Research

Viability for codifying and documenting architectural design decisions with tool support

Rafael Capilla1, *, †, Juan C. Dueñas2 and Francisco Nava1

1Departamento de Ciencias de la Computación, Universidad Rey Juan Carlos, c/Tulipán s/n, 28933 Madrid, Spain
2Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, Ciudad Universitaria, 28040 Madrid, Spain

SUMMARY

Current software architecture practices have been focused on modeling and documenting the architecture of a software system by means of several architectural views. In practice, the standard architecture documentation lacks explicit description of the decisions made and their underlying rationale, which often leads to knowledge loss. This fact strongly affects the maintenance activities as we need to spend additional effort to replay the decisions made as well as to understand the changes performed in the design. Hence, codifying this architectural knowledge is a challenging task that requires adequate tool support. In this research, we test the capabilities of Architecture Design Decision Support System (ADDSS), a web-based tool for supporting the creation, maintenance, use, and documentation of architectural design decisions (ADD) with their architectures. We used ADDSS to codify architectural knowledge and to maintain those trace links between the design decisions and other software artefacts that would help in the maintenance operations. We illustrate the usage of the tool through four different experiences and discuss the potential benefits of using this architectural knowledge and its impact on the maintenance and evolution activities.

Received 19 August 2008; Revised 28 August 2009; Accepted 23 September 2009

KEY WORDS: software architecture; design decisions; architectural knowledge; design rationale; traceability; software maintenance

*Correspondence to: Rafael Capilla, Departamento de Ciencias de la Computación, Universidad Rey Juan Carlos, c/Tulipán s/n, 28933 Madrid, Spain.
†E-mail: rafael.capilla@urjc.es

Contract/grant sponsor: Spanish Ministry of Education and Research; contract/grant number: URJC-CM-2006-CET-0603

Copyright © 2009 John Wiley & Sons, Ltd.
INTRODUCTION

The traditional perspective of software architecture considers it as a set of interrelated components and connectors that are used to represent the main functional parts of a software system [1,2]. Recently, the software architecture research community has faced the need to consider architecture design decisions (ADD) as first class entities that should be stored and documented, embodied with the standard architecture documentation. This shift is mentioned by Bosch [3], who states that ‘The key difference from traditional approaches is that we do not view a software architecture as a set of components and connectors, but rather as the composition of a set of ADD’. Bass et al. [4] justify this as ‘A software architecture is a coherent, justified collection of system’s earliest set of design decisions. These decisions will affect much of what the system will become’.

The idea that considers design rationale (DR) as a key element in an architecture description, and the rationale and principles that guide the design and evolution of software architectures were stated in the early 90s by Perry and Wolf [5] with the formula: architecture = (Elements, Form, Rationale). This rationale becomes useful to understand the decisions made and to explain the reasons that motivated the selection of a particular design choice. Kruchten et al. [6] updated Perry’s formula and defined architecture knowledge (AK) with the equation: AK = design decisions + design. However, this AK encompasses not only the decisions and the rationale but also all other architectural significant information.

Architectural knowledge is often tacit, and, as such, difficult to reproduce. Worse still, the design decisions do not accomplish the traditional architecture modeling activities. Therefore, it is highly desirable that this AK can be codified and formalized to make it usable in the future and be available to be transferred to others. Knowledge codification converts and organizes tacit knowledge into explicit knowledge (i.e. knowledge which is classified and accessible for the organizational members). Codification is ‘to systematize and store information that represents knowledge for a company’ [7], and to store this in databases [8] accessible to the staff. Codifying the tacit architectural knowledge is difficult and requires intensive investment that needs to be justified by further knowledge reuse processes. To ease this activity, codifying design decisions should be supported by specific tools able to assist software architects. This need is motivated in [9], where the authors propose a set of requirements that should be taken into account to support the reasoning and decision-making activities in the software architecture.

The main contribution of this paper is to prove the viability for codifying and documenting design decisions alongside their architectures. Our approach uses a flexible template that codifies architectural knowledge and extends the traditional documentation based on architecture views [10], to a new view called the ‘decision view’ [9], which acts as a new crosscutting view as the design decisions affect the information documented in the other views [10]. The process by which knowledge is synthesized and elicited is known as ‘capturing knowledge’.

In support of this idea, this paper presents the Architecture Design Decision Support System (ADDSS) tool [11]: a web-based research tool specifically designed to assist software architects in the capturing process of ADD and DR. The tool also supports the necessary links to other software artefacts for helping software engineers in the further maintenance and evolution of architecture products. Decisions with ADDSS are captured by following an iterative refinement process, in the same way that software architects develop their architectures. ADDSS eases the visualization of the architecture’s evolution, which is considered as a result of the decisions stored. Moreover, explicit
links are created between requirements, decisions, and architectures to facilitate traceability in the maintenance operations. Such features serve, for instance, to track the root causes of changes and to estimate better the impact of the changes as well. We are aware of the existence of knowledge management (KM) tools that handle generic knowledge as well as certain architectural knowledge, but we believe that the particularities of the architecting processes and the characterization of ADD need specific tooling able to provide software architects with a way to describe these AK without the burden of learning and using other techniques beyond their natural domain. Hence, the scope of this work focuses on specific tools supporting ADD and not on the discussion of general purpose KM tools.

The remainder of the paper is organized as follows. We first motivate this research with examples of the decision-making activity that occurs along typical architecture development processes. Second, we review the related work. Third, we used the architecture of a real system to show the capabilities of ADDSS to codify and document architectural design decisions (ADD), and describe in detail the architecture and the most relevant implementation parts and characteristics of the tool. After that, we summarize the results and the lessons learned from several experiences carried out using ADDSS and highlight and discuss its main strengths and limitations, placing emphasis on the maintenance and evolution activities. We conclude with a summary of the key contributions of our research and ideas for future work.

MOTIVATION

The concept of rationale management in software engineering was first applied in the 1980s, and it was not until the 1990s that Perry and Wolf [5] introduced this as a key element in the software architecture. Unfortunately, architecture descriptions found nowadays are still based on the current IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, known as ANSI/IEEE 1471-2000 [12], which hardly mentions the notion of DR. However, this standard is now under review (the upcoming ISO/IEC 42010 standard) and will soon provide a better description of the DR in the software architecture.

As stated in [13], ‘rationale’ is the justification behind decisions, and DR is gaining, since 2004, an enormous popularity in the software architecture field as the best way to explain the motivation of decisions made and to use these to bridge the traditional gap between the requirements and architectures. The limitations of the current DR methods and the knowledge capture problem, which has generally not provided practical results [13], have led to the recent research. Bosch [3] advocates for explicit representation of design decisions, which should be considered as a first class entities that guide the architectural construction process. The rationale that motivates the decisions, the constraints and the design rules, are elements that should be recorded. Tyree and Akerman [14] point out the need to consider ADD to understand the impact of the choices made by the architects. They focus on the process by which design decisions are carried out and provide a template list of the attributes for codifying them.

The fact that design decisions are frequently forgotten in the development process generates undocumented DR [2] which may lead to high-cost maintenance processes as decisions have to be reproduced when designs are no longer available. Thus, explicitly documented knowledge facilitates the task of reproducing past decisions and reduces the need for architecture recovery processes.
In the software maintenance, frequent changes in the requirements often push to modifications directly to code that are no longer reflected in the architecture, and thus it becomes necessary to align the modifications performed at the design level with the changes carried out at the code level. The SEURAT approach supports the use of rationale in the software maintenance in order to detect inconsistencies that may indicate problems with the design [15]. Because recording DR is considered as a time-consuming activity, and designers are reluctant to keep track of their decisions, we need to motivate users to codify this knowledge as a way to decrease the maintenance effort.

In order to highlight the importance of the DR, we devise four different scenarios.

**Scenario 1. Codification of design decisions:** In this scenario, users codify a set of design decisions, which are motivated by the requirements, in order to produce a software architecture. In this scenario users have to capture and explain the decisions and its rationale in a form that can be understood by others.

