ELSEVIER

The Journal of Systems and Software 61 (2002) 47–57

The Journal of Systems and Software

www.elsevier.com/locate/jss

# Experiences with ALMA: Architecture-Level Modifiability Analysis

Nico Lassing[a], PerOlof Bengtsson[b], Hans van Vliet[c,*], Jan Bosch[d]

[a] Accenture, Amsterdam, The Netherlands

[b] Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Ronneby, Sweden

[c] Faculty of Science, Division of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands

[d] Department of Mathematics and Computing Science, University of Groningen, Groningen, The Netherlands

Received 21 November 2000; received in revised form 7 March 2001; accepted 22 March 2001

**Abstract**

Modifiability is an important quality for software systems, because a large part of the costs associated with these systems is spent on modifications. The effort, and therefore cost, that is required for these modifications is largely determined by a system's software architecture. Analysis of software architectures is therefore an important technique to achieve modifiability and reduce maintenance costs. However, few techniques for software architecture analysis currently exist. Based on our experiences with software architecture analysis of modifiability, we have developed ALMA, an architecture-level modifiability analysis method consisting of five steps. In this paper we report on our experiences with ALMA. We illustrate our experiences with examples from two case studies of software architecture analysis of modifiability. These case studies concern a system for mobile positioning at Ericsson Software Technology AB and a system for freight handling at DFDS Fraktarna. Our experiences are related to each step of the analysis process. In addition, we made some observations on software architecture analysis of modifiability in general. © 2002 Elsevier Science Inc. All rights reserved.

## 1. Introduction

Software evolves, whether we like it or not. Studies indicate that more than 50% of total life cycle cost is spent after initial development (Lientz and Swanson, 1980; Nosek and Palvia, 1990). This high cost of software maintenance is caused by the incorporation of all kinds of anticipated and unanticipated changes after the system has been delivered. Bass et al. (1998) make a distinction between local, non-local, and architectural changes. A local change involves the modification of a single component. A non-local change requires modifications in multiple components. An architectural change affects the fundamental ways in which the components interact, and is likely to require changes all over the system. Obviously, local changes are preferable to non-local and architectural changes.

During software development, a number of strategies are applied to ease a system's evolution, such as separation of concerns, information hiding, the application of design patterns, etc. Note, however, that the use of 'improved' software engineering technologies does not necessarily incur lower maintenance costs. For instance, Dekleva (1992) found no difference in maintenance costs between systems developed using development methodologies considered modern at that time, and those developed using a more traditional approach. Development methodologies were considered modern if they produced an implementation-independent logical representation of a system's function. Dekleva found that such modern development methodologies lead to changes in the *allocation* of maintenance time. Less time was spent on bug fixing, evaluating change requests, and the like, while more time was spent on implementing changes imposed by external factors and functional enhancements. Apparently, those modern methodologies enable the realization of significant enhancements, while traditional methodologies facilitate patching but discourage users to request major changes. We may conjecture a similar phenomenon, i.e. maintenance will be different but not necessarily cheaper, for architecture- based development versus non-architecture-based development.

* Corresponding author. Tel.: +31-20-444-7768; fax: +31-20-444-7653.
*E-mail address:* hans@cs.vu.nl (H. van Vliet).

We have no empirical data to substantiate this conjecture, yet.

Another strategy that may be employed to enhance the modifiability of a system is to assess the quality of interim results produced in the development process. In particular, the software architecture of the system may be assessed to determine the evolutionary capabilities of the system yet to be built. IEEE Standard 1471 (IEEE, 2000) defines software architecture as 'the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution'. The software architecture captures the first design decisions concerning the system.

In an architecture-based development process, the result of these design decisions is represented in an architectural description. The main mechanisms to represent architectural descriptions are architecture description languages (ADLs) and view models. ADLs resemble module interconnection languages (MILs) (Prieto-Diaz and Neighbors, 1986). For an overview of ADLs, see (Medvidovic and Taylor, 2000).

View models are based on the principle that a software architecture cannot be captured in a single view. View models define a number of aspects of a software architecture that should be addressed in a software architecture description, e.g. (Kruchten, 1995) or (Soni et al., 1995). The views in both these models are represented as box-and-line diagrams. Their semantics are defined rather informally. A more recent version of the latter model uses UML to define its semantics more precisely (Hofmeister et al., 1999).

Early design decisions captured in these models have a considerable impact on various quality attributes of the system, including its modifiability. It thus pays off to assess these quality attributes at this early stage. We have defined a generalized, adaptable method for architecture-level modifiability analysis (ALMA), which is based on the Software Architecture Analysis Method (SAAM) (Kazman et al., 1996). ALMA is a generalization of earlier work by the authors independently (Bengtsson and Bosch, 1999; Lassing et al., 1999). In this paper we illustrate our experiences in using ALMA in two case studies. One case concerns a system developed by Ericsson Software Technology for positioning mobile telephones, and the other case concerns a system for freight handling at DFDS Fraktarna. The observations made in this paper corroborate and strengthen our individual experiences in earlier studies (Bengtsson and Bosch, 1999; Lassing et al., 1999).

In Section 2, we sketch ALMA. The steps of this method are used in subsequent sections to structure our experiences and put them into perspective. Section 3 introduces the two cases. Our experiences are described in Section 4. In Section 5, we conclude with some summarizing statements.

