

# Rationale promotes Learning about Architectural Knowledge

Torgeir Dingsøy<sup>1,2</sup>, Patricia Lago<sup>3</sup>, Hans van Vliet<sup>3</sup>

<sup>1</sup>SINTEF Information and Communication Technology, Norway

<sup>2</sup>Dept. of Computer and Information Science, Norwegian University of Science and Technology, Norway

<sup>3</sup>Vrije Universiteit, Amsterdam, The Netherlands

**Abstract.** In order to deliver software products with high-quality architecture, companies are dependent on managing their knowledge of the architecture well. Our impression of learning about software architecture today, is that it mainly consists of discovering and correcting weak spots in the architecture. We believe the architecture could improve faster if the learning processes were more thorough, and underlying rationale in the architecture uncovered. In this paper, we argue why double-loop learning in the context of software architecture is important, and show an example of such a learning process. We suggest a process to promote double-loop learning about architecture.

## 1 Introduction

Software evolves, whether we like it or not. For that reason, issues like comprehensibility, quality, and flexibility are important concerns. For that reason also, we not only bother about today's requirements during development but also, and maybe even more so, about the requirements of tomorrow.

This is one of the main reasons for the importance of software architecture, as for instance stated in (Bass et al, 2003): a software architecture manifests the major early design decisions. These early decisions determine the system's development, deployment, and evolution. Whether a design decision is major or not really can only be ascertained with hindsight, when we try to change the system. A priori, it is often not at all clear if and why one design decision is more important than another (Fowler, 2003).

Usually, the architecture documentation only contains the result of the design decisions, and not the reasoning behind them. The reasons for a design decision and other forces that drive those decisions are often not explicitly captured. They are tacit knowledge, essential for the solution chosen, but not documented. At a later stage, it then becomes difficult to trace the reasons of certain design decisions. The future evolutionary capabilities of a system can be better assessed if this type of knowledge is explicit (Bosch, 2004, Lago and van Vliet, 2005). We use the term rationale as a general denominator for the forces that drive architectural design decisions.

In this paper, we discuss how explicit rationale can be of help to a learning organization. Learning is paramount in an organization that strives to improve the

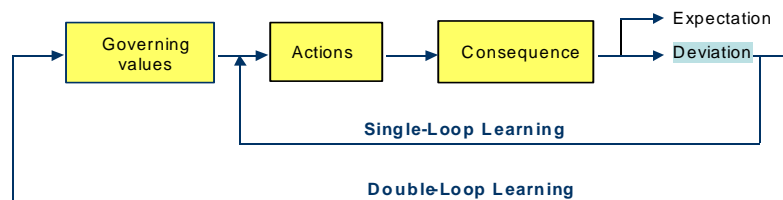
quality of its architectures. Making architectural decisions explicit is one step, but not enough. We also need to make the underlying rationale explicit and exploit these to reason and learn about the architecture and its qualities. A model for structuring architectural knowledge is given in (de Boer, 2005).

The paper is structured as follows: First we discuss learning and rationale in software architecture, then present approaches to managing architectural knowledge through strategies for knowledge management. We present an example of double-loop learning and a process aimed at promoting double-loop learning. Thereafter, we discuss the process for learning with respect to the example and general knowledge management strategies and conclude.

## 2 Learning and Rationale in Software Architecture

There are many models which explain learning in an organization. Nonaka and Takeuchi (1995) suggested the SECI-model, Wenger (1998) describes learning as a social phenomenon, happening in what he calls “communities of practice”. Others have studied learning by distinguishing learning after what effect it has on an organization. Argyris and Schön (1996) introduced the terms single- and double-loop learning. We will put special emphasis on the single and double-loop learning theory, as we think single-loop learning characterizes learning about software architecture today, and aiming for double-loop learning raises interesting perspectives on how to improve the learning effect.

A company usually has a set of norms for performance, strategies for achieving them, and some assumptions that bind norms and strategies together, what Argyris and Schön call the governing values.



