How Well can we Predict Changes at Architecture Design Time?*

Nico Lassing (a), Daan Rijsenbrij (b) and Hans van Vliet (c)†
(a) Accenture, Amsterdam, The Netherlands
(b) Cap Gemini Ernst & Young, Utrecht, The Netherlands
(c) Faculty of Sciences, Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Two years ago, we analyzed the architecture of Sagitta 2000/SD, a large business information system being developed on behalf of Dutch Customs. We were in particular interested in assessing the capabilities of the system to accommodate future complex changes. We asked stakeholders to bring forward possible changes to the system, and next investigated how these changes would affect the software architecture. Since then, the system has been implemented and used, and actual modifications have been proposed and realized. We studied all 117 change requests submitted since our initial analysis. The present paper addresses how well we have been able to predict complex changes during our initial analysis, and how and to what extent the process to elicit and assess the impact of such changes might be improved. This study suggests that architecture analysis can be improved if we explicitly challenge the initial requirements. The study also hints at some fundamental limitations of this type of analysis: (1) fundamental modifiability-related decisions need not be visible in the documentation available, (2) the actual evolution of a system remains, to a large extent, unpredictable and (3) some changes concern complex components, and this complexity might not be known at the architecture level, and/or be unavoidable.

Keywords: software evolution, risk assessment, impact analysis, software architecture analysis, modifiability

1 Introduction

Many organizations today operate in a world in which change is ubiquitous. Business developments and new technology force them to adapt their information systems regularly. Studies have shown that a large part of the costs associated with an information system is spent on its evolution (Nosek and Palvia, 1990). In order to provide for evolutionary capabilities of a system, it helps to be able to predict changes, and to assess their impact early in the development process.

In 1999, we analyzed the modifiability of Sagitta 2000/SD, a large business information system being developed on behalf of Dutch Customs. We asked stakeholders to bring forward possible changes to the system, and next investigated how these changes would affect the software architecture. Since then, the system has been implemented and used, and actual modifications have been proposed and realized. Two years later, we revisited the Dutch Tax and Customs Administration, collected information on the actual evolution of the system, and compared this with the initial set of anticipated changes.

This study is part of our research into a method for Architecture-Level Modifiability Analysis (ALMA) (Bengtsson et al., 2000). Like many other methods for software architecture analysis, such as SAAM (Kazman et al., 1996) and ATAM (Kazman et al., 1998), our method is scenario-based, which means that it uses concrete scenarios to assess the quality of a system based on its software architecture. At some point

*This research was done while the authors were at the Vrije Universiteit

†Corresponding author. Tel (31) 20 444 7768; fax (31) 20 444 7728
E-mail addresses: nico.lassing@accenture.com, daan.rijsenbrij@capgemini.nl, hans@cs.vu.nl

in the analysis, these scenarios are elicited from the stakeholders, and the software architecture is analyzed to determine their impact. The scenarios describe future uses of the system, uses not yet accounted for in the requirements. We term these *change scenarios*, as opposed to operational scenarios which do refer to the requirements.

The change scenario elicitation process is tricky. The type of change scenarios we are interested in depends on the goal of our analysis. If our goal is to predict the future maintenance cost of the system, we are interested in identifying change scenarios that are likely to occur during the operational life of the system. If our goal is to identify the risks of the architectural choices made, we are interested in scenarios which are particularly difficult to accomplish. How then do we know we have elicited the right scenarios? How do we know we have identified enough scenarios? And, since the system we are investigating hasn't been implemented yet, how do we determine the impact of these changes on the system?

In our research, we are interested in identifying architectural risks. We are thus interested in *complex* change scenarios. To help structuring the scenario elicitation process, we use a framework with categories of change scenarios which we have found to be complex at the architecture level. This framework is used to classify change scenarios as well as to probe stakeholders to formulate change scenarios. In this paper we discuss the validity of this framework. The aim of this validation is to gain insight in the predictive value of the results delivered by our analysis. This helps us understand the possibilities and limitations of architecture-level modifiability analysis, and offers handles for improvement. The validation also gives general insight in the limitations of architecture-level modifiability analysis. To the best of our knowledge, this is the first longitudinal study addressing this.

The remainder of the paper is divided into seven sections. Section 2 addresses the role of scenarios in software architecture analysis. In section 3 we give an overview of ALMA, our generic method for Architecture-Level Modifiability Analysis. In section 4 we give a brief introduction to Sagitta 2000/SD and its software architecture. Section 5 discusses the change request data used as input to this validation study. Section 6 discusses how well we were able to *predict* complex changes, while section 7 discusses how well we were able to predict *complex* changes. Section 8 contains our conclusions.

2 Scenarios in Software Architecture Analysis

Different techniques can be used for software architecture analysis. In (Abowd et al., 1997), a distinction is made between questioning techniques, which are qualitative in nature, and measurement techniques, which provide quantitative information. The first group includes scenarios, questionnaires and checklists. The latter group includes metrics and simulations. Most methods for software architecture analysis use scenarios, including the Software Architecture Analysis Method (SAAM) (Kazman et al., 1996), the Architecture Tradeoff Analysis Method (ATAM) (Kazman et al., 1998), the methods discussed in (Bengtsson and Bosch, 1999) and (Lassing et al., 1999), and ALMA, the Architecture-Level Modifiability Analysis method presented in (Bengtsson et al., 2000).

One of the reasons for using scenarios in architecture analysis is that they are very flexible; scenarios can be used for evaluating almost any quality attribute. For instance, we can use scenarios that represent requirement changes to analyze modifiability, scenarios that represent threats to analyze security, or scenarios that represent failures to analyze reliability. In addition, scenarios are usually very concrete, enabling us to make detailed statements about their effect.

Software architecture analysis is not the only area of software development in which scenarios are used; a number of other disciplines have adopted scenarios as well. In (Antón and Potts, 1998), an overview is given of the use of scenarios in software development. They distinguish two ways in which scenarios can be used in software development. The first is to use scenarios to represent the operations of some proposed system. The other is to use scenarios to describe and envisage future uses of the system. Scenarios of the former type are called operational scenarios and scenarios of the latter type evolutionary scenarios.

