Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
The Journal of Systems and Software 69 (2004) 129–147
The Journal of Systems and Software
www.elsevier.com/locate/jss

Architecture-level modifiability analysis (ALMA)

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

[a] Ericsson AB, Inovation Development, KA/EPK/LU/RR, P.O. Box 518, Karlskrona 37123, Sweden
[b] Accenture, Amsterdam, The Netherlands
[c] Department of Mathematics and Computer Science, University of Groningen, Groningen, The Netherlands
[d] Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

Received 8 September 2001; received in revised form 29 March 2002; accepted 3 June 2002

Abstract

Several studies have shown that 50–70% of the total lifecycle cost for a software system is spent on evolving the system. Organizations aim to reduce the cost of these adaptations, by addressing modifiability during the system's development. The software architecture plays an important role in achieving this, but few methods for architecture-level modifiability analysis exist. Independently, the authors have been working on scenario-based software architecture analysis methods that focus exclusively on modifiability. Combining these methods led to architecture-level modifiability analysis (ALMA), a unified architecture-level analysis method that focuses on modifiability, distinguishes multiple analysis goals, has explicit assumptions and provides repeatable techniques for performing the steps. ALMA consists of five main steps, i.e. goal selection, software architecture description, change scenario elicitation, change scenario evaluation and interpretation. The method has been validated through its application in several cases, including software architectures at Ericsson Software Technology, DFDS Fraktarna, Althin Medical, the Dutch Department of Defense and the Dutch Tax and Customs Administration.

Keywords: Software architecture; Architecture analysis; Modifiability; Scenarios

1. Introduction

The world around most software systems is constantly changing. This requires software systems to be modified several times after their initial development. A number of studies (Ecklund et al., 1996; Lientz and Swanson, 1980) have shown that 50–70% of the total lifecycle cost of a software system is spent on modifications after initial development. 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.

Based on an understanding of the expected future evolution of the software system, design solutions can prepare for the future incorporation of new and changed requirements. In particular, the software architecture of the system affects the effort of making different types of modifications. One problem, however, is that the software architect has few means or techniques for determining whether the goal of high modifiability has been achieved by the design solutions.

One area of research addressing the above is *software architecture analysis*. In software architecture analysis, the software architecture of a system is analyzed to predict one or more quality attributes. Software architecture analysis method (SAAM) (Kazman et al., 1996) is a method for assessing software architectures with respect to different quality attributes. The principle of SAAM is to elicit scenarios from the stakeholders and explore their effects on the software architecture. SAAM provides few explicit techniques to be used in the different steps; it leaves much to the experience of the assessor. As such, the repeatability of the analysis may be compromized.

In their earlier work, the authors have independently worked on scenario-based software architecture analysis methods that focus exclusively on modifiability; Bengtsson and Bosch (1999a) have defined a method for predicting maintenance effort based on a system's software architecture and Lassing et al. (1999a) defined a method

* Corresponding author. Tel.: +46-455395073.
E-mail address: perolof.bengtsson@epk.ericsson.se (P. Bengtsson).

0164-1212/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/S0164-1212(03)00080-3

with focus on identifying inflexibility at the software architecture level. Comparing these methods revealed similarities and differences (Lassing et al., 2001). They share a similar structure and are both based on scenarios, but the techniques used in the various steps differ. These differences can largely be explained from the methods' founding assumptions. It was mainly the goals pursued by the analysis that influenced the assumptions. In combining the methods, these assumptions became explicit decisions and led to a unified architecture-level modifiability analysis method that; distinguishes multiple analysis goals, has visible assumptions and provides repeatable techniques for performing the steps. This method is called architecture-level modifiability analysis (ALMA).

ALMA consists of five main steps, i.e. goal selection, software architecture description, scenario elicitation, scenario evaluation and interpretation. We found that modifiability analysis generally has one of three goals, i.e. prediction of future maintenance cost, identification of system inflexibility and comparison of two or more alternative architectures. Depending on the goal, the method employs different techniques in some of the main steps.

The method has been applied successfully to a number of industrial cases, including software architectures at Ericsson Software Technology, DFDS Fraktarna, Althin Medical, the Dutch Department of Defense and the Dutch Tax and Customs Administration. The domains of these software architectures differ considerably, i.e. telecommunications, logistics, embedded systems and business information systems.

The remainder of the paper is organized as follows. In Section 2, we discuss the relationships between software architecture and modifiability. Section 3 discusses various approaches to software architecture analysis. Section 4 presents an overview of the main steps of ALMA. In Sections 5–7, we show how to proceed for each of the goals and which techniques to use. These sections are illustrated using case studies that we performed using ALMA. Section 8 discusses the validation of the method. Finally, we conclude with a summary of the paper in Section 9.

2. Modifiability in software architecture

As we mentioned, this paper concerns with software architecture analysis of modifiability. In the next two sections, we discuss our perspective on two concepts, viz. *modifiability* and *software architecture*.

2.1. Modifiability

Maintenance generally presents the major cost factor of the lifecycle of a software systems. Consequently, stakeholders are very interested in that the system is designed such that future changes will be relatively easy to implement, and thus decreases maintenance cost incurred by these changes. Typical questions that stakeholders pose during the early design stages, i.e. software architecture design, include:

- What alternative construction requires the lowest cost to maintain, i.e. cost to accommodate changes?
- What kind of effort will be needed to develop coming releases of the system?
- Where are the trouble spots in the system with respect to accommodating changes?

These questions concern a system's ability to be modified. In the software engineering literature a large number of definitions of qualities exist that are related to this ability. A recent classification of qualities is given in the ISO 9126 standard (ISO/IEC, 2000). This standard includes the following definition, related to modifying systems:

> Maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in environment, and in requirements and functional specification.

For our purpose the scope of this definition is too broad. It captures a number of rationales for system modifications in a single definition, i.e. bugs, changes in the environment, changes in the requirements and changes in the functional specification. We have limited ALMA to consider only the latter three of these change categories and get the following definition of modifiability, which we use in the remainder of the paper:

> The modifiability of a software system is the ease with which it can be modified to changes in the environment, requirements or functional specification.

This definition demonstrates the essential difference between maintainability and modifiability, namely that maintainability is concerned with the correction of bugs whereas modifiability is not.

2.2. Software architecture

The design and use of explicit software architecture has received increasing amounts of attention during the last decade. Typically, three arguments for defining a software architecture are used (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, the decisions captured in the soft-

ware architecture can be transferred to other systems. The work presented in this paper is concerned with the second aspect, i.e. early analysis of quality attributes, in particular modifiability.

A commonly used definition of software architecture is this one (Bass et al., 1998):

> The software architecture of a program or computer system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

This definition emphasizes that the software architecture concerns the structure of the system. The definition given by the IEEE (IEEE, 2000) emphasizes other aspects of software architecture:

> Architecture is 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.

This definition stresses that a system's software architecture is not only the model of the system at a certain point in time, but it also includes principles that guide its design and evolution.

In this paper we will consider a system's software architecture as the first design decisions concerning the system's structure: the decomposition of the system into components, the relationships between these components and the relationship to its environment. These design decisions may have a considerable influence on the quality characteristics of the resulting system. However, these qualities are not completely independent from each other, and may interact; a decision that has a positive effect on one quality might be very negative for another quality. Tradeoffs between qualities are inevitable and need to be made explicit during architectural design. This is especially important because architectural design decisions are generally very hard to change at a later stage. So, the rationale for software architecture analysis is to analyze the effect of these design decisions before it becomes prohibitively expensive to correct them.

3. Software architecture analysis

Most engineering disciplines have techniques and methods to predict quality attributes of the system being designed before it has been built. Prediction techniques can be used in every stage of the development process. Code metrics, for example, have been investigated as a predictor of the effort of implementing changes in a software system (Li and Henry, 1993). The drawback of this approach is that it can only be applied when a substantial part of the system's code has been written. Other authors investigate design-level metrics, such as metrics for object-oriented design (Briand et al., 1999). Such an approach is useful in the stages of development where (part of) the object-oriented design is finished.

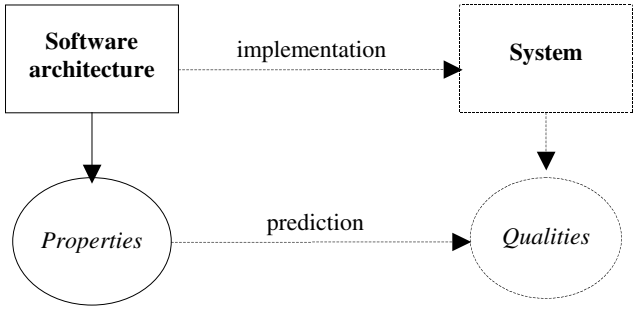

Fig. 1. Software architecture analysis.

