Evolution-Centered Architectural Design Decisions Management

Meiru Che, Dewayne E. Perry
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78705
United States
meiruche@utexas.edu, perry@mail.utexas.edu

Abstract: Software architecture is considered as a set of architectural design decisions. Managing the evolution of architectural design decisions helps to maintain consistency between requirements and the deployed system, and is also necessary for reducing architectural knowledge evaporation. In this paper, we propose a UML metamodel based on the Triple View Model from our previous research work. The UML metamodel incorporates evolution-centered characteristics to manage architectural design decision evolution. It helps to capture and trace the evolution of architectural design decisions explicitly, and reduces the evaporation of architectural knowledge that results from decisions evolution. We conduct a case study to illustrate the effectiveness of the metamodel.

Key–Words: Architectural design decisions, architectural knowledge evolution, design knowledge management

1 Introduction

Software architecture plays a foundational role in achieving system functional and non-functional requirements. The architecting process provides a high-level framework to support designing, developing, testing, and maintaining software systems after deployment. The traditional concept of software architecture focuses on components and connectors, as Perry/Wolf proposed in [18]. Perry and Wolf considered the selection of elements and their form to be architectural design decisions, and the justification for these decisions to be found in the rationale. It was not until 2004, with Bosch's paper [3] at the European Workshop on Software Architecture, that software architecture has finally come to be considered as a set of architectural design decisions (ADDs). This specific focus on architectural design decisions led to a broader focus on architectural knowledge [16].

Since architectural knowledge representation and knowledge evaporation have major influence on complexity and cost of system evolution, communication among stakeholders, and software architecture reuse, effectively capturing and representing ADDs can help to organize architectural knowledge and reduce its evaporation, thus providing a better control on many fundamental architectural drift and erosion problems [18] in the software life cycle. A number of models and tools to capture, manage, and share ADDs have been proposed in the recent years [4], [14], [21]. In the research of architectural knowledge management, managing the evolution of ADDs is one of the most critical aspects requiring more attention in research and industry [17]. However, the existing models do not support architecture evolution very well [5].

In order to address this need, we propose a UML metamodel based on our Triple View Model (TVM) [6] that is developed for managing ADDs documentation. The core idea of this UML metamodel is to ensure that the evolution of ADDs in the software architecting process can be captured and tracked properly, thus the evolutionary architectural knowledge can be shared by all the stakeholders without evaporation. Several evolution-centered characteristics are incorporated into the metamodel, which enable us to achieve this goal. We subsequently illustrate a case study to validate our UML metamodel.

We make the following three contributions:

1) **The UML metamodel** - A fine-grained definition for the TVM [6], which designs each ADD category in the TVM as a UML class with multiple attributes to describe the ADD information;

2) **Evolution-centered characteristics** - The evolutionary characteristics in the UML Metamodel, which help to capture architectural knowledge for managing the evolution of ADDs in a software architecting process;

3) **An illustrative case study** - A brief validation for the UML metamodel using an industrial project. The results demonstrate the effectiveness of the metamodel on managing ADDs evolution.

2 Triple View Model

In order to capture the complete architectural design decisions set, we have proposed the Triple View Model to clarify the notions of ADDs, and to cover the key features in the architecting process [6].

2.1 Framework

The TVM is defined by three views: the element view, the constraint view, and the intent view. This is analogous to Perry/Wolf model's elements, form, and rationale but with expanded content and specific representations [18]. Each view in the TVM is a subset of ADDs, and the three views constitute an entire ADDs set. Specifically, the three views mean three different aspects when creating an architecture, i.e., "what", "how", and "why", as shown in Figure 1. The three aspects aim to cover design decisions on "what" elements should be selected in an architecture, "how" these elements combine and interact with each other, and "why" a certain decision is made.

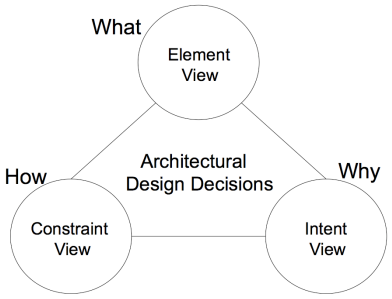

Figure 1: Triple View Model Framework

The detailed contents of each view in the TVM are illustrated in Figure 2.