Most disciplines use operational scenarios. In object-oriented analysis and design (Jacobson et al., 1993), for instance, operational scenarios are used to demonstrate the interaction between software components in response to certain stimuli. In requirements engineering ((Sutcliffe et al., 1998), (Filippidou, 1998), (Weidenhaupt et al., 1998)), operational scenarios are used to capture operations of the systems that can be discussed with the system's users. And in human-computer interaction (HCI) (Carroll and Rosson,

1992), operational scenarios are used to represent the interactions that users may have with the system when performing certain tasks.

However, evolutionary scenarios are also used. In (Ecklund et al., 1996), for instance, evolutionary scenarios are used to record future requirements on a system. They call these *change cases*.

For software architecture analysis both operational and evolutionary scenarios are used. Operational scenarios are used by the analyst to demonstrate how the software architecture satisfies its requirements. They are called direct scenarios in SAAM and use case scenarios in ATAM. Evolutionary scenarios, on the other hand, are used to explore how well an architecture is suited for future uses of the system. SAAM calls these indirect scenarios, and ATAM calls them growth scenarios or exploratory scenarios. ALMA only uses evolutionary scenarios. In this paper, we refer to them as *change scenarios*.

Change scenarios build on the basic principle introduced in (Parnas, 1972). To evaluate the modularization of a system, Parnas describes circumstances under which the system's design decisions may change and explores how these circumstances influence the system. The description of such a circumstance is a change scenario. Change scenarios thus do not concern the requirements for the *first* version of the system. Rather, they concern the requirements for *subsequent* versions.

In software architecture analysis, scenarios are usually represented much less elaborately than in requirements engineering. Our change scenarios are not described in narrative stories with episodes, goals, roles, etc. Rather, scenarios in software architecture analysis tend to be one-sentence descriptions, closer to vignettes (Kazman et al., 1996). The amount of detail given should be sufficient to enable the analyst to determine its impact on the architecture under investigation. A number of examples of the use of this type of change scenario in software architecture analysis is given in subsequent sections.

3 Architecture-Level Modifiability Analysis (ALMA)

The generic method for modifiability assessment of software architecture that we advocate is based on the Software Architecture Analysis Method (SAAM) (Kazman et al., 1996). Like SAAM, ALMA is scenario-based, which means that we use concrete changes to assess the modifiability of a system, based on its software architecture. The main difference between ALMA and SAAM is that SAAM does not explicitly address the need for differences in the techniques depending on the goal of the analysis. In ALMA, this is an essential part, and the need to clearly identify the goal of the analysis, and to only have one goal for each analysis, is emphasized.

ALMA consists of five steps. These steps are not always performed in the indicated sequence, it is often necessary to iterate over the various steps. The steps are:

1. Set goal: determine the aim of the analysis
2. Describe software architecture: give a description of the relevant parts of the software architecture
3. Elicit change scenarios: find the set of relevant change scenarios
4. Evaluate change scenarios: determine the effect of the set of change scenarios
5. Interpret the results: draw conclusions from the analysis results

We give a brief overview of the steps of ALMA. A more elaborate discussion of the full method is given in (Bengtsson et al., 2000).

3.1 Goal setting

The first step to take in software architecture analysis of modifiability is to set the analysis goal. The goal determines the type of results that will be delivered by the analysis. In addition, the goal influences the choice of techniques to be used in subsequent steps. Different goals ask for different techniques. With respect to modifiability, the following goals can be pursued:

- **Risk assessment**: finding types of changes for which the system is inflexible. We are then interested in scenarios which are particularly difficult to accomplish. In terms of effort and probability, we are then interested in the outliers.

  In software testing, it is sometimes stated that a test is only useful if it reveals a fault in the software. Pursuing this analogy, we may state that, in the risk-oriented approach to software architecture analysis, a change scenario is only useful if it exposes a risk, i.e. the changes it induces are difficult to accomplish. This approach presupposes that we are able to estimate the probability that a given change scenario will occur. So, for risk assessment we look for a set of change scenarios that complies with the operational *risk* profile of the system.

- **Maintenance cost prediction**: estimating the cost of (adaptive) maintenance effort for the system in a given period. In a very general sense, we then use a maintenance cost function $C_{average}$ (the average cost per change scenario) of the form

$$C_{average} = \frac{\sum_{i=0}^{n} C(\textit{change}_i) \times p(\textit{change}_i)}{n}$$

  where $C(\textit{change}_i)$ denotes the effort or cost required to realize the i-th change scenario, and $p(\textit{change}_i)$ denotes the probability this scenario will occur.

  This approach requires cost estimates associated with each change scenario. At the global level we are dealing with, where only large-grained components and their interfaces have been identified, the accuracy of these estimates, and the predictions made on the basis thereof, need to be considered carefully. This approach also presupposes that we are able to estimate the probability that a given change scenario will occur. If no such information is available, we assume a uniform distribution. In either case, we are interested in identifying those change scenarios that are likely to occur during the operational life of the system. I.e. we look for a set of change scenarios that matches the operational *change* profile of the system.

- **Software architecture comparison**: comparing two or more candidate software architectures to find the most appropriate one. The difference with the two aforementioned goals is that in comparison we make relative statements about a number of candidate software architectures, while with the other goals we make absolute statements about a single candidate.

  In comparison we are interested in finding the differences between the two software architectures, so the scenario elicitation should be aimed at finding change scenarios that are treated differently by the candidates.

3.2 Software architecture description

After the goal of the analysis is set, the next step is to create a description of the software architecture. To do so, two sources of information are useful. The architecture designs used within the development team are an important source. In addition, we may interview the architect(s) of the system for additional information.

The software architecture description has a dual role in the analysis. It should be detailed enough to enable us to do an architecture level impact analysis for the set of change scenarios, i.e. assess their effect based on the software architecture. Conversely, to be able to assess the modifiability of an architecture, its description should make the major decisions that affect modifiability explicit. In our experience, software architecture descriptions that are available at the start of the analysis often do not meet these requirements. Part of the discussion with the architect then is focused on making the modifiability-affecting decisions explicit, i.e. creating the appropriate architectural description.