**Scenario 2. Maintenance of a software architecture:** In this scenario, a typical maintenance process is carried out over the architecture developed some time ago. The maintenance is based on new customer demands or due to execution errors that may lead, for instance, to enhance and corrective maintenance [16]. Often, due to the project pressure (e.g. a time-to-market requirement), the changes performed in the code are not reflected in the architecture and the design becomes obsolete. In addition, the disappearance of the development team and the replacement by new team members may cause that the knowledge that often resides in the minds of creators vaporizes forever. Therefore, recreating past decisions becomes a priority, as reverse engineering tasks are required to recover the architecture from code descriptions.

**Scenario 3. Track trace links:** A third scenario may occur when we need to track the impact in the design when the requirements change or to know the root causes of changes. In this scenario, the design decisions are used to provide the necessary trace links between the requirements and design or even code.

**Scenario 4. Select between different design choices:** The fourth scenario supports the selection between different design alternatives before the final decision is made. One or more design alternatives are considered and evaluated as valid design choices. Hence, we provide mechanisms to store and reproduce the reasoning activity during the selection of different design alternatives, as decisions may be approved or rejected.

Supporting the aforementioned scenarios requires adequate tooling to assist software architects in their decision-making activities when design decisions are used. The following section discusses the recent tools specifically developed toward this goal.

**RELATED WORK**

In the period 1992–2004, a set of tools supporting software engineering with rationale were developed (e.g. IBIS, gIBIS, QOC, DRL, REMAP, WinWin, etc.) [13], but none of them dealt with rationale in software architectures. Since 2005, new research prototype tools specifically designed for codifying ADD and DR have appeared in the field. Previous work [17] evaluated five existing software architecture tools against the requirements that these tools should handle to include design decisions as a key feature, but only one of the five tools evaluated (i.e. Compendium) partially supported the concept of design decision as a first class entity. Other tools such as ArchStudio (http://www.isr.uci.edu/projects/archstudio/) and SoftArch [18] provided tracing features from the
requirements to architecture plus some support for codifying DR. To date, a few research prototypes enable the codification of ADD as an add-on to architecture models. As stated in [19], a loss of architectural knowledge may lead to maintenance problems, such as the violation of design rules when the system evolves. In addition, design decisions are often intertwined affecting many parts of the design and old decisions may rapidly erode a system or its architecture. To tackle the problems derived from the lack of rationale in software architecture, some authors [20] proposed to enrich the architecture with DR and use the design decisions as bridge between the rationale and the architecture.

Based on these ideas, the Archium tool is presented to model architectures at the same time that decisions are captured. Archium is a Java tool [21] that integrates the requirements, decisions, architectures, and implementation models, and provides traceability among a wide range of concepts. It uses an architecture description language (ADL) that allows the architect to describe the elements from a ‘components & connector’ view to represent ADD, design fragments, and rationale. Archium provides support for architecture views [10,22,23] and viewpoints [24]. Its primary focus is the architectural implementation phase, and it consists of a compiler, a run-time platform, and a visualization tool. The compiler checks the internal consistency of the decisions and produces executable models. The run-time platform provides services to support the functionality of the tool. The tool visualizes decisions using a dependency graph that gives an indication of the consequences of the decision, and supports two types of traceability relationships: (i) formal relationships, which are trace links defined in the Archium meta-model and based on well-defined semantics and (ii) informal links, which are references made in the textual descriptions to model elements. Up to five different types of relationships between the decisions and requirements are defined in the Archium’s requirements model.

The Architecture Rationale and Element Linkage (AREL) is a tool to capture architecture decisions, DR, and design options based on a UML meta-model [25]. In AREL, UML entities are linked to show the relationships between design decisions, design concerns, and design options. AREL supports architecture viewpoints to help designers classify and search for architecture elements. Traceability between requirements, design elements, and DR is achieved through an automated mechanism to support forward and backward tracing. The result of such tracing is shown as UML diagrams. The evolution history of the decisions is captured with eAREL, which is an extension of AREL that counts with specific links and attributes to address this feature. A commercial UML tool, Enterprise Architect, was used to codify and construct AREL and eAREL models and provide templates to extend the standard UML. Architects can create AREL model elements by simply drag and drop from a tool-box during the design modeling phase. The current version of the AREL tool supports automatic tracing. However, the automation is not extended to eAREL and therefore evolution trace is a manual procedure.

The Process-based Architecture Knowledge Management Environment (PAKME) is a web-based AK management system that supports architectural knowledge in architecting activities [26,27]. PAKME is built on top of the Hipergate open-source groupware platform, and provides collaborative features including contact management, project management, and online collaboration tools among others. PAKME consists of four components: user interface, knowledge management, search, and reporting. It supports codification for identifying, eliciting, and storing knowledge that is categorized into four AK management services: knowledge acquisition, knowledge maintenance, knowledge retrieval, and knowledge presentation. It codifies the knowledge of ADD and its rationale using
a set of text templates that are also used to present the knowledge to the user. Other artifacts such as patterns and scenarios can also be codified with PAKME. Different types of relationships between decisions [6] can be defined to support the traceability issues. Knowledge use and reuse are supported in PAKME by means of search facilities.

Knowledge Architect [28] is a tool suite for capturing, managing, and sharing AK, and consists of an AK repository called the Knowledge Architect Server, which stores the knowledge entities, and a number of Knowledge Architect Clients, aimed to capture and manage AK in different formats and contexts. Decisions are highlighted in word documents using a Word plug-in and stored in the same document along with their architectures. The Knowledge Architect Explorer is a client visualization tool that supports the analysis of the captured AK and enables users to search and navigate through traceability links among the knowledge entities.

These research tools represent a first attempt to support decision-making activities in the software architecture. Some of the tools mentioned use text templates, similar to those defined in [6,14], to codify the attributes that describe the decisions, as well as the existing reusable chunks of knowledge such as design patterns and architectural styles [29,30]. Often, the underlying rationale and the reasons that lead to a set of decisions are not explicitly upfront and the assumptions [31–34] that inspire the decisions remain hidden. For instance, PAKME does not provide explicit attributes to describe the assumptions, but it uses a separate template for codifying the rationale of a decision and a field that describes the inspiration of the design option. In addition, the dependencies and the relationships described in the tools are used to bridge the gap between the requirements and architectures, and modeling such relationships becomes a major goal for the maintenance of evolution activities. Jansen and Bosch [35] mention this problem and they define a composition model that is responsible for relating the changes in the decision model to the elements in the architectural model. Knowledge networks and ontologies [36,37] can also be used to organize the relationships between different knowledge artifacts.

OVERVIEW OF ADDSS

In this section we illustrate the ADDSS’ user interface via an exemplar application belonging to the architecture of a real virtual reality system (VR-Church) [38]. We emphasize the characterization of the decisions and the processes that occur along the architecting activity using ADDSS. Registered users in ADDSS may have different roles and privileges to create and visualize the architecture projects, such as Figure 1 shows. Hence, distributed teams can access and share the design decisions from different projects.

Once a project and the architecture information are stored, the user must upload the requirements that will motivate the design decisions. ADDSS stores the basic information about the requirements, consisting of a name, a description, the requirement’s number, and the type of the requirement. Figure 2 shows how requirements are captured and visualized.

After the requirements for the new architecture have been stored, the user can start building the architecture of the system. ADDSS offers the following characteristics:

1. Characterization of architectural knowledge, which supports (i) codification of design decisions using specific templates of attributes that store the decisions and their rationale. The mandatory and optional attributes provide a flexible mechanism for characterizing and
documenting the design decisions for different user needs and (ii) storage of design patterns and architecture styles that can be retrieved as design solutions during the characterization of the decisions.

2. **Model traceability links** aim to represent the relationship between the requirements, architectures, and decisions. These links are defined to bridge the gap between the requirements and the architectures. In addition, links between decisions can be defined and used to estimate the impact when a decision is added, removed, or changed, and help maintainers to track the changes better.

