JOURNAL OF INTERNATIONAL ACADEMIC RESEARCH FOR MULTIDISCIPLINARY
Impact Factor 1.393, ISSN: 2320-5083, Volume 2, Issue 1, February 2014

METHOD FOR SOFTWARE MAINTENANCE RISK ASSESSMENT AT ARCHITECTURE LEVEL

RONALD TOMBE*
DR. STEPHEN KIMANI**
DR. GEORGE OKEYO***

* Dept. of Computing Sciences, Kisii University, Kenya
** Dept. of Computing, Jomo Kenyatta University of Agriculture and Technology, Kenya
***Dept. of Computing, Jomo Kenyatta University of Agriculture and Technology. Kenya

ABSTRACT+

Successful software project maintenance necessitates a well-defined strategy to manage changes and minimize risks associated with the future operation of the software. Software maintainers usually are not engaged in the initial software development cycle. Before maintainers can modify a program, they must understand how it operates. The community of Software engineering has proposed several methods to evaluate software architectures with respect to desired quality attributes performance, usability, and so on. There is, however, little effort on a systematically way for risk assessment at the architecture analysis level. It is difficult to find exact estimates for the probability of failure of individual components and connectors in the system during the early phases of software life cycle, thus risk assessment and analysis for software architectures can be performed on UML specifications such as scenarios and use cases since they model the abstract architecture and implementation details and describe the system using compositions of components and connectors. In this paper, we analyse the well known scenario-based software architecture evaluation methods using an evaluation framework created in this paper. The framework considers each method from the point of view of method context, stakeholders, structure, and reliability. The comparison reveals that most of the studied methods are structurally similar but there are a number of differences among their activities and techniques. Hence, some methods overlap, which guides us to identify five activities that can form a method for software risk Assessment at architecture level during maintenance.

1 INTRODUCTION

Software maintenance is classified into adaptive, corrective, preventive and perfective (Somerville, 1996). Most organizations are concerned about the costs of software maintenance, for it has been increasing steadily and many companies spend approximately 80% of their software budget on maintenance (Pigoski, 1997). The process of risk assessment is useful in identifying complex modules that require detailed inspection, estimating

potentially troublesome modules. According to the NASA-STD-8719.13A standard, risk is a function of the anticipated frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity. This standard defines several types of risk such as, for example, availability risk, acceptance risk, performance risk, cost risk, schedule risk, etc. For the purpose of this research, the definition by NASA-STD-8719 is adopted which defines risk as a combination of two factors: probability of malfunctioning (failure) and the consequence of malfunctioning (severity). The probability of failure depends on the probability of existence of a fault combined with the possibility of exercising that fault in a scenario in which a failure will be triggered. Though a fault is a feature of a system that precludes it from operating according to its specification, a failure occurs if the actual output of the system for some input differs from the expected output (Ammar, 2000).

Software Architecture models abstract design and implementation details and describe the system using compositions of components and connectors (Garlan, 1996). A component can be as simple as an object, a class, or a procedure, and as elaborate as a package of classes or procedures. Connectors can be as simple as procedure calls or as elaborate as client-server protocols, links between distributed databases, or middle wares. Risk assessment during maintenance can be performed at different phases. This research envisages that risk assessment at the architecture level is more beneficial for early detection and correction of problems which would be much less costly than detection at the code level. Hence, to design the software architecture to meet the quality requirements is to reduce the risks of not achieving the required quality levels. The authors of this paper propose a method which can be applied for software maintenance risk assessment at the architecture analysis level. The method will be useful to the software engineers in assessing the maintenance risks that are likely to occur in order to mitigate them before expensive resources are used for programming parts that could be removed later in the development process.

1.1 Research questions

1. What are the current approaches to software architecture analysis methods?
2. How will the maintenance tasks of software under revision be formulated to map into architecture design for maintenance task risk assessment purposes?
3. How will the risks that are likely to occur during the maintenance of software be assessed?

2 Background Work

2.1 Software architecture

Garlan (1996) defines software architecture as the structure of components, and their interrelationships, and the principles and guides that control the design and evolution in time. According to (Bass, 1998) software architecture is as an abstract structural description of the software system in terms of its main components and the relationships among them.

Lassing (1999) and Risjenbrij (1999) in their analysis method of flexibility found that for architectural analysis, the external environment is as just important as the internal entity of a system. Their opinion was that the definition of software architecture should consist of two parts, namely; macro architecture, which focuses on the environment of the system, and a micro architecture which covers the internal structure of a system. Software architecture is designed to address the different perspectives that one could have of architecture (Dobrica, 2000). Each perspective is described as a view; the reason behind multiple views is to separate different aspects into separate views to help people manage complexity. The information relevant to one view is different from that of others and should be described using the most appropriate technique for each view. Several view-based architectural models have been developed. “4+1 model” is one of the popular ones which organizes software architecture into the following views (Kruchten, 1995):

1. The logical view describes the static structure of the system, as derived from its domain.
2. The process view describes the (dynamic) concurrency, distribution and synchronization aspects of the system.
3. The development view shows the (static) organization of the system in terms of technical facilities of the development environment.
4. The physical view describes the mapping of the system onto hardware, databases, and communication infrastructure.
5. The scenarios tie the other views together into externally usable system services.

Software architecture manifests its usefulness in the life cycle in the following ways:

- An architecture is often the first artifact in a design that represents decisions on how requirements of all types are to be achieved. As the manifestation of early design decisions, the architecture represents those design decisions that are hardest to change (Mettala, 1992) and hence are deserving of the most careful consideration.
- Software architecture is a key artifact in achieving successful product line engineering, the disciplined structured development of a family of similar systems

with less effort, expense, and risk than developing each system independently (Parnas, 1976).

