Design Erosion in Evolving Software Products

Jilles van Gurp, Jan Bosch, Sjaak Brinkkemper

University of Groningen, Vrije Universiteit Amsterdam

{jilles|jan.bosch}@cs.rug.nl, Sjaak@cs.vu.nl

Abstract. *Design erosion affects most, if not all, software systems. As these systems age, it becomes ever more difficult to make new changes until eventually it is more feasible to replace (or at least refactor) the software than it is to continue to the regular maintenance. In earlier work we have already identified a number of potential causes for this phenomenon. The case study presented in this paper, examines two eroded subsystems of a large software product. We look at various aspects of how the company involved has identified that the systems were eroded and how they managed to recover from that situation.*

1. Introduction

In this paper, we present the preliminary results of two case studies, which were conducted on two subsystems within the same company. Due to the preliminary and confidential nature of the case study and its results, we will not elaborate any further on the domain of the software or nature of the company involved in this paper. However, a full paper that will include these details is nearing completion.

For the moment, it is enough to specify that the company involved is a large multinational that, for the past few decades, has developed a large software product, which has been deployed on numerous (thousands) of customer sites worldwide. The software product, this company makes, consists of a number of application modules and an infrastructure layer that is common to these application modules. In the first case study, we examined the evolution of a component in the infrastructure layer. In the second case study, one of the application modules was examined. The purpose of the case studies was to explore the problems and issues encountered in large software developing organizations, such as the company involved in this study, with respect to design erosion.

Design erosion is a problem that affects most, if not all, large software systems. The phenomenon is also known as architectural drift [5], software aging [6] or architecture erosion [4]. Essentially the problem is that as software evolves, the software is incrementally changed to meet new requirements, fix defects or optimize quality attributes (adaptive, corrective and perfective maintenance [8]). However, these requirements may conflict with requirements in earlier iterations or may change the assumptions under which design decisions in earlier iterations were made. When faced with such requirement conflicts, there are two strategies for adapting the system to incorporate the changes:

- **An optimal design strategy.** No compromises are made with respect to design quality and the design of the software is enhanced in such a way that the new requirements can be incorporated without compromising the design integrity. While this strategy typically results in a good design, the associated cost may make it infeasible for some changes.
- **A minimal effort strategy.** Often complicated design changes can be avoided by stretching the design rules of the existing design a bit. While this may have consequences for the quality of the design, this strategy can be very effective in meeting the requirements on short notice.

In [3], we concluded that it is inevitable that in real world systems the first strategy is not always feasible. Consequently, cost considerations or time constraints sometimes force developers to take less than ideal design decisions. Over time, these less than ideal design decisions accumulate, resulting in what we call design erosion. Eroded software systems are typically hard to understand due to the many sub-optimal design solutions that have accumulated and complicated the design. Consequently, additional changes become harder and eventually may even become infeasible. When this happens, the only ways to resolve the situation are to either repair (e.g. using refactoring techniques) or replace the software. Both types of resolutions typically require a significant effort. In [3] we list a number of real-world projects that were affected by design erosion. In these examples, the subsequent effort to repair/replace the software spanned several years.

In a world that is increasingly relying on a growing quantity of ever-larger software, design erosion presents a serious problem. Affected software cannot be easily replaced or repaired. Failing to do so, however, may cause maintenance cost to rise and limits the flexibility of the affected software. Ultimately, eroded software may threaten the existence of the company that produces it as well as the existence of companies that use the software.

The cases we report on in this case-study, concern software subsystems that are part of a large software system that have both been affected by design erosion to such an extent that in both cases, the company chose to undertake an effort to address the issues, which in both cases implied several person-years of work. In one of the cases, this effort involved the refactoring of tens of thousands of lines of code. In the other case, the affected component had to be replaced by a new one to address the issues. The old version, representing a decade of evolutionary development and refinements, had to be discarded.

In the remainder of this paper, we will first discuss the research questions of this case study and our research method. After that, we will present some preliminary conclusions of the case-study. As outlined above, at this point, we cannot go into detail on the case studies themselves, however.