## 2. Overview of ALMA

The method ALMA that we propose is based on change scenarios. A change scenario is a description of a specific event that may occur in the life cycle of a system and requires the system to be modified. Change scenarios resemble change cases, as defined by Ecklund et al. (1996). Both change scenarios and change cases capture potential change. Change cases are tied to methodologies supporting use cases; change scenarios do not require such a context. By exploring the effect of change scenarios on the software architecture, we can make predictions of the effort that is required to implement the changes once the system is built. This enables us to judge the modifiability of the system that will be built with the software architecture under analysis.

ALMA has a fixed structure, consisting of the following five steps:

1. Set goal: determine the aim of the analysis.
2. Describe software architecture: give a description of the relevant parts of the software architecture.
3. Elicit change scenarios: find the set of relevant change scenarios.
4. Evaluate change scenarios: determine the effect of the set of change scenarios.
5. Interpret the results: draw conclusions from the analysis results.

The main difference between ALMA and SAAM (Kazman et al., 1996) is that SAAM does not explicitly address the need for differences in the techniques used as part of the analysis depending on the goal of the analysis. In ALMA this is an essential part and the need to clearly distinguish the goal of the analysis and to only have one goal for each analysis instance is emphasized.

We will now give a brief overview of the steps. A more elaborate discussion of the full method is given in (Bengtsson et al., 2000). Obviously, the above steps are not strictly separated when performing an analysis, but will often be iterated over in more or less subtle ways.

### 2.1. Goal setting

Architecture assessment must take place within the context of stakeholders' needs or requirements. The first step to take in the analysis is to set the analysis goal. The goal determines the type of results that will be delivered by the analysis. In addition, the goal influences the choice of techniques to be used in subsequent steps. Different goals ask for different techniques. With respect to modifiability, the following goals can be pursued:

- *Risk assessment*: finding types of changes for which the system is inflexible. We are then interested in scenarios that are particularly difficult to accomplish. In terms of effort and probability, we are then interested in the outliers. In software testing, it is sometimes stated that a test is only useful if it reveals a fault in the software. Pursuing this analogy, we may state that, in the risk-oriented approach to software architecture analysis, a change scenario is only useful if it exposes a risk, i.e. the changes it induces are difficult to accomplish. So, for risk assessment we look for a set of change scenarios that complies with the operational *risk* profile of the system.

- *Maintenance cost prediction*: estimating the cost of maintenance effort for the system in a given period. This estimate only concerns modifications caused by changes in the environment, in requirements, or in the functional specification. The cost incurred by bug fixing is excluded, since it is highly dependent upon factors not yet known at this point in time. In a very general sense, we then use a maintenance cost function $C_{\text{average}}$ (the average cost per change scenario) of the form

$$C_{\text{average}} = \frac{\sum_{i=0}^{n} C(\text{change}_i) \cdot p(\text{change}_i)}{n},$$

where $C(\text{change}_i)$ denotes the effort or cost required to realize the $i$th change scenario, and $p(\text{change}_i)$ denotes the probability this scenario will occur, called *scenario weight* in ALMA.
This approach presupposes that we are able to estimate the probability that a given change scenario will occur. If no such information is available, we assume a uniform distribution. In either case, we are interested in identifying those change scenarios that are likely to occur during the operational life of the system. I.e. we look for a set of change scenarios that matches the operational *change* profile of the system.
- *Software architecture comparison*: comparing two or more candidate software architectures to find the most appropriate one. The difference with the two aforementioned goals is that in comparison we make relative statements about a number of candidate software architectures, while with the other goals we make absolute statements about a single candidate. In comparison we are interested in finding the differences between the software architectures, so the scenario elicitation should be aimed at finding change scenarios that are treated differently by the candidates.

## 2.2. Software architecture description

After the goal of the analysis is set, the next step is to create a description of the software architecture. To do so, the architecture designs used within the development team are an important source of information. In addition, we may ask the architect(s) of the system for additional information, e.g. as component size estimates, or interviewing them for additional architecture information.

Software architecture description serves two purposes within the analysis. First and foremost, it should result in a description that is detailed enough to enable architecture level impact analysis for the set of change scenarios, i.e. assess their effect based on the software architecture. The need for multiple views of a system's software architecture is widely recognized (IEEE, 2000; Kruchten, 1995; Soni et al., 1995). For architecture impact analysis, several such views are used. For example, Figs. 1 and 2 give two views that were used in the analysis of EASY (for a description of EASY, see Section 3.2). Fig. 1 gives the context view, i.e. the system in its environment. Fig. 2 gives the conceptual view, i.e. the architectural approach taken for the system.

In addition, software architecture description provides us with insight into the domain of the system

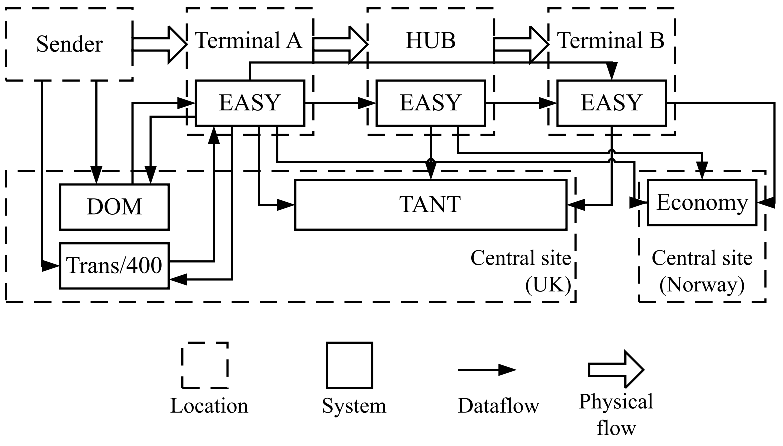

Fig. 1. Context view of EASY.

