Criteria for the Evaluation of Implemented Architectures

Eric Bouwers
Software Improvement Group
Amsterdam, The Netherlands
E-mail: e.bouwers@sig.nl

Joost Visser
Software Improvement Group
Amsterdam, The Netherlands
E-mail: j.visser@sig.nl

Arie van Deursen
Delft University of Technology
The Netherlands
E-mail: Arie.vanDeursen@tudelft.nl

Abstract

Software architecture evaluation methods aim at identifying potential maintainability problems for a given architecture. Several of these methods exist, which typically prescribe the structure of the evaluation process. Often left implicit, however, are the concrete system attributes that need to be studied in order to assess the maintainability of implemented architectures.

To determine this set of attributes, we have performed an empirical study on over 40 commercial architectural evaluations conducted during the past two years as part of a systematic "Software Risk Assessment". We present this study and we explain how the identified attributes can be projected on various architectural system properties, which provides an overview of criteria for the evaluation of the maintainability of implemented software architectures.

1 Introduction

Any software system that is used will need maintenance in order to keep up with new demands and changing business requirements [11]. From this perspective, a good software architecture is desired because, according to Clements et al. [5]; *"Architectures allow or preclude nearly all of the system's quality attributes"*. Because of this, it is not surprising that a wide range of software architecture evaluation methodologies exists (for overviews see [2, 7]) for selecting an architecture that minimizes business risks.

Examining the review of Babar et al. [2], we conclude that almost all of the discussed methods focus on evaluating the quality of a designed architecture, i.e., evaluating the architecture before it is implemented. In contrast, the so-called *late architecture evaluations* [13] are focused on assessing the quality of an *implemented architecture*. Taking a closer look at the late architectural evaluation methods we notice that they only define the structure of the evaluation in the form of roles (e.g., evaluation team, architect, stakeholders) and steps (e.g., the nine steps of the ATAM [5]). Although this structure provides a basic framework, it does not explain which properties of a system should be studied. Usually, finding out which properties to study is part of the process itself.

Fortunately, there is research available that provides examples of system properties to study, see for example Kazman et al. [9] or Murphy et al. [14]. These techniques mainly focus on extracting a high-level (module) view of a system in terms of components and their (call)-relations. This view is then compared with a previously designed architecture. In this light, the quality of the implemented architecture is directly coupled with the conformance to the original designed architecture.

Unfortunately, in many cases the documentation of the architecture is not available or out-of-date [12]. Also, the architecture of a system contains more than the relationships amongst the main components. Many researchers agree that one needs to inspect a system using multiple views to get a complete overview of the architecture of a system [3, 10].

In order to evaluate the maintainability of a system, the Software Improvement Group (SIG) has developed the source-based Software Risk Assessments (SRA) [6] method, which it uses to assess systems on a commercial basis. Part of this method is dedicated to evaluating the implemented architecture of a software system. During the course of an SRA, a Maintainability Model [8] is used. This model provides an overview of several system properties to consider, including three system properties which address architectural issues from different perspectives.

Most system properties used within the Maintainability Model can be assessed by auditing a single system attribute. For example, the system attribute *lines of code* can be used to assess the system property *Volume*. Unfortunately, architectural system properties are often too broad to be assessed by a single system attribute. Instead, several system attributes need to be judged and combined to come to a balanced quality rating. This quality rating is currently based on expert opinion. To avoid inconsistencies in the quality ratings, systems are always assessed by multiple experts. However, it would be beneficial if the rating of an architec-

978-1-4244-4828-9/09/$25.00 2009 IEEE Proc. ICSM 2009, Edmonton, Canada

ture's quality could be (partially) derived in a more formalized way.

To help this we have conducted an empirical study to reach two goals: 1) identifying the system attributes the SIG experts have used to assess these architectural system properties, and 2) finding out how the system attributes are normally projected onto the three architectural system properties. Combining the answers to these research questions leads to an overview of criteria for evaluating the maintainability of implemented architectures.

The paper is structured as follows: we first introduce the environment in which the SIG evaluates implemented architectures in Section 2. After this, the architectural properties are introduced in Section 3. A problem statement and ensuing research questions are formulated in Section 4, followed by the design of our empirical study in Section 5. The results of the study are used to formulate answers to our research questions in Section 6. A discussion of the relevance of the study and threats to its validity is provided in Section 7, after which related work is discussed in Section 8. Finally, Section 9 provides a conclusion and pointers for future work.