The first steps in meeting the quality requirements are taken in the design of the software architecture. The design decisions strongly influence the quality levels that may be achieved and it is important to verify that the software architecture supports the required quality levels. To do so, we study the software architecture (indicated by the solid lines in Fig. 1). Based on these properties, the analysis aims at predicting the not yet implemented systems' qualities (indicated by the dotted lines in Fig. 1). Because the architecture design does not fully specify the system there are still many design decisions to be made once in later stages. This is a limitation on the predictions based on the software architecture. It cannot be guaranteed that subsequent design decisions will not worsen the level of a quality attribute in the implemented system.

Within the development process, software architecture analysis can be performed with two main perspectives of the assessors, i.e. internal and external. An external perspective is when an assessment team that is external to the development project carries out the analysis and is mainly used towards the end of the software architecture design phase for acceptance testing, or during the design phase as an audit instrument. An internal perspective is when the software architect performs software analysis as an integrated part of the software architecture design process. The selected perspective does not necessarily control the approach taken for the analysis, but rather influences the interpretation of the results.

In the following subsections we discuss the use of scenarios in software architecture analysis. Section 3.1 discusses the rationale for using scenarios and Section 3.2 gives an overview of existing software architecture analysis methods that are based on scenarios. Section

3.3 concludes this section and gives a rationale for the method discussed in this paper.

3.1. Using scenarios for software architecture analysis

Several techniques can be used for software architecture analysis. In a survey of such techniques (Abowd et al., 1996) make a distinction between questioning techniques and measurement techniques. To the first group belong scenarios, questionnaires and checklists and the latter group consists of metrics and simulations.

A number of the techniques are useful for analyzing modifiability. For instance, checklists are used for software architecture analysis to address issues related to different qualities, e.g. modifiability (AT&T, 1993). The checklists are used to check whether or not the development team follows 'good' software engineering practices. However, this does not guarantee that likely changes to the system will be easily implemented. This issue is addressed in scenario-based software architecture analysis methods, which investigate the effect of concrete changes to assess a system's modifiability.

Parnas introduced the basic principle of scenario-based analysis already in 1972 (Parnas, 1972). In scenario-based analysis, possible sequences of events are defined and their effect on the system is assessed. Although Parnas did not use the term scenario, currently such a description of a sequence of events is referred to as a scenario.

In modifiability analysis scenarios, or *change scenarios*, are used to capture future events that require the system to be adapted. This use of the term scenario differs from approaches where it is used solely to refer to usage scenarios, e.g. object-oriented design (Jacobson et al., 1992) with its use cases (scenarios) and scenario-based requirements engineering (Sutcliffe et al., 1998).

The main reason for using change scenarios in software architecture analysis of modifiability is that they are very concrete, enabling detailed analysis and statements about their impact. At the same time, this can pose a threat to our analysis if one does not select the set of change scenarios with care. The implication of having scenarios that are too specific is that the results of the analysis cannot be generalized to other instances of change. This issue is elaborated in Section 4.3.

3.2. Related scenario-based methods for software architecture analysis

There are currently a small number of scenario-based methods for software architecture analysis. One of the first methods that used scenarios was SAAM (Kazman et al., 1996). SAAM consists of four major steps: (1) develop scenarios, (2) describe candidate architecture(s), (3) evaluate scenarios and (4) overall evaluation. SAAM is stakeholder centric and does not focus on a specific quality attribute. The scenarios brought forward by the stakeholders determine which system qualities are investigated.

SAAM provides few explicit techniques for the different steps and much is left to the experience of the assessor. For instance, SAAM only indicates that the parties involved in the analysis should understand the description technique used; it does not prescribe any architectural views or information. The same holds for the scenario elicitation and evaluation steps. As a result, the repeatability of the analysis may be compromized.

SAAM has evolved into the Architecture Tradeoff Analysis Method (ATAM) (Kazman et al., 2000). The purpose of ATAM is to assess the consequences of architectural decisions compared to quality attribute requirements. ATAM uses scenarios for identifying important quality attribute requirements for the system and for testing the architectural decisions' ability to meet these requirements. The quality requirements are then related to architectural decisions, captured in Architecture-Based Architectural Styles (ABAS) to find tradeoff points. Decisions involving tradeoff points should be carefully examined. The emphasis of ATAM is not on an in-depth assessment of a single quality attribute, but rather on identifying *tradeoffs* between quality attributes.

The authors of this paper have focused on a single quality attribute, viz. modifiability. Bengtsson and Bosch (1999a) defined a method for architecture-level maintenance prediction. This method is based on a set of scenarios to represent the expected changes in the life cycle of the system. Each scenario is assigned a probability of occurrence to come to a change scenario profile. For each scenario it is analyzed how the change could be implemented and the amount of code the change would require and the total maintenance effort can be estimated.

Lassing et al. (1999a) extended the SAAM method to address the issue of repeatability, by defining techniques for the various process steps. The approach was to provide techniques on how to find those changes that are difficult to implement in the system, i.e. reveal the system's inflexibility.

3.3. Rationale for ALMA

ALMA is the result of combining the aforementioned methods (Bengtsson and Bosch, 1999a; Lassing et al., 1999a). The methods use a similar structure, and are both based on scenarios. The differences in techniques are largely caused by the differences in the goal of the analysis. This led us to define a unified process, which has the following characteristics:

- Focus on modifiability
- Distinguish multiple analysis goals

- Make implicit assumptions explicit
- Provide repeatable techniques for performing the steps

In ALMA we have deliberately chosen to limit ourselves to modifiability. The reason is that analyses of different quality attributes have their own specific problems and possibilities. It is important to analyze other quality attributes as well. In fact, to be able to make good architecture design it is a requirement. This means that for a complete assessment the method presented in this paper should be accompanied by architecture-level analysis methods for other quality attributes.

Architecture-level modifiability analysis can meet different goals. What technique that is the most appropriate techniques depends on this goal. For example, when the goal is to compare two different alternatives, we should use techniques that focus on revealing differences. ALMA distinguishes the following goals for architecture-level modifiability analysis: maintenance effort prediction, risk assessment, and comparison of candidate architectures.

The predecessors to ALMA focused on different goals. This led to differences in required information, different approaches to scenario elicitation, including the notion of a 'good scenario'. Now these assumptions are explicit decisions, which increases the repeatability of the method and improves the basis for interpreting the findings.

One of the important properties of ALMA is that it provides the analyst with techniques for the different steps of the method, relieving him of the burden to think of these for each analysis. Again, this increases the repeatability of the analysis process.

4. Five steps of ALMA

ALMA has a 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 scenarios: find the set of relevant scenarios
4. Evaluate scenarios: determine the effect of the set of scenarios
5. Interpret the results: draw conclusions from the analysis results

When performing an analysis, the separation between the tasks is not very strict and it is often necessary to iterate over various steps. Nevertheless, in the next subsections we will present the steps as if they are performed in strict sequence.

The different techniques in each step cannot be chosen a random because certain relationships exists. These are discussed in Section 4.6.

4.1. Goal setting

The first activity in ALMA is concerned with determining the goal of the analysis. In architecture-level modifiability analysis we can pursue the following goals:

- *Maintenance cost prediction*: estimate the effort that is required to modify the system to accommodate future changes
- *Risk assessment*: identify the types of changes for which the software architecture is inflexible
- *Software architecture selection*: compare two or more candidate software architectures and select the optimal candidate

4.2. Architecture description

In the second step of ALMA, architecture description, the information about the software architecture is collected. Generally speaking, modifiability analysis requires architectural information that allows the analyst to evaluate the scenarios. Scenario evaluation concerns two steps: analysis of the impact of the scenarios and expressing this impact.

Architecture-level impact analysis is to identify the architectural elements affected by a change scenario. This includes the components that are affected directly, but also the indirect effects of changes on other parts of the architecture. The effect of the scenario is expressed using some measurement scale, depending on the goal of the modifiability analysis.

The software architecture of a system has several aspects that cannot be captured in a single model (Hofmeister et al., 1999b). Instead they should be represented using architectural viewpoints (Hofmeister et al., 1999a; Kruchten, 1995). In architecture-level impact analysis the views that are used should provide information about the following:

- The decomposition of the system in components
- The relationships between components
- The relationships to the system's environment

The components in the architecture can be seen as a functional decomposition of the system, i.e. the allocation of the system's functions to different components. Relationships between components and between the system and its environment come in different forms and are often defined implicitly. Information about these dependencies is crucial to impact analysis as they determine whether modifications to a component will cause ripple effects or not. Not all dependencies between

components are known at the software architecture level. Some of them are introduced during lower-level design and implementation. These relationships may cause unforeseen ripple effects when modifications are made to a component. As a result, it is not always possible to determine the full impact of a scenario and this affects the overall analysis accuracy.

As mentioned in Section 2.2, the software architecture is the result of early design decisions; during architecture design the software architecture evolves: it is extended gradually with architectural decisions made over time. As a consequence, the amount of information available about the software architecture is dependent on the point in time at which the analysis is performed.

