Tool support for Architectural Decisions

Anton Jansen, Jan van der Ven, Paris Avgeriou, Dieter K. Hammer
University of Groningen
Department of Mathematics and Computing Science
PO Box 800, 9700AV Groningen, The Netherlands
[anton|salvador|paris|dieter]@cs.rug.nl

Abstract

In contrast to software architecture models, architectural decisions are often not explicitly documented, and therefore eventually lost. This contributes to major problems such as high-cost system evolution, stakeholders miscommunication, and limited reusability of core system assets. An approach is outlined that systematically and semi-automatically documents architectural decisions and allows them to be effectively shared by the stakeholders. A first attempt is presented that partially implements the approach by binding architectural decisions, models and the system implementation. The approach is demonstrated with an example demonstrating its usefulness with regards to some industrial use cases.

1. Introduction

Current research trends in software architecture focus on the treatment of architectural decisions [16, 26, 18] as first-class entities and their explicit representation in the architectural documentation. From this point of view, a software system's architecture is no longer perceived as interacting components and connectors, but rather as a set of architectural decisions [13]. This paradigm shift has been initiated in order to alleviate a major shortcoming in the field of software architecture: *Architectural Knowledge Vaporization* [6, 30].

Architectural decisions are one of the most significant forms of architectural knowledge [30]. Consequently, architectural knowledge vaporizes because most of the architectural decisions are not documented in the architectural document and cannot be explicitly derived from the architectural models. They merely exist in the form of tacit knowledge in the heads of architects or other stakeholders, and inevitably dissipate. Note, that this knowledge vaporization is accelerated if no architectural documentation is created or maintained in the first place.

Architectural knowledge vaporization due to the loss of architectural decisions is most critical, as it leads to a number of problems that the software industry is struggling with:

- **Expensive system evolution**. As the systems need to change in order to deal with new requirements, new architectural decisions need to be taken. However, the documentation of existing architectural decisions that reflect the original intent of the architects is lacking. This in turn causes the adding, removing, or changing of decisions to be highly problematic. Architects may violate, override, or neglect to remove existing decisions, as they are unaware of them. This issue, which is also known as *architectural erosion* [22], results in high evolution costs.

- **Lack of stakeholder communication**. The stakeholders come from different backgrounds and have different concerns that the architecture document must address. If the architectural decisions are not documented and shared among the stakeholders, it is difficult to perform tradeoffs, resolve conflicts, and set common goals, as the reasons behind the architecture are not clear to everyone.

- **Limited reusability**. Architectural reuse cannot be effectively performed when the architectural decisions are implicitly hidden in the architecture. To reuse architectural artifacts, we need to know the alternatives, and the rationale behind each of them, as to avoid making the same mistakes. Otherwise the architects need to 're-invent the wheel'.

The complex nature and role of architectural decisions requires a systematic and partially automated approach that can explicitly document and subsequently incorporate them in the architecting process. The development of such an approach is explained in a bottom up fashion in this paper. The explanation starts with the notion of architectural decisions

Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

and continues from the point of view of sharing and using these decisions by relevant stakeholders.

We have worked with industrial partners to understand the exact problems they face with respect to loss of architectural decisions. We demonstrate how the system stakeholders exactly can use architectural decisions with the help of a use-case model. We then introduce a first attempt on implementing the proposed approach, a research prototype entitled Archium that aims primarily at capturing architectural decisions and weaving them into the development process. The Archium tool is demonstrated through an example. The contribution of this paper is therefore a first attempt of putting the theory of architectural knowledge into practice.

The rest of the paper is structured as follows: in Section 2 the notion of architectural decisions is presented. Section 3 gives an overview of the proposed approach. Section 4 presents the implementation of this approach in the Archium tool, followed by the tool demonstration through an example in Section 5. Related work is discussed in Section 6. The paper concludes with conclusions and future work in Section 7.

2 Architectural Decisions

To solve the problem of knowledge vaporization and attack the associated problems of expensive system evolution, lack of stakeholder communication, and limited reused we need to effectively upgrade the status of architectural decisions to first-class entities. However, first we need to understand their nature and their role in software architecture. Based on our earlier work [6, 13, 30], we have come to the following conclusions on architectural decisions so far:

- They are cross-cutting to a great part or the whole of the design. Each decision usually involves a number of architectural components and connectors and influence a number of quality attributes.
- They are interlaced in the context of a system's architecture and they may have complex dependencies with each other. These dependencies are usually not easily understood which further hinders modelling them and analyzing them (e.g. for consistency).
- They are taken to realize requirements (or stakeholders' concerns), and conversely requirements must result in architectural decisions. This two-way traceability between requirements and decisions is essential for understanding why the architectural decisions were taken.
- They must result in architectural models, and conversely architectural models must be rationalized by architectural decisions. This two-way traceability between decisions and models is essential for understanding how the architectural decisions affect the system.
- They are derived in a rich context: they result from choosing one out of several alternatives, they usually represent a trade-off, they are accompanied by a rationale, and they have positive and negative consequences on the overall quality of the system architecture.