In the element view, the ADDs describe "what" elements should be selected in an architecting process. We define computation elements, data elements, and connector elements in this view. Computation elements represent processes, services, and interfaces in a software system. Data elements indicate data accessed by computation elements. Both computation elements and data elements are regarded as components in software architecture, and connector elements are (at minimum) communication channels (that is, mechanisms to capture interactions) between those components in the architecture. Note that the ADDs in the element view consist of traditional architecture concepts, which are mainly represented by components and connectors.

In the constraint view, the ADDs are defined as behavior, properties, and relationships. They describe constraints on system operations and are typically derived from requirement specifications. Specifically, behavior illustrates what a system should do and what it should not do in general. It specifies prescriptions and proscriptions based on requirement specifications, and influences the design decisions in the element view. Properties are defined as constraints on a single element in the element view, and relationships are constraints on interactions and configurations among different elements.

The ADDs in the intent view are composed of rationale and best-practices in the architecting process. Rationale, which includes alternatives, motivations, trade-offs, justifications and reasons, is generated when analyzing and justifying every decision that is made. Best-practices are styles and patterns we choose for system architecture and design. The architectural decisions in the intent view mainly exist as tacit knowledge [20], and we need to document them during the decision making process, so that stakeholders can clearly understand these tacit architectural knowledge during the architecting process. What's more, the consistent communication among different stakeholders effectively decreases architectural knowledge evaporation.

2.2 Scenario-Based Approach to ADDs Management

The TVM is the foundation of ADDs documentation and evolution. In the SceMethod [6], we aim to obtain and specify the element view, the constraint view, and the intent view through end-user scenarios represented by Message Sequence Charts (MSCs). Figure 3 illustrates the SceMethod process.

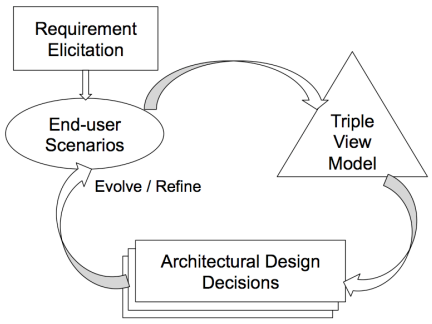

Figure 3: The SceMethod Process

For the sake of brevity, we will not discuss the detailed process for the SceMethod that is proposed in [6]. The derived ADDs results by applying the SceMethod are as follows:

For the element view:

Computation Elements = {*Agent Instances*}

Data Elements = {*Interaction Messages*}

Connector Elements = {*Channels between Agents*}

For the constraint view:

Behavior = {*Prescriptions; Proscriptions*}

Prescriptions = {*Positive Scenarios*}

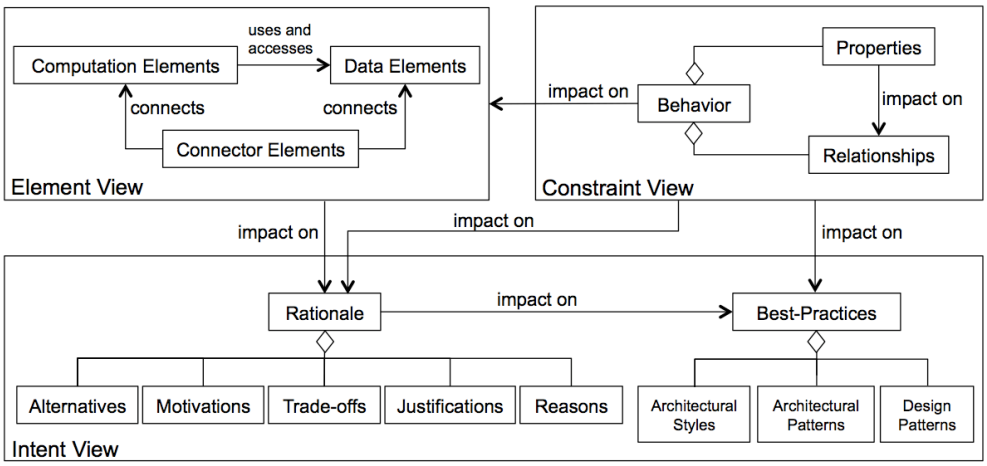

Figure 2: Triple View Model for Architectural Design Decisions

Proscriptions = {*Negative Scenarios; Exceptions*}
Properties = {*Receive; Issue; Check*}
Relationships = {*Event Traces by Path Expressions*}
For the intent view:
Rationale = {*Answers or Solutions to The Intent-Related Questions*}
Best-Practices = {*Architectural Styles; Architectural Patterns; Design Patterns*}

3 Evolution-Centered UML Metamodel

3.1 Metamodel

Based on the previous research work where we focused on managing the documentation and evolution of ADDs, we develop a UML metamodel of our Triple View Model. The UML metamodel provides more detailed evolution-centered characteristics which enable us to manage the evolution of architectural knowledge.

Figure 4, 5, and 6 illustrate the UML metamodel. In Figure 4, we can see that computation elements, data elements and connector elements in the element view are specified as classes where each of them has a bounch of attributes to describe its information. *Behavior* and *Relationship* classes describe the ADDs in the constraint view, and the architectural decisions for the "properties" of each element are merged into the corresponding element class as attributes. In Figure 5 and 6, the ADDs on "Rationale" and "Best-Practices" are described as specific classes that extended the general *Rationale* and *Best-Practices* class.

3.2 Evolution-Centered Characteristics

The metamodel aims to manage the evolution of architectural design decisions. It has the following evolution-centered characteristics that enable us to

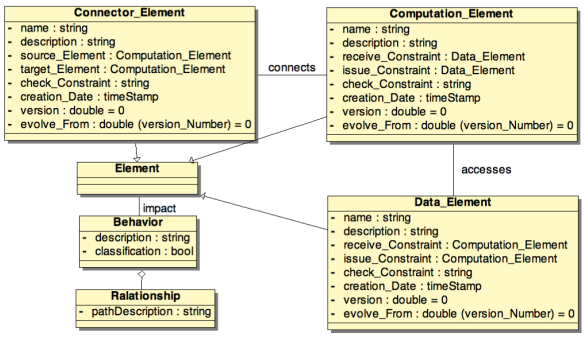

Figure 4: Metamodel for the Element and the Constraint View

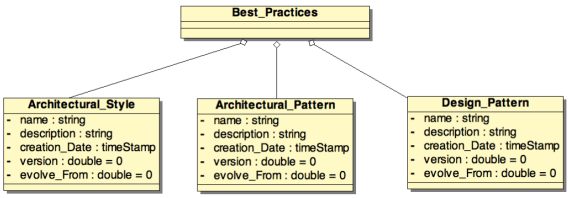

Figure 5: Metamodel for Best-Practices

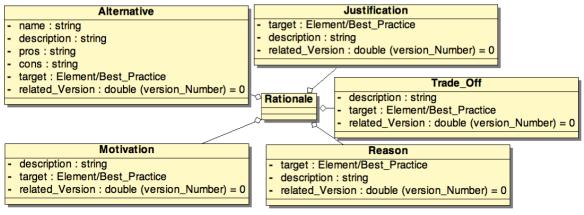

Figure 6: Metamodel for Rationale

make ADDs evolution explicitly, so that all the stakeholders can share architectural knowledge, specifically, decreasing the evaporation of the evolutionary knowledge in the software architecting process.

3.2.1 Evolution-Related Attributes

We define several evolution-related attributes to describe the ADDs classes in the metamodel. We use *creation_Date* to record the specific time stamp when a certain ADD is made. *Version* is used to specify a version number assigned to each ADD, which helps to manage multiple copies of a certain ADD during the evolutionary process. Moreover, we aim to record the evolution history by an attribute called *evolve_From*. When a new ADD is made that is evolved from an existing one, the *evolve_From* attribute is used to indicate the version of the previous ADD based on which the newly ADD is evolving. Another evolution-related attribute we propose is the *related_Version* for the Rationale classes, which is used to specify the version number of an ADD the rationale describes.

