# Common Findings and Lessons Learned from Software Architecture and Design Analysis

Klaus Wolfmaier and Rudolf Ramler
*Software Competence Center Hagenberg GmbH*
*Hauptstrasse 99, A-4232 Hagenberg*
*{klaus.wolfmaier, rudolf.ramler}@scch.at*

## Abstract

*The foundation for any software system is its architecture. It defines the components and the relevant relations among them [1]. Software architecture and design analysis makes the quality of the architecture and design visible and palpable. This paper describes an approach we distilled from software architecture and design analysis in several industrial projects. It summarizes our experience and the lessons learned from analyzing software systems up to two million lines of code. The approach applies static analysis techniques and relies on metrics to identify problem areas.*

## 1. Introduction and Background

The Software Competence Center Hagenberg GmbH (SCCH) has conducted several architecture and design analyses of mid-sized to large software systems as a service for industrial projects. Our experience shows that architecture and design of the analyzed software systems have a critical impact on the overall success of all of the projects. With our analysis we provided an insight into the state and quality of the systems' architectures and designs, we revealed critical problems in respect of system requirements and we supported risk assessment and informed management decisions. The results of the analysis helped to answer questions like "Does the software system comply to the required standards?", "Has the defined design been implemented?" or "Can we extend the software system with reasonable effort?".

In this paper we describe the approach we developed for software architecture and design analysis and, furthermore, we summarize our experience and the lessons learned from analyzing several large and complex software systems. We illustrate the description of our approach by discussing common findings and sanitized analysis results.

The experience presented in this paper is based on the analysis of ten industrial software systems in the range from 100 KLOC to 2.000 KLOC. The systems are implemented in Java and C/C++ and cover various application domains, from business software to embedded systems. The average effort for conducting an analysis has been about 10 person-days for two engineers experienced in this kind of analysis and the necessary tools. The applied techniques rely mainly on static analysis of the system architecture and design. A set of metrics is used to initially identify problem areas that require further investigation [2]. Thereby we follow the conclusion that "Simply collecting one metric rarely gives us the answer – on the contrary, it usually gives us the question." [3]

The remainder of the paper is structured as follows: Section 2 describes our approach for software architecture and design analysis while Section 3 discusses our common findings that illustrate typical issues. In Section 4 we summarize our experience and lessons learned. Section 5 gives an outlook on our next steps to further evolve the presented approach.

## 2. Analysis Approach

The approach for architecture and design analysis has been compiled and matured based on the experience of analyzing several mid-sized to large software systems over the last two years. Hence, the approach is primarily designed for effectively and efficiently analyzing software systems starting at 100 KLOC up to 2.000 KLOC and we expect the approach to fit for larger systems as well.

The average effort for the analysis is approximately 8 to 12 person-days in a timeframe of two to three weeks, depending on the size of the system, its technical aspects, and the specific questions to be answered. For example, due to compiler- and language-specific variants and incompatibilities, analyzing systems im-

plemented in C/C++ is usually more challenging and time-consuming than analyzing systems of about the same size and complexity implemented in Java.

To adequately handle the involved technologies (e.g., EJB), libraries (e.g., STL), frameworks (e.g., Struts), as well as the domain-specific requirements (e.g., of embedded systems), an analysis is best conducted by two engineers working together: One domain expert and one quality engineer familiar with the applied tools and the approach.

The analysis contains four subsequent steps, ending with a workshop to present and discuss the analysis results. The findings are documented in a detailed analysis report that includes an overview of the system structure and basic measurements, an overview of the identified strengths and weaknesses, a condensed list of all identified issues, and an executive summary. However, depending on the requirements of the particular project, the approach and the analysis report can easily be customized. The following four steps describe our analysis approach.

**(1) Define objectives.** In the first step, together with the customer, architectural and technical aspects as well as the scope of the software system are identified, the questions to be addressed by the analysis are discussed, and the analysis objectives are defined. This first step also involves a meeting with system architects and developers, who provide technical background information about the system and its environment.

**(2) Set up environment.** In the second step, the system to be analyzed is set up and the required toolset is customized to the specific requirements of the analysis. Together with a system architect or developer, the subsystems and the intended system architecture are defined. Subsystems are aggregations of logical parts of the system with defined interfaces on an abstraction level above the levels provided by programming languages. The defined system architecture serves as the reference architecture for the architecture conformance check. A collection of metrics is selected to answer the specific questions of the analysis.

**(3) Explore and analyze.** In the third step, a high level analysis of the system is conducted. With the help of selected metrics, critical parts are identified and the analysis is drilled down to these parts if necessary. Furthermore, the implemented architecture is compared to the defined architecture to expose conformance violations. These violations are categorized based on the estimated impact on the system architecture and the results of the analysis are summarized in the report.

