

Attributes for Characterizing the Evolution of Architectural Design Decisions

Rafael Capilla, Francisco Nava
Department of Computer Science
Universidad Rey Juan Carlos
C/ Tulipán s/n, 28933,
Madrid, Spain
 {rafael.capilla,francisco.nava}@urjc.es

Antony Tang
Swinburne University of Technology
Faculty of ICT
Melbourne Vic 3122,
Australia
atang@ict.swin.edu.au

Abstract

Software architecture has been widely used to describe the design of a software system. Its maintenance over time can be costly, especially when maintainers have to recover software architecture knowledge due to poor design documentation. Capturing design decisions is one important aspect in documenting design and even though there has been some work in this area, there has been little emphasis on the evolution of design decisions. In this paper, we analyze design decision models and the issues of not capturing evolving decisions. To tackle these issues, we propose a set of decision attributes that can be used to support evolving decision models.

1. Introduction

Lehman [15] first recognized in the software engineering community that all software systems evolve. Such evolution is seen as a response to the changing needs and requirements of customers [16]. Moreover, the software development process influences the evolvability of any software system significantly [21]. As such, there is a need to have a software development approach that supports flexible and adaptable software evolution [8]. There are different reasons why software systems may evolve. For instance, new customer requirements such as incorporating a new feature, meeting a quality requirement, or changes to the system due to system malfunctioning, would lead to some kind of maintenance.

System evolution affects software artifacts such as requirements, architecture design and source code, and they should be synchronized with the changes, but often the changes that are made to the source code are not reflected in the architecture design. It is also infrequent to see the reasons behind design decisions being

documented for the system modifications. Therefore, over time a system can become costly to maintain as architects may have to recover the architecture descriptions from code. These issues are partially caused by losing design decisions when the architecture was created for the first time or during the evolution of the system. This paper analyzes the current design decision models and identifies the lack of support for evolving design decisions. In this work we propose two sets of attributes to enhance such decision models: (a) capture information about the evolution of design decisions and (b) capture their relationships with design artifacts. The structure of the paper is as follows. Section 2 introduces design decisions in software architecture. In section 3 we outline existing decision models with supporting tools. Section 4 describes attributes for capturing the design decision evolution and we conclude in section 5.

2. Design Decisions in Software Architecture

To date, software architecture development has mainly focused on the creation of the architecture using architectural viewpoints to represent the interests of different stakeholders [2]. Recently, the software architecture research community has regarded architecture design decisions as first class entities that should be recorded and maintained. As stated in [10], rationale is the justification behind decisions. This concept is based on earlier works in design rationale systems such as gIBIS [7], DRL [14] and QOC [17] where argumentation is used to analyse design decisions. Early in the 90s, Perry and Wolf [20] discussed the importance of the rationale for software architecture construction. Design rationale (DR) is a key element that has to be recorded [6] for documenting the reasons that led to a particular architecture, but design decisions are often not explicitly documented.

More recently, Bosch [3] stated that instead of using the traditional view of architecture as a set of components and connectors, we should see this “as the result of a set of architectural design decisions”. From the stakeholder perspective where different architecture views are used to describe the architecture of a system, a “new” architectural view, so-called the “decision view” was first proposed in [9]. The “decision view” comprises a set of attributes for representing and documenting architecture design decisions as a way to bridge the gap between requirements and architectural products (see also [25]). Consequently, recording the design decisions that lead to a particular architecture is of major interest to software architects and practitioners. We posit that the evolution process can decrease the maintenance effort if we are able to replay the decisions made by reducing the loss of architecture design information [22]. Other related work connecting the rationale to the evolution of software engineering processes can be found in [18] [19].

3. Design Decisions Models

Recently, a number of papers have been published on modelling design decisions in software architecture, but only a few explicitly deal with the evolution of the design decisions. As mentioned before, architecture design and its supporting design decisions should evolve together. The design rationale would be incomplete if decision models do not support design evolution. In this section, we analyze different design decision models from three perspectives: (a) attributes in a model that support design evolution; (b) support of design decision links to design artifacts through its life-cycle; (c) tool support for architectural design decisions.