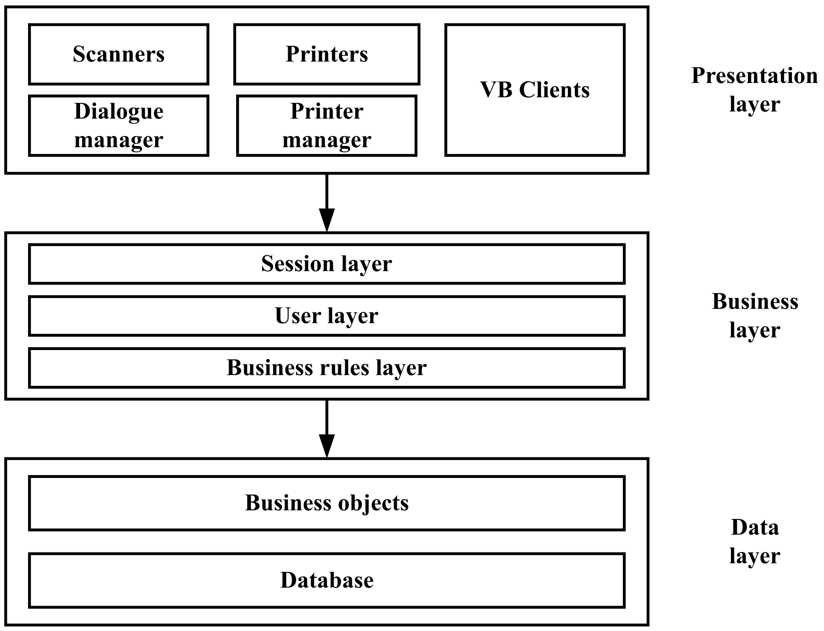

Fig. 2. Conceptual view of EASY.

under analysis. This is important because it allows us to understand the scenarios that the stakeholders bring forward in the next step.

If the goal of the assessment is to predict maintenance cost, we need estimates of the size of architecture components, e.g. in terms of lines of code. Note though that not a single line of code is written yet when doing an architecture assessment, and such estimates are based on quite global information.

## 2.3. Scenario elicitation

One of the most important steps in modifiability analysis is the elicitation of a set of change scenarios. This set of change scenarios captures the events that stakeholders expect to occur in the future of the system. The main technique to elicit this set is to interview stakeholders, because they are in the best position to predict what may happen in the future of the system. In addition, they are able to judge the likelihood of the change scenarios obtained. This likelihood is important because unlikely scenarios are not relevant for the analysis and can be discarded. The aim of the scenario elicitation step is to come to a set of change scenarios that supports the goal that we have set for the analysis.

## 2.4. Scenario evaluation

After eliciting a set of change scenarios, we determine their effect on the system. To do so, we perform architecture-level impact analysis for each of the scenarios individually. This means that we determine the components of the system and components of other systems that have to be adapted to implement the change scenario. Also, components to be added or deleted are identified. This task is typically performed in collaboration with members of the development team.

## 2.5. Interpretation

After we have determined the effect of the change scenarios, we can interpret these results to come to a conclusion about the system under analysis. The way the results are interpreted is again dependent on the goal of the analysis.

If the goal of the analysis is risk assessment the results of the scenario evaluation are investigated to determine which change scenarios pose risks, i.e. for which scenarios the product of probability and costs is too high. The owner of the system should decide on the criteria for determining which values are still acceptable. When risks are found, various risk mitigation strategies are possible: avoidance (take measures to avoid that the scenario will occur or take action to limit their effect, for instance, by use of code- generation tools), transfer (e.g. choose another software architecture) and acceptance (accept the risks).

If the goal of the analysis is to do maintenance prediction, the aim of this step is to come to an estimate of the amount of effort that is required for maintenance activities in the coming period. If the goal of the analysis is architecture comparison, we compare the results of the evaluation of the two sets of scenarios and choose the most appropriate candidate architecture.

## 3. Case descriptions

In this section we introduce the two system used in the case studies. The goal of the case studies was to conduct a software architecture analysis of modifiability on each of the two systems. We adopted the action research paradigm and took upon ourselves the role of being external analysts.

The domains of the cases were very different, namely, business information systems and mobile telecommunications services systems. The differences between the domains (see Table 1) illustrate the scope of applicability of our method. In the remainder of this section we will introduce the two systems.

### *3.1. Ericsson Mobile Positioning Center (MPC)*

The MPC is a system for locating mobile phones in a cellular mobile phone network and reporting their geographical position. The network operators may implement their own services based on the positioning service. The positioning is not only intended for commercial services but also for locating emergency calls.

The MPC client/server system consists of the MPC server and a Graphical User Interface (GUI) as a client. The MPC server handles all communication with external systems to retrieve positioning data and is also responsible for the processing of positioning data. The MPC server may also be divided into two units to adhere to the standard proposal that is under development.

The MPC GUI is used for system administration, such as configuring for which mobile phones a user is allowed the position, and configuring alarm handling. The MPC system also generates billing information to allow the network operators to charge for the services they provide.

The MPC is typically deployed as one unit at the network operators, but additional MPCs can be used in the network for redundancy. Each operator that offers positioning services needs an MPC system or the like in the network.

The goal of the assessment was to predict the maintenance cost for the current system. Hence, we focused the elicitation of change scenarios on finding likely changes. Scenarios were collected by interviewing three stakeholders in the project. First we interviewed one of the software architects, then we interviewed a designer in the project, and finally we interviewed the operative product manager. Because the product is intended to serve several customers we had no direct access to the customer as a stakeholder in the assessment. However, one of the responsibilities of the operative product manager position is to stay informed of the market demands and issues. The interviews lasted between 1 and 2 h.

