Research Report 11/00

Analyzing Software Architectures for Modifiability

by

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

Department of
Software Engineering and Computer Science
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HK/R-RES—00/11—SE

Analyzing Software Architectures for Modifiability
by PerOlof Bengtsson, Nico Lassing, Jan Bosch, Hans van Vliet

ISSN 1103-1581
ISRN HK/R-RES—00/11—SE

Printed by Psilander Grafiska, Karlskrona 2000

Analyzing Software Architectures for Modifiability

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

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

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

Abstract

Several studies have shown that 50% to 70% of the total lifecycle cost for a software system is spent on evolving the system. Since incorporating anticipated changes generally requires considerably less effort than unanticipated changes, it is important to prepare a software system for likely changes during development. The software architecture plays an important role in achieving this, but few methods for modifiability analysis exist. In this paper, we propose an analysis method for software architecture modifiability that has successfully been applied in several cases. The method consists of five main steps, i.e. goal selection, software architecture description, scenario elicitation, scenario evaluation and interpretation.

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 [9, 22] have shown that 50% to 70% of the total lifecycle cost of a software system is spent after initial development. One of the reasons for the high cost of software maintenance is that productivity of changing existing code is at least an order of magnitude lower than developing new software or relatively independent extensions to an existing software system. Because of this, incorporating anticipated changes, changes for which the software system has been prepared, requires generally considerably less effort than including unanticipated changes.

Preparing a software system for likely changes is an activity that takes place during the development of a software system. In particular, the software architecture of the system plays an important role in achieving this. Based on an understanding of the expected future evolution of the software system, the software architect can employ various design solutions to prepare for the future incorporation of new and changed requirements.

One problem, however, is that the software architect has few means or techniques available for determining whether the goal of high modifiability has been achieved by the set of employed design solutions, either in terms of predicted maintenance cost or with respect to avoiding inflexibility in the current design. Similarly, few techniques are available for selecting among a number of alternative architectures to be used for a system.

One area of research addressing the above is *software architecture analysis*. In software architecture analysis, the software architecture of a software system is analyzed to predict one or more quality attributes. At present, a number of methods for software architecture analysis exist. Examples include

SAAM [14], architecture level prediction of maintenance [5] and inflexibility assessment [18]. Although these methods do share a number of similarities, there are fundamental differences as well. In earlier work [20], we have found that much of the differences can be explained based on the difference in goal pursued by each method.

In response to the aforementioned issues, we present a generalized, adaptable method for software architecture analysis of modifiability in this paper. The method consists of five main steps, i.e. goal selection, software architecture description, scenario elicitation, scenario evaluation and interpretation. We have 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 is adapted by using different techniques in some of the main steps.

The contribution of this paper is that it presents a method for analyzing software architecture modifiability that has been validated through its application in several industrial cases. This method is a generalization of earlier work by the authors performed independently [5, 18] and represents, as such, an extensive body of experience in the domain of software architecture analysis for modifiability.

The method has been applied successfully to a number of industrial cases, including software architectures at Ericsson Software Technology, DFDS Fraktarna, Althin Medical and the Dutch Department of Defense. The domains of these software architectures differ considerably, i.e. telecommunications, logistics, embedded systems and administrative information systems. The software architects at the organizations were positive to the analysis method and judged the results of the analysis as valuable.

The remainder of the paper is organized as follows. In the next section, we discuss the relations between software architecture and modifiability. Section 3 discusses various approaches to software architecture analysis. Section 4 presents an overview of our method and the main steps of the method. In the four subsequent sections, we discuss the main steps of the method in more detail. In section 9, we present, for each assessment goal, an example of performing modifiability analysis in practice. Finally, we conclude with a summary of the paper in section 10.

2. Modifiability in software architecture

As we mentioned in the introduction, this paper is concerned with software architecture analysis of modifiability. However, before we proceed to present our method, we need to introduce our perspective on a number of concepts, such as *modifiability* and *software architecture*. In the next section, we discuss modifiability, whereas software architecture is discussed in section 2.2.

2.1 Modifiability

Maintenance cost generally presents the major cost factor during the lifecycle of a software system. Consequently, when developing a software system, stakeholders are very interested that the system is designed in such a way that future changes will be relatively easy to implement, since this decreases maintenance cost. 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. the lowest cost to accommodate future 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?
- Is there any other construction that is significantly easier to modify than the present one?

Before we present the definition of modifiability that we will use throughout the paper, we investigate related definitions in the literature. A large number of definitions of qualities exist that are related to the ability of a software system to be modified. A very early classification of qualities [25] includes the following definitions:

> *Maintainability is the effort required to locate and fix an error in an operational program.*
>
> *Flexibility is the effort required to modify an operational program.*

A more recent classification of qualities is given in the ISO 9126 standard [12]. 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.*

Although different in wording, the definitions are almost identical in their semantics. The limitation of these definitions with respect to our purpose is that their scope is too broad. They capture 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. Since the focus of this paper is on software architecture analysis, only the second, third and fourth categories are relevant. Consequently, we focus on these categories and ignore the first. This results in 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 also concerned with the correction of bugs whereas modifiability is not.

2.2 Software architecture

The design and use of an explicit software architecture has received increasing amounts of attention during the last decade. Typically, three arguments for defining a software architecture are used. 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. The work presented in this paper is concerned with the second reason, i.e. early analysis of quality attributes, in particular modifiability.

A commonly used definition of software architecture is the one given in Bass *et al.* [3]:

> *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 structural aspects of the system. The definition given by the IEEE emphasizes other aspects of software architecture [11]:

The software architecture is the highest-level conception of the system in its environment.

This definition stresses that a system's software architecture is an abstraction of the system. In addition, it indicates that the environment should be addressed in the software architecture.

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 have a considerable influence on various qualities of the resulting system, including its performance, reliability and modifiability. However, these qualities are not orthogonal, i.e. they interact; a decision that has a positive effect on one quality might be very negative for another quality. Therefore, tradeoffs between qualities are inevitable and need to be handled explicitly during architectural design. This is especially important because architectural design decisions are generally very hard to change at a later stage. So, we should analyze the effect of these design decisions before it becomes prohibitively expensive to correct them. That is the rationale for performing explicit software architecture analysis.

3. Software architecture analysis

Most engineering disciplines provide techniques and methods that allow one to predict quality attributes of the system under design before it has been built or, in the case of, for instance, reliability and maintainability, before it has been in operation for a period of time. The availability of such techniques is of high importance since, without these, engineers may construct systems that fail to fulfill their quality requirements.

Prediction techniques and methods can be employed 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 [21]. The drawback of this approach is that it can only be applied when the code is present, which is only after a large effort has been put in the development of the system. At that time it often is prohibitively expensive to correct earlier design decisions. Other authors investigate design-level metrics, such as metrics for object-oriented design [8]. Again, such an approach is only valid in the stages after the software architecture is completed.

Since the first steps to achieving qualities are put in the software architecture, analysis of the software architecture is important to realize the required quality. This requires techniques and methods that allow one to predict a system's quality based on the software architecture. This is the domain of software architecture analysis, illustrated in Figure 1. This figure shows that this type of analysis aims at stating predictions about the qualities of a system that has not been implemented yet (indicated by the dotted lines). To do so, we study relevant properties of the artifact that is present, namely the software architecture (indicated by the solid lines).

We can distinguish various perspectives on the role of software architecture analysis within the development process: the analysis can be carried out by an external assessment team or, as an integrated part of the software architecture design process, by the software architect. We refer to the first case as external analysis and to the latter as internal analysis. An external analysis can be 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. The selected perspective does not influence the approach taken for the analysis.