The exact properties and relationships of the architectural decisions [6, 17] are still the topic of ongoing research. Currently, some properties [26, 21, 13] and relationships [18] have been identified. In this paper, the definition from [30] is used for architectural decisions:

> A description of the choice and considered alternatives that (partially) realize one or more requirements. Alternatives 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.

A description of an architectural decision can therefore be divided in two parts: a description of the choice and the associated alternatives. The description of the choice consists of elements like: problem, motivation, cause, context, choice (i.e. the decision), and the resulting architectural modification. The description of an alternative include: design rules and constraints, consequences, pros and cons of the alternative [30]. For a more in-depth description how architectural decisions can be described see [26, 30].

3 A knowledge grid for architectural decisions

3.1 Introduction

In order to support and semi-automate the introduction and management of architectural decisions in the architecting process an appropriate tool is required. In specific, this tool should be a Knowledge Grid [33]: "an intelligent and sustainable interconnection environment that enables people and machines to effectively capture, publish, share and manage knowledge resources". A knowledge grid for architectural decisions should be a system that supports the effective collaboration of teams, problem solving, and decision making. It should also use ontologies to represent the complex nature of architectural decisions, as well as their dense inter-dependencies.

To resolve the problem of knowledge vaporization, this system must support architects to add, remove, analyze, and

view the architectural decisions made in the architecting process. It must effectively visualize architectural decisions from a number of different viewpoints, depending on what the stakeholders wish to look at.

Furthermore, it must enable easy sharing of decisions between stakeholders. It must be integrated with the tools used by architects, as it must connect the architectural decisions to documents written in the various tools or design environments. Through this, it enables traceability of these decisions to the artifacts of the architecting process. The system must also help the architect trace the architectural decisions to the requirements on one side and the implementation on the other side.

The Griffin research project [11] is currently working on tools, techniques and methods that will perform the various tasks needed for building this knowledge grid. The project has achieved so far two main contributions: a use case model [29] that describes the required usages of the envisioned knowledge grid, and a domain model [10] that describes the basic concepts and their relationships for storing, sharing, and using architectural decisions. In the next subsection, the use case model is briefly sketched while in Section 4 we present an implementation of this knowledge grid: the research prototype Archium.

3.2 Use Cases of Industrial Relevance

In the context of the Griffin project, we have done a thorough investigation on the demands for sharing architectural decisions in the architecting process. First, the demands of the project industrial partners were investigated. Interviews were held with several employees from these partners; architects as well as architecture reviewers were interviewed. The industrial partners are from different domains, namely embedded systems, information systems, and radio astronomy. The reports of the interviews were sent back to the interviewee, and suggested corrections were processed.

The results of the first investigation are presented as a set of 27 use cases [29]. Some use cases cannot directly be linked to these interviews and/or current daily practice. They are either use cases that emerged from related work [18], or from the Griffin research team. The use cases have been described according to the UML 2.0 [27] specification.

The use cases present a wide area of issues concerning the problems discussed in the introduction. To narrow this list, we ranked the use cases on the basis of the number of occurrences in the interview reports. This ranking reflects the importance of the use cases for the industrial partners. A list of the nine most important use case follows, while a more elaborated description is given in Section 4:

(UC1) Retrieve architectural decision

(UC2) Add an architectural decision

(UC3) Check for consistency

(UC4) Validate the set of architectural decisions against the requirements

(UC5) Check implementation against architectural decisions

(UC6) Get consequences of an architectural decision

(UC7) Check for completeness

(UC8) Detect patterns of architectural decision dependencies

(UC9) Check for superfluous architectural decisions

According to this ranking, the basic add and retrieve functionality for architectural decisions is the most wanted. Note, that modifying an design decision is seen as adding another design decisions, which changes the effect of an earlier decision. The more sophisticated evaluation and checking mechanisms are considered somewhat less important, but are desired.

The problems identified in the introduction section are clearly reflected in the use cases. The **expensive system evolution** is tackled by checking the consistency (UC3) and completeness (UC7), but also the requirement validation (UC4) and implementation check (UC5) are very useful in this context. Adding (UC2) and retrieving (UC1) of architectural decisions is essential to alleviate the **lack of stakeholder communication**. Getting the consequences of an architectural decision (UC6) helps in increasing the insight in the effects of a change when the system is evolving. This use case is also the main functionality used to improve the **limited reusability** of architectures. The detection of patterns (UC8) and check for superfluous decisions (UC9) give more insight into the specific architecture, and the potential problems with this architecture.

The following section discusses how the Archium tool, an implementation of the knowledge grid for architectural decisions, fulfils the described use cases.

4 The Archium tool

4.1 Introduction

The Archium tool [13, 2] is a prototype implementation of the envisioned knowledge grid presented in the previous section, and realizes a part of the Griffin project use cases [29]. The Archium prototype is a *specialized version* of the envisioned knowledge grid, as it provides a more pragmatic approach to the usage of architectural decisions: it links the architecting process with the system implementation through transformations.