Initially, we base the analysis on the information that is available. If the information proves to be insufficient to determine the effect of the scenarios found, there are two options. First, it may be that it is not possible to determine the effect of the scenario precisely. Alternatively, the architect that assists in scenario evaluation may fill in the missing information.

4.3. Change scenario elicitation

Change scenario elicitation is the process of finding and selecting the change scenarios to be used in the evaluation step. Eliciting change scenarios involves such activities as identifying stakeholders to interview, documenting the scenarios that result from those interviews, etc. Although the elicitation shares some aspects with other fields, e.g. in requirements engineering scenarios are elicited from stakeholders and documented for later use as well (Sutcliffe et al., 1998), there are some issues of specific concern. The first issue is that the number of possible changes to a system is almost infinite. In order to make scenario-based software architecture analysis feasible, we use a combination of two techniques: (1) equivalence classes and (2) classification of change categories. Partitioning the space of scenarios into equivalence classes enables us to treat one scenario as a representative of a class of scenarios, thus limiting the number of scenarios that have to be considered. However, not all equivalence classes are just as relevant for each analysis. Deciding on important change scenarios requires a selection criterion. The other technique, classification of change categories, is used to focus our attention on the scenarios that satisfy this selection criterion. We can employ two approaches for selecting a set of scenarios: top-down and bottom-up. When we use a top-down approach, we use some predefined classification of change categories to guide the search for change scenarios. This classification may derive from the domain of interest, knowledge of potentially complex scenarios, or some other external knowledge source. In interviews with stakeholders, the analyst uses this classification scheme to stimulate the interviewee to bring forward relevant scenarios. This approach is top-down, because we start with high-level classes of changes and then descend to concrete change scenarios.

When using a bottom-up approach, we do not have a predefined classification scheme, and leave it to the stakeholders being interviewed to come up with a relevant set of scenarios. The analyst then categorizes the scenarios and stakeholders review this categorization. This approach is bottom-up, because we start with concrete scenarios and then move to more abstract classes of scenarios, resulting in an explicitly defined and populated set of change categories.

In practice, we often iterate between the approaches such that the elicited change scenarios are used to build up or refine the classification scheme. This (refined) scheme is next used to guide the search for additional scenarios. In addition to the selection criterion, elicitation also need a stopping criterion to decide when we have collected a representative set of scenarios. We have found a sufficient number of change scenarios when: (1) we have explicitly considered all change categories and (2) new change scenarios do not affect the classification structure.

The second issue is that of deciding on desired change scenarios, i.e. the selection criterion. The selection criterion for scenarios is closely tied to the goal we pursue in the analysis:

- If the goal is to estimate maintenance effort, we want to select scenarios that correspond to changes that have a high probability of occurring during the operational life of the system.
- If the goal is to assess risks, we want to select scenarios that expose those risks.
- If the goal is to compare different architectures, we follow either of the above schemes and concentrate on scenarios that highlight differences between those architectures.

Sections 5–7 show how these criteria are translated into techniques for scenario elicitation.

ATAM also classifies scenarios into *direct* and *indirect* scenarios (Kazman et al., 1999) to guide the elicitation process. Direct scenarios represent uses of the system and thus are not unlike use cases (Jacobson et al., 1992). Direct scenarios are used in architecture analysis to elicit information about the architecture. Indirect scenarios represent changes to the system, i.e. change scenarios. Additionally ATAM distinguishes two classes of indirect scenarios: growth scenarios and exploratory scenarios (Kazman et al., 2000). Growth scenarios represent anticipated future changes, and are similar to the scenarios we look for to predict maintenance effort. Exploratory scenarios stress the system; their goal is to expose the limits of the current design. Exploratory scenarios resemble the scenarios we look for to assess

risk. However, certain types of changes we consider complex, such as those involving different owners, need not expose boundary conditions of the system being analyzed. ATAM uses two mechanisms to structure the set of scenarios as well as the elicitation process: utility trees and facilitated brainstorming. Utility trees provide a top-down mechanism, typically guided by the top-down decomposition of relevant quality attributes in the assessment. Our top-down mechanism only concerns modifiability, and is organized differently. Facilitated brainstorming is a bottom-up mechanism like the one we employ.

4.4. Change scenario evaluation

ALMA's next step is to evaluate the effect of the change scenarios on the architecture. In this step, the analyst cooperates with the architects and designers. Together with them the analyst determines the impact of the change scenarios and expresses the results in a way suitable for the goal of our analysis. This is architecture-level impact analysis.

To the best of our knowledge no architecture-level impact analysis method has been published yet. A number of authors (Bohner, 1991; Kung et al., 1994) have discussed impact analysis methods focused on source code. In general, impact analysis consists of the following steps:

1. Identify the affected components
2. Determine the effect on the components
3. Determine ripple effects

The first step is to determine the components that need to be modified to implement the change scenario. The second step is concerned with identifying the functions of the components that are affected by the changes. Changes may propagate over system boundaries; changes in the environment may impact the system or changes to the system may affect the environment. So, we also need to consider the systems in the environment and their interfaces.

The third step is to determine the ripple effects of the identified modifications. The occurrence of ripple effects is a recursive phenomenon in that each ripple *may* have additional ripple effects. Because not all information is available at the architecture level, we may have to make assumptions about the occurrence of ripple effects.

At one extreme, we may assume that there are no ripple effects but this is a too optimistic assumption. At the other extreme, we may assume that each component related to the affected component requires changes but this is a too pessimistic assumption. In practice, we rely on the architects and designers to determine whether adaptations to a component have ripple effects on other components.

However, empirical results on source code impact analysis indicate that software engineers, when doing impact analysis predict only half of the necessary changes (Lindvall and Sandahl, 1998). Lindvall and Runesson (1998) report the results from a study of the changes made to the source code between two releases and whether these changes are visible in the object-oriented design. Their conclusion is that about 40% of the changes are visible in the object-oriented design. Some 80% of the changes would become visible, if not only the object-oriented design is considered but also the method bodies. These results suggest that we should expect impact analysis at the architectural level to be less complete in the sense that not all changes will be detected. This issue should be taken into account in the interpretation of the results.

Next, we must express the results of the impact analysis either quantitatively, by describing the changes needed, or qualitatively by using quantitative measures. For instance, we can rank the effects of scenarios on a five level scale (++, +, +/−, −, −−). This allows us to compare the effect of scenarios. We can also express the effort required for a scenario, such as an estimate of the modification size using metrics like lines of code, or function points. Sections 5–7 discuss change scenario evaluation for each of the analysis goals.

4.5. Interpretation

When we have finished the change scenario evaluation, we need to interpret the results to draw our conclusions concerning the software architecture. The interpretation of the results depends entirely on the goal of the analysis and the system requirements.

4.6. Relating the steps

The techniques used for the various steps cannot be chosen at random (Fig. 2).

Based on the goal of the analysis, we select the scenario elicitation approach. When the goal is to estimate the maintenance costs for a system, we require a set of scenarios that is representative for the actual events that will occur in the future of the system. On the other hand when the goal is risk assessment we are interested to find scenarios that have complex associated changes.

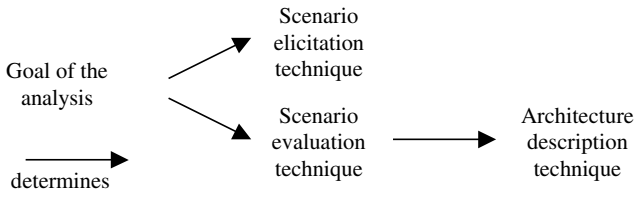

Fig. 2. Relating techniques.

Similar goes for the scenario evaluation technique. If the goal of the analysis is to compare two or more candidate software architectures, we need to express the impact of the scenario using at least an ordinal scale to indicate a relation between the different candidates, i.e. which software architecture supports the changes best.

The evaluation technique then determines what information should be used for the description of the software architecture. For instance, if we want to express the impact of the scenario using the number of lines of code that is affected, the description of the software architecture should include a size estimate for each component.

In the following sections we illustrate the use of ALMA using a number of examples of analysis of modifiability. Section 5 illustrates maintenance prediction based on an analysis of a telecommunications system that we performed at Ericsson Software Technology AB. In the same way, risk assessment is illustrated in Section 6 based on an analysis of a logistic system that we performed at DFDS Fraktarna AB. Finally, architecture selection is illustrated in Section 7, using an analysis of an embedded system that was performed at EC-Gruppen AB.

5. Case study I. Maintenance prediction

We illustrate how to use ALMA for maintenance prediction in this article with a case study we performed at Ericsson Software Technology AB. Ericsson carries the Mobile Positioning Center (MPC) as one of their products for locating mobile phones in a cellular network and reporting their geographical position. The MPC extends a cellular telecom network with a mobile device positioning service, and enables network operators to implement services using the positioning information provided by the MPC.