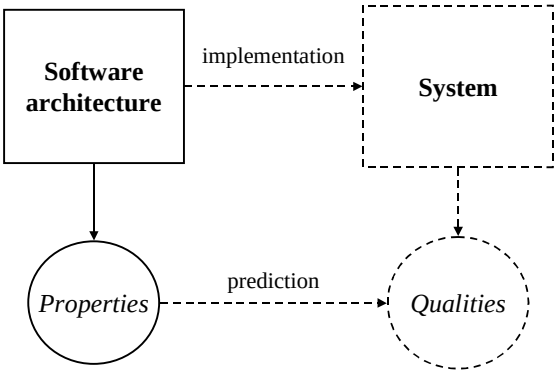

Figure 1: Software architecture analysis

In this section we demonstrate 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.

3.1 Using scenarios for software architecture analysis

A number of techniques can be used for software architecture analysis. In a recent overview of these techniques [1] a distinction is made between questioning techniques, which are qualitative in nature, and measurement techniques, which provide quantitative data. To the first group belong scenarios, questionnaires and checklists and the latter group consists of metrics and simulations.

In this paper we focus on software architecture analysis of modifiability. A number of the aforementioned techniques is useful for analyzing this quality. At AT&T [2], for instance, checklists are employed for software architecture analysis addressing issues related to different qualities, among others modifiability. These lists are used to check whether the development team followed 'good' software engineering practices, i.e. practices that are assumed to increase the quality of the system. However, this does not guarantee that the system will be modifiable with respect to changes that are likely to occur in the life cycle of the system. This issue is addressed in scenario-based software architecture analysis methods, which investigate the effect of concrete changes to assess a system's modifiability.

The basic principle of scenario-based analysis of modifiability is to define possible change scenarios, i.e. sequences of events, and assess their impact on the system. In software architecture analysis of modifiability, we are not concerned with changes that occurred in the past. Instead, we are interested in potential future changes. Such a description of a possible future change we refer to as a change scenario. Our use of the term scenario is different from conventional approaches where it is used solely to refer to usage scenarios. Examples of these approaches include object-oriented design [13] with its use cases (scenarios) and scenario-based requirements engineering [27].

The main reason for using change scenarios in software architecture analysis is that they are usually very concrete, enabling us to make detailed statements about their impact. At the same time, this could pose a threat to the value of our analysis. If our set of change scenarios is not carefully selected, the value of our analysis could be limited to the scenarios considered. We use two techniques to maximize the chances of obtaining a limited set of scenarios that is representative for the group of scenarios that we are interested in. First, we partition the space of scenarios into equivalence classes. Second, we impose some classification structure on the scenario space and use this classification to search for scenarios.

Changes that have the same impact from the point of view we are interested in are said to belong to the same equivalence class. We may express this impact in different ways: the amount of effort needed, an account of the lines of code that have been affected, an account of the components that have been

adapted, etc. Different changes may have the same impact. This is likely to be true at the source code level, but it is certainly true at the architectural level, where we have abstracted many details. Rather than considering all possible changes, we may now confine ourselves to considering one candidate from each equivalence class. For example, if screen I/O is handled by one component of the system whose modifiability we are interested in, we may consider one change scenario for the system which involves a different type of screen I/O, and assume that this change is as good as any other change of the same type. We may thus view a change scenario as an archetype for the equivalence class it belongs to.

Since we have not the built the system yet, we cannot be sure that different changes have the same effect, i.e. belong to the same equivalence class. Our decomposition into equivalence classes is an approximation, at best. For example, it may be that certain changes in screen I/O have unforeseen ripple effects in other components of the system. Our assumption that one such change is as good as any other does not hold in this case: our equivalence class partitioning is imperfect.

Our second technique to increase the quality of the scenario set involves a categorization of the equivalence classes according to the goal of our analysis. For instance, if the goal is to estimate maintenance cost, we distinguish a number of categories of likely changes. If, on the other hand, our goal is to do a risk assessment, we will distinguish categories of high-risk changes. In either case, the scenario elicitation process is focused on finding representative scenarios for each identified category.

3.2 Existing scenario-based methods for software architecture analysis

A number of scenario-based methods for software architecture analysis currently exist. One of the first methods for software architecture analysis that used scenarios was SAAM (Software Architecture Analysis Method) [14]. SAAM consists of four major steps: (1) develop scenarios, (2) describe candidate architecture(s), (3) evaluate scenarios and (4) overall evaluation. Step 1 is aimed to collect those changes that may affect the system. In step 2, the software architecture of the system or systems under analysis is described. SAAM does not enforce a specific technique for doing so; it merely states that all parties involved in the analysis should understand the technique. Step 3 is the actual analysis of the effect of the scenarios. To do this, the components that are affected by each scenario are identified, as well as their impact. The results of this are then used to express the impact of a scenario. In addition, these results are used to determine whether components are affected by several, unrelated, scenarios, which might indicate that they are too large. The results of step 3 are used in step 4 to reach an overall evaluation.

SAAM was succeeded by the Architecture Tradeoff Analysis Method, or ATAM [15]. ATAM also uses scenarios, but considers a number of qualities concurrently. In fact, it can be regarded as a framework for analysis, in which various qualities are analyzed and tradeoff points between them are identified. The principle of ATAM is to analyze various qualities in isolation first and then find those architectural elements that influence multiple qualities and vary them until the ‘right’ solution is found. Because ATAM is a framework for software architecture analysis, it does not prescribe methods for analyzing the individual qualities. This means that the method presented in this paper could very well be used within an ATAM analysis to assess modifiability.

Bengtsson and Bosch [5] defined a scenario-based method for software architecture analysis that is aimed at estimating the effort required for maintaining a system. This is done by defining a set of scenarios that is representative for the changes that are expected to occur during the life cycle of the system. For each of the scenarios the implementation effort is then determined and, together with some weight, this leads to an estimation of the total maintenance effort.

The method proposed by Lassing *et al.* [19] takes a somewhat different approach. This method aims to assess risks with respect to modifiability in the system, i.e. at finding those scenarios that are very hard, or even impossible to implement. To this end, the scenario elicitation process is not so much focused on finding likely scenarios, but on exposing scenarios that have complex associated changes.

These methods were defined based on different assumptions and are intended to be used in different situations. We explored these assumptions [20] and based on this knowledge, we were able to define a generalized, adaptable method for software architecture analysis of modifiability.

4. Software architecture analysis of modifiability in five steps

The method for software architecture analysis that we advocate 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

Depending on the situation, we select a combination of techniques to be used in the various steps. The combination of techniques cannot be chosen at random, certain relationships between techniques exist. These relationships are shown in Figure 2. The available techniques are discussed in the subsequent sections.

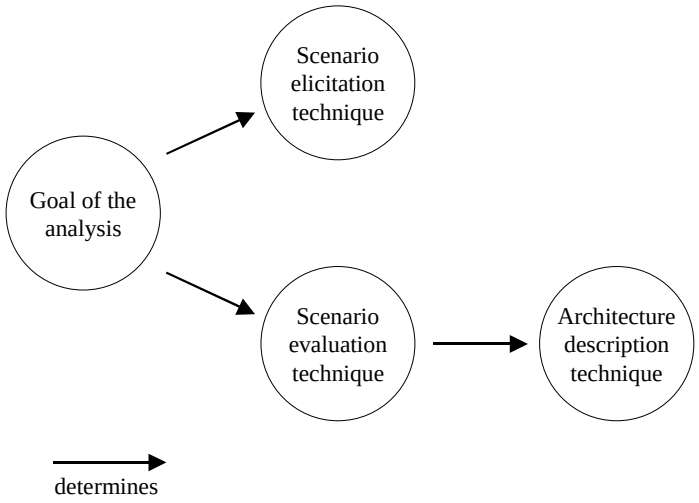

Figure 2: Relating techniques

The first activity of the method is concerned with determining the goal of the analysis. In software architecture analysis of modifiability 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

Based on the goal of the analysis, we select a technique for scenario elicitation. As mentioned, different situations require different elicitation techniques. For instance, when we want to estimate the maintenance costs for a system, we are interested in a different set of scenarios than when we perform risk assessment. In the first case, we require a set of scenarios that is representative for the actual events that will occur in the life cycle of the system and in the latter case we are interested to find scenarios that have complex associated changes.