The Archium tool integrates an architectural description language (ADL) [20] with Java. This language allows an architect to describe the elements from a component & connector view [9], and to express architectural decisions, design fragments [13], and rationale. Archium combines the above by modeling the relationship between architectural decisions and the architectural entities (e.g. components and connectors) in detail. In our earlier work [13], we described how these two aspects are integrated with each other. In this paper, the focus is on *how* these concepts are used. Instead of focussing on *what* these concepts are, as we did in our earlier work [13, 30].

The remainder of this section presents how Archium realizes the Griffin use cases. For each use case, the use of Archium is explained, which if followed by a description of the traceability that is used to achieve many of these use cases. The section concludes, with an overview on how Archium tool is realized.

4.2 Use case realization

Retrieve architectural decision

Use-case: Given the architectural model (or a part of it), trace back to the architectural decision it is based on. Provide the architectural drivers and the rationale of the decisions.

Archium: The visualization of the Archium tool, allows the architect to select a component or connector. This will cause the relevant architectural decisions to appear on the screen, while the architectural decision responsible for the existence of the element is highlighted with a separate color. Hovering above an architectural decision reveals in a tooltip the relevant rationale of this decision. An example of this is presented in figure 1.

Add an architectural decision

Use-case: Add an architectural decision. Prior to adding a decision, certain prerequisites should be satisfied, i.e. questions that need to be answered in order to be able to take the decision. Also, the evaluated alternatives and rationale about the decision can be recorded.

Archium: In Archium, architectural decisions are considered as first class entities. Adding an architectural decision is therefore the definition of the corresponding element in the Archium language, which allows for the description of the alternatives and rationale of a decision. In this sense, the Archium language acts as a formalized template for architectural decisions.

Check for consistency

Use-case: Check if the current set of architectural decisions is internally consistent. Check if the chosen alternatives have inconsistent consequences on the architectural model.

Archium: The Archium compiler and run-time environment do numerous checks to ensure consistency. For example, the compiler ensures communication integrity [1] by checking whether a component refers to architectural elements that are not defined in the components required interface. Another example is the check Archium makes on whether the functional dependencies of an architectural decision are satisfied before an architectural decision is applied.

Validate the set of architectural decisions against the requirements

Use-case: Trace the requirements to the decisions. Check if the requirements are all sufficiently covered by the decisions that are taken.

Archium: Archium supports the tracing of requirements to architectural decisions and can check whether all requirements are addressed in one or more architectural decisions. The Archium compiler warns about requirements that are not addressed.

Check implementation against architectural decisions

Use-case: At a certain moment in time, the architect would like to see to what extent the implementation effort of the development team is in line with the architectural decisions. Consequently, the architect wants to know where in the development process people ignore or disregard the made decisions.

Archium: The Archium compiler includes a code transformation process, which analyzes the architectural elements (e.g. components, connectors) and transforms them where applicable to Java classes. If the implementation team ignores or disregards the architectural decisions made, either the compiler or run-time environment will warn and prohibit violations of the architectural decisions. Consequently, either the implementation team has to realign their implementation with the architecture, or define new architectural decisions to adapt it.

Get consequences of an architectural decision

Use-case: The main consequences of a decision are the changes in the model when a decision is executed. Furthmore, new decision topics can be introduced, as a consequence of an architectural decision. This use case involves getting insight in the consequences of the decision.

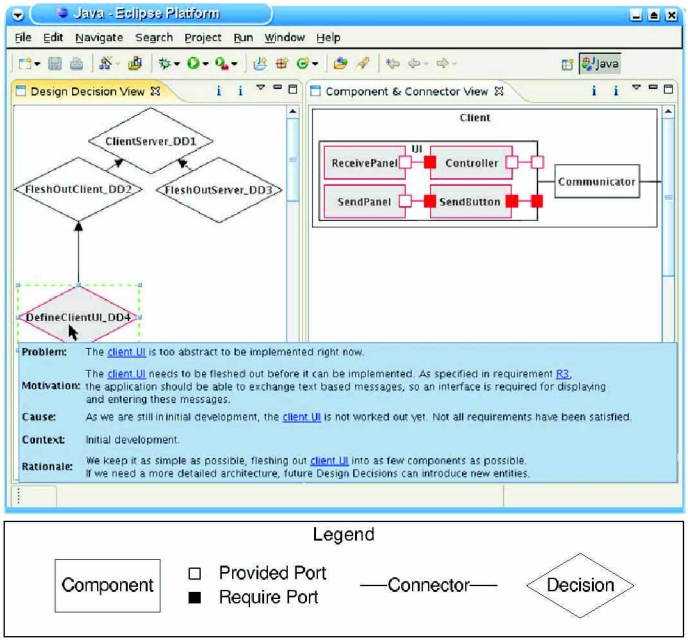

Figure 1. Impact of an architectural decision

Archium: Archium provides a visualization of an architectural decision by a dependency graph, which gives an indication of the consequences of the decision. An architect can asses the consequences of an architectural decision in the visualizer by hovering over the dependency relationships and see in a tool-tip what architectural entities are responsible for a dependency. This helps the architect with evaluating the consequences of an architectural decision. The architect can also select an architectural decision and see its impact on the architecture (see figure 1). This is possible, as the Archium tool explicitly traces the change of an architectural decision to the architecture [13].

