Delft University of Technology
Software Engineering Research Group
Technical Report Series

A Light-weight Sanity Check for Implemented Architectures

Eric Bouwers and Arie van Deursen

Report TUD-SERG-2010-003

TUD-SERG-2010-003

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in IEEE Software, Special Issue on Software Evolution: Maintaining Stakeholders Satisfaction in a Changing World, July/August 2010, IEEE Computer Society.

A Lightweight Sanity Check for Implemented Architectures

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

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

Abstract

Architecture evaluations offer many benefits, including the early detection of problems and a better understanding of the possibilities of a system. Although many methods are available to evaluate an architecture, studies have shown that the adoption of architecture evaluations in industry is low. A reason for this lack of adoption is that there is limited out-of-the-box process and tool support available to start performing architecture reviews.

In this article we introduce LiSCIA, a Light-weight Sanity Check for Implemented Architectures. It can be used out-of-the-box to perform a first architectural evaluation of a system. The check is based on years of experience in evaluating the maintainability of software systems. By periodically performing this check, the erosion of the implemented architecture as the system (and its requirements) evolves over time can be controlled.

Keywords Software Architectures, Software Architecture Evaluation, Architecture Erosion, Software Quality

1 Introduction

Software architecture has been loosely defined as the organizational structure of a software system including components, connectors, constraints, and rationale [10]. Evaluating a software architecture of a system helps in checking whether the architecture complies with the design goals and wishes of the stakeholders. Additionally, the evaluation can result in a common understanding of the architecture, its strengths and weaknesses. All of this helps to determine which quality criteria the system meets, since *"Architectures allow or preclude nearly all of the system's quality attributes"* [6].

Many architecture evaluation methods are available [2, 8]. Unfortunately, a survey conducted by Babar et al. [1] showed that the adoption of architecture evaluations in industry is low. Their conclusion is that *"there is limited out of the box process and tool support for companies that want to start doing architecture evaluations"*.

In this paper we propose a way to bridge this gap, by presenting a *Lightweight Sanity Check for Implemented Architectures* (LiSCIA). It is based on nine years of experience in the evaluation of over 100 different industrial software systems, as well as on our earlier research on maintainability indicators [4, 7, 9].

LiSCIA is a concrete, easy-to-apply, architecture evaluation method of which the goal is to obtain insight in a system's quality within a day. By applying LiSCIA at the start of a software project and then periodically, for example every 6 months or at every release, potential problems with the implemented architecture can be spotted quickly and dealt with it at an early stage.

2 Background

Existing methodologies for architecture evaluations have been divided into *early* and so-called *late* evaluations [8]. Early evaluations focus on designed architectures, while late architecture evaluations focus on an architecture after it has been implemented. LiSCIA falls in the latter category as it is aimed at evaluating an actually implemented architecture.

Our experience shows that recurrent evaluation of an implemented architecture helps to identify *architecture erosion* [13], the steady decay of the quality of an implemented architecture. In the past years, the Software Improvement Group (SIG) has been offering this type of recurrent evaluations as part of its Software Monitoring service [11] and Software Risk Assessments [7] (SRAs). In both services, the technical quality of a system is examined and linked to business risks. In an SRA this is done once, while a Software Monitor follows the evolution of a system over a longer period of time.

Recently, we have conducted a study using over 40 risk assessment reports of the past two years. We identified 15 system attributes that influence the quality of an implemented architecture [4]. These 15 attributes, together with our experience in monitoring the development of software systems in the past years, form the basis of LiSCIA. Because of this, LiSCIA can be seen as an operationalization of the identified attributes. Additionally, LiSCIA represents a basic formalization of the steps we normally undertake at the start of an SRA and during the re-evaluations within a Software Monitoring project.

LiSCIA focuses on the *maintainability* quality attribute of a software system. Due to the light-weight nature, a complete architecture evaluation is not offered. However, by recurrently applying LiSCIA, insights into the current status of the implemented architecture of a system can be obtained. With this recurring insight, the erosion of the implemented architecture can be controlled. Additionally, the result of LiSCIA offers a platform to discuss current issues and can justify refactorings or a broader architecture evaluation.

3 LiSCIA

For the design of LiSCIA, the following key-issues were taken into account to ensure that it is practical, yet generally applicable:

- The evaluation takes an evaluator no more than a day.
- The evaluation includes ways to improve the system, i.e., it helps the evaluator to define actions.
- The evaluation is not limited to a specific programming language or technology.
- The evaluation is able to handle different levels of abstraction.

