The goal of software architecture analysis: confidence building or risk assessment

Nico Lassing, Daan Rijsenbrij and Hans van Vliet
Faculty of Sciences
Vrije Universiteit, Amsterdam
{nlassing, daan, hans}@cs.vu.nl

Abstract

Software architecture analysis methods of flexibility tend to concentrate on averages. Based on the changes incurred by a number of scenarios, the flexibility of the system is assessed. This approach holds the danger that really complex scenarios, scenarios that impose real risks, are overlooked or their effect gets smoothened. We propose a method aimed at finding and assessing the really complicated scenarios. We provide a two-dimensional framework to structure the process of finding those complicated scenarios.

1. Introduction

In recent years, research into software architectures has become an important topic within the domain of software engineering. Analysis of software architectures is one of the main issues within this field. The aim of analyzing a system's software architecture is to predict the quality of that system before it has been built. By doing so, we hope to minimize the risk that, once built, the quality of the system is too low, or at least we hope to find out what the potential risks are.

One of the quality attributes for which such an analysis can be performed is flexibility. This attribute has to do with the ease with which a system can be adapted to changes. Potential risks relating to flexibility are that it is extremely difficult to adapt a system to certain changes or that it is even impossible to do so.

A number of authors have proposed methods for software architecture analysis of flexibility, the most important of which are Kazman *et al.* (1996) and Bengtsson and Bosch (1999). Both methods use scenarios to capture events that could happen in the life cycle of a system. Assessing the effect of these scenarios on the system then helps us judge the system's flexibility.

However, these methods suffer from some limitations concerning the identification of scenarios, as already mentioned in Lassing *et al.* (1999a). We conjecture that an analysis of flexibility should focus on finding those scenarios whose realization is especially complex, or even impossible. The previously mentioned methods provide little or no support for doing so. They emphasize averages, rather than extremes.

These different approaches to software architecture analysis resemble different approaches to testing. In random testing, system testing, etc., a large number of test cases that mimic typical usage scenarios, are executed. The goal of this process is to obtain confidence in the daily

operation of a software system. The intention of functional and structural testing methods, on the other hand, is to provoke failure behavior (see van Vliet (1993)). In a somewhat similar vein, the goal of current methods for software architecture analysis is to obtain confidence that the average maintenance costs will be reasonable. The goal of the method we advocate is to reveal risks.

Section 2 describes flexibility at the software architecture level. Section 3 discusses possible goals of software architecture analysis. Section 4 introduces a two-dimensional framework to classify scenarios, and explains how we can use this framework is software architecture analysis.

2. Flexibility at the software architecture level

More and more organizations turn to describing the software architecture of the information systems they are building. One of the most important reasons for doing so is that the software architecture represents the first design decisions for that system, with respect to the structure of the system, such as the division of the system in components, the relation between these components and the relation between the system and its environment. By explicitly laying down these decisions, we are able to analyze their appropriateness and correct them if necessary. We can distinguish two reasons why it is important to do so:

1. These decisions have a major impact on the quality of the resulting system
2. These decisions are very expensive to change at a later stage

Bass *et al.* (1998) claim that the software architecture inhibits or enables a system's quality attributes. If we focus on the quality attribute flexibility, this means that the decisions made in the software architecture have a large influence on the effort that is needed to adapt a system to changes. The most important reason for this is that the software architecture decides on the distribution of functionality of the system over its components, the relationships between these components and the relationship between the system and its environment. These decisions determine which components and relations have to be adapted as a result of certain changes and this in turn has a great impact on the effort needed to implement the changes. So, the software architecture has a great influence on the flexibility of a system.

However, there are a number of restrictions. First of all, making the 'right' decisions in the software architecture is not a guarantee for flexibility; other steps in the development process are important as well. For instance, suppose that the software architecture is such that just one component of the system has to be adapted because of a change but that the implementation of this component is hardly documented. It may then be very difficult to implement that change. On the other hand, there are alternative ways to achieve flexibility. For instance, some of the existing CASE-tools enable us to specify a system and then generate source code for different technical environments. Using these tools, adapting the system to changes in the technical environment is just a matter of regenerating the system with different parameters. So, although a system's software architecture influences its flexibility, other factors should not be ignored.

3. Confidence building versus risk assessment

A number of authors have proposed methods for software architecture analysis of flexibility (or modifiability or maintainability as some of them call it), such as Kazman *et al.* (1996) and Bengtsson and Bosch (1999). These methods use scenarios to analyze the flexibility of a system. A scenario is a potential event that can occur in the life cycle of a system. If the event occurs, the system has to be adapted. Assessing the effect of these scenarios provides insight into the flexibility of the system.

