On Software Architecture Analysis of Flexibility

Complexity of Changes: Size isn't Everything

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

Abstract

The decisions made in a system's software architecture have a major impact on the quality of the resulting system. The goal of software architecture analysis is to assess whether these decisions lead to the required quality. Our research concentrates on software architecture analysis of administrative systems. In this paper, we focus on the quality attribute flexibility, because it is considered to be an important property for this class of systems. In our opinion, existing methods are not sufficient for analyzing this quality attribute within this domain, because they offer little support for identifying scenarios that are really hard to realize. In addition, they do not take into account all factors that cause complexity of changes for systems within this domain. Therefore, we have defined an alternative method that does, which is presented in this paper.

1. Introduction

Recently, software architecture has become one of the main topics of interest within the study of software engineering. Software architecture represents the manifestation of the earliest design decisions about the structure of a system (Bass *et al.* (1998)). These decisions have a major impact on the quality of the system: they inhibit or enable some of a system's quality attributes. However, not all quality attributes are primarily determined by the software architecture, some are more dependent on the implementation of the system. Usability and correctness are examples of such attributes. The quality attributes that are most likely to be determined by the software architecture are those that relate to the structure of the system, such as performance and flexibility. To achieve these quality attributes it is important to investigate the appropriateness of the design decisions relating to the structure as soon as possible, the more so since these decisions are generally very expensive to correct at a later stage. This is one of the motives of software architecture analysis, namely to judge a system's quality attributes based on nothing more than the software architecture.

Our research focuses on the software architecture analysis of flexibility within the domain of administrative information systems. In previous research (Lassing *et al.* (1999) and Lassing *et al.* (To be published)), we have found that existing models for architecture analysis are not directly usable for systems of this type. We have defined an alternative method for doing so, which is presented in this paper.

This paper has the following structure. Section 2 defines flexibility for information systems and explores how flexibility could be analyzed at the software architecture level. In section 3 we review existing methods for software architecture analysis of flexibility. In section 4, we discuss their limitations and introduce an alternative method. Section 5 contains some concluding remarks.

2. Software architecture analysis of flexibility

Our research focuses on flexibility of administrative information systems. Flexibility is generally regarded an important quality for modern administrative information systems, although it is not always clear what it exactly is. In our opinion, flexibility has to do with the effort needed to adapt a system to changes either in the requirements or in the environment. The less effort is needed for such adaptations, the greater the flexibility of the system. To achieve the desired level of flexibility it is important to analyze the flexibility of a system at an early stage, the more so since the earliest design decisions, which are captured in the software architecture, have a major impact on the flexibility of a system. So, ideally, we would like to analyze the system's flexibility based on the software architecture. However, to do so, we encounter a number of difficulties.

The first difficulty is related to flexibility in general, namely that there is no absolute measure to express a system's flexibility. A system will never be flexible with respect to all changes; it's just that some changes are easier to implement than others. However, we can estimate the flexibility of a system by looking at a number of changes and assessing how much effort is required to implement them. These changes then make flexibility tangible. This is also how we can analyze the flexibility of a system, namely by looking at a number of possible changes and estimating their impact on the system. The value of such an analysis depends on our ability to predict the changes that could happen in the life of a system. We can make an educated guess of these changes using scenarios. These scenarios represent events that could happen in the life of a system. We stress that such an analysis does not lead to an absolute measure of the system's flexibility; its judgements are merely valid within the specified context.

Another difficulty, which is related to software architecture analysis in general, is that the implementation of the system is not and cannot be considered, because it is unknown at the software architecture level. Although we acknowledge that the implementation will have some influence on the flexibility of the system, we claim that the structural aspects of the system are of greater importance for flexibility. However, this means that the results of a software architecture analysis should be interpreted with care. The analysis should be used to gain insight into the flexibility of the resulting system, not for making absolute statements.

3. Existing methods

A number of authors have demonstrated methods for software architecture analysis of flexibility, the most prominent of which are Kazman *et al.* (1996) and Bengtsson and Bosch (1999). Both methods are based on scenarios, meaning that the flexibility of systems is assessed by simulating changes to their requirements and their environment. Although their basic principle is the same, they also have a number of differences.

The Software Architecture Analysis Method (or SAAM), discussed in Kazman *et al.* (1996), is presented as a general method for analyzing various quality attributes at the software architecture level, one of which is flexibility. It can be used both for analyzing the flexibility of a single software architecture (as for example in Kazman *et al.* (1996)) and for comparing the flexibility of a number of software architectures (as for example in Bass *et al.* (1998)). In case of an analysis of a single software architecture the goal is to determine whether anticipated or expected changes are difficult to implement. In case of an analysis of several systems, the goal is to compare software architectures and choose the most adequate one. SAAM consists of the following steps:

1. Develop scenarios: identify possible events that may happen in the life of the system.
2. Describe candidate architecture(s): give a common representation of the candidate architecture(s).
3. Classify scenarios: determine whether a scenario requires modifications to the system. Scenarios that require no modifications are called *direct* and scenarios that do require modifications are called *indirect*.
4. Perform scenario evaluations: determine for each indirect scenario the cost of the modifications associated with it, by listing the components and connectors that are affected.
5. Reveal scenario interaction: determine which scenarios *interact*, meaning that they affect the same component. Interaction of unrelated scenarios could indicate a poor separation of functionality.
6. Overall evaluation: by assigning weights to the scenarios and scenario interactions in terms of their relative importance, we are able to come to an overall evaluation of the candidate software architecture(s).