### *3.2. EASY*

EASY is a system that is currently being developed by Cap Gemini Ernst & Young for DFDS Fraktarna, a Swedish distributor of freight. EASY was developed to monitor the location of groupage (freight with a weight between 31 kg and 1 t) through the company's distribution system. To do so, the unique barcode of groupage is scanned after each step in the distribution system using a mobile barcode scanner. This information is then stored.

Part of the architecture of EASY is shown in Figs. 1 and 2. Fig. 1 shows that each terminal has an autonomous instance of EASY. As a result, EASY had to be designed in such a way that it can be used for both large sites and small sites. At large sites, the system uses a large number of barcode scanners, workstations and servers, but at small sites one barcode scanner and one workstation could be enough.

All information that is collected at a site is stored in the local instance of EASY. In addition, this information is sent to a central system called TANT that is used for tracking and tracing of freight. Based on the information that is stored, TANT is always able to determine where a groupage of freight is, or should be, at a certain point in time.

The goal that was set for the analysis was risk assessment, i.e. we were interested in finding changes for which the system is inflexible. For scenario elicitation we held interviews with different stakeholders of the system

Table 1
Comparison of system characteristics

| Aspect | EASY | MPC |
|---|---|---|
| Type of system | Business information system | Mobile telecom |
| Customer situation | One customer | Targeting hundreds of installations |
| Domain standards | No domain specific standard, several proprietary information systems to interface | Standardization ongoing, company engaged in the effort |
| Human system interaction | Several points of human interaction, e.g. multiple parcel handling points, label printing, administration, and planning | Few points of human interaction, administration only. All other interaction is handled by other systems |
| Deployment | Distributed over 28 sites, each with a variant of small to large set up | Possibly divided on two physical nodes |
| System release status | System successfully tested in pilot study | Second release with major changes under development |

individually. These stakeholders were: an architect of the system, one of the designers and a representative of the customer. So, there were three interviews, each of which lasted 1–2 h. In the following section we discuss the experiences that we had in these interviews.

## 4. Experiences

This section gives a description of the experiences that we acquired during the definition and use of ALMA. These experiences will be presented using the five steps of the method. They will be illustrated using examples from the two case studies introduced before.

### 4.1. Goal setting

#### 4.1.1. Modifiability analysis requires a clear goal

The goal of the analysis determines what techniques to be used in the following analysis steps. For instance, when the goal is to make a risk assessment the elicitation technique to prefer is the guided interview. In a guided interview the analyst stimulates the stakeholders to bring forward change scenarios that are especially complex, thereby reducing the chance that certain complex change scenarios are overlooked (see also Section 4.3.3).

When we discussed the planning for these case studies in our first meeting with the companies we in fact already discussed the goal for the analysis of their respective software architectures. We, the analysts, at that time did not fully understand the impact of the analysis goal to the following steps. In retrospect, we were right in making a choice for a specific goal at the very outset of the analysis.

The techniques in the following steps of the analysis method are different in significant ways for important reasons. It is far from trivial to combine the techniques or to perform the steps of the analysis using, for example, elicitation techniques for different goals in parallel.

Hence, make sure that there is *one* clear goal set for the analysis. The solution in our case was that we performed a prediction-type analysis for the mobile positioning system and a risk assessment for EASY.

### 4.2. Architecture description

#### 4.2.1. Views

Several authors have proposed view models, consisting of a number of architectural views (Kruchten, 1995; Soni et al., 1995). Each view focuses on a particular aspect of the software architecture, e.g. its static structure or its dynamic aspect. We found that in software architecture analysis of modifiability a number of these views is required, but not all of them. The goal of the architecture description is to provide input for the following steps of the analysis or, more specifically, for determining the changes required for the change scenarios. We found that the views most useful for doing so are the view that shows the architectural approach taken for the system, i.e. the conceptual view, and the view that shows the way the system is structured in the development environment, i.e. the development view. However, to explore the full effect of a scenario it is not sufficient to look at just one of these.

For instance, in our analysis of EASY the owner of the system suggested the following change scenario. He mentioned that, from a systems management perspective, it is probably too expensive to have an instance of EASY at all terminals. He indicated that the number of instances should be reduced, while maintaining the same functionality. To determine the effect of this scenario, we had to look at the view of the architecture that showed the division of the system in loosely coupled local instances, i.e. the view that showed the architectural approach for the system. The architect indicated that to implement this scenario there should be some kind of centralization, i.e. instead of an autonomous instance of the system at each site we would get a limited number of instances at centralized locations. This meant that one important assumption of the system was no longer valid, namely the fact that an instance of EASY only contains information about groupage that is processed at the freight terminal where the instance is located. To explore the effect of this change, we investigated the development view and found that a number of components had to be adapted to incorporate this change. So, one view was not sufficient to explore the effect of the scenario, but we did not use any views relating to the dynamics of the system. The same applied to the other change scenarios; we only used the conceptual and the development view to explore their effect.

However, for some goals we needed additional information. This issue is discussed in subsequent experiences.

#### 4.2.2. The system's environment

In earlier research (Lassing et al., 1999) we found that in some cases it is important that the environment of the system is also modeled. We found this to be most useful in risk assessment. The reason for this is that changes that include the environment are considered more complex than changes that are limited to the system itself. For maintenance prediction the environment is less important since the scope of the prediction is generally limited to the system. In that case we focus on the effort that is required to adapt the system itself. Changes to its environment are not included in the prediction.