We can distinguish at least three approaches to software architecture analysis of flexibility. The first approach is to use software architecture analysis to predict the total maintenance effort that is needed in the life cycle of a system. This allows us to estimate the average cost of maintaining the system. Figure 1 shows a hypothetical example of the result of this approach.

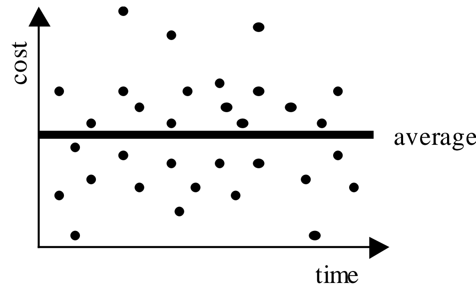

Figure 1: Estimating the average cost of maintenance

This approach is taken by Bengtsson and Bosch in their method, in which they use a cost-function of the following form (C_m: total cost of maintenance; $p(sc_i)$: probability of occurrence of scenario i; $C(sc_i)$: cost of implementing scenario i):

$$C_m = \sum_{i=0}^{n} p(sc_i) \cdot C(sc_i)$$

A possible extension to this approach is to include the variance in our analysis. We could, for instance, look for the '80%-interval' of the scenarios. This provides us insight in the variation of the cost of the various scenarios, or in other words how much the maintenance cost in a year is likely to deviate from the average cost. This is illustrated in Figure 2.

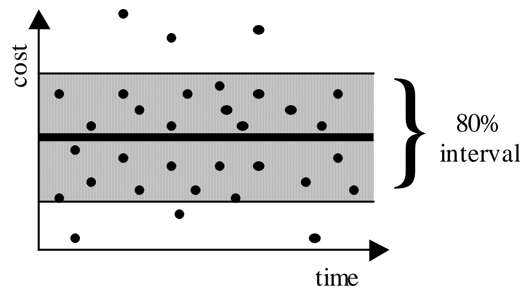

Figure 2: The variation

Neither of these methods tells us anything about the quality of the scenarios, or when to stop inventing new ones. It is not unthinkable that a lot of 'easy' scenarios are being developed, thus giving a false impression of the system's quality. The unconscious goal of the analysis process may be to convince oneself of the superior quality of this architecture. This danger may hold the more if the architects themselves are involved in the analysis. In software testing, it is sometimes stated that a test is only useful if it reveals a fault. Pursuing this analogy, we may state that a scenario is only useful if the changes it induces are difficult to accomplish. So, in our view the search for scenarios should be aimed at discovering the outliers shown in Figure 3.

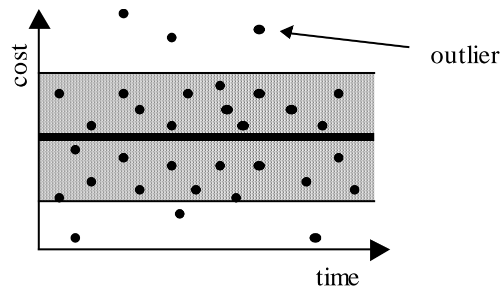

Figure 3: Discovering outliers

4. Towards a taxonomy of changes

Existing methods for software architecture analysis of flexibility have little support for finding outliers. Their approach is to look for scenarios that are likely to happen and then assess their impact. We propose an approach that is based on the opposite, namely to look for scenarios that are complicated and only then to judge whether these are likely to happen. This approach builds on the definition of a number of categories of scenarios of which we know that they are hard to implement. In Lassing *et al.* (1999b) we report on a number of factors which influence the complexity of changes associated with scenarios. Based on these factors we have defined the following categories of complex scenarios:

1. Scenarios that require adaptations to the system, which have external effects: when the changes necessary for these scenarios are applied to system, other systems have to be adapted as well.
2. Scenarios that require adaptations to the environment of the system, which in turn affect the system itself: in this case, the environment is the source of changes to which system has to be adapted.
3. Scenarios that require adaptations to the macro architecture: these scenarios require changes to the relationship between the system and other systems
4. Scenarios that require adaptations to the micro architecture: these scenarios require changes to the internal structure of the system
5. Scenarios that introduce version conflicts: these scenarios require changes to components that are shared by several systems, which leads to distinct versions of these components and then causes conflicts.

The main difference between scenarios of the first category and those of the second category is the initiator of the scenario. In the first case, the owner of the system initiates the scenarios and in second case, the scenarios are initiated by the owner of another system. This means that for scenarios of the first category, the owner of the system has to convince others of the importance of these scenarios. Scenarios of the second category, on the other hand, may be forced upon the owner of the system and the system has to be adapted, whether the owner likes it or not. So, the environment can be both a complicating factor in the implementation of scenarios, as in the first case, or a source of changes, as in the second case. This is one of the reasons why we have decided to view a system's software architecture on two levels: (1) the role of the system within the environment, which we call the macro architecture, and (2) the internal structure of the system, which we call the micro architecture.