Summarizing, it may be said that SAAM analyzes flexibility by estimating the effort needed for implementing scenarios. To this end, the number of components and connectors that is affected by a scenario is used as a predictor for the effort that is needed to adapt the system. In addition, sub-optimal separation of functionality is detected using scenario interaction.

The method proposed by Bengtsson and Bosch (1999) is aimed at estimating the total maintenance effort for a single system, based on its software architecture. Their notion of maintainability closely corresponds to our notion of flexibility. This method uses the following steps:

1. Identify categories of maintenance tasks: formulate classes of expected changes.
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.
4. Estimate the size of all elements: to be able to assess the size of changes, the size of all components of the system is determined.
5. Script the scenarios: for each scenario, determine the components that are affected and to which extent, resulting in the size of the impact 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.

In short, this method analyzes flexibility by looking at the impact of scenarios. It uses the size of changes as a predictor for the effort needed to adapt a system to a scenario.

4. An alternative method

We claim that the methods presented in the previous section are not sufficient for software architecture analysis within the domain of administrative systems, for two reasons. First of all, we think that the goal of analysis of flexibility should be to expose boundaries in the software architecture with respect to flexibility. This means that an analysis 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, both with respect to evaluation and the scenario identification. And secondly, they assume that the complexity of a scenario is totally determined within the system, by the number of components that is affected and their size. In Lassing *et al.* (To be published), we have noted that, for administrative systems, the complexity of a scenario is not primarily caused by the amount of source code of the system that has to be adapted, but also by a number of other factors.

To handle these drawbacks, we have developed an alternative method for software architecture analysis. The main differences between this method and the methods presented in the previous section are in the way in which we arrive at the scenarios and in the way in which we evaluate their impact. Our method consists of the following steps:

1. Describe software architecture
2. Identify relevant scenarios
3. Evaluate the effect of scenarios

Although the steps are listed as though they are performed sequentially, they are not. In order to describe the relevant aspects of the software architecture it is important to have a perception of the scenarios that are evaluated. And on the other hand, without a general idea of the software architecture it is impossible to define relevant scenarios. So, the first two steps have to be performed in parallel. In addition, the third step may reveal that not all relevant aspects of the software architecture have been included in the description of step 1.

The goal of step 2 should be to find scenarios that are complex to realize. As an aid to do so we have defined a number of classes of scenarios that are possibly complex to realize. These classes were defined given the factors that influence the complexity of scenarios. These factors are included in our measurement instrument which is used in step 3 to evaluate the complexity of scenarios. We will first introduce this measurement instrument and then return to the classes of complex scenarios.

The goal of the measurement instrument is to provide insight into the complexity of scenarios for administrative systems. To this end, it includes a number of factors that are based on the findings of previous research (Lassing *et al.* (1999) and Lassing *et al.* (To be published)).

First of all, we found that systems within this domain are not isolated. They increasingly become integrated with one another. As a result, the system's environment becomes more and more important. The environment is not only a source of changes, but it could also put up barriers for changes to a system. To expose these difficulties, we have decided to divide the description of the software architecture into two parts: (1) the role of the system within the environment, which we will call the macro architecture, and (2) the internal structure of the system, which we will call the micro architecture. Similarly, we will divide the effect of a scenario into two parts: (1) the effect on the system, and (2) the effect on the environment.

The first factor that we use to express the complexity of a scenario is the impact on the software architecture, divided into the effect on the macro architecture level and the effect on the micro architecture level. At both levels, we recognize four possible levels of impact:

1. Scenario has no impact
2. Scenario affects one component
3. Scenario affects several components
4. Scenario affects software architecture

At the micro architecture level, the components of the levels 2 and 3 represent subsystems of the system and, at the macro architecture level, they represent systems in the environment. This scale helps us distinguish between scenarios that have locality of change, i.e. scenarios for which only a single component has to be adapted, scenarios that do not have locality of change, i.e. scenarios for which several components have to be adapted, and scenarios that do not fit within the current software architecture. We stress that we do not consider the size of the components and extent to which they are affected. The reason for this is that we think that the complexity of changes is hardly caused by the number of lines of code that has to be adapted.

Another notion that is becoming increasingly important for administrative systems is ownership. Under the influence of increasing integration and the use of off-the-shelf components (commercial and otherwise), the number of owners involved in an information system tends to increase. In our research we found that scenarios affecting components that belong to different owners are inherently more complex to realize than those whose impact is limited to the components of a single owner, mainly because of the additional need for coordination between the various parties. Therefore, we have included this factor in our measurement instrument.