- Architecture is usually the first artifact to be examined when a programmer (particularly a maintenance programmer) unfamiliar with the system begins to work on it.

2.2 Architectural descriptions

Architectural descriptions indicate the system's computation and data components as well as the relationship between the components (Gobrica, 2000).The result of an architectural evaluation process depends on how well the description is made.

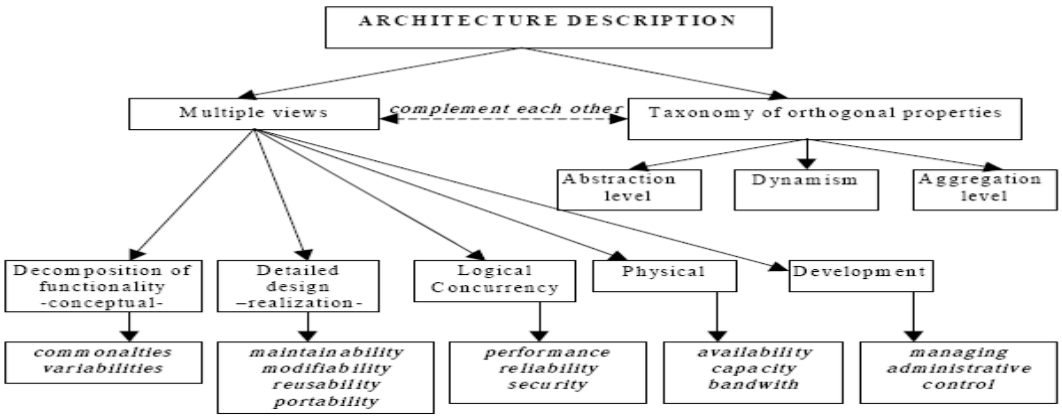

Figure 1:- Architectural description and the relevance to the analysis of quality attribute (Gobrica, 2000).

The quality attributes in the architecture description can be exemplified based on the architectural view for further assessment in meeting the software system requirements.

2.3 Software Architecture and Scenarios

Jacobson (1995) defines a scenario as a possible set of events that might reasonably take place. Its purpose is to motivate and document thinking about current problems, possible occurrences, and assumptions relating to these occurrences, action opportunities, and risks (Kazman, 1996). Scenarios are important tools for putting into effect architecture in order to attain information about a system's fitness with respect to a set of desired quality attributes (Abowd, 1998). Scenarios are have been used and documented as a technique during requirements elicitation, especially with respect to the operator of the system (Gough, 1995). Scenarios can be used to express the particular instances of each quality attribute important to the customer of a system (Rick, 1996). They have also been used during design as a method of comparing design alternatives (Abowd, 1998). Architecture under consideration can then be analyzed with respect to how well or how easily it satisfies the constraints imposed by

each scenario. Architectural analysis cannot give precise measures or metrics of fitness (Abowd, 1996). Such measures need to be understood in terms of qualities metrics. Further not all scenarios describe architecture-level issues. For example, a portability scenario might have architectural implications (such as determining how machine dependencies should be isolated). Furthermore, some scenarios simply cannot be evaluated using architectural information.

3 Method

3.1 Context

The framework consists of maintenance requirements specification using the UML use-cases to capture scenario requirements as per the software maintenance task(s)to be performed. This will then be translated into use case model (s), from which the analysis model will be derived, and then the design model of the subsystem will be designed from the analysis model to map into the existing architectural design of the existing system. In figure 6, the phases of the method are presented graphically. Each phase is explained in the sub sections below:

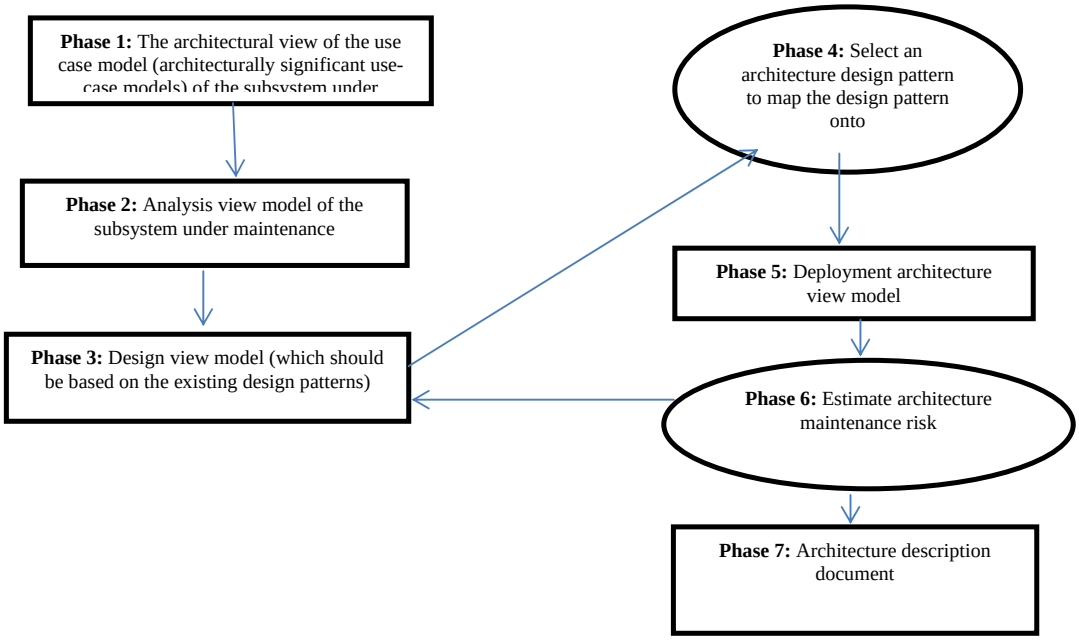

Figure 2:- A framework for software architectural analysis for maintenance risk assessment

3.1.1 Phase 1: The Architectural view of the use-case phase

To assess for maintainability, a set of scenarios is developed to concretize the actual meaning of the maintenance requirement, system users and other stakeholders are involved in

this phase. The maintainability requirements are then specified by scenarios that capture typical changes in requirements. The scenarios can then be used to evaluate the number of changes required to adapt the architecture to the new situation. The architectural view of the use-case model presents the significant actors and use cases of the subsystem under maintenance. Architecturally significant use-cases are the ones that will help mitigate the most serious risks, i.e. those that are most important to the users of the system covering the important functionality during maintenance as an initial requirement document. This provides the baseline on which the architecture is understood by maintainers so that it is operationalized.