5.1. Goal setting

The goal of the analysis of the MPC system was to get a prediction of the costs of modifying the system for changes that were likely in the future. For this we need to find scenarios that are likely to occur in the future of the system, estimates of the amount of change required for each scenario, and a model to derive the prediction. The following sections describe how this works and illustrate the process using the analysis that we performed of the MPC system.

5.2. Architecture description

Before the evaluation of the scenarios may start we need a description of the software architecture. In case of maintenance prediction we need information to estimate the amount of modifications required to realize the scenario. First, we need to be able to identify in what parts of the architecture a specific function is realized. For this task we may use the logical view (Kruchten, 1995), or the conceptual and module view (Hofmeister et al., 1999a). After we have determined what functions and parts need to be modified, we determine the ripple effects. For this task we also need information about the relationships between the parts, which can, for the most part, be found in the aforementioned views.

In addition to the basic information about the architecture we also need size estimates of the components to derive our prediction from. We should choose the same size metric as already in use for time and effort estimates in the project planning. The size estimates need to conform to the same scale unit as is used for the productivity measures in the project or company, e.g. lines of code, or function points.

In the MPC case, the architecture-level design documentation was available for all the basic architecture information described in Section 4.2. The architecture descriptions contained the following views:

- System environment that presents the system's relation to the environment.
- Sequence diagrams that present the behavior for key functions in the system.
- An architecture overview presenting the how the system is conceptually organized in layers.
- Component view that presents the components in the system.

The size estimates of the MPC components needed for the prediction were obtained by asking the architect to estimate the component sizes for the proposed architecture. The architect used the size data from the previous project as a frame of reference. The estimates were expressed as lines of code in the final implementation of this version of the product. For example, the Http Adaptor component was estimated at 2 kLOC and the Protocol Router at 3 kLOC.

5.3. Change scenario elicitation

The next step is to elicit a set of change scenarios. First, we have to decide which stakeholders to interview for scenarios. When selecting the stakeholders to interview it is important to select persons that have different responsibilities to cover as many relevant angles on the system's future changes as possible. The following stakeholder areas or their equivalents should at least be considered: architecture design, product management, marketing and sales, help desk or customer training, installation and maintenance, and design and implementation.

Projects that develop software for mass markets are dependent on indirect knowledge about customer and market demands, viz. marketing, sales and support.

For the MPC analysis we selected and interviewed the following stakeholders of the MPC to elicit change scenarios:

- the software architect,
- the operative product manager,
- a designer involved in the MPC system's development, and
- an applications designer developing services using the MPC.

The preferred elicitation technique in this case is bottom-up, since it is assumed that the stakeholders have the better knowledge about likely scenarios. The following scenarios are examples from the total of 32 change scenarios that were elicited during the interviews with the MPC stakeholders:

- Change MPC to provide a digital map with position marked
- Add geodetic conversion methods to the MPC

When the stakeholders have been interviewed we synthesize all the scenarios and categorize them. Duplicate scenarios are removed. The categories should be specific to the case and help determine if any important categories of scenarios were missed. We present the complete and categorized list of scenarios to the stakeholders, and ask them to revise the list if they find it necessary. The revision process is repeated until no stakeholder makes any more changes. After incorporating the comments from the MPC stakeholders we came to the following five categories of change scenarios:

- Distribution and deployment changes
- Positioning method changes
- Changes to the mobile network
- Changes of position request interface/protocol
- In-operation quality requirements changes

For the purpose of maintenance prediction, there is one more step to perform. The probability of the scenarios to occur must be estimated. We call this scenario weight since it is used in the prediction to determine the scenario's influence on the end result. We collect the weights for each scenario by asking the appropriate stakeholder to estimate the number of times changes of the type represented by the scenario can be expected. When the weights have been decided we normalize them, $\mathrm{NW}(S_n)$, by dividing the estimated number of changes, or weight, of the scenario, $W(S_n)$, by the sum of the weights of all the scenarios (Eq. (1)). The result of the normalization should be that all scenarios now have a weight between zero and one, and that the sum of all scenario weights is exactly one. We call the list of scenarios with normalized weights the scenario profile. Normalizing scenario weights:

$$\mathrm{NW}(S_n) = W(S_n) \Big/ \sum_n W(S_n) \tag{1}$$

To obtain the weights for the MPC scenarios we decided to ask the operative product manager since the future development of the product mainly is the responsibility of that role. The probability estimates were obtained by interviewing the operative product manager and for each scenario in the final scenario profile asking how many times an instance of the scenario was likely to occur. Table 1 contains the weights of two of the scenarios.

5.4. Change scenario evaluation

After we have elicited a set of change scenarios and their weights, we evaluate their effect. First, we need to understand the scenario and the functions that are required to realize the scenario. Some of these functions may already be present in the architecture and we need to determine if they must be modified for this scenario, or not. Some functions may need to be added and we should determine where those functions should be added in the architecture; either they can be added to existing components, in which case it is a kind of modification, or they can be added as separate components. To this

Table 1
The impact and the calculation of average effort per scenario

ID	Change scenario	Weight	Impact	
			Modified	New
1	Change MPC to provide a digital map with position marked	0.0345	25% of 10 kLOC + 10% of 25 kLOC = 5 kLOC	5 kLOC = 5 kLOC
2	New geodetic conversions methods per installation	0.0023	10% of 10 kLOC + 75% of 5 kLOC = 4.75 kLOC	None
...	...	...	...	...
Sum:			0.0345 * 5 kLOC + 0.0023 * 4.75 kLOC = ~183 LOC/scenario	0.0345 * 5 kLOC = ~173 LOC/scenario

end, we should consult the software architecture descriptions and possibly the architect if the descriptions are inconclusive. Once the functions have been identified we must trace the effects of the identified changes, and determine the ripple effects. To use the analysis results in the prediction we must express the impact estimates for each scenario as:

- The size of the modification to existing components. This can be derived from an estimate of the ratio of change for the modified components, e.g. half of the component needs to be changed, and then using the component size to derive the change volume.
- The estimated size of components that need to be introduced.

Obviously, the change volume must be expressed by the same unit of scale as you used in the previous steps, e.g. lines of code, function points or object points.

The results in our case were, per change scenario, a set of estimates of the modification volume for each affected component, as depicted in Table 1. Note that the table excludes 30 scenarios for business sensitivity reasons, which makes the figures somewhat different than in the real case.

We separate the modifications made to existing components from the new components based on the hypothesis that the productivity of writing new code is higher than the productivity of making modifications to existing code.

5.5. Interpretation

Prediction of maintenance effort requires a model that is based on the cost drivers of your maintenance processes. Organizations have different strategies for doing maintenance and this affects how the cost and effort relate to the change traffic and modified components. For instance, three different maintenance strategies are found to have significant differences in cost (Stark and Oman, 1997). To come to the prediction we need to decide on a prediction model. We propose a simple model (Eq. (2)) that assumes that the change volume is the main cost driver and that we have a productivity figure for the cost of adding new code and modifying old code. For each scenario we have two impact size estimates, changed code, CC, and new code, NC, and a weight. We also have the productivity estimates for these two types of changes. The individual sums of the product of the impact size estimates and the weight of the scenario multiplied with their associated productivity, P_{cc} and P_{nc}, make up the total effort needed for the changes in the scenario profile. To get the average, this number is divided by the number of scenarios in the scenario profile, $C(S)$. The total effort for the period is obtained by multiplying the average effort with the expected number of changes. Total maintenance effort calculation:

$$\text{Total effort} = \frac{\sum_{\text{IA}} \left(\left(\sum_{CC_j} (\text{size}_j \cdot \text{weight}_j) \right) \cdot P_{cc} + \left(\sum_{NC_j} (\text{size}_j \cdot \text{weight}_j) \right) \cdot P_{nc} \right)}{C(S)} \cdot \text{changes}_{\text{estimated}} \tag{2}$$

In the analysis of the MPC system, each release of the system is carried out as its own development project. The strategy is to make changes by designing the new version of the system based on the current one, making the modifications to the code baseline and then testing the system. For this assessment we assume that the overall productivity per software engineer, i.e. not only writing the source statements, in this process is on the same level as those published in the software engineering literature:

- 40 LOC/month for modifying existing code (Henry and Cain, 1997), and
- 250 LOC/month for adding new code (Maxwell et al., 1996).

These productivity measurements vary between projects and companies and the numbers used here should not be taken as universally applicable. Instead, the productivity data from the organization that is responsible for the system's maintenance should be used when using this method. If there are other significant cost drivers in maintenance processes, the prediction model should be modified. When we decide on a different model we should make sure that all the necessary information is available from the previous steps.

5.6. Conclusions

In this section we have presented the techniques for the steps in ALMA to perform maintenance predictions. The software architecture description is augmented with size estimates. Concerning the prediction we present a prediction model based on the estimated change volume and productivity ratios. The techniques presented are based on our experience with architecture analysis (Bengtsson and Bosch, 1999a,b, 2000; Lassing et al., 2001).