2 Software Risk Assessments

The SIG has developed the Software Risk Assessment method to evaluate the maintainability of a software system. A first version of this method was described more than five years ago [6]. Since then, the SIG experts have used this method to assess over 80 systems, almost all from industry. In the course of these assessments, the method has been refined to better suit the purpose of the SRA. This section contains a description of the latest version of the SRA method reflecting this experience, to shown in which environment the SIG normally evaluates implemented architectures.

Goals and Deliverables The goal of an SRA is to answer the question a company has about the quality of their software system(s). Typical examples for the need of an SRA include package selection, quality assurance or deciding whether to maintain or rebuild a given system. A more detailed description of these scenarios is given in [6].

The outcome of an SRA is a report containing objective measurements of the source code, an objective representation of the concerns of the business and an expert assessment of the relation between the measurements and the concerns. Lastly, a set of scenarios for reducing the impact of potential risks is given. The duration of the project typically ranges between six to eight weeks.

Roles Figure 1 illustrates the different roles and responsibilities in the SRA process. The SRA Consultant is responsible for the overall process and delivering the final report. The SRA Analyst assists the SRA Consultant, mainly on the technical level, and is responsible for running the source code analysis, interpreting its outcome and supporting the SRA Consultant during technical interviews. The SRA Client is the organization that requested the SRA to whom the final report is delivered. The System Client is the organization that is using, or is going to use the system. In most cases, the System Client and the SRA Client are the same. Lastly, the System Supplier is the organization that has developed/maintained the system.

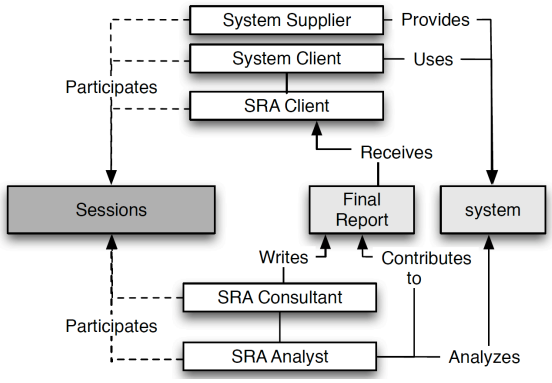

Figure 1. Roles and communications within the SRA process.

Sessions The SRA process includes four different sessions followed by the delivery of the final report. Before starting the first session the System Supplier transfers a copy of the source code of the system, as well as available documentation, to the SRA Analyst.

The first session is the *Technical Session* and is attended by the SRA Consultant, the SRA Analyst and the System Supplier. Within this session the process and target of the SRA are explained. Additionally, this session focuses on collecting all relevant technical information of the system.

After the Technical Session, the SRA Analyst starts the extraction of source-code facts from the system. Simultaneously, the SRA Consultant conducts a *Strategy Session* together with the System Client and the SRA Client in order to precisely identify the business goals of the client.

In the *Validation Session*, the SRA Consultant, assisted by the SRA Analyst, presents the derived facts to the System Supplier and the System Client. This session provides the opportunity to identify errors in the retrieved facts.

In the last step, the SRA Consultant and the SRA Analyst map the source-code facts onto the concerns of the business and derives scenarios that confirm or mitigate the concerns. Additionally, the scenarios for reducing the impact of the risks are identified. All of this is written down in the final

report which is presented in the *Final Presentation*. After this session, the final report is delivered to the SRA Client.

Experience The described SRA method has been successfully applied in the past four years. An internal report about customer satisfaction shows that our customers are highly satisfied with the outcome of their SRA. The survey over the year 2008 (with a response rate of 60 percent) reveals that over 90 percent of the clients are definitely interested in a new SRA (giving it a four out of five). Also, almost 80 percent of the clients would definitely recommend the SRA service to others (a number rated important by, for example, Reichheld [15]). Additionally, we have seen a steady growth in the number of SRA's carried out in the last two years. This increased demand for risk assessments justifies a further investment in making the process more systematic, which is the goal of the present paper.

3 Architectural System Properties

As part of an SRA, a software system is evaluated on a number of system properties, including both code-level properties and architectural properties. Code-level properties include the *volume* of the system, the *complexity* of its units, the degree of *redundancy* in its code lines etc. These code-level properties can be measured in a fairly direct way, by gathering source code metrics, aggregating them, and comparing them to statistically determined thresholds. A more detailed description of SIG's measurement model for code-level system properties can be found elsewhere [8].

To evaluate the implemented architecture of the system, three architectural system properties are distinguished, corresponding to different, but complementary, perspectives.