A software architecture description usually consists of a number of views, where a view is a representation of a system from a certain perspective. As we have argued in (Lassing et al., 2001), several architecture views are required for architecture-level modifiability analysis. At the software architecture level modifiability has to do with separation of functionality and dependencies, i.e. *how do we distribute the functionality over components?* and *how are these components related?* Allocation of functionality determines which components have to be adapted to realize certain changes and dependencies determine how

changes to a component affect other components. In an architectural description in which modifiability is addressed these decisions should be made explicit.

The aforementioned questions focus on the system's internals. For business information systems it is not sufficient to study only the internals of the system. Such systems are rarely isolated; they are often part of a larger suite of systems. At the system's level questions similar to the ones at the component level recur: *how do we distribute functionality over systems?* and *what are the dependencies between these systems?*. Therefore, we split the description of a system's software architecture into two parts: (1) the *external architecture*: the software architecture at the systems level, and (2) the *internal architecture*: the software architecture of the internals of the system.

3.3 Scenario elicitation

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

3.4 Scenario evaluation

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

3.5 Interpretation

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

If the goal of the analysis is risk assessment the results of the scenario evaluation are investigated to determine which change scenarios pose risks, i.e. for which scenarios the product of probability and costs is too high. The criteria for determining which values are still acceptable should be based on managerial decisions by the owner of the system. When risks are found, various risk mitigation strategies are possible: avoidance (take measures to avoid that the scenario will occur or take action to limit their effect, for instance, by use of code-generation tools), transfer (e.g. choose another software architecture) and acceptance (accept the risks).

If the goal of the analysis is to do maintenance prediction, the aim of this step is to come to an estimate of the amount of effort that is required for maintenance activities in the first period after the system is made operational. Again, the owner of the system should decide whether this estimate is acceptable. If not, one may decide to choose another architecture, renegotiate quality and/or functional requirements, etc. If the goal of the analysis is architecture comparison, we compare the results of the evaluation of the two sets of scenarios and choose the most appropriate candidate architecture. Again, it is important that the likelihood of the scenarios is considered in this selection process.

4 Sagitta 2000/SD

Sagitta 2000/SD is a large information system that is being developed by the Tax and Customs Computer and Software Centre of the Dutch Tax and Customs Administration on behalf of Dutch Customs. The

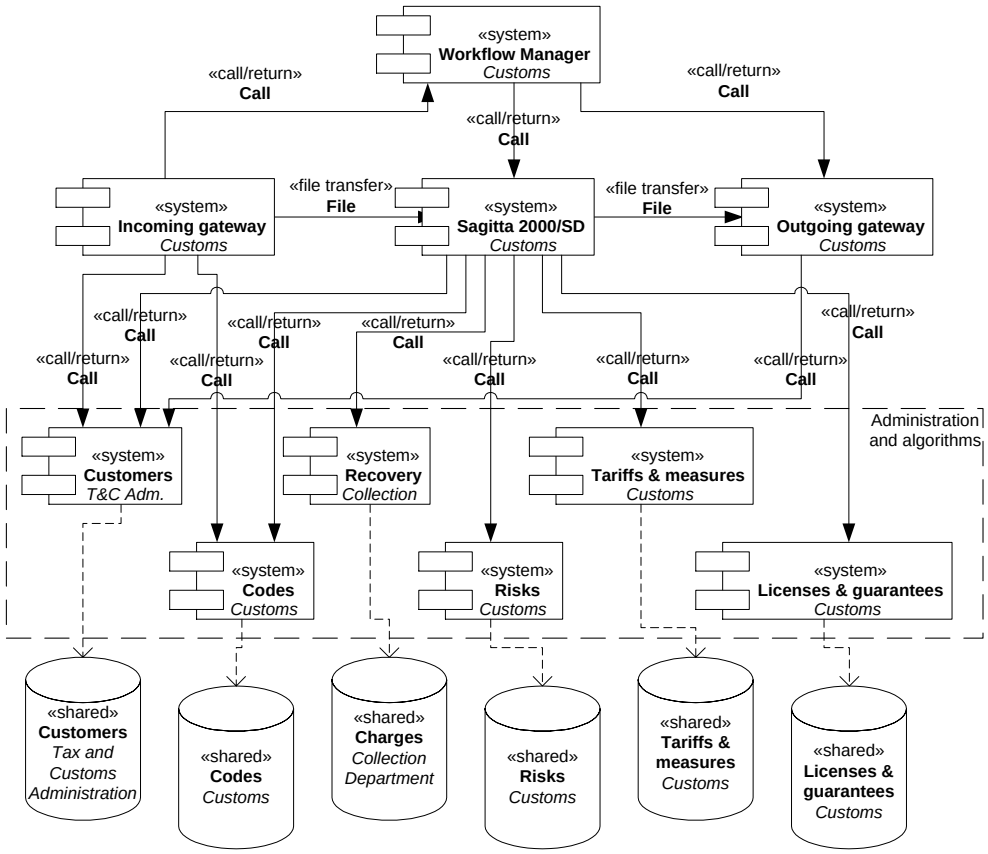

Figure 1: Context view of Sagitta 2000/SD

development effort of Sagitta 2000/SD is estimated at more than 60 person-years. Sagitta 2000/SD supports processing of supplementary declarations of large companies regarding their imports of goods. We were asked to analyze its modifiability.

The software architecture of this system[1] is depicted in Figures 1 and 2. Both figures employ the UML notation (Rumbaugh et al., 1998). In either figure, the boxes denote components and the arrows denote connectors between components. While the connectors in Figure 1 are type-labeled, those in Figure 2 are not further specified at this level of detail.