**(4) Transfer results.** A results workshop together with architects and developers of the system forms the fourth and final step of the analysis. As input for the workshop, the draft analysis report is sent to all workshop participants in advance. At the workshop the results of the architecture and design analysis are discussed and the root causes of the identified issues are examined. Optionally, a list of necessary improvement actions, derived from the analysis results, is developed. In general, the workshop discussions follow the structure of the analysis report.

## 3. Common Findings

In this section we describe our common findings that surfaced in the analyses of architectures and designs of different software systems. All the issues we discuss in this section have been found in more than half of the analyzed systems and had a severe impact on the quality of the architecture and design. In addition, each of these issues turned out to be a reliable starting point for digging for more specific problems in the analyzed system. The section is organized according the following categories of issues:

- Size limits
- Object-oriented concepts
- Cycles and bottlenecks
- Architecture conformance
- Coding guidelines

To support this approach, we have derived critical limits for common metrics that can be determined with the help of most static analysis tools. In Table 1 we present a selection of these metrics, which have proven useful as rule-of-thumb indicators to guide the effort estimation for the analysis and the detailed investigation of the system. Based on our experience, we list the typical (acceptable) range and the critical upper limit for each of these metrics.

| Metric | Typical Range | Critical Limit |
|---|---|---|
| RLOC/LOC | 50 % | n.a. |
| Classes/Package | 20 - 30 | 100 |
| Methods/Class | 5 - 10 | 25 |
| LOC/Class | 100 – 250 | 500 |
| LOC/Method | 10 – 25 | 100 |
| Parameter/Method | 1 - 5 | 10 |
| Classes in a cycle | 2 | 4 |
| Subsystems in a cycle | 0 | 1 |
| References to/from classes | 2 - 10 | 20 |
| References to/from subsystems | 0 - 4 | 10 |

**Table 1: Selected Metrics**

**Size limits.** From experience we know that a typical system has a ratio between relevant lines of code (RLOC) and total lines of code (LOC) of about 1 to 2, which is – generally speaking – the number of lines containing at least one code statement compared to the number of total lines. We use this ratio for a first estimation of the size of the software system and the effort of the analysis. We estimate the expected effort by comparing the numbers to historical data, whereby the ratio between RLOC and LOC is mainly used for adjusting the estimation. So far we have never encountered a system where the ratio between RLOC to LOC was considerably larger than 1 to 2. Some systems have a lower ratio, which is an indicator for a poorly structured and weakly commented code-base that increases the analysis effort.

Various other size metrics are used to gain a quick overview of problem areas and to verify our first estimation. In particular, we look at the number of classes per package, the number of methods per class, and the lines of code per class. Numbers that exceed critical upper limits (see Table 1) are subject to a further, detailed investigation.

Packages containing more than 100 classes do not necessarily indicate a problem, e.g., when most of the classes are anonymous inner classes. Nevertheless, we have found packages with more than 1.000 non-inner classes. In that case, the development suffered from change impacts, as there was no separation of stable and unstable system parts. Similarly, we have found large classes (up to 4.000 and 5.000 LOC). Since we consider classes with more than 500 LOC as critical, we go into detail. In many cases these classes suffer from lack of cohesion and exhibit a weak coverage by unit tests due to the complex and laborious test setup necessary.

By counting the number of packages and classes, which exceed the critical upper limits, we can estimate how much effort the detailed analysis requires and whether we have to adjust our initial estimation. Furthermore, critical items (e.g., packages or classes) that exceed one limit also tend to exceed other limits. For example, methods with many parameters require more validation code to ensure preconditions and have an increased LOC per method count. Hence, we prioritize the items for a detailed analysis by the number of metrics that exceed the critical limit.

**Object-oriented concepts.** Further indicators for problems are missing or decayed object-oriented concepts, e.g., violations of data encapsulation, inheritance or polymorphism. Useful metrics are the accumulated number of type conversions (type casts), the number of methods returning generic objects such as java.lang.Object in Java, write access to attributes in a class in a different package, classes with references to subclasses, or massive use of friend-classes in C++.

**Cycles and bottlenecks.** Violations at the architectural or design level are often indicated by cycles and bottlenecks. Cyclical relationships between classes or subsystems are critical as they usually cause ripple effects. However, cycles between classes are very common, so we investigate cycles only if four or more classes are involved. In general, cycles are the more critical the more items are involved or the higher the abstraction level of the involved items is. On the level of subsystems we therefore investigate all cycles.

Items are bottlenecks if they refer to many other items and are used by many other items. Changes to bottlenecks usually affect large parts of the system. In the case of bottlenecks, we consider classes that refer to more than 20 other classes and which are referred to by more than 20 other classes as critical. On the level of subsystems the critical limit is 10 referring and referred to subsystems.

