2011 Ninth Working IEEE/IFIP Conference on Software Architecture

Automatic Tracing of Decisions to Architecture and Implementation

Georg Buchgeher
Software Competence Center Hagenberg
Hagenberg, Austria
georg.buchgeher@scch.at

Rainer Weinreich
Johannes Kepler University Linz
Linz, Austria
rainer.weinreich@jku.at

***Abstract*—Traceability requires capturing the relations between software artifacts like requirements, architecture and implementation explicitly. Manual discovery and recovery of tracing information by studying documents, architecture documentation and implementation is time-intensive, costly, and may miss important information not found in the analyzed artifacts. Approaches for explicitly capturing traces exist, but either require manual capturing or lack comprehensive tracing to both architecture and implementation. In this paper we present an approach for (semi)automatically capturing traceability relationships from requirements and design decisions to architecture and implementation. Traces are captured in a non-intrusive way during architecture design and implementation. The captured traces are integrated with a semi-formally defined architecture description model and serve as the basis for different kinds of architecture-related activities.**

***Keywords*-Traceability; Requirements; Software Architecture; Design Decisions; Architecture Knowledge Management**

I. Introduction

Software development is a decision making process that involves multiple stakeholders. Customers decide about a system's functional and non-functional requirements, software architects make decisions about how these requirements are addressed in a system's architecture design and developers decide how the system's requirements and its architecture are addressed in detailed design and finally realized in the system implementation. Decisions exist at different levels of abstraction. Jansen classifies decisions using a model called the *funnel of decision making* [1]. He distinguishes between decisions of the problem space and decisions of the solution space and organizes them at different levels of abstraction. Decisions of the problem space are requirements, whereas decisions of the solution space are design decisions. Decisions are made at the architectural, the detailed design, and the implementation level. Decisions made at the architectural level are called architectural design decisions (ADDs) [2] and have gained in importance with the rise of software architecture knowledge management [3]. ADDs provide an additional view on the architecture that captures the knowledge and rationale behind architectural solutions [4]. ADDs also act as a bridge between requirements and architectural solution structures [5].

Decisions made throughout the development process are related to each other. Typically decisions made at higher abstraction levels narrow the scope of decision making at lower abstraction levels [1]. Decisions are not only related to other decisions but to resulting development artifacts as well. Lago and Avgeriou [5] note that the relationships between requirements, ADDs and architectural solutions are not 1-to-1 but N-to-M. They also state that the traceability between requirements, ADDs and architectural structures would be "highly desirable" but difficult to achieve "due to the lack of knowledge on how to explicitly and unambiguously relate the sets, as well as the effort to do so" [5].

Understanding the interrelationships of requirements, ADDs, architectural structures and other development artifacts is essential for many activities in the development process like architecture evaluation and analysis [6] and system evolution and maintenance [7]. The aim of architecture evaluation is to check whether a defined architecture addresses the system requirements. Captured traces support evaluation processes by making reasoning paths between analysis models, including requirements, design decisions, and architectural structures, explicit [6]. For example, architects can use this information for justifying decisions and for reminding themselves of the rationale of their decisions [8]. The same applies when analyzing the system implementation for conformance to requirements and design decisions. During system maintenance and evolution a large amount of time is spent for program comprehension [9][10]. Developers need to understand a system, before they can make modifications, in order not to break functional and non-functional requirements and not to violate the system's architecture. Lago et al. [11] describe traceability as "the ability to describe and follow the life of a software artifact" and as "a means for modeling the relations between software artifacts in an explicit way". Traceability closes the gap between requirements, architecture and implementation and facilitates program comprehension and impact analysis. These analytical activities are not only important during the system maintenance phase but also during the development process. Especially with the rise of agile and lean development, in which requirements, architecture and implementation are developed in small increments and concurrently, making changes to a system has become an integral part of

978-0-7695-4351-2/11 $26.00 © 2011 IEEE
DOI 10.1109/WICSA.2011.16

development processes.

Tracing relationships are often not made explicit and remain tacit knowledge that easily gets lost over time. Relationships between development artifacts then have to be (re)discovered manually by studying architecture documentation and by analyzing the system implementation, which is time consuming and thus expensive [11]. Missing, incomplete and inconsistent requirement documents and architecture documentation as well as poorly documented code can make this a difficult task. Hunt and Thomas describe this process as software archaeology [12].

Approaches for tracing exist in research as well as in practice (see Section IV). However, they still show some deficiencies. Most solutions either only support tracing between decisions and architecture or between decisions and implementation but not both. Further in many approaches - especially for tracing between decisions and architecture - traces have to be defined manually, which is not only time intensive and tedious but also increases the risk that traces are captured only partially or that they are not captured at all.

In this paper we present an approach for (semi)automatically capturing traces from decisions to architecture and implementation. We create tracing links during architecture design and implementation processes by analyzing modifications of development artifacts. The captured traces are added to a semi-formally defined architecture model. Our approach supports fine-grained tracing to single elements of an architecture model and even down to the method level of the system implementation. We have created a prototype implementation for validating our approach that has been integrated with the LISA toolkit for architecture-centric software development [13]. The presented concepts can be applied to any approach and toolset based on a semi-formal architecture model (like UML and ADLs).

