Preserving Architectural Knowledge Through Domain-Specific Modeling

Femi G. Olumofin and Vojislav B. Mišić *
University of Manitoba, Winnipeg, Manitoba, Canada

Abstract

We investigate the feasibility of applying the principles of domain-specific modeling to the problem of capturing and preservation architectural modeling knowledge. The proposed solution is based on the existing architecture assessment methods, rather than architecture modeling ones, and it uses sequences of design decisions, rather than simple, unordered sets. We highlight how architecture-based development could benefit from the proposed approach in terms of both methodology and tool support.

1 Introduction

For many years, the two main objectives of software development were (and still are) how to improve the productivity of the development process and how to improve the quality of the resulting products. A recent addition to the wealth of techniques proposed to address those objectives is the approach known as domain-specific modeling (DSM). The trademark feature of DSM is the shift from the traditional practice of specifying problem solution using lower-level programming constructs, to the one at a higher level of abstraction in which the solution is spelled out in terms of concepts culled from the very domain of the problem being solved. A metamodel of the problem domain is constructed by an expert; subsequent development by less experienced developers uses this metamodel and the associated automated tools to build actual solutions. Significant improvements in software development productivity (and, to a lesser extent, software quality) can be achieved in this manner, mostly because the complexity is limited by focusing on a single, well defined problem domain [11].

In this paper, we investigate the feasibility of applying domain-specific modeling to software architecture – a highly abstract area which should be well suited for the DSM approach. Traditionally, the architecture of a software-intensive system presents a structure (or structures) comprising the elements of the system, their externally observable properties, and static as well as dynamic relationships between those elements [1]. The architecture of a system is the first, and arguably the most important, set of artifacts that sets the directions for subsequent product development and controls the quality attributes of the final product. Architecture is usually developed by established experts from both the problem domain and general system domain. The domain experts ensure that quality goals such as performance,

*The work presented in this paper was partially supported by the NSERC Discovery Grant.

reliability, security, and availability, among others, are properly addressed during the architecture definition stage of the metamodel development. The system experts cater to the requirements of feasibility and development efficiency, as well as maintainability and evolvability of the architecture and derived systems. Obviously, the process involves high-level modeling and, possibly, metamodeling. Thus, it is worth considering whether an architecture metamodel could be developed to aid in this process, and what would be the benefits (and, possibly, drawbacks) of this approach.

The remaining part of the paper is structured as follows: Section 2 demonstrates the attendant problems of current architecture modeling practice through a small example. Section 3 explores how we should model software architecture to be able to address those problems. Some implementation issues and possible uses of this approach are discussed in Section 4. Finally, Section 5 concludes this paper.

2 Components and Connectors Are Not Enough

The field of software architecture is maturing. Several modeling methodologies for software architecture development have emerged, most of which represent the concepts of components and connectors as first-class entities of the design. While those concepts are certainly closer to the solution domain than the problem domain, they did help build highly successful architectural models for different kinds of systems. Documentation-wise, architectures are typically documented as sets of views that correspond to different aspects of the architecture [2]. But despite such advances, maintenance and evolution of an existing software architecture is still a difficult and error-prone task.

A possible explanation for this is the well known concept of knowledge vaporization [4, 8, 12]. Namely, in the development process, explicit artifacts are constructed for the architecture by making a series of design decisions aimed at satisfying the requirements whilst addressing the underlying quality goals. In this process, thorough knowledge about the problem domain is gradually accumulated, but the bulk of it remains implicit in the minds of the architect(s) rather than being captured and explicitly documented. Once the development is finished, architects move on to new tasks and this knowledge fades from memory. The relevance of explicit documentation also tends to diminish, on account of changes in the requirements and/or actual implementation.

As an example, let us consider the runtime architectural view of a *prepaid* cell phone airtime top-up application which is shown in Fig. 1. The customer uses her *mobile client* to send an airtime top-up instruction in text format to the telco's *SMS_Center*. This text message is retrieved by the *TextSender* and forwarded to *TextListener*, or temporarily stored in *Telco_DB* if *TextListener* can't be reached at the moment. *TextListener* forwards this message to *TextParser*, which processes it and delivers it (together with appropriate status information) to the *Scheduler* that invokes the necessary transaction processing components *TX_proc*. The result of this processing, which involves interaction with the customer's bank, is sent to *TX_Notifier*, which formats the response and notifies *TopupWriter* to credit the customer's airtime account and dispatch an appropriate status message back to the customer.

Subsequent evolution and maintenance activities are based on the existing architectural artifacts, rather than on the knowledge that guided the design choices in the development phase. In the example above, neither the textual description nor the components-and-connectors diagram provide clues as to why the decision to temporarily log customer's instructions in *TelcoDB* was made in the first place.