Although the result of the prediction is an effort estimate, we currently lack a frame of reference. The question is how good the results are in comparison to other architecture alternatives, e.g. the competitors. We have addressed this issue by also providing optimal- and worst-case predictions for the architecture and thus creating the frame of reference from the same assessment (Bosch and Bengtsson, 2001).

6. Case study II. Risk assessment

The second case study in which we applied ALMA concerns the architecture analysis that we performed for DFDS Fraktarna AB, a Swedish distributor of freight. The system that we investigated is a business information system called EASY and it will be used to track the position of groupage (freight with a weight between 31 kg and 1 ton) through the company's distribution system. During our analysis, EASY was under development by Cap Gemini Ernst & Young.

6.1. Goal setting

The goal pursued in this case study is risk assessment: we investigate the architecture to identify modifiability risks. This meant that the techniques that we use in the various steps of the analysis are aimed at finding complex change scenarios. The subsequent sections discuss how this works.

6.2. Architecture description

To come to a description of EASY's software architecture, we studied the available documentation and interviewed one of the architects and a designer. For risk assessment it is important that the architecture description contains information to determine whether a change scenario will be complex to implement. In earlier work (Lassing et al., 1999a) we have distinguished four viewpoints that are useful for doing so. We claim that a distinction should be made between internals of the system and its environment when the system is part of a larger whole. We call the internals of the system the *internal* architecture and the system in its environment the *external* architecture.

The reason why the environment is included in the description is that this allows the analyst to determine dependencies between the system and its environment. The environment is a complicating factor in the implementation of changes, as well as a source of changes. Either, the modifiability of the system is threatened, or the environment causes changes for which the system should be modifiable. So, for risk assessment the environment should be included in the description. The following viewpoints are distinguished for the system's environment, i.e. the external architecture level (Lassing et al., 1999c):

- The context viewpoint: an overview of the system and the systems in its environment with which it communicates. We determine the owners of other systems in the system's context. The owner of a system is the person or organizational unit that is (financially) responsible for the system.
  In the evaluation this view is used to assess which systems have to be adapted to implement a change scenario and who is involved in these adaptations.
- The technical infrastructure viewpoint: an overview of the system's dependencies on technical elements, e.g. operating system, database management system, etc. Common use of infrastructure elements brings about dependencies between these systems: when a system owner decides to make changes to elements of the technical infrastructure, this may affect other systems as well.

At the internal architecture level, the internals of the system are addressed. This allows us to determine the required changes to the system. To this end, the following two viewpoints are distinguished:

- The conceptual viewpoint: an overview of the high-level design elements of the system, representing concepts from the system's domain.
- The development viewpoint: an overview of decisions related to the structure of the implementation of the system. We use this viewpoint in the analysis of modifiability to determine whether adaptations to a system are limited to a certain type of component. For instance, if business logic and data access are implemented in separate components, then the impact of changes to the underlying data model can be confined to the data access components.

The latter two viewpoints were identified before as the logical view (Kruchten, 1995) and the development view (Hofmeister et al., 1999a) as the conceptual and code architecture.

Fig. 3 shows the context view for EASY. This figure shows that EASY communicates with four other systems:

- Domestic freight system (DOM): the system that handles registration of domestic freight
- Trans/400: the system for registration of international freight
- TANT: the system storing all information about the location of groupage
- Economy: system for handling financial data

The details of the communication between EASY and these systems should be added in a textual description.

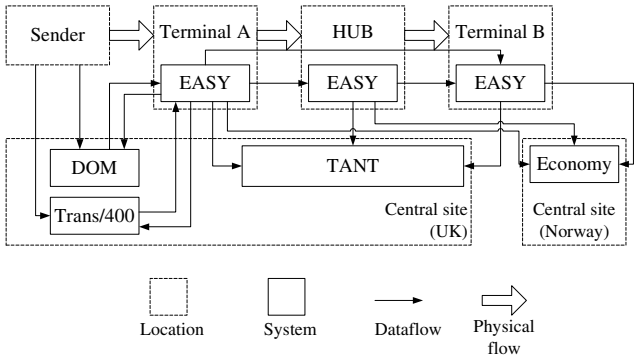

Fig. 3. Context view of EASY.

For instance, we should add that the communication between the systems takes place through messages using an asynchronous, message-oriented, middleware.

The conceptual view of EASY is shown in Fig. 4. This figure shows that the system is divided into three layers: the presentation layer, the business layer and the data layer. The presentation layer consists of all devices that interact with users, the business layer contains all business specific functionality and the data layer manages the persistent data of the system.

6.3. Change scenario elicitation

In change scenario elicitation for risk assessment, the aim is to elicit scenarios that expose risks. The first technique that we can use is to explicitly ask the stakeholders to bring forward change scenarios that they think will be complex to realize. We found that, in general, this is hard for stakeholders; they tend to bring forward change scenarios that they already anticipated when designing the architecture (Lassing et al., 2001).

Another elicitation technique is that the analyst guides the interviews based on his knowledge of the architectural solution, as obtained in the previous step, and knowledge of complex changes. Although the interviewer guides the elicitation process, it is important that the stakeholders being interviewed volunteer the change scenarios. This prevents irrelevant scenarios from being included in the analysis.

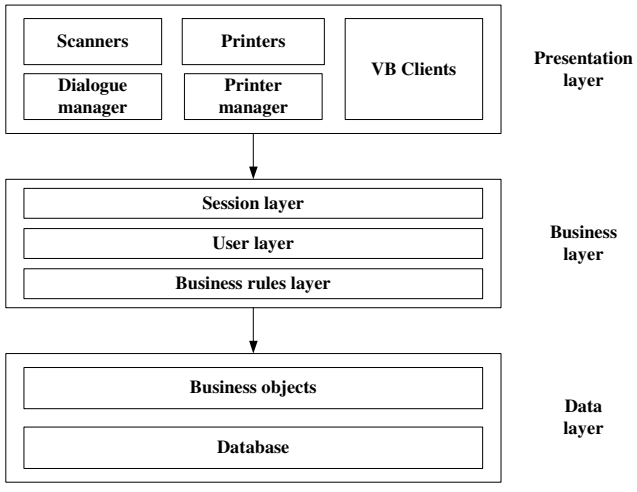

Fig. 4. Conceptual view of EASY.

One of the sources of knowledge of complex scenarios is the analyst's experience with architecture analysis. For instance, we found change scenarios to be more complex generally when initiated by the owner of the system and require other systems to be adapted (Lassing et al., 1999b). Assisted by this knowledge, the analyst may ask the stakeholders during scenario elicitation for such change scenarios, i.e. 'Could you think of a change to the system that affects other systems as well?'. We have formulated a classification scheme that includes the categories of change scenarios found to be complex in the domain of business information systems (Lassing et al., 1999b). This scheme assists the analyst during scenario elicitation. Such schemes are most likely domain specific; changes that are considered complex in one domain are not necessarily complex in another domain as well. The categories that we distinguish in the domain of business information systems are:

- Change scenarios that are initiated by the owner of the system under analysis, but require adaptations to other systems.
- Change scenarios that are initiated by others than the owner of the system under analysis, but do require adaptations to that system.
- Change scenarios that require adaptations to the external architecture.
- Change scenarios that require adaptations to the internal architecture.
- Change scenarios that introduce version conflicts.

These change scenarios may come from a number of sources:

- Changes in the functional specification
- Changes in the requirements
- Changes in the technical environment
- Other sources

In the interviews, the analyst tries to cover all the sources of changes for all the categories of changes.

For change scenario elicitation in the analysis of EASY we interviewed three stakeholders, viz. a software architect, a software designer and a representative of the owner of the system. These interviews took approximately 1–2 h. In these interviews we employed a top-down approach; we directed the interviews towards complex scenarios using the above-mentioned framework.

One of the change scenarios found in the elicitation step of the analysis of EASY is that the middleware used

by the TANT system changes. The effect of this scenario is that EASY has to be adapted to this new middleware as well, because otherwise it will no longer be able to communicate with the TANT system. This change scenario falls in the category 'Change scenarios that are initiated by others that the owner of the system under analysis, but do require adaptations to that system' and its source is a change in the technical environment. This scenario represents the equivalence class of all changes to the middleware, i.e. it is not specific for a special type of middleware. Similarly, we tried to discover change scenarios for all other categories. This process resulted in 21 change scenarios, a number of which are shown in Table 2.

The elicitation framework also serves as the stopping criterion for scenario elicitation: we stop the elicitation process when all cells have been explicitly considered. To do so we use an iterative process; after each interview we use the framework to classify the scenarios found. In the next interviews we focus on the cells that are scarcely populated or empty. By going through all cells in the scheme we are able to judge whether all risks are considered.