3.2.2 Traceable Evolution Chain

Besides the aforementioned evolution-related attributes, the UML metamodel provides a complete evolution chain for every ADD's evolutionary change, which enable us to keep tracking the evolution history of the ADDs set. Specifically, the *evolve_From* attribute provides a bridge to establish the evolution chain for ADDs. Through the evolution chain, the architect and other stakeholders are able to trace the changing information of a certain ADD, and they can share consistent architectural knowledge. Additionally, a traceable evolution chain keeps all the evolution history explicitly and hence significantly reducing the evaporation of the evolutionary architectural knowledge during the software development process.

3.2.3 Version-Specific Rationale

We can see that in all the rationale classes we have an attribute called *related_Version*, which is used to record the specific version number of an element or a best-practice that a rationale describes. The version-specific rationale classes provide us multiple ways of managing the evolutionary knowledge of ADDs by either tracking the rationale for a target ADD for its multiple versions or tracking the rationale for a certain version of an ADD. Thus, the tacit knowledge can be obtained according to the specific requirement on ADDs evolution.

4 Case Study

We select the same industrial project as we did for the TVM to illustrate how the UML metamodel works on managing the evolution of ADDs. It is an industrial project provided by the Italian electrical company ENEL [1], and provides us a real industrial environment for making architectural design decisions and managing decisions' evolution. In this project, a power plant monitoring system is to be established to improve power plant efficiency, to reduce operation and maintenance costs, and to avoid forced outages. The main requirements of the power plant monitoring system are gathered from [8], [9].

From the results of the previous case study on the TVM and the SceMethod [6], we obtain the following ADDs in three different views, which are shown in Table 1, 2, and 3.

Table 1: The Element View Results

Computation Elements	Sensor Manager
	FaultDetection Engine
	Alarm Manager
	UpdateDB Manager
	UserInteraction Manager
	QueryDB Manager
Data Elements	Sensor Information
	Fault Information
	Alarm Information
	Alarm Diagnosis
	Fault Diagnosis
	User Request
	Query Answer
Connector Elements	Sensor Connector
	FaultDetectionAlarm Connector
	UpdateDB Connector
	QueryDB Connector

Table 2: Properties Results in The Constraint View

Elements	Receive	Issue	Check
Sensor Manager (S_M)	Field Data	S_I	Data Correctness
FaultDetection Engine (FD_E)	S_I	F_I, F_D	Sanity, Consistency
Alarm Manager (A_M)	F_I	A_I, A_D	Fault Detected
UpdateDB Manager (UDB_M)	A_I, A_D, S_I, F_D	-	-
UserInteraction Manager (UI_M)	User Operations	U_R	-
QueryDB Manager (QDB_M)	U_R	Q_A	-
Sensor Information (S_I)	S_M	FD_E	Sanity, Consistency
Fault Information (F_I)	FD_E	A_M	Fault Detected
Alarm Information (A_I)	A_M	UDB_M	Fault Detected
Alarm Diagnosis (A_D)	A_M	UDB_M	Alarm Transmitted
Fault Diagnosis (F_D)	FD_E	UDB_M	Fault Detected
User Request (U_R)	UI_M	QDB_M	-
Query Answer (Q_A)	QDB_M	UI_M	-
Sensor Connector (S_C)	S_M	FD_E	Data Correctness
FaultDetectionAlarm Connector (FDA_C)	FD_E	A_M	Sanity, Consistency
UpdateDB Connector (UDB_C)	S_M, FD_E, A_M	UDB_M	Secure, TimeConstraint=2s
QueryDB Connector (QDB_C)	UI_M	QDB_M	TimeConstraint=5s