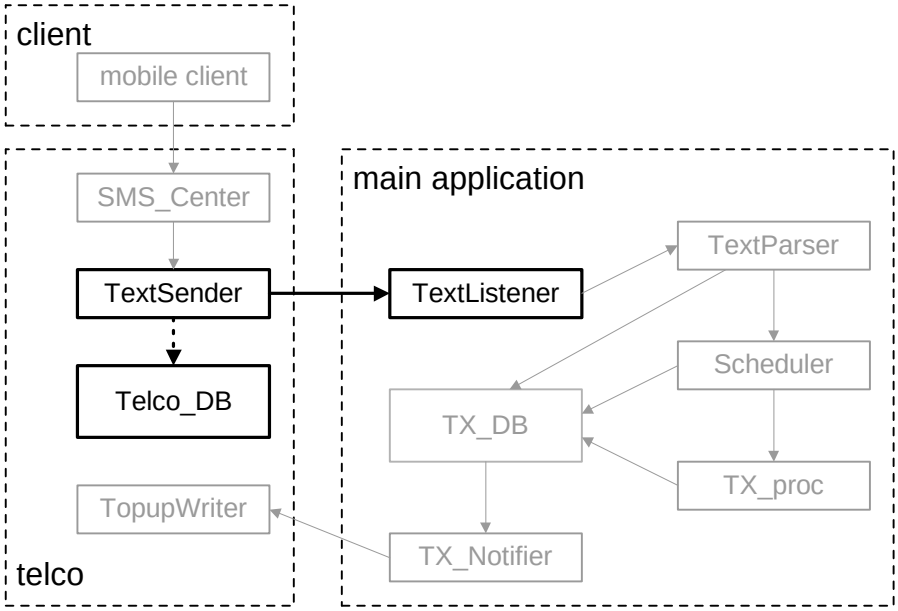

Figure 1. A runtime view of the airtime top-up architecture.

While implicit domain knowledge can be, and usually is, re-created when necessary, the process is neither quick nor easy, and it incurs a substantial risk of changing the architecture in ways that counter the initial quality goals, rather than support and align with them.

The main reason behind the loss of crucial architectural knowledge obtained in the design process and used to shape the design is simple: the current paradigm for describing software architecture models does not support that task. The only things that can be recorded are the final artifacts, but not the road that led to them. Question is, can domain-specific modeling help alleviate the problem(s) outlined above? The need to apply the domain-specific modeling paradigm to architectural modeling—for example, by representing architectural design decisions as first-class entities—has been recognized for a while [4]. In this context, a design decision is an abstraction that is closer to the problem domain than the solution domain. In fact, modeling software architecture in terms of design decisions follows a direction which is strikingly similar to the common DSM practice. The common activities of selection and interconnection of components and connectors (both of which firmly based on the solution domain) are, then, just a necessary, but by no means sufficient, ingredient of architectural modeling.

But how do we arrive at such a modeling technique? And how do we allow architectural knowledge to be captured explicitly within the model definition? According to [8], the solution is a three-step process. First, we need to devise a way to move architectural knowledge from the tacit level to the documented level. Second, we should create a formal framework for representing and reasoning about design decisions, just as has been done for components and connectors. Finally, tools should be developed to exploit a repository of such knowledge representations in order to simplify architecture evolution activities. Furthermore, there is perhaps use for an ontology of architectural design decisions that should record, store, and provide information about the varieties of design decisions and the connections between them [7].

A different solution has been advocated in [4], where architectural models are represented

as a *set* of design decisions. The relationship between a software architecture and associated design decisions uses a metamodel called archium, which consists of a combination of deltas, architectural fragments, and design decisions. A delta is used to track changes in component behaviors, whilst an architectural fragment is used to scope design decisions to some set of interacting components, possibly including appropriate deltas. Although the archium metamodel development is still at the preliminary stage, it appears fair to say that it is complex to understand, and a more simplified modeling approach might be more appropriate.

Some of the issues related to representing, capturing, and using design rationale in the generic context of design rationale systems have been discussed and systematized in [9].

3 Improving the Architectural Modeling Practice

Obviously, domain-specific modeling has a role to play; but architecture-specific domain concepts have to be identified and refined to suit the task. Two main theses appear to be valid in this context.

Our first thesis is that *architecture assessment methods, rather than architecture development methods, should be used as the foundation for domain modeling of software architecture designs and related design knowledge*. This is due to the fact that architecture assessment methods, a number of which have evolved over time [3, 5, 6, 10] already focus on architecture quality attributes and related design decisions. In fact, the most complete corpus of knowledge about the design decisions that lead to an architecture usually emerges as a by-product of the assessment exercise.