We used this approach in the analysis of EASY; the end result is shown in Table 3. The table includes a '+' in the cells for which we found one or more change scenarios and a '−' in the cells for which we did not find any change scenario. One of the striking things about this table is that it includes a large number of minus signs. One of the reasons for this is that the integration between EASY and other systems is not very tight. As a result, we found relatively few scenarios associated to the environment.

Table 2
Examples of change scenarios for EASY

Source	Change scenario
Technical infrastructure	Change of middleware used for TANT
Technical infrastructure	Use of different type of scanners
Technical infrastructure	Change operating system used for EASY
Functional specification	Introduction of new terminals
Requirements	Substantially more parcels to be handled
Requirements	Reducing the number of scan points of freight

6.4. Change scenario evaluation

For risk assessment it is important that the results of the scenarios are expressed in such a way that it allows the analyst to see whether they pose any risks. The complexity of a change scenario is determined in the following steps (Lassing et al., 1999a):

1. *Initiator of the changes*: the first step in the evaluation of the change scenarios is to determine the initiator of the scenario.
2. *Impact level*: the second step is to determine the extent of the required adaptations. To do so, we use the architecture description and perform architecture-level impact analysis in cooperation with members of the system's development team. In maintenance prediction, the number of lines of code is used to express the impact, but in risk assessment we limit ourselves to four levels of impact: (1) the change scenario is already supported and has no impact; (2) the change scenario affects a single component; (3) the change scenario affects several components, and (4) the change scenario affects the software architecture. In expressing the impact, we make a distinction between the effect at the internal architecture level and the effect at the external architecture level, i.e. the effect on the internals and the effect on the environment. The result of this step consists of two measures: the internal architecture-level impact and the external architecture-level impact.
3. *Multiple owners*: the next step in the evaluation of the scenario is to determine who is involved in the implementation of the changes required for the scenario. Components might be owned by other parties and used in other systems and contexts as well. Changes to these components have to be negotiated and

Table 3
Classification scheme for eliciting complex change scenarios

	Change in the functional specification	Change in the requirements	Change in the technical environment	Other sources
Change scenarios that require adaptations to the system and these adaptations have external effects	+	−	−	−
Change scenarios that require adaptations to the environment of the system and these adaptations affect the system	+	−	+	−
Change scenarios that require adaptations to the external architecture	−	−	−	−
Change scenarios that require adaptations to the internal architecture	+	+	−	−
Change scenarios that introduce version conflicts	−	−	−	−

synchronized with other system owners before they can be incorporated.

4. *Version conflicts*: the final step in the evaluation of the change scenario is to determine whether the scenario leads to different versions of a component and whether this introduces additional complexity. The occurrence of a version conflict is expressed on an ordinal scale: (1) the change scenario introduces no different versions of components, (2) the change scenario does introduce different versions of components or (3) the change scenario creates version conflicts between components. The introduction of different complicates configuration management and requires maintenance of the different versions.

In the analysis of EASY, we used this model to express the effect of the change scenarios acquired in the previous step. One of the scenarios that we found is that the owner of TANT changes the middleware used in the TANT system. The first step is to determine the initiator of this change scenario. The initiator of this scenario is the owner of TANT.

The next step is to determine the required changes, both at the external architecture level and at the internal architecture level. At the external architecture level, TANT itself and all systems that communicate with it through middleware have to be adapted. Fig. 3 shows that EASY communicates with TANT, so it has to be adapted. EASY and TANT have different owners meaning that coordination between these owners is required to implement the changes: a new release of TANT can only be brought into operation when the new release of EASY is finished, and vice versa.

The next step is to investigate the internal architecture of EASY, as it has to be adapted for this scenario. It is not apparent from the architecture descriptions, which components have to be adapted. We have to consult the architect to find out which components access TANT using TANT's middleware. It turns out that access to TANT is not contained in one component, which means that several components have to be adapted for this scenario. These components have the same owner. The last step is to find out whether the scenario leads to version conflicts. We found that of each component only one version will exist, so there will not be any version conflicts.

We applied the same procedure to the other change scenarios. In Table 4 the results of some of them are expressed using the measurement instrument.

6.5. Interpretation

For risk assessment, the analyst will have to determine which change scenarios are risks with respect to the modifiability of the system. It is important that this interpretation is done in consultation with stakeholders. Together with them, the analyst estimates the likelihood of each scenario and whether the required changes are too complicated. The owner of the system should decide on the criteria to be used.

We focus on the change scenarios that fall into one of the risk categories identified in the previous section. In the analysis of EASY, for instance, one of the change scenarios that may pose a risk is that the middleware of TANT is changed. This scenario requires changes to systems of different owners including EASY, for which a number of components have to be adapted. Based on that, the stakeholders classified this change scenario as complex. However, when asked for the likelihood of this scenario they indicated that the probability of the change scenario is very low. As a result, the change scenario is not classified as a risk. Similar was done for the other scenarios and the conclusion was that two of the scenarios found could be classified as risks.

6.6. Conclusions

This section concerns risk assessment using architecture analysis. For architecture description we distinguish four viewpoints that are required in architecture-level modifiability analysis for business information systems. For change scenario elicitation we have presented a classification framework, which consists of categories of change scenarios that have complicated associated changes.

The viewpoints, the classification framework and the evaluation instrument result from our experiences with architecture analysis (Lassing et al., 1999a,c, 2001). However, for interpretation we do know which information is required but we do not have a frame of reference to interpret these results. In its current form this step of the method relies on the stakeholders and the

Table 4
Evaluation of scenario

Scenario	Initiator	External architecture level			Internal architecture level		
		Impact level	Multiple owners	Version conflicts	Impact level	Multiple owners	Version conflicts
TANT middleware replaced	Owner of TANT	3	+	1	3	–	1
Operating system replaced	Owner of EASY	1	–	1	4	–	1
Reduction of scan-points	Owner of EASY	2	+	1	3	–	1

analyst's experience to determine whether a change scenario poses a risk or not.

7. Case study III. Software architecture comparison

In the third case we used ALMA to compare two architecture versions between design iterations of a beer can inspection system. The beer can system is a research system developed as part of a joint research project between the company EC-Gruppen AB and the software architecture research group at the Blekinge Institute of Technology. Bengtsson and Bosch (1998) have reported on this project and here we only present the assessment activities from the case. The inspection system is an embedded system located at the beginning of a beer can fill-process and its task is to remove dirty or damaged beer cans from the input stream. Clean cans should pass the system without any further action.

7.1. Goal setting

The goal with the analysis was to compare two alternative architecture designs. In this section we illustrate how to use ALMA to compare the new version of the architecture with the old to confirm that the new version was indeed an improvement over the previous version. In total we made six design iterations each with an associated analysis. In comparing candidates we may of course choose to perform a prediction for each candidate and compare the results, or perform a risk assessment and compare the results. The third option is to compare the candidates using scenarios that are expected to reveal the critical differences between the candidates, e.g. scenarios describing changes that are extreme in pushing the limits of the system. For this we need change scenarios that highlight the differences between the candidates. In this section we show how this works.

7.2. Architecture description

To compare two software architecture candidates we need descriptions of both. Each description should provide the basic information needed to evaluate the scenarios, viz. system component decomposition, component relations, and relations with the system's environment (Section 4.2).

In the beer can case we had already the logical view (Kruchten, 1995) that described the system's decomposition and the relations of the components using very basic UML-like notation, illustrated in Figs. 5 and 6. The system is an embedded control system with no firm relations to other software systems and therefore there was no need to describe the latter in an architectural view. Although the logical view was described rather

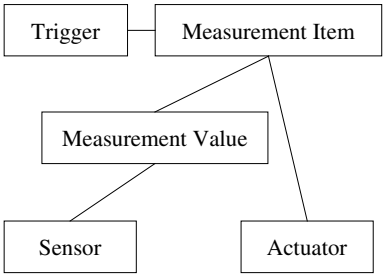

Fig. 5. Beer can inspection architecture after first iteration.

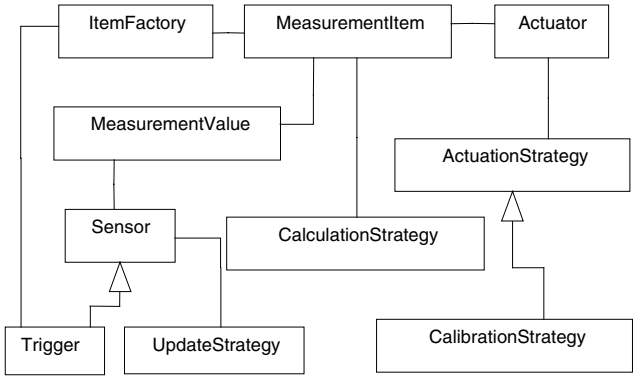

Fig. 6. The final beer can inspection architecture.

rudimentarily, it did not present any major problem, since we, the assessors, were also part of the design process we also had the informal information available to us.