3.1. Design Decision Evolution

Documenting design decisions is one of the major claims for improving the processes that are closely related to evolution problems. It was found in a survey that although design rationale is considered important, practicing architects do not document them systematically [22]. Tyree and Akerman [24] propose a template with a list of attributes for characterizing architecture design decisions. This template is used to capture the information associated to each design decision as well as to provide an architecture decisions model. However, there is little information in this model to record decisions evolution. Kruchten et al. [13] mention that architecture design is only a small part of the architecture knowledge (AK), and they advocate the concept of **AK = design decisions + design**. The authors distinguish four types of design

decisions according to implicit and explicit knowledge documentation. In addition, they have proposed a set of attributes to characterize architecture design decisions to provide some support for recording decision evolution. The current lack of agreement about which attributes would be relevant to characterize architectural knowledge is the main motivation for the authors in [5] to propose a set of mandatory and optional attributes that can be tailored for each particular organization. Also, a meta-model to support the decision model including additional attributes to manage the evolution of architecture design decisions are discussed [5]. For instance, some of these attributes are: Date and version, obsolete decisions (indicates the status of a decision when is out of date), validity (similar to the status but it can express whether a decision is right or wrong during the life of the system), reuse times / rating (it represents a score given for reuse purposes), and trace links (similar to dependencies but used for maintenance purposes under an evolutionary context). Nevertheless, the authors don't describe explicitly decision dependencies that are needed to manage the changes for maintenance and evolution activities, particularly when the modification or deletion of a decision has consequences to other related decisions, requirements or architecture artifacts.

3.2. Design Decision Links

Design decisions are useful explanations when they are associated with design artifacts such as requirements or design objects. As software architecture evolves, decisions, design artifacts and their relationships also evolve. New relationships may be formed and old relationships may be updated or replaced. The maintenance of such decision relationships with respect to the design artifacts is an important part of decision evolution. In this section, we discuss different decision models that support such links. The Architectural Rationale and Elements Linkage (AREL) approach [23] connects decisions to design artifacts through links. This model provides traceability between decisions and design artifacts through backward and forward tracing. The extension to AREL to support evolution is called eAREL. This model uses a UML stereotype labeled `<<ARsupersede>>` to express that an architectural rationale (AR) that justifies the creation of a design decision may have several versions. In this model, there is no information on why decisions have evolved from one version to the next.

In [13] and [26] the authors propose different types of dependencies that can be defined to clarify the type of relationship between design decisions. In [13], a

category of dependencies between design decisions is proposed. Up to 11 types of dependencies are discussed (e.g.: constraints, forbids, enables, subsumes, overrides, etc.), as well as the dependencies with external artifacts. In [26] a concern traceability map (CT-map) forms a net of architectural design decisions between software requirements and architectural concerns (typically identified during architectural evaluation). Decision evolutions are denoted by different types of relationships like: *refines-to*, *is-a-rationale-of*, *supports*, *impairs* and *is-dependend-upon-by*.

In this paper we will not distinguish whether a decision has been selected (a choice made) or not (an alternative decision) because all of them are decisions that are created and continuously evaluated during the evolution activities. We are aware that changing the status of a decision may have impact on the network of decisions. For instance, if a decision becomes obsolete, and such decision is removed, then that part of the network of decisions has to be reconstructed to maintain the integrity of the overall set of decisions. On the other hand, incomplete links imply that decisions are not fully traceable and may violate some constraint or dependency rules, resulting in inconsistent design.

3.3. Design Decision Tool Support

An overview of tool support for architecture evolution is described in [11], but only one of the tools mentioned in the paper exhibit a limited ability to record design decisions. Four of the five tools examined include some relationships between the architecture and their realization, while only two of them include features for supporting the evolution of products, such as: versioning, changes, or component replacement during runtime. In this section, we examine some existing tools that explicitly support design decision representation and their ability to track decision evolution.