3. **Iterative architecture development process** comprises the main architecting activity in which architectures and decisions are stored for each refinement of the architecture. Users of ADDSS can easily visualize the evolution of the design and navigate and browse through the decisions.

4. **Support for different architecture views** facilitates the creation and visualization of different perspectives of the same architecture.

5. **Query system** which provides a basic knowledge extraction facility to obtain information about the requirements, decisions, architectures, and their relationships.

Figure 1. List of available projects stored in ADDSS.
6. **Documentation of decisions** extends the traditional architecture documentation following the goals stated in the ‘decision view’ [9], as decisions are automatically documented as PDF files that contain detailed descriptions of the decisions made for each architecture, the requirements for each decision, and the trace links between the design decisions.
Characterization of architectural knowledge

This section discusses the first characteristic (1), which describes how the architectural knowledge (e.g. decisions, patterns) is represented in ADDSS. Capturing the design decisions implies some kind of representation or formalization technique. ADDSS codifies the AK using a set of attributes that are defined in [6,14,39,40]. To store the DR in a more agile way, ADDSS does not use a large template of attributes; instead, it captures a subset of attributes that are classified into mandatory and optional attributes. Therefore, different users with different needs can select the amount of information they want to store and tailor it for the needs of each particular organization or individual.

The mandatory attributes defined in ADDSS are:

- **Decision**: The name of the decision.
- **Pattern and type of pattern**: These specify a well-known design pattern selected as a valid design solution. A pattern classification is also provided. Users are not forced to select a pattern for a particular decision. These patterns are stored in the ADDSS repository as a knowledge base of reusable design solutions.
- **Responsible and date**: The person responsible for making a decision and the date when the decision was made. These two attributes are handled automatically by ADDSS.
- **Version**: A version number is automatically assigned to the current decision. This attribute facilitates evolution activities.
- **Rationale**: This field is used to describe the motivation of a decision as it explains why a particular decision is chosen.
- **Description**: A text field used to describe the decision made.
- **Category**: This attribute assigns a category to a decision. The values allowed are **Main**, **Derived**, and **Alternative**. The category attribute serves to classify the decisions and see which of them are considered as the alternatives of a main decision or a specialization of a previous one. This attribute facilitates the understanding during the reasoning activity and for review during the maintenance.
- **Status**: This attribute is used in combination with the category attribute to assist in the evaluation of the decision. The values admitted are **Pending**, **Approved**, **Rejected**, and **Obsolete**. Initially, all decisions are considered as pending, and during the reasoning activity this status may change to approved or rejected. In addition, during the evolution of the system, some revisited decisions may change its status to obsolete if the user considers these as outdated. The status attribute helps to track the history of the decisions during evolution activities.

The attributes mentioned above are shown in the screenshot of Figure 3, where the user codifies a design decision and stores the information that characterizes the decision using a text template. On the left side of Figure 3, users can select the requirements that motivate the current decision. Complementary to the information shown in Figure 3, users can capture a set of optional attributes that enhance the description of the decisions. At any time during the knowledge-capturing process, users can select the optional items they want to store through the ‘profile’ menu option shown in Figure 4(a). Once the user has profiled his/her specific needs, the information is captured in a pop-up window with the current decision (see Figure 4(b)).
Model traceability links

This section discusses characteristic number 2, which defines the links between requirements, decisions, and architectures, and it is particularly important for maintenance. During the codification of the decisions, users select the requirements that motivate a particular decision (left side of Figure 5) and ADDSS stores the trace links between the requirements and decisions transparently to the user. Those requirements, which have been used before, are labeled (U), so that we can know how many requirements have been fulfilled during the architecting process.

ADDSS can define explicit links to previous decisions and create a dependency network of decisions, which helps to track these during the maintenance operations. The right side of Figure 5 displays a dependency established between the current decision and a previous one by simply marking a check box. To ease the characterization of these links, ADDSS defines only the basic dependencies among decisions but not the type of the link, which in many cases implies an additional reasoning activity to select the most suitable dependency type. One disadvantage of this approach is that the semantics of the link is not captured.
Once a set of decisions is made, the user can upload an architecture, which is the result of that set of decisions. ADDSS uploads an image of the architecture developed using external modeling tools. The figure representing the architecture is converted to a thumbnail image and displayed to the user, such as Figure 6 shows. As a result, a link between the decisions made and the architecture
uploaded is defined. Therefore, explicit links between the requirements, decisions, and architectures are established in ADDSS to provide full traceability between different software artefacts and to allow forward and backward traceability. We must mention that the trace links between the decisions and the architecture products refer to complete architectures and not to the smaller parts of the design, as ADDSS is not a modeling tool. Architects can upload small refinements of the architecture to show the evolution of changes in detail, but there is no way to define links from the decisions to concrete architecture pieces.

**Iterative architecture development process**

This section discusses the third characteristic of ADDSS, which simulates the means by which software architects build their designs as an iterative process where the architectures and decisions are created and stored together. Prior to the characterization of the decisions, the user has to select an architecture view for which the decisions and designs will be created. The decisions and the architectures are captured and visualized concurrently during the design phrase. Figure 7 shows the result of the architecting process supported by ADDSS.
The architectures are shown in the order in which they were created as they represent the successive refinements of the design resultant from the decision-making activity. Hence, the evolution of the architecture can be easily displayed, as users can navigate and browse through the decisions and their corresponding architectures. None of the tools presented in the related work provide this facility. ADDSS displays up to five architectures in the same row and with the same size, in order to avoid the horizontal scroll and increase the usability. Further, the details of the architecture can be easily displayed by simply clicking on the thumbnail image selected in Figure 7. ADDSS enlarges the images to its original size in a pop-up window, as shown in Figure 8.

Support for different architecture views

The fourth characteristic of ADDSS allows the architect to select between several architecture views (see Figure 7) for depicting their architectures for different stakeholders. Users can display on the same screen different UML diagrams, each of them belonging to a different view.
Figure 7. Visualization of the evolution (refinements) of a sample architecture.

Query system

The characteristic number 5 of ADDSS provides a basic ‘query’ facility that enables the extraction of knowledge that can be used during or after the architecting process to (i) produce information about the relationships between requirements, decisions, and architectures; (ii) know which requirements are the root causes of a particular architecture; and (iii) estimate which artefacts can be impacted by a change in a requirement or a decision. The extraction of such a knowledge is realized through a set of predefined queries that relate the different software artefacts stored in the tool’s repository. For instance, users can exploit the forward trace links to know which decisions or architectures are related to one or several requirements. Conversely, following backward trace links enables to discover those requirements that are the motivations of a change in a decision or which decisions would be affected if a user modifies a particular architecture. ADDSS does not allow for free queries based on logical relationships; rather it exploits the trace links stored during the architecting phase.
Document decisions

The last characteristic of ADDSS documents the design decisions explicitly, which is helpful for reviews or to communicate decisions to others. Documenting the design decisions and its rationale provides a better and quicker understanding of the system, in particular for novice users. Architecture documentation often becomes obsolete for several reasons, and tracking the changes made in the design or requirements to ensure the conformance between the requirements and design, or even code is frequently a time-consuming task. This situation becomes more critical in those organizations that have high staff turnover and the decisions made are never documented. Therefore, documenting design decisions during the architecture development process becomes a major goal.

ADDSS enhances the standard architecture documentation by documenting explicitly the design decisions and their rationale and produces automatic PDF reports containing such information. This facility can be accessed through the ‘report’ menu option displayed in Figure 9(a), and the PDF document is displayed in a pop-up window, such as Figure 9(b) shows. Figure 9(c) shows some details of the documentation generated by ADDSS. This document includes: (i) information about the project; (ii) requirements list; (iii) the decisions made for each requirement with the
Figure 9. Documentation generated by ADDSS for a whole project.
resultant architecture for a set of decisions; and (iv) the chain of the dependencies between the design decisions (i.e. the dependency network). The documentation generated by ADDSS helps users to check the connections between requirements, decisions, and architectures, and know which decisions are related to other decisions. This documentation can be easily shared and distributed among other stakeholders for communication, learning, or assessment purposes. Documented design decisions reduce the burden of the maintenance operations as stakeholders can learn how a particular system was built, which design alternatives were considered, and which design decisions have to be reproduced if needed. As a result, documenting this architectural knowledge leverages the understanding of the system for less expert users or for newcomers to the organization.