LiSCIA is divided into two different phases, a start-up phase (done once) and an evaluation phase (performed for every evaluation). The result of the start-up phase is an overview report, which is the input for the evaluation phase. The result of the evaluation phase is an evaluation report, containing the results of the evaluation and actions to be taken. These actions might require adjustments to the overview report. Both the (possibly adjusted) overview report and the evaluation report serve as input to a re-evaluation of the system. An illustration of the complete process is given in Figure 1.

Before describing the two phases in depth, three key elements of LiSCIA need to be defined: the *module*, the *unit* and the *container*.

3.1 Definitions

LiSCIA uses the module viewtype [5] to reason about the structure of an implemented architecture. This viewtype divides the system into coherent chunks of functionality called *modules*. A module can represent some business functionality, such as ”Accounting“ and “Stocks”, or a more technical functionality, such as “GUI” and “XML-processing”. LiSCIA can be applied to both decompositions. Better yet, LiSCIA can be applied to the same version of a system using different decompositions. In this way, different modularization-views on the architecture can be explored. Each of these views can give different insights, which can lead to a better understanding of the implemented architecture as a whole.

A *unit* within LiSCIA is a logical block of source-code that implements some sort of functionality. The typical unit in LiSCIA is that of a source-file. Units are grouped into *containers*; within LiSCIA the normal container is a directory on the file-system.

Using the source-file as a unit complies with the notion that files are typically the dominant decomposition of functionality [14]. In other words, most programming languages use the file as a logical grouping of functionality. An argument against this decomposition is that some technologies offer a more fine-grained granularity of functionality. For example, for the Java language, the classes (or even methods) can be seen as a separate decomposition of functionality. The choice for files is made to make the method independent of the evaluated technology.

3.2 Start-up Phase

In the start-up phase, the evaluator first has to define the modules of the system under review. In some cases, the modules are defined in the (technical) documentation. In other cases, the modules are apparent from, for example, the directory, package or namespace structure. When high-level documentation is not available, the modules can be obtained from an interview with the developers of the system. In our experience, developers have no problem with producing a description (and drawing) of the modules and the relationships between the modules of the system they work on. These descriptions are often wrong in details, but adjustments to these descriptions can be defined as actions in the evaluation phase.

After defining the modules, each unit in the system should be placed under one of the defined modules. This is done by placing patterns on the names of the units and the containers to which they belong. For example, if all units in the container called ”gui“ are part of the GUI-module, a logical pattern for this module would be `.*/gui/.*`.

Every unit should only be matched to a single module to ensure a well-balanced evaluation. When a unit should actually be placed under multiple modules, it indicates that this unit implements parts of different functionalities, which is a sign of limited separation of concerns. These units should either be split up into different parts, or a module capturing

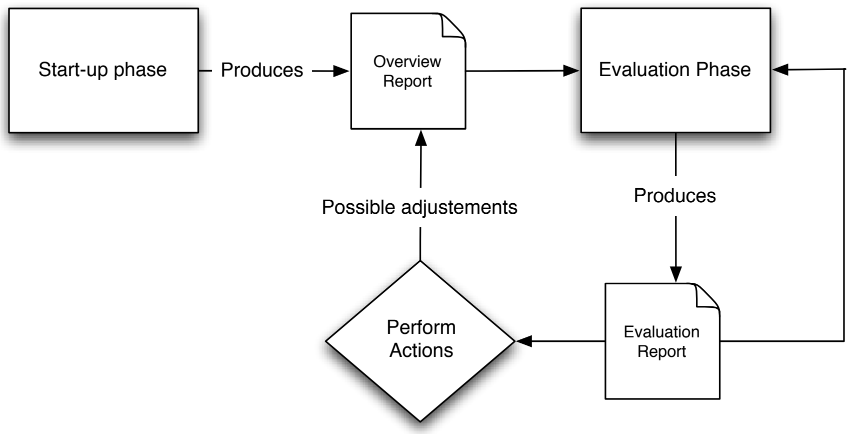

Figure 1. Overall flow of LiSCIA

the two functionalities should be introduced.

As a last step in the start-up phase, the evaluator should identify the types of technologies used within the project. For LiSCIA, the term "technologies" can be read as programming languages, but can also include frameworks, libraries, build tools and possibly even hardware platforms.

A description of the modules, the patterns and a list of technologies is documented in the report of this phase.