High-level Design The architectural property of high-level design is aimed at the technical division of the overall system into layers or other organizational and/or technological components. A typical example is the division of the system into a data layer, a business logic layer, and a user interface, following the *three-tier* architectural style.

Modularisation The architectural property of modularisation concerns the division of the main technical building blocks into functional modules. A typical example would be modules for account management, interest calculation, payment processing, annual reporting, etc. A single functional area is often addressed by several related modules, situated in distinct technical layers.

Separation of Concerns The architectural property of separation of concerns deals with the division of tasks over the modules within layers and over the source code units within modules. For example, within the module for payment processing, the tasks of user authentication, input validation, transaction logging, etc. may be addressed separately or in a tangled fashion. Also, some tasks may be handled fully at the data layer, while others are handled by a combination of units at the data and business logic layers.

Thus, these three architectural properties cover organizational elements at increasingly higher degrees of granularity: layers or components, modules, and tasks or concerns. At the granularity of high-level design, the focus is on technological choices. At the granularity of modules, the functional break-down takes center stage. Finally, at the granularity of concerns, the interplay of technical and functional divisions is addressed.

Unlike code-level properties, the architectural properties are not evaluated on the basis of source code metrics alone. Though certain source code metrics may be considered by the evaluator, many other factors are taken into account that are not readily quantified. In fact, the evaluation requires interpretation of a wide variety of observations and extensive software engineering expertise. In the remainder of this paper, we delve deeper into the exact criteria that are applied for this evaluation.

4 Problem Statement

The criteria employed by SRA Consultants for the evaluation of implemented architectures have emerged from practice. The overall distinction between High-level Design, Modularization, and Separation of Concerns emerged early and has been used in a stable fashion throughout many years. However, the observations to underpin judgements about these architectural properties were selected and used on a per-evaluation basis. The SRA Consultants may share a common understanding of observable system attributes and how they influence architectural properties, but this common understanding has not been documented in an evaluation-independent and re-usable form.

The lack of a documented set of observable attributes leads to a number of limitations. Firstly, without documentation, the evaluation method can only be taught by example to new SRA consultants, which is a time-consuming process. Secondly, the structure of the argument that backs up each evaluation must be constructed from scratch each time, even though they follow the same pattern. In practise, previous arguments are used as templates for new ones, while it could be more efficient to refer to a common model. Thirdly, a documented set of relevant system attributes would augment the traceability, reproducibility, and evaluator-independence of the evaluation method. Lastly, to use the architecture evaluation results for comparing systems, e.g. in order to benchmark the architecture of a system under evaluation against the architectures of previously

evaluated systems, a documented and shared overview of criteria is indispensable.

In order to discover and document a set of observable system attributes that can be used for evaluating implemented architectures, we have conducted an empirical study [16] into the evaluations performed by SRA Consultants of the SIG over several years. In particular, we set out to find answers to the following research questions:

Q1 Which set of system attributes do experts normally take into account when determining the quality rating of the three architectural system properties?

Q2 How do these system attributes influence the architectural system properties?

The answers to **Q1** documents which observable system attributes are relevant for architectural evaluation, while the answer to **Q2** documents which properties are influenced by them. Together, the answers to these questions help to remove the above-mentioned limitations.

5 Empirical Study

5.1 Design

We took the guidelines as proposed by Wohlin [16] into account to design our empirical study. The main input for this part are the final reports of 44 SRA's conducted between December 2006 and August 2008. Older reports do not consider the Maintainability Model and are therefore not taken into account. The reports contain a total of 54 system ratings and are written by seven different SRA Consultants. The systems that where reviewed cover a wide range of languages, sizes, ratings and business area's. An overview of this data is given in Figure 2. Note that the two lower barcharts respectively show the number of systems with a specific rating and the number of systems of a specific size.

For each report, we extract the arguments used for the quality rating for each of the architectural system properties. These arguments can be extracted from a table that appears in most final reports. This table lists all the system properties from the Maintainability Model, the rating for each system property, and a small argumentation for this rating. Additionally, each system property is discussed in a separate paragraph in the appendix of the report. When there is no table we only use the information extracted from that paragraph. In case of ambiguity we let the arguments in the table take precedence because these are the arguments most likely used to determine the final rating. From the list of all arguments we extract the set of system attributes by examining which system attributes are mentioned in the arguments. The result of this first step is given in Section 5.2.

After mining the list of system attributes, we iterate through all the reports a second time. In this iteration we determine which system attributes are used to rate each of the three architectural system properties. This is done in a separate iteration because the first step has given us a stable set of system attributes to work with, which makes it easier to categorize all arguments consistently. The result of this second step is given in Section 5.3.