Something similar goes for the scenario evaluation technique. Based on the analysis goal, we select a technique for evaluating and expressing the effect of the scenario. For example, if the goal of the analysis is to compare two or more candidate software architecture, we need to express the impact of the scenario using some kind of ordinal scale. This scale should indicate a ranking between the different candidates, i.e. which software architecture supports the required changes best.

The evaluation technique then determines what techniques 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.

When performing an analysis, the separation between the tasks is not very strict. It is often necessary to iterate over various steps. For instance, when performing scenario evaluation, a more detailed description of the software architecture may be required or, when performing scenario evaluation, new scenarios may come up. Nevertheless, in this paper, we will present the steps as if they are performed in strict sequence. Section 5 discusses the issues related to software architecture description. Section 6 treats the scenario elicitation process. Section 7 presents the various techniques for determining and expressing the effect of scenarios and section 8 discusses the way in which the results should be presented.

5. Description of the software architecture

Software architecture analysis requires information about the architecture. During architecture design, the software architecture evolves: it is extended gradually with architectural decisions made over time. As a consequence, the detail and amount of the information available about the software architecture is dependent on the point in time at which the analysis is performed. In software architecture analysis we use the information that is available. So, the later the analysis is performed the more detailed the information that is available to the analyst.

For the information that is available, we have to decide which aspects of the software architecture are required for the analysis. Different aspects of the architecture are best described using different techniques. In this section we will first discuss the current view on software architecture description. Subsequently, we will discuss information needed to perform modifiability analysis.

5.1 Software architecture description

One of the current foci in software architecture description research is to address the different perspectives one could have of the architecture, and describe them as views. The rationale of the resulting view models is that the information relevant for one view is different from that of others and should be described separately. Hence, different views contain different kinds of information and should be described using the most appropriate technique for each view. Several models have been proposed that include a number of views that should be described in the software architecture [10, 16]. For instance, the 4+1 View Model [16] consists of the following views:

- The *logical view*, which is the object model of the design.
- The *process view*, which captures the concurrency and synchronization aspects of the system.
- The *physical view*, which describes the mapping(s) of the software onto hardware and reflects its distributed aspect.
- The *development view*, which describes the static organization of the software in its development environment.
- The *scenario view*, or the +1-view, which illustrates the way the architecture satisfies the requirements and defines the relations among the other views.

A similar set of views was distinguished in [10]:

- The *conceptual architecture view*, which describes the system in terms of its major design elements and the relationships among them.
- The *module architecture view*, which is the decomposition of the system and the partitioning of modules into layers.
- The *execution architecture view*, which describes how modules are mapped to the elements provided by the runtime platform, and how these are mapped to the hardware architecture.
- The *code architecture view*, which describes the mapping of runtime entities in the execution view to deployment components, the mapping of the modules from the module architecture to source components and how the deployment components are produced from source components.

These view models have in common that they address the static structure, the dynamic aspect, the physical layout and the development of the system. Bass *et al.* [3] introduce the concept of architecture structures as a synonym of views. In addition to four structures that are identical to the views in the 4+1 View model, the authors mention the uses structure, the calls structure, the data flow, the control flow and the class structure.

Lassing *et al.* [18] have argued that a distinction should be made between internals of the system and its environment when the system is part of a larger whole. They call the internals of the system the *micro* architecture and the system in its environment the *macro* architecture. In their terminology, the views in both [10] and [16] only concern the micro architecture of a system.

In general, it is the responsibility of the architect to decide on the views to be used for describing the architecture. To this end, the architect considers the driving forces for the software architecture and the purpose of this description, e.g. the analyses that are planned, and selects the views accordingly. Once the architect has determined the views that are required, appropriate description techniques and notations need to be selected. For each of the aforementioned views a wide range of description techniques can be found. On the one end, there are informal techniques using boxes and lines with little or no explicit semantics. On the other extreme, we have more detailed and formal techniques, such as architecture description languages (or ADLs), with strict explicit semantics. Presently, the trend is to use notation techniques from UML, which could be viewed as located in the middle of the scale.

5.2 Information required for modifiability analysis

Modifiability analysis requires architectural information that allows the scenarios to be evaluated. In scenario evaluation we analyze the impact of the scenario and, depending on the goal of the modifiability analysis, make a number of estimates. Impact analysis is concerned with identifying the architectural elements affected by a change. This includes, obviously, the components that are affected directly, but also the indirect effects of changes on other parts of the architecture. The following pieces of information are essential for the impact analysis (see Figure 3):

- The decomposition in components
- The relationships between the components
- The (lower-level) design and implementation of the architecture
- 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 functions of the systems are assigned to different components. Initially, this assignment is often performed implicitly but during the rest of the development a component's provided interface will be fully specified.

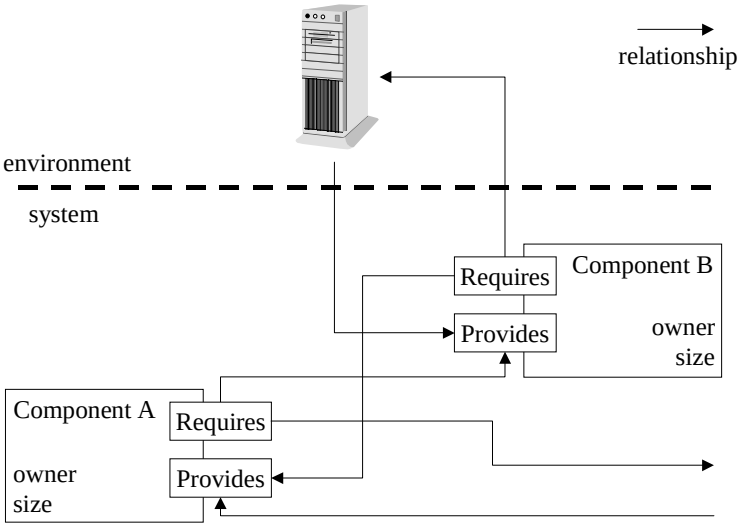

Figure 3: Required information about architecture

The meaning of a relationship between components may vary and is often defined implicitly. The relationships may imply functional dependencies, i.e. a component uses and depends on the provided functions of the other component. The relationships may also indicate synchronization, data flow or some other kind of dependency. Dependencies to other components are very valuable in impact analysis. Hence, the required interfaces for the components are among the first to be considered. In general, more information and more explicitly defined semantics of the relations facilitate more accurate results of the modifiability analysis. This is because it is the main source for tracing dependencies.

Not all dependencies between components are already visible in the software architecture. Some of them are introduced during lower level design and implementation. These relationships will also affect the ripple effects caused by the modifications made to a component. So, if this information is not available in the scenario evaluation, we have to make assumptions about these relationships. This means that the impact of the scenario may not be determined accurately and this affects the overall analysis accuracy. We can address this issue by using design patterns. The use of design patterns allows the architect to decide on more detailed design issues without going into the lower level details. Consider for example a scenario where we want to add a media type to a system. The estimated ripple effects would be overestimated if we did not know that the component creating the media objects uses the abstract factory pattern to create the media objects. This information does not necessarily show up in the architecture description.

The relationships of the system to its environment allow the analyst to determine dependencies towards the context of the system. Some scenarios require that the environment is adapted. However, the system's owner often has no or limited control over the system's environment, because the systems in its environment are owned by other parties. Ultimately, this may mean that scenarios for which modifications to the environment are required prove to be impossible, because the other system owners do not want to implement the required changes. Another consequence of the limited control is that uncontrollable changes in the environment may require the system to adapt. So, the environment is a complicating factor in the implementation of changes, as well as a source of changes.