This brings us to the third and fourth categories, which are the scenarios that require adaptations to the micro architecture or the macro architecture. Adaptations to the software architecture, either macro or micro, should be prevented as much as possible, because they are the

most radical type of changes. Not only do they affect the internals of a number of components, but they also affect the way in which these components collaborate.

The fifth and final category we have defined are the scenarios that introduce version conflicts. This problem may occur in a situation where components are shared by several systems. When a system then requires changes to one of these components, different versions of this component may be introduced. Problems may then arise when these versions conflict. This means that the corresponding scenario cannot be implemented as such.

The changes that are incurred by scenarios may come from a number of sources. We distinguish the following four sources of changes:

I. Functional requirements
II. Quality requirements
III. External components used
IV. Technical environment

The first source of changes is the set of functional requirements. Examples of changes in the functional requirements are features that have to be added or unwanted functionality that has to be deleted. The second source of changes is the set of quality requirements. Changes that can occur in the quality requirements are, for instance, the need for increased performance or the need for increased security. The third source of changes is the set of external components used. When these components change, the system may have to be adapted. This situation often occurs when a system makes use of components of another system, or when generic components are shared by a number of systems. The main problem is that these components are often owned by others, which means that the owner of the system concerned does not have full control over them. As a result, these changes are sometimes forced upon the system and its owner. Something similar applies to the fourth source of changes, namely the technical environment. In more and more organizations the technical environment is shared by several systems. So, just like the external components it could happen that the system has to be adapted to changes that are initiated by others than the owner of the system concerned. This is something that has to be taken into account when analyzing a system's (in)flexibility.

We can combine the categories of complex scenarios with these sources of changes. This leads to the following two-dimensional framework that may help us discover complicated scenarios:

	Source I: Changes in the functional requirements	Source II: Changes in the quality requirements	Source III: Changes in the external components	Source IV: Changes in the technical environment
Category 1: Adaptations to the system with external effects				
Category 2: Adaptations to the environment with effects on the system				
Category 3: Adaptations to the macro architecture				
Category 4: Adaptations to the micro architecture				
Category 5: Introduction of version conflicts				

The best way to use this diagram is to start by identifying scenarios by reading documentation and interviewing stakeholders. This leads to an initial set of scenarios. The next step then is to classify these scenarios using the above-mentioned framework. This classification provides insight into the completeness of our initial set of scenarios. Particularly the empty cells in the framework deserve our attention. They may indicate that we have missed scenarios, or that scenarios for that cell just do not exist. In one of the case studies we conducted, for instance, we did not find any scenarios that had external effects. It appeared that this was not because we missed a scenario, but because the system was used by no other system. So, the framework is most useful as an aid in the identification of scenarios.

5. Conclusion

We started this paper with the proposition that we can distinguish two approaches to software architecture analysis of flexibility, namely confidence building and risk assessment. Existing methods of software architecture analysis of flexibility focus mainly on confidence building, rather than on risk assessment. Their aim is to find a set of scenarios that is representative for the events that can be expected in the life cycle of a system. In our view, this holds the danger that really complex scenarios are overlooked. We propose an alternative method, whose aim is to discover complex scenarios that impose real risks. One of the main parts of this method is a two-dimensional framework to assess the completeness of the set of identified scenarios. The first dimension in this framework consists of five categories of complex scenarios and the second dimension consists of four sources of changes. Classifying the scenarios we have found in one of the cells of this framework shows in which categories we might have missed one or more scenarios. Alternatively, it might be that those categories are irrelevant for the system analyzed.

In further research, we will focus on the completeness of the set of complex scenarios and on the relative importance of each of the cells in the framework.

Acknowledgements

This research is mainly financed by Cap Gemini Netherlands.

References

L. Bass, P. Clement and R. Kazman (1998). *Software Architecture in Practice*. Addison Wesley, Reading, USA.

P.O. Bengtsson and J. Bosch (1999). Architecture Level Prediction of Software Maintenance. *Proceedings of the International Conference on Software Engineering '99*.

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

N.H. Lassing, D.B.B. Rijsenbrij and J.C. van Vliet (1999a). On Software Architecture Analysis of Flexibility. Complexity of Changes: Size isn't everything. *Proceedings of the Second Nordic Workshop on Software Architecture*.

N.H. Lassing, D.B.B. Rijsenbrij and J.C. van Vliet (1999b). Towards a broader view on software architecture analysis. *Proceedings of the Sixth Asia-Pacific Software Engineering Conference '99*.

J.C. van Vliet (1993). *Software engineering: principles and practice*. John Wiley & Sons Ltd., Chichester.