7.3. Change scenario elicitation

The comparison of two candidate architectures requires a single common frame of reference. In ALMA the frame of reference is the set of change scenarios that is used to assess the modifiability of all the candidate architectures. The scenario set is the representation of the modifiability requirements that are the same for all candidates.

In change scenario elicitation for comparing candidate software architectures, the aim is to elicit scenarios that reveal the relevant differences between the candidates. The idea is to stress test the candidates. Similar to the other two goals, we need to decide which stakeholders should contribute to the scenario list. The stakeholders are then interviewed and asked to enumerate the scenarios they consider critical or extreme for the system in relation to modifiability. The contributors are then asked to review and revise the synthesized list in the same way as for the maintenance prediction.

Because the beer can case was a research project, there were only us, the architects, to interview for scenarios. The scenarios were then elicited by discussing

Table 5
Results of the analysis

	Iteration no.						Comment
	Initial	1	2	3	4	Final	
Change scenario 1	1/4	1/5	1/6	2/9	3/9	2/10	Slightly improved
Change scenario 2	4/4	4/5	3/6	2/9	3/9	2/10	Improved
Change scenario 3	4/4	5/5	6/6	9/9	2/9	2/10	Greatly improved from 3 to 4
Change scenario 4	4/4	3/5	3/6	3/9	3/9	3/10	Improved
Change scenario 5	4/4	5/5	6/6	9/9	9/9	10/10	Same

them among ourselves and we reached the stopping criterion when we considered the coverage of the scenarios to be complete. The scenarios we found were the following:

1. The types of input or output devices used in the system are excluded from the suppliers' assortment and need to be changed. The corresponding software needs to be updated. Modifications of this category should only affect the component interfacing the hardware device.
2. Advances in technology allow a more accurate or faster calculation to be used. The software needs to be modified to implement new calculation algorithms.
3. The method for calibration is modified, e.g., from user activation to automated intervals.
4. The external systems interface for data exchange change. The interfacing system is updated and requires change.
5. The hardware platform is updated, with new processor and I/O interface.

7.4. Change scenario evaluation

The effect of each scenario in the profile is evaluated against all candidate architectures. The evaluation is done by applying the impact analysis described in Section 4.4: identify the affected components, determine the effect on those components, and determine ripple effects. We may choose to evaluate and express the results in one of the following ways:

- For each scenario, determine the candidate architecture that supports it best, or conclude that there are no differences. The results are expressed as a list of the scenarios with the best candidate for each scenario.
- For each scenario, rank the candidate architectures depending on their support for the scenario and summarize the ranks.
- For each scenario, determine the effect on the candidate architectures and express this effect using at least an ordinal scale, e.g. a five level scale such as the one we introduced earlier, or the number of lines of code affected. If the comparison is based the predicted maintenance effort, use the techniques described in Section 5.4 and if the comparison is based on modifiability risk, use the same techniques described in Section 6.4.

In the beer can inspection case we chose to express the impact as the component modification ratio, i.e. modified components divided by the total number of components. For each new version we evaluated the scenarios and compared to the previous version. Table 5 shows the initial result, the results after each of the iterations and the final result.

7.5. Interpretation

When the goal of the analysis is to compare candidate architectures, the interpretation is aimed at selecting the best candidate. In Section 7.4 we distinguished three approaches to compare candidate architectures: (1) appoint the best candidate for each scenario, (2) rank the candidates for each scenario, and (3) estimate the effort of each scenario for all candidates on some scale.

When using the first approach, the results are interpreted such that the architecture candidate that supports most scenarios is considered best. In some cases we are unable to decide on the candidate that supports a change scenario best, for instance because they require the same amount of effort. These scenarios do not help discriminate between the candidates and do not require any attention in the interpretation. This interpretation of the results allows the analyst to understand the differences between the candidate architectures in terms of effects of concrete changes.

When we rank the candidates for each scenario, we get an overview of the differences between the candidate architectures. Based on some selection criterion, we then select the candidate architecture that is most suitable for the system. For instance, we may select the candidate that is considered best for most scenarios. Or, alternatively, we may choose the candidate that is never considered worst in any scenario. The stakeholders, together with the analyst, decide on the selection criterion to be used.

When we use the third approach, the interpretation can be done in two ways: comprehensive or aggregated.

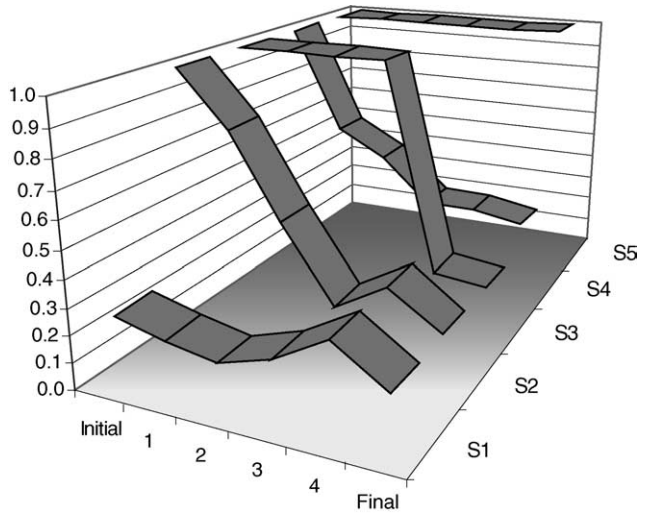

Fig. 7. Analysis results for all design and analysis iterations.

Using the first interpretation, the analyst considers the results of all scenarios to select a candidate. For instance, the analyst could, based on all the scenarios, decide that certain scenarios are the most important scenarios to see which candidate architecture is best. This type of interpretation relies the experience and analytical skills of the analyst. The other approach is to aggregate the results for all candidate architectures using some kind of algorithm or formula, such as the prediction model in Section 5.5. The predicted effort is then compared and the best candidate has the lowest predicted effort. This interpretation focuses on the whole rather than the differences. These two interpretation techniques can be used together to come to a better judgment, because they are based on the same results.

In the beer can inspection case we chose the more comprehensive approach to interpreting the results. By doing so we get more insights as to what problems to be addressed in the next iteration of the design. In Table 5 we see that the architecture was improved in terms of component modification ratio for the main part of the scenarios and remained on the same level for two of the change scenarios. Fig. 7 illustrates the improvements of the architecture during the iterations. This figure summarizes the results of all six analyses for each change scenario (S1 to S5) made during the design iterations. We can clearly see that for change scenario 1 (S1) and change scenario 2 (S2), the fourth iteration worsened the results. However, the same iteration greatly improved the results for change scenario 3 (S3).

7.6. Conclusions

In this section we have presented techniques to use ALMA to compare two or more candidate software architectures. The technique is based on finding extreme scenarios to stress the architecture and expose differences. Evaluation for this goal can be performed in slightly different ways: determine the best candidate for each scenario, rank the scenario, or express the effect of the scenario on an ordinal scale for both candidates. Interpretation of the results is dependent on the evaluation scheme used. It may determine what candidate is best for most scenarios, or summarize the ranks and compare the sum, or compare the quantitative results.

In the beer can inspection case we concluded from the assessment by comparing the ratio of required modifications that the design iterations had on total improved the architecture regarding modifiability, i.e the last version required less components to change. Although the path, as presented in Fig. 7, shows that some steps did not improve the outcome for all scenarios, the end result is indeed an improvement. Worth noting is that the fifth scenario did not improve at all during the iterations and had, when concerned with this comparison only, no contribution to the result.

8. Empirical foundation

Scientific merit and validation is important when offering new methods, including the method presented in this article. Controlled experiments are increasingly advocated in the software engineering community (Tichy, 1998; Wohlin et al., 2000). Although a controlled experiment can indeed be a powerful means to test theories, it is not the only method that righteously can claim scientific merit in testing theories. In Robson case studies are thoroughly discussed with respect to their strengths, weaknesses and validity as methods to test theories (Robson, 1993).

In our case, the key has been to reveal the underlying assumptions and test their validity empirically. This is a bottom-up approach to validating the method as a whole. In our research on this method for software architecture analysis, we have tested our assumptions in the following ways:

We have reported on a controlled laboratory experiment on the process of scenario elicitation (Bengtsson and Bosch, 2000). The main conclusion was that scenario profiles generally are better when prepared by individual members that combine their scenarios.