The effectiveness of the scenario-based approach will be largely dependent on the representativeness of the scenarios. If the scenarios form accurate samples, the evaluation will also provide an accurate result.

3.1.2 Phase 2: The Analysis view phase

In this phase the structure of subsystem classifiers (analysis classes) and the relationship between the classifiers are established from use-cases analysis. The analysis models are also used to describe the collaborations that realize the use-cases. This facilitates understanding of the interaction patterns that describe how the use cases realization is performed or executed.

3.1.3 Phase 3: The design view phase

The design model phase is created using the analysis model as the primary input, but it is then adapted to the selected implementation environment. The design model defines classes, subsystems, interfaces, relationships between classes and collaborations that realize the use cases. The architectural view of the design model presents the most architecturally important classifiers of the design model: i.e. the most important subsystems, interfaces as well as the most important classes, mainly the active classes on the deployment model. DESIGN PATTERNS promote reuse of solutions to recurring design problems by naming and cataloging these solutions.

Many design patterns, for example, those described in the popular Gamma et al. (E. Gamma, 2004) book, promote adaptability, by supporting modifications through specialization. Developers can adapt a system built using these patterns by creating new concrete classes with desired functionality rather than by direct modifications to existing classes. Design structure is characterized by class-size, and class participation in inheritance relationships and design patterns.

3.1.4 Phase 4: Architectural pattern selection phase

The architectural design context affects class change proneness after accounting for the effect of single class properties such as class size. Thus the need to examine the architectural design patterns templates which are general collaborations that can be specialized as defined by the templates. Understanding the architectural patterns will be instrumental for understanding the hardware of the system that is maintained so as to help design the system on top of that hardware for instance the client/server architecture pattern defines the structure to the deployment model and suggests how components should be allocated to nodes. Many architectural patterns can be applied on a single system (Ivar and Grady, 1997).

3.1.5 Phase 5: Deployment phase

The deployment model defines the physical system architecture in terms of the connected nodes i.e. the capability of the nodes such as the processing capacity, memory size, bandwidth and availability will be considered. The nodes and connections of the deployment diagram and the allocation of the active objects should be depicted on the deployment diagram.

3.1.6 Phase 6: Risk Estimation phase

Estimation of the risk on the ripple effects of the changes (maintenance) to be made on the deployment components in respect with the interacting components in order to assess the overall risk that might be associated to during the maintenance of a system component.

3.1.7 Phase 7: Architecture description

This is a document which has the view of the models of the system, views of the use-case, analysis, design, implementation and deployment models. The architecture description describes the parts of the system that are important for all developers and other stakeholders to understand.

3.2 Analysis of Scenario-based Software Architecture Evaluation Methods

Scenario-based evaluation methods evaluate software architecture's ability with respect to a set of scenarios of interest. Scenario is brief descriptions of a single interaction of a stakeholder with a system (Bass, 1998). The scenario-based evaluation methods offer a systematic means to investigate a software architecture using scenarios. These methods determine whether software architecture can execute a scenario or not. Evaluation team explores/maps the scenario onto the software architecture to find out the desired architectural components and their interactions, which can accomplish the tasks expressed through the scenario.

3.2.1 Scenario Architecture Analysis Method (SAAM)

(Kazman, 1994) proposed SAAM in 1993 to compare competing software architectures. The goal of SAAM (Scenario-based Software Architecture Analysis Method) is to verify basic architectural assumptions and principles against documents that describe the desired properties of an application. This analysis helps assess the risks inherent in an architecture. SAAM guides the inspection of the architecture, focusing on potential trouble spots such as requirement conflicts or incomplete design specification from a particular stakeholder's perspective.

(Clements, 1995) asserted that the outputs of this method include quality sensitive scenarios, map-ping between those scenarios and architectural components, and the estimated effort required to realize the scenarios on the software architecture. The basic activities of SAAM are illustrated in Figure 2.

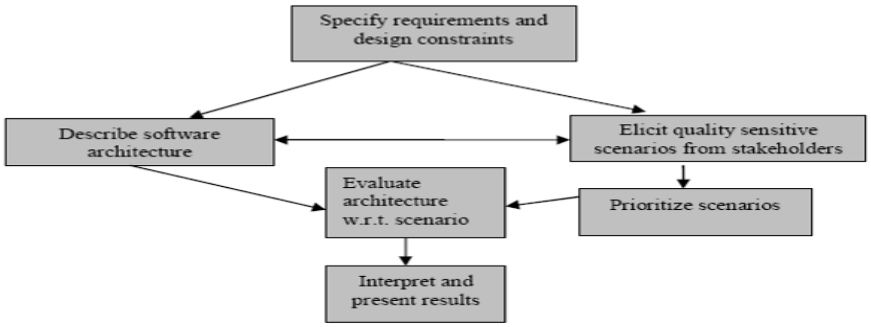

Figure 3:- (Abowd, 1998)The steps of SAAM are as follows (Kazman, 1996);

1. Describe candidate architecture: The candidate architecture is described which includes the system's computation and data components, as well as all component relationships, sometimes called connectors. The candidate architecture or architectures should be described in a syntactic architectural notation that is well-understood by the parties involved in the analysis.
2. Develop scenarios: Development of scenarios for various stakeholders; the scenarios illustrate the kinds of activities the system must support and the anticipated changes that will be made to the system over time.
3. Perform scenario evaluations: Scenarios are categorized into direct and indirect scenarios. For each indirect task scenario the required changes to the architecture are listed and the cost of performing these changes is estimated. A modification to the

architecture means that either a new component or connection is introduced or an existing component or connection requires a change in its specification.