The information required for impact analysis is not the only information required in modifiability analysis. Depending on the goal of the analysis, we may also need the estimated size of the architecture components and the ownership of the architecture components. The size estimate of the components is required when the goal of the analysis is to make effort predictions based on the modification volume. The size serves as a basis for estimates of the effort required for the modifications. Risks assessment

requires information about component ownership. Components might be owned by other parties and used in other systems and contexts as well. The impact analysis then needs to extend beyond the current system and future changes to a component have to be synchronized with other system owners before they can be incorporated.

6. Techniques for eliciting change scenarios

Change scenario elicitation is the process of finding and selecting the change scenarios that are to be used in the evaluation step of the analysis. In this section we present a general introduction to the change scenario elicitation process and give a more detailed discussion of the different techniques we employ.

6.1 Introduction to change scenario elicitation

Eliciting change scenarios involves such activities as identifying stakeholders to interview, properly documenting the scenarios that result from those interviews, etc. Many of these activities are not specific to software architecture analysis. A number of issues, however, are of specific concern in this case. 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. As discussed in section 3.1, partitioning the space of scenarios into equivalence classes enables us to treat one scenario as a representative of a whole 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. In addition to the selection criterion, we also need a stopping criterion: we must be able to decide when we have collected a representative set of scenarios.

In general, 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 of change categories may derive from the domain of interest, as in Figure 4, knowledge of potentially complex scenarios, as in Figure 5, 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 sufficiently complete set of scenarios. We then tacitly assume that these stakeholders have some implicit categorization scheme at the back of their head. This approach is bottom-up, because we start with concrete scenarios and then move to more abstract classes of scenarios.

In both cases, the stopping criterion derives from the change scenario classification scheme, be it explicit or implicit: we continue eliciting scenarios until a sufficiently complete coverage of the classification scheme has been obtained.

In practice, we often iterate between the approaches, i.e. the change scenarios that result from interviews are used to build up or refine our classification scheme. This (refined) scheme is next used to guide the search for additional 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.

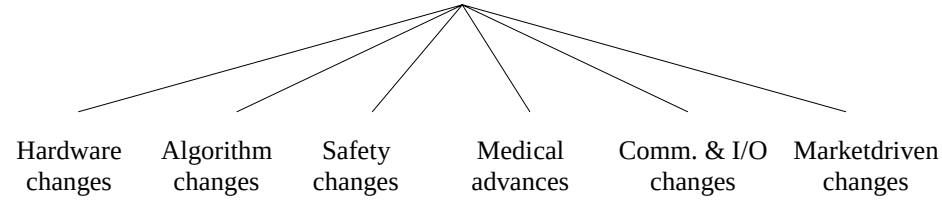

Figure 4: Deriving change categories from problem domain of a haemodialysis system [6]

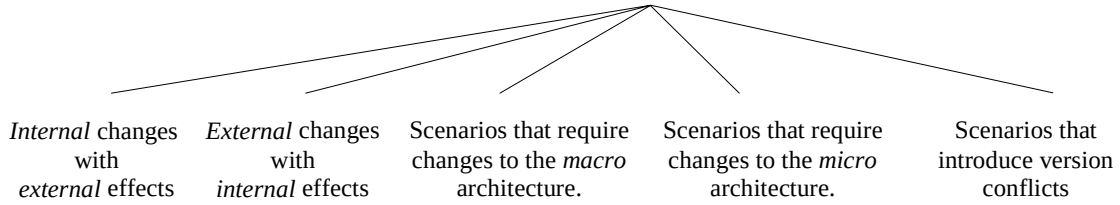

Figure 5: Deriving change categories from knowledge of complexity for business information systems [19]

The change scenario elicitation process in SAAM [14] is essentially bottom-up. It is assumed that the people involved in the elicitation process, viz. the domain experts, architects, analysts, and the like, collectively possess the necessary knowledge of the architecture under evaluation to come up with a sufficiently complete set of scenarios. There is no explicit stopping criterion; the process stops when the participants decide so.

The process in the method proposed by Bengtsson and Bosch [5] is iterative in nature. They start with a bottom-up step. The change scenarios thus obtained are used to form a hierarchical classification structure. This classification structure is based on the knowledge gained so far about the domain and system structure. Next, a top-down step follows to fill in holes still present in this classification structure. This process iterates until the analyst feels a sufficiently complete coverage has been obtained.

The process in the method proposed by Lassing *et al.* [19] is top-down. A predefined classification scheme of complex change categories is used to guide the elicitation process. The process stops if each of the categories has been explicitly considered.

6.2 Elicitation techniques

More than one technique can be used for eliciting the scenarios. The elicitation technique to use depends on the characteristics of the set of change scenarios that is desired, 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.

For each of these goals, we will now discuss the issues related to the elicitation of a relevant set of scenarios.

6.2.1 Maintenance prediction

When the goal of the analysis is maintenance prediction, the preferred elicitation technique is to interview the stakeholders for the changes they anticipate. This is the bottom-up approach, where the initiative is with the stakeholder being interviewed. The stopping criterion, in this case, is that all the selected stakeholders have been interviewed for their change scenarios and that they have commented the change scenario classification made and none of the added change scenarios affect the classification.

For the purpose of maintenance prediction, we do not only require the scenarios themselves, but also some sort of weight to indicate their importance. This weight is used in the interpretation and determines the influence that the scenario has on the end result. An important issue is that each scenario is representative for a class of changes, i.e. its equivalence class. The weight should thus be assigned based on the characteristics of the whole equivalence class. Most likely, we would choose the weight to represent the estimated probability of occurrence of an arbitrary scenario of the underlying equivalence class. The stakeholders are the most important source for assigning the weights, because they are able to judge the importance of the scenarios.

6.2.2 Risk assessment

When the goal of the analysis is risk analysis the preferred technique for scenario elicitation is to focus the interviews on changes that are expected to be complex. This is the top-down approach, where the interviewer guides the elicitation process, and actively tries to fill in all the holes. Although the interviewer guides the process, it is important that the stakeholders being interviewed bring the change scenarios forward. This prevents irrelevant scenarios from being included in the analysis. The equivalence classes that are found in this process represent groups of scenarios that have a similar complexity, i.e. pose a similar risk.

We mentioned earlier that a classification scheme could be used as a guide in the elicitation process. The classification scheme that is used in this case should be based on knowledge of complex changes. This knowledge is most likely domain specific; changes that are considered complex in one domain are not necessarily complex in another domain as well. Table 1 shows a classification scheme for finding complex scenarios within the area of business information systems [19].

In the interviews, the analyst tries to cover all the cells of this scheme. For some systems a number of cells in the scheme remain empty, but they should at least be considered. For instance, for a stand-alone system, there will be no scenarios in which the environment plays a role. The stopping criterion is closely related to this scheme: we stop the elicitation process when all cells have been considered explicitly.

Table 1: Classification scheme for eliciting complex change scenarios

	Changes in the functional specification	Changes in the requirements	Changes in the technical environment	Other sources
Change scenarios that require adaptations to the system and these adaptations have external effects	*Functions of the system under analysis are used by other systems. One of them is modified and, as a result, other systems have to be modified as well.*	*A requirement of the system is changed (for example a higher performance is required). To realize this, systems that are used by the system under analysis have to be modified as well.*	*Part of the technical environment of the system is modified. Because this part of the technical environment is shared with other systems, these systems have to be modified as well.*	*For reasons, other than the aforementioned three, part of the system has to be adapted. As a result of these changes, other systems have to be adapted as well.*
Change scenarios that require adaptations to the environment of the system and these adaptations affect the system	*The system uses functions of other systems. One of these functions is modified and, as a result, the system under analysis has to be modified as well.*	*A requirement of another system is changed. To realize this, the system under analysis has to be modified as well.*	*Part of the technical environment is modified for another system. Because this part of the technical environment is used by the system under analysis, it has to be modified as well.*	*For reasons, other than the aforementioned three, a system has to be adapted. As a result of these changes, other systems have to be adapted as well.*
Change scenarios that require adaptations to the macro architecture	*Because of changes in the functional specification, the structure of systems has to be modified.*	*Changes in the requirements cannot be realized in the current macro architecture and, therefore, the structure of systems has to be modified.*	*Necessary changes in the technical environment require the structure of systems to be modified.*	*Because of other changes, the structure of systems has to be modified.*
Change scenarios that require adaptations to the micro architecture	*Because of changes in the functional specification, the internal structure of the system under analysis has to be modified.*	*Changes in the requirements cannot be realized in the current micro architecture and, therefore, the internal structure of the system under analysis has to be modified.*	*Necessary changes in the technical environment require the structure of the system to be modified.*	*Because of other changes, the structure of the system has to be modified.*
Change scenarios that introduce version conflicts	*Changes in the functional specification introduce different versions of the same component. However, these different versions cannot coexist because of conflicts*	*Changes in the requirements introduce different versions of the same component. However, these different versions cannot coexist because of conflicts*	*Changes in the technical environment introduce different versions of the same component. However, these different versions cannot coexist because of conflicts*	*Other changes introduce different versions of the same component. However, these different versions cannot coexist because of conflicts*