The work described in this paper is an extension of the work presented in [14], where we described an approach for integrating decisions (requirements and design decisions) with an architecture representation. Initially traceability links had to be defined manually in our approach. Although creation was already supported using various techniques [14] we discovered that the definition of traceability links was still too tedious when validating the approach in different projects. Developers either only defined few traceability links or deferred the definition of traceability links to phases after or between design and implementation activities, which also resulted in only partially captured traceability information.

The remainder of this paper is organized as follows: In the next section we describe our approach for (semi)automatically capturing traces between decisions, architecture and implementation. In Section III we demonstrate our approach using a simple example scenario. Section IV discusses related work. In Section V we describe the steps we took to validate our approach. We conclude with a summary and an outlook on current and future work in Section VI.

II. Approach

We begin by discussing previous work that is foundation of our work. We then describe the basic idea of our approach and finally discuss the technical realization of our work.

A. Previous Work

The work described in this paper is part of the SAE project for architecture-centric software development [15]. The project is centered around an approach named LISA (**L**anguage for **I**ntegrated **S**oftware **A**rchitecture) consisting of two main elements: an architecture description metamodel (*LISA Model*) and a set of integrated tools (*LISA Toolkit*) that operate on this model and support architecture-related activities like architecture design, analysis and implementation.

Fig. 1 provides a high level overview of the LISA Model. As shown in the figure, the model consists of sub models for describing decisions, solution structures, and activities. In the decision sub model we distinguish between requirements and design decisions; both can be architecturally significant or not. Due to the similarity between requirements and design decisions [16] requirements and design decisions are described nearly identically and the kind of decision can be changed after capturing a decision. The structures sub models contain elements for describing architectural structures. This includes static & dynamic component structures (component definitions, instances and configurations) and structures of object-oriented programming languages (like namespaces, classes, and their relationships). The activities sub model contains events for describing design and implementation activities. Events represent modifications of a LISA-based architecture model including decisions and architectural structures. They introduce a chronological dimension and also contain information about the user who

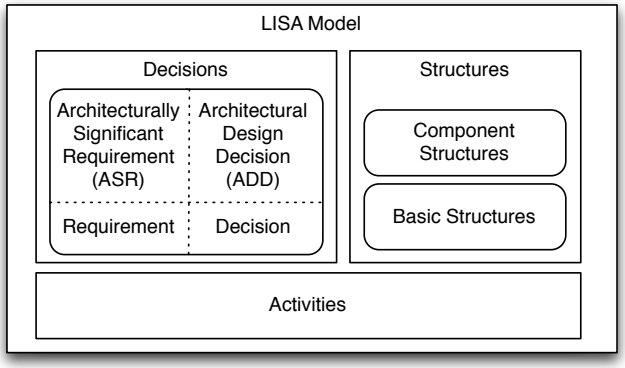

Figure 1. LISA Model Overview

is responsible for the performed change. In sum, the LISA model supports description of decisions, architectural and implementation structures, and architectural activities within a single model. A single model approach facilitates the description of traces between the elements of the model.

B. Basic Idea

The basic idea underlying our approach is quite simple. Developers implicitly map decisions to architecture and implementation elements as part of their daily work. For example, software architects take requirements as input and address them in architecture design by making design decisions and creating architectural solution structures. Developers take requirements and the software architecture as input, perform detailed design and create the system implementation. The work of architects and developers results in manipulations (additions, deletions and modifications) of architectural structures and development artifacts. In our approach we simply log these modifications and extract tracing targets by analyzing these modifications.

The approach basically consists of three phases depicted in Fig. 2. First the user sets the context of his work by selecting the decision(s) he is currently working on. We name these decision(s) *active decision(s)*. In the second phase developers performs their daily work, i.e., they perform architecture design and implementation tasks. During these tasks we create and log modification events. A modification event consists of a short description of the performed modification and a set of architecture and implementation elements that have been manipulated during the performed activity. These elements are the potential targets for traces. Finally, when the developer has finished working on a decision, he needs to review the captured tracing targets. The review process itself consists of several steps:

1) *Selection of architecture and implementation elements:* In this step the user selects architecture and implementation elements from the captured tracing targets that should be linked to the active decision. This step gives the user the opportunity to remove incorrect and unnecessary traces.
2) *Selection of obsolete elements:* Previously defined traces may become obsolete due to modifications of the system. In this step the user can review and select previously defined traces that have become obsolete. Selected obsolete traces will be deleted.
3) *Definition of relationships between decisions:* In this step the user can define relationships between the active decision and other decisions. Potential candidates for related decisions are selected from decisions that are already linked to the elements selected in step 1. Further candidates are decisions that have been added and modified while working on the active decision. For instance, this can be the case when a coarse grained decision is refined to finer-grained decisions.

At the end of the review process the traces are created and added to the architecture model. In Section III the whole process is demonstrated using a simple example scenario.

C. Technical Realization

In the following we explain the automatic detection of tracing targets and the integration of traces in the LISA model from a technical perspective.