3.3 Evaluation Phase

The evaluation phase of LiSCIA consists of answering a list of questions concerning the architectural elements identified in the start-up phase. Many of the questions are grouped into pairs. Usually, the first question asks for a specific situation, after which a second question asks for an explanation. The answer to the first question is either 'yes' or 'no', while the answer to the second question is open-ended. This set-up requires the evaluator to be explicit, but leaves room for explaining why a certain situation occurs.

In addition to the questions, LiSCIA provides a set of actions linked to the questions. These actions can be used as a guide to answer the questions. In principle, the answers to the open-ended questions must explain why the action belonging to a question does not need to be taken. The actions are defined in such a way that when there is no valid reason to ignore the action, the maintainability of the implemented architecture can benefit from performing the given action.

The questions and actions are divided into five different categories. These categories cover the grouping of sources, the technologies used in the system, and the functionality, size, and dependencies of modules. Table 1 lists some key-properties of these categories such as topic, number of questions and number of actions. The complete list of 28 questions and 28 actions can be found online[1].

Each category contains questions related to the current situation, and questions related to the previous evaluation. During the first evaluation session, the questions in the latter category can be ignored, since their primary objective is to reveal the reasons for differences between the versions compared.

3.3.1 Source Groups

A good example of the re-evaluation questions can be found in the first category of questions. As a first step in this category, the evaluator has to determine whether all units belong to a module given the patterns described in the overview report. During the start-up phase, the patterns for the module are designed to capture all units. Because of this, it is to be expected that during the first evaluation, all units are placed under a module. Since most of the questions in this category are about units that are not matched to a module, the questions in this category can be ignored for this evaluation session.

During later evaluation sessions, it sometimes happens that new units have been added to the system that are not matched by any module-pattern. We have experienced this situations on a multitude of occasions. In some cases, these units are simply misplaced by mistake and the units can easily be moved to their correct location. In other cases, a new module (together with a new pattern) needs to be introduced because new functionality is introduced. Questions regarding the ideas behind this new module, and possibly additional modules, are also part of this category of questions.

Even though it is to be expected that the mapping of units to modules is complete during the first evaluation, this is

[1]http://www.sig.eu/en/liscia

Name	Topics of Interest	#Questions	#Actions
Source Groups	current grouping of units in modules, future modules	4	4
Module Functionality	decomposition of functionality over modules	5	6
Module Size	size of modules, distribution of system size over modules, growth of modules	6	5
Module Dependencies	expected, circular, unwanted dependencies and changed dependencies	6	7
Technologies	combination, version, usage and size distribution of the used technologies	7	6

Table 1. Key properties of the categories of LiSCIA

not always the case, especially when existing documentation is used as part of the overview report. For example, in one of our assessments we evaluated a system containing over 4 million lines of code. The existing documentation described several modules. In addition, the documentation included a file describing the mapping of each source-file to one of these modules. Evaluating this mapping carefully, we discovered that over 30 percent of the source-files could not be assigned to a module. Additionally, a considerable part of the mapping contained source-files that did not exists anymore. One of the actions defined for this system was to re-order the source-files to better resemble the module structure as outlined in the documentation. This example illustrates that it is important to verify existing documentation against the current implementation, instead of assuming that the documentation is correct.

3.3.2 Module Functionality

Within the second category, the evaluator focusses on how the functionality of the system is spread out over the modules. For example, one of the questions asks whether the functionality of each module can be described in a single sentence. The question help the evaluator in determining whether the current decomposition of functionality is not too generic. In several cases, we encountered systems that defined a module called 'Utilities' (or something similar). When the exact functionality of this module is described, it turns out that this module does not only contain generic functionality, but also contains business functionality, specific parsing functionality, or an object model. In these cases, the action is to split up the module in a truly generic part and separate modules for the more specific types of functionality.

3.3.3 Module Size

This third category of questions is related to the size of the modules. In order to keep LiSCIA usable for a large range of systems, the exact definition of "size of a module" is intentionally kept abstract. Nevertheless, in most cases the size of a module can be represented by the sum of the Lines of Code of all units in the module.

The questions in this category are not only related to the size of the individual modules, but also consider the distribution of the size of the complete system over the modules. Additionally, this category contains questions related to the growth of the modules. These last type of questions are especially useful in detecting unwanted evolution within the system, but also help in detecting unwanted development effort.

For example, in one of our monitoring projects we had several modules marked as "old". These old modules contained poor quality legacy code which was still used, but which was not actively maintained. After inspection of the growth of the size of the modules, it was discovered that new functionality was still being added to the "old" modules. The explanation given was that it was easier to add the new functionality in this module. Even though it was easier, it also resulted in the addition of large and complex pieces of code because it had to follow the structure of the legacy-code. The action that resulted from this observation was the migration of the functionality to a new module, where it would be easier to maintain and test.

