Reengineering Required: Quality Deficits of Industrial Software[*]

Nils Göde

CQSE GmbH
Lichtenbergstr. 8, 85748 Garching bei München, Germany
goede@cqse.eu

Abstract

A lack of comprehension, obstacles to changing the source code, and a high number of defects increase the costs of developing and maintaining software. Unawareness or ignorance of existing quality deficits may boost development costs and result in situations where an expensive rewrite or complete abandonment are the only viable options. This paper summarizes recurring quality problems that we often find in industrial software during our quality audits. These problems highlight areas where reengineering is required and continuous quality control is advisable.

1 Introduction

During our activity as quality auditors, we get to know a lot of different software systems. Our goal is to identify quality deficits within these systems that already hinder the current maintenance or may become obstacles for future development. And although each system is unique with respect to requirements and its environment, we identified a number of recurring quality deficits that we find in almost every system. It is worth being aware of these problems, because if they are not solved, they naturally get worse over time and may have severe consequences.

In the remainder of this paper we share our experiences from analyzing a variety of industrial projects. We describe the major quality deficits we commonly observe during our audits. The descriptions can be used as a checklist and serve as starting points for reengineering activity. The information can also be used to identify areas for future reengineering research.

2 Quality Deficits

The list of quality deficits described in this section is not exhaustive. That is, systems may suffer from other problems apart from those described here. In addition, the order of the descriptions does not have any meaning.

[*]This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

Redundancy. Duplicated code occupies the first place in Fowler's "stink parade of bad smells" [1]. And in fact, we find a notable number of duplicated code fragments—*code clones*—in every system. Copy-and-paste programming is not a rare phenomenon but the usual practice. Consequently, most systems contain between 20% to 40% of redundant code. Individual systems contain more than 50% redundant code.

During our quality audits, we observe two of the major negative consequences of code clones. First, we find a lot of clones that evolved consistently during the system's history. These duplicated code fragments require additional effort, because changes need to be propagated to all copies. Second, unwanted inconsistencies are often bugs that have been fixed in one place but not in the copied code fragments [3].

Code Anomalies. Nowadays, there are static analysis tools for many programming languages that automatically locate suspicious code fragments and identify bugs. In the context of Java, *PMD* and *FindBugs* are among the most prominent. But although these tools are used within most development teams, the results are often ignored.

Depending on the configuration, these tools deliver code anomalies of different severity. While some are only style issues, others influence the resource usage, and yet others reflect incorrect behavior. While a lot of these findings may not require immediate attention, those with high severity should be cared for as soon as possible. It is not uncommon that we find dereferences of invalid pointers or infinite loops in productive software using these static analysis tools.

Error Handling. A consistent handling or errors and exceptions is important for a system to recover from unexpected situations and provide appropriate information to identify the cause of the error. Unfortunately, error and exception handling is inconsistent and incomplete in most systems. In C++ systems, for example, we find a lot of inconsistency in whether an error is dealt with by throwing an exception or by the return values of functions. In addition, it is often unclear in which situations, which information is logged to which destination.

In Java systems, there is often no clear decision of whether checked or unchecked exceptions are used.

Apart from that, exceptions are often dealt with by catching the top-level exception classes *Exception* or even *Throwable*. This has high risk of also unintentionally catching and hiding other errors, e.g., null-pointer dereferences or stack overflows due to an infinite recursion. Paired with a large number of code anomalies, inappropriate error handling is a severe problem that may lead to incomprehensible system behavior and make errors untraceable.

Architecture. While every software system certainly has an implicit implemented architecture, there is often no common understanding and no explicit specification of that architecture. This causes a number of problems. For example, there is no guideline of how things should be implemented. An architecture specification should provide directions for how certain things are to be implemented. Apart from that, the architecture cannot be discussed and evaluated without a specification. The lack of a specification also prevents an architecture conformance analysis where the specified architecture is compared to the implemented architecture.

In contrast to a lack of specification, there are also projects where the specification is too detailed. The problem of having too much detail is that the specification requires frequent changes and is likely to get outdated very soon. We also observed that only parts of a system are specified when the level of detail is very high. Large parts of the system remain unspecified.

Documentation. Another problem is a lack of documentation—external as well as internal. In many cases, there is no description of the system and its structure. This includes, for example, an appropriate specification of the architecture. The lack of external documentation increases the time needed to understand the system or locate relevant parts for a given maintenance task. This is especially true for new developers that join the team.

Internal documentation refers to the comments that are contained within the source code to describe the system. In many cases, the source code does not contain any comments at all. Thus, it is often unclear what the purose of a particular class is or what a given method does. In other cases, the amount of comments is misinterpreted as the quality of the documentation. This leads to comments that contain mostly redundant information.

3 Continuous Reengineering

None of the problems described in the previous section occurs from one day to the other. The majority of quality deficits evolves in small steps and slowly gets worse over time. Although this fact is widely known, hardly any countermeasures are taken in practice.

One option to prevent this decay is following a simple policy, which ensures that none of the problems gets worse when the system is changed. If followed strictly, the quality will inevitably stop getting worse. Implementing such a policy requires to differentiate between old legacy problems and new problems that were caused by the latest changes. One approach to implemented such a policy is *delta analysis*, described in our earlier work [2].

Based on such a policy, the next step is to clean up code fragments that are changed as part of a change request. These small reengineering activities help to continuously increase the quality of the software. The limited scope allows to clean up with reasonable effort. The extra reengineering effort is small since these code fragments need to be understood and tested anyway.

4 Larger Reengineering Projects

Not all quality improvements can be broken down into small steps that can be done as part of the daily maintenance work. In such cases, larger reengineering projects have to be carried out. It it not surprising that development teams that invest more time in continuous reengineering are less often faced with larger reengineering projects.

In any case, it is important to have an appropriate level of test coverage to ensure that no new problems are introduced by the reengineering activity. We find that development teams with a sophisticated regression test infrastructure are much more likely to carry out reengineering projects compared to teams without appropriate test coverage.

5 Conclusion

Despite the diversity of software systems, there are common quality deficits, which we find in most systems. The major problems are a high level of redundancy, many code anomalies, inconsistent error handling, no explicit architecture specification, and inappropriate documentation. All of these problems can also be seen as starting points for reengineering activity to improve the quality.

Needless to say that our results cannot automatically be generalized, because we have seen only a limited number of systems. In addition, there is an infinite number of peculiarities and each system may suffer from problems that are much worse than those described in this paper. However, we still think the common problems are worth knowing about as they point out where future reengineering research is needed. They can also serve as a checklist to test one's own system for likely quality deficits.

References

[1] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[2] N. Göde and F. Deissenboeck. Delta analysis. *Softwaretechnik-Trends*, 32(2), 2012.

[3] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.