1) Detection of Tracing Targets: The capturing of tracing targets as well as the management of active decisions is handled by an event-logging component (see Fig. 3). Potential tracing targets are captured while the users are concerned with architecture design and implementation. The LISA toolkit creates modification events in response to modifications of architecture and implementation, which are collected by the logging component and assigned to all active decisions. This process is performed automatically in the background without requiring any user interaction. When creating modification events we distinguish between modifications of the architecture model and modifications of the system implementation. Both kinds of events are created differently.

Logging Modifications of the Architecture Model: The LISA toolkit supports logging of architecture modification events by default. Model changes are performed by op-

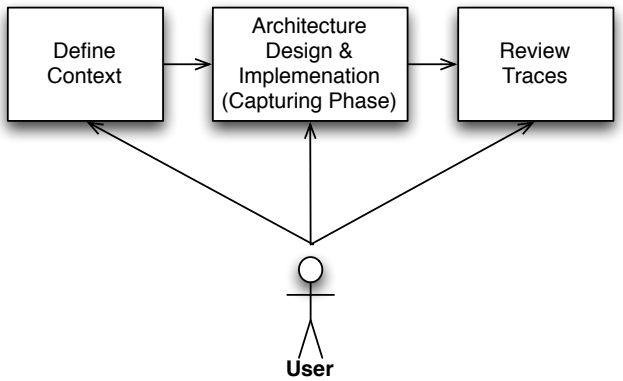

Figure 2. Workflow for Capturing Traces

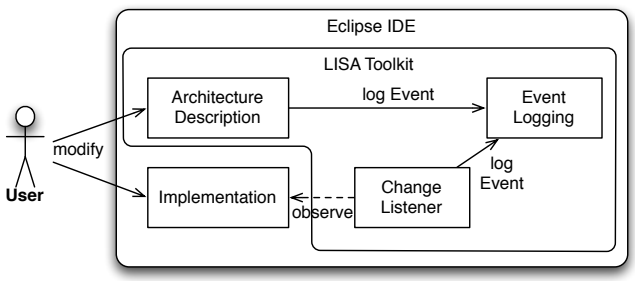

Figure 3. Logging of Tracing Targets

erations that encapsulate all information for performing the requested model change. These operations also provide information for creating a modification event, like an event description and the architecture elements modified by the operation. Whenever such an operation is executed, the toolkit automatically creates an event and sends the event to the logging component.

Logging Modifications of the System Implementation: Logging changes of the system implementation is more complex. A listener component observes the system implementation for changes, analyzes these changes, and creates corresponding modification events. For detecting modifications of the system implementation we use facilities provided by the Eclipse Java Development Tools (JDT)[1] . The JDT provide the concept of an *element changed listener* that is notified whenever elements of the implementation are modified. Each change notification contains a description of the performed change, from which we extract the required information for creating a modification event. Using an element change listener, we can detect the following kinds of modification events:

- Creation, deletion and renaming of classes
- Adding, removing and renaming methods
- Changing method signatures (parameters, return types and visibility)
- Changing the implementation of method bodies

After having identified the affected elements (classes and methods) as well as the kind of modification, the corresponding elements are looked up in the architecture model. Both classes and methods are part of a LISA-based architecture model. After the corresponding architectural elements have been found, we create a modification event that is then being sent to the logging component. As part of the analysis process of implementation changes we also update elements of the architecture model like classes and methods in order to keep the architecture model up-to-date and consistent with the system implementation. We should note that the currently implemented solution works only for Java-based software systems since we use the Eclipse JDT as mentioned above.

2) Definition of Traces: A *DecisionTrace* describes a relationship between a decision and an architecture element in the LISA meta-model (see Fig. 4). An architecture element can be any element of the architecture model. Decisions themselves are architecture elements and thus can be the target of a trace. Traces between decisions and architecture elements are navigable in both ways and thus permit forward and backward tracing between decisions and architecture elements. Each decision trace has a kind attribute for specifying the type of the relationships. Currently we support relationship kinds that have been proposed by Kruchten in [7].

[1]http://www.eclipse.org/jdt/

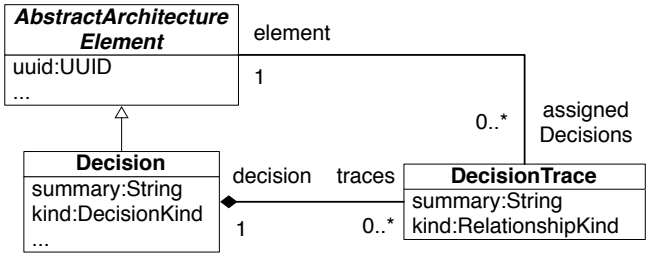

Figure 4. Decision Traces Model

Each architecture element has an automatically generated unique identifier (UUID). UUIDs provide two main advantages when compared to logical names: First, it is a common concept applicable for any architecture element. Using UUIDs also elements without identity-giving names can be referenced. For instance, a trace can reference a connection between two components. Second, traces are not broken and need not to be updated when the name of an element is changed. This makes traces resistant to refactoring, which is important in the light of system evolution.

A more detailed description of our decision model and how we integrate traces in the LISA model can be found in [14]. There are, however, some noteworthy changes to this previous work. We previously used the more generic term *issue* instead of decision. Since this term is usually associated with some serious problem in a software system, we changed it to decision, which is also more in line with other work in this area. Also, relationships between decisions and between decisions and other architectural elements were previously separated and have been unified in the current model.