3.3.4 Module Dependencies

To answer the questions in this category, the dependencies between modules need to be available. Similar to the size of a module, the concept of "dependencies between modules" is kept abstract. However, the dependencies between modules can be calculated by first determining the dependencies between units (for example, the calls between methods inside the source-files), after which these dependencies can be lifted to the module level.

The questions in this category help the evaluator to assess the wanted, unwanted and circular dependencies between modules. In addition, questions about added and removed dependencies are defined for when a previous evaluation is available. One of the questions in the latter category is whether there are any new dependencies added, as was the case on one of our monitoring projects. The follow-up question is whether this dependency is expected, which, in the case of this project, it was not. The new dependency was not allowed according to previously defined layering rules. After reporting this violation of the architecture, the project-lead was surprised and asked the developers for an explanation. The developers recognized the violation, and explained that it was introduced because a third module was not finished yet. As soon as this third module was finished, the dependency would be removed. All of this was documented in an evaluation report. Four months later, when the third module was finished, the developers were reminded of this undesired dependency and it was removed.

3.3.5 Technologies

The last category of LiSCIA assist the evaluator to evaluate the combination of the technologies used within the system. In addition, the questions in this category deal with the age, the usage and support for each of the used technologies. As an example, one of the questions asks whether the latest version of each technology is used. In many projects we have seen that this is not the case. The usual explanation for this is that there is no time to upgrade the system. In these situations, an action is defined to upgrade to the latest technology as soon as possible. This is to prevent the situation in which a legacy technology is used without an easy upgrade path. A different explanation for the same situation is that management decided to always use the second-last version of a technology, because this version has already proven itself in practice. In these cases, there is no action defined, but the explanation for not implementing the action is documented.

3.3.6 Result

The answers to all the questions and, if applicable, a list of actions is documented in the report of this phase. With this report, certain refactorings can be justified. In addition, the report provides a basic overview of the architecture as it is currently implemented. Because of this, the report can serve as a factual basis for discussions about the current architecture. Lastly, the report is used as input to the next evaluation session.

4 Discussion

Because LiSCIA is based on monitoring existing software systems, we were able to provide examples in the last section showing how the different parts of the method can be used to discover problems in an implemented architecture. Currently, we are applying the complete LiSCIA method as described in this article more and more in our current practice. The consultants who implemented the method thusfar gave positive reactions. They appreciated the structure of the methodology, as well as the type and the ordering of the questions. One of the consultants was particularly pleased because the straight-forward application of LiSCIA has, in his words, "mercilessly led to the discovery of the embedding of a forked open source system in the code."

Naturally, there were also points of critique. For example, one of the consultants stressed that LiSCIA should be applied by an expert outside the development team, pointing out that "... it is hard for a software engineer to remain objective when it concerns his own code". It is indeed true that LiSCIA relies heavily on the opinion of the evaluator. After all, the evaluator is the person who decides whether a given explanation is "good enough" in the setting of the project. Therefore we believe that it is a good idea to let a second expert examine the evaluation report, just to make sure that none of the explanations is flawed.

Different consultants also pointed out a second limitation of LiSCIA. They stated that the method relies on some of the functionality of our internal toolset, which might limit the usability outside of our company. This is a correct observation, since LiSCIA relies on tool-support to statically determine the size of units of the system and the dependencies between these units. However, we believe that these types of measurements can be done by freely available open source tools. The only investment needed is in the aggregation of the raw output of these tools, which is a relatively minor investment.

A third limitation of LiSCIA is the fact that it is only aimed to discover potential risks related to maintainability. Additionally, because LiSCIA uses only a single viewpoint to evaluate the architecture it is likely that it does not even cover all potential risks in this area.

These limitations are seen as one of the strengths of the method. First of all, we believe that when a system is not maintainable, dealing with other quality issues such as performance and reliability becomes more difficult. Because of this, a first focus on maintainability is justified. Also, the limited focus of LiSCIA allows for a precise and concrete design that can be easily implemented in current projects without too much effort. Moreover, LiSCIA is deliberately positioned as a light-weight check to ensure that it is seen as a stepping-stone towards more broader architecture evaluations. In addition, the collection of questions and actions reflect years of experience in conducting architectural evaluations, therefore we are confident that they cover the most important maintainability risks.