**ADDSS with the architecture of a virtual reality system**

In order to validate the utility of ADDSS for codifying design decisions, we used the tool to store and document the decisions of a virtual reality system developed in 2004 at the Rey Juan Carlos University (URJC). The target system consisted of the construction of a virtual reality church (VR-church) with a virtual tour to show the inside of an ancient church located in a small village in the north of Spain. This virtual tour shows the most interesting places inside the church and virtual users can learn about the cultural heritage through information text points that are displayed along the virtual tour [38]. The size of the source code files is around 25 kB and the size of the 3D model of the church is 13.3 MB. A special characteristic of VR systems deals with the large number of requirements needed for modeling the virtual scene. In our case study we stored in ADDSS the functional, non-functional, hardware requirements, and a subset of the 3D modeling requirements that might affect some parts of the architecture. Table I summarizes the number and the type of the requirements identified.

<table>
<thead>
<tr>
<th>Requirement type</th>
<th>Number of requirements identified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>28</td>
</tr>
<tr>
<td>Non-functional</td>
<td>5</td>
</tr>
<tr>
<td>Modeling requirements of the virtual scene</td>
<td>62</td>
</tr>
<tr>
<td>Hardware and platform-specific</td>
<td>6</td>
</tr>
</tbody>
</table>

Our goal was to reproduce the construction of the software architecture of the VR-Church system and codify the design decisions made during the development phase. Two experts in the virtual reality systems domain and two of the co-authors of this work, acting as software architects, participated in the case study. Once all the relevant stakeholders were registered in the tool, we started with the first iteration of the architecture. ADDSS displays first a blank image belonging to the first architecture product that will stored, such as shown in Figure 10(a). Afterwards, the architect can start making decisions, which are codified using the template of Figure 10(b).

On each iteration, the architect makes decisions that are evaluated before the final design choices are made. Hence, decisions and architectures are created together, and architectures are uploaded and displayed (Figure 10(c)) during the design activity. At the end of the design phase, the evolution of the architecture of the VR-Church system is shown in Figure 10(d). During any development or maintenance activity, design decisions can be evaluated one or several times, and this information
is stored using the status and category attributes. Figure 11 simulates part of the reasoning activity. When capturing the decisions for the first time, (1) an architect assigns for the first time a ‘pending’ status and an ‘alternative’ category to each decision. After several decisions are stored, (2) the architect evaluates the different design alternatives and selects the best or the optimal ones. For instance, during the first iteration of the architecture of the VR-Church system, four main alternative decisions were evaluated and two of them were selected as valid. Finally, (3) the architect changes the status of the selected choices as ‘approved’ and its category to ‘main’ or ‘derived’. Non-selected decisions also change their status to ‘rejected’.

The same process is repeated until all the requirements are fulfilled and all the pending decisions resolved. As a result, five iterations containing four intermediate architecture products and one final architecture were stored. In the example, we codified 32 coarse-grained main design decisions, including 15 alternatives, and 24 decisions out of 32 were approved. For the sake of simplicity, Appendix A summarizes the coarse-grained decisions codified during the five iterations: the second column in the table represents all the decisions evaluated including the alternative ones, whereas the third column shows those that were selected. The last column describes the rationale (i.e. the motivation) of the decisions selected. Additionally, a summary of the dependencies defined between
the decisions is presented in Appendix B. The following conclusions can be extracted as a result of the example carried out.  

*Facility to codify decisions*: We observed from the capturing process that the main decisions (i.e. coarse-grained decisions mostly based on architectural and design patterns) were easier to codify than the fine-grained ones (e.g. a decision that implies the creation of a class attribute or a variation point in the architecture), because users can select many of these key decisions from the list of
patterns stored in ADDSS as small. In both cases, the viability for codifying knowledge was proven successfully.

**Flexibility to codify knowledge:** Storing a small number of attributes and give the architect the chance to capture additional information items (i.e. optional attributes in ADDSS) reduced the effort spent during the knowledge capturing, increased the degree of flexibility during the characterization of a decision, and the user’s satisfaction.

**Granularity and relevancy of decisions:** As the granularity of the decisions decreases (i.e. fine-grained decisions), the number of decisions and their dependencies increases as well as the complexity of the decisions network. As not all the decisions can be stored, the architect and other relevant stakeholders should determine which decisions are more important to be stored. For instance, in an architecture recovery process, it is unnecessary to reverse all the minor decisions belonging to small refactorings of the system, whereas only the most important ones should be recovered. Hence, codifying only the most relevant decisions should be enough to provide a general understanding of the system and make the set of decisions more maintainable and keep these under control. Main or key decisions are often made in the early stages of the architecture development process as they determine the shape of the architecture and they can be associated to the selection of: (i) a whole architecture determined by its quality attributes; (ii) a design or architectural pattern; (iii) a class or package, or (iv) a particular middleware, protocol, or framework. Therefore, a balance should be achieved to define the granularity of the decisions to be captured.

**Dependency types:** Codifying only basic dependencies makes our approach more agile rather than codifying the semantics of the links, but in some cases a loss of accuracy in the links could make the decisions network more difficult to understand.

**Integrity of the decisions network:** The dependencies between decisions clearly impact the maintenance of the architecture. Adding, removing, or modifying a decision brings consequences to the decisions network that should be reorganized, and the integrity of the dependencies must be checked to ensure the consistency of the changes made. The current version of ADDSS does not yet provide automatic mechanisms to evaluate the modifications in the network of the decisions, which have to be tracked manually.

**ADDSS ARCHITECTURE**

ADDSS is a web-based tool for codifying and documenting architectural design decisions. The main reasons for creating a web tool are: (i) to provide access for remote users and (ii) to provide collaborative capabilities that could be used by distributed teams for knowledge sharing purposes. Figure 12 shows the client-server architecture of ADDSS. The client interface is compatible with the two most popular browsers (i.e. Internet Explorer and Mozilla Firefox) by means of compatible style sheets (cascade style sheets (CSS)). The functionality of the core components is supported in the server by PHP scripts. ADDSS uses a relational database to store the information about projects, architectures, users, patterns, and design decisions. The decisions are codified and maintained using the decision support module. ADDSS employs external libraries to produce the PDF documents containing the decisions and their architectures. The tool can visualize the evolution of the architectures by means of thumbnail images that are produced when the image of the architecture is...
uploaded. Finally, the knowledge search module produces the relevant relationships between the different software artefacts using a set of predefined queries.

**Structural view**

Figure 13 illustrates the main elements of the software architecture of ADDSS. We used a three-layered style to organize the logical parts of the system (i.e. static view of the architecture). The presentation layer provides a web interface that handles the packages supporting the user management, architecture visualization, online architecture documentation, and multilingual support. The user management package provides the user registration and password encryption facilities and defines different user roles with different permissions that are used to access the information stored in the ADDSS database. The architecture visualization package generates and displays the thumbnail images of the architectures uploaded into the tool. The reporting facilities package uses the documentGenerator class to produce automatically online PDF reports from the information stored in the database. Finally, the multilingual support allows the user to select between English and Spanish dynamically without having duplicated web pages for different languages.