Check for completeness

Use-case: Check if all the decision topics are covered sufficiently in the architectural decisions taken.

Archium: The Archium compiler tries to check and warn when one or more rationale elements (e.g. motivations, causes, and problems) are missing. Furthermore, it provides errors when no implementation is provided for a chosen alternative, or no alternative is chosen for an architectural decision.

Detect patterns of architectural decision dependencies

Use-case: In order to be able to check the soundness of the architecture, it is needed to analyze the decisions taken, and the dependencies between these decisions. Identify patterns in the graphs of decisions that can be interpreted in a useful fashion, or lead to guidelines for the architects. For example: decisions being hubs (”Godlike” decisions), circularity of a set of decisions, and decisions that gain weight over time and are thus more difficult to change or remove.

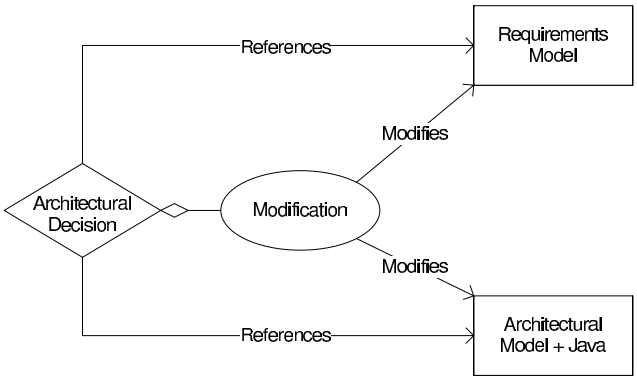

Figure 2. Traceability in Archium

Archium: Archium offers a visualization, which provides a view on the functional dependencies between architectural decisions. This is not an automatic pattern detection (a difficult task by definition), but visualizing the dependencies does support the architect in identifying such patterns.

Check for superfluous architectural decisions

Use-case: The architect wants to know if there are superfluous decisions. This can occur in two situations. Decisions can overlap (i.e. are redundant), e.g. parts of the decisions describe the same, or decisions do not affect the current architectural model at all (i.e. are unnecessary).

Archium: Archium supports the architect in identifying *one* class of superfluous decisions: unnecessary architectural decisions that do not have a function within the architecture anymore. These architectural decision do not have a dependency relationship to the main architectural decisions of the application.

4.3 Traceability

Archium can support many of the use cases due to its ability to provide traceability among different concepts. The traceability helps one to get a better understanding of the design. Figure 2 presents how this works within Archium. Central to the traceability of Archium is the concept of an *Architectural Decision*. It includes a *Modification* part, which alters the *Architectural Model/ Implementation* and the *Requirements Model*. Note, that the architectural model includes other architectural decisions. Remark as well that the architectural decision also includes alternative modifications, which have not been chosen. The

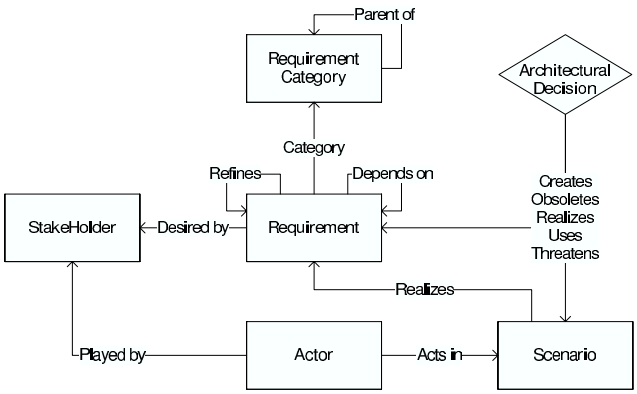

Figure 3. Requirements Model of Archium

different relationships Archium supports between model elements can be classified in two distinct types:

Formal relationships are relationships that are defined in the Archium meta-model. Archium provides explicit language constructs to express these relationships. The Archium tooling (see also section 4.4) checks and constrains these relationships. Furthermore, the tooling uses these relations to satisfy some of the use cases. For example, most of the modification relationships (see figure 2) are formally defined and used by the Archium tool to determine the impact of an Architectural Decision and relate it to the affected components in the architectural model.

Informal links are relationships that are defined in the textual descriptions of various Archium concepts (e.g. a motivation or problem of a design decision). They work similar to hyperlinks and allow the expression of a relationship between two model elements. However, these links do not (formally) define the semantics of this relationship. Informal links are defined by putting a reference to a model element between square brackets (e.g. "[ComponentX]") in a textual description. To make it easier to relate elements, the informal links are context aware, i.e. they follow the naming scope of the surrounding model element in which they are used.

