the architecture.
- *Solutions*. To solve the (described) problem, one or more potential solutions can be thought up and proposed. For each of the proposed solutions, the following elements can be identified:
  - *Description*. This element describes the solution being proposed. The required modifications are explained and rationale for these modifications is provided.
  - *Design rules*. Design rules define partial specifications to which the realization of one or more architectural entities have to conform to. This defines parts of how the solution should be realized.
  - *Design constraints*. They define limitations or constraints on the further design of one or more architectural entities. These limitations and constraints are to be obeyed by future decisions for this solution to work.
  - *Pros*. Describes the expected benefit(s) from this solution to the overall design and the impact on the requirements.
  - *Cons*. Describes the expected negative impact on the overall design, as opposed to the *Pros*.
- *Trade-off*. The different solutions have typically different impacts on the quality attributes of an architecture. Hence, a *Choice* should decide on one or more *trade-offs*. In some cases, these trade-offs can be rather complex. This element describes the different quality attributes a trade-off has to be made between.
- *Choice*. For a problem there are often multiple solutions proposed, but only one of them can be chosen to solve the described problem. The choice involves selecting different *trade-offs* using the *pros* and *cons* of the solutions as arguments to rationalize the selection of a particular solution.
- *Architectural modification*. The chosen *solution* in the *decision* can affect one or more architectural entities and this element describes the changes to these elements.

3. Recovering architectural design decisions

This section introduces the Architectural Design Decision Recovery Approach (ADDRA). The approach is presented in two sections, this section presents the recovery steps that make up ADDRA, whereas the next section explains the knowledge externalization process that underly these steps.

ADDRA tries to recover architectural design decisions and documents them using the conceptual model presented in the previous section. The approach uses a combination of existing recovery techniques and the tacit knowledge of the original architect to recover architectural design decisions.

The basic idea of ADDRA is that changes in the architecture are due to architectural design decisions, thereby forming a clue about them. The tacit knowledge of the original architect is used to trace back from these changes to the decisions they originated from. The changes are used to focus the tacit knowledge of the architect and allow for a systematic approach of documenting this knowledge.

ADDRA consists of five steps, which are organized in an iterative process (see Fig. 3). Steps 1–4 are concerned with finding the changes in the architecture caused by

architectural design decisions. ADDRA first recovers relevant software architecture views for different releases of the system (steps 1–3). The differences between the views of one release are then compared to another subsequent release forming the *architectural delta* between two releases (step 4). The architectural delta in turn is used to recover the architectural design decisions (step 5). An example of an iteration of these steps is presented in Fig. 4. Remark that in the case of initial design, step 4 is skipped, as no earlier design exists and the recovered architecture views of step 3 are used as architectural delta instead.

ADDRA is performed iteratively and can be triggered at two points. The first point is in step 3 (see Fig. 3). When the information for recovering the software architecture views is inadequate, another iteration of step 2 takes place. The second point is after step 5, the description of the recovered architectural design decisions is deemed inadequate. In this case, ADDRA returns to step 3 to sharpen the description of the software architecture views. This process continues until no more significant progress can be made.

The recovery steps of ADDRA are far from trivial. Steps 1–3 (and partially 4) are not completely solved yet and remain under research by the design recovery community (see also Section 7.2). We briefly present these steps, point out potential tools, problems, and trade-offs to be made. The main focus of this paper is however on step 5; the recovery of architectural design decisions. This step is therefore presented in more detail.

3.1. Step 1: define and select releases

The first step in the recovery method is to define and select the releases of the system under consideration. Releases are snapshots of the system and its associated artifacts at a specific moment in time. A selection is made to limit the amount of low-level information the architect has to deal with. In this selection process, the following concerns play a role:

- *Effort*. Extracting design decisions from various artifacts is a *very* time-consuming operation. Therefore, a selection of the releases is made, which reduces the number of releases to examine. The exact number of releases to select is a balance between accuracy and effort. Selecting more releases improves the detection of relevant changes and thus improves the accuracy of the recovery, but also takes more effort to realize.
- *Timing*. Releases should have been an achievement target (e.g. a deliverable or a major release) in the past. If this is not the case, the release will likely be a snapshot of an unstable design situation. In this situation, design features are more likely in an undetermined state where a lot of design decisions are pending a decision. This will be reflected in the potential consistency and completeness among the artifacts, thus complicating recovery.
- *Scope*. The changes between two releases (i.e. their delta), should be chosen in such a way that the scope of the delta is the right size. If the scope of a delta is too large, then the chance that multiple design decisions have been taken on top of each other becomes greater. This makes the extraction of individual design decisions more difficult. However, the opposite is also not desirable. If the scope of the delta is too small, a design decision can still be work in progress and only parts will be visible. This will obscure the results of a design decision, thus hampering the recognition of the design decision. A rough estimation of this scope can be estimated by looking at changes in logs or log files, code metrics, and the development time between releases. The changes give an insight into the issues addressed and the scale of

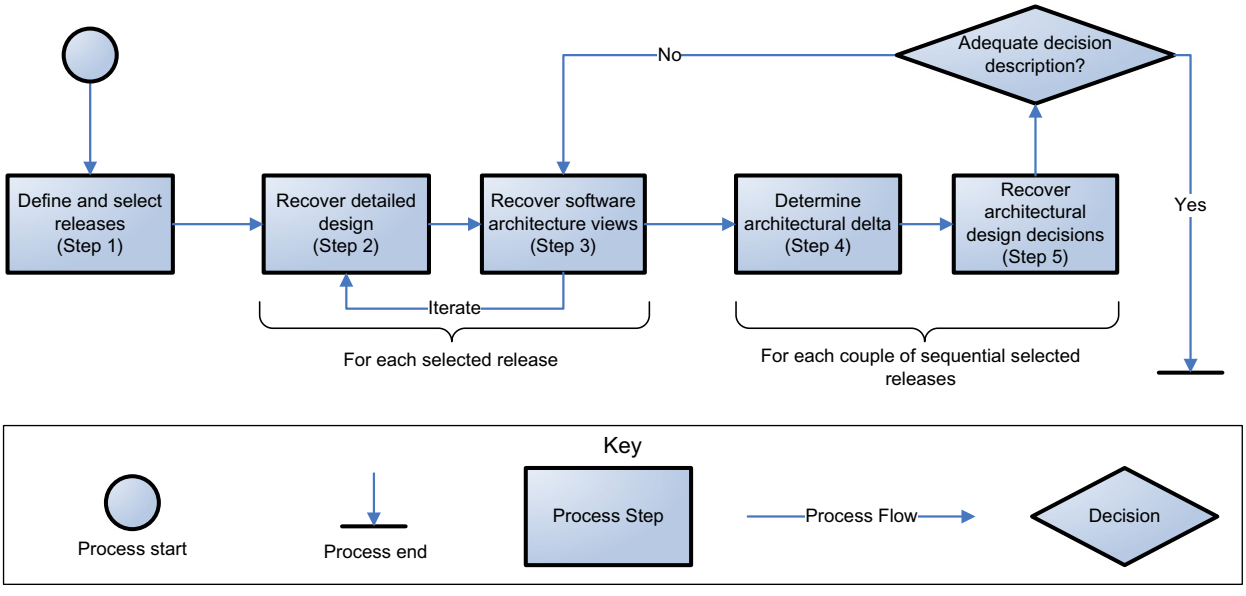

Fig. 3. Overview of the steps of ADDRA.

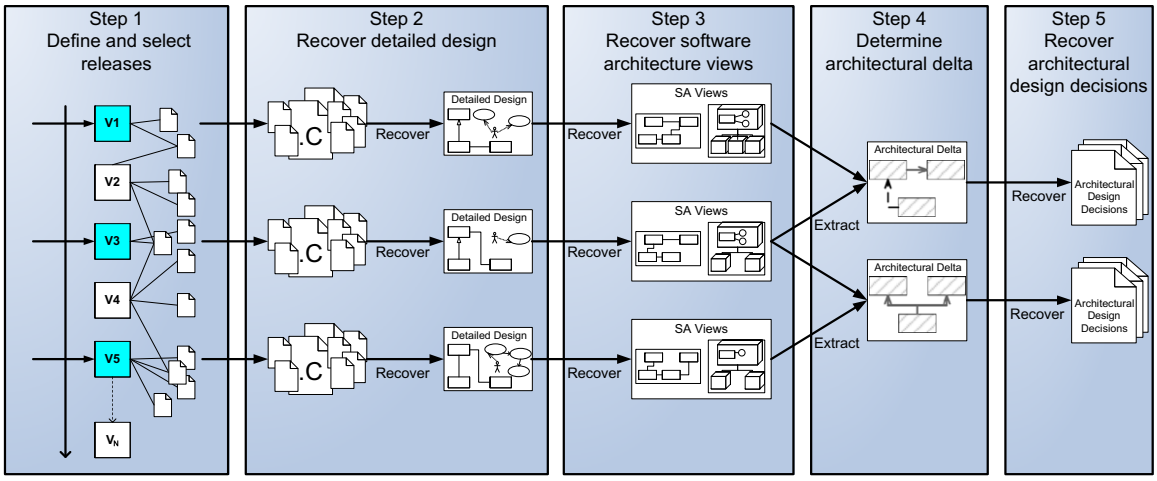

Fig. 4. One iteration of ADDRA.

impact of the design decisions made between two releases of a delta.

- *Availability and accuracy*. Both the availability and accuracy of the artifacts influence the usefulness of a particular release for the recovery process. The availability of artifacts (e.g. requirements, source code, and changelogs) for a release determines the potential value of the release. Accuracy is important as well, as inaccurate artifacts can easily lead to wrong conclusions about the system and therefore to the recovery of non-relevant and inappropriate decisions. The different versions of the artifacts should be linked to releases, such that an overview is created of the available artifacts. After this, the accuracy of (relevant) artifacts can roughly be determined by a quick scan and use of past experiences.

An open research challenge is to determine useful heuristics, which can guide the selection process. One could think of adapting existing cost prediction models (e.g. COCOMO II (Boehm et al., 2000)) to this end. However, to the best of our knowledge, no work exists in this area.

3.2. Step 2: detailed design

For each selected release, the detailed design is recovered. The detailed design forms the basis for abstraction later on to recover the software architecture. Therefore, the goal is not to recover a complete detailed design, but merely a useful abstraction that can be used later on. The detailed design can be recovered with the help of one or more recovery tools. Although these tools help in supporting the recovery of the detailed design, they still require the expert knowledge and guidance of the architect to find the right abstractions (van Deursen, 2002).

Depending on the relevant software architecture views of the next step, one or more detailed design views might be of interest. These detailed design views are usually represented in the Unified Modeling Language (UML). The main interest is typically in recovering the class (Gueheneuc, 2004), object (Tonella and Potrich, 2002) diagrams, and the use-cases. The first two focus on the structural aspects of the system. Recovery tools, such as Gueheneuc (2004) and Tonella and Potrich (2002), can typically recover (parts of) these views. The use-cases provide hints for issues that are addressed in the system and not easy to recover with tools, although tools like the one from Qin et al. (2003) can help in their recovery. If the dynamic aspect of the architecture is relevant, the processes, threads, and state-charts of the detailed design are of interest. Behavioral recovery tools like Discotect (Yan et al., 2004) are useful in these cases. More information regarding the manual reconstruction of different detailed design views and UML can be found in Booch et al. (1998).