To illustrate this point, consider the following change scenario that we found during the analysis of EASY. In scenario elicitation, we came across the scenario that represented the event that the middleware of EASY is replaced with another type. This change not only affects EASY, but also requires systems with which EASY communicates to be adapted. This means that this change cannot be implemented without consulting the owners of these other systems and hence it may be considered a risk. For maintenance prediction, however, we would only be interested in the effort that is required for adapting EASY to this new middleware. In that case, the environment of the system is not used in the analysis.

#### *4.2.3. Need for additional information*

Another observation that we made is that for some goals we need additional information in our analysis that is normally not seen as part of the software architecture. For instance, for maintenance prediction, we want to make estimates of the size of the required changes. To do so, we need not only information about the structure of the system, but also concerning the size of the components. These numbers are normally not available at the software architecture level, so they have to be estimated by the architects and/or designers.

For risk assessment of business information systems we need another type of additional information. In that case, we need to know about the system owners that are involved in a change (Lassing et al., 1999). This information is normally not included in the software architecture designs, so it has to be obtained separately from the stakeholders.

### *4.3. Scenario elicitation*

#### *4.3.1. Different perspectives*

Scenario elicitation is usually done by interviewing different stakeholders of the system under analysis (Abowd et al., 1996). We found that it is important to have a mix of people from the 'customer side' and from the 'development side'. Different stakeholders have different goals, different knowledge, different insights, and different biases. This all adds to the diversity of the set of scenarios. In testing, different test techniques reveal different types of errors (Kamsties and Lott, 1995). In vacuuming, one is likely to pick up more dirt if the rug is vacuumed in two directions. Similarly, in architecture analysis, it helps to collect scenarios from a variety of sources.

In our analysis of EASY, we found that the customer attached more importance on scenarios aimed at decreasing the cost of ownership of the system, e.g. introduce thin-clients instead of PCs. The architect and the designer of the system focused more on scenarios that aimed for changes in growth or configuration, e.g. the number of terminals change and integration with new suppliers' systems.

#### *4.3.2. The architect's bias*

We have experienced a particular recurring type of bias when interviewing the architect of the system being assessed. In both case studies, the architect, when asked to come up with change scenarios, came up with scenarios that he had already anticipated when designing the architecture.

Of course, this is no surprise. A good architect anticipates for the most probable changes that the system will undergo in the future. Another reason for this phenomenon could be that the architect of the system is, implicitly, trying to convince himself and his environment that he has taken all the right decisions. After all, it is his job to devise a flexible architecture. It requires a special kind of kink to destroy what you yourself have just created. This is the same problem programmers have when testing their own code. It requires skill and perseverance of the analyst to take this mental hurdle and elicit relevant scenarios from this stakeholder.

For example, in the analysis of EASY, the architect mentioned that the number of freight terminals could change. This type of change is very well supported by the chosen architecture solution. Similarly, in the analysis of the MPC, the architect mentioned the change scenario 'physically divide the MPC into a serving and a gateway MPC', which was in fact already supported by the architecture. So, relying only on the architect to come up with change scenarios may lead us to belief that all future changes are supported. This again stresses the importance of having a mix of people for scenario elicitation.

#### *4.3.3. Structured scenario elicitation*

When interviewing stakeholders, recurring questions are:

- Does this scenario add anything to the set of scenarios obtained so far?
- Is this scenario relevant?
- In what direction should we look for the next scenario?
- Did we gather enough scenarios?

Unguided scenario elicitation relies on the experience of the analyst and stakeholders in architecture assessment. The elicitation process then stops if there is mutual confidence in the quality and completeness of the set of scenarios. This may be termed the *empirical* approach (Carroll and Rosson, 1992). One of the downsides of this approach that we have experienced is that the stakeholders' horizon of future changes is very short. Most change scenarios suggested by the stakeholders relate to issues very close in time, e.g. anticipated changes in the current release.

To address this issue we have found it very helpful to have some organizing principle while eliciting scenarios. This organizing principle takes the form of a, possibly

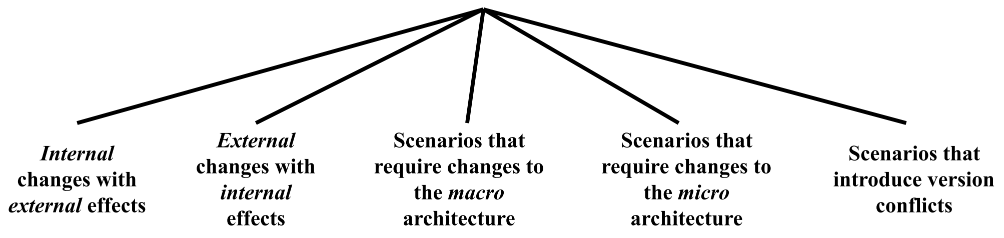
*Internal* changes with *external* effects
*External* changes with *internal* effects
Scenarios that require changes to the *macro* architecture
Scenarios that require changes to the *micro* architecture
Scenarios that introduce version conflicts

Fig. 3. Deriving change scenarios from knowledge of complexity of changes in business information systems.