The formal relationships between architectural decisions and the requirements model of Archium is presented in figure 3. In our previous work [13], we already presented how Archium relates (and therefore provides traceability) architectural decisions with an architectural model. Therefore, the focus in this paper is on the traceability between requirements and architectural decisions.

The requirements model (see figure 3) is relatively small, as the primary focus of Archium is on the design part. The model defines five different relationships between an architectural decision and a requirement or scenario. The *creates* relationship is used in the refinement process and traces

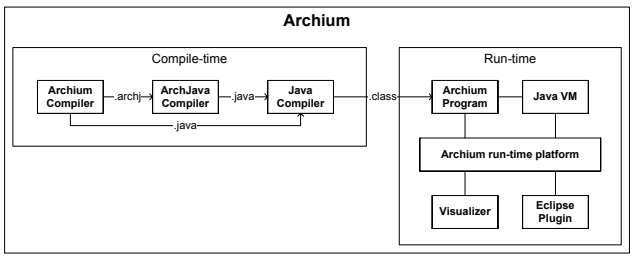

Figure 4. The Archium tool architecture

which requirements came forth of which architectural decision. This is important, as often requirements or scenarios become apparent after an architectural decision is made, as a lot of requirements and/or scenarios only make sense after particular decisions.

The *obsoletes* relation describes the opposite situation, due to new insights certain requirements or scenarios may become obsolete and no longer relevant for the design. The *uses* relationship denotes that an architectural decision uses a requirement or scenario in its rationale to decide between multiple alternatives. The *realizes* relationship denotes that an architectural decisions tries to achieve (a part of) a requirement or scenario. The *threatens* relationship has the opposite semantic, i.e. an architectural decision makes the achievement of a particular requirement or scenario harder. Remark that the existence of an relationship between an architectural decision and a requirement is a confirmation of the fact a requirement is architectural significant.

4.4 Architecture of the Archium tool

Both the requirements model and the informal links are implemented as part of the Archium language. This language includes an ADL (integrated with Java), design decisions, and the requirements model presented in the previous section. The concepts behind the design decisions and ADL part have been explained in earlier work [13]. The language is implemented and used in the Archium tool. Figure 4 presents the architecture of this tool. The tool consists of two main parts: a compile-time part that transforms Archium code into Java classes; and a run-time part that performs run-time analysis and support, and executes Archium programs.

The compile-time part works as a pipes-and-filters system [24]. It commences with the *Archium Compiler*, which transforms Archium code (described in the Archium language) into ArchJava [1] and Java code. The *ArchJava* and *Java Compiler* subsequently transform the latter into Java classes. From a user perspective, this compile pipeline is completely transparent. The *Archium Compiler* invokes the *ArchJava* and *Java compiler* and provides the user with

feedback (e.g. compile errors and warnings) in terms of the input Archium code.

The Java classes generated by the compile pipeline constitute the *Archium Program* component, which is executed by the run-time part of the Archium tool. The *Archium Program* uses the *Archium run-time platform*, among other things, for composition of components and reconfiguration of the connections made by connectors. The *Archium run-time platform* also provides services to the *Visualizer* and the *Eclipse Plugin*, in order to allow architects to inspect the architecture and receive notifications of changes made to it.

The frontend of the *Archium Compiler* has been created with the help of the Java Compiler Compiler (JavaCC) [14], which is a parser generator and abstract syntax tree builder. The input for JavaCC is generated by our ArmPrep tool[2]. ArmPrep is a program, which semi-automatically merges two JavaCC grammar specifications. This tool is used to merge the Archium language with the Java language.

The semantic analyzer of the *Archium Compiler* analyzes the generated abstract syntax tree for the following constraints:

- Type checking for various constructs in the Archium language, e.g. whether the interface of a connector is compatible with a port of a component.
- Naming convention checking, as the Archium language consists of several different concepts, compliance to the naming conventions is particular important.
- Relation checking, especially the relations between the architectural decisions rationale and the architectural entities.

The other constraint checks, like communication integrity [1] and Java constraints, are handled by the ArchJava and Java compiler respectively.

The backend of the *Archium Compiler* consists of a template code generator. The code generator uses Velocity Templates [31] to generate ArchJava and Java code for the various Archium ADL concepts like components, connectors, and architectural decisions. The generated code, that is the *Archium Program*, uses the *Archium run-time platform* to create and maintain a run-time representation of the architectural model, requirements, and the architectural decisions. This representation enables the user to explore and make use of the traceability provided within the Archium model. The *Archium run-time platform* provides a service to this representation using Java Remote Method Invocation (RMI). Both the *Eclipse plugin* and *Visualizer* use this service to analyze, trace, and visualize the Archium model. For the visualization, both utilize the JGraph toolkit [15] to render and automatically layout the architectural decisions and components & connectors.