The final factor that is included in our measurement instrument is whether a scenario leads to different versions of some architectural element. The presence of different versions may introduce a number of difficulties. Ultimately, they may result in changes to architectural elements that we initially unaffected by a scenario. We have distinguished the following four levels of difficulties related to versions:

1. No problem with different versions
2. The presence of different versions is undesirable, but not prohibitive
3. The presence of different versions creates complications related to configuration management
4. The presence of different versions creates conflicts

So, now we have three factors to express the effect of a scenario: (1) the level of impact, (2) the need for coordination between different owners and (3) the presence of version conflicts. Applying this instrument in a software architecture analysis leads to results like Table 1.

Table 1: Example of the measurement instrument

		Macro architecture level			**Micro architecture level**		
	Initiator of scenario	Impact level	Multiple owners	Version conflict	Impact level	Multiple owners	Version conflict
Scenario 1	Unit A	3	+	3	1	-	3
Scenario 2	Unit B	3	+	4	2	+	1
...							
Scenario n	Unit X	1	-	1	4	-	1

Table 1 also includes the initiator of a scenario, which is the organizational unit that has most interest in the implementation of a scenario. This fact is included because it helps us to determine the feasibility of a scenario. A scenario that is initiated by one unit and affects architectural elements of another may not be feasible, because the other unit may not have an interest in implementing the required changes. So, the initiator also plays a part in the complexity of the changes related to a scenario.

Based on the factors defined above, we may formulate a number of classes of scenarios that are particularly complicated to implement. In Lassing *et al.* (To be published), we already found the following:

1. *Scenarios initiated outside the organizational unit owning the system under analysis that do affect the system under analysis*: This class of scenarios causes changes that the system under analysis possibly has to follow, although they may not be immediately beneficial to the organizational unit owning the system.
2. *Scenarios initiated by the organizational unit owning the system under analysis that affect architectural elements belonging to other owners*: This class of scenarios causes changes for which the organizational unit owning the system under analysis needs to consult with others.
3. *Scenarios that affect the micro architecture*: This class of scenarios necessitates changes to the way in which the system under analysis is structured.

We can probably add a number of other classes of complex scenarios, such as the following:

1. *Scenarios that affect the macro architecture*: This class of scenarios forces changes to the way in which the systems work together.
2. *Scenarios that cause versions conflicts*: This class of scenarios may result in side effects on systems or subsystems that were initially unaffected.

This list is not yet complete, further research should be used to formulate a number of other classes. Eventually, this list of classes might help the analyst define scenarios for the system under analysis. He/she should consider these classes and determine whether scenarios exist that have such an effect on the system under analysis. We believe that this increases the chance that complex scenarios are identified.

The principal shortcoming of our approach is that our measurement instrument includes a number of factors that are not entirely comparable. Therefore, it is not always possible to compare the results of different scenario. For that reason, we emphasize that the results should be interpreted with care. The method is most useful as an aid in the analysis of flexibility.

5. Conclusion

This paper is on software architecture analysis of flexibility of administrative systems. We have discussed two existing methods for doing so, the Software Architecture Analysis Method and the method by Bengtsson and Bosch. We have indicated that these methods are not sufficient for analysis of flexibility of administrative systems, due to two major limitations. Firstly, they provide little support for finding scenarios that are complex to realize. We think that a method for software architecture analysis of flexibility should do so, because these scenarios are the ones that we want to pay attention to. We may even want to prevent a software architecture in which their realization is hard or impossible. Secondly, the existing methods assume that the complexity of changes is totally determined within the system. They only consider the number of components and the extent to which these components are affected of importance for the complexity of changes. We have argued that other factors might be more important.

To handle these drawbacks, we have defined an alternative method. This method provides support for finding complicated scenarios, by defining a number of generic classes of possibly complicated scenarios. The analyst can use these classes to define scenarios for the system under analysis. In addition, the method includes a measurement instrument that helps us to gain insight into the complexity of scenarios. In the measurement instrument, the effect of a scenario is divided into the effect on the environment of the system (the ‘macro architecture’) and the effect on the system itself (the ‘micro architecture’). To express the effect of the scenario at both the macro architecture and the micro architecture level, we have distinguished a number of factors: (1) the impact on systems and subsystems, (2) whether multiple owners are involved in the required changes and (3) the occurrence of version conflicts. Further research should not only focus on

finding more classes of complex scenarios, but also at verifying that the factors that we have distinguished are the ones that cause the complexity of scenarios.

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 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 (1999). Flexibility of the ComBAD architecture. In: P. Donohoe (ed.). *Software architecture: Proceedings of the First Working IFIP Conference on Software Architecture*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

N.H. Lassing, D.B.B. Rijsenbrij and J.C. van Vliet (To be published). Towards a broader view on software architecture analysis.