4. Reveal scenario interaction: Different indirect scenarios that require changes to the same components or connections are said to interact at the corresponding component. Determining scenario interaction is a process of identifying scenarios that affect a common set of components. Scenario interaction measures the extent to which the architecture supports an appropriate separation of concerns. Semantically close scenarios should interact at the same component. Semantically distinct scenarios that interact indicate an improper decomposition.
5. Overall evaluation: Finally, each scenario and the scenario interactions are weighted in terms of their relative importance and this weighting used to determine an overall ranking. The weighting chosen will reflect the relative importance of the quality factors that the scenarios manifest. This is a biased process, involving all of the stake-holders in the system.

The weakness of the SAAM method on Architectural analysis is that it does not give precise measures or metrics of fitness (Abowd, 1996). Such measures need to be understood in terms of qualities metrics.

3.2.2 Extended scenario-based architecture analysis method by integration (ESAAMI)

Software architecture evaluation is a human and knowledge-intensive activity that can be expensive practice if each evaluation starts from scratch. SAAM does not put any emphasis on knowledge management for reusability (Graham, 2008) and (Roy, 2008), e.g., SAAM does not provide templates for scenarios allowing their future reuse. (Molter, 1999) proposed ESAAMI (Extending SAAM by Integration in the Domain) to integrate SAAM in a domain-centric and reuse-based development process.

The conventional SAAM analysis in an architectural-centric development process considers only the problem description, requirements, statement and architecture description. ESAAMI is a combination of analytical and reuse concepts and is achieved by integrating the SAAM in the domain -specific and reuse-based development process (figure 3) (Molter, 1999).Three factors influence the reusability of an architecture are identified by the author of this method. These factors are: a common basis for a variety of systems, a sufficient flexibility to cope with variation among systems, and the documentation of properties to make them available for the selection of architecture and its customization.

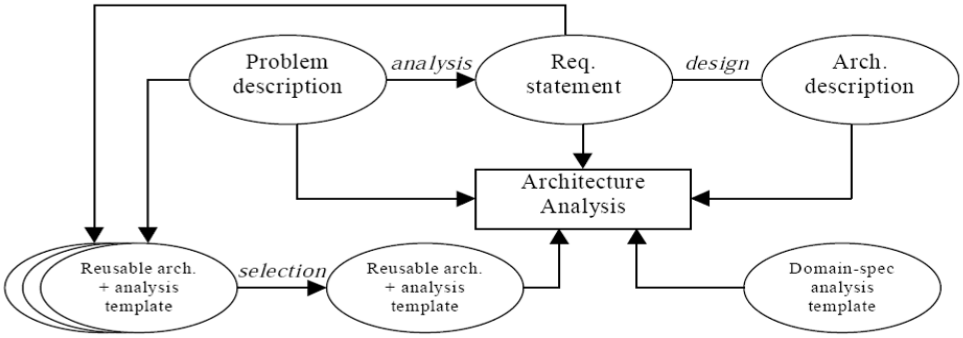

Figure 4:- SAAM integrated in the domain specific and reuse-based development (Molter, 1999).

A consistent basis of SAAM could be provided by reusable products which are collected in analysis templates. The reuse of analysis templates reduces the cost of the analysis and speeds up the process. The additional reusable products that can be deployed in the various steps of the analysis method are:

- Protoscenarios, which are generic descriptions of reuse situations or interactions with the system. These are intended to be used in the scenario elicitation step of subsequent architecture analysis after a selection and refinement process.
- Evaluation protocols, proto-evaluations and architectural hints, which are utilized during the step of scenario evaluation. These additional reusable assets are present in the protocols of the earlier evaluations in different projects, examples of descriptions of how the scenario can be performed using a set of abstract architecture elements and hints associated to each scenario indicating which architectural structures would make the scenario convenient to handle.
- Weights established in different old projects in the same domain, thus making the results of the analysis comparable.

ESAAMI extends SAAM with two new techniques (Molter, 1999). One is based on reusing the domain knowledge by providing analysis templates representing the essential features of the domain. The degree of reuse is improved by concentrating on the domain. In this context, the analysis template is formulated on an abstraction level defined by communality by a large function of the systems domain, and without referring to system specific architecture elements. The other technique is the specific knowledge about a reusable architecture. Thus, a reusable architecture is packaged with a tailored analysis template focused on the distinctive characteristic of the architecture. All this packages represent an input for the selection process

of a reusable architecture. The selected one is a starting point for the new system and method steps.

From the practical view point the first step of this method is to use a reusable architecture to be deployed in a new system. It has to be ensured that the reusable architecture provides an adequate basis for the system to meet its requirements. SAAM estimates the effort for implementing scenarios that illustrate the requirements in the target system, predicting the effort required to realize a given part of systems functionality. The selected architecture is then adapted and refined to meet the new requirements. The same set of scenarios is re-evaluated by SAAM to guarantee that the implemented system does not violate the initial design principles of the architecture, and that the initial assumption of the system still holds. The results of this analysis are themselves part of the new- built system.