Archium [12] is a Java tool for supporting architecture design decisions. It includes a compiler and a run-time environment. Archium supports the tracing of requirements to architectural decisions and checks implementation against architectural design decisions. If a decision is disregarded, the tool warns and prohibits violations of the decisions. Alternative decisions are supported in Archium to check whether the chosen alternatives have inconsistent consequences on the architectural model. Archium provides a visualization of an architecture decision by a dependency graph which helps the architect to evaluate the consequences of a decision. Checking for superfluous decisions and circularity of set of decisions is also provided (not automatically). Archium defines informal links between

decisions without describing formally the semantics of the relationship and it defines five types of relationships between a decision and a requirement.

PAKME [1] is a web-based architecture knowledge management tool built on the top of an open source groupware platform (Hipergate). PAKME supports external dependencies which can be queried to extract relevant conclusions. It captures relationships between decisions with timestamp, decisions history, and all the types of relationships described in [15]. In practice, it will support all the types defined in table 1 except the “excludes when” type.

ADDSS 1.0 [4] is a web-based tool for recording, managing, and documenting architecture design decisions under an iterative process. It supports a basic dependency model for relating decisions. ADDSS can establish complete traceability (backward and forward) between requirements, decisions, and architectures, but detecting inconsistencies of decisions is not implemented. The new ADDSS 2.0, which has been released, supports the status of decisions and the date when the decision was made. Also, it is planned to support versioning for recording and tracking the history of a particular decision and a model to detect the impact of changing or removing a decision.

Finally, the AREL approach is a model which relates decisions, architecture products and design rationale [23]. The Enterprise Architect tool is used to construct the AREL models and to capture design rationale by using extended UML profiles. The complementary approach eAREL [23] supports decision evolution and their tracing by means of versioning links. It supports backward and forward tracing through history. Each decision encapsulates its rationale, but there is only one link type, i.e. “depends-on”, defined in this method.

4. Evolution of Design Decisions

The works described in the previous section have largely omitted the decision evolution in the system life-cycle. This work is motivated to reduce maintenance effort by supporting evolvable AK. We are concerned with characterizing the evolution of architecture decisions. We distinguish between two types of evolution data: (a) those attributes that annotate decisions to record the history and the status of decisions at a given time, and (b) the attributes that record the relationships between design decisions and between design decisions and artifacts.

4.1. Evolution Support of Architecture Design Decisions

Evolutionary information is described by a set of attributes in this section. They are based on a combination of previous works [5] [13] [23] (such as we explained in previous sections) that support design evolution.

The following attribute list is used to enhance previous attribute templates with specific information to track the evolution of architectural design decisions. The attributes we propose to record and use are the following.

- **Date:** The date in which the decision was created or last modified.
- **Version:** The version number identifies the version of a decision. Because decisions are not recorded in files like software components or architecture documents do, i.e. a file may contain an image or description of the architecture design, we advocate using an internal version control system instead of version control systems like CVS. Also, CVS focuses on changes at the file level instead of its ability to record semantic information like design decisions. Otherwise, the integration with existing version systems and software configuration management (SCM) systems would be desirable.
- **Author / Responsible:** In most project brainstormings someone may prompt: *Who made this decision?* Therefore, knowing the responsible person is important to allow any follow-up with the person.
- **Status:** This is used to annotate the status of current decisions. The values can be: planned, pending, approved, rejected, and obsolete. For instance, knowing which decisions are obsolete supports their removal from the knowledge base in order to contain the amount of relevant information. Similar values for describing the state of the decision have been proposed in [13].
- **Validity:** This attribute describes if a decision is valid for a certain period of time until it becomes invalid. Under a maintenance context in testing and evaluation activities, selected decisions can be proven as right or wrong, and knowing this may take from hours or days (e.g.: testing) to weeks or months (evaluation on the customer site). Thus, the values we propose for this item are: valid, invalid, and being tested. Those decisions which have been proven valid should be marked “approved” with a specific date, version, and a responsible person.
- **Life-time:** Indicates when a decision should be reviewed (e.g.: after testing, after next iteration of

the architecture, etc.). This attribute can be used in combination with the validity attribute.