Validation of the results is done in two ways. First, we conduct interviews with two experienced SRA Consultants in which we ask for an explanation of how they usually evaluate the three architectural system properties. Secondly, we present our findings to a group of ten SRA Consultants. In both cases the authors are not amongst the SRA Consultants. During the validation the SRA Consultants can identify new system attributes or projections. When this is not the case we conclude that the results are valid and provide a good overview of the current practice. The interview process and reports of the interviews are described in Section 5.4. The results of the study are validated in Section 5.5.

5.2 Report Study Results

In order to extract the system attributes from the arguments used in the reports we used an iterative process. The first report provided us an initial set of system attributes, after which we tried to place the arguments used in the second report under these system attributes. When an argument could not be placed under an existing system attribute we introduced a new system attribute based on a general description of the used argument. Adding a new system attribute was done conservatively in order to keep the list of system attributes manageable.

Finding the system attributes used in an argument was in most cases straightforward. For example, the argument "Usage of many different technologies" clearly touches upon the *Technology Combination* attribute. On the other hand, the argument "Implementation of data-access logic is bad" does not directly mention a system attribute. After reading the accompanying paragraph it became clear that the code for data-access was scattered all over the system. Therefore, this argument touches upon the *Functional Duplication* and the *Module Functionality* attribute.

Using this process we have identified 15 system attributes that are used in the evaluation process. The list of found system attributes is given in Table 1 and includes items one would typically expect such as layering or the use of frameworks, as well as less common attributes such as the (un)likelihood of certain technology combinations (e.g., Java and Pascal). For each system attribute we provide a name, a definition and an operational procedure to quantify the attribute called an "assessment approach".

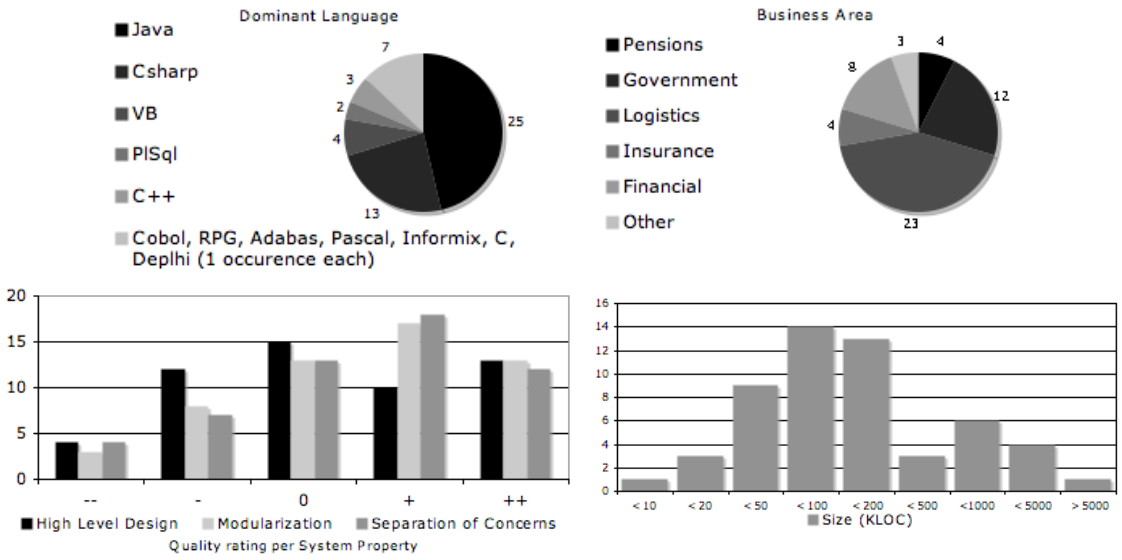

Figure 2. Distribution of the key characteristics of the 54 subject (sub)-systems