Figure 1 shows the context view of Sagitta 2000/SD. The purpose of this view is to show the systems in the environment of Sagitta 2000/SD and their communication. It is a view of the external architecture. Supplementary declarations are submitted by customers through the incoming gateway system. This system converts the declarations to a standard format and transfers them to Sagitta 2000/SD. The declarations are then checked in Sagitta 2000/SD and messages are sent to the customers through the outgoing gateway. The Workflow Manager controls the flow of declarations through the systems. The administration and algorithms systems provide access to a number of shared databases and systems that provide common functionality. These databases and systems are not directly accessible, but their data and functions are made available through ‘services’ of the administration and algorithms systems. Examples of these services are the ‘customer’ service, which is used to access data about customers, and the ‘tariffs and measures’ service, which determines the tax rates and measures that apply to a group of goods. The systems in the environment of Sagitta 2000/SD are owned and maintained by different departments.

The conceptual view is shown in Figure 2. This view shows the architectural solution that was cho-

[1]This case is treated more extensively in (Lassing et al., 2001). For the sake of brevity, we only consider the context and conceptual view of the system here.

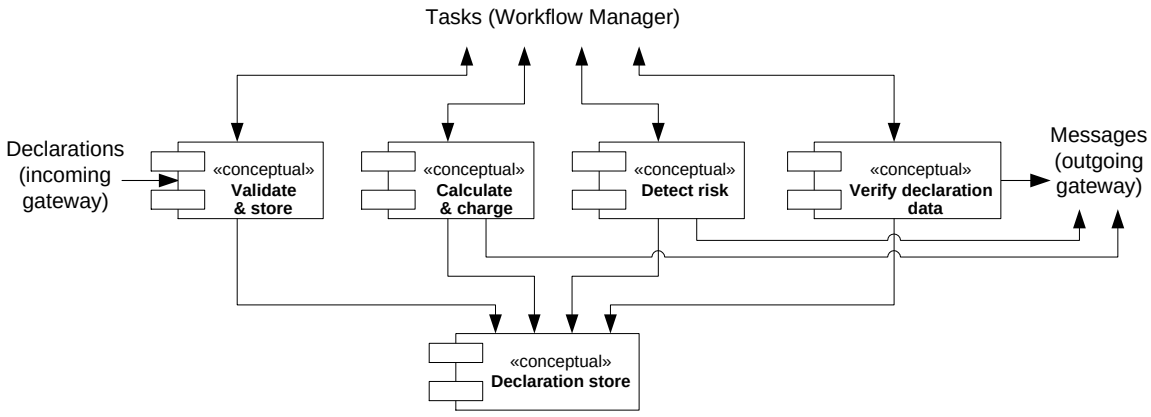

Figure 2: Conceptual view of Sagitta 2000/SD

sen for Sagitta 2000/SD. The conceptual view is a view of the internal architecture. Declarations enter the system from the incoming gateway, undergo a number of transformations, controlled by the Workflow Manager, and finally customers are informed about their assessments. The transformations that the declarations undergo are the following. First, the correctness of the declarations is validated, after which they are stored in the declaration store. The next step is to determine the sum of the charges, which is sent as an estimated tax assessment to the customer. After that, the verification procedure is determined and the customer may be asked to submit additional documents. The thoroughness of the verification is determined by the risk of the declaration and the required measures. The risk is determined using a number of factors, such as the level of the charge. Subsequently, the actual verification of the data of the declarations is performed according to the procedure determined in the previous step and a final tax assessment is sent to the customers.

5 Sagitta 2000/SD data

The input to this validation study consists of the change requests for Sagitta 2000/SD that were submitted in the period between the end of our initial analysis and the start of our validation effort. This period runs from september 1999 until february 2001. During this period 117 CRs were submitted, which were stored in a reporting tool together with an analysis of their effect and an estimation of the effort required. Accepted CRs are incorporated in one of the system releases, of which there were three during this period. It should be noted that these CRs are not the only source of changes realized in these releases; the releases also include changes that originate from the planned evolution of the system and adaptations to the technical environment. Changes of the former type are not documented individually and changes of the latter type are documented in another way. In this validation we limit ourselves to the set of changes stored in the CR tool, because the documentation we have at our disposal about the others is not sufficient for our analysis purposes. As a consequence, we only consider scenarios from our initial analysis that relate to changes with a functional flavor, and ignore those that pertain to the technical infrastructure only.

A first examination of the set of CRs learns that some of them concern situations in which the functional requirements were not implemented correctly. We do not include these CRs in this study, because they represent implementation bugs which were explicitly excluded from our initial analysis. To isolate the CRs that concern implementation bugs, we classify the CRs along two dimensions: (1) whether or not the CR leads to new functionality and (2) whether or not the functional specification was initially correct and complete with respect to the CR.

The result of this classification is shown in Table 1. The upper row in this table (‘New functionality added’) contains the CRs that lead to new system functions, either because of changed circumstances (left-hand column) or because these functions were not identified yet in the initial requirements analysis phase (right-hand column). The bottom row (‘New functionality not added’) contains the CRs that do not extend

Table 1: Change requests and the requirements

	Functional spec correct	Functional spec incorrect	Functional spec incomplete
New functionality added	28 CRs	-	6 CRs
New functionality not added	61 CRs	22 CRs	-

the system's functionality, but do require adaptations to the system. The CRs of this row either concern corrections to the functional specifications resulting in adaptations to existing functions (middle column), or inconsistencies between the way the system is implemented and its functional specification (left-hand column). The 61 CRs of the latter category represent implementation bugs. This leaves us with 56 CRs that are used in the remainder of this paper: 28 CRs that concern changes to the initial specs, 6 CRs that concern new functionality, and 22 CRs that concern errors in the initial specs.