- **Scope:** This attribute provides the information on components and designs that are affected by the decision. Therefore, we will know which decisions have to be modified or which components and decisions have to be reviewed.
- **Evolution Reason:** This attribute documents the main cause(s) of change. This could be a reference to requirements change, or evolution due to some remote parts of the system being updated.
- **Derived Decision Flag:** In some cases a decision has to be refined during its evolution. Therefore, a decision may be composed by other sub-decisions that meet more detailed sub-goals in addition to the main one. Hence, new sub-decisions that refine the main one should have its derived-decision flag set.
- **To-be Architecture:** Sometimes architecture decisions are planned for the future but the implementation is pending at present. Some of the reasons are lack of project funding or it is a strategic decision awaiting suitable technologies. This attribute describes the criteria in which the to-be architecture is invoked, and it also describes the likely impact to the system.
- **Number of decisions and links:** This field is used to compute the number of decisions made and the links between decisions and between decisions, requirements and design objects of each version. By comparing these numbers over different decision versions, it provides a simple way to detect if a system is evolving to be more complex.
- **Number of changes in decisions:** This attribute allows us to track the number of decision changes over versions and to use such information to predict areas in the architecture that are prone to change.

4.2. Evolution of Design Decision Links

In the previous section we have introduced a set of attributes that we believe to be important for describing the evolution of design decisions. Recording such information is not enough because we also need to track the relationships among the network of decisions, as well as tracking decisions and their related artifacts (i.e.: requirements, architectures, code modules). Hence, these links are equally important in managing the evolution of the system over time.

The links between design decisions define a relationship between two or more evolving decisions. These links can be used to track the evolution of the decisions for managing the complexity of the system at a given time. Such links provide decision traceability

which is used to understand the impact over a set of decisions when other decisions are modified or even removed.

In addition, the links between requirements and design objects are defined for traceability purposes. As requirements change, design decisions are used as connectors that bridge the gap between requirements and design objects, so we could know, for instance, which design objects are affected by a change in a particular requirement. Backward and forward traceability can be tracked between requirements and architectures. These links should exist to provide traceability and to support future evolution of the system. As the architecture evolves, the complexity of a decision model, as represented by a graph, would change. Computing the number of links between decisions over different versions can help us understand if a product is becoming more complex. In this work we propose the following attributes to support the evolution of design decision links:

- **Versioning of links:** This attribute describes the version of the link with specific information that includes the date and the version and its associated artifacts.
- **Evolution of links between inter-related decisions.** Decisions are not isolated from other decisions. When we make a decision, the constraints from the requirements specification may limit the design choices in the decision space. Because decisions are inter-related, as the system evolves and more changes are made, the network of decisions becomes bigger and more difficult to trace. Furthermore, if a decision is no longer valid, the integrity of the decision network should be validated, through the links, to discover the impacts on other decisions. Relevant information about a link such as link name, date, link type, and link history should be captured.
- **Evolution of links between requirements and design objects:** Similar to the previous case, the evolution of a system may cause the relationship between requirements and decisions to change. For instance, when requirements change, the number of links may vary and they may become harder to track. In particular, nested decisions make it difficult for an architect or a system maintainer to track to the origins of that change. This attribute includes information to make tracking more meaningful: link name, the date, and the history of the link.
- **Link category:** This field is used to distinguish between different types of links (see Table 1). A structural link is a relationship between two or

more decisions and they define the network of design decisions. Structural links provide a view of inter-related decisions which is useful for maintenance purposes. A behavioral link describes a link between an artifact and a decision. A behavior link describes the relationship of two entities in a relationship, often used during runtime. For instance, in product lines, variability models are used to produce allowable product configurations, and to remove incorrect configurations. Hence, the “excludes when” link could depict such relationship. If a new requirement is incompatible with such condition or link, the evolution of the system can be dictated by the link type. The link types can be used by the designer to identify the nature of a relationship.

