Experience with ALMA

Nico Lassing (a), Daan Rijsenbrij (b) and Hans van Vliet (c)*
(a) Accenture, Amsterdam, The Netherlands
(b) Cap Gemini Ernst & Young, Utrecht, The Netherlands
(c) Faculty of Sciences, Vrije Universiteit, Amsterdam, The Netherlands

Abstract

We discuss our experiences in using ALMA, our method for Architecture-Level Modifiability Analysis. Like many other methods for software architecture analysis, such as SAAM and ATAM, our method is scenario-based. We found that the scenario elicitation process is tricky, and depends on the goal set for the analysis. Also, our experience in applying ALMA to business information systems indicates that it is important to also model the environment of the system. Data from a longitudinal study in which we collected information on the actual evolution of a system whose modifiability we analyzed earlier, provides further valuable insights into the applicability of ALMA, and points at some general limitations of this type of analysis.

1. Introduction

This paper is about ALMA, a method for Architecture-Level Modifiability Analysis (Bengtsson et al., 2000). Like many other methods for software architecture analysis, such as SAAM (Kazman et al., 1996) and ATAM (Kazman et al., 1998), our method is scenario-based, which means that it uses concrete scenarios to assess the quality of a system based on its software architecture. At some point in the analysis, these scenarios are elicited from the stakeholders, and the software architecture is analyzed to determine their impact. Since we are concerned with assessing the modifiability of the architecture, our scenarios describe future uses of the system, uses not yet accounted for in the requirements. We term these *change scenarios*, as opposed to operational scenarios which do refer to the requirements.

The remainder of the paper is divided into three sections. Section 2 gives an overview of ALMA following the format of the SARA Report. ALMA distinguishes several goals for modifiability analysis. The emphasis of the research carried out at the Vrije Universiteit has been on risk analysis. Section 3 discusses our experiences with this type of modifiability analysis. Section 4 summarizes our experiences.

*Corresponding author. Tel (31) 20 444 7768; fax (31) 20 444 7728
E-mail addresses: nico.lassing@accenture.com, daan.rijsenbrij@capgemini.nl, hans@cs.vu.nl

2. About ALMA

Name: ALMA (Architecture-Level Modifiability Analysis).

Context: Evaluation. ALMA supports part of the inception phase, viz. selecting the goal (risk assessment, maintenance prediction, comparison of architectures), but its main focus is on the actual review. The techniques applied during the actual review depend on the goal chosen.

Purpose: Assessing modifiability.

Input Architecture description.

Output Identification of changes that are difficult to accomplish, or an estimate of maintenance costs, or a comparison of architectures w.r.t. maintenance costs or risks.

Steps: ALMA has the following steps:

1. Set goal: determine the aim of the analysis (risk assessment, maintenance cost prediction, or software architecture comparison).
2. Describe the software architecture(s).
3. Elicit change scenarios.
4. Evaluate change scenarios.
5. Interpret results.

Roles: architect(s), one or two assessors, stakeholders.

Estimates: n.a.

References: The overall method is described in (Bengtsson et al., 2000). Experiences with the maintenance prediction 'version' are collected in (Bengtsson, 2002). Experiences with the risk assessment 'version' are collected in (Lassing, 2002). See also (Lassing et al., 1999), (Lassing et al., 2001), (Lassing et al., 2002a), (Lassing et al., 2002b).

Tools: none.

Alternatives: SAAM, ATAM, FAAM (aimed at supporting repeated assessments; more emphasis on process aspects of the assessment; see (Dolan, 2001))

3. Experiences

3.1. Viewpoints on Modifiability

A software architecture description usually consists of a number of views, where a view is a representation of a system from a certain perspective. At the software architecture level modifiability has to do with separation of functionality and dependencies, i.e. *how do we distribute the functionality over components?* and *how are these components related?* Allocation of functionality determines which components have to be adapted to realize certain changes and dependencies determine how changes to a component affect other components. In an architectural description in which modifiability is addressed these decisions should be made explicit. These decisions are generally adequately addressed in existing view models (such as the logical and development views in (Kruchten, 1995)).

The aforementioned questions focus on the system's internals. For business information systems it is not sufficient to study only the internals of the system. Such systems are rarely isolated; they are often part of a larger suite of systems. At the system's level questions similar to the ones at the component level recur: *how do we distribute functionality over systems?* and *what are the dependencies between these systems?*. Therefore, we split the description of a system's software architecture into two parts: (1) the *external architecture*: the software architecture at the systems level, and (2) the *internal architecture*: the software architecture of the internals of the system. The two viewpoints for the external architecture are:

- the **context viewpoint**: an overview of the system and the systems in its environment with which it communicates, and
- the **technical infrastructure viewpoint**: an overview of the dependencies of the system on elements of the technical infrastructure.

