Reengineering Automotive Software at Bosch

Jochen Quante

Robert Bosch GmbH
Corporate Sector Research and Advance Engineering Software
P. O. Box 30 02 40, 70442 Stuttgart, Germany

Jochen.Quante@de.bosch.com

Abstract

Software Reengineering has been a topic at Bosch for 10 years now. This paper reports about our experiences during this time, about ongoing reengineering activities, and about potential topics for the research community.

1 Introduction

It is a well-known fact that software ages [5]. Nearly 30 years ago, Lehman and Belady noted that software has to change continuously in order to remain useful, and that a software's complexity increases as the software is changed [3]. The combination of these "Lehman's laws" means that complexity is steadily increasing – if no counteractive measures are taken.

Lehman's laws are valid as well for automotive software. Software is becoming more and more important in this domain, which means a big change since the domain is traditionally very hardware-oriented. Software enables new security, comfort, and economic functions which allow manufacturers to set themselves apart from competitors. For example, an engine control system may consist of up to one million lines of C code. However, the increasing amount of software in automobiles comes along with an increasing complexity, rising time and cost pressure, and high quality demands. This necessitates measures for controlling and limiting this ever-increasing complexity.

To account for this fact, a research project was established at Bosch in 2002. Its goal was to identify best practices and patterns that are applied in automotive software development and make them accessible to all development units within Bosch. For those units that did not yet use them, this meant reengineering their code such that they incorporate the new patterns. Consequently, these patterns had to be accompanied by tools and methods that help in this transformation process.

2 Reengineering Phase Model

SEI's horseshoe model is quite well-known in reengineering [1]. At Bosch, we established a similar software reengineering phase model. The basic reconstruction and construction steps are surrounded by two additional phases. The model thus consists of four phases:

1. Identification of modules with unnecessarily high implementation complexity.
2. Reconstruction of functionality and requirements of the module.
3. Construction of a new solution that implements the same basic functionality, but applies best practices and patterns for its design.
4. Verification and validation of the new solution.

In the following sections, the four phases are discussed in more detail.

Identification

At first, one has to identify those modules that bear the highest potential for improvement. Often, developers have a good feeling about which modules should be regarded. However, an automated approach that identifies those "hot spots" is desirable. To achieve this, we evaluated the "Maintainability Index" by Oman and Hagemeister [4], which is based on correlating metrics with expert opinions on maintainability. The result was that the index indicates modules of bad maintainability with a precision of 50% and 83% recall [6], which is considered "good enough" by our business units. However, we did not only find positive "best practice" patterns in the code, but also negative patterns. Such "bad smell patterns" can be automatically detected, and the results are used to improve the maintainability rating.[1] The different indicators are then integrated using a software quality model.

Reconstruction

Before constructing a new solution, the existing solution has to be understood first. This is often a laborious task, specially when the software has grown for

[1] Unfortunately, research is mostly focused on bad smell patterns in object-oriented programs, whereas smells in C code are hardly covered.

decades and when many different people have manipulated its code. This part takes a lot of time – specially when you first have to learn the domain knowledge that is necessary for the respective module. Support for the understanding of such modules is therefore highly desirable, since it can save a lot of time and money.

Existing approaches mostly focus on the "big picture", like the extraction of architectural views. There exist only few approaches that help in understanding a single module or function. Because of their dataflow oriented nature in our domain, even single modules can be hard to comprehend. Different views on a single module could help with this issue. Possible examples for that include annotation and abstraction techniques, data-flow views on imperative code, and hiding or highlighting parts of code or models. There appears to be a wide range of research opportunities in this area.

Construction

The basic result of the search for patterns in automotive software is that things are easier to understand the better they follow the principle of separation of concerns. However, the news here is not that separation of concerns improves the maintainability of software – this fact has been known for a long time. What we experienced is that the application of this principle is also possible in resource-restricted environments, and that it can even reduce resource usage. Therefore, the most important aspect in creating a new solution is to improve the separation of concerns without spending additional resources. In our dataflow-oriented software, this often means introducing state machines for separating control flow from data flow, or identifying and separating the essence of a function from the rest.

Verification and Validation

The last step is done by performing the standard module and integration tests.

3 Experiences

We made the experience that it costs a lot of effort to reengineer even a single module. You need to build up a deep understanding of the module's implementation, and also of the relevant domain knowledge. This confirms Fjedstad and Hamlens finding that 50% of all maintenance effort is spent for understanding the program to be changed [2]. Therefore, our current activities aim at developing tools and methods that better support program comprehension (also see Section 2). In particular, our focus is to create methods and tools that are tailored to the specialities of our domain [7].

One difficult aspect of reengineering is the proof of its cost effectiveness. It is very hard to show that a reengineering will be beneficial before it has been performed, and it is as difficult to prove that it was worth the effort afterwards. However, a high number of successful reengineerings seems to convince people that it does in fact pay off.

Another experience is that reengineering does not only improve the code's readability, but also its resource usage. We observed a noticeable reduction of resource usage in most cases. In some cases, the memory usage could be reduced by up to 50%. This is an effect that is better noticeable and measurable than the long-term effect of improved maintainability, and it helps a lot to convince people to invest in reengineering activities.

4 Transfer

Reengineering has been a topic at Bosch Corporate Research for many years now. A lot of concrete reengineerings were performed in collaboration with the business units. Meanwhile, the business units themselves recognize the usefulness or even inevitability of reengineering and perform reengineering on their own. Along with an ongoing Software Reengineering qualification program, this helps to further improve the quality of our software and control its ever-growing complexity.

References

[1] J. Bergey, D. Smith, N. Weiderman, and S. Woods. Options analysis for reengineering (OAR): Issues and conceptual approach, 1999. Technical Note CMU/SEI-99-TN-014.

[2] R. K. Fjedstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *Proc. of the GUIDE 48*, 1979.

[3] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change.* Academic Press, 1985.

[4] P. W. Oman and J. R. Hagemeister. Constructing and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3), 1994.

[5] D. L. Parnas. Software aging. In *Proc. of 16th ICSE*, pages 279–287, 1994.

[6] J. Quante, T. Grundler, and A. Thums. Maintainability index revisited: Adaption and evaluation for Bosch automotive software. In *3. Workshop zur Software-Qualitätsmodellierung und -bewertung*, Feb. 2010. TUM-I1001.

[7] V. Schulte-Coerne, A. Thums, and J. Quante. Automotive software: Characteristics and reengineering challenges. *Softwaretechnik-Trends*, 29(2), May 2009.