Molter (1999) signals that the danger that the objectivity of the analysis may suffer due to packaging of an analysis template together with a reusable architecture. This problem results in similar to the well-known effects of solution-oriented as opposed to problem-oriented. A combination of evaluation / domain –and project- specific scenarios is a solution to avoid this type of problem. In this way, it is recommended to take project- specific properties into consideration, while at the same time exploiting knowledge about the specific architecture and the system domain.

3.2.3 Scenario Architecture Analysis Method improvement for evolution and reusability (SAMEER)

From the point of two particular quality attributes, evaluation and reusability, SAAM is extended in SAMEER (Lung, 1997) and (Kazman, 1997). The authors of SAMEER introduce a framework and a set of architectural views. The framework for information gathering and analysis consist of four activities: Gathering information about stake holders (Bot, 1996) and (Lung, 1996), Architecture, quality and scenarios; modeling reusable artifacts; analyzing; and evaluating. The method considers the following architectural views as critical for this type of software architecture analysis: static, map, dynamic, and resource. The static view integrates and extends SAAM to address classification and generalization of a systems components and functions and the connection between components. This classification and generalization of components and connections facilitates the estimation of the cost or effort required for changes to be made. Additionally, to further improve SAAM two kinds of sources of information, the required changes and domain experts' experiences, are considered. Compared to SAAM where the risk is estimated by just counting the number of changes both

the information sources give a better suggestion about how the system could support each quality objectives or the risk levels of the system evolution, or how to reuse across software domain systems.

An important point exposed by this method is that even if scenarios are considered the main drivers to evaluate various areas of architecture, the architectural views can also review deeper information (Dobrica, 2000). Scenarios describe an important functionality that the system must support or recognize, where the system will need to be changed over time. Scenarios and the structural view are effective in identifying components that need to be modified, or are useful for preventive and adaptive maintenance activities. Analysis of scenario interactions is a critical step in SAAM. A high degree of scenario interaction may indicate that a component is poorly isolated. However, the static view may show that this is just the nature of a particular architectural pattern. The dynamic view is appropriate to examine the behavior aspect to validate the control and communication to be handled in an expected manner. The mapping between components and functions could reveal the cohesion and coupling aspects of the system.

Furthermore, the method gives a practical answer to the question regarding when to stop generating scenarios. The techniques are applied here is, scenario generation which is closely tied to various types of objectives, stakeholder architecture and quality .Based on the objectives and domain experts' knowledge, the scenarios are identified and clustered to make sure that each objective is well covered. The second technique applied to validate the balance of scenarios with respect to objective is quality Function Deployment (QFD) (Lung, 1996) and (Day, 1993).

3.2.4 Scenario- based architecture-re-engineering (SBAR)

(Bengtsson, 1998) presented a scenario based method of the architecture re-engineering that focuses on multiple software qualities (reusability and maintainability). Various quality attribute research communities have proposed their own design systems (Karlsson, 1995). All these methods focus on a single quality attribute and treat all approaches unsatisfactory because a balance of various quality attributes is needed in the design of any realistic system of this method .The contribution of this method are the architecture design and the scenario based evaluation of the software qualities of a detailed architecture of a system (figure 5). A particularity of this method is that for assessing the architecture of the existing system, the system itself can be used. The goal of the evaluation method is to estimate the potential of the designed architecture to reach the software quality requirements.

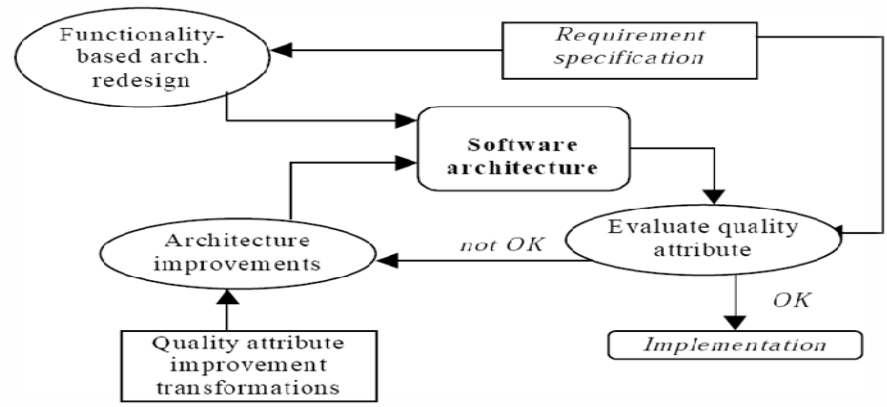

Figure 5:- Reengineering and architecture analysis by (Bengtsson, 1998).

- In SBAR, four different techniques for assessing quality attributes are identified: scenarios, simulation, mathematical modeling and experience-based reasoning.
- Scenarios: This technique is recommended for the quality attributes of the development, such as maintainability and reusability.
- Simulation: Simulation completes the scenario based approach, being useful or evaluating operational software qualities such as performance as fault tolerance.
- Mathematical modeling: Mathematical models allow a static evaluation of architectural design models. This technique is an alternative to simulation since both approaches are primarily suitable for assessing operational software qualities. To evaluate operational software qualities, the existent mathematical models or metrics developed by various research communities for high performance-computing (Barbacci, 1998), reliability (Runeson, 1995), and real-time systems (Liu,1999), could be used.
- Experience based reasoning: This approach is founded on experience and logical reasoning based on that experience. Experienced engineers often have valuable insights that may prove extremely helpful in avoiding bad design decision and finding issues that need further evaluations. Although these experiences generally are based on anecdotal evidence, a logical line of reasoning can justify most of them. This approach is different from the other approaches. Firstly, the evaluation process is less explicit and more based on

subjective factors such as intuition and experience. Secondly this technique makes use of the tacit knowledge of the involved persons. For each quality attribute, the evaluation, in this case the designer can select the most suitable approach.