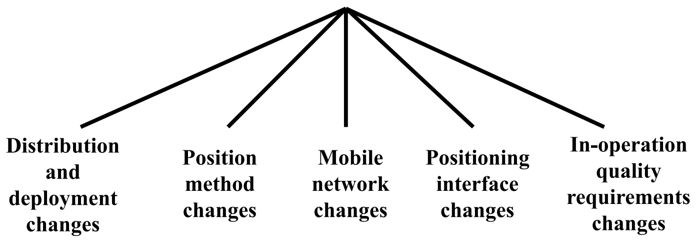
Distribution and deployment changes
Position method changes
Mobile network changes
Positioning interface changes
In-operation quality requirements changes

Fig. 4. Deriving change scenarios from problem domain.

hierarchical, classification of scenarios to draw from. For instance, in our risk assessment of EASY, we used a categorization of high-risk changes as given in Fig. 3. This categorization has been developed over time, while gaining experience with this type of assessment. When the focus of the assessment is to estimate maintenance cost, the classification tends to follow the logical structure of the domain of the system, as in Fig. 4. If this hierarchy is not known a priori, an iterative scheme of gathering scenarios, structuring those scenarios, gathering more scenarios, etc., can be followed. In either case, this approach might be called *analytical*.

In the analytical approach, the classification scheme is used to guide the search for additional scenarios. At the same time, the classification scheme defines a stopping criterion: we stop searching for yet another change scenario if: (1) we have explicitly considered all categories from the classification scheme, and (2) new change scenarios do not affect the classification structure.

### *4.3.4. Scenario probability confusion*

The maintenance prediction depends on estimating the probabilities for each scenario, which we in ALMA call scenario weights. Of course, the sum of these weights, being probabilities, should be one. The problem for the stakeholders is that they often know that a certain change *will* occur. Intuitively, they feel that the weight for that change scenario should be one. Problems then arise if another scenario *will also* occur.

The key to understanding the weighting is to perform it in two steps, estimation and normalization (see Table 2). The prediction is aimed for a certain period of time, e.g. between two releases of the system. If there is a scenario, say scenario 1, that we know will happen, we first estimate how many times changes in the scenario's equivalence class will occur. Suppose one change belonging to the equivalence class will occur. For another change scenario, scenario 2, we expect changes of that equivalence class to occur, say, three times during the same period. This estimation is done for all scenarios. Then we perform step two, normalization. The weights are calculated for each scenario by dividing the number of changes estimated for that scenario by the sum of the estimates of the complete set of scenarios. In the example just given, the result is that the weight of the first scenario is 0.25 and the weight of the second scenario is 0.75 (rightmost column in Table 2).

Table 2
Weighting process

| | Step 1 | Step 2 | Result |
|---|---|---|---|
| | Estimated number of changes | Normalization | |
| Scenario 1 | 1 | 1/(1 + 3) | 1/4 |
| Scenario 2 | 3 | 3/(1 + 3) | 3/4 |

## *4.4. Scenario evaluation*

### *4.4.1. Ripple effects*

The activity of scenario evaluation is concerned with determining the effect of a scenario on the architecture. For instance, if the effects are small and localized, one may conclude that the architecture is highly modifiable for at least this scenario. Alternatively, if the effects are distributed over several components, the conclusion is generally that the architecture supports this scenario poorly.

However, how does one determine the effects of a scenario? Often, it is relatively easy to identify one or a few components that are directly affected by the scenario. The affected functionality, as described in the scenario, can directly be traced to the component that implements the main part of that functionality.

The problem that we have experienced in several cases is that *ripple effects* are difficult to identify. Simply put, a ripple effect is the result of changes to the interface of the component directly affected by a change scenario. Since the provided and/or required interfaces of the component change, the change ripples to the components connected to these interfaces. Although the software architect, during software architecture design, has a reasonable understanding of the decomposition of functionality, it nevertheless often proves difficult to decide whether a change scenario will affect the provided and required interfaces of the component and, in this way, affects other components in the architecture.

The main factor influencing this problem is the amount of detail present in the description of the software architecture. This is determined by the amount of detail available to the software architect, i.e. his or her understanding of the problem. In a study Lindvall and Runesson (1998) have shown that even detailed design descriptions lack information necessary for performing an accurate analysis of ripple effects. Since less detail is available at the software architecture level, this is an even more relevant problem. In addition, Lindvall and Sandahl (1998) showed that even experienced developers clearly underestimate the number of classes impacted by a change. On the other hand, they were almost always right when identifying classes that needed change.

To illustrate this problem, consider the following example. In the MPC case, one of the scenarios concerned the implementation of support for standardized remote management. Although this change could be evaluated by identifying the components directly affected, it was extremely hard to foresee the changes that would be required to the interface of these components. As a consequence, there was a very high degree of uncertainty as to the exact amount of change needed.

### *4.5. Interpretation of the results*

#### *4.5.1. Lack of frame of reference*

Once scenario evaluation has been performed, we have to associate conclusions with these results. The process of formulating these conclusions we refer to as *interpretation*. The experience we have is that generally we lack a frame of reference for interpreting the results of scenario evaluation and often there are no historical data to compare with.

For instance, in maintenance prediction, the result is a prediction of the average size of a change, in lines of code, function points, or some other size measure. In the analysis of the MPC we came to the conclusion that, on average, there would be 270 lines of code affected per change scenario. Is this number 'good' or 'bad'? Can we develop an architecture for MPC that requires an average of 100 LOC per change? To decide if the architecture is acceptable in terms of modifiability, a frame of reference is needed.

#### *4.5.2. Role of the owner in interpretation*

Our identification of the above leads us immediately to the next experience: the *owner of the system* for which the software architecture is designed plays a crucial role in the interpretation process. In the end, the owner has to decide whether the results of the assessment are acceptable or not. This is particularly the case for one of the three possible goals of software architecture analysis, i.e. risk assessment. The scenario evaluation will give insight in the boundaries of the software architecture with respect to incorporating new requirements, but the owner has to decide whether these boundaries, and associated risks, are acceptable or not.