**Architecture conformance.** In addition to general architecture and design violations, we provide architecture conformance checking [4] to analyze violations according to a reference architecture. Together with the architects and developers of the analyzed software system we define a reference-architecture, i.e., the initially planned architecture, and compare it to the implemented architecture. Any violation, e.g., the bypassing of a layer in a multi-tier architecture, has to be investigated as it has an impact on the system's ability to fulfill its requirements. So far, each software system we have checked for architecture conformance, contained critical violations. Typical are up references from a low-level class or sub-system to a high-level class, bypassing of a layer, and dependencies within a layer.

**Coding guidelines.** Finally, as a last check, we look for violations of low-level coding guidelines that are indicated by tools like Checkstyle [5], PClint [6], or PMD [7]. We use the aggregated number of violations as a rough quality indicator and compare the results with our analysis. The findings are usually limited unless coding guidelines have been applied in development or a customized set of guidelines is used to analyze the system. Although the amount of data is overwhelming (result files may exceed 700MB) without further processing, it provides little useful information.

## 4. Lessons Learned

Architecture and design analysis for industrial projects provided a rich source for lessons to be learned. In the

following we give a brief overview of insights on our approach and describe useful suggestions and limitations.

- The analysis approach is designed for mid-sized to large software systems from 100 KLOC up to 2.000 KLOC. The analysis of significantly larger systems is possible but requires more effort and time. For smaller systems, however, reviews of selected code parts together with the developers are more effective.
- It is absolutely necessary to involve the architects and developers of the analyzed system to acquire the necessary information about the system and its environment to conduct the analysis in the given time period.
- The analysis of low-level coding guidelines promises potential for identifying many different kinds of issues. Nevertheless, the effectiveness of such an analysis depends to a great extend on a customized set of coding guidelines. Otherwise, only the aggregated numbers of coding guideline violations may be a general indicator for problem areas.
- Tool support is vital to handle the large amount of data collected in analysis (e.g., to define filters for calculated metrics) and to visualize the collected data (e.g., call reference violations for subsystems). A tool that has proven particularly useful for analyzing large software systems is the Sotograph [8], by Software Tomograhpy, that we use together with related tools. For small systems, the main part of the analysis can be done with browsing and cross referencing support provided by state-of-the-art development environments.
- Generated code has to be treated independently from other parts of the system in order to avoid diluted metrics. Often, due to the code generator, the code is hardly readable or has an intricate structure to comply with code generation requirements.
- The analysis of Web or container-based applications is usually difficult as not all information is contained in the analyzed source code. The required information is spread over various source files, configuration files, glue code, etc. The described approach may not be suitable at all for this type of applications.
- Dynamic aspects do not play a prominent role as the proposed approach mainly relies on static analysis. However, many questions regarding the analyzed system can only be answered by applying dynamic analysis. As dynamic analysis takes more effort, it is – from our point of view –useful as part of follow-up analyses.

## 5. Summary and Future Work

In this paper we have presented an approach to analyze software architectures and designs. Over the past years the approach has been distilled from our experience of analyzing architectures and designs of software systems in numerous industrial projects up to 2.000 KLOC and in a wide variety of domains. We illustrated the approach by discussing common findings we derived and lessons we learned when establishing architecture and design analysis as a successful service for industrial projects.

As experience showed, large and complex software systems typically contain problems in their architecture and design. However, not all identified problems can be immediately fixed and, in some cases, it may even be preferable from an economic point of view to leave issues temporarily unresolved.

Nevertheless, as software systems evolve these issues bear the risk that the architecture and design deteriorates further and the implemented architecture diverges from the intended or documented architecture. It is therefore essential to identify, monitor and manage the state of all unresolved issues and their effect on the architecture and design in order to evolve the system without jeopardizing critical requirements.

To support the monitoring of known issues, future work includes the establishment of a periodical analysis of architectures and designs throughout the development process and the systematic identification of trends and early warning signals. This helps identifying issues early and prevents the deterioration of architecture and design.

## 6. References


[1] Clements, P., R. Kazman, and M. Klein: *Evaluating Software Architectures: Methods and Case Studies,* Addison Wesley, 2001.

[2] Ebert, C., R. Dumke, M. Bundschuh, and A. Schmietendorf: *Best Practices in Software Measurement.* Springer, 2004

[3] Armour, P.G.: *Beware of Counting LOC.* Communications of the ACM, 47(3), March 2004, pp. 21-24

[4] Bischofberger, W., J. Kühl, and S. Löffler: *Sotograph – A Pragmatic Approach to Source Code Architecture Conformance Checking.* Proc. of European Workshop on Software Architectures (EWSA), LNCS 3047, Springer, 2004

[5] Checkstyle: http://checkstyle.sourceforge.net/

[6] PCLint: http://www.gimpel.com/html/pcl.htm

[7] PMD: http://pmd.sourceforge.net/

[8] Sotograph: http://www.sotograph.com/