Scenario based evaluation of a software quality consist of defining a representative set of scenarios, analyzing the architecture and summarizing the results (Graham, 2008). The selected scenarios concertize the actual meaning of the attribute is assessed by the analysis of the architecture in the context defined by each defined by each individual scenario for attribute is assessed by the analysis. Posing typical questions for the quality attributes can be helpful. The results from each analysis of the architecture and scenarios are then summarized into overall results, e.g. the number of acceptable scenarios versus the number of the not accepted ones.

The assessment process consists of defining a set of scenarios for each software quality, manually executing the scenarios for the architecture and subsequently interpreting the results. The assessment can be performed in a complete or statistical manner. In the first approach, a set of scenarios is defined: combined together, they cover the concrete instances of the software quality. If all scenarios are executed without problems, the quality attribute of the architecture can handle and scenarios that the architecture is optimal. The second approach is to define a set of scenarios that makes a representative sample without covering all possible cases. The ratio between scenarios that the architecture can handle and the scenarios not handled well by the architecture can handle provides an indication of how well the architecture provides an indication of how well the architecture fulfills the software quality requirements. Both approaches obviously have disadvantages. A disadvantage of the first approach is that generally impossible to define a complete set of scenarios. The definition of scenarios is. The definition of a representative set of scenarios is the weak point in the second approach, since it is unclear how one decides that a scenario set is representative.

3.2.5 Scenario-based Architecture Level Usability Analysis (SALUTA)

SALUTA is a specialized framework directed towards the assessment of usability quality attributes. SALUTA is the first method to assess usability before the implementation of a software architecture (Folmer, 2003). SALUTA is the first method to assess usability before the implementation of software architecture (Folmer, 2003).

SALUTA does not use any specific architectural view to describe software architecture. It extracts two types of information from the software architecture: (1) usability patterns, the design patterns used to solve a particular scenario and (2) usability properties, the architectural decisions that affect the usability attribute. Usability patterns and properties are identified by analyzing the software architecture, using functional design documentation, and interviewing software architect(s).

SALUTA divides usability into four sub-attributes: satisfaction, learn ability, efficiency and reliability (Graham, 2008). SALUTA elicits usage scenarios from usability requirement specifications. These scenarios represent different use of a system for different types of users and context of use. Using these usage scenarios, SALUTA creates usage profile that represents the required usability of the system. To create a usage profile, users, their tasks and context of use are identified for each usage scenario. The usability sub-attributes are quantified to express the required usability of the system for each usage scenario.

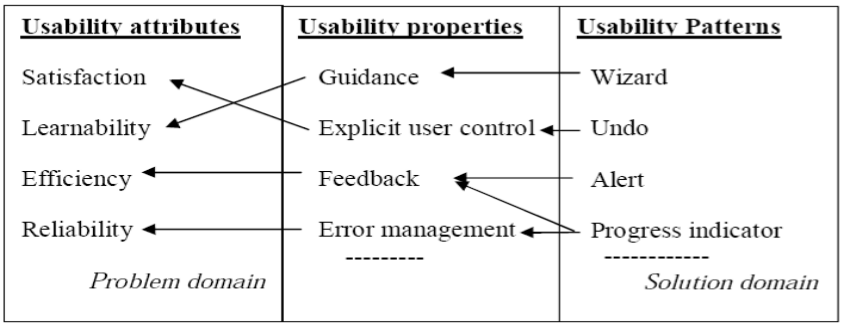

Figure 6:- Subset of relationship between usability patterns, properties and attributes (Folmer, 2003)

3.2.6 Architecture-Level Prediction of Software Maintenance (ALPSM)

The goal of ALPSM is to predict the maintenance effort required to address a change scenario (Bengtsson, 1998).The main contribution of this method consist of the architecture level where this prediction is performed. ALPSM defines a maintenance profile, like a set of change scenarios tasks. A scenario describes an action, or sequence of actions that might occur as related to the system. Hence a change of scenario describes a certain maintenance tasks. Using the maintenance profile, the architecture is evaluated using the scenario describes a certain maintenance effort for a software system can be estimated. The method has a number of inputs the requirements specifications, the design of the architecture, expertise from software engineers and possibly historical maintenance data. The method consists of the following six steps.

1. Identify categories of maintenance tasks: Formulate classes of expected changes based on application or program description.
2. Synthesize scenarios: For each of the maintenance tasks, a representative set of scenarios is defined.
3. Assign each scenario a weight: The scenarios are assigned a weight based on their probability of occurring during a particular time interval.
4. Estimate the size of elements: To be able to assess the size of changes, the size of all components of the system is determined. One of the three techniques can be used for estimating the size of components: Using estimation technique of choice, an adaption of an object oriented metric or, when historical data from similar applications or earlier releases is available, existing size data can be used and extrapolated to new.
5. Script the scenarios: for each scenario determines the components that are affected and to what extent they will be changed, this resulting in the size of the impact of the realization of the scenario.
6. Calculate the predicted maintenance effort: The total maintenance effort is predicted by summing the size of the impact of the scenarios multiplied by their probability.

4 Findings

We found that the ALPSM analyzes maintainability by looking at the impact of scenarios. It uses the size of changes as a predictor for the effort needed to adopt the system to a scenario. The ALPSM does not to address risk assessment thus the need to improve the model so as to incorporate the risk assessment aspect during software maintenances. Further we found that in software architecture analysis of change a number of these views is required. 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 implementing the change scenarios. We found that the views most useful for doing so are the views that shows the architectural approach taken for the system, i.e. the conceptual view, and the view that shows the way the system is structured in the development environment, i.e. the development view. For risk assessment of business information systems we need another type of additional information. In that case, we need to know about the system owners that are involved in a change (Lassing et al., 1999). This information is normally not included in the software architecture designs, so it has to be obtained separately from the stakeholders.