Consider, for instance, the change scenario for EASY that we already mentioned in Section 4.2.2, namely the replacement of the middleware used for EASY. This scenario not only affects EASY, but also the systems with which EASY communicates. This means that the owners of these systems have to be convinced to adapt their systems. This may not be a problem, but it *could* be. Only the owner of the system can judge such issues and therefore it is crucial to have him/her involved in the interpretation.

The system owner also plays an important role when other goals for software modifiability analysis are selected, i.e. maintenance cost prediction or software architecture comparison. In this case, the responsibility of the owner is primarily towards the change profile that is used for performing the assessment. The results of the scenario evaluation are accurate to the extent the profile represents the actual evolution path of the software system.

### *4.6. General experiences*

#### *4.6.1. Software architecture analysis is an ad hoc activity*

Three arguments are used for explicitly defining a software architecture (Bass et al., 1998). First, it provides an artifact that allows for discussion by the stakeholders very early in the design process. Second, it allows for early assessment or analysis of quality attributes. Finally, it supports the communication between software architects and software engineers since it captures the earliest and most important design decisions. In our experience, the second of these is least applied in practice. The software architecture is seen as a valuable intermediate product in the development process, but its potential with respect to quality assessment is not fully exploited.

In our experiences, software architecture analysis is mostly performed on an ad hoc basis. We are called in as an external assessment team, and our analysis is mostly used at the end of the software architecture design phase as a tool for acceptance testing ('toll-gate approach') or during the design phase as an audit instrument to assess whether the project is on course. In either case, the assessment mostly is not solidly embedded in the development process. As a consequence, earlier activities do not prepare for the assessment, and follow-up activities are uncertain.

If software architecture analysis were an integral part of the development process, earlier phases or activities would result in the necessary architectural descriptions, planning would include time and effort of the assessor as well as the stakeholders being interviewed, design iterations because of findings of the assessment would be part of the process, the results of an architecture assessment would be on the agenda of a project's steering committee, and so on. This type of embedding of software architecture analysis in the development process, which is in many ways similar to the embedding of design reviews in this process, is still very uncommon.

#### 4.6.2. *Accuracy of analysis is unclear*

A second general experience is that we lack means to decide upon the accuracy of the results of our analysis. We are not sure whether the numbers that maintenance prediction produces (such as the ones mentioned in Section 4.5.1) are accurate, and whether risk assessment gives an overview of all risks. On the other hand, it is doubtful whether this kind of accuracy is at all achievable. The choice for a particular software architecture influences the resulting system and its outward appearance. This in turn will affect the type of change requests put forward. Next, the configuration control board, which assesses the change requests, will partly base its decisions on characteristics of the architecture. Had a different architecture been chosen, then the set of change requests proposed and the set of change requests approved would likely be different. Architectural choices have a feedback impact on change behavior, much like cost estimates have an impact on project behavior (Abdel-Hamid and Madnick, 1986).

A further limitation of this type of architecture assessment is that it focuses on aspects of the product only, and neglects the influence of the process. For instance, Avritzer and Weyuker (1998) found that quite a large number of problems uncovered in architecture reviews had to do with the process, such as not clearly identified stakeholders, unrealistic deployment date, no quality assurance organization in place, and so on.

## 5. Conclusions

Software architecture is generally regarded as an important instrument to increase the quality of software systems. One of the qualities to which this applies is modifiability. We have defined a generalized five-step method ALMA (Bengtsson et al., 2000). In this paper we report on 13 experiences we acquired developing and using ALMA. These experiences are illustrated using two case studies that we performed: the MPC system developed by Ericsson Software Technology for positioning mobile telephones and the EASY system for freight handling at DFDS Fraktarna.

With respect to the first step of the analysis method, goal setting, we found that it is important to decide on *one goal* for the analysis. For the second step of the method, architecture description, we experienced that the impact of a change scenario may span several architectural views. However, we also found that views relating to the system's dynamics are not required in modifiability analysis. But for some analysis goals we need information that is not included in existing architecture view models. For risk assessment, we found the system's environment and information about system owners useful in evaluating change scenarios, whereas these are not required when performing maintenance prediction. However, for maintenance prediction, we require information about the size of the components, which is normally not considered to be part of the software architecture design.

Concerning the third step of the method, scenario elicitation, we made the following four main observations. First, it is important to interview different stakeholders to capture scenarios from different perspectives. We also found that the architect has a certain bias in proposing scenarios that have already been considered in the design of the architecture. Another observation we made was that the time horizon of stakeholders is rather short when proposing change scenarios and that guided elicitation might help in improving this. A more specific experience was that the weighting scheme used for maintenance prediction appeared somewhat confusing to the stakeholders.

Concerning the fourth step of the method, scenario evaluation, we experienced that ripple effects are hard to identify since they are the result of details not yet known at the software architecture level. Regarding the fifth step of the method, interpretation, we experienced that the lack of a frame of reference makes the interpretation less certain, i.e. we are unable to tell whether the predicted effort is relatively high or low, or whether we captured all or just a few risks. Because of this, the owner plays an important role in the interpretation of the results.

We also made some general experiences that are not directly related to one of ALMA's steps. First, we have found that, in practice, software architecture analysis is an ad hoc activity that is not explicitly planned for. Second, the validity of the analysis is unclear as to the accuracy of the prediction and the completeness of the risk analysis.

In our view, the case studies have provided valuable experiences that will contribute to a better understanding of scenario-based analysis.