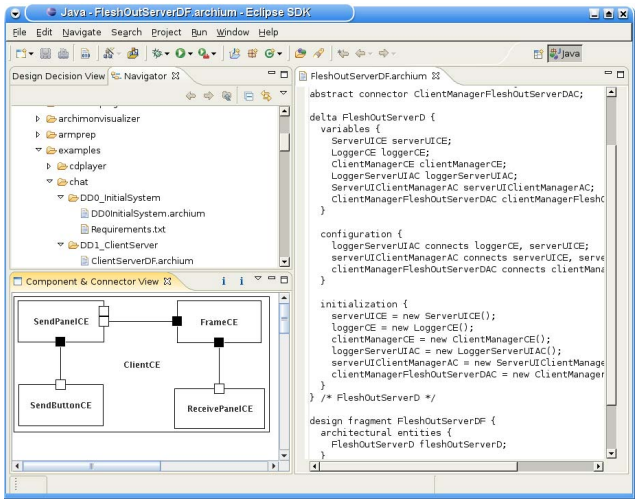

Figure 5. The Archium Eclipse plugin

5 Chat example

5.1 Introduction

To illustrate the Archium tool, an example of a chat program is presented in this section. The example consists of nine architectural decisions, which define the architecture of the chat program. The architectural decisions are numbered chronologically, and marked with the term AD (e.g. AD1 is the first Architectural Decision).

Writing programs in Archium can be done with the Archium Eclipse plug-in. Figure 5 shows a screenshot of this IDE. On the bottom left, a component & connector view of the architecture of the (running) application is visualized. The boxes in this figure represent components, the squares are the ports (black squares are required ports, white provided) and the lines represent connectors. On the right, the main editor for the Archium code is shown. In the remainder of this section, two situations are discussed to illustrate what Archium can do.

5.2 Usage scenarios

After the first four architectural decisions have been made (AD1-AD4), the architecture has been decomposed in a *Client*/*Server* style (AD1). The *Client* component consists of a *UI* and *Communicator* (AD2). The *UI* handles the interaction with the user, while the *Communicator* takes care of the communication with the *Server*. Architectural decision AD3 concerns the structure of the server, but this is out of scope for this example. Figure 6 shows the state of the component and connector view after AD1-AD4. The boxes in this figure denote components, the lines connec-

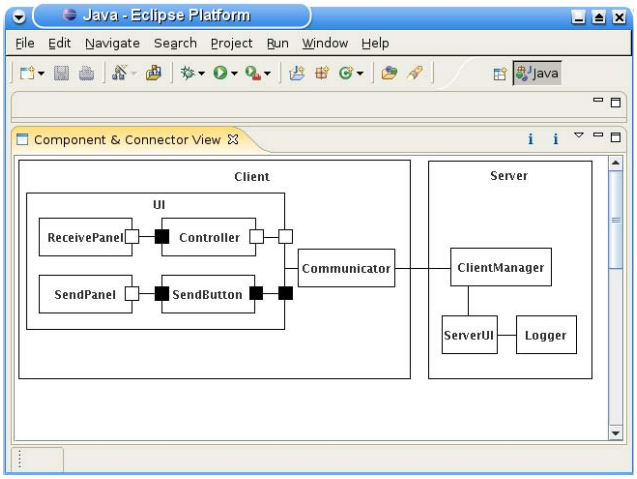

Figure 6. Chat example architecture after AD4

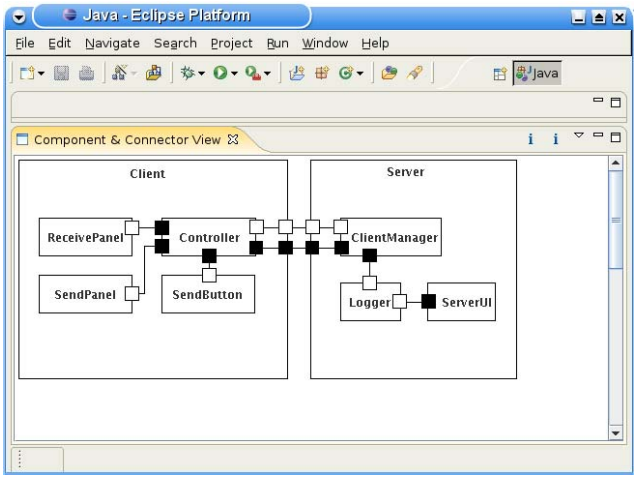

Figure 7. Chat example architecture after AD8

tors, and the little squares are provided ports (white square) and required ports (black square).

While implementing the *UI* (AD4), it seemed that the *Communicator* became redundant. The communication could easily be handled by the components used in the *UI*. Before deciding to remove the *Communicator*, the decisions dependant on the *Communicator* are checked. This revealed that AD2 is only affected. An architectural decision (AD5) is made to remove the *Communicator* (UC5 and UC2: Check implementation against architectural decisions, add a decision). The Archium compiler is rerun and the chat program is executed to check for any problems (UC3: Check for consistency).