6.2.3 Candidate architecture comparison

When the goal of the analysis is to compare candidate architectures, only those scenarios that expose differences between these architectures are of interest in the analysis. Therefore, it is important that the interviewer focuses the elicitation process on finding those scenarios. This means that a top-down approach is preferred, where the analyst has some knowledge of the dissimilarities between the candidate architectures. To come to a meaningful comparison between the candidates, we should have a single set of change scenarios that is evaluated for both candidate architectures.

7. Evaluation of the effect of scenarios

Once we have a set of change scenarios that we are confident with, the subsequent activity is to evaluate their effect on the architecture and express the results in a way suitable for the goal of our analysis. The purpose of evaluating the change scenarios is to trace and estimate the impact they will have. The trace is concerned with finding out the modifications needed including possible ripple effects. The estimate is concerned with determining the extent of the impact. In fact, evaluation of change scenarios is impact analysis at the architectural level.

7.1 Existing approaches to impact analysis

To the best of our knowledge no impact analysis technique for software architecture has been published yet. Turver and Munro [29] propose an early impact analysis technique based on the documentation, the themes within the documents, and a graph theory model. Other authors [7, 17] have discussed impact analysis methods focused on source code. Based on the source code and a particular change to that source code, Kung *et al.* [17] propose a way to do automatic ripple effects analysis of the change. Lindvall and Sandahl [24] have reported on a study on the accuracy of the designers in predicting necessary changes. Empirical results on source code impact analysis indicate that software engineers, when doing impact analysis predict only half of the necessary changes. However, it is noted that the modifications identified are correct. Lindvall and Runesson [23] reports 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.

7.2 Steps in scenario evaluation

In practice the process of impact analysis is not clearly divided into sequential steps. However, to visualize what is done we divide the evaluation process into the following steps:

1. List the affected components
2. Locate the operations that have to be modified
3. Determine ripple effects

The first step is to determine what components are affected by the change described in the change scenario. Change scenarios provide a description of a specific change, either in the environment of the system or the system itself. In cooperation with the architects and designers, the analyst determines which components of the system have to be modified for the change scenario. Changes may propagate over system boundaries, either changes in the environment impact the system or changes to the system affect the environment. In the first case, the analyst considers the relationships between the system and its environment and determines what components of the system are affected. In the latter case the analyst also considers these relationships and determines the systems in the environment that are affected.

The second step is concerned with identifying the functions of the components that are affected by the changes. These functions are found by studying the interface of the component. We also need to consider the systems in the environment and their interfaces. Depending on the detail and amount of information we have available we may have to make assumptions on these interfaces.

The third step is to determine the ripple effects of the implementation 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 have to make assumptions on

the occurrence of ripple effects. At one extreme, we may assume that there are no ripple effects. This is a very optimistic assumption and it is not very realistic. Alternatively, we may assume that each component related to the affected component requires changes. This is an overly pessimistic assumption and it will overexaggerates the effect of a change scenario. As mentioned in section 5.2, more information on lower level design and implementation, for instance using design patterns, may improve our prediction of ripple effects.

7.3 Expressing the results of scenario evaluation

To use the results of the scenario evaluation in our analysis we must express them in some way. We can choose to express the results is a qualitative way, e.g. a description of the changes that are needed for each change scenario in the set. On the other hand, we can also choose to express the results in a quantitative way. We can give a ranking between the effects of scenarios, e.g. a five level scale (++, +, +/-, -, --) for the effect of a scenario. This allows us to compare the effect of scenarios. We can also make absolute statements about the effort required for a scenario, such as an estimate of the size of the required modification. This can be done using metrics like lines of code, function points or object points.

For predicting the maintenance effort the analyst needs data about the size of the modification of existing components, as well as the size of new components for each change scenario. The extent of the modifications is expressed by their size, such as lines of code, function points or object points. In [5] the size of the required changes for each change scenario is expressed as the number of lines of code affected. This figure is calculated for each change scenario by summing for all components affected the product of their estimated size and the percentage of change. They assume that this measure correlates with the effort required for implementing the changes.

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. In [18] an evaluation model for risk assessment for business information systems is proposed. In this model, the results of scenario evaluation are expressed as a set of measures for both the micro and the macro architecture level. The factors considered are the impact level, whether multiple owners are involved in the changes, and the introduction of version problems. The impact level is expressed on an ordinal scale: (1) the change scenario has no impact, (2) the change scenario affects a single component, (3) the change scenario affects several components or (4) the change scenario affects the software architecture. The fourth level indicates that the current structure of the system (at the micro architecture level) or systems (at the macro architecture level) is no longer usable when the scenario has to be implemented. The involvement of multiple owners is indicated using a Boolean value. The occurrence of version conflict is also 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 versions is a complicating factor, because it complicates configuration management and requires maintenance of the different versions. Moreover, when the different versions of the components conflict with each other, the scenario might be impossible to implement. In addition to these factors, the initiator of the scenario is also recorded, because it shows the source of the change. This results in a diagram like Table 2. As mentioned, this model originated from work in the area of business information system. In other areas, a set of different measures may be more suitable to expose potential risks.

Table 2: The results of scenario evaluation for risk assessment

		Macro architecture level			**Micro architecture level**		
	Initiator	Impact level	Multiple owners	Version conflicts	Impact level	Multiple owners	Version conflicts
Change scenario		1-4	+/–	1-3	1-4	+/–	1-3

The way the results should be expressed when comparing software architectures depends on the comparison criterion that is used. We can express the differences between the candidate architectures in three ways:

- For each scenario we can determine the candidate architecture that supports it best, or if there are no differences
- For each scenario we can rank the candidate architectures depending on their support for the scenario
- For each scenario we can determine the effect on all candidate architectures and express this effect using some scale, e.g. a five level scale such as the one we introduced earlier or the number of lines of code affected

In the next step, these results are then interpreted using the comparison criterion.

8. Analysis results and 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 on the goal of the analysis and the system requirements. In this section, we will discuss how the results should be interpreted for each of the analysis goals.

8.1 Interpretation for maintenance cost prediction

Prediction of maintenance effort requires a model that is based on the cost drivers of the 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, Stark and Oman [26] studied three categories of maintenance strategies and found that there are significant differences in cost between these strategies. The prediction model that is used in the analysis should fit the maintenance strategy of the organization.

Bengtsson and Bosch [5] propose a prediction model that is aimed at predicting the maintenance effort. They use the estimated size of the modifications and the weights of the scenarios to calculate the average required effort per change scenario. The total maintenance effort for a certain period of time is then found by multiplying the predicted number of changes to be implemented in that period by the average effort per change scenario.

8.2 Interpretation for risk assessment

For risk assessment, the analyst will have to interpret the results of the scenario evaluation and determine which scenarios are risks for the system. This interpretation has to be done in consultation with stakeholders. Together with them, the analyst estimates the likelihood of each scenario and whether it poses a threat to the system.