5 Related Work

5.1 Evaluating Architectures

Comparing LiSCIA against the architecture evaluation techniques mentioned by Babar [2] and Dobricia [8], there are several distinctions. First, where most of the methods are designed to evaluate the designed architecture, LiSCIA is specifically aimed at the implemented architecture. However, one can argue that these methods can be applied to a design of the architecture that is extracted from the implementation. While this is true, we believe that extracting the complete architecture from an implementation is difficult and labour intensive, which might eliminate the benefits of performing the architecture evaluation.

The most distinctive difference between LiSCIA and other available architecture evaluation methods is the fact that LiSCIA pre-defines a notion of quality, namely that of maintainability. Other available architecture evaluation methods, even the ones explicitly aimed at an implemented architecture such as [3], do not provide such a notion. Instead, virtually all methods contain a phase in which the notion of quality should be defined by the evaluators. As discussed before, this limits the use of LiSCIA to a specific purpose, but makes it easier to start with performing an architecture evaluation.

5.2 Software Erosion

Approaches for dealing with software erosion typically try to incorporate a solution into the design of the architecture. This approach helps in avoiding software erosion, but van Gurp et al. [15] conclude that *"even an optimal design strategy for the design phase does not deliver an optimal design"*. In other words, there is no way to completely avoid changes to an implemented architecture. Therefore, the architecture that is currently implemented should be taken into account when dealing with software erosion.

One approach that does this is the one proposed by Medvidovic [12]. In this approach, an evaluator first extracts the main components of a systems architecture and then maps these components to a notion of the "ideal" architecture for a system. Through this mapping, the evolution of the architecture can be controlled by first applying the desired change to the ideal architecture, after which the implemented architecture can be adapted.

The main difference between this (and similar) approaches and LiSCIA is the time at which erosion is dealt with. LiSCIA tries to detect erosion when it has actually happened, whereas other approaches try to prevent erosion from happening. Since these approaches are complementary, we believe that they are both useful and necessary. We realize that dealing with erosion after is has happened is more difficult and costly, but it is better to deal with erosion as soon as it has been introduced rather than when other issues need to be solved.

6 Conclusion

LiSCIA provides a lightweight sanity check to keep control over the erosion of an implemented architecture. A first introduction of LiSCIA within our company has received positive feedback. LiSCIA is simple to apply, and therefore will not provide the same depth as a full-fledged architecture evaluation. In spite of that, the results are useful and help in detecting software erosion.

We are currently evaluating the formal LiSCIA method by applying it in our current practice. In addition, we are very interested to see whether LiSCIA is useful in environments outside SIG. For this, we call upon you to try out LiSCIA and share the results with us. Combining our own experience with the feedback of the community we hope to report on an improved version of LiSCIA in the coming year. The complete LiSCIA method can be found online at:

`http://www.sig.eu/en/liscia`

Acknowledgements The authors would like to thank all the colleagues at the Software Improvement Group for their time and contributions to the making of the LiSCIA methodology and this article.

About the authors Eric Bouwers is a software engineer at the Software Improvement Group and a part-time PhD student at Delft University of Technology. He is interested in architectural and linguistic aspects of software quality. He can be reached at e.bouwers@sig.eu.

Arie van Deursen is a full professor in Software Engineering at Delft University of Technology, where he leads the Software Engineering Research Group. He can be reached at Arie.vanDeursen@tudelft.nl.

References

[1] M. Babar and I. Gorton. Software architecture review: The state of practice. *Computer*, 42(7):26–32, 2009.

[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*, page 309. IEEE Computer Society, 2004.

[3] P. Bengtsson and J. Bosch. Scenario-based software architecture reengineering. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 308. IEEE Computer Society, 1998.

[4] E. Bouwers, J. Visser, and A. v. Deursen. Criteria for the evaluation of implemented architectures. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 73–83. IEEE Computer Society, 2009.

[5] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA, 2003.

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

[7] A. v. Deursen and T. Kuipers. Source-based software risk assessment. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 385. IEEE Computer Society, 2003.

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

[9] 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.

[10] P. Kogut and P. Clements. The software architecture renaissance. *The Software Engineering Institute, Carnegie Mellon University*, 3:11–18, 1994.

[11] T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. In *Software Audit and Metrics*, pages 118–128. INSTICC Press, 2004.

[12] N. Medvidovic and V. Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2):225–256, 2006.

[13] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[14] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[15] J. van Gurp and J. Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

TUD-SERG-2010-003
ISSN 1872-5392