The second situation arises when the requirement for the user interface changes: multiple user interfaces should be supported. The current state of the architecture is presented in figure 7. The architect wonders what the consequences of this change in requirements are and traces the original requirement to AD2, where an initial decomposition of the *Client* is made in an *UI* and *Communicator* (UC6: Get consequences of an architectural decision).

However, the *Communicator* is no longer part of the architecture. Tracing the architectural decision dependencies the architect finds the place where the *Communicator* was removed, AD5. As described above, AD5 unfolds the *UI* and removes the *Communicator* component. The rationale of this architectural decision is that the responsibility of the *Communicator* has been relocated to the *Controller* to allow for easy integration with the *UI* components.

This knowledge leads the architect to think up two alternatives to deal with the changed requirement:

- Reintroducing the *UI*. This should contain all the components in the client with exception of the *Controller*. The *Controller* can be reused with different implementations of the *UI*.
- A new user interface is regarded as a new implementation of the *Client*, thereby creating a specific *Client* for each user interface.

From the rationale of AD5, the architect knows that separating the *UI* and the *Controller* will not be easy and the last alternative is probably the easiest to achieve.

6 Related work

Software architecture design methods [4, 5] focus on describing how sound architectural decisions can be made. Architecture assessment methods, like ATAM [4], assess the quality attributes of a software architecture, and the outcome of such an assessment steers the direction of the decision-making process. Our approach focuses on providing ways to capture these architectural decisions, and in the case of Archium, explicitly couple them to the implementation.

Software documentation approaches [9, 12] provide guidelines for the documentation of software architectures. However, these approaches do not explicitly capture the way to take architectural decisions and the rationale behind those decisions.

Architectural Description Languages (ADLs) [20] do not capture the decisions making process in software architecting either. There are two notable exceptions. One is formed by the architectural change management tool Mae [28], which tracks changes of elements in an architectural model using a revision management system. However, this approach lacks the notion of architectural decisions and does

not capture considered alternatives or rationale about the architectural model. The second exception is the domain specific ADL EAML[23], which models architectures for enterprise applications. In EAML, architectural decisions justify rationale, which provides the architecture description that in turn influences the architectural decisions. However, EAML does not describe *how* architectural decisions influence the architecture description, which is something Archium does.

Architectural styles and patterns [24, 8] describe common (collections of) architectural decisions, with known benefits and drawbacks. Tactics [4] are similar, as they provide clues and hints about what kind of techniques can help in certain situations. However, they do not provide a complete architectural decision perspective, as presented in this paper.

Currently, there is more attention in the software architecture community for the decisions behind the architectural model. Kruchten [16], stresses the importance of architectural decisions, and presents classifications of architectural decisions and the relationship between them. Tyree and Akerman [26] provide a first approach on documenting design decisions for software architectures. Both approaches model architectural decisions separately and do not integrate them with the architectural model. Closely related to this is the work of Lago and van Vliet[19], who models assumptions on which architectural decisions are often based, but not the architectural decisions themselves.

Integration of rationale with design is also done in the field of design rationale. The SEURAT [7] system, maintains rationale in a RationaleExplorer, which is loosely coupled to the source code. This rationale has to be added to the design tool, to let the rationale of the architecture and the implementation be maintained correctly. DRPG [3] couples rationale of well-known design patterns with elements in a Java implementation. Just like SEURAT, DRPG also depends on the fact that the rationale of the design patterns is added to the system in advance. The importance of having support for design rationale was emphasized by the survey conducted by Tang et al. [25]. The results emphasized the current lack of good tool support for managing design rationale.

From the knowledge management perspective, a web based tool for managing architectural knowledge is presented in [21]. They use tasks to describe the usage of architectural knowledge. These tasks are much more abstract then the use cases defined in this paper (e.g. architectural knowledge use, architectural knowledge distribution). They do propose a framework for capturing architectural knowledge, by using techniques for enquiring knowledge from human sources, and by mining used patterns. They provide templates for noting down the knowledge. However, they do not integrate the AK with the design process, but distil it, thus it remains separated from the design artifacts.

Finally, another relevant approach is the investigation of the traceability from the architecture to the requirements [32]. Wang uses Concern Traceability maps to reengineer the relationships between the requirements, and to identify the root causes. Methods similar to the proposed ACCA method could be used to generate architectural decision information for Archium.

7 Conclusions & Future work

We believe that the field of software architecture will make significant progress when architectural decisions are treated with the same importance as architectural models. This paper presented nine use cases that described the benefits of architecting with architectural decisions treated as first-class entities. The presented use cases covered the basic functionality for a support tool: adding and retrieving decisions, checks on the relationship with requirements as well as the implementation, and visualization of the relationship of architectural decisions with each other.