For example, if the results of the evaluation of the change scenarios are presented as in Table 2, we can draw the following conclusions. The initiator of the scenario tells us whether the scenario can be avoided or not. A scenario initiated by the system's owner that affects the system's environment may not be feasible at all, because owners of other systems may be reluctant to implement the required changes. Also, some changes are initiated in the system's environment and require the system under analysis to be modified. Although they may not be directly beneficial to the owner of the system under analysis, they may have to be implemented anyway. When the system cannot be modified to these changes, this scenario could be a risk for the system. The next measure included in Table 2 is the impact level. The impact level allows us to understand the magnitude of the required changes and their complexity. The impact is interpreted differently at the macro architecture level than at the micro architecture level.

Changes at the macro architecture level involve the environment and, therefore, they are generally more complex to implement. In addition, changes that require either the micro or the macro architecture to be adapted are also considered more complex than changes for which just a number of components have to be adapted. Changes that involve multiple owners are generally also more complex, because of the additional coordination that is required between them. Changes are also considered to be more complex if they introduce multiple versions of the same component. Multiple versions of the same component may prevent the scenario from being implemented, in case of version conflicts, or introduce additional, possibly unforeseen, changes.

8.3 Interpretation for software architecture comparison

When the goal of the analysis is to compare candidate architectures, the interpretation is aimed at selecting the best candidate. In section 7.3 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 effort. These scenarios do not help discriminate between the candidates and are ignored in the interpretation. This interpretation of the results allows the analyst to understand the differences between the candidate architectures in terms of concrete changes.

When we rank the candidates for each scenario, we get an overview of the differences between the group of candidate architectures. Based on some selection criterion, we can then select the candidate architecture that is most suitable for the system. For instance, we can select the candidate that is considered best for most scenarios. Or, alternatively, we can 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. Using the first interpretation, the analyst considers the results of all scenarios to select a candidate. For instance, the analyst could consider only the most important scenarios to see which candidate architecture supports them the 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 architecture using some kind of algorithm. For example, this can be done using the prediction model in section 8.1. 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.

9. Software architecture analysis of modifiability in practice

In this section we illustrate the use of our method using a number of examples of analysis of modifiability. These examples are based on case studies in which we successfully applied our approach. Not only do the domains of the systems in these examples differ, but the goals of the analyses also differ. Section 9.1 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 9.2 based on an analysis of a logistic system that we performed at DFDS Fraktarna AB. Finally, in section 9.3 architecture selection is illustrated using an analysis of an embedded system that was performed at EC-Gruppen AB.

9.1 Maintenance prediction

We illustrate the case where the goal is maintenance prediction based on the case study we performed at Ericsson Software Technology AB. Ericsson Software Technology develop a system called the Mobile Positioning Centre (MPC). MPC is a system for locating mobile phones in a cellular network and reporting the geographical position. The cellular network operators then implement their own services based on the position received from the MPC. 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 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 the customers for the services they provide.

We set the goal to be a prediction of the maintenance effort. The system already existed in one version, and Ericsson was developing the second version, so we used the descriptions of the software architecture that were already available. The architecture description contained the following views, which cover the information described in section 5.2, except for size measures:

- Systems Environment
- Sequence Diagrams
- Logical View
- Component view (Figure 6)

For the elicitation technique we did the change scenario elicitation bottom up, as described in section 6.1, to focus more on the likely change scenarios. We interviewed the following stakeholders of the MPC to elicit change scenarios:

- the software architect,
- the operative product manager,
- a designer developing the MPC system, and
- a applications designer developing services using the MPC.

We had a meeting with each of the selected stakeholders to get information about the system and the software architecture and to elicit change scenarios. The meetings started with an introduction to the system or the software architecture held by the interviewed stakeholder, and then shifted into the elicitation interview. The first stakeholder we interviewed was the software architect of the MPC. The following scenarios are examples from the 32 change scenarios that were elicited during the interviews:

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

After all the selected stakeholders were interviewed, we categorized the change scenarios into different categories based on the information we had received about the system. We removed duplicate scenarios from the set. The interviewed stakeholders reviewed the categories and change scenarios, to see if any categories or scenarios were missing or were superfluous. After incorporating their comments we came to six categories of change scenarios. These are some examples of the final categories:

- Positioning Method Changes
- Changes to the mobile network
- Changes of position request interface/protocol

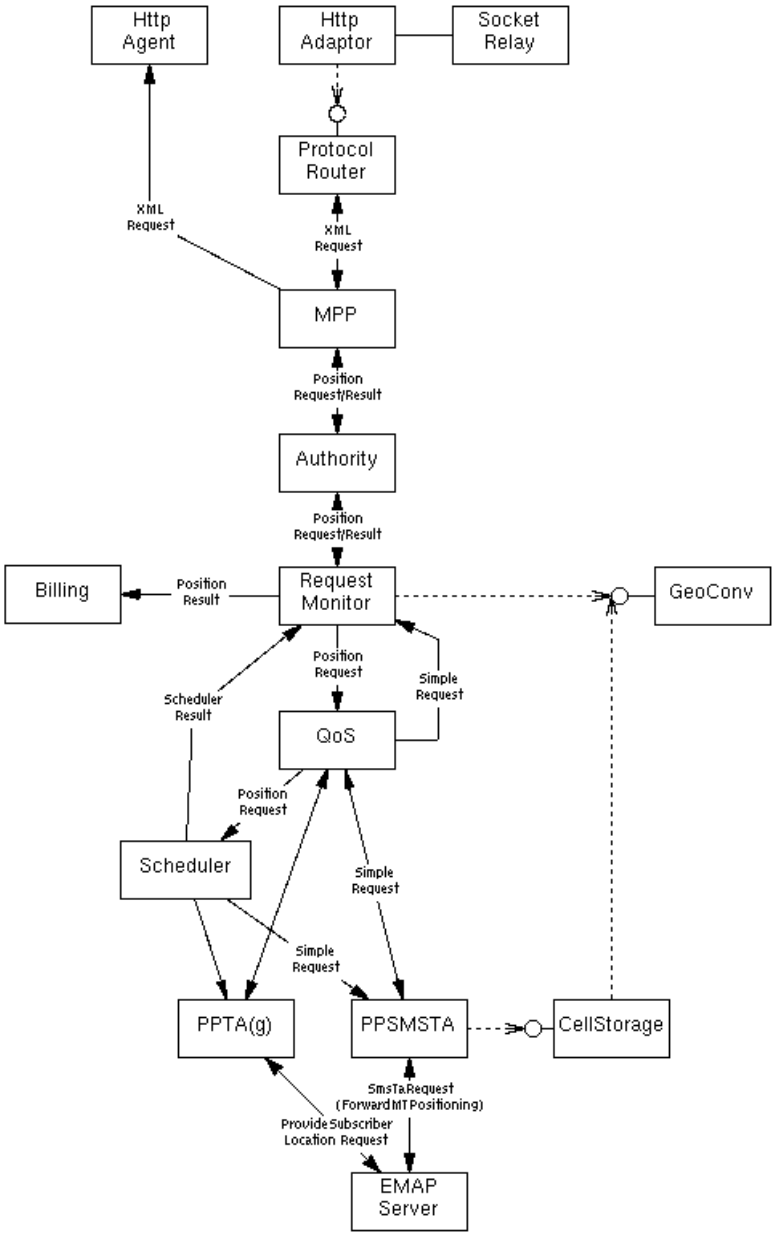

Figure 6: The component view for a part of the MPC

After that, the information about the scenarios and architecture needed to be supplemented with size measures for each component of the software architecture (section 5.2) and also a weight for each change scenario in the set (section 6.2.1). The architect presented us with the measure or estimated size of each component in the software architecture, expressed as lines of code. The operational product manager provided us with the estimated weightings for each change scenario in the set. We choose to ask the operative product manager for the probabilities because the future of the product is more related to this role than the designer and the architect. The weights of the scenarios are then normalized by dividing the estimated weight of the scenario by the sum of the weights of all the scenarios. The previously presented scenarios then get a normalized weight, like in table 3.