Table 2: Mapping change requests to change scenarios

Change scenario	**CR**
S.1 What needs to be changed to include support for administrative audits in Sagitta 2000/SD?	–
S.2 Is it possible to include support for fully automated processing of supplementary declarations? (without human intervention)	–
S.3 What needs to be changed to enable customers to submit their supplementary declaration through e-mail, through EDI or on paper?	354
S.4 What needs to be changed to register additional information about clients?	–
S.5 What happens when the new European code system is adopted?	399
S.6 What happens when the underlying data model of Sagitta 2000/SD is changed?	332, 393
S.7 What changes are necessary to adapt the process belonging to the processing of supplementary declarations?	333
S.8 What is needed to make parts of Sagitta 2000/SD available to other systems?	–
S.9 What needs to be changed when RIN (new tax recovery system) is adopted?	–
S.10 What needs to be changed when BVR (new relation management system) is adopted?	–
S.11 What needs to be changed when the new workflow management system is adopted?	–
S.12 What needs to be changed when the incoming gateway comes under the responsibility of the central department concerned with all inputs?	–
S.13 What needs to be changed when the final incoming gateway uses another format than the temporary incoming gateway?	–

6 Predicting complex changes

As noted in the introduction, we are interested in identifying architectural risks and, thus, we look for complex change scenarios. We use a framework with categories of complex change scenarios to help us structure the scenario elicitation process. In this section we study whether we were able to predict the complex changes that have so far occurred in the life cycle of Sagitta 2000/SD. More specifically, we want to find out which complex changes we did not foresee and why. To find these changes, we classify the CRs along two dimensions:

- Did we foresee the CR in our initial analysis?
- Would our method classify the CR's impact as complex?

Our first step is to determine which CRs we did foresee during our initial analysis. The mapping of the set of CRs to the change scenarios identified in our analysis is shown in Table 2. We foresaw five CRs (332,

333, 354, 393 and 399[2]). Nine of the change scenarios identify did not actually happen.

The next step is to find out which of the CRs are complex according to ALMA, our analysis method. To this end, we classify the CRs using our scenario classification framework for risk assessment. In this framework, we distinguish the following categories of complex changes:

- Changes that involve different system owners. A system owner is an organizational unit financially responsible for the development and management of a system, or simply put the entity that has to pay for adaptations to the system. Changes are more complex if different owners are involved. Not only because of the additional coordination between parties, but also because all owners involved have to be persuaded to implement the necessary changes. We make a further distinction between changes initiated by the owner of the system under analysis, and changes initiated by others but require the system under investigation to be adapted.

- Changes that affect the architecture. These concern changes that affect the architecture, as opposed to changes affecting individual components only. In terms of figures 1 and 2, a change affects the architecture if it results in the addition or deletion of one or more components or connectors, or the semantics of one or more components or connectors changes. Overhauling the architecture is risky. We make a further distinguish between changes that affect the internal architecture only (i.e. the internals of Sagitta 2000/SD) and changes that affect the external architecture (Sagitta 2000/SD and the systems it interacts with).

- Changes that introduce version conflicts. Finally, changes are considered complex if they result in the presence of different versions of some architectural element. For instance, if a certain change in Sagitta 2000/SD necessitates a change in one of the services provided by the administration and algorithms systems (see figure 1), and another system using that same service does not implement the same change, we may have to cope with two versions of that service. Different versions of an architectural element may introduce a number of difficulties. Eventually, this may require changes to elements that were initially unaffected by the change.

The result is shown in Table 3. Three CRs are complex because they are initiated by the owner of Sagitta 2000/SD but affect other systems as well, eight CRs are complex because they require adaptations to the internal architecture. The other CRs are not complex.

Table 3: CRs in the scenario classification framework

Not complex	45 change requests
Initiated by the owner of the system under analysis, but require adaptations to other systems	3 change requests (**354**, 400, 424)
Initiated by others than the owner of the system under analysis, but require adaptations to that system	-
Require adaptations to the external architecture	-
Require adaptations to the internal architecture	8 change requests (306, 319, 324, 329, **333**, 334, 396, 397)
Introduce version conflicts	-
Total	56 change requests

Table 4 shows the result of combining both classifications. In the remainder of this section, we direct our attention to the upper row of this table – the goal of the architecture analysis of Sagitta 2000/SD was risk assessment and, therefore, we focus on complex changes. 11 CRs are complex, only two of which we had foreseen. Table 5 elaborates the two complex CRs that we did foresee. For both CRs, this table includes: (1) a short description of the CR, (2) an overview of its effect and (3) ALMA's view on its complexity.

[2]For the moment, we refer to CRs using their numbers only; some of the CRs are elaborated below.

Table 4: Results of classifying CRs (foreseen vs complex)

	Foreseen	Not foreseen
Complex (ALMA)	2 change requests (333, 354)	9 change requests (306, 319, 324, 329, 334, 396, 397, 400, 424)
Not complex (ALMA)	3 change requests (332, 393, 399)	42 change requests

Table 5: Foreseen CRs with complex associated changes

CR	**Description**
333	*Remodeling the business process for handling supplementary declarations:* The current business process turned out to have some limitations. Therefore, a number of changes to the business process were required. These were implemented collectively using this CR.
	Effect: The aim of the architectural solution with a separate Workflow Manager was to support changes in the business process by limiting their effect to the tables of the Workflow Manager. However, it turned out that the changes that were required for this CR could not be realized by adapting these tables only; dependencies between components involved in these processes made that the system's structure had to be adapted as well.
	ALMA: Complex, because the internal architecture of the system had to be adapted.
354	*Send the specification of taxes payable to customers using a medium other than paper:* It was the intention that the specification of taxes payable would be sent to customers on paper. However, in some cases, it turns out that this specification is too large to be sent on paper. So, another medium should be used instead.
	Effect: The outgoing gateway had to be adapted to support the new medium for sending the specification.
	ALMA: This CR is initiated by the owner of Sagitta 2000/SD but affects the outgoing gateway, for which another owner is responsible. Therefore, ALMA classifies this CR as complex.

In the initial analysis of Sagitta 2000/SD, we already identified CR 333 as change scenario S.7. When we evaluated the change scenario at that time, however, it was considered not complex, because the evaluation suggested that its impact would be limited to the Workflow Manager. When the change scenario actually occurred and was submitted as CR 333, it turned out that its effect was more dramatic; due to dependencies between components, the changes required the structure of the system to be adapted. We had not identified these ripple effects in our earlier analysis. This illustrates one of the experiences we discuss in (Lassing et al., 2000): although the software architect will have a reasonable understanding of the decomposition of functionality, it nevertheless often proves difficult to decide whether a change scenario will affect the interfaces of a component and, through these, affect other components in the architecture. The main reason for this is the amount of detail present in the description of the architecture, which in turn is determined by the architect's understanding of the problem and its solution.

