Experiences with SAA of Modifiability

Nico Lassing[*], PerOlof Bengtsson[**], Hans van Vliet[*] and Jan Bosch[**]

[*] *Faculty of Sciences*
Division of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

[**] *Department of Software Engineering and Computer Science*
University of Karlskrona/Ronneby
Ronneby, Sweden

Abstract

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

1. Introduction

Software architecture is generally regarded as an important instrument to increase the quality of software systems. One of the qualities to which this applies is modifiability. Modifiability is related to the effort required to modify a system to changes in the requirements, the functional specification or the environment. It is important that these changes are already anticipated during software architecture design. To make sure that this is the case, software architecture analysis is an important tool. However, few methods for analysis of modifiability currently exist.

We have defined a generalized, adaptable method for software architecture analysis of modifiability. This method is a generalization of earlier work by the authors independently [4, 9].

In this paper we illustrate our experiences in using this method in two case studies. One case concerns a system developed by Ericsson Software Technology for positioning mobile telephones, and the other case concerns a system for freight handling at DFDS Fraktarna. The observations made in this paper corroborate and strengthen our individual experiences in earlier studies [4, 9].

In section 2, we describe our method for software architecture analysis of modifiability. Section 3 introduces the two cases. Our experiences are described in section 4. In section 5, we conclude with some summarizing statements.

2. Method overview

The method for software architecture analysis of modifiability that we advocate is based on change scenarios. A change scenario is a description of a specific event that may occur in the life cycle of a system and requires the system to be modified. By exploring the effect of these change scenarios on the software architecture, we can make predictions of the effort that is required to implement the changes once the system is built. This enables us to judge the modifiability of the system that will be built with the software architecture under analysis.

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

1. Set goal: determine the aim of the analysis
2. Describe software architecture: give a description of the relevant parts of the software architecture

3. Elicit change scenarios: find the set of relevant change scenarios
4. Evaluate change scenarios: determine the effect of the set of change scenarios
5. Interpret the results: draw conclusions from the analysis results

Obviously, the separation between these tasks is not very strict when performing an analysis. It will often be necessary to iterate over various steps.

We will now give a brief overview of the steps. A more elaborate discussion of the full method is given in [5].

2.1 Goal setting

The first step to take in software architecture analysis of modifiability is to set the analysis goal. The goal determines the type of results that will be delivered by the analysis. Additionally, we select the techniques to be used in the next steps, based on this goal. We can pursue the following goals:

– Maintenance cost prediction: estimating the cost of maintenance tasks in a given period
– Risk assessment: finding types of changes for which the system is inflexible
– Software architecture selection: comparing two or more candidate software architectures to find the most appropriate one.

2.2 Software architecture description

After we have selected the goal of the analysis, the next step is to create a description of the software architecture. To do so, we use two sources of information. The architecture designs present within the development team are an important first step. For additional information, we may interview the architect(s) of the system.

This step serves two purposes. First and foremost, it should result in a description that is detailed enough to enable us to evaluate the effect of the set of change scenarios. In addition, it provides us with insight into the functionality of the system under analysis.

2.3 Scenario elicitation

One of the most important steps in modifiability analysis is the elicitation of a set of change scenarios. This set of change scenarios captures the events that stakeholders expect to occur in the future of the system. The main technique to elicit this set is to interview stakeholders, because they are in the best position to predict what may happen in the future of the system. In addition, they are able to judge the likelihood of the change scenarios that we have found. However, relying on stakeholders to come up with scenarios also poses a threat to the completeness of the analysis, because scenarios that are not foreseen by the stakeholders are not considered in the next steps of the analysis.

The aim of this step is to come to a set of change scenarios that supports the goal that we have set for the analysis. If the goal of the analysis is to do maintenance cost prediction, the set of scenarios should be representative for the changes that could occur in a certain period of time. On the other hand, if the goal of the analysis is risk assessment, we should focus on scenarios that are complicated to implement. Finally, if the goal of the analysis is software architecture selection, we are most interested in scenarios that highlight changes between the candidate architectures. So, the result of this step is dependent on the goal of the analysis.