Although recovery tools are very helpful in recovering the detailed design, they do have some shortcomings:

- *Distributed and dynamic behavior*. Recovery tools have difficulty in recovering the dynamic and distributed behavior of programs. Therefore, the recovered detailed design by these tools is often not complete.
- *Language specific*. Most useful recovery tools are very language specific, but in practice many software systems use multiple languages (e.g. C#, C++, C). Integrating the results of the various recovery tools together is often cumbersome or impossible due to the different concepts they use.
- *Configuration*. Configuring a recovery tool for a specific system is often not trivial. Often the knowledge of a domain expert is needed to filter out noise in the recovered results, which is created by used third party components, frameworks, and the specific platform used. Furthermore, some recovery tools require extensive interaction with the architect to come to a right configuration.

3.3. Step 3: software architecture views

The third step consists of recovering one or more views on the architecture. This step is very difficult to perform and can only successfully be achieved by the original archi-

tect. The views are recovered for each selected release, thereby providing a description of the software architecture at specific moments in time. To reduce effort, only a selected number of views are recovered. For this selection, two factors are considered: the relevance of the view and the relationship to other releases. As with the selection of relevant releases, the number of views to select is a trade-off between effort and accuracy.

Relevant views are the views that describe parts of the solution(s) to the main concerns at hand. Identifying these views is based on the recovered detailed design, design documents about the architecture, and knowledge from the architect. These information sources help in determining relevant views in the following way:

- The recovery of the detailed design refreshes the memory of the architect, which can provide clues to relevant concerns. Furthermore, it provides a stable ground for abstraction to various views on the architecture, which eases the creation of these views. For example, package diagrams form a basis to abstract to a module view.
- Design documents are created with the intent to describe certain aspects of the architecture, which were meant to address the main concerns at that time. Before a design document can be used, the consistency with the recovered detailed design should be checked. The document may not have been well maintained, thereby being no longer consistent with the design it describes. Identifying these inconsistencies provide valuable clues for design decisions later on.
- The guidance and knowledge of the architect remains the main factor in this selection process. In the end, the architect can make the best distinction between relevant and less relevant aspects and therefore mostly determines relevant views.

One set of views is needed for all the releases. In the next step of the recovery, similar views on the architecture are compared with each other. This requires that the same views be recovered for subsequent releases. Hence, the decision to create one particular view for a release influences the decision to create the same view for the previous and next selected releases. Furthermore, for these views to be comparable with each other they should conform to the same viewtype definition (Clements et al., 2002), i.e. use the same definitions, entities, relationships, naming, and notation. A natural way to achieve this is to use the recovered view of a previous release as a starting point for the next one and adapt it to conform to the associated recovered detailed design of step 2.

3.4. Step 4: architectural delta

The architectural delta of two releases is the change in the architecture that is required to go from the architecture of one release to another. Since the identified changes result from architectural design decisions, they can be used later on for the reconstruction of the architectural design decisions (see Section 3.5). Note, that this does include architectural design decisions that are unconsciously made. Even more so, they do not have to be made by the architect at all.

The architectural delta is determined by looking at the differences between the architectural representations of each release. The architectural views, representing the architecture, were reconstructed earlier on (see Section 3.3). The architectural delta is view independent, but differences between sequential views are a part of the architectural delta. The differences between these views *together* form an approximation of the architectural delta. To create this approximation, each entity (e.g. component, connector, module, uses relationship, etc.) found in the views is classified in one of the following five categories (this is similar to the work of Ohst et al. (2003) for UML diagrams):

- *Unmodified* entities have not been modified during the evolution. These elements are not part of the architectural delta, but represent the stable part of a system during change.
- *Modified* entities have been modified due to architectural design decisions. However, the entities still exist in the same view of both releases.
- *New* entities are entities, which did not exist in the oldest of the two releases, but does exist in the newest of the two. These entities represent new elements introduced into the design by architectural design decisions.
- *Moved* entities are entities that have been deleted in the new design. However, the aspects they represented are still in the design, but are now represented by other entities. Typical unit operations (Bass et al., 1998) that result in moved entities are abstraction, uniform decomposition, and compression of entities.
- *Deleted* entities represent aspects that are no longer relevant for the new design. These entities are not available anymore in the new design (not even moved), but did exist in the old design.

Inspecting the differences in the views determines to which category an entity belongs. For each couple of sequential releases the corresponding views are compared with each other. The identified differences between both views discriminates the entities in separate groups: *(un)modified, new, deleted* or *moved*. *(Un)modified* entities are entities available in both views. *New* entities are only available in the view of the latest release. On the other hand, *deleted* or *moved* entities are only available in the view of the first of the two releases.

Discriminating between *moved* and *deleted* entities is complicated and has no clear solution. To make this distinction we use the implicit knowledge of an architect. The architect can identify potential relationships between a *new* entity and a *deleted or moved* entity. *Deleted or moved* entities falling outside this group can be marked as *deleted*. Further detailed design or even code inspections are used

to determine the true nature of the remaining *deleted or moved* entities.

The architectural delta can be visualized for each viewpoint the entities have been classified for. For each of the type of entities defined in the viewpoint five different representations are made, which represent their classification. To visualize the changing aspect of the architectural delta for a particular viewpoint, unmodified entities are left out of the visualization. An exception is made if these entities are needed for providing a context for other entities, e.g. changing relationships. Corresponding numbers on the entities express the relationship between the moved entities and the new or modified entities incorporating them. An example of this notation for the module view is presented in the bottom part of Fig. 9, which also demonstrates the use of the correspondence numbers.

3.5. Step 5: architectural design decisions

In this last step, the architectural design decisions are recovered. The architectural design decisions are recovered using the template of Section 2 (see Fig. 6). This requires that the knowledge of the architect is externalized into a documented form. This is achieved with the architectural delta and a supporting externalization process.

Fig. 5 presents an overview of this externalization process. It illustrates the various activities and the order in which they are executed. The remainder of this section explains each step of this process. To execute these steps, several supporting knowledge transformations are needed, which are presented at the end of this section.

3.5.1. Step 5.1: analyze architectural delta

The first step is to examine the architectural delta for clues, that is: find changes in the delta that can be related together to form an architectural modification of a design decision.

3.5.2. Step 5.2: analyze situation

After this, the situation is analyzed and three intertwining and interfering activities happen: the definition of the context, describing the solution the architectural modification represents, and identifying which problem the modification tries to tackle. The last activity tries to recover the scope of the problem space that this decision addresses (see Section 2). Often this will entail issues coming from the (partial) fulfillment of one or more requirements. Once these three activities are completed, a fitting name describing the design decisions is defined.

3.5.3. Step 5.3: recover origin

The origin of the decision is recovered by determining the cause and the motivation of the problem at hand. Often a problem comes from the (partial) fulfillment of one or more requirements. In these cases, requirements form an excellent motivation of the problem. Hence, the motivation element creates traceability between the requirements and a part of the software architecture. Both motivation and cause provide a basis for the further refinement of the solution found, as the relationship with the problem space becomes clearer (and thereby the understanding of the solution).

3.5.4. Step 5.4: think up/recover alternatives

The next step is to think up or recover alternative solutions. Although several architectural solutions can be proposed in an architectural design decision, most of the time only one will actually be implemented (and therefore documented). For the reconstruction of architectural design decisions, this poses a problem, because information about the solutions not chosen is often unavailable in the arti-

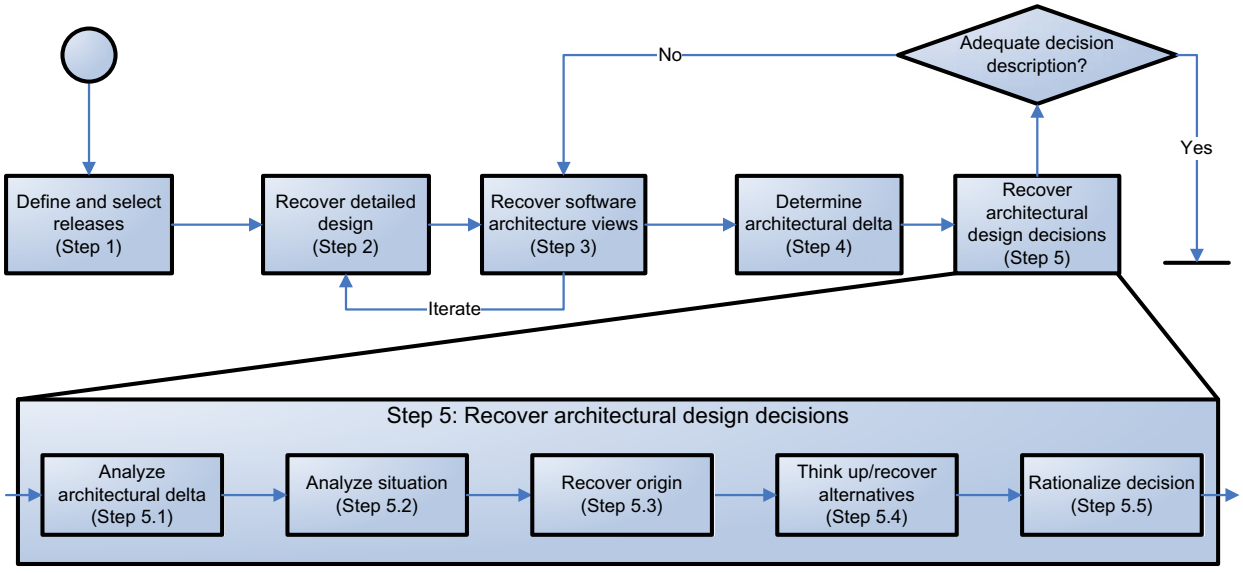

Fig. 5. Overview of the externalization process of step 5.

facts. Consequently, both the rationale of the Decision element (see Fig. 2) and the alternative solutions are lost.

Exceptions are architectural design decisions in which earlier made choices are undone. Due to lost rationale, the undoing of a choice is often a costly operation. Consequently, considerable effort is spent on motivating why the old choice is not appropriate anymore and a potential new one is. In this process, implicit knowledge of the design is made explicit.

A similar approach can be taken for the lost alternative solutions and rationale, but an alternative exists as well. *New* alternatives can be thought up and considered, as long as they fit in the current architecture and address the problem of the design decision. Due to the fact that the goal of documenting the architectural design decisions is to make the architecture more comprehensible. Therefore, it doesn't matter where the rationale comes from (Parnas and Clements, 1986). A discussion about the limitations this approach imposes is presented in Section 6.2.3.

Intertwined with the definition of alternative solutions is the description of the pros and cons of each solution. Often this activity leads to insights into new potential alternative solutions. A good source for pros and cons is formed by the expected consequences a solution will have on the quality attributes of the architecture. The expected consequences can be based on application generic knowledge (Lago and Avgeriou, 2006), which is typically documented in patterns (Harrison et al., 2007).