**Fig. 1.** Argyris and Schön’s model of Single and Double-loop learning.

From studying an organization’s plans, documents, how it is organized, we are able to identify governing values. However, often other values than the ones espoused are used in practice, what is called the organizational action. This knowledge required to take organizational actions is often tacit. Argyris and Schön claim that learning happens when employees construct, test and reorganize the governing values and the organizational action. Learning is then called *single-looped* if it is sufficient to modify organizational action without questioning the underlying governing values. If discovering an error leads to norms being questioned and revised, the learning is called *double-looped*, because the error is connected both to organizational action and

to the governing values. Thus, double-loop learning has the potential for creating more thorough improvements than single-loop learning.

Bass et al (2003) use the term Architecture Business Cycle (ABC) to denote the observation that architects do not work in a vacuum. The architect is influenced by the requirements posed by stakeholders and the software development organization, by the technical environment, and by his earlier experience. These factors influence the architectural solution he devises, and experiences with this solution in turn affect the stakeholders, the development organization and the architect.

As a consequence, learning is primarily single-looped. To promote double-loop learning in software architecture, we need first to make the underlying rationale explicit in the documentation. Secondly, we need a method using this integrated documentation to guide the learning process. This paper discusses a method that promotes double-loop learning in software architecture by covering the two aspects above.

### 3 Managing Architectural Knowledge

Hanssen et. al. (1999) found two main strategies for managing knowledge: codification and personalization. Codification means to codify knowledge in such a form that it can be stored in databases and shared throughout a company. Personalization is a strategy depending on people as the knowledge carriers – information technology has the role of helping people communicate.

The nature of a company's business should decide on what strategy to follow. Codification supports widespread reuse of knowledge, which is typically suitable for companies offering standardized products, where development of the products is well-understood. Another factor which could lead to a codification strategy is if people working in the company rely on explicit knowledge to perform the work. It is likely that the need for a strategy will change over time, as the products the company produces gets more mature.

General techniques for supporting a codification strategy are to make use of processes to externalize tacit knowledge, such as post-mortem reviews of completed projects (Birk, et al., 2002) and interviewing experts (Eriksson, 1992).

Personalization strategies are supported through, for example, making overviews of the competence in a company, making arenas for knowledge exchange and mentor programs (Björnsson and Dingsøy, 2005) and supporting special interest groups (Wenger 1998).

In order to uncover tacit assumptions in architecture, there are two main methods to facilitate double-loop learning: technical analysis of the developed product and reflection amongst the designers. We will present and discuss each of these methods in the following:

**Technical analysis:** We can learn about the software architecture by studying explicit sources. This type of information may be available in three forms: through documents about architectural decisions, explicitly undocumented because of personal or company reasons, or explicit but undocumented – a decision is taken by an architect but the reasoning is lost. In the latter cases, we have to fall back to other

sources, such as CVS or source code, as described in the next section. This type of documents can be analyzed to find rationale. We can combine different explicitly available sources (what Nonaka and Takeuchi call “combination” in the SECI model), for example versions of architectural design documents or design documents and reference design documents.

**Reflection:** Reflection is a method to uncover tacit knowledge that exists in humans, what Nonaka and Takeuchi refer to as “externalization”. Argyris and Schön distinguish two types of reflection. Reflection *in* action is to think about your actions while performing them, reflection *on* action is to think through actions after they have been performed. Reflection in action can be facilitated through discussions when taking decisions. Involving people with different responsibilities in taking architectural decisions should lead to discussions which uncovers tacit knowledge from the stakeholders. Reflection on action can be promoted through asking “reflective questions”. In a project postmortem analysis, a common way to discuss the project is to find reasons for successes and failures. Discussing such causes will often show conflicts between company or technological policies and practice in projects. In order to focus on architectural decisions, a postmortem can be organized around two questions: 1. Which major technical decisions were right? 2. Which major technical decisions were not right? Greenwood (1998) lists several frameworks of questions for reflection on action in general. Software architecture analysis methods such as ATAM (Clements et al, 2002) promote reflection on action specifically for software architectural concerns.