2.4 Scenario evaluation

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

The way the results of this step are expressed depends on the goal of the analysis. If the goal of the analysis is maintenance prediction, the results should be expressed in such a way that we are able to calculate the effort that is required for implementing the change scenarios. Which information is required for doing this depends on the prediction model used. For example, in [4] the impact is calculated using the size of the affected components and the percentage of change, resulting in the number of lines of code affected. So, in that case the changes should be expressed using those measures.

For risk assessment, the results should be expressed in such a way that they provide insight into the complexity of the changes associated with each change scenario. This means that the effects should be expressed using measures that affect the complexity of changes. In [9] a number of factors is mentioned that influence this complexity in the area of business information systems: (1) impact on software architecture, (2) the involvement of multiple owners in a change scenario and (3) the introduction of different versions of the same component. For a more elaborate treatment of this model, we refer to [5, 9].

For architecture comparison, the results of the analysis should be expressed in such a way that they show the differences between the candidate architectures. We can do this in a number of ways:

– For each change scenario we can determine the candidate architecture that supports it best, or if there are no differences

Table 1: Comparison of system characteristics

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

– For each change scenario we can rank the candidate architectures depending on their support for the scenario
– For each change scenario we can determine the effect on all candidate architectures and express this effect using some scale

2.5 Interpretation

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

If the goal of the analysis is to do maintenance prediction, the aim of this step is to come to an estimate of the amount of effort that is required for maintenance activities in the coming period. In [4] a model for maintenance prediction is proposed. In this model, the maintenance effort per change scenario is calculated by taking the sum of the products of the effort required for each scenario and a weight indicating their relative importance.

If the goal of the analysis is risk assessment the results of the scenario evaluation are investigated to determine which change scenarios are a risk when the system has to be adapted. In this process, the likelihood of the scenarios should also be taken into account: change scenarios that are extremely complicated but also highly unlikely, may not be considered risks.

If the goal of the analysis is architecture comparison, we compare the results of the evaluation of the two scenarios and choose the most appropriate candidate architecture. Various strategies can be employed to decide upon that issue, depending on the evaluation technique used in the previous step. For instance, when we rank the candidate architectures we may select the candidate that supports the most scenarios. One the other hand, if we express the effect of each scenario on some scale, we may choose the candidate that does not rank lowest for any of the scenarios. Again, it is important that the likelihood of the scenarios is considered in this selection process.

3. Case descriptions

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

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

3.1 Ericsson Mobile Positioning Center (MPC)

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

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

The MPC GUI is used for system administration, such as controlling what users should be able to position what mobile phones and configuring alarm handling. The MPC system also generates billing information to allow the network operators to charge for the services they provide.

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

3.2 EASY

EASY is a system that is currently being developed by Cap Gemini Sweden for DFDS Fraktarna, a Swedish distributor of freight. EASY was developed to monitor the location of groupage (freight with a weight between 31 kg and 1 ton) through the company's distribution system. To do so, the unique barcode of groupage is scanned after each step in the distribution system using a mobile barcode scanner. This information is then stored.

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

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

4. Experiences

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

4.1 Goal setting

4.1.1 Modifiability analysis requires a clear goal

The goal of the analysis determines what techniques to be used in the following analysis steps. For instance, when the goal is to make a risk assessment the elicitation technique to prefer is the guided interview, and the use of predefined scenario categories that are especially complex. When we discussed the planning for these case studies in our first meeting with the companies we in fact already discussed the goal for the analysis of their respective software architectures. We, the analysts, at that time did not fully understand the impact of the analysis goal to the following steps. In retrospect, we were right in making a choice for a specific goal at the very outset of the analysis.