III. Example

We illustrate our approach using a simple scenario for capturing traces during the development of a simple course management system where teachers can create and manage courses and students can enroll for courses. In the described scenario, a developer is working on the course management functionality. He will design and implement a component for adding a new course.

A. Context Definition

First, the developer needs to define the context of his work by selecting the decision(s) he is currently working on. In Fig. 5 the developer selects the decision *Manage Courses* from a dialog showing all previously captured decisions. If necessary, new decisions can also be created using this dialog.

Fig. 6 shows the *Decision Context* view, which gives an overview of all currently active decisions. It is possible to work on multiple active decisions at the same time. This can be the case when a coarse grained decision is refined into multiple fine-grained decisions. Active decisions can

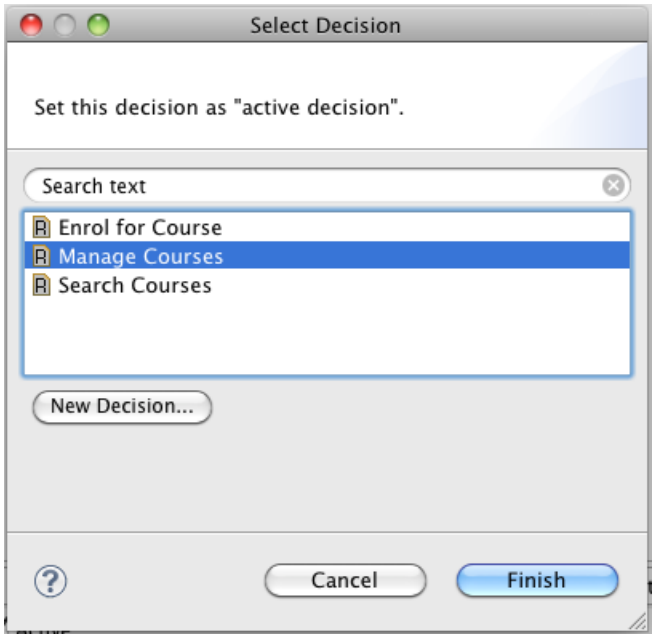

Figure 5. Selection of Active Decision(s)

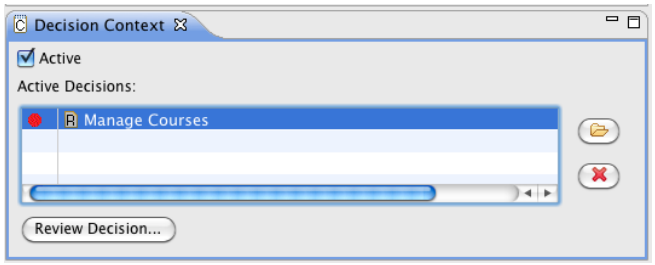

Figure 6. Decision Context View

be suspended, which also suspends linking of modification events with this decision. This is useful when the user decides to interrupt work on one decision to temporarily work on another. After setting the context and setting one ore more active decisions, developers can begin with architecture design and implementation activities.

B. Architecture Design

As part of the architecture design process the user makes design decisions and defines architectural solution structures. In our example the developer makes two design decisions. He decides to create a separate component for creating new courses and he decides to use Enterprise Java Beans (EJB) as component technology. The defined architecture solution structures are depicted in Fig. 7. The developer has created a component (*CreateCourseComponent*) providing a single service (*CreateCourseService*), whose interface is described in a contract (*CreateCourseContract*).