The Archium tool is a first attempt of realizing the presented use cases and is aimed at the later stages within design. It weaves architectural decisions into architectural models and connects them to the implementation. Archium supports architects in maintaining the architectural decisions taken in the architecting process. We have described the functionality of Archium by explaining how it fulfils the use cases. Specific instances of the use cases in Archium have been explained with an example. Because Archium is able to store the architectural decisions explicitly as artifacts of the architectural model, it decreases the effects of knowledge vaporization.

The Archium tool has not been tested yet in an industrial setting, so empirical verification data is not yet available. This will take place in the scope of the GRIFFIN project, where currently four industrial case studies are being conducted at four different industrial companies. The cases concern different aspects of managing and sharing of architectural decisions. Special attention is given to the integration with the tools currently used by architects (e.g. Microsoft Word, System Architect, Rationale Rose) in the architecting process. These case studies are in the exploratory phase, and help us to validate the use cases [29] and the domain model [10]. In the future, we will evolve Archium according to the outcome of these case studies. Furthermore, we plan todo some experiments to investigate the balance between the effort of capturing architectural knowledge and its benefits. We are also thinking about integrating Archium with other tools, support for more architectural views, and support for team work.

Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

Acknowledgements

This research has partially been sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge. We would like to thank the people from the GRIFFIN project for the cooperation in creating the use cases.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the 24th international conference on Software engineering*, pages 187–197. ACM Press, 2002.

[2] Archium website, . http://www.archium.net.

[3] E. L. A. Baniassad, G. C. Murphy, and C. Schwanninger. Design pattern rationale graphs: Linking design to source. In *Proceedings of the 25th ICSE*, pages 352–362, May 2003.

[4] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice 2nd ed.* Addison Wesley, 2003.

[5] J. Bosch. *Design & Use of Software Architectures, Adopting and evolving a product-line approach*. ACM Press/Addison Wesley, 2000.

[6] J. Bosch. Software architecture: The next step. In *Software Architecture, First European Workshop (EWSA)*, volume 3047 of *LNCS*, pages 194–199. Springer, May 2004.

[7] J. E. Burge and D. C. Brown. An integrated approach for software design checking using design rationale. In *1st International Conference on Design Computing and Cognition (DCC '04)*, pages 557–576, July 2004.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns*. John Wiley & Sons, Inc., 1996.

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

[10] R. Farenhorst, R. C. de Boer, R. Deckers, P. Lago, and H. van Vliet. What's in constructing a domain model for architectural knowledge? In *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE2006)*, July 2006.

[11] Griffin project website, . http://griffin.cs.vu.nl.

[12] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison Wesley, 2000.

[13] A. G. J. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of WICSA 5*, pages 109–119, November 2005.

[14] JavaCC website, . http://javacc.dev.java.net/.

[15] JGraph website, . http://www.jgraph.org.

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

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

[18] P. Kruchten, P. Lago, H. van Vliet, and T. Wolf. Building up and exploiting architectural knowledge. In *WICSA 5*, November 2005.

[19] P. Lago and H. van Vliet. Explicit assumptions enrich architectural models. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 206–214, New York, NY, USA, 2005. ACM Press.

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

[21] I. G. Muhammad Ali Babar and R. Jeffery. Toward a framework for capturing and using architecture design knowledge. Technical Report UNSW-CSE-TR-0513, University of New South Wales, Australia ans National ICT Australia Ltd., june 2005.

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

[23] S. Sarkar and S. Thonse. Eaml- architecture modeling language for enterprise applications. In *CEC-EAST '04: Proceedings of the E-Commerce Technology for Dynamic E-Business, IEEE International Conference on (CEC-East'04)*, pages 40–47, Washington, DC, USA, 2004. IEEE Computer Society.

[24] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[25] A. Tang, M. A. Babar, I. Gorton, and J. Han. 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)*, pages 89–99, November 2005.

[26] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecure. *IEEE Software*, 22(2):19–27, 2005.

[27] The Unified Modeling Language (UML) website, . http://www.uml.org/.

[28] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic. Taming architectural evolution. In *Proceedings of the 8th European software engineering conference*, pages 1–10. ACM Press, 2001.

[29] J. S. van der Ven, A. G. J. Jansen, P. Avgeriou, and D. K. Hammer. Using architectural decisions. In *Second International Conference on the Quality of Software Architecture (Qosa 2006)*, 2006.

[30] J. S. van der Ven, A. G. J. Jansen, J. A. G. Nijhuis, and J. Bosch. Design decisions: The bridge between rationale and architecture. In A. H. Dutoit, R. McCall, I. Mistrik, and B. Paech, editors, *Rationale Management in Software Engineering*, chapter 16, pages 329–348. Springer-Verlag, march 2006.

[31] Velocity website, . http://jakarta.apache.org/velocity.

[32] Z. Wang, K. Sherdil, and N. H. Madhavji. ACCA: An architecture-centric concern analysis method. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, November 2005.

[33] H. Zhuge. *The Knowledge Grid*. World Scientific Publishing Company, 2004.