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

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

4.2 Architecture description

4.2.1 Views

Several authors have proposed view models, consisting of a number of architectural views [8, 11]. Each view focuses on a particular aspect of the software architecture, e.g. its static structure or its dynamic aspect. We found that in software architecture analysis of modifiability a number of these views is required, but not all of them. The goal of the architecture description is to provide input for the following steps of the analysis or, more specifically, for determining the changes required for the change scenarios. We found that the views most useful for doing so are the view that shows the architectural approach taken for the system, i.e. the conceptual view, and view that shows the way the system is structured in the development environment, i.e. the development view. However, to explore the full effect of a scenario it is not sufficient to look at just one of these.

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

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

4.2.2 The system's environment

In earlier research [9] we found that in some cases it is important that the environment of the system is also modeled. We found this to be most useful in risk assessment. The reason for this is that changes that include the environment are considered more complex than changes that are limited to the system itself. For maintenance prediction the environment is less important since the scope of the prediction is generally limited to the system. In that case we focus on the effort that is required to adapt the system itself. Changes to its environment are not included in the prediction.

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

4.2.3 Need for additional information

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

For risk assessment of business information systems we need another type of additional information. In that case, we need to know about the system owners that are involved in a change [9]. This information is normally not included in the software architecture designs, so it has to be obtained separately from the stakeholders.

4.3 Scenario elicitation

4.3.1 Different perspectives

Scenario elicitation is usually done by interviewing different stakeholders of the system under analysis. We found that it is important to have a mix of people from the 'customer side' and from the 'development side'. Different stakeholders have different goals, different knowledge, different insights, and different biases. This all adds to the diversity of the set of scenarios. In testing, different test techniques reveal different types of errors [7]. In vacuuming, one is likely to pick up more dirt if the rug is vacuumed in two directions. Similarly, in architecture analysis, it helps to collect scenarios from a variety of sources.

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

4.3.2 The architect's bias

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

A possible reason for this could be that the architect of the system is, implicitly, trying to convince himself and his environment that he has taken all the right decisions. After all, it is his job to devise a flexible architecture. It requires a special kind of kink to destroy what you yourself have just created. This is the same problem programmers have when testing their own code. It requires skill and perseverance of the analyst to take this mental hurdle and elicit relevant scenarios from this stakeholder.

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

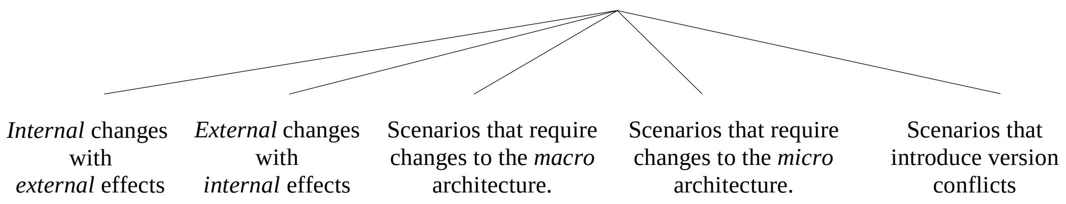
Internal changes with *external* effects
External changes with *internal* effects
Scenarios that require changes to the *macro* architecture.
Scenarios that require changes to the *micro* architecture.
Scenarios that introduce version conflicts

Figure 1: Deriving change scnearios from knowledge of complexity of changes in business information systems [9]

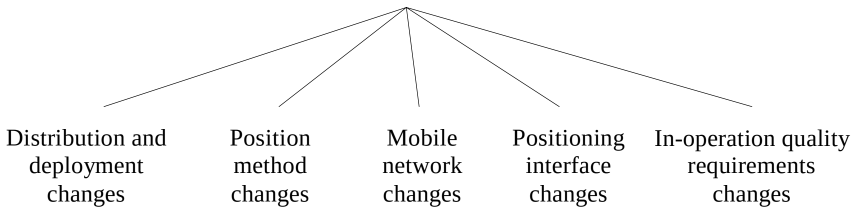
Distribution and deployment changes
Position method changes
Mobile network changes
Positioning interface changes
In-operation quality requirements changes

Figure 2: Deriving change scenarios from problem domain

4.3.3 Structured scenario elicitation

When interviewing stakeholders, recurring questions are:

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

Unguided scenario elicitation relies on the experience of the analyst and stakeholders in architecture assessment. The elicitation process then stops if there is mutual confidence in the quality and completeness of the set of scenarios. This may be termed the *empirical* approach [6]. One of the downsides of this approach that we have experienced is that the stakeholders' horizon of future changes is very short. Most change scenarios suggested by the stakeholders relate to issues very close in time, e.g. anticipated changes in the current release.