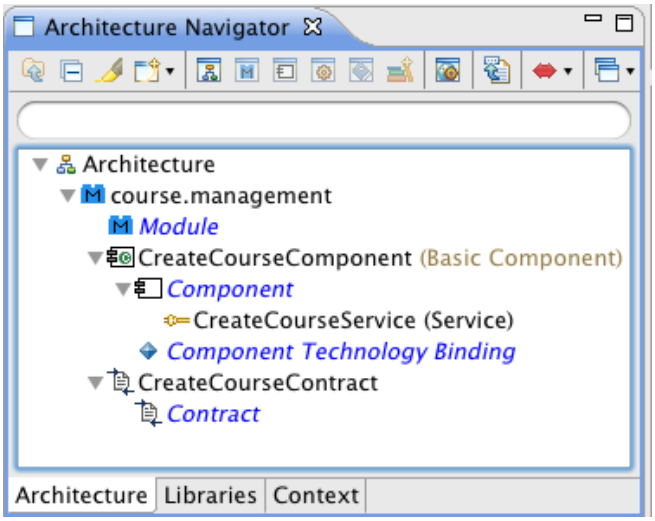

Figure 7. Result of the Architecture Design Process

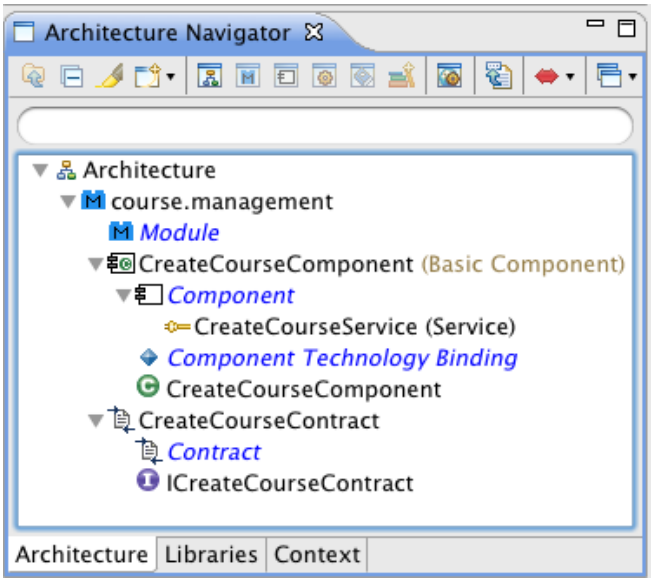

Figure 8. Architecture Model with Implementation Elements

C. Architecture Implementation

In the next step the developer creates classes and interfaces for implementing the defined component (see Fig. 8). Architecture design and implementation can be performed incrementally and intertwined.

D. Background Processing

During the architecture design and implementation process modification events are created and logged. Fig. 9 depicts a screenshot of the *Event Log* view. As shown in the figure every modification is described as log entry. Each log entry contains a timestamp, a description and the name of the user who caused the event. Further architecture and

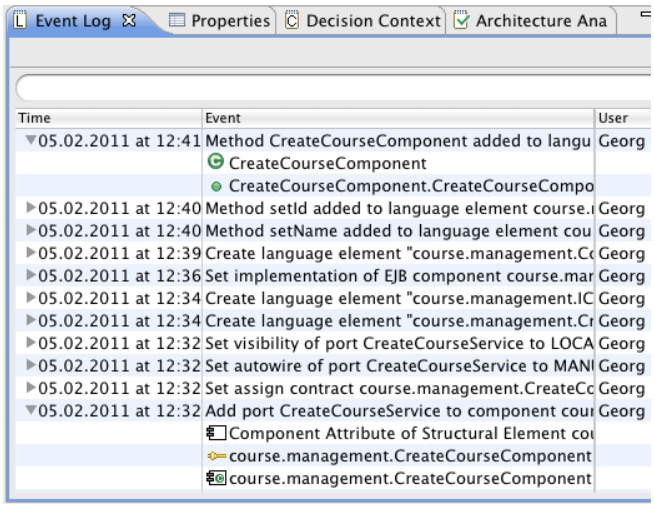

Figure 9. Recorded Modification Events

implementation elements that are affected by an event are shown as child elements of the event.

E. Review Process

After the developer has finished his work on an active decision, he has to review the captured tracing targets. A wizard guides the user through the steps of the review process that have been described in Section 2.B. Fig. 10 shows the first page of the review dialog. This page lists all architecture elements that have been identified as potential tracing targets. On this page the user can remove incorrect and unnecessary tracing targets. As shown in the figure the list contains all elements that have been created and modified during architecture design and implementation. We omit the page for selecting obsolete traces because previously no traces have been defined.

Fig. 11 shows the wizard page for defining relationships between decisions. As described above during architecture design, two design decisions have been captured. These decisions have also been identified as potential tracing targets. In our example the user decides that one decision is related to the active decision. He checks the related decision and defines the relationship kind.

F. Visualization

After finishing the review wizard, the traces are created and added as part of the architecture model. Traces are visualized in architecture diagrams and in source code editors as well. As shown in Fig. 12 the elements of architecture design and implementation that have been defined in steps B and C are now "annotated" with note icons and numbers. The icon indicates an assigned decision and the numbers indicate the number of assigned decisions. The first number shows the number of decisions assigned directly to an element; the second number shows the number of decisions assigned to