### 3.2 An Example of Double-Loop Learning

Two major components can be identified in an application developed by “SalesPlus” (Roeller et al, 2006): the Business Manager which manages the operational aspects by sales managers, and the SalesPlus Core. The latter is further decomposed into a number of components, the three major ones being the full-text search, the query engine, and the promotion module. The first creates a list of items that match the user’s keyword. The second provides the browsing functionality with which the user can explore the catalogue. Finally, the promotion component manipulates the results of the other two components, depending on company interests.

Available data is analyzed for “suspicious effects”, and is input to guided interviews with the architects, which results in a list of assumptions about decisions made. We can analyze the Version Control System (CVS) archive and generate statistics. We may look for components that are changed most frequently, or components that have the largest number of authors. This revealed that a lot of people had been working on a component called Bridge, because each part of the system is, unofficially, assigned to one developer (assumption A). This made it virtually impossible to keep the interface component up-to-date, since none of the developers has a deep enough understanding of all parts of the system, and the responsible developers regularly forgot to update their part of the Bridge (assumption B).

The Business Manager component provides its users with the opportunity to change the business rules. Managers at SalesPlus assumed that companies would be willing to train their sales managers to use this web-based tool (assumption C). It

turns out that some clients plan to generate meta-tools that combine all subsystems they use under one interface.

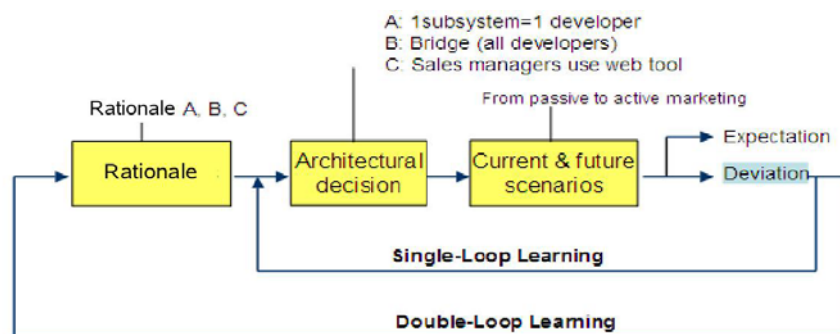
The rationale thus uncovered can be used to assess some of the challenges SalesPlus may be faced with in the future. An example hereof concerns changes in the promotion-search-report cycle. Early versions of SalesPlus provided support for passive marketing: end-users get optimal access to a catalogue and are free to choose from the offers. Lately, more and more opportunities for active marketing are provided, whereby sales managers can influence the behavior of their clients. For example, promotions to related products are shown. Or, more subtly, one may change the order of showing search results to influence buying behavior. The Business Manager component is a milestone in this development, since sales managers no longer need technical support. Eventually, a cycle is foreseen in which (1) the sales manager configures a promotion, (2) analyses the influences on sales after some days and (3) fine-tunes his promotions.

This scenario is threatened from two sides. If sales managers refuse to use the Business Manager (assumption C), it is a bad idea to purely bind the cycle functionality to the Business Manager. The other problem derives from the person-centric development (assumption A). On the one hand, the Business Manager couples functionality ever tighter; on the other hand the knowledge about subfields like search or promotions is spread over different people.

To promote double-loop learning in software architecture we challenge the architecture and enact the learning process through the following steps:

- (1) Analyze the impact of scenarios on architectural decisions;
- (2) Reason about the associated rationale, to identify if/how it may change;
- (3) Modify the rationale and/or the architectural decisions to accommodate the scenario. It is in this step that we choose between the single-loop and the double-loop learning. If the scenario can be accommodated by selecting an alternative architectural decision, but the underlying rationale remains invariant, the single-loop learning is followed. If instead the realization of the scenario impacts the rationale underlying the current architectural decisions, the double-loop learning is followed.