Viewpoints that depict the external architecture are often not explicitly mentioned and dealt with in architecture assessment methods. For instance, a relevant issue at this level is: who are the *owners* of the components? Changes involving different owners generally pose more risks; see also (Lassing et al., 2001).

3.2. Scenario Elicitation

In our experience, the change scenario elicitation process is tricky. The type of change scenarios we are interested in depends on the goal of our analysis. If our goal is to predict the future maintenance cost of the system, we are interested in identifying change scenarios that are likely to occur during the operational life of the system. If our goal is to identify the risks of the architectural choices made, we are interested in scenarios which are particularly difficult to accomplish. How then do we know we have elicited the right scenarios? How do we know we have identified enough scenarios? And, since the system we are investigating hasn't necessarily been implemented yet, how do we determine the impact of these changes on the system?

In our research, we are interested in identifying architectural risks. We are thus interested in *complex* change scenarios, scenarios describing changes that are particularly difficult to accomplish. To help structuring the scenario elicitation process, we use a framework with categories of change scenarios which we have found to be complex at the architecture level. This framework is used to classify change scenarios as well as to probe stakeholders to formulate change scenarios. In this framework, we distinguish the following categories of complex changes:

- Changes that involve different system owners. A system owner is an organizational unit financially responsible for the development and management of a system, or simply put the entity that has to pay for adaptations to the system. Changes are more complex if different owners are involved. Not only because of the additional coordination between parties, but also because all owners involved have to be persuaded to implement the necessary changes. We make a further distinction between changes initiated by the owner of the system under analysis, and changes initiated by others but require the system under investigation to be adapted.
- Changes that affect the architecture. These concern changes that affect the architecture, as opposed to changes affecting individual components only. A change affects the architecture if it results in the addition or deletion of one or more components or connectors, or the semantics of one or more components or connectors changes. Overhauling the architecture is risky.
- Changes that introduce version conflicts. Finally, changes are considered complex if they result in the presence of different versions of some architectural element. Different versions of an architectural element may introduce a number of difficulties. Eventually, this may require changes to elements that were initially unaffected by the change.

In (Lassing et al., 2002b), we discuss the validity of this framework; see also the next subsection.

3.3. Findings from Longitudinal Study

In 1999, we analyzed the modifiability of Sagitta 2000/SD, a large business information system being developed on behalf of Dutch Customs. We asked stakeholders to bring forward possible changes to the system, and next investigated how these changes would affect the software architecture. Since then, the system has been implemented and used, and actual modifications have been proposed and realized. Two years later, we revisited the Dutch Tax and Customs Administration, collected information on the actual evolution of the system, and compared this with the initial set of anticipated changes.

The input to this validation study consists of the change requests that were submitted since our initial study. During this period 117 CRs were submitted, which were stored in a reporting tool together with an analysis of their effort. Accepted CRs are incorporated in one of the system releases. It should be noted that these CRs are not the only source of changes realized; the releases also include changes that result from the planned evolution of the system and adaptations to the technical environment. Changes of the former type are not documented individually and changes of the latter type are documented in another way. So we only consider scenarios that relate to changes with a functional flavor.

The 117 CRs can be classified as follows:

- 61 CRS represent implementation bugs.
- 28 CRs concern the addition of new functionality because of changed circumstances.
- 6 CRs concern the addition of new functionality because these functions were not identified yet in the initial requirements analysis phase.
- 22 CRs concern errors in the initial requirements.

Of interest for our analysis are the 56 CRs from the latter three categories. In this analysis, we asked ourselves two questions: (1) were we able to *predict* complex changes, and (2) were we able to predict *complex* changes?

The short answer to the first question is: we missed quite a few. The two largest categories concern requirements that were initially not identified, and new functionality which incurred the addition of one or a few components. One CR was identified during the initial analysis, classified as not complex, but when it got actually implemented turned out to have a more dramatic impact; due to dependencies between components, the change required the structure of the system to be adapted. We had not identified these ripple effects in our earlier analysis. This experience corroborates findings reported on in (Lindvall and Sandahl, 1998) and (Lindvall and Runesson, 1998). The effect of one CR was not visible in the viewpoints we considered adequate for modifiability (see section 3.1 and (Lassing et al., 2001). Sagitta 2000/SD is based on the principle of one user per tax declaration. This CR asked to change the system such that large declarations can be processed by several staff members. The views of the system's architecture do not show this. Nevertheless, this is apparently an important architectural decision related to modifiability. We could of course extend the views with narrative text indicating the principle of one user per declaration, but how does one decide that this information is that relevant?