3.5.5. Step 5.5: rationalize decision

In the last step, the rationale of the decision element of the architectural design decision (see Section 2.2) is determined. This rationale should describe the reasons for choosing a particular solution. The previous step recovered the pros and cons for each solution. Based on this, the different trade-offs are determined that are made between these solutions. The rationale of the choice is based on the trade-off(s) being preferred in the chosen solution. Hence, we have to identify these preferred trade-off(s). Usually, these trade-off(s) come from the pros of the selected solution, as a design decision is intended to improve the design. Note however, that the rationale might be different from the original used rationale, due to a difference in the considered alternatives, but as pointed out in the previous step this is not a problem.

4. The knowledge externalization process

So far, we have explained the steps that need to be taken to recover architectural design decisions. However, the issue *how* the knowledge externalization process used in these steps should be performed has not been answered. To address this issue, we use Nonaka's theory of knowledge creation (Nonaka and Takeuchi, 1995). This theory distinguishes two types of knowledge: tacit and explicit knowledge. Tacit knowledge is the knowledge inside the heads of people, explicit knowledge is knowledge captured in documents, models, etc. One can convert one type of knowledge into another one by means of a knowledge conversion. Fig. 7 presents the knowledge conversions Nonaka distinguishes. In short, these are the following:

- *Externalization* is the process of articulating tacit into explicit concepts, which is the main focus of this paper.

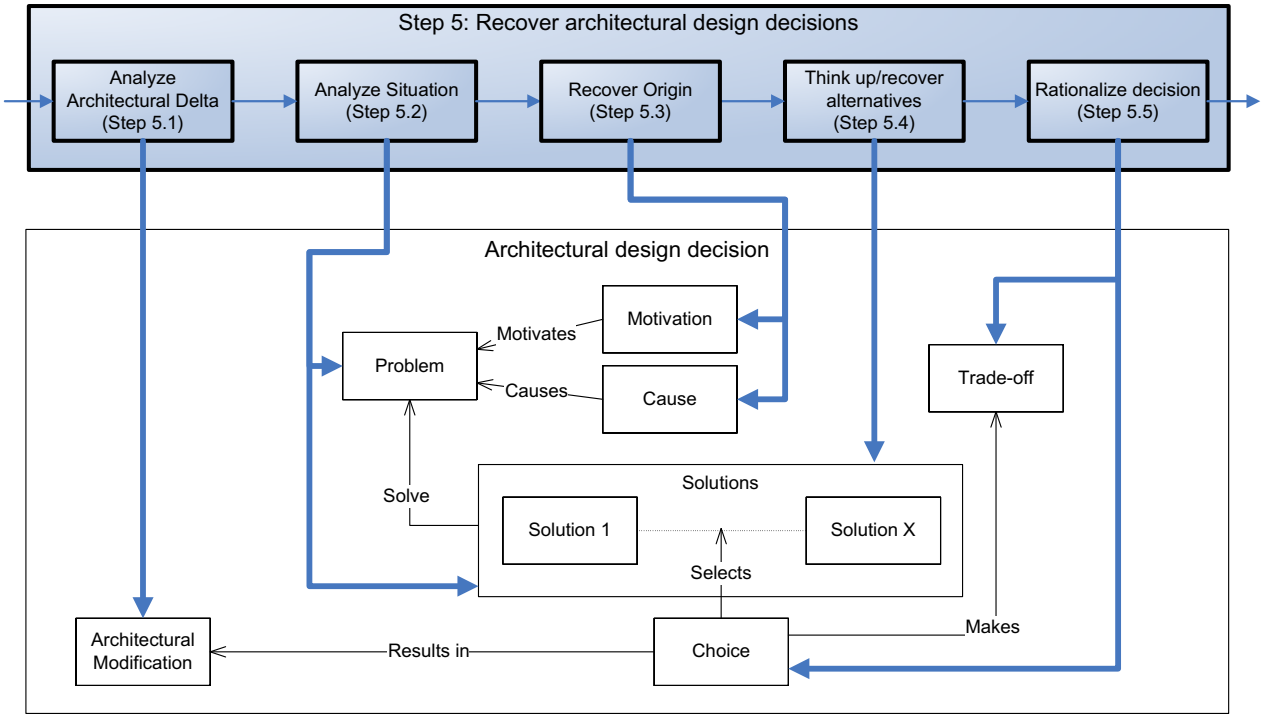

Fig. 6. The externalization process in relationship to the conceptual model.

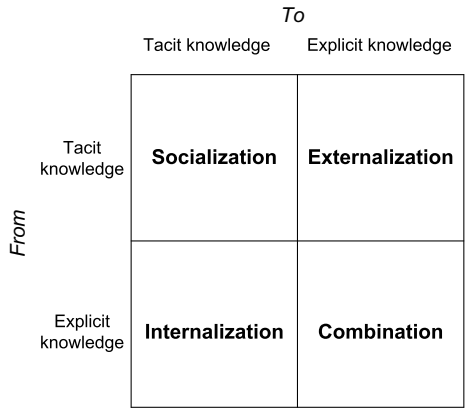

Fig. 7. Nonaka (1994) four modes of knowledge conversion.

- *Internalization* is the process of embodying explicit knowledge into tacit knowledge.
- *Socialization* is the process of sharing experiences and thereby creating tacit knowledge, such as a shared mental model and technical skills.
- *Combination* is the process of systemizing concepts into a body of knowledge (e.g. a document).

Based on these knowledge conversions, we identified four techniques for supporting the externalization process when recovering architectural design decisions. These techniques are often combined and used together in each of the different steps of the architecture design decision recovery process (see Fig. 5). The four techniques we identified are the following:

- *Introspection* is a form of externalization in which the architect asks him/herself questions and tries to come up with answers.
- *Inspection* is a combination of internalization and externalization. Inspection of various artifacts improves the understanding one has and allows one to express this knowledge.
- *Discussion* is a combination of socialization and externalization. During discussion tacit knowledge is shared, which is later made explicit.
- *Generalized domain knowledge* is a form of combination, where the situation at hand is combined with generalized domain knowledge from literature.

In the externalization process, the starting technique is often introspection by the architect. This can give rise to questions that cannot be directly answered. Hence, these questions require inspection or the use of generalized domain knowledge to be answered. The discussion technique is useful for validating the results of introspection. Furthermore, it can provide direction, similar to the generalized domain knowledge, when the knowledge of the architect is not sufficient. In the remainder of this section, each technique is explained in more detail.

4.1. Introspection (Externalization)

The main technique for transforming tacit knowledge into explicit knowledge is introspection or reflection. The architect asks him/herself questions and tries to come up with answers (see Section 6.2.1 for a discussion when the architect does not perform the recovery). Two distinct types of introspection can be discerned:

Knowledge of design under consideration. In this case, introspection examines the tacit knowledge of the architect about the design being recovered. This is primarily used to recover the line of reasoning once used or rationalize a new one. If certain questions cannot be answered, these questions are a starting point for inspection. Typical questions for the architect to ask him/herself in this case are:

- Why is this particular change made?
- What was the problem the change tried to resolve?
- Which requirements are involved?
- Which stakeholders were involved with this change/problem and what are their stakes?
- Are there other design decisions involved?

Knowledge of similar designs. An alternative form of introspection is to use past design experiences. Contrasting the situation at hand with earlier made designs can provide valuable knowledge. Typical questions to ask in this case are:

- Where does this situation deviate from similar past designed systems? Why are these deviations made?
- Where is this situation similar to past designed systems? What kind of decisions are always made in these cases? Why are these decisions made?

4.2. Inspection (Internalization + Externalization)

Inspecting the architectural delta provides clues of architectural design decisions. These clues form the basis for introspection and discussion, as they provide a context and focussing point. From our experience, the following things are fruitful to inspect:

- Isolated changes often form a good starting point to unravel architectural entities, which are affected by multiple changes. Inspection helps in determining whether the change is isolated or is a combined effect with other changes of the delta that are due to the same cause.
- Design decisions are typically made to refine earlier ones. Therefore, it is useful to inspect relationships between a change and other previous or further changes to the same and related architectural elements.
- Every change originated from a design decision. If there are still unaccounted changes, closer inspection of the commonalities and differences usually provides the required insight.

4.3. Discussion (Socialization + Externalization)

Discussion entails explaining, defending, and arguing about design decisions. This provides a natural way to trigger the externalization process. Furthermore, the interaction with others decreases the chance on a narrow minded line of reasoning. A good basis for discussion includes the following points:

- Discussion about problems. Is the problem really the problem or are there underlying issues leading up to different problems?
- Brainstorming about potential alternatives, which is also done in ATAM (Bass et al., 2003).

4.4. Generalized domain knowledge. (Combination)

Another technique for supporting the externalization process is comparing and contrasting the situation at hand with generalized situations known in the domain, i.e. use application generic knowledge (Lago and van Vliet, 2005). This includes using the knowledge from:

Architectural styles and patterns. Styles and patterns can be seen as distilled generalized architectural design decisions (Jansen and Bosch, 2005; Harrison et al., 2007). If (part of) the delta can be identified as a derived result from a style or pattern, the documented knowledge of these styles and patterns is useful. This documentation contains further directions on issues and trade-offs that need to be addressed when applying them. Consequently, this provides concrete guidelines on the decision making and therefore the architectural design decisions the delta is a result off. Good guiding questions in this perspective are the following:

- How is this solution specialized from the general style?
- What is the documented rationale to use such a style or pattern?
- How are the future directions tackled of a style or pattern addressed?

Tactics. Tactics describe architectural solutions for improving specific quality attributes of an architecture (Bass et al., 2003). This provides the opportunity to trace back an architectural solution to a problematic quality attribute. The question to answer in this case is:

- Does the architectural delta look similar to the architectural solutions proposed in a tactic?

Reference frameworks or platforms. For specific domains, reference frameworks or platforms (e.g. NET, J2EE) are available. They provide standard solutions to common problems in these domains. The same issues are often addressed in the system under examination. Furthermore, reference frameworks and platforms often provide explicit variability for critical design decisions, e.g. the type of scheduler to use in a real-time operating system. The system under examination can be examined to see if it makes these critical design decisions as well.

5. Case study: Athena

ADDRA is validated in this section by applying the approach on a case, which is called Athena. First, an introduction to the case is presented, followed by a description of the application of the different steps of ADDRA in subsequent subsections.

Athena is a system for (automatically) judging, reviewing, manipulating, and archiving computer program sources. The primary use is supporting students to learn programming. To develop the programming skills of a student, he or she has to practice a lot. Small programming exercises are often used for this end. However, providing feedback on these exercises is a laborious and time-consuming effort.

Athena helps students by testing their solutions to functional correctness and provides feedback (e.g. test results, test inputs, compilation information, etc.) on this. The system is capable of providing personal feedback to large groups of students in a relative short time. Consequently, the speed and quality of learning is increased when using a supportive submit system (Jansen, 2004). For a more elaborate description of the case, we refer to Jansen et al. (2003).

5.1. Step 1: define and select releases

The first step in the recovery approach is to identify and select relevant releases (see Section 3.1). During the evolution of Athena, all documents and code artifacts were maintained in a version control repository. This enables us to track the evolution of the Athena artifacts available in the repository, thus the software system itself.