Similar experiences are reported in (Lindvall and Runesson, 1998) and (Lindvall and Sandahl, 1998). In (Lindvall and Runesson, 1998), the authors show that even detailed design descriptions lack information necessary for performing an accurate analysis of ripple effects. Since less detail is available at the software architecture level, this is an even more relevant problem there. In (Lindvall and Sandahl, 1998), it is shown that even experienced developers clearly underestimate the number of components impacted by a change (though they were almost always right in identifying the components that needed change).

Each change scenario denotes a class of (similar) changes. In our analysis, we assumed that each change to the business process had the same effect on the software architecture. Apparently, though, some of these changes are more complex than others, and this particular change scenario was, on hindsight, defined too broadly. This resembles equivalence class partitioning as applied in software testing (van Vliet, 2000). If the partitioning is perfect, then the testset is perfect too; if not, there is a chance that some errors go unnoticed.

Table 6: Unforeseen CRs with complex associated changes

CR	Description
324	*Reasons for holding a supplementary declaration cannot be closed for a group of articles of a supplementary declaration:* When a supplementary declaration is processed, Sagitta 2000/SD checks whether conditions apply that require the declaration to be checked by a staff member. Examples of such conditions are additional documents that have to be handed in or the need for physical inspection of the goods. These conditions are called *reasons for holding*. The staff member has to check these and close all of them manually, before the declaration can continue through the process. Currently, it is only possible to close reasons for holding one by one. However, some declarations result in thousands of such reasons. Therefore, it should be possible to close a number of reasons for holding in one step, based on some selection criterion.
	Effect: To realize this CR, four components had to be added to the system, one for each of the four types of reasons for holding. These have to be integrated into the verification process of the system. In addition, conditions that allow one to select a group of reasons for holding have to be formulated .
	ALMA: This CR involves integrating four new components to the system so its effect can be classified as a change to the internal architecture and, therefore, as complex.
329	*Declarations can only be processed by one staff member at a time:* Large declarations are often processed by several staff members. In the current system, this is not possible. This CR is aimed to change this situation.
	Effect: This CR contradicted with the architectural approach of the system – a declaration is processed by only one staff member at a time – and thereby it affected almost all components of the system.
	ALMA: Because the approach of the system is changed, requiring the structure of the system to be adapted, ALMA classifies this CR as a change to the internal architecture of the system and, therefore, as complex.
400	*No historic information about entrepreneurs is kept:* Information about entrepreneurs is used to check whether the entity that submits a supplementary declaration is registered as an entrepreneur. However, no historic information on entrepreneurs is maintained. So, when an entrepeneur discontinues his business he is removed from the database and if he thereafter submits a declaration, this declaration is rejected.
	Effect: To maintain historic information about entrepreneurs, the datamodel that stores these had to be adapted as well as the 'customer' service that provides access to the information about entrepreneurs.
	ALMA: This CR is initiated by the owner of Sagitta 2000/SD but affects the customer service and the underlying system, for which other owners are responsible. Therefore, ALMA classifies this CR as complex.
424	*Customs means are not calculated when an article originates from a non-EU country, but passes through a EU country:* Currently, the tariffs calculation system, TGV, does not cater for the case that an article arrives from an EU country, but was originally sent from a country outside the EU. In those cases, custom means are not calculated properly.
	Effect: This CR required the interface of TGV to be adapated; not only should the country of origin be passed on to TGV, but also the country of dispatch. Furthermore, rules should be added to TGV concerning the tariffs and measures that apply to this type of article.
	ALMA: This change was initiated by the owner of Sagitta 2000, but it required another system, TGV, to be adapted. Therefore, ALMA classifies this CR as complex.

We now turn our attention to the most interesting CRs of Table 4: those that we did not foresee but would have liked to (upper right-hand cell). Table 6 elaborates some of these CRs [3].

Two CRs concern 'deep' functional changes (CRs 306 and 424). A functional change is termed 'deep' if its implemenentation requires the input of a domain expert. For instance, calculation of custom means for products that are shipped from one EU country to another EU country, which originate in a non-EU country, requires such expert input. The Computer and Software Centre of Dutch Customs is a centralised organization, with little end-user involvement in its software development projects. As a consequence, the stakeholders that we interviewed at that time were mostly people from the 'development side' too. This emphasizes once more that we should involve a mix of stakeholders from the 'development side' and 'user side'. An alternative explanation is that this type of change is difficult to predict at all, because they

[3]Note that some of the change requests were not implemented immediately and they reappear, sometimes specified in more detail. Change request 324, for instance, concerns the closure of reasons for holding a supplementary declaration for a group of articles. Similar changes are proposed in change requests 396 and 397.

originate from changes in national and EU regulations that are far from predictable. It is just not possible to foresee all changes.

Another category of CRs (CRs 319, 324, 334, 396 and 397) result from requirements that were initially not identified. It is not illogical that these CRs were not foreseen in the change scenario elicitation phase of the initial analysis. Had the stakeholders known that these requirements existed, they would have been implemented right from the start.

For the other CRs (CRs 329 and 400), no clear pattern can be distinguished.

A number of CRs (CRs 306, 319, 324, 329, 334, 396 and 397) are considered complex by ALMA because their effect is classified as a change to the internal architecture. Most of these are classified as such because new components are added to the system. We may question whether adding one or a few components really is the type of risk that we want to detect in software architecture analysis; they do not really affect the fundamental organization of the system, i.e. its software architecture.

CR 329, on the other hand, definitely does affect the system's software architecture. The whole system is based on the principle that a declaration is only processed by one staff member – it was explicitly stated so in the system's requirements; abandoning this principle would impact almost every aspect of the system. This CR represents the type of change that we should focus on in the analysis.