CRs classified as not complex by our method, but classified as complex by the expert we consulted, generally concern changes to complex components. This complexity can be functional – i.e. it concerns a component encapsulating complex domain functionality, and adapting such components requires the input of a domain expert, or the complexity is technical, and then it needs to be carried out by an experienced developer.

This study suggests several improvements to ALMA:

- A total of 28 CRs issued during the analysis period concern cases where the requirements were not entirely correct to begin with. Although this may be the result of the particular development process followed for Sagitta 2000/SD, we expect this issue to occur in the life cycle of other systems as well. Apparently, we have been a bit optimistic in our change scenario elicitation in assuming the initial requirements to be correct. This optimism is shared by other software architecture analysis methods; none of them addresses this issue explicitly. The architecture analysis should improve if we explicitly challenge the requirements we start with.
- Adding a few components to and deleting a few components from the system had better not be classified as changes to the system's internal architecture. This predicate should be reserved for 'bigger' changes, changes that affect the fundamental organization of the system.

The study also hints at a number of fundamental limitations of this type of analysis:

- Fundamental modifiability-related decisions need not be visible in the architecture descriptions available. Though we may improve our guidelines as to the documentation to be included in our viewpoints, they will never be complete, and the involvement of expert stakeholders remains important.

- We did not predict a number of changes. They may have been overlooked by stakeholders during scenario elicitation. Our scenario elicitation process in particular involved fewer stakeholders from the user side than would have been desirable. On the other hand, the actual evolution of a system remains, to a large extent, an unpredictable process.
- The complexity of a number of CRs is caused by the fact that they concern complex components. Even if we could measure the complexity of individual components at the architecture level, we might not be able to reduce this complexity. Some components are inherently complex.

A more elaborate analysis of the results from this longitudinal study can be found in (Lassing et al., 2002b).

4. Conclusions

Our experience with applying ALMA to a variety of systems can be summarized as follows:

- A business information system is rarely isolated. To assess the modifiability of such a system, we need views which depict that system as a component within a larger suite of systems.
- The stopping criterion for scenario elicitation is difficult. We have developed a framework with categories of change scenarios to help structure the elicitation process. This framework is helpful, but not a panacea.
- Architecture-level modifiability analysis considerably improves if we explicitly challenge the initial requirements.

References

Bengtsson, P. (2002). *Architecture-Level Modifiability Analysis*. PhD thesis, Blekinge Institute of Technology, Sweden.

Bengtsson, P., Lassing, N., Bosch, J., and van Vliet, H. (2000). Analyzing software architectures for modifiability. Technical Report HK-R-RES–00/11-SE, Högskolan Karlskrona/Ronneby.

Dolan, T. J. (2001). *Architecture Assessment of Information-System Families*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands.

Kazman, R., Abowd, G., Bass, L., and Clements, P. (1996). Scenario-Based analysis of software architecture. *IEEE Software*, 13(6):47–56.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, J. (1998). The architecture tradeoff analysis method. In *Proceedings of the 4th International Conference on Engineering of Complex Computer Systems (ICECCS'98)*, pages 68–78, Montery, CA. IEEE CS Press.

Kruchten, P. B. (1995). The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50.

Lassing, N. (2002). *Architecture-Level Modifiability Analysis*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands.

Lassing, N., Bengtsson, P., van Vliet, H., and Bosch, J. (2002a). Experiences with software architecture analysis of modifiability. *Journal of Systems and Software*, 61(1):47–57.

Lassing, N., Rijsenbrij, D., and van Vliet, H. (1999). Towards a broader view on software architecture analysis of flexibility. In *Proceedings of the 6th Asia-Pacific Software Engineering Conference '99 (APSEC'99)*, pages 238–245, Los Alamitos, CA. IEEE CS Press.

Lassing, N., Rijsenbrij, D., and van Vliet, H. (2001). Viewpoints on modifiability. *International Journal on Software Engineering and Knowledge Engineering*, 11(4):453–478.

Lassing, N., Rijsenbrij, D., and van Vliet, H. (2002b). How ell can we predict changes at architecture design time. *To appear, Journal of Systems and Software*.

Lindvall, M. and Runesson, M. (1998). The visibility of maintenance in object models: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 54–62, Los Alamitos, CA. IEEE CS Press.

Lindvall, M. and Sandahl, K. (1998). How well do experienced software developers predict software change? *Journal of Systems and Software*, 43(1):19–27.