5 Method for Software Risk Assessment at the Architecture Level

The Architecture level prediction software maintenance method (ALPSM) does not provide mechanisms to address the risks that are associated with the maintenance changes; this is what this research sort to extend on.

The method for software maintenance risk assessment at the architecture level consists of the following steps.

1. Identify categories of maintenance tasks from the scenarios. Model the scenarios using UML specifications:
2. Synthesize scenarios: For each of the maintenance tasks, a representative set of scenarios will be defined.
3. Map the scenarios into the architectural design: For each scenario determines the components that are affected and to what extent they will be changed, this results in the size of the impact of the realization of the scenario
4. Map the Participating classes of the scenarios as presented in UML specifications model(s) to a published design pattern that best matches the Model.
5. Risk assessment: Establish that the impact of a change scenario; estimate the risk on the ripple effects of the changes (maintenance) to be made on a component in respect with the interacting components in order to predict the overall risk that might be associated to during the maintenance of a system

6 Conclusions

In this paper we presented experiences from applying the software architecture analysis methods in the previously described case studies. With respect to the first phase of the analysis methods, goal setting, we found that it is important to decide on the objective for the analysis. For the second phase of the method, architecture description, we established that the impact of a change scenario may span several architectural views. Furthermore, we found that views describing to the system's dynamics are not required in change analysis. But for some analysis goals we need information that is not included in existing architecture view models. For risk assessment, we establish the system's environment and information about system owners useful in evaluating change scenarios. The main contribution of this paper is

the extension of the Architecture level prediction software maintenance method (ALPSM) by (Bengtsson, 1998) to provide for risk assessment at the architecture level analysis during maintenance. The survey also highlighted a number of issues which existing methods do not sufficiently address. Only one method, ALPSM provides comprehensive process support for maintenance risk assessment. Finally, the five major activities of the Method for software Maintenance Risk assessment at the architecture level as established in this paper are:

- Identify categories of maintenance tasks from the scenarios. Model the scenarios using UML specifications:
- Synthesize scenarios: For each of the maintenance tasks, a representative set of scenarios will be defined.
- Map the scenarios into the architectural design: For each scenario determines the components that are affected and to what extent they will be changed, this results in the size of the impact of the realization of the scenario.
- Map the Participating classes of the scenarios as presented in UML specifications model(s) to a published design pattern that best matches the Model.
- Risk assessment: Establish that the impact of a change scenario; estimate the risk on the ripple effects of the changes (maintenance) to be made on a component in respect with the interacting components in order to predict the overall risk that might be associated to during the maintenance of a system

7 **Future Works**

It will be interesting to see how the method established in this research will be applied for risk assessment at the architecture level of a software project under maintenance.

8. References

1. Maryoly Ortega, Mara A. Perez, and Teresita Rojas. A systemic quality model for evaluating software products. Laboratorio de Investigacinenistemas de Informacin, 2002.
2. Shari Lawrence Pfleeger. Software Engineering Theory and practice. Prentice Hall, 2001.
3. Jonathan L. Krein. Design Patterns in Software Maintenance. Second International Workshop on Replication in Empirical Software Engineering Research. 2011 [Accessed on 12, May 2013]
4. Donald G. Firesmith. Common concepts underlying safety, security, and survivability engineering. Carnegie Mellon Software Engineering Institute - Technical Note CMU/SEI-2003-TN-033, December 2003.
5. International Standard. ISO/IEC 9126-1. Institute of Electrical and Electronics Engineers, Part 1,2,3: Quality model, 2001
6. Roger S. Pressman. Software Engineering a practitioner's Approach. McGraw-Hill, Inc., 2006.
7. C. D. Knutson, J. L. Krein, L. Prechelt, and N. Juristo, "1st international workshop on replication in empirical software engineering research (RESER)," in Proceedings of the International Conference on Software Engineering. New York, NY, USA: ACM, 2010, pp. 461–462.