To address this issue we have found it very helpful to have some organizing principle while eliciting scenarios. This organizing principle takes the form of a, possibly hierarchical, classification of scenarios to draw from. For instance, in our risk assessment of EASY, we used a categorization of high-risk changes as given in Figure 1. This categorization has been developed over time, while gaining experience with this type of assessment. When the focus of the assessment is to estimate maintenance cost, the classification tends to follow the logical structure of the domain of the system, as in Figure 2. If this hierarchy is not known a priori, an iterative scheme of gathering scenarios, structuring those scenarios, gathering more scenarios, etc., can be followed. In either case, this approach might be called *analytical*.

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

4.3.4 Weight confusion

The weighting scheme used in maintenance prediction appears confusing to the stakeholders that are asked to provide the weights. The weight for each scenario is supposed to be the relative frequency of occurrence for the equivalent class it represents [5], i.e. given the period to be predicted the weight is the normalized number of times the scenario is believed to occur.

The problem for the stakeholders is that they often know that a certain change *will* occur. Intuitively, they feel that the weight for that change scenario should be one. However, this is contradicted when there is yet another scenario that *will also* occur, because they understand that two scenarios cannot have the normalized weight of one.

The key to understanding the weighting is to think of it as two steps, estimation and normalization (see Tables 2 and 3). The prediction is aimed for a certain period of time, e.g. between two releases of the system. In case there is a scenario, scenario 1, that we know will happen, we first estimate how many times it will occur. Another change scenario, scenario 2, we expect to occur, let us say, three times during the predicted period. Then we perform step two, normalization. The weights are calculated by dividing the estimate number of times a change scenario will occur by the sum of the estimates of the whole set of scenarios.

Table 2: Weighting step one: Estimate

Scenario 1	1
Scenario 2	3

Table 3: Weighting step two: Normalize

Scenario 1	1 / (1+3) = ¼
Scenario 2	3 / (1+3) = ¾

4.4 Scenario evaluation

4.4.1 Ripple effects

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

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

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

The main factor influencing this problem is the amount of detail present in the description of the software architecture. This is determined by the amount of detail available to the software architect, i.e. his or her understanding of the problem. In addition, a study [9] have shown that even detailed design descriptions lack information necessary for performing an accurate analysis of ripple effects. Since less detail is available at the software architecture level, this is an even more relevant problem.

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

4.5 Interpretation of the results

4.5.1 Lack of frame of reference

Once scenario evaluation has been performed, we have to associate conclusions with these results. The process of formulating these conclusions we refer to as *interpretation*. For instance, the result of scenario evaluation may be that, on the average, for each change scenario, 2.3 components are affected to an extent of 25%. The process of interpretation is concerned with deciding whether this is acceptable or not.

The first experience that we have from the interpretation is that we lack a frame of reference for interpreting the results of scenario evaluation. Is the number mentioned above 'good' or 'bad'? For instance, does an architecture exist with considerably better associated numbers with respect to maintainability? For instance, it may be the case that an architecture could be designed that supports an average of 1 changed component to an extent of 15% per change scenario. The result of scenario evaluation is relative, but we have no means or techniques to relate it to other numbers or results.

For instance, in the analysis of the MPC we came to the conclusion that there would be 270 lines of code affected per change scenario. However, we did not know whether this number was 'good', or that it is possible to come to a software architecture that is significantly better.

