Stopping (and reversing) the architectural erosion of software systems. An industrial case study

Bernhard Merkle
Research & Development, Software Engineering
SICK AG Waldkirch, Germany
Bernhard.Merkle@sick.de
Bernhard.Merkle@gmail.com

Abstract: During the evolution of a software system it becomes more and more difficult to understand the originally planned software architecture. An architectural erosion happens for various reasons during the development phases.

In this paper we describe an approach to stop and reverse architectural degeneration of software systems. Using this method, the software architecture can even be improved, especially on the long term.

Many projects suffer a loss of the initial software architecture over the project lifetime and end up with a nearly unmanageable code base. Symptoms are e.g. undocumented software architectures, unwanted or unknown dependencies, and an unknown number of changes. In the end such systems often end like the "big ball of mud" and the only solution is to throw them away and start again from scratch. In most cases however one can not afford to lose the investment in the current code base, therefore it is essential to keep an eye on the architecture and monitor it on a regular basis. Recently several tools have become available to accomplish and automate this task and we were able to apply it to a critical project.

SICK AG is the worldwide leading provider of sensor solutions, especially laser scanners. We describe a real case study where we were able to stop and even reverse the architectural erosion.

1 Introduction

This paper describes a case study where we have been able to stop the architectural erosion of a large software system. We first give a short overview in which the terms of the software architecture area are defined and the main problem is outlined, which can manifest itself through different facets. In the following chapter we present a case study: a real software system developed by SICK AG, which suffered from an architectural erosion and was difficult to maintain and further extend.
2 Software Architecture

IEEE 1471-2000 [IEEE 2000] defines software architecture as “The fundamental organization of a system, embodied in its components, their relationship to each other and the environment, and the principles governing its design and evolution”. However two major problems can be found in lots of projects:

- Relationships are insufficiently defined: This means only the “main” relationships are defined in most projects, and they are defined in a very coarse manner, e.g. there is usually no differentiation between required, forbidden and optional relationships.

- Design and Evolution: The initial design is often covered and documented as mentioned above, but most problems arise with the evolution of the software (and hence the architecture).

3 Software Architecture Analysis

Software Architecture Analysis (SAA) can be applied for different purposes: e.g, Checking architectural consistency, Rating, Quantifying, Reviewing an architecture, Discovering an architecture, Measuring (real) facts, Monitoring changes.

3.1 Checking architectural consistency

The general approach is as follows: On the one hand the desired architecture with the requirements is described as the “should”-architecture. On the other hand an “is”-architecture exists which is extracted from the current code base. Via corresponding tool support a difference report can be generated, that shows or enumerates where the “should”-architecture and “is”-architecture differ, e.g. forbidden upward calls in a layered architecture from lower layers to upper layers can be detected.

3.2 Applying SAA on an industrial application: A case study

The Java application of our case study consists of about 300 kLOC and grew over about 6 years. Initially the software architecture was well defined, source code and architecture corresponded and the modules were designed in most cases as independent plug-ins. However, meanwhile the architecture degradation proceeded over the years and the modules were now tied together very closely. Most violations were calls from lower layers to upper layers and were able to remove them via moving the relevant classes to higher level packages. The problem with bidirectional dependencies was also resolved. The SAA-Tool displays the number of dependencies for each direction. A typical pattern in our software was a high number of dependencies in one direction and a lower number in the opposite direction.