8. Joc Sanders and Eugene Curran. Software Quality, A Framework for success in software Development and Support. Addison - Wesley Publishing Company, 1995.
9. Stephen H. Kan. Metrics and Models in Software Quality Engineering. Addison Wesley, 2003.
10. Center for Software Engineering. OO Analysis and Design: Modeling, Integration, Abstraction, Spring 2002.
11. J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation', in proceedings of 1999 IEEE Engineering of Computer Based Systems Symposium(ECBS99), Nashville, USA, March 1999
12. L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison Wesley,1998
13. L. Bass, P. Clements and R. K. Kazman. Software Architecture in Practice. SEI Series in Software Engineering. Addison-Wesley, 1998.
14. J. E. Gaffney. Metrics in software quality assurance. Proceedings of the ACM '81 conference, pages 126-130, March 1981.
15. Ronan Fitzpatrick. Software quality definitions and strategic issues. Staffordshire University,1996.
16. L. Buglione and Abran A. Geometrical and statistical foundations of a three-dimensional model of software performance. Advances in engineering software, 30:913{919, 1999.
17. Osman Balci. Credibility assessment of simulation results. Proceedings of the 18th conference on Winter simulation, pages 38-44, 1986.
18. Mettala, E., Graham, M. (eds.), "The Domain-Specific Software Architecture Program", CMU/SEI-92-SR-9, Software Engineering Institute, Carnegie Mellon University, 1992.
19. Parnas, D, "On the design and development of program families," IEEE Transactions on Software Engineering, SE-2(1), 1976, 1-9.
20. Gough, P., Fodemski, F., Higgins, S., Ray, S., "Scenarios - an Industrial Case Study and Hypermedia Enhancements", Proceedings of the Second IEEE International Symposium on Requirements Engineering, York, England, March, 1995, 10-17.
21. L.Dobrica&E.Niemela. A survey on software architecture analysis methods. IEEE Trans. On Software Engineering, Vol. 28, No. 7, pp.638-654, July 2002.
22. Kazman, R., Bass, L., Abowd, G., Webb, M., "SAAM: A Method for Analyzing the Properties of Software Architectures", Proceedings of ICSE 16, Sorrento, Italy, May 1994, 81-90.
23. G. Abowd. Analyzing Development Qualities at the Architecture Level: The Software Architecture Analysis Method. in: L. Bass, P. Clements, and R. Kazman (eds.). Software Architecture in Practice, Addison-Wesley 1998.
24. R.Kazman, G.Abowd, L.Bass&P. Clements. Scenario -Based Analysis of Software Architecture. IEEE Software, pp. 47-55, Nov. 1996.
25. Jacobson, "The Use-Case Construct in Object-Oriented Software Engineering," Scenario-Based Design: Envisioning Work and Technology in System Development, J. Carroll, ed.,John Wiley & Sons, New York, 1995, pp. 309-336.
26. M. Morisio, I. Stamelos, A. Tsoukias, A new method to evaluate software artifacts against predefined profiles, in: Proceeding of SSEKE'02, ACM, Italy, 2002.
27. T. Rosqvist, M. Koskela, H. Harju, Software quality evaluation based on expert judgement, Software Quality Journal 11 (2003) 39–55.
28. M. Lorenz, J. Kidd, Object-Oriented Software Metrics, Prentice Hall, 1994.
29. P. Bengtsson, J. Bosch. Architecture Level Prediction of Software Maintenance. In the Proceedings on 3rd European Conference on Software Maintenance and Reengineering, pp. 139-147, 1999.
30. E. Folmer and J. Bosch. Architecting for usability: a survey. Journal of systems and software, Elsevier, pp. 61-78, , 2002
31. Lehman, M.M., Perry, D.E. Ramil, J.F. Implications of evolution metrics on software maintenance. In Proceedings of the Int'l Conference on Software Maintenance (ICSM 1998), IEEE-CS Press, 208-219.

32. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A. Does code decay? Assessing the evidence from change management data. IEEE Trans. Software Engineering, 27, (January 2001), 1-12.
33. Sommerville I. Software Engineering. Addison-Wesley: Harlow, UK, 1996; 666–672.
34. Desharnais J, Pare F,MayaM, St-Pierre D. Implementing a measurement program in software maintenance: An experience report based on Basili's approach. Proceedings of the Conference of the International Function Points Users Group. International Function Points Users Group: Mequon WI, 1997; 143–152.
35. Martin J, Mcclure C. Software Maintenance: The Problem and its Solutions. Prentice-Hall: Englewood Cliffs NJ, 1983;
36. Pigoski TM. Practical Software Maintenance: Best Practices for Managing Your Software Investment. Wiley: New York NY, 1997; 32–33.
37. Low GC, Jeffery DR. Function points in the estimation and evaluation of the software process. IEEE Transactions on Software Engineering 1990; **16**(1):64–71.
38. NASA Technical Std. NASA-STD-8719.13A, Software Safety, 1997.
39. K. El Emam and W. Melo, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics," Technical Report NRC 43609, Nat'l Research Council Canada, Inst. for Information Technology, 1999.
40. S. Yacoub, H. Ammar, and T. Robinson, "Dynamic Metrics for Object-Oriented Designs," Proc. Sixth Int'l Symp. Software Metrics (Metrics '99), pp 50-61, 1999.
41. J. Rumbaugh, I. Jacobson, and G. Booch, the Unified Modeling Language Reference Manual. Addison-Wesley, 1999.
42. Booch G, Object-Oriented Analysis and Design with Applications (2nd edition), Benjamin/Cummings Publishing Company, 1994.
43. Jacobson I, M. Christerson, P. Jonsson, G. Övergaard, Object-oriented software engineering. A use case approach, Addison-Wesley, 1992.
44. Smith C, Performance Engineering of Software Systems, Addison-Wesley, 1990.
45. J.W.S. Liu, R. Ha, 'Efficient Methods of Validating Timing Constraints,' in Advanced in Real-Time Systems, S.H. Son (ed.), Prentice Hall, pp. 199-223,1995.
46. Gorton, L. Zhu. Tool Support for Just-in-Time Architecture Reconstruction and Evaluation: An Experience Report. International Conference on Software Engineering (ICSE) 2005, St Loius, USA, ACM Press.
47. R. Kazman, et al., "Experience with Performing Architecture Trade off Analysis," 21st Int. Conf. on Software Engineering Los Angeles, CA, USA, 1999.
48. J.-C. Laprie and K. Kanoun, "Handbook of Software Reliability and System Reliability," in Software Reliability Engineering, M. R. Lyu, Ed.: Computing McGraw-Hill, 1996, pp. 27-69.
49. R. de Lemos, "Idealised Fault Tolerant Architectural Element," Int. Conf. on Dependable Systems and Networks, Workshop on Architecting Dependable Systems, Philadelphia, PA, USA, 2006.
50. M. H. Klein, et al., "Attribute-Based Architecture Styles," 1st Working IFIP Conf. on Software Architecture, San Antonio, TX, USA, 1999.
51. E. Gamma, R. Helm, J. R., and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison- Wesley, Reading MA, 1995.