Name	Description	Assessment Approach
Abstraction	How well are input, output and functionality shielded throughout the system.	Inspecting maximum Inheritance Depth, create a call-graph showing the path between user interface and back-end.
Functional Duplication	The amount of functional duplication within the system.	Browsing the source code, identifying chunks of duplicated functionality.
Layering	The functional decomposition of the system into layers.	Inspecting the call graph on module level.
Libraries / Frameworks	The usage of standard libraries and frameworks.	Inspecting the list of imports, structure of the source- and build-files.
Logic in Database	The encoding of business logic in the database.	Inspecting the size and complexity of stored procedures and triggers.
Module Dependencies	The static dependencies (i.e., calls, includes) between modules.	Inspecting the call-graph on module level, matching this against expected dependencies.
Module Functionality	The match between the expected and encoded functionality within a module.	Expected module functionality is determined by interviews and available documentation, encoded functionality is determined by browsing the code.
Module Inconsistency	Whether similar modules have a different type of set-up.	Inspecting the structure of the source / method calls within a module.
Module Size	The match between expected size of a module and the actual size.	Expected module size is determined by the encoded functionality, actual size is measured by summing the LOC of all files in a module.
Relation Documentation / Implementation	The correctness of the relationship between the available documentation and the source code.	Manual inspecting of both the source code and the documentation.
Source Grouping	The complexity of grouping sources into modules.	Creation of filters to put sources into modules.
Technology Age	The age of the used languages and platforms.	Finding the technologies used is done by inspecting the different types of source code. Used platforms are determined by reading the documentation and through the technical session.
Technology Usage	Adherence to coding standards, patterns, and best practices.	Browsing the source code, using language specific style-checkers.
Technology Combination	How well the combination of technologies is expected to work.	Finding of the technologies is done in the same way as *Technology Age*, how common the combination is is based on expert opinion.
Textual Duplication	The amount of textual duplication within the system.	Checking the values of a duplication report.

Table 1. System Attributes mentioned in the rating of properties

	High Level Design	Modularization	Separation of Concerns
Abstraction	8	3	2
Functional Duplication	2	6	18
Layering	28	1	20
Libraries / Frameworks	22	1	1
Logic in Database	1	1	3
Module Dependencies	7	11	6
Module Functionality	4	32	13
Module Inconsistency	0	1	0
Module Size	1	1	0
Relation Documentation / Implementation	2	3	0
Source Grouping	0	14	2
Technology Age	13	0	0
Technology Usage	7	3	0
Technology Combination	5	1	0
Textual Duplication	0	0	4

Table 2. Number of times a system attribute is named in the rating of a system property.

5.3 Projection Results

After defining the set of system attributes we examined the reports in a second iteration and determined which system attributes are used as an argument for which system property. The result of this survey can be found in Table 2. Note that several system attributes can be mentioned in the rating of each of the system properties, which can result in more than 54 system attributes per system property.

5.4 Interview Description

The interviews with the two SRA Consultants took place on two different occasions. In both cases the SRA Consultant was asked to explain how he usually determines the rating of the three architectural system properties. Since the goal of these interviews is to validate our findings we did not provide the list of system attributes. Even though we did not impose a time-limit both interviews took around 60 minutes to complete. The reports of the two interviews are described below.

Expert 1 The first expert normally uses an analogy with the different dimensions of the board of a tic-tac-toe game. Each architectural system property is modeled as a separation of the functionality along one of the axis in the game. This analogy is made to explain the differences between the architectural system properties to the management of an organization more easily.

Modularization is explained as the *vertical* separation of functionality. The expert looks for modules in the code based on e.g. the directory structure, naming convention of files, packaging structure, etc. Roughly speaking, four modules are usually expected for a system with $<$ 20 KLOC, up to 10 modules for a system $<$ 100 KLOC and up to 20 modules in larger systems. After this, the files in the modules are inspected to discern if these modules encode certain functionality in a consistent manner. The sizes of the modules are inspected to see whether the distribution of the code is expected given the functionality encoded in the module, or whether there is an indication of poorly chosen modules (e.g. 1 module of 10KLOC and 15 modules of 100 LOC).

Separation of Concerns is explained as the *horizontal* separation of functionality. This involves, for example, the layering of the system. Are there layers for specific purposes such as presentation, data-access and business-logic? Is there one and only one place where communication with external systems or with the database is handled? Also, framework usage and violations between layers are taken into account in the rating of this characteristic. Finally, the interweaving of (for example) the definition of SQL-code and business-logic or embedding Java in JSP is considered to have a negative impact on Separation of Concerns.

High Level Design is explained as the *diagonal* separation of concerns. This is usually measured by inspecting the call-graph on module level and determining the absence or presence of loops (so each dependency between modules is only one-way). Also, the usage of modern programming languages and platforms are taken into account for this system property.

Expert 2 The interview with the second expert revealed the following definitions of the different architectural system properties. For High Level Design, the interaction of the system under assessment and other systems is examined. A typical question for this is: ”is there a clearly defined communication channel to the outside world?” Also, the expert looks for a high level division of the system into layers with separate functionality. Furthermore, the relation between the provided documentation (if any) and the source code is assessed. This relation is usually given a low priority except when there are large differences. Lastly, when frameworks contribute to the layering (for example frameworks for dependency injection or persistence) the usage of these frameworks is taken into account.