An important lesson to be learned from the above is that we may improve our change scenario elicitation process by: (1) challenging the requirements and not accepting them as is and (2) only classifying a change scenario as a change to the internal architecture when it affects fundamental design decisions, not when it merely requires adding or deleting a few components. The question remains how we decide whether a change affects fundamental design decisions. Concerning CR 329, for instance, the views of the system's architecture presented in section 4 do not show that Sagitta 2000/SD is based on the principle of one user per declaration. Nevertheless, this is apparently an important architectural decision related to modifiability. In this case, we could extend the views with narrative text indicating the principle of one user per declaration, but how does one decide that this information is that relevant?

7 Predicting the complexity of changes

Based on the set of CRs we are also able to determine how well our scenario classification framework allows us to predict the complexity associated with a change. To this end, we compare our classification of the complexity of each CR with an expert's opinion and study the differences.

ALMA's classification of the CRs has already been shown in Table 3. To get an expert's view on the complexity of the changes, we asked an experienced developer from Sagitta 2000/SD's development team, who was involved in implementing the change requests, to classify each of the CRs as either complex or not complex. This expert classified 20 CRs as complex and 32 CRs as not complex. For the remaining four CRs, the expert was unable to determine the complexity because these CRs were rejected. Based on their preliminary analysis, however, we were able to classify them in the classification framework.

Table 7: Complex in our view vs. complex in the expert's view

	Complex (expert)	Not complex (expert)	Unknown (expert)
Complex (ALMA)	9 change requests (306, 319, 324, 329, 333, 334, 396, 397, 424)	2 change requests (354, 400)	–
Not complex (ALMA)	11 change requests (280, 293, 303, 312, 332, 393, 418, 423, 427, 436, 445)	30 change requests	4 change requests (323, 336, 356, 426)

The (dis-)agreements between ALMA's classification and the expert's opinion are shown in Table 7. Two cells of this table are of particular interest, viz. those cells where ALMA and the expert disagree on the complexity of the required changes.

ALMA classifies two CRs as complex, which were classified as not complex by the expert. ALMA classifies these CRs as complex because they involve several system owners and, therefore, require additional negotiation and coordination between these owners (see the description of CR 400 in Table 6). We might call this 'politically complex'. However, the expert classified these CRs as not complex. Our expert is a developer: he may not have been part of the political fencing involved. Alternatively, it might be that the CR was not complex because the owners agreed on the necessity of the required changes and these changes were fairly simple to implement.

The opposite case concerns the CRs that the expert considers complex, while ALMA predicted they would not be. This cell of Table 7 contains 11 CRs, some of which are discussed in Table 8.

Table 8: CRs that the expert classifies as 'complex' and ALMA as 'not complex'

CR	Description
303	*Function component 'Calculate' (EBF017) wrongfully requests guarantees from customers:* In some cases, customers have to provide guarantees for their tax debt. Whether or not a guarantee is required, is determined by function component 'Calculate' (EBF017). However, it turns out that in a number of cases EBF017 does not properly decide on this issue and wrongfully requests guarantees from customers.
	Effect: To correct the indicated problem, EBF017 had to be adapted.
	Expert's opinion: Complex, because EBF017 contains very complex functionality. The adaptations to this component could only be implemented with help from a domain expert.
	ALMA: The effect of this CR is limited to one component, so ALMA classifies its effect as not complex.
393	*Declarations should have a repeating group of additional codes for articles and materials:* A supplementary declaration consists of a list of articles including a specification of the materials that they are made of. Both articles and their specification include 'additional codes'. These codes are, for instance, used by the tariffs calculation system, TGV, to determine the tariffs and measures that apply to the article. Currently, it is only possible to store a single additional code for each article and each material on a supplementary declaration. Due to future EU regulations, articles and materials may have to be accompanied by a number of these codes.
	Effect: The datamodel of Sagitta 2000/SD had to be adapted. And, as a result of that, the components of the system that access the data involved had to be adapted.
	Expert's opinion: Complex, because the adaptation to the datamodel affected a large part of the system, and had a number of unforeseen ripple effects.
	ALMA: This CR involves adaptations to a number of components, all of which belong to the same owner. So, according to ALMA's evaluation instrument, this CR is classified as not complex.
427	*When correcting a declaration it is possible to have the same data in a group twice:* A number of attributes of a declaration consist of a list. The system should check that these lists do not contain the same information more than once. However, it turns out that the system fails to do so when correcting a declaration.
	Effect: To solve this issue, the function component 'Correct' (EBF170/095) had to be adapted.
	Expert's opinion: Complex, because function component EBF170/095 is technologically very complex. The adaptation to this component had to be implemented by an experienced developer.
	ALMA: The effect of this CR is limited to one component, so ALMA classifies its effect as not complex.

A number of these 11 CRs are classified as complex by the expert because they concern a small subset of function components, that are either functionally or technologically complex. CR 303, for instance, was classified as complex, because it requires adaptations to the function component 'Calculate charges' (EBF017), which is functionally complex, meaning that adaptations to this component require the input of a domain expert. Something similar holds for CRs 303, 312, 418, 423 and 436.

CRs 293 and 427, on the other hand, were classified complex by the expert, because they affect function components that are technologically complex. Adaptations to these components can only be carried out by an experienced developer.

The other CRs (332, 393, 445) were classified as complex by the expert because they concerned changes to the system's datamodel. The complexity of these changes is brought about by the dependencies that exist between components of the system and the datamodel resulting in foreseen and unforeseen ripple effects. In a sense, the datamodel is just another complex component. Again, the decoupling of the datamodel and

its uses in the rest of the system is something that should be addressed in later development stages.

The complexity of these changes is caused by the complexity of a small subset of the components. We should be careful not to jump to the conclusion that this indicates that the software architecture of the system is 'wrong'. Consider, for instance, the study of software metrics by (Redmond and Ah-Chuen, 1990), in which they evaluated various complexity metrics for a number of systems, including the MINIX operating system. For this system, they found that the component that handles ASCII escape character sequences from the keyboard had the highest complexity. Although it is possible to reduce this complexity by splitting the component's functionality over several components, this would also reduce the understandability of the system as a whole. Apparently, this component is 'justifiably complex'. It might well be that the same holds for the complex components of Sagitta 2000/SD.