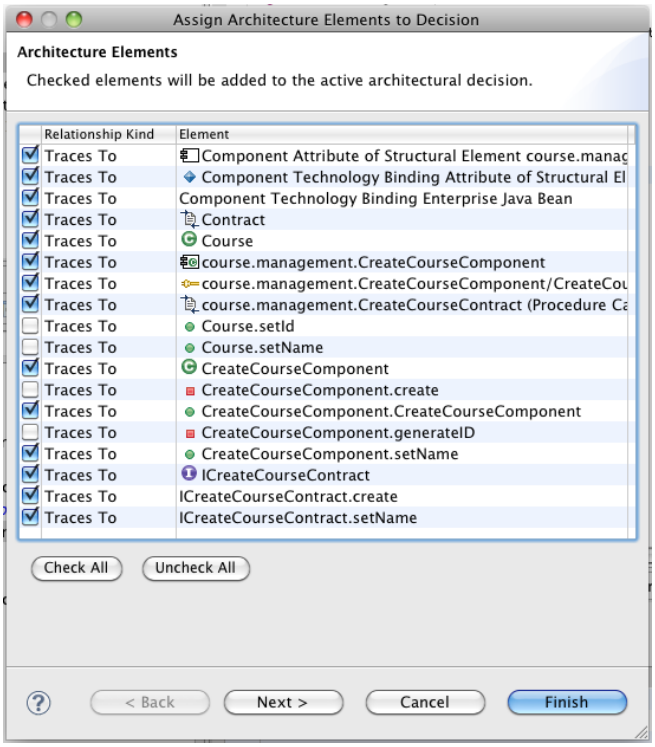

Figure 10. Selected Tracing Targets

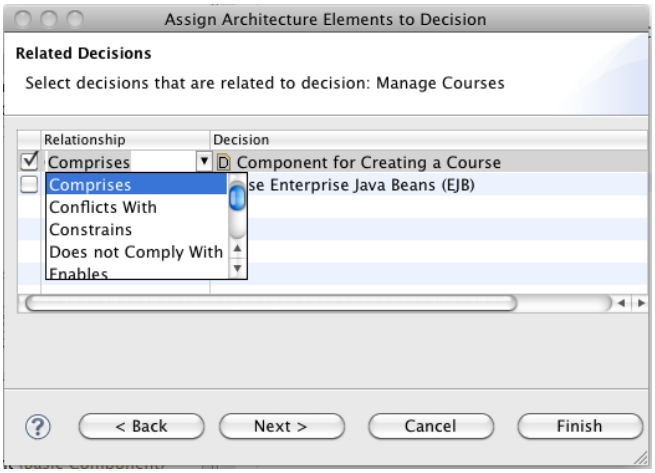

Figure 11. Definition of Relationships between Decisions

child elements. Descriptions of assigned decisions are shown via tooltips. It is also possible to open a decision editor for showing details and manipulating decisions.

Assigned decisions are also shown in implementation artifacts. Fig. 13 shows the implementation of the *ICreateCourseContract* interface. As shown in the figure the interface (see 1 in Fig. 13) as well as single methods (see 2 in Fig. 13) are annotated with decisions. The summary of decisions is displayed on demand as tooltip. A decision can be opened in the decision editor by clicking on the

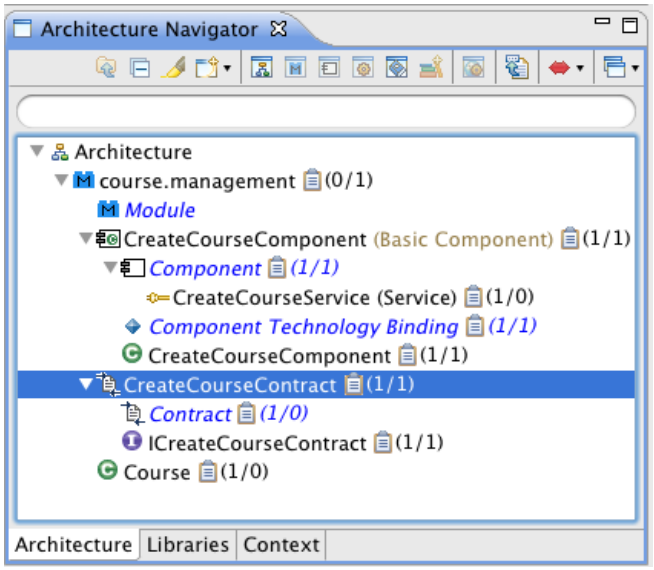

Figure 12. Architecture Model with Annotated Traces

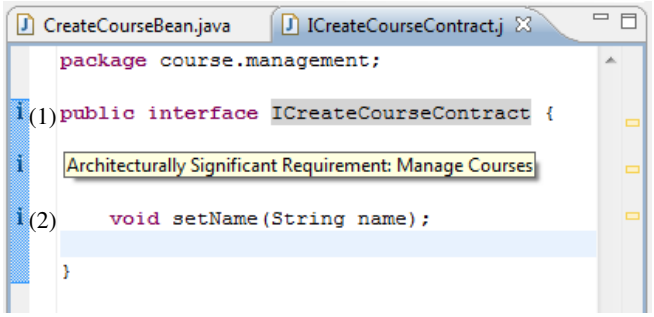

Figure 13. Code Editor With Tracing Annotations

annotation.

IV. Related Work

A number of approaches for tracing exist in research as well as in practice. In particular, most architecture knowledge management tools provide support for traceability [17]. We focus in specifically on related work in two areas: automatic tracing and tracing to both architecture and implementation.

Eclipse Mylyn[2] supports tracing between tasks (assigned pieces of work) and implementation artifacts. Tasks are usually stored in issue management systems like Bugzilla and Jira. Mylyn provides a *task context* model, which aims at "filtering and ranking information presented by the development environment" [18]. The task context is used for focusing on implementation artifacts that are likely to be relevant for the currently performed task based on programmer's activity and existing structural relationships between program artifacts [18]. Similar to our approach, Mylyn users have to define the context of their work. Within the defined context relationships between tasks and implementation artifacts are created by monitoring the developers activities (this part of Mylyn has influenced our work). Contrary to Mylyn we currently do not perform structural analysis to detect additional tracing targets, except for detecting relationships between decisions. Both approaches are integrated with the Eclipse IDE. IDE integration is a requirement for fine-grained monitoring of implementation activities. The central difference between the two approaches is that we support comprehensive tracing of decisions to architecture and implementation, while Mylyn only permits tracing between tasks (which can be decisions) and implementation artifacts. Further, Mylyn does not support the detection of relationships between tasks. These relationships have to be defined manually. We support the detection of relationships between decisions as part of the review process.