Modularization is rated by the way the system is divided into logical modules. This division is based on the package

	High Level Design	Modularization	Separation of Concerns
Abstraction			E2
Functional Duplication			E1
Layering	E2		E1
Libraries / Frameworks	E2		E1, E2
Logic in Database			
Module Dependencies	E1	E2	
Module Functionality	E2	E1, E2	E2
Module Inconsistency		E1	
Module Size		E1	
Relation Doc. / Impl.	E2	E2	
Source Grouping		E1	
Technology Age	E1		
Technology Usage			
Technology Combination			
Textual Duplication			

E1 = mentioned by expert 1, E2 = mentioned by expert 2.

Table 3. System attributes used per system property.

or directory structure, interviews with the customer and the presence of clearly defined subsystems. Furthermore, the relation between the documentation and the module structure is taken into account. Given the modules, an effort is made to put each module into one of the layers of the system. This is done based on the functionality of the module. This also includes the division of modules into functional modules and modules that act as utility-repositories. Finally, the dependencies between the modules is assessed using the call-graph. A good call-graph shows the layering where each module is part of one layer and each layer depends on one lower layer. Furthermore, a good call-graph shows all the utility modules because these modules only receive calls. A bi-directional dependency in the call-graph usually hints at an implementation or design flaw.

Separation of Concerns is rated by looking at the separation of functionality within modules. An example of this is whether the interfacing between two components in a module is separated from the implementation. A different example is whether a module that allows access to the outside worlds implements this access as a thin layer on top of 'real' functionality instead of encoding business logic into the functions / objects that provide the actual access. Also, frameworks that do not directly contribute to the layering of the system are taken into account.

5.5 Validation

When we process the reports of the interviews with the experts in the same we as we analyzed the final reports, we see that they do not introduce any new system attributes. All of the arguments used for each system property can be placed under the 15 system attributes listed in Table 1. For example, the first expert explains that Modularization is judged by looking at the sizes of the modules, which corresponds with the *Module Size* attribute. A second example is that he mentions "... the use of modern programming languages ..." as an argument for High Level Architecture. This corresponds to the *Technology Age* attribute. The experts opinion of how the system attributes project onto system properties is summarized in Table 3.

Additionally, the presentation of the findings to a group of ten SRA consultants did not lead to an addition of new system attributes. During the discussion that followed the presentation the SRA Consultants concluded that they did not miss system attributes they normally use. Also, they agreed that the projection of the system attributes as given in Table 2 provides a general overview of the current practice.

6 Answers to research questions

Q1: Which system attributes do experts take into account when evaluating architectural system properties? The 15 system attributes presented and defined in Table 1 are taken into account for the evaluation of architectural system properties by SIG's software assessment experts.

Note that some of these attributes, e.g. *Module Inconsistency* and *Module Size*, occurred with a very low frequency (see Table 2). Still, these attributes were also mentioned in the expert interviews (see Table 3), which indicates that they are actively used and should therefore not be excluded from the list.

Q2: How do these system attributes influence the architectural system properties? The data in Table 2 provides the raw historical data of how the system attributes have influenced the architectural system properties. From this data, we can mathematically deduce a) which system attributes are most important for each system property, and b) which system property each system attributes influences most. The answers to these questions provides an overview of how the system properties are influenced by the system attributes.

a) Which system attributes are most important for each system property? When all system attributes are of equal importance for each system property they would have been

	High Level Design	Modularization	Separation of Concerns
Abstraction	X		
Functional Duplication		X	X
Layering	X		X
Libraries / Frameworks	X		
Logic in Database			
Module Dependencies	X	X	X
Module Functionality		X	X
Module Inconsistency			
Module Size			
Relation Doc. / Impl.			
Source Grouping		X	
Technology Age	X		
Technology Usage	X		
Technology Combination			
Textual Duplication			

Table 4. Most Important System Attributes Per System Property

	High Level Design	Modularization	Separation of Concerns
Abstraction	X		
Functional Duplication			X
Layering	X		X
Libraries / Frameworks	X		
Logic in Database			X
Module Dependencies		X	
Module Functionality		X	
Module Inconsistency		X	
Module Size	X	X	
Relation Doc. / Impl.	X	X	
Source Grouping		X	
Technology Age	X		
Technology Usage	X		
Technology Combination	X		
Textual Duplication			X

Table 5. Most Important System Properties Per System Attribute

used an uniform number of times. By first calculating the average number of usages for each system property, we can filter out the system attributes with a lower usage count than this average. This filtered set gives us an overview of the system attributes which are most important for that system property.