When applying our UML metamodel as a fine-grained way, we further defined all of these ADDs as objects for the corresponding UML classes. We finally derived seventeen objects for the elements, three objects for the architectural/design style and patterns, and about fifty objects for the rationale during the entire architecting process. The contents in Table 2 are

Table 3: Questions For Establishing The Intent View

Rationale	(**Motivation**) What is the motivation to establish the monitoring system?
	(**Alternatives**) How can we get the six computation elements?
	(**Reasons**) Why do we need the computation element "FaultDetection Engine"?
	(**Trade-offs**) What is the trade-off between using "Sensor Manager" or not?
	(**Justifications**) How to justify "Alarm Manager" works according to the requirements?
	⋮
Best-Practices	(**Architectural styles**) What kind of architectural style we can use to establish the system?
	(**Architectural patterns**) Is the layers architectural pattern applicable to the system?
	(**Design patterns**) Is there any design pattern we can adopt to design the system?
	⋮

transferred as the values of attributes in the element objects.

When we first applied the UML metamodel, we got the initial version of each object in every class. During the evolutionary change, the ADDs in the element, the constraint, and the intent view will also be changed, and should be tracked and updated explicitly. With the UML metamodel, we do not need to update evolutionary changes from the Triple View Model, but only add new objects for specifying the corresponding changes. As in the case study we did for the TVM [6], some new reliability requirements are added to the system afterward. One requirement is that "once a fault is detected by the FaultDetection Engine, the alarm should be raised within 5 seconds", which is a new limitation included in the requirement specifications. Based on this newly added requirement, we need to introduce a new object for the *FaultDetectionAlarm* connector element that evolves from the previous one. The evolutionary change is shown in Figure 7.

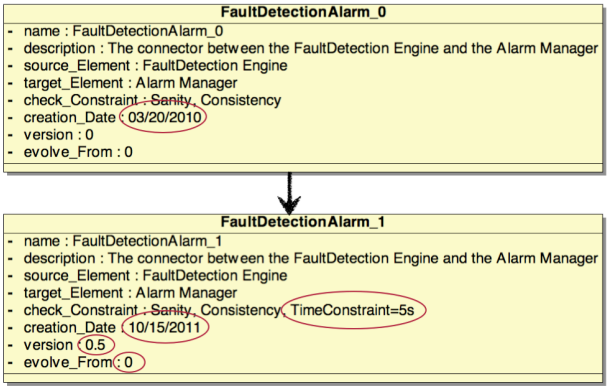

Figure 7: An Evolutionary Change

In Figure 7, we can see that a new object *FaultDetectionAlarm_1* is added based on the changing requirement. The value of the attribute *check_Constraint* has been changed, and the *creation_Date* and the *version* respectively specified the new creation date and the version number of this object. We trace the evolution from the attribute *evolve_From*, and the evolution chain provides us the snapshot of the evolutionary history. Most of the time, the ADDs in the intent view evolve as well if the decisions in element view or the constraint view change. Hence we also need to document the reason and the justification as ADD objects, in order to specify the rationale for the time constraint of the *FaultDectectionAlarm* connector. In this way, we explicitly record how the *FaultDetectionAlarm* connector evolves in the architecting process and the rationale behind the new decision during the decision making process.

Due to the space limitations, we will not specify all the possible decisions in the architectural evolution. Using the UML metamodel we propose, we can effectively manage the evolutionary change of architectural design decisions. The stakeholders in the software development process can share consistent architectural knowledge on ADDs evolution without knowledge evaporation.

5 Related Work

The key concepts of the traditional view on software architecture are components and connectors [2], [18]. Nowadays, software architecture has been seen as a set of architectural design decisions [3], [13], [19]. The architectural decisions in the software architecting process are increasingly focused by researchers and practitioners [11], [15], and architectural design decisions are also considered to be a part of architectural knowledge [16]. In [10], a systematic review for architectural knowledge is presented, and different definitions on architectural knowledge and how they are relevant to each other are discussed as well.