The Rational Team Concert (also known as Jazz)[3] supports tracing between work items (like tasks, defects and user stories) and other development artifacts. Artifacts can be added to work items as attachments. Further Jazz automatically detects links to artifacts like classes or other work items by textual analysis and converts them into hyperlinks. While artifact traceability has been an important aspect for the development of Jazz [19], it provides no means for tracing work items (that can be decisions) to an architecture description. Jazz is a powerful platform for collaborative development that supports many development tasks like project planning and monitoring, implementation, testing, build and configuration management, but it provides no support for architecture design and description and consequently no means for tracing to architecture.

SEURAT (Software Engineering Using RATionale system) [20] is an Eclipse plug-in for rationale management. It supports tracing from requirements to (architectural) design decisions and code. SEURAT provides no support for modeling architectural structures. Traces to implementation artifacts can be defined to classes, methods and instance variables. However, these traces have to be defined manually. SEURAT focuses on rationale management during system maintenance, while our approach aims at supporting any activity where relationships between decisions, architecture and implementation need to be analyzed.

AREL (Architecture Rationale and Elements Linkage) [21] is an approach for capturing design rationale and linking this rationale to the design outcome. The approach consists of an architecture model for design rationale and a corresponding toolkit. AREL supports the description of design concerns (which can be requirements), design decisions and design outcomes that can be linked to each other. AREL extends UML. Similar to our approach AREL supports describing requirements, design decisions as well as

[2]http://www.eclipse.org/mylyn/

[3]http://jazz.net/

architecture and implementation structures within one single model. The AREL toolkit supports automatic forward and backward tracing by generating traceability diagrams. The created diagrams can be used for impact analysis as well as root-cause analysis. AREL requires that traces are defined manually, while we detect traces automatically.

V. Validation

In order to validate our approach we followed an action research approach [22] with multiple iterations in order to get early feedback for improving our work. We used the approach in a student project (Project A). We asked developers of a medical information system to use our approach in their project (Project C). And we also used our approach for the development of the LISA toolkit itself (Project B).

A. Quantitative Analysis

Table 1 shows some metrics taken from the three projects. The table shows for each project the overall number of captured decisions with traces, the number of captured traces, and the average number of traces per decision. In the case of the student project 42 decisions with 255 traces have been captured over a period of three months, which makes an average of 6.0 traces per decision. In the case of the development of the LISA toolkit we have captured traces over a period of 6 months. During this time 183 decisions have been captured. For 146 decisions we defined traces to architecture and implementation elements. In sum, we captured 885 traces, which makes an average of 6.0 traces per decision. In both cases traces have been created both manually and automatically. In a previous industrial project (Project C) traces still had to be defined manually, since support for automatic capturing was not available at the time. In this project 41 decisions and 138 traces have been captured over a period of 6 months, which results to 3.4 traces per decisions.

Automatic capturing has not always been activated by all members of the projects with automatic tracing support (A and B), which resulted in missing data in these projects. For this reason, we have added functionality for extracting tracing information in retrospect from logged design and implementation activities in the most recent version of our approach. Despite this problem, the collected data shows that the average number of captured traces per decision has been higher in projects with support for automatic capturing. While this is an indication for the usefulness of our approach, it is no empirical evidence, as it would result from a controlled experiment. For this reason, we have also performed a qualitative analysis of our approach.

Table I
Captured Traces per Decision

Project	Decisions With Traces	No. of Traces	Average No. of Traces per Decision
A	42	255	6.0
B	146	885	6.0
C	41	138	3.4

B. Qualitative Analysis

Many features of our approach have been added and modified in response to discussions and requests of developers using the approach in their projects. Initially we completely omitted a review phase for filtering incorrect and unnecessary traces. This turned out to be a drawback and the described review process was introduced and subsequently extended. Also the possibility to remove previously defined traces was a request from developers. Initially we supported only a single *active decision* and provided no means for switching between *active decisions* by suspending single decisions.

We further made the following observations:

Manually defined traces: Not all traces can be defined automatically using our approach, especially if decisions are captured some time after architecture and implementation structures have already been defined. Also, reviewing automatically captured traces requires human intervention.

Tracing down to the method level: Defining traces down to the method level of classes was considered useful only when a class addresses multiple decisions. If a class only addresses a single decision, it is sufficient to define a trace to the class. The decision whether to trace only to the class or to its methods has to be made by the developer during the review phase.

Broken traces: One problem that emerged is that traces may become inconsistent. Changes to the architecture are no problem since they are only performed through the toolkit where this problem can be prevented. Traces to the implementation are resistant to refactorings of the system implementation like renaming classes and methods and changing method signatures. However, traces can be broken if the implementation is changed without using the LISA toolkit.

Undetected traces: Currently we only detect traces to architecture implementation elements that are created and modified as part of the architecture design and implementation phase. References to existing elements that are simply (re)used are not detected because these elements are not manipulated. For instance, if a class A defines a reference to an already existing class B, class B is not detected as a tracing target. This problem can be addressed by performing additional structural analysis of architecture descriptions and system implementation and will be subject of future work.