For example, when we add all mentioning of system attributes for *Modularization* we get a total of 78 usages. When all system attributes would contribute to *Modularization* in the same way we expect each system attribute to be used $78/(number\ of\ system\ attributes) = 78/15 = 5.2$ times. Since this is not the case, we can filter out the most important system attributes for *Modularization* by stripping away all system attributes which where mentioned less than five times. This leaves us with only the four most important system attributes for *Modularization*, see Table 4.

Comparing the projection of the specific experts shown in Table 3 with the projection of the historical opinion in Table 4 we observe that some deviation between the two exists. We believe that this deviation stems from the fact that the projection of the experts is extracted from a single free-form description of the assessment process, while the projection in a report contains the consolidation of multiple discussions amongst SRA Consultants. Also, during the interviews the SRA Consultants most likely mentioned only those system attributes that they consider to be important in general, which are not necessarily the same as the system attributes that are most important for each system property.

b) which system property is influenced most by each system attribute? In order to compute the answer to this question we again compute a threshold for average use, but in this case we use this threshold to filter out the most important system property for each system attribute. For example, the system attribute *Technology Combination* is used a total of 6 times. This provides us with a threshold of $6/(number\ of\ system\ properties) = 6/3 = 2$. Using this threshold we conclude that the system property *High Level Design* is most relevant for this system attribute. The algorithm described above results in the listing for which system property is influenced most by each system attribute shown in Table 5.

7 Discussion

Applicability By documenting the system attributes used to evaluate implemented architectures, as well as their projection onto system properties, the limitations mentioned in Section 4 can be lifted. Training new consultants becomes easier because the documentation is available, argu-

mentation of the evaluation can refer to this documentation which increases efficiency, and by using the documentation as a guideline during evaluations the traceability and reproducibility of the evaluation method increases. Lastly, ratings determined with the documentation in mind allow comparison against a benchmark of earlier evaluations, providing better insight into the quality of the implemented architecture under review.

In general, we can think of other useful applications for both the list of system attributes and the projection onto system properties. For example, we believe that the list of system attributes can directly be used in existing architectural evaluation methods to make them more operational. Investigating the impact of using a common set of attributes within, for example, the ATAM [5] is part of our future work. Furthermore, we like to investigate whether the usage of a standard set of system attributes makes it easier to compare the results of different evaluation methods.

More speculatively, we envision that the list of system attributes combined with the projection can be turned into a light-weight "sanity check" for implemented architectures. This can, for example, be done by providing questionnaires with a few qualitative questions about each system attribute. When multiple persons familiar with the system fill out these questionnaires the general opinion about the quality of the implemented architecture can quickly be determined by averaging the answers. Naturally, the results of such a sanity check are not of the same quality as a complete architecture evaluation. However, we believe that performing such a check on a set of system can be useful to, for instance, filter out the system that is in most dire need of a complete evaluation.

In any case, the list of system attributes serves as a "wish"-list for researcher to develop automatic, qualitative measurements. Taking into account the importance of the system attributes determined by the projection we can quickly spot for which system attribute there is a need for qualitative, easily (or more preferably automatically) calculable metrics. Developing these type of metrics is part of our future work.

Threats to Validity A first threat to validity is whether the data sources of the empirical study are representative. We believe that the set of reports is representative because Figure 2 shows that the used reports were written for systems covering a wide range of industries, system sizes and programming languages. Also, the lower-left chart in Figure 2 shows that the 54 ratings cover the complete spectrum of quality ratings. This implies that the study is not based on only problematic systems.

The data extracted from the interviews might not be representative because we have only interviewed two SRA Consultants. Even though the two interviewed SRA Consultants are amongst the most experienced consultants in our company, the low number of interviewees might lead to system attributes not being discovered through these interviews. This threat is countered by the fact that we used the interviews only as a secondary source of validation.

A second threat is the reliability of the measurements of the report study. Even though a consistent process was followed, a different person might find different arguments in the reports. An inter-rater test would help in countering this threat. Performing such an inter-rater test will be part of our future work.

Lastly, a threat is that our results cannot be generalized towards an environment outside the SIG. Even though, we believe that the procedure for our case study is described in such a way that it can be replicated, we cannot say for sure whether a replication of our study in a different environment will reproduce similar results. Even if the process of an SRA as explained in Section 2 is followed, the final outcome of the SRA's, and therefore the results of the case-study, are dependent on the skills of the experts, and the nature and properties of the assessed software systems. Actual replication of our study is needed in order to counter this threat.