Apparently, it is difficult to adapt complex components. Therefore, the interfaces of these components should be very carefully selected; complex components should not be affected by changes not directly related to that specific component. The only existing software architecture analysis method that addresses this issue to some extent is SAAM (Kazman et al., 1996). By revealing scenario interactions – different scenarios that affect the same component – SAAM allows us to assess the allocation of functionality to components. Unrelated scenarios that affect the same component suggest suboptimal allocation of functionality. However, SAAM does not consider the complexity of the components themselves in this process; all components are treated alike. So, even if we would have studied scenario interactions, we would not have been able to foresee the complexity of these changes.

It is probably not possible to do more than that at the architecture level. After all, the architecture is an abstraction of the system, meaning that not all of the low-level details are known at this level yet. The complexity of individual components is an issue which merits attention in later development stages.

8 Conclusions

In this paper we report on a validation study that we performed for our Architecture-Level Modifiability Analysis method (ALMA). We revisited the Dutch Tax and Customs Administration to examine the actual evolution of Sagitta 2000/SD, a system whose software architecture we had analyzed two years before. The aim of this study was twofold: (1) assess our ability to *predict* complex changes and (2) assess our ability to predict *complex* changes. To this end, we collected the change requests (CRs) that were submitted since our initial analysis and compared these with the change scenarios that we found in the initial analysis. This study suggests a number of improvements to our analysis method:

- A total of 28 CRs issued during the analysis period concern cases where the requirements were not entirely correct to begin with. Although this may be the result of the particular development process followed for Sagitta 2000/SD, we expect this issue to occur in the life cycle of other systems as well. Apparently, we have been a bit optimistic in our change scenario elicitation in assuming the initial requirements to be correct. This optimism is shared by other software architecture analysis methods; none of them addresses this issue explicitly. The architecture analysis should improve if we explicitly challenge these requirements.

- Adding a few components to and deleting a few components from the system had better not be classified as changes to the system's internal architecture. This predicate should be reserved for changes that affect the fundamental organization of the system.

The study also hints at a number of fundamental limitations of this type of analysis:

- Fundamental modifiability-related decisions need not be visible in the viewpoints available. Though we may improve our guidelines as to the documentation to be included in our viewpoints, they will never be complete, and the involvement of expert stakeholders remains important.

- We did not predict a number of changes. They may have been overlooked by stakeholders during scenario elicitation. Our scenario elicitation process in particular involved fewer stakeholders from the user side than would have been desirable. On the other hand, the actual evolution of a system remains, to a large extent, an unpredictable process.

- The complexity of a number of CRs is caused by the fact that they concern complex components. We cannot really predict the complexity of individual components at the architecture level. And even if we could, we might not be able to reduce this complexity. Some components are inherently complex.

Acknowledgements

This research is mainly financed by Cap Gemini Ernst & Young. We thank the Dutch Tax and Customs Administration and its Computer and Software Centre for their cooperation. We are especially grateful to Kurt Kiezebrink and Lourens Riemens for their valuable comments and input. In addition, we would like to thank the anonymous referees for their feedback.

References

Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., and Zamerski, A. (1997). Recommended best industrial practice for software architecture evaluation. Technical Report CMU/SEI-96-TR-025, Software Engineering Institute.

Antón, A. I. and Potts, C. (1998). A representational framework for scenarios of system use. *Requirements Engineering Journal*, 3(3):219–241.

Bengtsson, P. and Bosch, J. (1999). Architecture-level prediction of software maintenance. In *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering*, pages 139–147, Los Alamitos, CA. IEEE CS Press.

Bengtsson, P., Lassing, N., Bosch, J., and van Vliet, H. (2000). Analyzing software architectures for modifiability. Technical Report HK-R-RES–00/11-SE, Högskolan Karlskrona/Ronneby.

Carroll, J. and Rosson, M. (1992). Getting around the Task-Artifact cycle. *ACM Transactions on Information Systems*, 10(2):181–212.

Ecklund, Jr, E. F., Delcambre, L. M., and Freiling, M. J. (1996). Change cases: Use cases that identify future requirements. In *Proceedings of OOPSLA '96*, pages 342–358, New York, NY. ACM.

Filippidou, D. (1998). Designing with Scenarios: A Critical Review of Current Research and Practice. *Requirements Engineering Journal*, 3(1):1–22.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1993). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA.

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

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, J. (1998). The architecture tradeoff analysis method. In *Proceedings of the 4th International Conference on Engineering of Complex Computer Systems (ICECCS'98)*, pages 68–78, Montery, CA. IEEE CS Press.

Lassing, N., Bengtsson, P., van Vliet, H., and Bosch, J. (2000). Experiences with software architecture analysis of modifiability. *Accepted for publication in Journal of Systems and Software*.

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

Lassing, N., Rijsenbrij, D., and van Vliet, H. (2001). Viewpoints on modifiability. *International Journal on Software Engineering and Knowledge Engineering*, 11(4):453–478.

Lindvall, M. and Runesson, M. (1998). The visibility of maintenance in object models: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 54–62, Los Alamitos, CA. IEEE CS Press.

Lindvall, M. and Sandahl, K. (1998). How well do experienced software developers predict software change? *Journal of Systems and Software*, 43(1):19–27.

Nosek, J. and Palvia, P. (1990). Software Maintenance Management: Changes in the Last Decade. *Journal of Software Maintenance*, 2(3):157–174.

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Redmond, J. and Ah-Chuen, R. (1990). Software metrics: A user's perspective. *Journal of Systems and Software*, 13(2):97–110.

Rumbaugh, J., Jacobson, I., and Booch, G. (1998). *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA.

Sutcliffe, A., Maiden, N., Minocha, S., and Manuel, D. (1998). Supporting Scenario-Based requirements engineering. *IEEE Transactions on Software Engineering*, 24(12):1072–1088.

van Vliet, H. (2000). *Software Engineering: Principles and Practice*. John Wiley & Sons, Chichester, England, second edition.

Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P. (1998). Scenarios in system development: current practice. *IEEE Software*, 15(2):34–45.