Completeness of the architecture model: Traces can only be captured if both architecture and implementation elements are contained in the architecture model. In the case of implementation elements we check automatically that

every class of the system implementation is also contained in the architecture model. Missing classes can be added to the architecture model via reverse engineering. Method descriptions are always added automatically.

VI. Conclusion

In this paper we have presented an approach for automatic tracing of decisions to architecture and implementation. Instead of manually defining such relationships we automatically identify potential tracing relationships by observing developers during architecture design and implementation processes. In a review phase developers can edit and confirm detected traces. Captured traces are added to a semi-formally defined architecture description model and connect requirements, architecture and implementation. Our approach integrates seamlessly with architecture design and implementation processes. The captured traces serve as a foundation for analytical activities like architecture analysis, program comprehension and impact analysis during system development, evolution, and maintenance.

From applying our approach in several projects we can draw the following conclusions: (1) We are currently not able to proof that automatic tracing significantly enhances the number of captured decisions and traces. While experience indicates that this might be the case, we have simply not enough data to support such a claim. (2) The review process is essential. It is necessary for constraining both the depth (class-level vs. method-level) and the width of potential tracing targets. It also shifts the burden from identifying and selecting tracing targets to excluding targets from a proposed target set, and it helps to support traces between decisions themselves by proposing related decision candidates. (3) Not all capturing can be performed automatically. Manual capturing needs to be supported as well. (4) Automatic capturing of traces also facilitates capturing of decisions themselves since developers need to select/define decisions they are working on, when capturing is activated. (5) Finally, architecture modeling and implementation need to integrate seamlessly to support comprehensive tracing as proposed in this paper.

Currently, we are working on the automatic detecting of potential conflicts and tradeoffs between traces and on supporting the evolution of traces throughout the development process.

References

[1] A. Jansen, "Architectural design decisions," Ph.D. dissertation, 2008.

[2] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 109–120.

[3] M. A. Babar, T. Dingsøyr, P. Lago, and H. van Vliet, Eds., *Software Architecture Knowledge Management: Theory and Practice*. Springer, 8 2009.

[4] H. van Vliet, "Software architecture knowledge management," *Software Engineering Conference, Australian*, vol. 0, pp. 24–31, 2008.

[5] P. Lago and P. Avgeriou, "First workshop on sharing and reusing architectural knowledge," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 5, pp. 32–36, 2006.

[6] H. Vliet, P. Avgeriou, R. C. Boer, V. Clerc, R. Farenhorst, A. Jansen, and P. Lago, "The griffin project: Lessons learned," in *Software Architecture Knowledge Management*, M. Ali Babar, T. Dingsøyr, P. Lago, and H. Vliet, Eds. Springer Berlin Heidelberg, 2009, pp. 137–154.

[7] P. Kruchten, "An ontology of architectural design decisions in software intensive systems," in *2nd Groningen Workshop on Software Variability*, 2004, pp. 54–61.

[8] P. Clements, *Documenting software architectures : views and beyond (2nd Edition)*. Upper Saddle River, NJ: Addison-Wesley, 2010.

[9] M. Shaw, R. Deline, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 314–335, 4 1995.

[10] D. Garlan, "Formal modeling and analysis of software architecture: Components, connectors, and events," in *Formal Methods for Software Architectures, Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003, Advanced Lectures*, ser. Lecture Notes in Computer Science, M. Bernardo and P. Inverardi, Eds. Springer, 2003, vol. 2804, pp. 1–24.

[11] P. Lago, H. Muccini, and H. van Vliet, "A scoped approach to traceability management," *Journal of Systems and Software*, vol. 82, no. 1, pp. 168–182, 2009.

[12] A. Hunt and D. Thomas, "Software archaeology," *IEEE Software*, vol. 19, no. 2, pp. 20–22, 2002.

[13] G. Buchgeher and R. Weinreich, "Tool demonstration: a toolkit for architecture-centric software development," in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, ser. PPPJ '10. New York, NY, USA: ACM, 2010, pp. 158–161.

[14] R. Weinreich and G. Buchgeher, "Integrating requirements and design decisions in architecture representation," in *Proceedings of the 4th European conference on Software architecture*, ser. ECSA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 86–101.

[15] G. Buchgeher and R. Weinreich, "Software architecture engineering," in *Hagenberg Research*, B. Buchberger et al., Ed. Springer, 2009, pp. 200–214.

[16] R. de Boer and H. van Vliet, "On the similarity between requirements and architecture," *Journal of Systems and Software*, vol. 82, no. 2009, pp. 544–550, 11 2008.

[17] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar, "A comparative study of architecture knowledge management tools," *Journal of Systems and Software*, 9 2009.

[18] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 1–11.

[19] R. Frost, "Jazz and the eclipse way of collaboration," *Software, IEEE*, vol. 24, no. 6, pp. 114–117, 11 2007.

[20] J. E. Burge and D. C. Brown, "Software engineering using rationale," *Journal of Systems and Software*, vol. 81, no. 3, pp. 395 – 413, 2007.

[21] A. Tang, Y. Jin, and J. Han, "A rationale-based architecture model for design traceability and reasoning," *Journal of Systems and Software*, vol. 80, no. 6, pp. 918–934, 2007.

[22] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, "Action research," *Commun. ACM*, vol. 42, pp. 94–97, 1 1999.