The decision management layer represents the core functionality of the decision-making activities. The projects & architectures info package defines the relationships between projects, architectures, architecture views, systems, requirements, and the architecture’s iterations. Hence, different projects can be stored in the system and each project may have one or several systems. Each system has a single architecture which is built under a set of iterations and based on a set of requirements that can be uploaded into the ADDSS repository. Software architectures can be described by one or several architecture views that represent the refinements of the design during the architecting process. Users can browse and navigate through these architectures in chronological order. The decision support package is aimed to represent the design decisions and the dependencies between such decisions as well as between the decisions, requirements, and architectures. These
Figure 13. Static view of ADDSS software architecture.
main decisions can be based on pre-defined design patterns and architectural styles (reusable patterns package), which are maintained in the ADDSS repository. In addition, the knowledge search package provides a set of pre-defined queries that can extract useful information from the knowledge repository and produce combined results relating requirements, decisions, and architectures. Finally, the data layer handles the web request from the clients and provides access to the information stored in the MySQL databases.

The decision-making activity

The core of ADDSS is to support the decision-making activity, which tries to simulate the reasoning activity of the software architect. Figure 14 shows a sequence diagram outlining the order and the details of this reasoning activity, including the roles of the main elements involved in such processes and the interactions among them. At the beginning, a user acting as a software architect obtains the architecturally significant requirements that have been previously stored for the target architecture (1, 2). After analyzing the requirements that will be implemented in the architecture (3), the user selects an architecture view (4) under which a set of the decisions will be made. Once the view is selected, the user can start the reasoning activity by making a set of design decisions.

ADDSS follows a strict iterative process that reflects the evolution of the architecture products. Users must check if there are any existing iterations containing architectures (5, 6), and then create new iterations (7, 8) to store new decisions and architectures. Once the user has selected a specific iteration (9), he/she can create a new design decision (10, 11) motivated by a subset of the requirements (12). The new design decision is then characterized (13) with the attributes provided by ADDSS. In this characterization process the user makes explicit the tacit knowledge that resides in his/her mind, and design patterns can be selected and retrieved (14, 15, 16) to provide a particular design solution and characterize the decision as well.

Because decisions can be made and discussed for approval at different moments during the design process, the user initially assigns a status (i.e. approved, rejected, pending, obsolete) indicating the current situation of the decision, and a category for classifying it into one of the existing categories (i.e. main, alternative, derived) (17). At the beginning, all the decisions should be classified as ‘alternative’ with ‘pending’ status until a deliberation process changes these values. Thus, the stakeholders can approve or reject the decisions and change their category and status. Those alternative decisions that were non-selected remain in the repository as ‘rejected’, as users can use them to replay the history of the design alternatives considered.

Once a decision is fully characterized, dependencies to the previous decisions can be defined (18). These dependencies form a network of decisions that can be used, for instance, to estimate the impact of adding, removing, or modifying a decision in the architecture, to track the root causes of the changes in the design, or to estimate the impact analysis when the requirements change.

Because the selection of the right design choices may involve several stakeholders that might discuss about the decisions to be made, users must evaluate (19) the decisions stored in ADDSS and assign a final status and category to them (20). This decision-making activity can be enacted several times until all the requirements are fulfilled. Finally, a user can obtain and browse the list of decisions made (21, 22) and upload the architecture (23) which is the result of the set of decisions made for that particular iteration.
Figure 14. Outline of the decision-making process with ADDSS.
IMPLEMENTATION

Figure 15 provides an outline of the key technologies used to implement the ADDSS architecture. ADDSS itself is implemented in PHP with AJAX (Asynchronous JavaScript and XML), and the client-server architecture of ADDSS processes the user requests from the clients that are handled by an Apache 2.0 server. The client interface uses compatible CSS to display adequately the HTML+JavaScript pages for, at least, the two main Web browsers: Internet Explorer and Mozilla Firefox. Part of the client-server interaction is supported by the AJAX technology to handle the user requests. User input is sent to the server by posting XML documents over XMLHttpRequest, and the server reciprocates with XML messages containing synchronization instructions, which are processed by pluggable client-side JavaScript modules. We selected AJAX to enhance the response from the server to visualize faster the images of the architectures during the iterative architecture construction process.

The user registration facility employs MD5 encryption routines to encrypt user passwords in order to preserve the confidential aspects of ADDSS users. ADDSS supports multilingual selection that is handled by PHP session variables. Hence, the same language is used until the user leaves its current session. The server is based on Apache+PHP+MySQL technologies to support the functionality of ADDSS and to store the information generated during the decision-making activity. Additionally, the thumbnail images of the architectures are generated using an external library called gd2.dll. The images are organized adequately in the directory file system and the links for accessing the images are stored in the ADDSS database. The explicit documentation of the design decisions and their rationale retrieved from the database is converted automatically to PDF documents on behalf of the external library fpdf.php. ADDSS produces three similar types of documents that include detailed information about the decisions, architectures, requirements, and relationships among all the software artefacts. Such a documentation can be generated for an entire project, a single architecture, or for a concrete architecture’s iteration.
EXPERIENCES

To date, we have carried out four case studies with ADDSS to observe the viability for codifying ADD in parallel with the architectures. In order to address this issue, we used case studies as our research methodology because they constitute a powerful and flexible technique, considered suitable for exploratory research, both prospectively and retrospectively [41]. Three of the four case studies deal with the decisions belonging to the virtual reality system already introduced in this paper. The other case study codifies the decisions of a tool for managing the variability in software product lines. The fourth experiment focuses more on the effort for codifying decisions across different phases of the life of the system. In all the cases, the participants had to capture the mandatory attributes described before to characterize the decisions.

First case study: The design of the first study comprising the participants and training, target system, and input material was carried out in 2006. The design of the case study included 22 master students of a computer science course offered at the Rey Juan Carlos University (URJC) of Madrid (Spain). At least around 40% of the subjects could be considered as senior software engineers with 3–4 years of experience in software companies. We trained the students in concepts about software, design decisions, as well as in the ADDSS tool version 1.0. We used 2 lectures of 2 h for training purposes. We organized the students into 11 pairs. As input material we provided the students with a subset of the requirements and design decisions of the VR-Church system consisting of 12 requirements and 26 design decisions belonging to the VR-Church system, but no alternative decisions were given. The participants had to store the design decisions and the architectures and produce the architecture for the VR-Church system. The subjects spent an average of 10 h in codifying the decisions of the VR-Church system and storing their architectures. Some teams spent less effort than others because of the seniority of some of the team members, as many of them worked for software companies. In general, most the teams employed between four and six iterations to deliver the final architecture. After performing the experiment, we conducted several interviews with the students and all of them had to fill us a questionnaire to evaluate the capabilities of ADDSS 1.0. The surveys indicated that the users highlighted the high usability of ADDSS, the ease of use, and the low learning effort required to use it. Most users felt that some important capabilities were missing, such as explicit support to store alternative design decisions, support for different architecture views, or a better integration with the software architecture modeling tools among others. We used scenario 1 to know whether the students were able to understand and codify properly the set of decisions given and produce a suitable architecture. The students appreciated the online reporting facility to automate the architecture documentation as it documents explicitly the design decisions made. Only a few team members established dependencies between the decisions, probably because they did not understand the importance of such kind of relationships. Furthermore, many users had difficulties to model the tacit knowledge of the decisions made (e.g. difficulty to codify their own expertise in an explicit form) as they had to store these in parallel with the modeling activities, but all of them understood the importance of having such knowledge recorded. As a general conclusion we extracted that our approach was valid to codify the design decisions with moderate effort and to test the feasibility of the codification strategy.