| Link Name | Link Type | Category | Purpose |
|--------------------------------------|-------------------|--------------------------|--|
| Depends on | Basic dependency | Structural | A dependency relationship between two or more decisions |
| Requires (strong) Enables (allow) | Need | Structural Behavioral | One decision needs form another to be successful. Composition can be seen a special case of need. Requires is a strong link, while enables is weak. For instance, if A and B are decisions, a require link means that for making B we need A, while the enable link means that is possible to make B even if A has not been made |
| Excludes | Incompatible | Structural Behavioral | A decision cannot be made if a link is connected to another decision which is incompatible |
| Excludes when | Incompatible when | Behavioral | Similar to the “excludes” relationship but only during runtime. The “when” implies a runtime condition |
| Special case | Exception | Structural Behavioral | An exception link specifies an exception decision case that deviates from a general decision case. For instance, a decision is made to use Java generally, but C# is used as an exception when developing web services. |

Table 1. Dependency types between decisions

In general, the semantics of the links between decisions can be extended with other link types but in order to simplify the selection of the link type during development or maintenance activities, we have tried to keep small the different link types. We have summarized many of these in [13], but we differentiate them by structural and behavioral links.

As we discussed before, we have put explicit emphasis on defining the attributes that can be used for tracking the evolution of architecture design decisions. We argue that this investment is justifiable because of the incremental productivity gains in future maintenance activities. To achieve this we will need specific tool support to record and track the design changes of a system as well as the decisions that lead to those changes. In the next section we discuss some lessons learned from two of the tools mentioned before to provide an insight to the proposed attribute list.

5. Lessons Learned from ADDSS and AREL/eAREL

The work described in this paper has partially been implemented in tools such as AREL/eAREL and ADDSS. However, there has not been any extensive empirical work on the capture and reuse of evolution information. Instead of providing empirical evidence as suggested in [27], we outline some lessons learned from the implementations as indicative results.

The implementation of ADDSS relies on an initial set of requirements and preliminary ideas described in the “decision view” [9] mentioned in section 2. Some of these requirements are the following: multi-perspective support for different architectural views, visual representation to understand and replay easily the decisions made, groupware support as the basis of a collaborative environments where distributed teams can interact each other (e.g.: for knowledge sharing purposes), and gradual formalization because the decision making process is a learning activity and decisions evolve over time. This basic set of initial requirements was extended and detailed in [4] to provide a complete list that lead to the construction of ADDSS 1.0. Basically, the tool follows a strict iterative process where one or several design decisions can be made for each of the iterations of the architecture. Decisions can be connected to other decisions by a basic dependency model (i.e.: “depends on” in table 1) and they can be modified as requirements of the target architecture / system evolve over time. In addition, the user can relate the requirements associated to one or several decisions and the architectural product which can be uploaded as the result of a set of decisions. Therefore, complete trace links between requirements and architectures can be established. At the end, explicit documentation in the form of PDF documents can be generated automatically containing the decisions made for each architectural product.

ADDSS 2.0, which has been released recently, implements some additional features such as described in [5]. For instance, support for multiple architecture

views is a feature implemented in the new version, which allows the storage of the decisions made for each particular view. Also, the status (i.e.: approved, pending rejected, obsolete) and a category (i.e.: main, derived, alternative) for each decision are stored as valuable information that can be used to describe part of the information of the evolution of the decision. The date and the responsible were implemented for the first time in version 1.0. For tracking the changes, root cause, and impact of the decisions made, the PDF documents that explicitly describe the rationale of the underlying design decision, describe now a chain with the dependencies between design decisions. Therefore, in addition to tracing between decisions, requirements and architectures can be better traced in the new version.