To select appropriate releases in the Athena case, code metrics, log files, and release cycle time information were used. However, design documents were not available. Based on this information most of the design decisions seem to be taken during the initial development of Athena.

Although the case study is much bigger, the focus of this paper is on two releases in the initial development. The first is a prototype release named `develop`, the second is the first beta release called `beta 1`.

5.2. Step 2: detailed design

For both releases, the software architecture was recovered based on the detailed design. A case tool was used to inspect the source code and the relationships of the various classes. While reconstructing the detailed design a big sheet of paper was used to record classes of interest and their relationships. Already reoccurring design patterns were discovered, which could be used to group detailed design entities together. In similar ways, abstract super classes, the relationship between internal and external provided modules, and the internal code organization delivered parts for the abstraction to the architectural views.

5.3. Step 3: software architecture views

Fig. 8 presents for both releases `develop` and `beta 1` the resulting module and component & connector views (Clements et al., 2002). The views contain the following relevant architectural entities:

- ***OmniORB***, ***JavaORB*** general functionality needed for the CORBA communication.
- ***Arbiter*** automatically tests and judges submissions of students.
- ***Client*** tools for sending in submissions, viewing results, and configuring the Athena system.
- ***Manager Interface***, ***Object Interface*** interface between *Client* and server (*Manager*, *Object*).
- ***Manager*** handles queries, creation, and destruction of specific *Objects*.
- ***Object*** representation of the domain objects, e.g. submissions, students, tests, test-sets, etc.
- ***Connection Broker*** Database abstraction and convenience layer to access the *Oracle 8i* database.

5.4. Step 4: architectural delta

To determine the architectural delta between the two releases, the views on the architecture of Athena are used. First, the elements of the views are compared for subsequent releases and classified in one of the following groups: *(un)modified*, *new*, or *deleted/moved*. For the module view (see Fig. 8), this is done by looking at the differences between the `develop` and `beta 1` releases. Inspection of both leads to the identification of two *deleted/ moved* uses relationships, four new uses relationships, and the rest being classified as *(un)modified*.

The next step is to discriminate between *deleted* or *moved* entities. For example, the case of the uses relationship between the *Client* and *JavaORB*. Closer inspection of the *Client* reveals that this relationship has been *moved* to the *new* uses relationship with the *Manager Broker*.

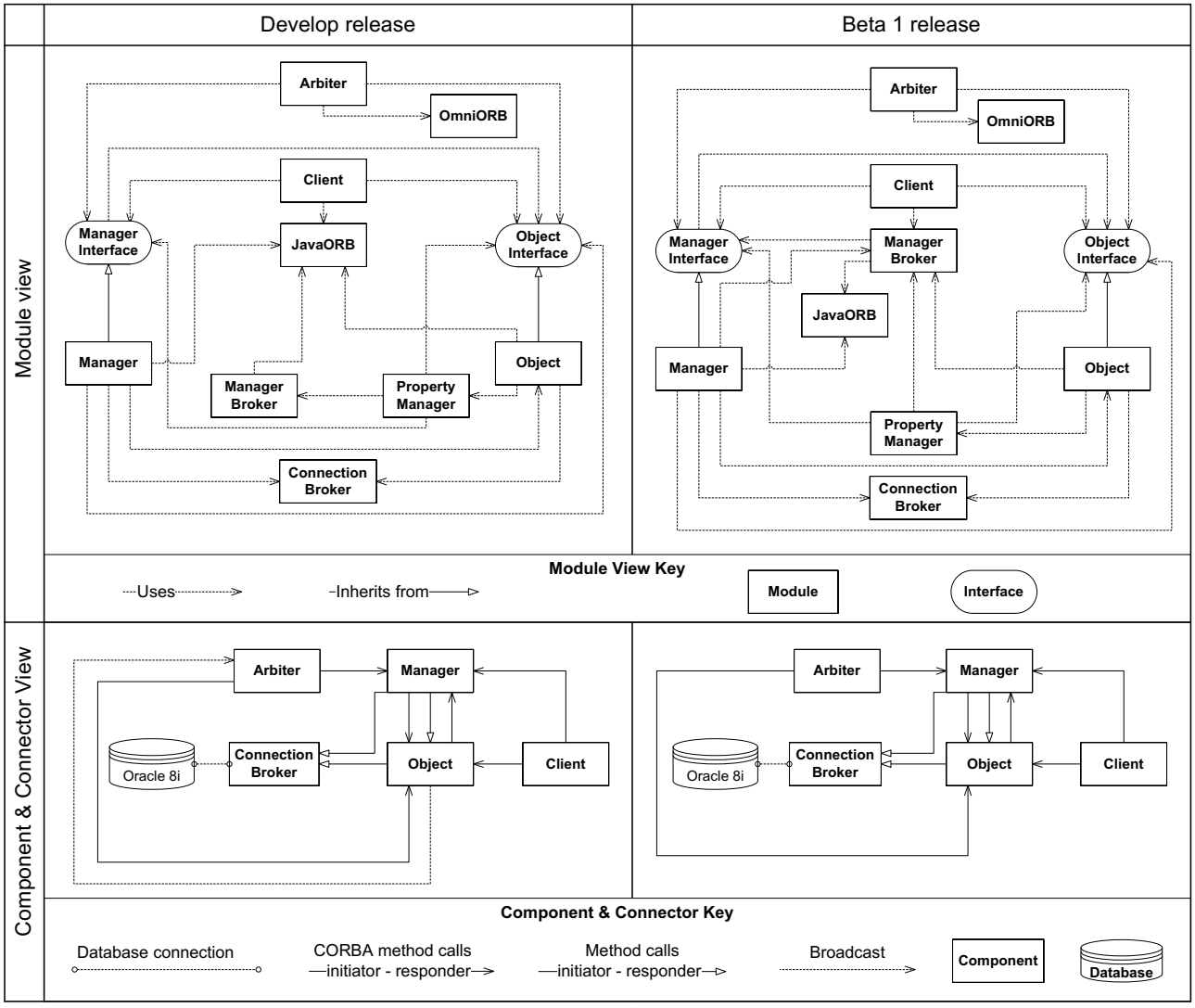

Fig. 8. Two views of the `develop` and `beta 1` releases of Athena.

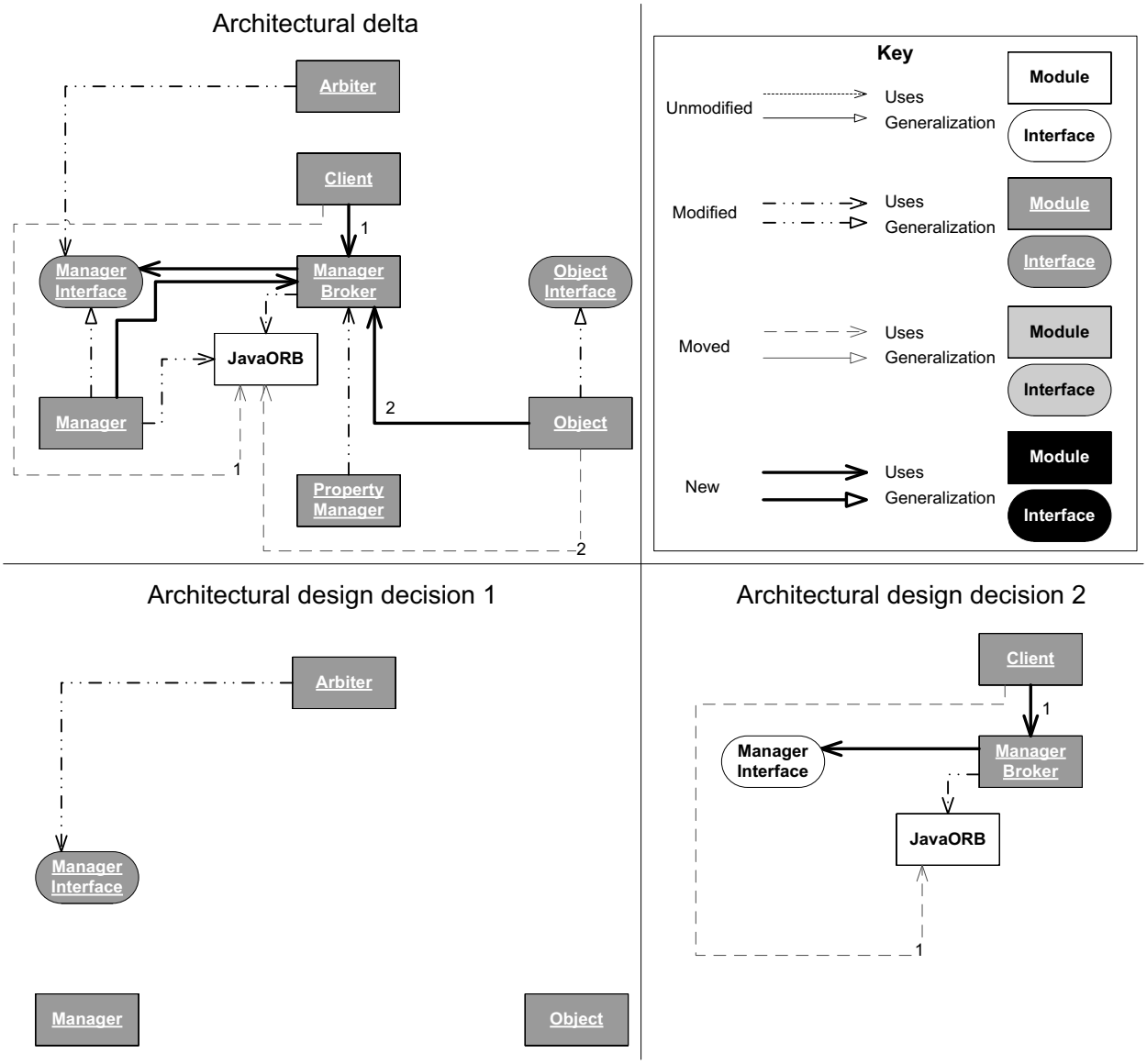

Fig. 9. The architectural changes between releases `develop` and `beta1` (top left) caused by architectural design decisions. Examples are the architectural module modifications of architectural design decision one (bottom left), and two (bottom right).

The distinction between *modified* and *unmodified* is partly based on the previous *moved* uses relationship. Substantial changes to the *Client* and *Manager Broker* have been made. Furthermore, the four *new* uses relationships have significantly affected some of the involved modules and interfaces, which are therefore marked as *modified*. The other elements have not significantly changed and are therefore classified as *unmodified*. The resulting architectural delta of the module view visualized in Fig. 9. In a similar way, the architectural delta of the component & connector view is constructed, as illustrated in Fig. 10.

5.5. Step 5: architectural design decisions

In this case study, a total of 15 different architectural design decisions have been recovered with ADDRA. Fig. 11 presents an overview of these decisions. Between releases `develop` and `beta 1` three different architectural design decisions were identified. Two of these decisions, the *Arbiter Job Control* and *Managing Manager References*, are presented here, while the third decision (the Domain Object Delegation decision) is left out due to space constraints.