Table 3: Change Scenarios with normalized weight

ID.	Change Scenario	Weight
1	Change MPC to provide a digital map with position marked.	0.0345
2	New geodetic conversions methods per installation.	0.0023
…	…	…
	Sum:	1.0000

Based on the now weighted change scenario list we performed the evaluation in cooperation with the architect as described in section 7.2 and 7.3. For each scenario we estimated:

- The ratio of the estimated modification of each component affected by the scenario.
- The size of any new component added to the architecture.

The results were, per change scenario, a set of estimates of the modification volume for each affected component, as depicted in table 4.

Table 4: 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
			183 + 173 * 0.5 = ~ 270 LOC / change	

We separated 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. For the prediction we assumed that the productivity was on the order of twice as high for new code. The average effort was then calculated according to the description in table 4. Note that the table excludes some 30 scenarios that make the figures somewhat different in the real case. Multiplying by the expected number changes and the productivity measure of the maintenance organization then made the prediction of the weighted average change size.

9.2 Risk assessment

We will illustrate the case where the goal of the analysis is risk assessment, based on the analysis that we performed for DFDS Fraktarna AB, a Swedish distributor of freight. The system that we investigated is called EASY and it will be used to track the position of freight through the company's distribution system, which consists of a number of terminals spread all over Sweden. During our analysis, EASY was under development by Cap Gemini Sweden.

Our first step in this analysis was to get insight into the functionality and the software architecture of the system. To do so, we studied the available documentation and interviewed one of the architects and a designer. In the description of the architecture we did not only address the internals of the system, but also the relationship of the system to its environment. The latter was important in this case, because ill-designed relationships between a system and its environment may be a source of risks. This step resulted in the following four views:

- context view: an overview of the systems with which EASY will interact
- technical infrastructure view: the relationships of EASY to the technical infrastructure
- conceptual view: an overview of the architectural approach taken for EASY

– development view: the organization of EASY in the development environment

For each of these views, we used an informal notation technique supplemented with a textual description, similar to the techniques used by the development team. This is illustrated for the context view in Figure 7. This figure shows that EASY communicates with two other systems: the domestic freight system and the Track & Trace system. The details of this communication should be added in a textual description. For instance, we should add that the communication between the systems takes place through messages using an asynchronous, message-oriented, middleware.

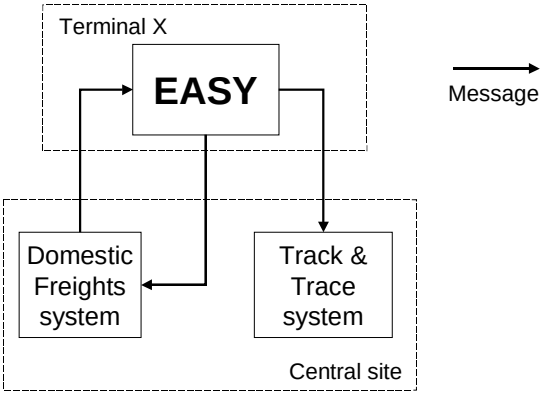

Figure 7: Context view of EASY

Since the goal of the analysis was to assess risks, the elicitation process was directed at finding scenarios that had complex associated changes. Scenario elicitation already started during the interviews with the architect and the designer. After that, we interviewed a stakeholder from the user organization as well, to get a more balanced picture of scenarios. In this elicitation process, the architectural description served as a guide to find complex scenarios. Based on our knowledge of complexity of changes we asked the stakeholders for specific scenarios. To do so, we used the scheme in Table 4, which was given before in section 6.2.2. This led, for instance, to the change scenario that the middleware used by the Track & Trace system changes. This scenario represents the equivalence class of all changes to the middleware, i.e. it is not specific for a special type of middleware.

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 Track & Trace system. This change scenario falls in the category 'Change scenarios that require adaptations to the environment of the system and these adaptations affect the system' and its source is a change in the technical environment. Similarly, we tried to discover change scenarios for all categories.

The above-mentioned scheme was also used as the stopping criterion in this analysis. By going through all cells in the scheme we were able to judge whether all risks were considered. In Table 3, the cells for which we found one or more scenarios are indicated with a '+' and the cells for which we did not are indicated with a '–'. We see a large number of minus signs in this table. The reason for this is that the integration between EASY and other systems is not very tight. As a result, we were not able to find many scenarios that included that environment.

Table 5: 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 macro architecture	–	–	–	–
Change scenarios that require adaptations to the micro architecture	+	+	–	–
Change scenarios that introduce version conflicts	–	–	–	–

As a next step, we determined the precise effect of the scenarios. To do so, we used the evaluation model for risk assessment of business information systems that was introduced in section 7.3. To demonstrate the use of this model, we will elaborate the scenario we mentioned earlier that explored the effect of a change of the middleware used in the Track & Trace system. At the macro architecture level, multiple systems have to be adapted for this scenario, viz. the Track & Trace system itself and EASY. These systems have different owners meaning that coordination between these owners is required to implement the changes: a new release of the Track & Trace system can only be brought into operation if the new release of EASY is finished, and vice versa. Because EASY has to be adapted for this scenario, we also had to investigate the micro architecture. At this level we found that several components of the system need to be adapted, because access to the middleware is not contained in one component. However, these components have the same owner, so there will not be multiple owners involved in the changes at the micro architecture level. In addition, only one version of these components will exist, so there will not be any version conflicts. Table 4 summarizes these results.

Table 6: Evaluation of scenario

		Macro architecture level			**Micro architecture level**		
	Initiator	Impact level	Multiple owners	Version conflicts	Impact level	Multiple owners	Version conflicts
Change of middleware in Track & Trace	Owner of Track & Trace	3	+	1	3	–	1

This table shows us that this scenario is initiated externally, but that it affects EASY. This means that the owner of EASY should reckon with the fact that the system has to be adapted due to this external event. Whether this is a risk for the system is up to the owner of the system to decide. So, using this approach, part of the interpretation is left to the stakeholders. Something similar was done for the other scenarios.

9.3 Architecture selection

We illustrate the case of comparing two architectures using the analysis we have performed on 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 research group at the University of Karlskrona/Ronneby. This project has also been reported on in Bengtsson and Bosch [4]. The inspection system is an embedded

system located at the beginning of a beer can filling process and its goal is to remove dirty beer cans from the input stream. Clean cans should just pass the system without any further action.

As a step in the software architecture design iterations we performed architecture analysis to compare the new architecture with the old to confirm that it was actually better. In total we made six iterations and six analyses. To illustrate the comparison we will look into more detail on an intermediate version of the software architecture (Figure 8) and the final version of the software architecture (Figure 9). The approach we took was the internal analysis approach, i.e. we designed the software architecture and we analyzed the software architecture as well. The benefit of this approach was that we were able to cope with the problem that not all information described in section 5.3 was present in the rather rudimentary architecture descriptions that we had made, as illustrated in Figure 8 and Figure 9.

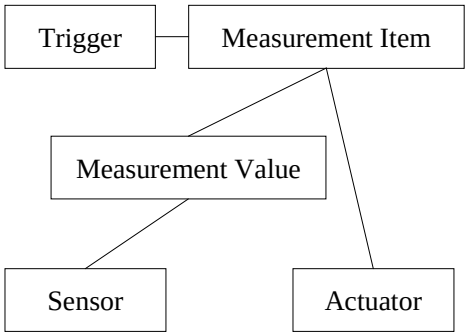

Figure 8: Beer can inspection architecture A

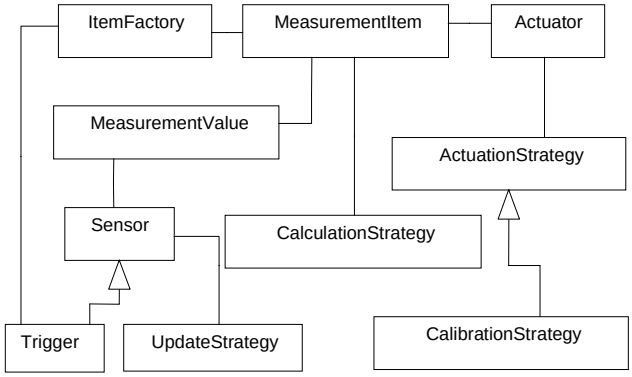

Figure 9: Beer can inspection architecture A'

We elicited the set of change scenarios in a rather informal fashion and based it on the discussions with the stakeholders that had already taken place. We were focused on finding change scenarios that described the changes that were likely to appear in the future of the system. In fact, we did not categorize the scenarios because the scenarios themselves were quite abstract. The scenarios that we found that way included 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.