Recently, the research on managing the evolution of ADDs has been focused in the software architecture area. A number of models and tools have been proposed for ADDs evolution management. In [12], an approach for assisting architects in reasoning architectural evolution paths has been described, and the concept of evolution style is defined in it. Some other techniques as discussed in [7] and [22] introduce different approaches for capturing architectural evolution and selection architectural evolution alternatives.

Our work presents a complete documentation of ADDs, specifically focusing on ADDs evolution management. Comparing the research work related to ours, the UML metamodel we proposed not only supports the documentation of ADDs evolution, but also enables us to trace the evolutionary changes. Moreover, the UML metamodel captures ADDs on rationale as well; therefore, the tacit architectural knowl-

edge in the architects' mind are explicitly recorded in order to keep the knowledge from being evaporated.

6 Conclusions and Future Work

A recent strand of software architecture research is that software architecture is considered as a set of architectural design decisions. Managing the evolution of ADDs helps to maintain consistency between requirements and the deployed system, and is also necessary for reducing architectural knowledge evaporation. In this paper, we propose a UML metamodel incorporating key evolution-centered characteristics to manage the evolution of ADDs. The goal of the UML metamodel is to ensure that the architectural knowledge on the evolutionary changes of ADDs can be recored and traced in a systematic way, in order to reduce architectural knowledge evaporation during the architecting process.

Our ongoing work is devoted to conducting more extensive evaluation on our UML metamodel. Furthermore, we plan to provide tool support to enable the practical application of the UML metamodel.

References:

[1] http://www.enel.com/en-GB/.

[2] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley, Boston, MA, USA, 1998.

[3] J. Bosch. Software architecture: The next step. In *EWSA*, pages 194–199, 2004.

[4] R. Capilla, F. Nava, S. Pérez, and J. C. Dueñas. A web-based tool for managing architectural design decisions. *SIGSOFT Softw. Eng. Notes*, 31, September 2006.

[5] R. Capilla, F. Nava, and A. Tang. Attributes for characterizing the evolution of architectural design decisions. *Software Evolvability, IEEE International Workshop on*, 0:15–22, 2007.

[6] M. Che and D. E. Perry. Scenario-based architectural design decisions documentation and evolution. In *ECBS*, pages 216–225, 2011.

[7] S. Ciraci, H. Sözer, and M. Aksit. Guiding architects in selecting architectural evolution alternatives. In *ECSA*, pages 252–260, 2011.

[8] A. Coen-porisini and D. Mandrioli. Using trio for designing a corba-based application. *Concurrency and Computation: Practice and Experience*, 12:981–1015, 2000.

[9] A. Coen-Porisini, M. Pradella, M. Rossi, and D. Mandrioli. A formal approach for designing corba-based applications. *TOSEM*, 12:107–151, April 2003.

[10] R. C. de Boer and R. Farenhorst. In search of 'architectural knowledge'. In *SHARK*, pages 71–78, 2008.

[11] J. C. Dueas and R. Capilla. The decision view of software architecture. In *ECSA*, pages 222–230, 2005.

[12] D. Garlan, J. M. Barnes, B. R. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. In *WICSA/ECSA*, pages 131–140, 2009.

[13] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *WICSA*, pages 109–120, 2005.

[14] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *WICSA*, pages 4–, 2007.

[15] P. Kruchten, R. Capilla, and J. C. Dueñas. The decision view's role in software architecture practice. *IEEE Softw.*, 26:36–42, March 2009.

[16] P. Kruchten, P. Lago, and H. V. Vliet. Building up and reasoning about architectural knowledge. In *QoSA*, pages 43–58, 2006.

[17] D. E. Perry. Issues in architecture evolution: Using design intent in maintenance and controlling dynamic evolution. In *ECSA*, pages 1–1, 2008.

[18] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17:40–52, October 1992.

[19] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[20] D. Tofan. Tacit architectural knowledge. In *ECSA Companion Volume*, pages 9–11, 2010.

[21] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Softw.*, 22:19–27, March 2005.

[22] A. Zalewski, S. Kijas, and D. Sokolowska. Capturing architecture evolution with maps of architectural decisions 2.0. In *ECSA*, pages 83–96, 2011.