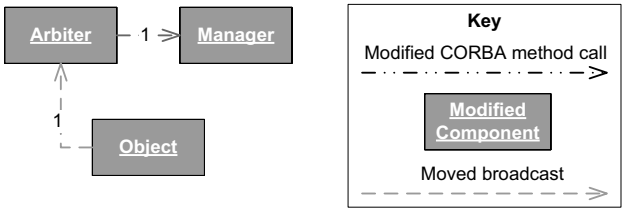

Fig. 10. Architectural delta of the c&c view of releases `develop` and `beta 1`.

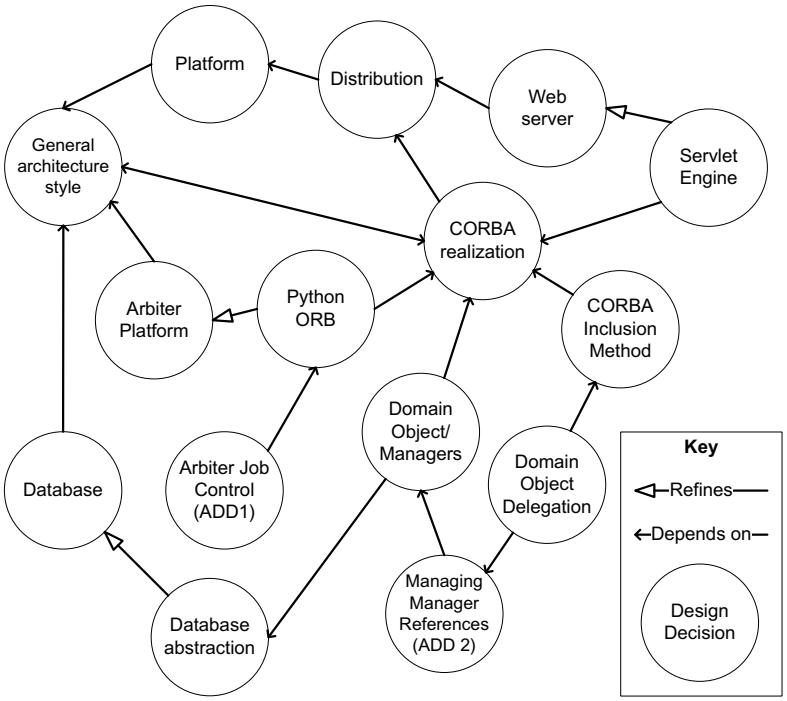

Fig. 11. The recovered architectural design decisions in the Athena case study.

The other 12 architectural design decisions have been recovered from the initial *develop* release.

For the *Arbiter Job Control* decision we exemplify each of the steps of ADDRA for the recovery of this decision. An in-depth description of the *Arbiter Job Control* and *Managing Manager References* decision is presented at the end of this section.

5.5.1. Step 5.1: analyze architectural delta

Inspection of the architectural delta of the component and connector view (Fig. 10) shows a moved broadcast relationship between the *Object* and *Arbiter* to modified CORBA method calls between the *Arbiter* and the *Manager*. For the delta of the module view *inspection* reveals that the corresponding modules (i.e. *Arbiter*, *Manager (Interface)*, *Object*) have been modified as well. Next, the visible changed relationships (i.e. modified, moved, and new) of these modules are inspected, which uncovers only significant changes in the uses relationship between the *Arbiter* and *Manager Interface* interface that have to do with the moved communication between the *Object* and *Arbiter*. Hence, the **architectural modification** identified is displayed in the bottom left of Fig. 9 for the module view and in Fig. 10 for the component & connector view.

5.5.2. Step 5.2: analyze situation

Further *inspection* revealed that the moved broadcast communication was used to notify the *Arbiter* of new submissions. The new **solution** instead lets the *Arbiter* poll the *Manager* for new submissions.

The problem this solution addresses is found with the help of *introspection*. We remembered that we ran performance tests, which uncovered a problem with the response time of the system when new submissions were created by students. This did not scale well with the number of *Arbiters* being deployed, as at any moment in time only one *Arbiter* was judging submissions, whereas the other *Arbiters* were idling. The intention of the architecture in this context is to process the submissions in parallel, which does not take place.

5.5.3. Step 5.3: recover origin

The **motivation** for this problem is found in a requirement with *introspection*, which originates from *knowledge of similar systems*. Past experiences with ACM programming contests showed such systems to have problems with providing timely feedback due to performance issues. As such, a requirement was defined that a submission should be tested and reported within 10 s to provide timely feedback to the students.

The **cause** of this performance problem is found by *discussion* among the developers and *inspection* of the source code responsible for the notification of the *Arbiters*. It turns out that the CORBA broadcast notifications work synchronously although they are defined as asynchronous. Further *inspection* reveals the cause of this behavior, which is presented at the end of this section.

5.5.4. Step 5.4: think up/recover alternatives

The polling solution that was recovered circumvents the problem by using a different communication strategy. An alternative **solution** that can be thought up is to address the cause of the problem, i.e. fix the CORBA broadcast implementation to work asynchronously. In this case, a dif-

ferent design strategy (a form of *generalized domain knowledge*) is used to think up this alternative.

5.5.5. Step 5.5: rationalize decision

The last step is to rationalize the **choice** by making a **trade-off** between the pros and cons of the solutions. Comparing the quality attributes of the two solutions (polling versus notification) finds differences in performance, maintainability, and ease of implementation (a form of costs). Of these three there is a trade-off between cost versus performance between the two solutions, as the broadcast solution has a negative impact on the costs and a positive influence on the performance and the polling solution has this influence and vice versa. With the help of *introspection* we know that the ease of implementation was the deciding factor to choose for the polling solution, as the pressure was high to deliver on time.

The other decisions were recovered in a similar way as the *Arbiter Job Control* decision. To document these architectural design decisions, a template is used that is based on the conceptual architectural model (see Fig. 2). Note, that the template presents the various concepts in a logical order, whereas ADDRA recovers these elements in a different order (see Fig. 6).

The template uses a condensed style, where each element of the conceptual model is typeset in bold, followed by an appropriate textual description of the element. An alternative style is to use a tabular approach, which takes up more space and is therefore not used in this paper.

Two noticeable adaptations are made to improve the template. First, the architectural modification is not documented as text, but uses the graphical notation for visualizing the architectural delta (see Section 3.4). Second, the solutions rationalized afterwards (see Section 3.5) are marked with an asterisk (*), thereby making an explicit distinction with recovered solutions. In the remainder of this section, the *Arbiter Job Control* and *Managing Manager References* decisions are presented in full detail using the template.

5.6. Architectural design decision 1: Arbiter Job Control

Problem: The response time of the system does not scale with the number of *Arbiter*s deployed. When running multiple *Arbiter*s only one judges submissions, the rest is waiting idle for work. *Motivation:* Multiple *Arbiter*s are needed to reach the performance requirement of a *10 s response time*. The current situation makes additional *Arbiter*s of no use. *Causes:* The CORBA broadcast facility that was used does not work as expected. The *Arbiter* is implemented in python, which in its core is a single threaded language. The *OmniORB* waits for the *Arbiter* to receive a broadcast message from a *Object*. Only after this has been done, the *OmniORB* will send an acknowledgment back to the *Object*. The broadcast is implemented in such a way that only after an acknowledgment the next receiver of a broadcast is notified. This results in the other *Arbiter*s having to wait in line to process the broadcast. All the while, they can be idle, waiting for new work to arrive. *Current architecture*: This design decision is made on basis of the software architecture as represented by the views for the *develop* release in Fig. 8.

- **Polling solution**
  Description: The *Arbiter* no longer receives broadcasts for new submissions from the *Object*. This solution uses a polling based approach instead. In this approach, the *Arbiter*s poll one of the *Manager*s for new work. If there is work, the work is returned as a pointer to the corresponding submission. **Design rules**: *Manager* provides a method for polling available submissions. **Design constraints**: None
  Pros: + Relatively easy to implement.
  Cons: − Load on the database increases.
  − Scalability of the number of *Arbiter*s decreases.
- **Broadcast solution***
  Description: The used CORBA broadcast facility is implemented in the *JavaORB*, which is a third party open source component. The solution is to modify this component in such a way it no longer waits on an acknowledgment before sending out the broadcast to the next receiver. Two threads are created, one that deals with sending out the broadcasts and one for handling the acknowledgments of these broadcasts. **Design rules**: The system should use the modified *JavaORB*. **Design constraints**: None
  Pros: + Very good response performance of the *Arbiter*s.
  Cons: − Maintenance increases, another component that evolves should be kept in sync.
  − Difficult to realize, as knowledge is required about the internal workings of the *JavaORB*.

Trade-off: Cost versus performance. **Choice**: The *Polling solution* was chosen, because it is easy to implement (i.e. has lower costs) and the expected negative impact on performance is not too great. The choice makes a trade-off in favor of costs (i.e. ease of implementation), as opposed to performance. **Architectural Modification**: From a module view perspective, the bottom left of Fig. 9 visualizes the changes and Fig. 10 does the same for the component & connector view.

5.7. Architectural design decision 2: Managing manager references

Problem: The various *Client*s all use their own code to initialize the *JavaORB* and resolve the needed references to the *Manager*s. **Motivation:** Removing this duplicate code reduces the chance of errors and makes it easier to replace the CORBA implementation (*JavaORB*). Maintenance costs are also reduced, because only one instance of the code has to be maintained. **Causes**: There is currently no standard way to resolve references to the various

*Manager*s. **Current architecture:** The architectural design decision is made in the context of the architecture after the *Arbiter Job Control* had been made. The module view is therefore the original module view of the `develop` release (see Fig. 8), modified by the architectural modification of this decision as visualized in the bottom left of Fig. 9. The component & connector view of the software architecture is the one depicted in Fig. 8 for the *beta 1* release.

- **Caching solution**
  Description: Remove the duplicate initialize code of the *JavaORB*. Add an extra layer in the form of the *Manager Broker*. This module can resolve references to the different *Manager*s, without any CORBA related information for the top layer. The *Manager Broker* caches the reference of a *Manager* for further repeated use. The caching is introduced to minimize communication overhead between the clients and the naming service providing the location of the *Manager*s. **Design rules**: None **Design constraints**: All access to the *Manager*s has to be done through the *Manager Broker* for the *Client*s.
  Pros: + Removing duplication increases maintainability.
  + Caching increases performance.
  Cons: − Caching decreases availability.
- **Layer solution***
  Description: This solution is similar to the *Caching Solution*, but without the caching part. **Design rules:** None **Design constraints**: All access to the *Manager*s has to be done through the *Manager Broker* for the *Client*s.
  Pros: + Removing duplication increases maintainability.
  + Solution is easy to realize.
  Cons − Performance decreases due to extra layer.

Trade-off: Performance versus availability. **Choice**: The *Caching Solution* was chosen, as performance is a major issue in this system and the unavailability of the system during upgrades is not regarded as problematic. The choice makes a trade-off preferring performance over availability. **Architectural Modification**: No modifications are visible in the component & connector view. For the module view, Fig. 9 visualizes the made architectural modifications.