These design decisions are documented and collected in the form of architectural approaches that correspond to different views. These decisions are also explored in detail to assess the fitness of a particular architecture to its stated requirements and quality goals. In this process, attendant risks, nonrisks, and sensitivity points are identified and analyzed. (In this context, sensitivity points are design decisions that shape the architecture in order to address one or more quality attributes; tradeoff points are sensitivity points where two or more quality attributes interact; finally, nonrisks are the assumptions that do not affect quality attributes in the current design, but that could become risks should some alternative design decisions be applied.) One might almost say that we already have the tool we need; but it should be used in a way that differs from the usual one.

To illustrate this point, let us look again at the example from previous Section. The final outcome—that 'text message is retrieved by the *TextSender* and forwarded to *TextListener*, or temporarily stored in *Telco_DB* if *TextListener* can't be reached at the moment'—has been decided after considering and discussing the following rationale in the following order:

i. Physical communication link between *TextSender*and *TextListener*is unreliable, therefore recording customer's instructions is necessary to avoid loss of messages in case of link failure. This decision is a sensitivity point that stems from an attempt to satisfy the quality goal of reliability.

ii. On the other hand, customer's instructions contain confidential data such as PIN issued by their bank – keeping them in the database constitutes a security risk. The decision is, then, a sensitivity point related to the quality goal of security.

iii. Since the two quality goals affected offer conflicting suggestions, we are dealing with a tradeoff point. Proper decision can only be reached by finding the optimum tradeoff between quality attributes [6].

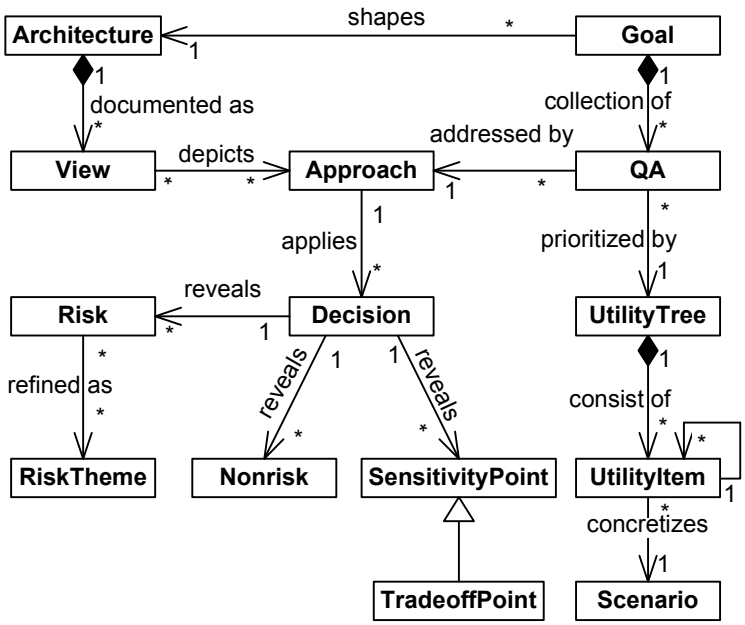

Figure 2. ATAM-based meta-model of architectural knowledge.

iv. Fortunately, *Telco_DB* is based on a particular DBMS technology that allows encryption of data items kept in a particular data area. Such items may be labeled with a expiration timeout, after which they 'disappear' from the database. A security breach of the *Telco_DB* can compromise only a limited number of such messages. This notion, then, diminishes the impact of the security risk, effectively converting it into a non-risk.

v. As a final precaution, the customers should be made aware of the fact that every instruction they issue has a predefined validity period; instructions not processed by this time will be dropped without notice. This decision is related to the quality goals of usability and reliability.

All of these decisions could easily be recorded using a suitable meta-model, and thus made available for subsequent use and refinement in the process of maintenance and evolution. The meta-model shown in Fig. 2, which draws its motivation from software architecture assessment techniques such as ATAM [6], could serve as a good starting point.

Here, we emphasize the words 'starting point'. Although techniques such as ATAM provide a good foundation for the development of a domain-specific approach to architecture modeling, they still need to be modified to maximize their effectiveness. First, facilities for capturing design decisions as first-class entities have to be present. Second, outputs generated by these methods have to be re-structured accordingly. Finally, domain-specific modeling has to be applied as early as possible – i.e., right from the start of architecture development. (As a bonus, the task of architecture assessment will be noticeably simplified if all relevant design rationale are properly recorded.)

Yet even a highly formalized meta-model such as the one shown in Fig. 2 may not allow for accurate and usable recording of *all* kinds of design knowledge. The reason is that this model captures and represents design decisions in a static manner which is inappropriate—or, rather, insufficient—for capturing the essence of the *process* through which the actual

model is developed. Our second thesis is that the *shift to a dynamic representation of design decisions would allow for more productive architecture development, and at the same time facilitate subsequent maintenance and evolution*.