## Acknowledgements

We are grateful to Cap Gemini Ernst & Young, and especially Daan Rijsenbrij, for the financial support of this research. In addition, we would like to thank the companies that enabled us to perform the case studies, i.e. DFDS Fraktarna and Ericsson Software Technology. We would especially like to thank Åse Petersén, David Olsson, Stefan Gustavsson and Staffan Johnsson of Ericsson Software Technology AB, Joakim Svensson and Patrik Eriksson of Cap Gemini Ernst & Young, Stefan Gunnarsson of DFDS Fraktarna for their valuable time and input.

## References

Abdel-Hamid, T.K., Madnick, S.E., 1986. Impact of schedule estimation on software project behavior. IEEE Software 3 (4), 70–75.

Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., Zamerski, A., 1996. Recommended best industrial practice for software architecture evaluation. Technical Report (CMU/SEI-96-TR-025), Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA.

Avritzer, A., Weyuker, E.J., 1998. Investigating metrics for architectural assessment. In: Proceedings of the Fifth International Software Metrics Symposium, Bethesda, Maryland, pp. 4–10.

Bass, L., Clements, P., Kazman, R., 1998. Software Architecture in Practice. Addison-Wesley, Reading, MA.

Bengtsson, P., Bosch, J., 1999. Architecture level prediction of software maintenance. In: Proceedings of Third EuroMicro Conference on Maintenance and Reengineering (CSMR'99), pp. 139–147.

Bengtsson, P., Lassing, N., Bosch, J., van Vliet, H., 2000. Analyzing software architectures for modifiability. Technical Report (HK- R-RES00/11-SE). University of Karlskrona/Ronneby, Ronneby.

Carroll, J.M., Rosson, M.B., 1992. Getting around the task-artifact cycle. ACM Transactions on Information Systems 10 (2), 181–212.

Dekleva, S.M., 1992. The influence of the information systems development approach on maintenance. MIS Quarterly 16 (3), 355–372.

Ecklund, Jr., E.F, Delcambre, L.M.L., Freiling, M.J., 1996. Change cases: use cases that identify future requirements. In: Proceedings OOPSLA '96. ACM, New York, pp. 342–358.

Hofmeister, C., Nord, R., Soni, D., 1999. Applied Software Architecture. Addison-Wesley, Reading, MA.

IEEE Architecture Working Group, 2000. Recommended practice for architectural description. IEEE Standard 1471–2000.

Kamsties, E., Lott, C.M., 1995. An empirical evaluation of three defect-detection techniques. In: Schäfer, W., Botella, P. (Eds.), Software Engineering – ESEC '95, pp. 362–383.

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

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

Lassing, N., Rijsenbrij, D., van Vliet, H., 1999. Towards a broader view on software architecture analysis of flexibility. In: Proceedings of the Sixth Asia–Pacific Software Engineering Conference (APSEC'99), pp. 238–245.

Lientz, B., Swanson, E., 1980. Software Maintenance Management. Addison-Wesley, Reading, MA.

Lindvall, M., Runesson, M., 1998. The visibility of maintenance in object models: an empirical study. In: Proceedings of International Conference on Software Maintenance, pp. 54–62.

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

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (1), 70–93.

Nosek, J., Palvia, P., 1990. Software maintenance management: changes in the last decade. Journal of Software Maintenance 2 (3), 157–174.

Prieto-Diaz, R., Neighbors, J., 1986. Module interconnection languages. Journal of Systems and Software 6 (4), 307–334.

Soni, D., Nord, R.L., Hofmeister, C., 1995. Software architecture in industrial applications. In: Proceedings of the 17th International Conference on Software Engineering, Seattle, WA.

**Nico Lassing** has a MSc in Business Informatics from the Vrije Universiteit in Amsterdam, The Netherlands. He has been a PhD student at the same university, where his research focuses on software architecture analysis, and more specifically on modifiability assessment of business information systems. He currently works as a consultant at Accenture.

**PerOlof 'PO' Bengtsson** received his MSc degree in Software Engineering from Blekinge Institute of Technology, Sweden (1997). Before that he has been working as a quality assurance and reuse consultant at the Ericsson subsidiary Ericsson Software Technology AB, Sweden, (1995–1997). He is currently at Blekinge Institute of Technology where he is a PhD student. His research interests include software architecture design and evaluation, especially methods for predicting software qualities from software architecture.

**Hans van Vliet** is a Professor in Software Engineering at the Vrije Universiteit. His research interests include software architecture and software measurement. Before joining the Vrije Universiteit, he worked as a researcher at the Centrum voor Wiskunde en Informatica (Amsterdam) and he spent a year as a visiting researcher at the IBM Almaden Research Center in San Jose, California. Hans has an MSc in Computer Science from the Vrije Universiteit and a PhD in Computer Science from the University of Amsterdam.

**Jan Bosch** is a professor of software engineering at the University of Groningen, where he heads the software engineering research group. He received a MSc degree from the University of Twente, The Netherlands, and a PhD degree from Lund University, Sweden. His research activities include software architecture design, software product lines, object-oriented frameworks and component-oriented programming. He is the author of a book "Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach" published by Pearson Education (Addison-Wesley & ACM Press), coeditor of two volumes in the Springer LNCS series and has (co-)authored more than 50 refereed journal and conference publications. He has organized numerous workshops and served on many programme committees, including the ICSR'6, CSMR'2000, ECBS'2000, SPLC1 and TOOLS conferences. He is the PC co-chair of the 2002 Working IEEE/IFIP Conference on Software Architecture (WICSA).