For all iterations we used the same set of change scenarios and expressed the impact as the component modification ratio, i.e. modified components divided by the total number of components. And the results of the analyses of the two architecture in this example was the following:

Table 7: Results of the analysis

	Software Architecture A	**Software Architecture A'**	
Change scenario 1	1/5	2/10	*Same*
Change scenario 2	4/5	2/10	*Improved*
Change scenario 3	5/5	2/10	*Greatly Improved*
Change scenario 4	3/5	3/10	*Improved*
Change scenario 5	5/5	10/10	*Same*

In Table 5 we can 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. Figure 10 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 fifth iteration worsened the results. However, the same iteration did greatly improve the results for change scenario 3 (S3).

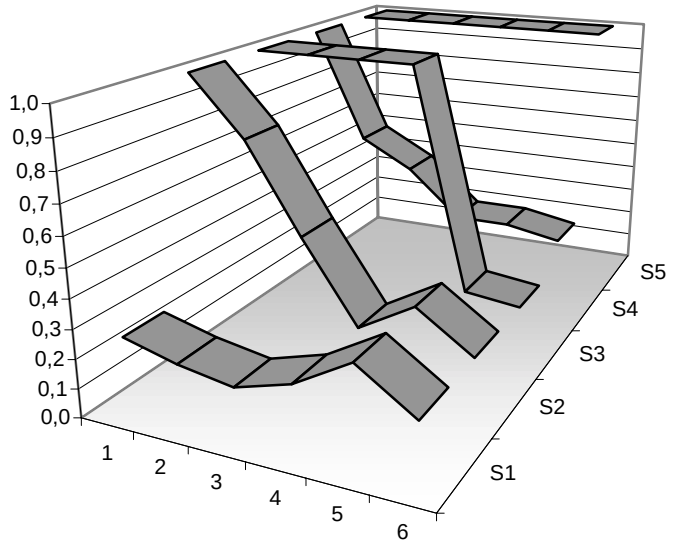

Figure 10: Analysis results for all design and analysis iterations

10. Summary

In this paper we propose 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.

In software architecture analysis of modifiability, we can pursue one of the following goals: maintenance prediction, risk assessment and software architecture selection. 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 selection 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 be detailed enough 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: 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 and discussed the issues and techniques for each of the steps. Finally, we have illustrated our method by elaborating three examples of analyses. In each of these analyses a different goal was pursued.

11. 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 Technologies, 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 Technologies, 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.

12. References

[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop and A. Zamerski, *Recommended Best Industrial Practice for Software Architecture Evaluation.* (CMU/SEI-96-TR-025). Pittsburg, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996.

[2] AT&T, *Best Current Practices: Software Architecture Validation*, Internal Report, AT&T, 1993.

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

[4] P. Bengtsson and J. Bosch, 'Scenario-based Software Architecture Reengineering', in *Proceedings of the 5th International Conference on Software Reuse (ICSR5)*, Los Alamitos, CA: IEEE CS Press, pp. 308-317, 1998.

[5] 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.

[6] P. Bengtsson and J. Bosch, 'Haemo Dialysis Software Architecture Design Experiences', In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. Los Angeles, CA: ACM Press, 1999, pp. 516-525.

[7] S.A. Bohner, 'Software Change Impact Analysis for Design Evolution', In *Proceedings of 8th International Conference on Maintenance and Re-engineering*, Los Alamitos, CA:IEEE CS Press, pp. 292-301, 1991.

[8] L. Briand, E. Arisholm, S. Counsell, F. Houdek and P. Thévenod-Foss, 'Emprical studies of Object-Oriented Artifacts, Methods, and Processes: State of The Art and Future Directions', *Empirical Software Engineering*, 4 (4): 387-404, 1999.

[9] R. Harrison, '*Maintenance Giant Sleeps Undisturbed in Federal Data Centers'*, Computerworld, March 9, 1987.

[10] C. Hofmeister, R. Nord and D. Soni, *Applied Software Architecture*, Reading, MA: Addison Wesley Longman, 1999

[11] IEEE Architecture Working Group. *Recommended practice for architectural description*. Draft IEEE Standard P1471/D4.1, IEEE, December 1998.

[12] ISO/IEC, Information technology – Software product quality –Part 1: Quality model, ISO/IEC FDIS 9126-1:2000(E)

[13] I. Jacobson, M. Christerson, P. Jonsson and G. Övergaard, *Object-oriented software engineering. A use case approach*, Readings, MA: Addison-Wesley, 1992.

[14] R. Kazman, G. Abowd, L. Bass and P. Clements, 'Scenario-Based Analysis of Software Architecture'. *IEEE Software*, 13 (6):47-56, 1996.

[15] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson and J. Carriere, 'The Architecture Tradeoff Analysis Method', In *Proceedings of the 4th International Conference on Engineering of Complex Computer Systems (ICECCS98)*, Montery, CA: IEEE CS Press, pages 68-78, 1998.

[16] P.B. Krutchen, 'The 4+1 View Model of Architecture', *IEEE Software*, pp. 42-50, November 1995.

[17] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. 'Change Impact in Object Oriented Software Maintenance', In *Proceedings of the Conference on Software Maintenance (ICSM'94)*, Los Alamitos, CA: IEEE CS Press, pp. 202-211, 1994.

[18] 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.

[19] N. Lassing, D. Rijsenbrij and H. van Vliet. '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*, Amsterdam, Netherlands: Vrije Universiteit, 1999.

[20] N. Lassing, P. Bengtsson, H.. van Vliet, J. Bosch and D. Rijsenbrij. Experiences with SAA of Modifiability. Högskolan Karlskrona/Ronneby Technical Report, HK-R-RES--00/10—SE, *Submitted for publication*, 2000.

[21] W. Li and S. Henry, 'OO Metrics that Predict Maintainability', *Journal of Systems Software*, 23(1):111-122, 1993
[22] B.P. Lientz and E.B. Swanson. *Software Maintenance Management*, Reading, MA: Addison-Wesley. 1980.
[23] 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.
[24] M. Lindvall and K. Sandahl, 'How Well do Experienced Software Developers Predict Software Change?', *Journal of Systems and Software*, 43(1): 19-27, 1998.
[25] J.A. McCall, P.K. Richards and G.F. Walters, *Factors in Software Quality*. RADC-TR-77-369, US Dept. of Commerce, 1977.
[26] G.E. Stark and P.W. Oman, 'Software Maintenance Management Strategies: Observations from the Field', *Software Maintenance: Research and Practice*, Vol. 9, 365-378, 1997.
[27] A.G. Sutcliffe, N.A.M. Maiden, S. Minocha and D. Manuel, 'Supporting Scenario-Based Requirements Engineering', *IEEE Transactions on Software Engineering*, 24(12):1072-1088, 1998.
[28] E.B. Swanson, 'The Dimensions of Maintenance', In *Proceedings of the 2nd International conference on Software Engineering*, Los Alamitos: IEEE CS Press, pp. 492-497, 1976.
[29] R. J. Turver, and M. Munro, 'An Early Impact Analysis Technique for Software Maintenance', *Software Maintenance: Research and Practice*, Volume 6, pp. 35-52, 1994.