4.5.2 Role of the owner in interpretation

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

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

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

4.6 General experiences

4.6.1 Software architecture analysis is an ad hoc activity

Three arguments are used for defining a software architecture [3]. First, it provides an artifact that allows for discussion by the stakeholders very early in the design process. Second, it allows for early assessment or analysis of quality attributes. Finally, it supports the communication between software architects and software engineers since it captures the earliest and most important design decisions. In our experience, the second of these is least applied in practice. The software architecture is seen as a valuable intermediate product in the development process, but its potential w.r.t. quality assessment is not fully exploited.

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

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

4.6.2 Accuracy of analysis is unclear

A second general experience is that we lack means to decide upon the accuracy of the results of our analysis. We are not sure whether the numbers that maintenance prediction produces are accurate, and whether risk assessment gives an overview of all risks. On the other hand, it is doubtful whether this kind of accuracy is at all achievable. The choice for a particular software architecture influences the resulting system and its outward appearance. This in turn will affect the type of change requests put forward. Next, the configuration control board which assesses the change requests, will partly base its decisions on characteristics of the architecture. Had a different architecture been chosen, then the set of change requests proposed and the set of change requests approved would likely be different. Architectural choices have a feedback impact on change behavior, much like cost estimates have an impact on project behavior [1].

A further limitation of this type of architecture assessment is that it focuses on aspects of the product only, and neglects the influence of the process. For instance, in [2] it was found that quite a large number of problems uncovered in architecture reviews had to do with the process, such as not clearly identified stakeholders, unrealistic deployment date, no quality assurance organization in place, and so on.

5. Conclusions

Software architecture is generally regarded as an important instrument to increase the quality of software systems. One of the qualities to which this applies is modifiability. We have defined a generalized five-step method for modifiability analysis of software architecture [5]. In this paper we report on 13 experiences we acquired developing and using this method. These experiences are illustrated using two case studies that we performed: the MPC system developed by Ericsson Software Technology for positioning mobile telephones and the EASY system for freight handling at DFDS Fraktarna.

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

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

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

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

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

6. Acknowledgements

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

7. References

[1] T.K. Abdel-Hamida and S.E. Madnick, Impact of Schedule Estimation on Software Project Behavior, *IEEE Software*, 3(4): 70-75, 1986.

[2] A. Avritzer and E.J. Weyuker, Investigating Metrics for Architectural Assessment, *Proceedings of the Fifth International Software Metrics Symposium*, Bethesda, Maryland, 1998, pp 4-10.

[3] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Reading, MA: Addison Wesley Longman, 1998.

[4] P. Bengtsson and J. Bosch, 'Architecture Level Prediction of Software Maintenance', In *Proceedings of 3rd EuroMicro Conference on Maintenance and Reengineering(CSMR'99)*, Los Alamitos, CA: IEEE CS Press, 1999, pp. 139-147.

[5] P. Bengtsson, N. Lassing, J. Bosch and H. van Vliet, *Analyzing Software Architectures for Modifiability* (HK-R-RES—00/11-SE), Submitted for publication, 2000.

[6] J.M. Carroll and M.B. Rosson. Getting around the Task-Artifact cycle. *ACM Transactions on Information System*s, 10(2):181–212, April 1992.

[7] E. Kamsties and C.M. Lott, An Emperical Evaluation of Three Defect-Detection Techniques, In: W. Schäfer and P. Botella (eds.). *Software Engineering -- ESEC '95*, Springer, 1995, pp. 362-383.

[8] P.B. Kruchten, 'The 4+1 View Model of Architecture', *IEEE Software*, 12(6):42–50, November 1995.

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

[10] M. Lindvall and M. Runesson, 'The Visibility of Maintenance in Object Models: An Empirical Study', In *Proceedings of International Conference on Software Maintenance*, Los Alamitos, CS: IEEE CS Press, pp. 54-62, 1998.

[11] D. Soni, R. L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proceedings of the 17th International Conference on Software Engineerin*g, Seattle, WA, 1995.