Second case study: Before starting our second evaluation, we improved the capabilities of ADDSS 1.0 to produce the version described in this work (i.e. ADDSS 2.0) (released in 2007). Some of
these new capabilities are: better architecture visualization, support for architecture views, support for alternative decisions, and attributes to characterize the status and category of the decisions. The design of the experiment involved two software architects and two experts in the virtual reality domain. As input material we used the same VR-Church system with the complete list of requirements consisting of 16 requirements, and no input decisions were used by the participants. The participants spent one month in the design and modeling activities of the VR-Church system, and four intermediate architecture products were built before the final architecture was released (5 iterations were made). During the decision-making activity, the architects codified 32 design decisions including 15 alternatives and they approved 24 out of 32. They established 30 dependencies between the decisions made. In the experience, we observed that the status and category attributes were useful as we tried to simulate the reasoning process during the architecting phase. Many of the decisions made were based on well-known design patterns and architecture styles and only a reduced set of these were fine-grained decisions belonging to variation points or class attributes. We observed as useful the trace links to connect the decisions to requirements and to architectures, whereas isolated decisions have limited usefulness, and its real value appears on the interconnection to other nodes (i.e. decisions). The links between the decisions help to understand the relationships between them, to detect incompatible decisions, and to track better traceability problems in successive refinements of the architecture during the maintenance operations. In this experience, scenario 1 was used for codifying the decisions and scenario 3 was carried out partially as we only modeled the dependencies between different software artefacts but we did not use these dependencies to track the system changes. We used scenario 4 to evaluate the alternative decisions and we assigned a status and a category to simulate the reasoning activity.

Third case study: The third experiment was carried out in collaboration with the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany during June, 2007. We tested the ADDSS capabilities in combination with a reverse engineering tool, developed at the Fraunhofer IESE, called Software Architecture Visualization and Evaluation (SAVE) [42]. The design of the experiment included one of the co-authors and one software engineer from the Fraunhofer IESE, both of them skilled software engineers. The participants did not need any additional training but several meetings to discuss the value and applicability of codifying design decisions were needed. As input material we used a set of architectures belonging to the DecisionModeler, a tool for managing the variability in Software Product Lines. These architectures were recovered with SAVE, which is a reverse engineering tool developed at the Fraunhofer IESE and is used to analyze the evolution of changes in existing systems. In addition, three non-functional requirements were defined and stored to motivate the selection of alternative decisions based on an evaluation of these quality attributes. Because SAVE is unable to describe the rationale of the changes made in the DecisionModeler, we used ADDSS to store and document such decisions as well as to maintain the main architectures recovered with SAVE. We employed four iterations to record such architectures but we did not store the intermediate designs belonging to the small refactorings made in the code of the DecisionModeler. ADDSS was proven useful and complementary to SAVE; but to scale-up ADDSS to industrial applications, the Fraunhofer recommended to extend the multi-user management features and to provide a way for resolving overlapping or incompatible decisions [43]. The combination of both tools was useful to produce a more accurate documentation of the system architecture. Also, we noticed that the codification of micro-architectural (i.e. fine grained)
decisions took more effort than the coarse-grained ones, and decisions belonging to small refactor-
ings in the code were not very useful to understand the major changes between versions. In addition
 to supporting scenario 1, we evaluated scenario 2, as the architecture of the DecisionModeler tool
 had to be maintained and the decisions stored were revisited several times. Therefore, we believe
 that the frequency of the reverse engineering operations can be reduced as the decisions stored
can be used to understand the evolution of the changes made in the system. Scenario 4 was also
 supported to select between different design alternatives, but we did not examine the status of the
 history of the decisions.

Fourth case study: The fourth and last experience was carried out during January 2008 and
the design of the experiment involved 17 master students from a regular computer science course
of the URJC. At least around 50% had around 3 years of experience in software companies
as senior software engineers [44]. For training purposes, there were three lectures, each of 2 h,
covering the notion of architectural design decision. The goal of the experiment was twofold:
(i) evaluate the knowledge-capturing process and revisit past decisions using the trace links and
(ii) measure the effort in reasoning and codifying the design decisions as well as in modeling tasks
for the following three phases: development, maintenance, and evolution. As input material we
gave the students a set of decisions and requirements for each of the phases mentioned before.
For the development phase we gave the students 16 requirements belonging to the VR-Church
system and five tables containing the design decisions already made in the second experience. The
students had to replay the development, phase storing in ADDSS the design decisions, and at the
same time model the architectures belonging to those decisions. The effort employed in both tasks
was measured separately. After that we started a maintenance phase and gave the subjects a new
set of 8 requirements. In this phase the subjects had to make new design decisions (not made
before) and model the architectures. Again, the students measured the effort spent in codifying
the design decisions and modeling their architectures. Finally, we carried out an evolution phase
in which the students had to revisit all the decisions made in the two previous phases and evolve
the architecture by refining the previous decisions. During the evolution phase, the students were
ordered to add fine-grained decisions in the form of class attributes, methods, and variation points
into the UML classes of the static view of the architecture. Scenario 1 was enacted successfully
to codify the decisions with the rationale as well as the resultant architectures. Scenarios 2 and 3
were enacted as the students had to maintain the architecture in the subsequent phases after the
development phase, and in some cases use the trace links to estimate which architecture products
would be affected by the new requirements. Finally, we used scenario 4 in the evolution phase to
remember the status and category of the previous decisions and change the status of some decisions
as some of them became obsolete. Because measuring the effort is not part of the goal of this
paper, the results can be found in [45]. After the experiment was carried out, we observed that (i)
fine-grained decisions introduced a higher level of complexity in the decisions network and not all
small decisions were interesting to codify and (ii) the students perceived as quite useful for the
maintenance and evolution phases to count with the decisions stored in the development phase as
they understood what was made after periods of inactivity between the phases. The overall expe-
rience was very positive as we evaluated the results in three different phases of the software life
cycle and measured the trend in the decisions made and the stability of the architecture based on the
number of dependencies between the decisions, which indicated the trend in the complexity of the
design.
Findings and lessons learned

To date, we have evaluated two different versions of ADDSS in different contexts and with different users. The main goal of the four evaluations carried out was to prove that codifying architectural knowledge is possible with tool support. All the evaluations highlighted the usability of ADDSS and its usefulness for codifying design decisions.

In the first evaluation we showed that decisions can be codified and documented as well as how these can be related to other products of the software life cycle. Users perceived ADDSS easy to use and many of them felt the tool useful for their architecting activities. The visualization feature of ADDSS showing the evolution of the architectures and decisions was also a strong point of the tool. On the contrary, the description of the decisions stored was not detailed enough as the students expected, and some of them had difficulties in transferring the tacit knowledge to an explicit form.

In the second evaluation we strengthened the capabilities of ADDSS to produce version 2.0, in particular including the support for alternative decisions was quite useful to explain better the reasoning activity. The retention of alternative decisions provides a convincing evidence for replaying architectural knowledge and the reasons for selecting or rejecting a particular design decision. Another observation is that coarse-grained decisions, mostly based on patterns, were easier to codify, manage, and maintain than having hundreds of fine-grained decisions. Both types of decisions are equally important but maintaining large sets of decisions is more difficult because of the increasing number of interdependencies among them. In addition, the possibility to select optional attributes to enhance the description of the decisions was a feature appreciated by the users as it allows tailoring the amount of knowledge to be captured to the different needs.

The third evaluation was slightly different from the previous two because it involved an external organization. In this case, we stored the decisions from several architectures recovered with a reverse engineering tool that belongs to the major refactorings made on a system. Because the decisions that led to such refactorings were never recorded, we used ADDSS to codify and reproduce the decisions made in the past and used these to support the future evolution of the system. Hence, we aligned the decisions at the architecture level with the corresponding modifications in the code. In addition, researchers from the Fraunhofer Institute, where the evaluation took place, perceived interesting to employ two complementary tools to facilitate the understanding of a particular system or architecture.

Finally, our fourth evaluation was quite interesting as we gathered some initial estimations of the effort spent in codifying the decisions for three different phases of the software life cycle and compared this effort with the one spent in modeling tasks. During the evolution of the architecture, we found that the storage of fine-grained decisions consumed a significant part of the effort. On the other hand, fine-grained decisions provide a more detailed degree of smaller parts of the architecture. In addition, these fine-grained decisions introduce much more complexity in the decisions network making it difficult to manage and understand it. The design decisions codified across different phases of the software life cycle showed evidence of its usefulness to revisit past decisions, recreate the design history and line of reasoning, and observe the trend in the complexity of the architecture based on the links between decisions.