6. Evaluation

The previous sections explained and exemplified the working of ADDRA. In this section, ADDRA is evaluated from three different perspectives. The first perspective is formed by the lessons learned while applying ADDRA on the Athena case. The second perspective presents the limitations of the approach. The third and final perspective outlines the benefits of ADDRA.

6.1. Lessons learned

6.1.1. Transitions between design decisions

The first lesson learned is the existence of transitions between design decisions. If an architectural design decision is undone and another (new) solution is chosen instead, a transition period exists. During this transition period, the design and the implementation will be incrementally adapted to the new (chosen) solution. It is not uncommon for these transition periods to last several months or even years, as business gives a higher priority to other decisions. If such a transition spans a period over one or more releases then it becomes part of the architectural delta of those releases. Consequently, there exists a transition period in which both solutions *coexist* and are partially realized, but as a *whole* realize the complete functionality.

Design decision models, including our own conceptual model, do not support these transitions, as these models view the transitions as atomic actions. To capture these transitions, an architectural design decision model should therefore support a more evolutionary model, which tracks the transition phases between design decisions.

Transitions between design decisions hinder architectural design decision recovery, because they complicate the architectural delta. The architectural deltas of two releases can contain ''leftovers'' of design decision solutions, which are still in transition to a new solution. Choosing the releases relatively far apart from each other in step 1 of ADDRA partially negates this problem. However, due to the possible long durations of the transition periods this might not always help.

6.1.2. Architectural views are subjective views

The second lesson learned is the subjectivity of architectural views. This is not a specific problem for ADDRA, but a general problem for recovery. Architectural views are subjective views, because abstraction is used to construct them. When architectural views are (re) constructed abstraction choices are made. Although the concepts a view should visualize are defined, the clustering used while abstracting is not. This leaves space for different interpretations of the system, i.e. non-deterministic reconstructing results for various views (van Deursen, 2002).

The reason for architectural representations to be subjective is the absence of architectural entities as first class citizens in the lower abstraction levels. No explicit relationship exists between the architectural entities (and the representations of them) and the entities implementing these architectural entities. A (standardized) first class representation of architectural entities could remedy this situation.

The subjectivity of architectural views has two important consequences for ADDRA. First, the architectural views might not be fully comparable to each other, as for the same entities different abstraction choices might be made. This results in an *approximation* of the architectural delta, which in turn hinders the recovery of architectural design decisions. ADDRA tries to counter this effect by reusing earlier made abstraction choices of previous releases (see Section 3.3).

Second, the subjectivity of the abstraction choices makes the outcome of ADDRA partially subjective as well. This is

because different people can make different abstraction choices, which can result in different architectural views on the selected releases. Therefore, they can recover a different approximation of the architectural delta and consequently can recover different architectural design decisions. ADDRA tries to counter this effect by employing the architect as the authoritative source for the abstraction choices. However, this does limit ADDRA to the availability of the architect (see also Section 6.2.1).

6.1.3. Solutions are sketchy or incompletely defined

The third lesson learned is the incomplete and sketchy way in which solutions are defined in an architectural design decision. Solutions can only give a general and incomplete description of how the problem at hand should be solved, as limited resources (e.g. time) only allow for an abstract description. Later in the project the solutions that are actually used are further refined.

Most of the knowledge systems and design decision models have support for this refinement process (see also Section 2). However, architectural representations lack the ability to represent this specialization process, because they focus on the *result* of the specialization process, not the individual steps. Some initial work dealing with this issue is the work of Roshandel et al. (2004), which tracks the evolution of an architecture description. Combining such an incremental architecture view with design decisions has been first done in our own work (Jansen and Bosch, 2005) and has been refined by Capilla et al. (2006).

For ADDRA this refinement process is not a problem, as ADDRA is performed after the fact. In other words, most of the necessary refining design decisions have already been made. However, as ADDRA does not recover these refining design decisions in the lower abstraction levels, the description of the solution in the recovered architectural design decision remains sketchy and incomplete. Therefore, traceability to these refining solutions is limited, which makes the recovered solutions sketchy and incomplete.

6.2. Limitations

ADDRA is not without limitations. In this subsection, the foremost limitations of ADDRA are outlined together with potential strategies how they could be addressed.

6.2.1. Availability of the architect

One important limitation of ADDRA is the assumption that the architect himself or herself performs the recovery process. This assumption is important, as ADDRA explicitly employs the tacit knowledge of the architect in the recovery process. For example, in step 5 the tacit knowledge of the architect is heavily used to transform the architectural delta into recovered architectural design decisions (see Section 4). However, in practice an architect usually does not have the time to perform an elaborate recovery as outlined in ADDRA.

A potential solution is to use other people for the time-consuming externalization and process. Although this reduces the externalization effort of the architect, additional socialization is needed to share the architectural knowledge of the architect with these people. Apart from the decision at which releases to look, steps 1–4 of ADDRA (see Section 3) could be performed by others. However, this does pose a risk to successful recovery, as the architectural delta might be misleading due to the subjectivity of the underlying architecture views (see Section 6.1.2).

The last step of ADDRA, step 5, requires the most tacit knowledge of the architect and is therefore hard to delegate to others. Supporting techniques (see Section 4), used in this step, that can be partially delegated are the inspection and use of generalized domain knowledge. For as both take explicit knowledge as input, which can be easily shared. The other two techniques, i.e. introspection and discussion, use tacit knowledge as input. Therefore these techniques require active participation of the architect, which makes it difficult to delegate them.

6.2.2. Selection of presented architectural views

In the Athena case study, ADDRA was presented using the module and component & connector views. However, ADDRA can be used for other architectural views as well. The idea behind the architectural delta (step 4, see Section 3.4) and the architectural design decision recovery techniques based on this delta (step 5, see Section 3.5) are view independent. In other words, they could also be used for other views. For example, a view from the allocation viewtype like the deployment view (Clements et al., 2002).

6.2.3. Lack of alternatives and trade-offs

ADDRA cannot always recover the considered alternatives and trade-offs of an architectural design decision. This is due to the absence of traces and identifiable effects of these decisions in the available explicit knowledge. Although ADDRA tries to recover these decisions by using tacit knowledge (see Section 4), this recovery is not always successful. In these cases, the tacit knowledge of the architect is inadequate, which is mainly due to the forgetful nature of the human brain.

The solution used in ADDRA for these cases is to think up potential alternative solutions that could have been used (see also Section 3.5.4). The trade-off made between the solution chosen and the thought up solutions can then be constructed by comparing the solutions with each other. Generally this strategy works rather well. However, a problem arises when in hindsight a superior alternative solution is thought up. The trade-off constructed in these cases then no longer aligns with the choice made in the decision. To solve this misalignment problem, the choice element should describe two parts. First, the element should describe the choice made among the inferior solutions. This is needed for the reader to understand the original choice. Second, it should describe the reason why the superior solution

was not an option on the table. This can be due to two different reasons:

- When the choice was made, the superior solution was excluded based on an invalid assumption.
- The superior solution was not considered at the time.

Both reasons are important architectural knowledge and should be documented in the choice element. In the case of exclusion on an invalid assumption, making these flawed assumptions explicit helps prevent the same mistake from being made again in the future. For both cases, the superior solution(s) form the perfect starting point to improve the design of the architecture later on. Furthermore, they are helpful in recovery, as it might well be that such decisions are undone and improved upon in later decisions.

6.3. Benefits

Although ADDRA is not without limitations, it still has it merits. In short, the approach has the following three benefits:

- *Systematic and disciplined approach*. ADDRA offers a detailed process that describes which steps are needed and in which order they should be executed. This process has two important benefits: (1) It steers the recovery process with clear goals and means, which prevents a recovery attempt getting "lost". (2) The systematic nature induced by this process reduces the chance of overlooking important architectural design decisions.
- *Explicit use of tacit knowledge*. Many architectural recovery approaches do not explicitly use tacit knowledge in their recovery. They focus more on the available explicit knowledge, e.g. source code, models and existing documentation. ADDRA is different, as it describes *how* the tacit knowledge of an architect can be used in conjunction with formal and documented knowledge. This gives ADDRA the benefit of finding more architectural design decisions than other approaches that do not use tacit knowledge.
- *Recovers the rationale behind the architecture*. ADDRA not only recovers what the architecture is, but also the underlying architectural design decisions of the architecture. The benefit of this is that these decisions make the architecture description more understandable, as the reader is supplied the rationale of how the architecture came to be.

7. Related work

The recovery of architectural design decisions is related to three research areas: software architecture, design recovery, and rationale management. Following is a description of each area and how it relates to recovering architectural design decisions.

7.1. Software architecture

Software architectures (Bass et al., 2003; Perry and Wolf, 1992; Shaw and Garlan, 1996) describe the the results of the main design decisions made for a system. As such, they can be seen as a collection of architectural design decisions (Bosch, 2004; Jansen and Bosch, 2005). Software architectures can be represented and described with the help of two types of approaches. One type is the documentation approach (Clements et al., 2002; Hofmeister et al., 2000; Kruchten, 1995), the other is an Architectural Description Language (ADL) (Medvidovic and Taylor, 2000; Oreizy et al., 1998). Both provide the architect with a vocabulary to reason about the architecture.

Documentation approaches (Clements et al., 2002; Hofmeister et al., 2000; Kruchten, 1995) use the concept of different views, i.e. a description of a particular aspect of the software architecture, to describe the software architecture as a whole. These approaches originally primarily focused themselves on the *result* of the decisions, not on the architectural design decisions themselves (Tyree and Akerman, 2005; van der Ven et al., 2006). Extensions to these approaches tie the views closer to the architecting process and architectural decisions (Hofmeister et al., 2005b; Bass et al., 2006).

As opposed to the documentation approaches, Architectural Description Languages (ADLs) (Medvidovic and Taylor, 2000; Oreizy et al., 1998) focus on a single aspect, e.g. the principle computation entities and their relationships. An ADL has exact semantics and describes allowed combinations of architectural entities and their relationships. Similar to the documentation approaches, the notion of architectural design decisions is also unknown in ADLs (Jansen and Bosch, 2004, 2005).

There is a growing interest in architectural design decisions within the software architecture community. Tyree and Akerman (2005) present a template for documenting architectural design decisions, which is tailored towards a forward engineering effort, as opposed to the template presented in this paper for recovery. Kruchten (2004) and Kruchten et al. (2006) have created a classification for design decisions and identify common relationships among them, which may be used to further classify the recovered decisions and create explicit relationships among them.

ADDRA focusses on the recovery of architectural design decisions. For forward engineering purposes, we have developed another approach called Archium (Jansen and Bosch, 2005; Jansen et al., 2007). The architectural decision model of Archium is very similar to the one presented in this paper (see Fig. 2). However, Archium extends this model by combining it with a requirements, architecture, and implementation model into one single unified model. This allows architects to document architectural design decisions with traces to related elements (e.g. requirements, or parts of the implementation).

Another useful purpose of the recovered architectural design decisions is for change impact analysis (Tang

et al., 2005), which allows for predictions about the effort required for undoing certain decisions. ADDRA can also be used to extend the pattern mining approach of Babar et al. (2006), where architectural design decisions are linked to well-known architectural styles (Shaw and Garlan, 1996) and patterns (Buschmann et al., 1996), as is done before for design patterns (Gamma et al., 1994) in Baniassad et al. (2003).