8 Related Work

In a study done at AT&T [1], fifty evaluation reports were mined for indicators that can predict the *risk* of a project. The authors found twelve main categories of issues. Unfortunately, these twelve categories where too broad to be useful, so they had to use more concrete issues to make a reliable method for predicting the risk of a project. Even though the domain of the study differs, it does show that it is useful to identify lower level system attributes.

As discussed in Section 7, the list of system attributes together with the analysis of the projection provides us with a first overview of criteria for evaluating implemented architectures. According to the overviews of Babar et al. [2] and Dobrica et al. [7], the only architectural evaluation method aimed at implemented architectures is introduced by Bengtsson et al. [4]. The method describes four different approaches for assessing quality attributes, i.e., scenarios, simulation, mathematical reasoning and objective reasoning. However, the actual criteria to use during the evaluation are not mentioned.

More recently, Lilienthal [12] defined a model to assess the complexity of implemented architecture. The system attributes she uses are similar to ours, but the model does not take into account systems implemented in multiple languages. Also, the environment of the system (e.g., libraries used, platforms it runs on) are not considered. Lastly, our work goes beyond the work of Lilienthal by providing not only an overview of criteria, but positioning it in a larger

process to assess the maintainability of systems in general.

There also exists some research in the area of assessing the individual system attributes. For example, Lindvall et al. [13] links the documented architecture of a system to the actual implementation. Their case study shows an example of how the connections between modules, a specific instantiation of our module dependencies, are used to assess this link. As mentioned before, both Kazman et al. [9] and Murphy et al. [14] use module dependencies to assess an implemented architecture. Even though this research provides a solid basis for assessing these attributes in isolation we believe that they do not cover all aspects of an implemented architecture.

9 Conclusion

This paper describes our steps for finding criteria for the evaluation of the maintainability of implemented architectures. The main contributions of this paper are:

- A description of an empirical study using over 40 SRA reports (Section 5)
- The identification of 15 system attributes that have an impact on the maintainability of an implemented architecture (Table 1)
- An analysis of the projection of the found system attributes onto three architectural system properties (Section 7)

Additionally, we have extended the work presented in [6] by giving a more detailed description of the SRA process in Section 2. Combining the identification of the system attributes with the analysis of the projection provides us with a first overview of concrete criteria for the evaluation of the maintainability of implemented architectures.

Future Work First of all, we are going to conduct an inter-rater test in order to further validate our results. Additionally, we would like to validate the criteria by applying it to future SRA's. More specifically, we are interested in finding out whether the defined criteria make it easier for SRA Consultants to assess the three architectural properties.

Also, we intend to look into ways to automate the quality assessment of the different system attributes. Automated measurements do not only make sure that the final quality assessment is more objective, it also helps in reproducing the results of an assessment.

Other areas of future work include the development of light-weight sanity check based on the 15 system attributes, and determining the impact of using the 15 system attributes in existing architecture evaluation methods such as the ATAM [5].

Acknowledgments The authors would like to thank all the colleagues at the Software Improvement Group for their interesting discussions, their feedback and their participation in the empirical study.

References

[1] A. Avritzer and E. J. Weyuker. Investigating metrics for architectural assessment. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*. IEEE Computer Society, 1998.

[2] M. Babar, L. Zhu, and D. R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. In *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference*. IEEE Computer Society, 2004.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, 2003.

[4] P. Bengtsson and J. Bosch. Scenario-based software architecture reengineering. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, June 1998.

[5] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures*. Addison-Wesley, 2005.

[6] A. v. Deursen and T. Kuipers. Source-based software risk assessment. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2003.

[7] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Trans. Software Eng.*, 28(7):638–653, 2002.

[8] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *QUATIC '07: Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39. IEEE Computer Society, 2007.

[9] R. Kazman and S. J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Eng.*, 6(2), 1999.

[10] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[11] M. M. Lehman. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[12] C. Lilienthal. Architectural complexity of large-scale software systems. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, 2009.

[13] M. Lindvall, R. Tesoriero Tvedt, and P. Costa. An empirically-based process for software architecture evaluation. *Empirical Softw. Engg.*, 8(1):83–108, 2003.

[14] G. C. Murphy, D. David Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28. ACM, 1995.

[15] F. Reichheld. The one number you need to grow. *Harvard business review*, 81(12):46–54, 2003.

[16] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.