In relation to other works, we believe that our approach is more complete than AREL [25] and Archium [21] as regards decision making, but these two tools offer some important modeling features not currently supported by ADDSS. Only PAKME [26,27] offers a similar approach, but
ADDSS has other capabilities, such as the facility to display the evolution of the architectures and to select between the mandatory and optional attributes to codify knowledge. The experience described in [45] proves the value of design rationale in the software architecture enacting different use cases to estimate the perceived value of the information that can be tailored for the different needs.

From the experiences carried out we noticed that the amount of knowledge to be codified influences the agility of the knowledge elicitation process, as users have difficulties to formalize tacit knowledge models into a usable form. Hence, the new activities and processes described in [46] for capturing, sharing, assessing, and learning knowledge should be introduced gradually as they change the traditional way in which architects perform their work and thus have to be convinced about the expected benefits.

At this point, some important questions arise in our minds: How detailed should be the knowledge that we need to capture? Are all the stakeholders interested in all the decisions? How should the information about decisions be embodied in the architecture documentation? These and other questions need to be answered in order to produce the right documentation.

Another complementary outcome is that these kinds of tools can be used as learning systems by less expert architects. Having the design decisions available facilitates the learning curve to reproduce the reasons that led to a particular architecture and helps maintainers to evolve their systems better. Such architectural knowledge adds value for assessment activities as it can be used to provide further recommendations based on right and wrong experiences. To conclude, this architectural knowledge should be codified as a reusable knowledge experience that can be used by others when the creators of such knowledge are not longer available.

Future work

From our experience using ADDSS, the key areas for future work will include the following:

- With maintenance in mind, it is clear the need to formalize the dependencies between the decisions and provide an automatic checking mechanism for the decisions in order to detect those overlaps and incompatible decisions. In addition, adding new types of dependencies will increase the expressiveness of such relationships [47,48], but we should carefully balance such formalization with the practical usage of these tools.
- Strengthen the mechanisms to control better the evolution and the history of the decisions, such as eAREL does [25].
- Provide better methods to extract the relevant knowledge that would help in the development phase or even for evaluation and assessment procedures [48].
- Provide active knowledge sharing [49–51] capabilities such as RSS and Wikis, also proposed in [46], to facilitate collaborative capabilities. Converting the knowledge stored in ADDSS to XML documents facilitates to exchange these across the Internet.
- In order to reduce the intrusiveness of tools like ADDSS, we should pursue the integration with other software engineering tools.
- Extend the capabilities of ADDSS to support product line development [1] such as connecting the design decisions to variability models [47,52–54].
- Validate the tool in an industrial setting and carry out the maintenance experiences in the long term.
The work presented in this paper and the tool developed so far, convey a simple but effective approach in the practice of the software architecture. We believe that some of the main benefits for having explicitly documented this tacit knowledge are: increase the awareness about the history of the architecture, facilitate maintenance and evolution activities, package this knowledge as a learning material, and avoid architectural erosion from undocumented reasons of changes.

SUMMARY

We have described our experiences in developing and using ADDSS, a tool for codifying, managing, and documenting architectural design decisions. Our approach helps software architects to build and maintain software architectures alongside with their design decisions. The partial automation for recording and documenting tacit knowledge advances the state of the art in software architecting as new activities accomplish the traditional modeling ones. From the evaluations carried out we can highlight that recording such architectural knowledge is possible and useful.

APPENDIX A

Summary of coarse-grained decisions made during the construction of the VR-church software architecture (Table AI).

<table>
<thead>
<tr>
<th>Project iteration</th>
<th>Alternative decisions</th>
<th>Decisions made</th>
<th>Design rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Three-layered style</td>
<td>A three-layered architectural style is selected to separate the functionality of the three main parts of the VR-church system</td>
<td>The three-layered style provides a clear separation of the concerns in the VR system</td>
</tr>
<tr>
<td></td>
<td>Two-layered style</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Pipe&amp;Filter style for connecting the graphic engines. This includes the following decisions:</td>
<td>A pipe&amp;filter style was selected and comprises the following decisions:</td>
<td>Pipeline architectural styles are commonly used in VR engines, and they constitute a proven solution for implementing the connection between the graphic engines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(i) Top layer: Visualization features and interaction with hardware devices</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ii) Middle layer: Business logic for the VR-church guided tour</td>
<td></td>
</tr>
<tr>
<td>Project iteration</td>
<td>Alternative decisions</td>
<td>Decisions made</td>
<td>Design rationale</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
<td>----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>(iii) Lower layer: Pipe&amp;Filter (pipeline) for graphic engines</td>
<td>(iii) Lower layer: Pipe&amp;Filter (pipeline) for graphic engines</td>
<td>2 Apply MVC</td>
<td>The MVC pattern was selected to explain the functionality of the two higher layers of the architecture. The MVC pattern provides a clear separation and good understanding for highly interactive systems and specially for VR systems because it decouples the processing layer from the presentation and I/O devices.</td>
</tr>
<tr>
<td></td>
<td>No style for connecting graphic engines</td>
<td>Don’t use MVC</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Apply MVC</td>
<td>Several controllers for each type of VR device and 1 view</td>
<td>A customizable controller and a view class have been defined</td>
</tr>
<tr>
<td></td>
<td>Define 1 customizable controller and 1 view</td>
<td>Define 1 controller and 1 view</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Attach all HW devices to the controller class</td>
<td>Attach HW devices to the controller class</td>
<td>Attach HW devices to the controller but also I/O immersive devices to the view class.</td>
</tr>
<tr>
<td></td>
<td>Attach HW devices to the controller and immersive devices. This decision is composed by the following decisions:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(i) A customizable attribute is defined in the controller class and two methods for initializing devices and processing events</td>
<td>(i) A customizable attribute is defined in the controller class and two methods for initializing devices and processing events</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ii) An observer is defined with the view to notify the changes from the model.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table AI. Continued.