- (4) Update the overall documentation of architectural knowledge.



**Fig. 2.** Double-loop learning in software architecture.

We now apply these steps to the example presented above. Figure 2 is the customization of double-loop learning (shown in figure 1) to software architecture. In the example, rationale A-C are linked to the depending architectural decisions, and the analyzed future scenario regards the shift from passive to active marketing.

We then apply the four process steps to the model in figure 2:

1. Analyze the impact of scenarios on architectural decisions. In our example, the decision directly influenced by the future evolution scenario is the encapsulation of the logic needed by Sales Managers to do their job, in one component, the Business Manager. This realizes the promotion-search-report cycle. Therefore, in this scenario the Business Manager becomes even more important.

2. Reasoning about the associated rationale. The rationale associated with the Business Manager are rationale C and A. The realization of rationale C becomes fundamental. Rationale A conflicts with C: the Business Manager is likely to remain out of date as it depends on features that have no clear responsibility.

3. Modify the rationale and/or the architectural decisions to accommodate the scenario. The result is to eliminate the conflict between rationale A and C, and to strengthen the realization of rationale C.

4. Update the overall documentation of architectural knowledge. The new abstraction layer must be linked to rationale C as well.

This example shows how rationale reasoning drives learning and hence the quality of the architecture. For example, the new architectural decision not only accommodates the new scenario, but it also strengthens the realization of rationale C.

### **3.3 A Suggested Process to Promote Double-Loop learning**

A good process to increase double-loop learning should combine elements from technical analysis and reflection to uncover important rationale in an efficient manner. The most important ones are those that will have a negative or positive effect on product quality and product maintenance. A main source of this kind of knowledge will be the people involved in designing the architecture, but as the example in section 3.2 shows, analysis of for example CVS logs can also lead to learning about the architecture. We suggest the following process, inspired by Greenwood (1998):

1. Discuss pros and cons of the software architecture in the initial design meeting. Document major design decisions as well as their rationale.
2. Organize discussions about the quality of the architecture during development; especially focus on the architecture's unforeseen consequences for the development.
3. Analyze the architecture after implementation through technical analysis of source code, CVS logs and other explicitly available information – look for suspicious effects. This step focuses on architectural knowledge derived from the evolution of the system.
4. Gather the responsible people for software architecture in a workshop – examine initial pros and cons of the architecture, suspicious effects, and ask:
  - a. Which major decisions were right?
  - b. Which major decisions were not right?
5. Challenge the architecture and enact the learning process through:

- a. Analyze the impact of scenarios on architectural decisions;
  - b. Reason about the associated rationale, to identify whether/how it may change;
  - c. Modify the rationale and/or the architectural decisions to accommodate the scenarios.
  - d. Update the overall documentation of architectural knowledge.
6. Finally, analyze the cost and benefits of suggested architectural changes, and implement the most beneficial ones.

Steps 1, 2, 5 and 6 are usually in some form part of architectural assessments, as discussed in (Clements et al, 2002). Here, however, the focus is on the architectural decisions and the rationale that drives them, and the goal is to promote double-loop learning. Steps 4 and 5 focus on the architecture “in action”, i.e. on its evolution over time. The example in the previous subsection in particular elaborates step 5 of the above process.

## 4 Discussion

We have suggested a process to promote double-loop learning about software architecture, based on theories from knowledge management and lessons learned from a study in a small, agile software product company (Roeller et al, 2006).

How should companies use the suggested process? We think it is important to root the process in an overall knowledge management strategy for a company as discussed by Hansen et al. (1999). If the company is large, the product is intended to stay for a long time in the market, or the complexity or size of the system is large, more knowledge about architecture should be codified. If a company has few employees who are able to have an ongoing discussion about architecture, and the likelihood that people will leave the company is small, it is economical to codify less knowledge. However, given the technical nature of software architecture, this is probably a field that requires codification, and also this is by definition knowledge which is very important to the company, which is also an argument for codification.