A preliminary evaluation was done in 2006 with ADDSS 1.0 at the University Rey Juan Carlos. Twenty-two master students organized in 11 teams who participated in the design of the architecture of a small virtual reality system. Some interesting results were observed. Most of the teams had to familiarize with the “new” process to record the design decisions in parallel with the modeling activities. Because no maintenance activity was done, we couldn’t evaluate the evolution of the decisions made, but most of the team members perceived important to store the attributes that describe the rationale of the decisions as they didn’t need to replay them during the successive iterations in the architecting phase. The average effort spent by the teams storing the decisions was around 10 hours additionally to the modeling process. For future maintenance processes, we should compare this overhead with the expected savings in which the decisions made could be reused. At this stage, we conjecture that the savings would be more significant for large systems rather than small systems.

An empirical study to assess the usefulness of behavior links supported by AREL has been performed. In this study, nine practicing architects who are familiar with a real-life electronic payment system were involved. The architects were asked hypothetical questions of which part of the design would change and why if certain requirements were to change. The results have shown that traceability with UML diagrams between design decisions to design artifacts is very useful to identify design changes and the reasons why. In some cases, inconsistency of the original design has been identified. However, the support of design evolution is yet to be implemented and tested.

6. Conclusions

Several design decision models and tools have recently been proposed to capture architecture design

decisions. In this paper, we have investigated four different design decision models and four existing tools to analyze the level of support for architecture evolution. We have found that these models and tools have limited support for architecture evolution. Only some of them provide certain attributes and dependency links that can be used to model design decision evolution. The lack of explicit support on the evolution of design decisions may render these models less than useful because design rationale may no longer be relevant or accurate after the system has evolved. We conjecture that evolution support for design decisions and decision links would help rectify this issue.

Based on previous work in design decision tools, we have enhanced and added new attributes to provide better support for maintaining design decisions over time. We have also proposed dependency links that better address the traceability of evolution of related design artifacts. In summary, we posit that these improvements would better support the evolution of design decisions during the maintenance phase.

The work for the future is to carry out empirical studies on the cost and benefit of employing such methods in software maintenance. We have not yet considered the complexity issue of decision network against the complexity of, for instance, the software products, but it seems an interesting approach to explore for the next future. We anticipate that this method would be useful to software systems that are complex and costly to maintain.