<table>
<thead>
<tr>
<th>Project iteration</th>
<th>Alternative decisions</th>
<th>Decisions made</th>
<th>Design rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(i) Define packages and classes for the object hierarchy</td>
<td>(i) Define packages and classes for representing the object hierarchy in the scene</td>
<td>The specification of the object hierarchy is quite important for VR applications because during start-up, not all the objects have to be displayed at the same time</td>
</tr>
<tr>
<td></td>
<td>(ii) Create classes to support 3D virtual effects</td>
<td>(ii) Create classes to support 3D virtual effects</td>
<td>Motion models may vary depending of the type of the VR system. For instance, a virtual user walking inside the church will have a different motion model from an observer inside the aircraft of a flight simulator system</td>
</tr>
<tr>
<td></td>
<td>(iii) Package to show text tourist points</td>
<td>(iii) Package for representing text for the tourist points in the tour</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(iv) Processing the events and sensors</td>
<td>(iv) Define one package and two classes for processing the events and sensors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(v) Define the speed of the virtual user and the motion model</td>
<td>(v) Define a class for supporting the virtual user, the speed, and the motion model</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(vi) Support for distribution</td>
<td>(vi) Define a class for supporting the distribution features for future used</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Define attributes and methods to refine the classes specified in previous iterations (this decision is composed by a variety of micro-decisions that are not shown in this table)</td>
<td>Define attributes and methods to refine the classes specified in previous iterations (this decision is composed by a variety of micro-decisions that are not shown in this table)</td>
<td>Variability and class attributes definition are specified to refine the architecture before the customization process is realized to produce a particular system architecture</td>
</tr>
<tr>
<td>4</td>
<td>Define the relationships between classes and packages</td>
<td>Definition of the relationships between the new classes and packages introduced before</td>
<td>Relationships are necessary to describe object hierarchies and the relationships between the main elements in the VR-church system. These links define the necessary relationships between</td>
</tr>
<tr>
<td>Project iteration</td>
<td>Alternative decisions</td>
<td>Decisions made</td>
<td>Design rationale</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------</td>
<td>----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>4</td>
<td>Specialize the classes for the virtual 3D effects</td>
<td>Specialize the classes for the virtual 3D effects</td>
<td>micro-decisions that depend on more coarse-grained decisions</td>
</tr>
<tr>
<td></td>
<td>Specifying the class inside the OpenGL package</td>
<td>Specifying the class inside the OpenGL package</td>
<td>We must specify which type of virtual effects will be supported, its main features, and the initialization method of each effect</td>
</tr>
<tr>
<td>4</td>
<td>Add a new class called ‘context’ to decouple the management of the events and the 3D virtual effects</td>
<td>Add a new class called ‘context’ to decouple the management of the events and the 3D virtual effects</td>
<td>The size of the OpenGL library requires to details the methods and attributes needed to display the information points in the virtual tour</td>
</tr>
<tr>
<td>5</td>
<td>Customization of the architecture. This decision comprises the following three decisions:</td>
<td>Customization of the architecture. This decision comprises the following three decisions:</td>
<td>The goal of introducing this new class is to support better the evolution of this part of the architecture</td>
</tr>
<tr>
<td></td>
<td>(i) Select the classes for 3D lighting</td>
<td>(i) Select the classes for 3D lighting</td>
<td>The classes removed from the architecture are not necessary for the functionality of the VR-church application</td>
</tr>
<tr>
<td></td>
<td>(ii) Remove the class for supporting distribution features</td>
<td>(ii) Remove the class for supporting distribution features</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(iii) Remove the class sensor from the events package</td>
<td>(iii) Remove the class sensor from the events package</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Two main object groups were defined in the object hierarchy in the first level. In the second level we defined three object groups for the external view and five for the internal view</td>
<td>Two main object groups were defined in the object hierarchy in the first level. In the second level we defined three object groups for the external view and five for the internal view</td>
<td>The performance quality attribute drives the customization of the object hierarchy in order to reduce the loading time of the scene-graph during start-up and rendering operations</td>
</tr>
<tr>
<td></td>
<td>No object groups defined for the external and internal view of the church. Only objects are specified in the hierarchy</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Only one object group was defined for the inner view of the church</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX B

Dependencies between coarse-grained decisions for the VR-church architecture (Table BI).

Table BI. Dependencies between coarse-grained decisions for the VR-church architecture.

<table>
<thead>
<tr>
<th>Project iteration</th>
<th>Identifier of the decision</th>
<th>Descriptions of decisions made</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D1</td>
<td>A three-layered architectural style is selected to separate the functionality of the three main parts of the VR-church system</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>D2</td>
<td>Top layer: Visualization features and interaction with hardware devices</td>
<td>D2, D3, D4 depend on D1</td>
</tr>
<tr>
<td></td>
<td>D3</td>
<td>Middle layer: Business logic for the VR-church guided tour</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D4</td>
<td>Lower layer: Pipe&amp;filter (pipeline) for graphic engines</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D5</td>
<td>The MVC pattern was selected to describe the functionality of the two higher layers in the architecture</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>D6</td>
<td>A customizable controller and a view class has been defined</td>
<td>D6 depends on D5</td>
</tr>
<tr>
<td></td>
<td>D7</td>
<td>Attach HW devices to the controller but also I/O immersive devices to the view class</td>
<td>D7 depends on D5</td>
</tr>
<tr>
<td></td>
<td>D8</td>
<td>A customizable attribute is defined in the controller class and two methods for initializing devices and processing events</td>
<td>D8, D9 depend on D6</td>
</tr>
<tr>
<td></td>
<td>D9</td>
<td>The observer is defined in the view to notify changes from the model. Methods for drawing and updating the view are included</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D10</td>
<td>Define one package and several classes for describe the object hierarchy in the scene</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>D11</td>
<td>Define classes to support 3D virtual effects</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D12</td>
<td>Package for representing the information text point for the virtual tour</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D13</td>
<td>Define one package and two classes for processing the events and sensors</td>
<td></td>
</tr>
</tbody>
</table>
Table B1. Continued.

<table>
<thead>
<tr>
<th>Project Iteration</th>
<th>Identifier of the decision</th>
<th>Descriptions of decisions made</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D14</td>
<td>Define a class for supporting the virtual user, the speed, and the motion model</td>
<td>Depends on D6</td>
</tr>
<tr>
<td>4</td>
<td>D15</td>
<td>Define a class for supporting the distribution features for future use</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>D16</td>
<td>Define the attributes and methods required to refine for the classes specified in previous iterations (this decision is composed of a variety of micro-decisions that are not shown in this table)</td>
<td>Depends on D6, D8, D9, D13, D14</td>
</tr>
<tr>
<td></td>
<td>D17</td>
<td>Specialize the classes for the virtual 3D effects</td>
<td>Depends on D11</td>
</tr>
<tr>
<td></td>
<td>D18</td>
<td>Specify the class inside the OpenGL package</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>D19</td>
<td>Add a new class called ‘context’ to decouple the management of the events and the 3D virtual effects</td>
<td>Depends on D11</td>
</tr>
<tr>
<td>5</td>
<td>D20</td>
<td>Definition of the relationships between the classes and packages</td>
<td>Depends on D6, [D10-D15], [D17-D19]</td>
</tr>
<tr>
<td></td>
<td>D21</td>
<td>Customization: Select the class for 3D lighting</td>
<td>Depends on D17, D19</td>
</tr>
<tr>
<td></td>
<td>D22</td>
<td>Customization: Remove the class that supports the distribution features</td>
<td>Depends on D15</td>
</tr>
<tr>
<td></td>
<td>D23</td>
<td>Customization: Remove the class sensor from the events package</td>
<td>Depends on D14</td>
</tr>
<tr>
<td></td>
<td>D24</td>
<td>Two main object groups were defined in the object hierarchy in the first level. In the second level we defined three object groups for the external view and five for the internal view.</td>
<td>Depends on D10</td>
</tr>
</tbody>
</table>

ACKNOWLEDGEMENTS

The authors acknowledge the developers of the tool for the many hours spent on testing the different releases. We also acknowledge people from the Fraunhofer IESE (Kaiserslautern Germany) for their valuable contribution to test the capabilities of our tool and for the suggestions for future improvements.

This work has been partially funded by the PILOH project of the Spanish Ministry of Education and Research programme under grant number URJC-CM-2006-CET-0603.

The work of Juan C. Dueñas is partially funded by the Eureka ITEA-MOSIS project of the Spanish Ministry of Industry, Tourism, and Commerce under the PROFIT programme with grant number FIT-340005-2007-35.
REFERENCES


AUTHORS’ BIOGRAPHIES

**Rafael Capilla** is an assistant professor of software engineering at the Universidad Rey Juan Carlos of Madrid (Spain) and holds a PhD in Computer Science. He has participated in several European and Spanish research projects as well as software projects for commercial PDA applications. His research interests are software architectures, particularly decision-making processes, product line engineering, software variability, and Internet technologies. He is also the research leader of the ADDSS tool. Capilla is a member of IEEE CS. Contact him at: rafael.capilla@urjc.es.

**Juan C. Dueñas** is a professor in the Department of Telematics Engineering at the Universidad Politécnica de Madrid (UPM) in the Telecommunications School. His research interests include Internet services, service-oriented architectures, software architecture, and software engineering. Dueñas has a PhD in telecommunication engineering from UPM. He has participated in several European IST and ESPRIT projects, as technical manager and principal UPM researcher. Currently he is the Deputy Director of Department of Telematics Engineering. Contact him at jcduenas@dit.upm.es.

**Francisco Nava** is an associate professor at the Universidad Rey Juan Carlos of Madrid (Spain). He holds a PhD in Computer Science. He has more than 25 years’ experience as a senior system analyst, project manager, and freelance consulting, as well as Director of the Information Systems Department at L’Oreal. His research focuses on software architecture and computer-based patient record. Contact him at: francisco.nava@urjc.es