In other words, modeling architectures should employ a *sequence* of design decisions, rather than a simpler set structure without any temporal dependencies between its elements. The rationale for this choice is simple: first, the interdependencies do exhibit certain temporal ordering, and second, the introduction of such ordering would facilitate and promote tool-based manipulations, whilst allowing for other uses of the architecture. In this manner, the focus of architecture evolution activities can shift from secondary, descriptive artifacts (i.e., documentation) to the more appropriate target: the sequence of design decisions that led to the development of the said architecture in the first place.

4 Implementing and Using the DSM

We are currently investigating possible ways in which the knowledge about the design decisions can be formalized. A promising avenue seems to be the view/viewpoint/perspective approach described in [13]. However, a new 'knowledge' view would need to be created, simply because the views described so far are unable to capture architectural knowledge for the purpose of guiding the activities of maintenance and evolution at a later time. The knowledge view would seek to explicitly represent design decisions as first-class entities, including their interdependencies, alternatives, constraints, rules, assumptions, and rationales, and to do so in a temporal ordering, thereby making it amenable to machine conversion and manipulation. The tasks of documenting design activities should be made an integral part of the development process, and automated as much as possible.

We note the well known developers' reluctance to document design decisions during development [8], and architects are no different than ordinary developers in that respect. Moreover, the introduction of another type of documentation may appear to be a step in the wrong direction. Providing automated tool support may also prove be a risky step, the more so because many such attempts in the past have failed to provide the promised benefits and gain wider acceptance. In authors' opinion, both of these facts may be attributed to the fact that documenting, manual or automated, is not made a transparent part of the design process. As long as documenting is perceived as an additional activity of secondary importance, developers will tend to focus on higher priority activity of design. Therefore, tool support should be an integral part of the design process, and it should be made as transparent as possible. Domain-specific modeling offers the benefit of having the design and documentation activities tightly integrated and supported though the same tool, such as MetaEdit+ [11].

Uses of architectural knowledge repository can be systematized through a number of use cases, similar to the concise description given in [8]. For example, a repository of temporally ordered architectural design decisions would allow the architects to gain better understanding of the architecture, to review the set of quality attributes which were considered in the process of forming the current solution, and even investigate alternative decisions that were not accepted. Developers can also benefit from being able to find out the path that led to system architecture, rather than just being handed the architecture with little justification. (This might essentially ease—or even remove—the 'ivory tower' syndrome which is a major objection to the conventional architecture modeling and development practices.)

In the context of domain-specific modeling, the possibility to review the design decisions made in the process of previous development is particularly important if the original domain

expert is no longer available.

Another important application of the repository of architectural design decisions is the training of architects and domain experts. In both cases, availability of a repository of architectural knowledge with a suitable tool interface would result in improved training and, ultimately, in the increase of the level of expertise available for the task.

5 Conclusion

In this paper, we outline an approach for applying the domain-specific modeling paradigm to the task of modeling architectural knowledge. We propose to model architectural knowledge and the resulting artifacts as a sequence of design decisions, and argue that this approach offers possibilities above and beyond what the current approaches are able to provide.

We are currently exploring ways to adapt existing architecture assessment methods in order to capture all required architectural knowledge for successful model evolution and training purposes.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, MA, 2nd edition, 2002.

[2] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[3] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures – Methods and Case Studies*. Addison-Wesley, Reading, MA, 2002.

[4] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proc. WICSA'05*, pages 109–120, Pittsburgh, PA, Nov. 2005.

[5] R. Kazman, L. Bass, M. Webb, and G. Abowd. SAAM: a method for analyzing the properties of software architectures. In *ICSE '94: Proc. 16th Int. Conf. Software Engineering*, pages 81–90, Sorrento, Italy, 1994.

[6] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proc. ICECCS '98*, pages 68–78, Monterey, CA, Aug. 1998.

[7] P. Kruchten. An Ontology of Architectural Design Decisions. In J. Bosch, editor, *Proceedings of the 2nd Workshop on Software Variability Management*, Groningen, NL, Dec. 2004.

[8] P. Kruchten. Building up and exploiting architectural knowledge. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120, 2005.

[9] J. Lee. Design rationale systems: Understanding the issues. *IEEE Expert: Intelligent Systems and Their Applications*, 12(3):78–85, 1997.

[10] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[11] J.-P. Tolvanen. MetaEdit+: domain-specific modeling for full code generation demonstrated [GPCE]. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 39–40, 2004.

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

[13] E. Woods and N. Rozanski. Using Architectural Perspectives. In *WICSA 5: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, Pittsburgh, PA, Nov. 2005.