We have performed seven different case studies in a variety of domains. These case studies are in essence based on action research (Argyris et al., 1985) where the researchers participate in the process and still perform empirical observations. In each case, assumptions have been tested: explicit assumptions, e.g. assumptions about the practice on documentation; implicit assumptions have also been tested, i.e. invalid assumptions have become visible when applying the method in different domains. The following case studies are part of the empirical foundation of this research:

- Measurement systems design and assessment performed with EC-Gruppen AB, Sweden (Bengtsson and Bosch, 1998). The application is an industrial embedded control system.
- Haemo dialysis design and assessment performed with Althin Medical AB and EC-Gruppen, Sweden (Bengtsson and Bosch, 1999a,b). The application is a medical treatment device.
- Mobile Positioning Center assessment case study performed with Ericsson Software Technology AB, Sweden (Lassing et al., 2001). The application is a telecommunications service provider system.
- EASY assessment case study performed with DFDS Fraktarna/Cap Gemini Ernst & Young in Sweden (Lassing et al., 2001). The application is a groupage/freight tracking system.
- ComBAD Framework assessment case study performed with Cap Gemini Ernst & Young in the Netherlands (Lassing et al., 1999c). The system is a domain specific architecture for administrative systems.
- MISOC2000 assessment case study performed with the Dutch Department of Defense Telematics Agency, The Netherlands (Lassing et al., 1999a). The application is a course and student administration system.
- Sagitta 2000 assessment case study performed with the Dutch Tax and Customs Administration, The Netherlands (Lassing et al., 2001). The application is a tax declaration processing system.

Research evolves by exploratory studies that lead to theories that are empirically tested. The results from these tests lead to modifications of our theories and new testing of the hypotheses, and so on. At this point we have no firm quantitative data that allow us to determine the accuracy of the predictions, nor the coverage of risks. This remains as future work.

9. Summary

In this paper we propose ALMA, a method for software architecture analysis of modifiability based on scenarios. This method is a generalization of earlier work by the authors and it consists of five major steps: (1) set goal, (2) describe the software architecture, (3) elicit change scenarios, (4) evaluate change scenarios and (5) interpret the results.

ALMA distinguishes the following goals that can be pursued in software architecture analysis of modifiability: maintenance prediction, risk assessment and software architecture comparison. Maintenance prediction is concerned with predicting the effort that is required for adapting the system to changes that will occur in the system's life cycle. In risk assessment we aim to expose the changes for which the software architecture is inflexible. Software architecture comparison is directed at exposing the differences between two candidate architectures. We have demonstrated that the goal pursued in the analysis influences the combination of techniques that is to be used in the subsequent steps.

After the goal of the analysis is set, we draw up a description of the software architecture. This description will be used in the evaluation of the scenarios. It should contain sufficient detail to perform an impact analysis for each of the scenarios. To do so, the following information is essential: the decomposition in components, the relationships between the components, the (lower-level) design and implementation of the architecture and the relationships to the system's environment.

The next step is to elicit a representative set of change scenarios. To find this set, we require both a selection criterion, i.e. a criterion that states which scenarios are important, and a stopping criterion, i.e. a criterion that determines when we have found sufficient scenarios. The selection criterion is directly deduced from the goal of the analysis. To find scenarios that satisfy this selection criterion, we use scenario classification. We have distinguished two approaches to scenario elicitation: a top-down approach, in which we use a classification to guide the elicitation process, and a bottom-up approach, in which we use concrete scenarios to build up the classification structure. In both cases, the stopping criterion is inferred from the classification scheme.

The next step of an analysis is to evaluate the effect of the set of change scenarios. To do so, we perform impact analysis for each of the scenarios in the set. The way the results of this step are expressed is dependent on the goal of the analysis: for each goal different information is required. The final step is then to interpret these results and to draw conclusions about the software architecture. Obviously, this step is also influenced by the goal of the analysis: the goal of the analysis determines the type of conclusions that we want to draw.

We have elaborated the various steps in this paper, discussed the issues and techniques for each of the steps and illustrated these by elaborating three case studies of architecture analysis. In each of these case studies a different goal was pursued.

Acknowledgements

We are very grateful to Cap Gemini Netherlands, and especially Daan Rijsenbrij, for their financial support of this research. We would also like to thank the companies that made it possible to conduct our case studies, Ericsson Software Technology AB, Cap Gemini Sweden, DFDS Fraktarna AB and EC-Gruppen AB for their cooperation. 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 Sweden, Stefan Gunnarsson of DFDS Fraktarna, Anders Kambrin and Mogens Lundholm of EC-Gruppen AB and the student Abdifatah Ahmed, for their valuable time and input. Finally, we would like to thank the anonymous referees for their feedback.

References

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, Pittsburgh, PA.

Argyris, C., Putnam, R., Smith, D., 1985. Action Science: Concepts, Methods, and Skills for Research and Intervention. Jossey-Bass, San Fransisco, CA.

AT&T, 1993. Best current practices: software architecture validation. Internal Report, AT&T, US.

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

Bengtsson, P., Bosch, J., 1998. Scenario-based software architecture reengineering. In: Proceedings of the 5th International Conference on Software Reuse (ICSR5). IEEE CS Press, Los Alamitos, CA, pp. 308–317.

Bengtsson, P., Bosch, J., 1999a. Architecture level prediction of software maintenance. In: Proceedings of 3rd EuroMicro Conference on Maintenance and Reengineering (CSMR'99). IEEE CS Press, Los Alamitos, CA, pp. 139–147.

Bengtsson, P., Bosch, J., 1999b. Haemo dialysis software architecture design experiences. In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99). ACM Press, Los Angeles, CA, pp. 516–525.

Bengtsson, P., Bosch, J., 2000. An experiment on creating scenario profiles for software change. Annals of Software Engineering 9, 59–78.

Bohner, S.A., 1991. Software change impact analysis for design evolution. In: Proceedings of 8th International Conference on Maintenance and Re-engineering. IEEE CS Press, Los Alamitos, CA, pp. 292–301.

Bosch, J., Bengtsson, P., 2001. Assessing optimal software architecture maintainability. In: Proceedings of Fifth European Conference on Software Maintenance and Reengineering (CSMR'01). IEEE CS Press, Los Alamitos, CA, pp. 168–175.

Briand, L., Arisholm, E., Counsell, S., Houdek, F., Thévenod-Foss, P., 1999. Emprical studies of object-oriented artifacts, methods, and processes: state of the art and future directions. Empirical Software Engineering 4 (4), 387–404.

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, pp. 342–358.

Henry, J.E., Cain, J.P., 1997. A quantitative comparison of perfective and corrective software maintenance. Journal of Software Maintenance: Research and Practice 9, 281–297.

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

Hofmeister, C., Nord, R.L., Soni, D., 1999b. Describing software architecture with UML. In: Donohoe, P. (Ed.), Software architecture: Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1). Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 145–159.

IEEE, 2000. Recommended practice for architectural description. IEEE Standard P1471.

ISO/IEC, 2000. Information technology—software product quality—Part 1: Quality model. ISO/IEC FDIS 9126-1:2000(E).

Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., 1992. Object-Oriented Software Engineering. A Use Case Driven Approach. Addison Wesley Longman, Reading, MA.

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

Kazman, R., Barbacci, M., Klein, M., Carrière, S.J., 1999. Experience with performing architecture tradeoff analysis. In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99). ACM Press, Los Angeles, CA, pp. 54–63.

Kazman, R., Klein, M., Clements, P., 2000. ATAM: Method for architecture evaluation (CMU/SEI-2000-TR-004). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

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

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., Chen, C., 1994. Change impact in object oriented software maintenance. In: Proceedings of the International Conference on Software Maintenance (ICSM'94). IEEE CS Press, Los Alamitos, CA, pp. 202–211.

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

Lassing, N., Rijsenbrij, D., van Vliet, H., 1999b. The goal of software architecture analysis: confidence building or risk assessment. In: Proceedings of the 1st Benelux Conference on State-of-the-art of ICT architecture. Vrije Universiteit, Amsterdam, The Netherlands.

Lassing, N., Rijsenbrij, D., van Vliet, H., 1999c. Flexibility of the ComBAD architecture. In: Donohoe, P. (Ed.), Software Architecture: Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1). Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 341–355.

Lassing, N., Rijsenbrij, D., van Vliet, H., 2001. Viewpoints on modifiability. International Journal of Software Engineering and Knowledge Engineering 11 (4).

Li, W., Henry, S., 1993. OO metrics that predict maintainability. Journal of Systems and Software 23 (1), 111–122.

Lientz, B.P., Swanson, E.B., 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 (ICSM'98). IEEE CS Press, Los Alamitos, CA, 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.

Maxwell, K.D., Van Wassenhove, L., Dutta, S., 1996. Software development productivity of European space, military, and industrial applications. IEEE Transactions on Software Engineering 22 (10), 706–718.

Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12), 1053–1058.

Robson, C., 1993. Real World Research. Blackwell Publishers, Oxford, UK.

Stark, G.E., Oman, P.W., 1997. Software maintenance management strategies: observations from the field. Software Maintenance: Research and Practice 9 (6), 365–378.

Sutcliffe, A.G., Maiden, N.A.M., Minocha, S., Manuel, D., 1998. Supporting scenario-based requirements engineering. IEEE Transactions on Software Engineering 24 (12), 1072–1088.

Tichy, W.F., 1998. Should computer scientists experiment more? IEEE Computer (May), 32–40.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Boston, MA.