7. References

- [1] Babar, M. A. and Gorton, I. A Tool for Managing Software Architecture Knowledge. Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge, ICSE Workshops, IEEE DL (2007).
- [2] Bass, L., Clements P. and Kazman R. Software Architecture in Practice, Addison-Wesley, 2nd edition, (2003).
- [3] Bosch, J. Software Architecture: The Next Step, Proceedings of the 1st European Workshop on Software Architecture (EWSA 2004), Springer-Verlag, LNCS 3047, pp. 194-199 (2004).
- [4] Capilla, R., Nava, F., Pérez, S. and Dueñas, J.C. A Web-based Tool for Managing Architectural Design Decisions, Proceedings of the 1st Workshop on Sharing and Reusing Architectural Knowledge, ACM Digital Library, Software Engineering Notes 31 (5) (2006).
- [5] Capilla, R., Nava, and Dueñas, J.C. Modeling and Documenting the Evolution of Architectural Design Decisions, Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge, ICSE Workshops, IEEE DL (2007).
- [6] Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. Documenting Software Architectures. Views and Beyond, Addison-Wesley (2003).
- [7] J. Conklin and M. Begeman, "gIBIS: a hypertext tool for exploratory policy discussion," in Proceedings of the 1988 ACM conference on Computer-supported cooperative work, pp. 140-152 (1998).
- [8] Deridder, D. and D'hondt, T. "A Concept-Centric Approach to Software Evolution", Workshop on Ontologies as Software Engineering Artefacts – OOPSLA'04, Vancouver, Canada, (2004).
- [9] Dueñas, J.C. and Capilla, R. The Decision View of Software Architecture, Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005), Springer-Verlag, LNCS 3047, pp. 222-230 (2005).
- [10] Dutoit A., McCall, R., Mistrik, I. and Paech B. (Eds). Rationale Management in Software Engineering, Springer-Verlag (2006).
- [11] Jansen, A. and Bosch, J. Evaluation of Tool Support for Architectural Evolution, 19th International Conference on Automated Software Engineering (ASE'04), pp. 375-378, (2004).
- [12] Jansen, A., van der Ven, J., Avgeriou, P. and Hammer, D.K. Tool Support for Architectural Decisions, 6th Working IEEE / IFIP Conference on Software Architecture (WICSA 2007), pp. 4, (2007).
- [13] Kruchten, P., Lago, P., and van Vliet, H., T. Building up and Reasoning About Architectural Knowledge, QoSA2006, Springer-Verlag LNCS 4214, pp. 43-58 (2006).
- [14] Lee, J. and Lai, K. What is Design Rationale?. In Design Rationale - Concepts, Techniques, and Use, T. Moran and J. Carroll, Eds. New Jersey: Lawrence Erlbaum, pp. 21-51 (1996).
- [15] Lehman, M.M. "The Programming Process", IBM Research Report RC 2722, New York, (1969).
- [16] Lehman, M.M., Perry, D.E., Fernández-Ramil, J., Turksi, W.M. and Wernick, P. Metrics and Laws of Software Evolution – The Nineties view, Proceedings of Metrics'97, Albuquerque, NM, pp. 20-32 (1997).
- [17] Maclean, A., Young, R., Bellotti, V. and Moran, T. Questions, Options and Criteria: Elements of Design Space Analysis, in Design Rationale - Concepts, Techniques, and Use, T. Moran and J. Carroll, Eds. New Jersey: Lawrence Erlbaum, pp. 53-105, (1996).
- [18] Ocampo A. and Soto M. Connecting the Rationale for Changes to the Evolution of a Process. Proceedings of Product Focused Software Development and Improvement, PROFES 2007, Springer-Verlag, to be published (2007).
- [19] Ocampo A. and Münch, J. The REMIS Approach for Rationale-driven Process Model Evolution, International Conference on Software Process, ICSP 2007, Minneapolis, USA, LNCS 4470, Springer-Verlag, pp. 12-40, (2007).

- [20] Perry, D.E. and Wolf, A.L. "Foundations for the Study of Software Architecture", *Software Engineering Notes*, ACM SIGSOFT, pp. 40-52, (1992).
- [21] M.Pizka, A.Bauer, "A Brief Top-Down and Bottom-Up Philosophy on Software Evolution", *Proceedings of the Seventh International Workshop on Principles of Software Evolution(IWPSE'04)*, IEEE Press, (2004).
- [22] Tang, A., Babar, M.A., Gorton, I. and Han, J.A. A Survey of the Use and Documentation of Architecture Design Rationale, 5th IEEE/IFIP Working Conference on Software Architecture, (2005).
- [23] Tang, A., Jin, Y. and Han, J. A Rationale-based Model for Design Traceability and Reasoning, *Journal of Systems and Software* 80, pp. 908-934, Elsevier (2007).
- [24] Tyree, J. and Akerman, A. *Architecture Decisions: Demystifying Architecture*. IEEE Software, vol. 22, no 2, pp. 19-27, (2005).
- [25] van der Ven J.S., Jansen, A.G., Nijhuis, J.A.G., and Bosch, J. *Design Decisions: The Bridge between the Rationale and Architecture*. In *Rationale Management in Software Engineering*, pp. 329-346, Springer-Verlag (2006).
- [26] Wang, A., Sherdil, K. and Madhavji, N.H. ACCA: An Architecture-centric Concern Analysis Method, 5th IEEE/IFIP Working Conference on Software Architecture, (2005).
- [27] Zelkowitz, M.V. and Wallace, D.R. Experimental Models for Validating Technology. *IEEE Computer* 31(5), pp. 23-31, (1998).