7.2. Design recovery

Design recovery usually focusses on recovering a design that is suitable for reengineering. Different techniques for recovery can be used, each yielding a different result.

Cluster analysis (Feijs et al., 1998; Krikhaar et al., 1999; Lakhotia, 1997) is about finding groups in source code or other data artifacts by computing distances or similarities between elements, these groups are subsequently identified as architectural components. For example, ACDC of Tzerpos and Holt (2000) is a clustering algorithm for C, which primarily uses the number of references made to the combination of a source and header file.

Instead of using computed distances or similarities, concern graphs (Robillard and Murphy, 2002) use a human expert to abstract from structural program dependencies. With automated tool support, like the FEAT tool (Robillard and Murphy, 2007), these kind of abstractions can easily be made without losing traceability to the source code where these abstractions originate from.

Concept analysis (van Deursen and Kuipers, 1999; Eisenbarth et al., 2001; Godin et al., 1995; Snelting and Tip, 1998) is a mathematical approach to building taxonomies, which can be used to recover a partial description of the architecture with respect to a specific set of related features. For example, Snelting and Tip (1998) use concept analysis to optimize inheritance relationships of classes, based on the usage of the methods and properties of these classes.

Design pattern recovery (Keller et al., 1999; Jakobac et al., 2005) tries to recover the design patterns (Gamma et al., 1994) that exist in a given design. The identified relationships between classes are compared to well-known specified design patterns, thereby identifying potential instances of design patterns.

Object identification (Cimitile et al., 1999; van Deursen and Kuipers, 1999; Wiggerts et al., 1997) is the search for candidate classes in a legacy system. The identified classes in turn can be used for reconstructing the design of the system.

Another recovery approach is to look at the dynamic behavior of the software and analyze the run-time events (e.g. method invocations) of a running program. Discotect from Yan et al. (2004) matches method invocations against user defined patterns, which represent architectural constructs to recover a representation of the run-time architecture. X-ray of Mendonça and Kramer (1998) also recovers the run-time architecture, but unlike Discotect uses a static analysis of the source code.

One or more of these recovery approaches can be used to recover the architecture in the steps 2 and 3 of ADDRA. The Symphony process (van Deursen et al., 2004) describes in detail how a number of these techniques could be used to recover certain architectural and detailed design views. However, none of these approaches do not recover software architectures along with their rationale (Keller et al., 1999), which is the focus of ADDRA.

7.3. Rationale management

Knowledge systems (Regli et al., 2000) model decision processes and try to capture the knowledge used in these processes. From a knowledge system perspective, making architectural design decisions is seen as a decision process, which decides how the architecture should change. Capturing knowledge of this decision process provides a basis for the justification, learning, and reuse of this knowledge for further decisions.

Design decision models (Lee, 1991; Potts and Bruns, 1988; Stepenson, 2002) refine this general decision process, as designing a design is much more a goal-oriented process than a general decision process (Lee, 1991). These models explicitly model the goal the design decision process wants to satisfy, as well as the design decisions and their rationale. Design decision models provide a basis to capture, describe, and reason about design decisions made in a design process. One implementation of a design decision model is SEURAT of Burge and Brown (2004), which enables software engineers to use a decision model within the Java IDE Eclipse.

The conceptual model of Section 2.2 is inspired on the conceptual design decision model presented by Stepenson (2002), which in turn is based on an abstraction of IBIS (Kunz and Rittel, 1970; Conklin and Begeman, 1989), DRCS (Lee, 1991), REMAP (Dhar and Jarke, 1988; Ramesh and Dhar, 1992), Redux (Petrie, 1992), and OCS Shell Core (Arango et al., 1991). Although design decision models give some indication of what an architectural *design decision* is and is not, they fail to relate software architectures to architectural design decisions (van der Ven et al., 2006).

8. Future work & conclusions

8.1. Conclusion

Software architecture documentation should not only describe the architecture of a system, but also *why* this architecture looks the way it does. The software architecture design decisions underlying the architecture provide this why. In practice, software architectures are often documented after the fact, i.e. when a system is realized and architectural design decisions have been taken. This paper presented ADDRA, an approach to recover architectural design decisions in an after the fact documentation effort.

To understand what architectural design decisions are, this paper presented a conceptual model. The conceptual model is used as input to the template used in ADDRA to document the recovered architectural design decisions. ADDRA recovers the changes made to the architecture in selected releases and relates these changes back to the architectural design decisions they originated from.

The ADDRA approach has been applied on the Athena case study to recover architectural design decisions. The case study identified three complicating factors for ADDRA: transitions between design decisions, subjectivity of architectural views, and the incompleteness of solutions. In addition, the required effort for recovery is a complicating factor, although recovery tools can lift some of the burden. Nevertheless, these tools cannot replace the tacit knowledge of the architect, which ADDRA uses to recover the rationale of architectural design decisions.

ADDRA is the first approach on recovering architectural design decisions explicitly. Hopefully, with the rising interest in architectural decisions in the software architecture community, ADDRA can be matured and adequate tool created that eases the recovery of architectural design decisions in after the fact documentation efforts.

8.2. Further work & validation

One direction for further work, is to compare and contrast forward, recovery, and hybrid approaches for capturing design decisions. This could provide a good overview of the current state of the art. In addition, issues that not have been addressed so far can be identified, thereby forming a research agenda for the future.

Another direction for further work is the validation of ADDRA. The application of ADDRA in one case study proves that architectural design decision recovery after the fact is to some extent possible. However, it is still unknown how ADDRA performs compared to an ad-hoc approach for recovering architectural design decisions. Furthermore, it remains unknown to what extent ADDRA is capable of recovering all the significant architectural design decisions. Further work on validation is therefore needed to find out the relative performance and recall rate of ADDRA.

We plan to perform such validation by conducting a controlled experiment. In this experiment, several teams consisting of two architects develop in parallel the same application. During development, dedicated scribes capture the architectural design decisions these teams make. After several releases, the architects of each team are asked to document their architectural decisions. Of each team, one architect will use ADDRA, the other one uses his/her own ad-hoc approach. The captured decisions during the development form a baseline to judge the recall rate of both approaches against.

Such an experiment is not without challenges. A first challenge is how to deal with the sensitivity of the experiment for the involved subjects. Only when enough subjects (and therefore teams) are involved in the same project can significant statistical confidence can be reached. A second challenge is that the experiment requires a minimum project size for architectural decisions to play a role. This is turn implies that a realistic duration is needed for development, which complicates replication of the experiment.

A third challenge is to find suitable test subjects. A choice has to be made whether to use student or industrial teams as test subjects. Both types of teams have their benefits and drawbacks. Industrial teams typically do not work on the same (part of a) system. This makes it harder to compare results between teams, as the system can become a discriminating factor in the experiment. To counter this effect, more industrial teams are needed to statistically overcome the influence of the system factor. Student teams on the other hand do not have this problem, as they could work on the same system. However, the results from student teams will be less convincing then those of industrial teams, because they are less experienced and are not exposed to the pressures found in industry. Concluding, in-depth validation of ADDRA is possible and is an interesting challenge for further work.

Acknowledgements

The authors would like to thank Tom Mens and the anonymous reviewers for their comments on earlier versions of this paper.

References

Arango, G., Bruneau, L., Cloarec, J.F., Feroldi, A., 1991. A tool shell for tracking design decisions. IEEE Software 8 (2), 75–83.

Babar, M., Gorton, I., Kitchenham, B., 2006. A framework for supporting architecture knowledge and rationale management. In: Dutoit, A.H., McCall, R., Mistrík, I., Paech, B. (Eds.), Rationale Management in Software Engineering. Springer-Verlag, pp. 237–254, Ch. 11.

Baniassad, E.L.A., Murphy, G.C., Schwanninger, C., 2003. Design pattern rationale graphs: linking design to source. In: Proceedings of the 25th ICSE, pp. 352–362.

Bass, L., Clements, P., Kazman, R., 1998. Software Architecture in Practice. Addison Wesley.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, second ed. Addison Wesley.

Bass, L., Clements, P., Nord, R.L., Stafford, J., 2006. Capturing and using rationale for a software architecture. In: Dutoit, A.H., McCall, R., Mistrik, I., Paech, B. (Eds.), Rationale Management in Software Engineering. Springer-Verlag, pp. 255–272 (Chapter 12).

Boehm, B.W., Horowitz, E., Madachy, R., Reifer, D., Clark, B.K., Steece, B., Brown, A.W., Chulani, S., Abts, C., 2000. Software Cost Estimation with Cocomo II. Prentice-Hall.

Booch, G., RumBaugh, J., Jacobson, I., 1998. The Unified Modeling Language User Guide. Addison Wesley.

Bosch, J., 2004. Software Architecture: The Next Step. In: Software Architecture, First European Workshop (EWSA). LNCS, vol. 3047. Springer, pp. 194–199.

Burge, J.E., Brown, D.C., 2004. An integrated approach for software design checking using design rationale. In: Proceedings of the First International Conference on Design Computing and Cognition (DCC'04), pp. 557–576.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. A System of Patterns. John Wiley & Sons Inc.

Capilla, R., Nava, F., Pérez, S., Dueñas, J.C., 2006. A web-based tool for managing architectural design decisions. SIGSOFT Software Engineering Notes 31 (5), 4.

Cimitile, A., De Lucia, A., Di Lucca, G.A., Fasolino, A.R., 1999. Identifying objects in legacy systems using design metrics. Journal of Systems and Software 44 (3), 199–211.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002. Documenting Software Architectures, Views and Beyond. Addison Wesley.

Conklin, E.J., Yakemovic, K.B., 1991. A process-oriented approach to design rationale. Human–Computer Interaction 6 (3/4).

Conklin, J., Begeman, M.L., 1989. Gibis: a tool for all reasons. Journal of the American Society for Information Science 40 (3), 200–213.

Dhar, V., Jarke, M., 1988. Dependency directed reasoning and learning in systems maintenance support. IEEE Transactions on Software Engineering 14 (2), 211–227.

Eisenbarth, T., Koschke, R., Simon, D., 2001. Aiding program comprehension by static and dynamic feature analysis. In: Proceedings of the International Conference on Software Maintenance (ICSM'01). IEEE Computer Society, pp. 602–611, November.

Falessi, D., Cantone, G., Becker, M., 2006. Documenting design decision rationale to improve individual and team design decision making: an experimental evaluation. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06). ACM Press, New York, NY, USA, pp. 134–143.

Feijs, L., Krikhaar, R., van Ommering, R., 1998. A relational approach to support software architecture analysis. Software – Practice and Experience 28 (4), 371–400.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley.

Godin, R., Mineau, G., Missaoui, R., St-Germain, M., Faraj, N., 1995. Applying concept formation to software reuse. International Journal of Software Engineering and Knowledge Engineering 5 (1), 119–142.

Gueheneuc, Y.-G., 2004. A systematic study of uml class diagram constituents for their abstract and precise recovery. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04). IEEE Computer Society, pp. 265–274.

Harrison, N.B., Avgeriou, P., Zdun, U., 2007. Architecture patterns as mechanisms for capturing architectural decisions. IEEE Software 24 (4), 38–45. ISSN 0704-7459.

Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P., 2005a. Generalizing a model of software architecture design from five industrial approaches. In: Proceedings of the Fifth IEEE/IFIP Working Conference on Software Architecture (WICSA 2005). IEEE Computer Society, pp. 77–88.

Hofmeister, C., Nord, R., Soni, D., 2000. Applied Software Architecture. Addison Wesley.

Hofmeister, C., Nord, R.L., Soni, D., 2005b. Global analysis: moving from software requirements specification to structural views of the software architecture. IEE Proceedings Software (August), 187–197.

IEEE/ANSI, 2000. Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard No. 1471-2000, Product No. SH94869-TBR.

Jakobac, V., Egyed, A., Medvidovic, N., 2005. Improving system understanding via interactive, tailorable, source code analysis. In: Cerioli, M. (Ed.), FASE, Lecture Notes in Computer Science, vol. 3442. Springer, pp. 253–268.

Jansen, A.G.J., 2004. Athena, a large scale programming lab support tool. In: Proceedings of the Dutch National Computer Science Education Congress (NIOC). Uitgeverij Passage, pp. 83–89.

Jansen, A.G.J., Bosch, J., 2004. Evaluation of tool support for architectural evolution. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004). IEEE, pp. 375–378.

Jansen, A.G.J., Bosch, J., 2005. Software architecture as a set of architectural design decisions. In: Proceedings of the Fifth IEEE/IFIP Working Conference on Software Architecture (WICSA 2005), pp. 109–119.

Jansen, A.G.J., van der Ven, J., Avgeriou, P., Hammer, D.K., 2007. Tool support for architectural decisions. In: Proceedings of the Sixth IEEE/IFIP Working Conference on Software Architecture (WICSA 2007).

Jansen, A.G.J., van Gurp, J., Bosch, J., 2003. Reconstructing architectural design decisions: a case study. Tech. Rep. IWI preprint 2003-7-02, Department of Mathematics and Computing Science, University of Groningen, PO Box 800, 9700 AV, The Netherlands.

Keller, R.K., Schauer, R., Robitaille, S., Page, P., 1999. Pattern-based reverse-engineering of design components. In: Proceedings of the 21st International Conference on Software Engineering (ICSE 1999). IEEE Computer Society, pp. 226–235, May.

Krikhaar, R.L., Postma, A., Sellink, A., Stroucken, M., Verhoef, C., 1999. A two-phase process for software architecture improvement. In: Proceedings of the International Conference on Software Maintenance (ICSM99), pp. 371–380.

Kruchten, P., 1995. The 4+1 view model of architecture. IEEE Software 12 (6), 42–50.

Kruchten, P., 2004. An ontology of architectural design decisions in software intensive systems. In: Proceedings of the second Groningen Workshop on Software Variability, pp. 54–61.

Kruchten, P., Lago, P., van Vliet, H., 2006. Building up and reasoning about architectural knowledge. In: Proceedings of the Second International Conference on the Quality of Software Architectures (QoSA 2006).

Kunz, W., Rittel, H.W.J., 1970. Issues as elements of information systems. Tech. Rep. Working Paper 131.

Lago, P., Avgeriou, P., 2006. First workshop on sharing and reusing architectural knowledge. SIGSOFT Software Engineering Notes 31 (5), 32–36.

Lago, P., van Vliet, H., 2005. Explicit assumptions enrich architectural models. In: ICSE'05: Proceedings of the 27th International Conference on Software Engineering. ACM Press, New York, NY, USA, pp. 206–214.

Lakhotia, A., 1997. A unified framework for expressing software subsystem classification techniques. Journal of Systems and Software 36 (3), 211–231.

Lee, J., 1991. Extending the potts and bruns model for recording design rationale. In: Proceedings of the 13th International Conference on Software Engineering (ICSE 1991). IEEE, pp. 114–125.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (1), 70–93.

Mendonça, N.C., Kramer, J., 1998. Developing an approach for the recovery of distributed software architectures. In: Proceedings of the Sixth IEEE International Workshop on Program Comprehension. IEEE, Ischia, Italy, pp. 28–36, The paper describes the initial work on the X-ray architecture recovery approach and tools.

Nonaka, I., 1994. A dynamic theory of organizational knowledge creation. Organization Science 5 (1), 14–37.

Nonaka, I., Takeuchi, H., 1995. The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation. Oxford University Press Inc., USA.

Ohst, D., Welle, M., Kelter, U., 2003. Differences between versions of uml diagrams. In: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering. ACM Press, pp. 227–236.

Oreizy, P., Medvidovic, N., Taylor, R.N., 1998. Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering (ICSE 1998). IEEE, pp. 177–186.

Parnas, D.L., Clements, P.C., 1986. A rational design process: How and why to fake it. IEEE Transactions on Software Engineering 12 (2), 251–257.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17 (4), 40–52.

Petrie, C., 1992. Constrained decision revision. In: Proceedings of the 10th AAAI Conference, pp. 393–400.

Potts, C., Bruns, G., 1988. Recording the reasons for design decisions. In: Proceedings of the 10th International Conference on Software Engineering (ICSE 1988). IEEE, pp. 418–427.

Qin, T., Zhang, L., Zhou, Z., Hao, D., Sun, J., 2003. Discovering use cases from source code using the branch-reserving call graph. In: Proceedings of the 10th Asia-Pacific Software Engineering Conference Software Engineering Conference (APSEC). IEEE Computer Society, Washington, DC, USA, p. 60.

Ramesh, B., Dhar, V., 1992. Supporting systems development by capturing deliberations during requirements engineering. IEEE Transactions on Software Engineering 18 (6), 498–510.

Regli, W., Hu, X., Atwood, M., Sun, W., 2000. A survey of design rationale systems: approaches, representation, capture and retrieval. Engineering with Computers 16 (3-4), 209–235.

Robillard, M.P., Murphy, G.C., 2002. Concern graphs: finding and describing concerns using structural program dependencies. In: ICSE'02: Proceedings of the 24th International Conference on Software Engineering. ACM Press, New York, NY, USA, pp. 406–416.

Robillard, M.P., Murphy, G.C., 2007. Representing concerns in source code. ACM Transactions on Software Engineering and Methodology 16 (1), 3.

Roshandel, R., Hoek, A.V.D., Mikic-Rakic, M., Medvidovic, N., 2004. Mae—a system model and environment for managing architectural evolution. ACM Transactions on Software Engineering and Methodology 13 (2), 240–276.

Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall Inc.

Snelting, G., Tip, F., 1998. Reengineering class hierarchies using concept analysis. In: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering. ACM Press, pp. 99–110.

Stepenson, Z.R., 2002. Change management in families of safety-critical embedded systems. Ph.D. thesis, University of York.

Tang, A., Babar, M.A., Gorton, I., Han, J., 2005. A survey of the use and documentation of architecture design rationale. In: Proceeding of the Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), pp. 89–99.

Tonella, P., Potrich, A., October 2002. Static and dynamic C++ code analysis for the recovery of the object diagram. In: Proceedings of the International Conference on Software Maintenance, pp. 54–63.

Tyree, J., Akerman, A., 2005. Architecture decisions: demystifying architecture. IEEE Software 22 (2), 19–27.

Tzerpos, V., Holt, R.C., 2000. ACDC: an algorithm for comprehension-driven clustering. In: Working Conference on Reverse Engineering (WCRE 2000). IEEE Computer Society, pp. 258–267.

van der Ven, J.S., Jansen, A.G.J., Nijhuis, J.A.G., Bosch, J., 2006. Design decisions: the bridge between rationale and architecture. In: Dutoit, A.H., McCall, R., Mistrik, I., Paech, B. (Eds.), Rationale Management in Software Engineering. Springer-Verlag, pp. 329–348 (Chapter 16).

van Deursen, A., 2002. Software architecture recovery and modelling: [wcre 2001 discussion forum report]. ACM SIGAPP Applied Computing Review 10 (1), 4–7.

van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C., 2004. Symphony: view-driven software architecture reconstruction. In: Proceedings of the 4th IEEE/IFIP Working Conference on Software Architecture (WICSA 2004). IEEE Computer Society, p. 122.

van Deursen, A., Kuipers, T., 1999. Identifying objects using cluster and concept analysis. In: Proceedings of the 21st International Conference on Software Engineering, ICSE-99. ACM, pp. 246–255.

Visconti, M., Cook, C.R., 2004. Assessing the state of software documentation practices. In: Bomarius, F., Iida, H. (Eds.), PROFES, Lecture Notes in Computer Science, vol. 3009. Springer, pp. 485–496.

Webster, 2006. http://www.webster.com.

Wiggerts, T., Bosma, H., Fielt, E., 1997. Scenarios for the identification of objects in legacy systems. In: Fourth Working Conference on Reverse Engineering (WCRE'97). IEEE Computer Society, pp. 24–32.

Yan, H., Garlan, D., Schmerl, B.R., Aldrich, J., Kazman, R., 2004. Discotect: a system for discovering architectures from running systems. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004). IEEE Computer Society, pp. 470–479.

Anton Jansen is a post doctoral research in the Griffin project at the University of Groningen, The Netherlands. He obtained a master of sciene degree in computer science at the University of Groningen in september 2002. From october 2002 - october 2006 he was a PhD associate at the same university under the supervision of Jan Bosch. His research is focussed on architectural design decisions, architectural composition, and design erosion. In his free time, he enjoys politics, computer games, and riding his bicycle.

Jan Bosch is VP, Engineering Process at Intuit Inc. Earlier, he was head of the Software and Application Technologies Laboratory at Nokia Research Center, Finland. Before joining Nokia, he headed the software engineering research group at the University of Groningen, The Netherlands, where he holds a professorship in software engineering. He received a MSc degree from the University of Twente, The Netherlands, and a PhD degree from Lund University, Sweden. His research activities include software architecture design, software product families, software variability management and component-oriented programming. He is the author of a book ''Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach'' published by Pearson Education (Addison-Wesley & ACM Press), (co-)editor of several books and volumes in, among others, the Springer LNCS series and (co-)author of a significant number of research articles. He has been guest editor for journal issues, chaired several conferences as general and program chair, served on many program committees and organized numerous workshops. Finally, he is and has been a member of the steering groups of the GCSE and WICSA conferences. When not working, Jan divides his time between his family, a spouse and three young boys, and sports, preferably long distance running and swimming.

Paris Avgeriou is an Assistant Professor, at the Department of Mathematics and Computing Science, University of Groningen, the Netherlands. He has worked as a Senior Researcher at the Fraunhofer IPSI, Germany, and the Software Engineering Competence Center, University of Luxembourg, as a visiting Lecturer at the Department of Computer Science, University of Cyprus, and as a research and teaching assistant at the National Technical University of Athens. He has participated in a number of European Union research projects, and co-organized international workshops in conferences such as ICSE, ECOOP, ICSR, UML, ACM SAC. He is a member of IEEE, ERCIM, Hillside Europe and acts as a PC member and reviewer for several conferences and journals. His research interests concern the area of software engineering and particularly software architecture, with a strong emphasis on modeling, evolution and patterns.