What are the benefits and limitations of the suggested process? We think the main benefit is the potential of double-loop or radical learning, which should have substantial impact on the technical quality of the software system.

The suggested process combines reflection *before* action in step one and five, reflection *in* action in step two, and reflection *on* action in step four. The process also uses both people and documents as knowledge carriers.

An argument against processes such as the one proposed is that of resources – there is constant pressure from project work that makes it difficult to give priority to work involving reflection. In practice, not making room for planned reflection leaves only reflection *in* action, and where there is no systematic approach to handle ideas about the architecture that come up throughout the project, the learning effect is likely to be lower. Also, the cost of conducting all steps except for step three should be insignificant relative to the total cost of a software development project.

The process is a suggestion. As far as we know there are no studies of how transfer of architectural knowledge happens in software companies today. We hope the process we suggest here can be further developed as such studies appear.

## 5 Conclusion and Further Work

The architecture of software products is the most important asset for companies delivering software. We have argued why double-loop learning is important, and showed an example of such a learning process. We also suggested a process to promote double-loop learning about architecture in software companies.

The ideas brought forward in the suggested process to promote double-loop learning is based on two studies in two companies, general findings on learning and the authors experience with software architecture. Future work will be to study companies that organize activities to promote double-loop learning and study the actual effects of these efforts.

## References

- Argyris, C. and Schön, D.A., *Organizational Learning II: Theory, Method and Practice*: Addison Wesley, 1996.
- Bass, L., Clements, P. and Kazman, R., *Software Architecture in Practice (Second Edition)*, Addison Wesley, 2003.
- Birk, A., Dingsøy, T., and Stålhane, T., Postmortem: Never leave a project without it, *IEEE Software*, special issue on knowledge management in software engineering. 19(3), 43 – 45, 2002.
- Boer, R.C. de, Farenhorst, R., Clerc, V., van der Ven, J.S., Lago, P., van Vliet, H., *A Model for structuring Software Architecture Project Memories*, submitted, 2005.
- Björnsson, F. O. and Dingsøy, T., *A Study of a Mentoring Program for Knowledge Transfer in a Small Software Consultancy Company*, in *Proceedings of PROFES 2005*, Springer LNCS 3547, pp. 245 - 256.
- Bosch, J., *Software Architecture: the Next Step*, in *Proceedings First European Workshop on Software Architecture (EWSA 2004)*, Springer Verlag, 2004, pp. 194-199.
- Clements, P., Kazman, R., and Klein, M., *Evaluating Software Architectures*, Addison-Wesley, 2002.
- Eriksson, H., *A Survey of Knowledge Acquisition Techniques and Tools and Their Relationship to Software Engineering*, *Journal of Systems and Software*. 19, 97-107, 1992.
- Fowler, M., *Who Needs an Architect*, *IEEE Software*. 20(5), 11-13, 2003.
- Greenwood, J., *The role of reflection in single and double-loop learning*, *Journal of Advanced Nursing*. 27, 1048 – 1053, 1998.
- Hansen, M.T., Nohria, N., and Tierney, T., *What is your strategy for managing knowledge?*, *Harvard Business Review*. 77(2), 106 – 116, 1999.
- Lago, P. and van Vliet, H., *Explicit Assumptions enrich Architectural Models*, *Proceedings International Conference on Software Engineering (ICSE)*, 2005, 206-214
- Nonaka, I. and Takeuchi, H., *The Knowledge-Creating Company*: Oxford University Press, 1995, ISBN 0-18.509269-4.
- Roeller, R., Lago, P. and van Vliet, H., *Recovering Architectural Assumptions*, *Journal of Systems and Software* 79(4): 552-573, 2006.
- Rus, I. and Lindvall, M., *Guest Editors' Introduction: Knowledge Management in Software Engineering*, *IEEE Software* 19(3), 26-38, 2002.
- Wenger, E., *Communities of practice: learning, meaning and identity*. Cambridge, UK: Cambridge University Press, 1998, ISBN 0-521-43017-8.