2. Research questions

The focus of our study is to explore how design erosion issues are identified, resolved and prevented in software developing organizations. Specifically, our study addresses the following research questions:

- **Symptoms.** What are the effects of design erosion on a system?
- **Identification.** How does an organization decide that their software is eroding and needs to be repaired? How does the decision process work?
- **Causes.** What are common causes for erosion?
- **Resolution.** What kinds of solutions are applied to fix an eroded system? How and when are decisions with respect to preservation and repair taken?
- **Prevention.** What practices help prevent erosion?

3. Methodology

In this section, we will outline the empirical research approach we have applied in the case studies and discuss its strengths and weaknesses. In his editorial for the journal of empirical software engineering [1], Victor Basili makes a plea for the use of empirical studies to validate theories and models that are the result of software engineering research. In a more recent publication, [2], Basili presents an overview of how empirical research has benefited NASA's Software Engineering Lab. When doing empirical research, a distinction can be made between qualitative empirical studies and quantitative studies. The approach advocated by Basili in [1] and [2], can be characterized as mostly quantitative. As can be seen in [2], collecting quantitative data is a labor-intensive process that needs to be tightly integrated with the development process. In a setting like NASA, where reliable, dependable software is required this is feasible. The results of the quantitative empirical research are used to optimize the development processes. However, in many other contexts this is much less feasible.

Qualitative data, on the other hand, is relatively easy to obtain and has the advantage of providing more explanatory information [7], which in an exploratory case study such as ours is very desirable. As is noted in [7], neither quantitative nor qualitative empirical research can prove a given hypothesis. Empirical research can only be used to support or refute a given hypothesis. A combination of both quantitative and qualitative studies is the best way of supporting a hypothesis [7].

In this exploratory case study, we use interviews as the primary tool of retrieving information. Consequently, our research is mostly of a qualitative nature. However, where possible, we complement the qualitative data with quantitative data provided by the interviewees (e.g. estimated defect rates, number of lines of code, etc.). Due to the confidentiality of such metrics within the company, a full quantitative study was not feasible. We have found that in general, software development organizations are very reluctant in providing or publishing such data.

In both case studies, the interviews followed the same pattern. We first met with the interviewees (software engineers, product architects, project managers) in a group for an introductory meeting. During this meeting, the purpose of the case study was communicated and a brainstorm session was held to select appropriate modules/components for further study. This meeting was also used for planning subsequent interviews. In the following meetings, both group and individual interviews were held during which more specific questions about the design and evolution of the system were asked.

In addition to interviews, we were given access to various documents including for example functional designs and requirements documentation. Using these documents, we were able to both verify/clarify certain statements of the interviewees as well as prepare specific questions in advance.

3.1 Case selection

Throughout both case studies, we have cooperated with the company's R&D department who were very much interested in the results of the case study for the sake of (a) providing an outsider analysis on the architecting and engineering practices, and (b) educating the product architects and software engineers with the results. Using their expertise and knowledge of the company's product portfolio, two representative sub-systems were selected for further study and contacts with staff working on these sub-systems were initiated. Before selecting the cases, we had several meetings with the R&D department during which we discussed the organizational structure, the company's product architecture and the goals for the case study. In addition, an estimate of the time that was needed for both cases was made.

We used the following criteria for the selection of the cases:

- The systems had to be old enough to have endured design evolution.
- During the evolution, there must have been significant changes in the requirements.
- It should be possible to interview both people who were involved in the initial development of the system and people who were involved in restructuring the system for new requirements.

3.2 Validity

To ensure the correctness of our data and conclusions, we have used two methods:

- **Cross-checking.** In both cases, we interviewed multiple developers. This allowed us to compare their answers and verify whether there were any contradictions. In both cases we were also given limited access to software documentation, which allowed us further validate the information we received.
- **Feedback.** An important part of qualitative research is feedback. The data presented in this article consists mostly of our interpretation of interviews. Verifying whether this interpretation is correct is therefore an essential part of ensuring the validity of our case study. After each meeting, a report detailing our conclusions and interpretation was communicated back without the interviewees for feedback. The feedback has made us confident that the interviewees share our interpretation and conclusions.

However, there are a number of problems with our research approach that may affect the validity of our findings:

- **Representativeness of the cases.** By limiting ourselves to one company and one software product, we risk that this case study's conclusions may not be applicable to other domains and companies. Both the corporate culture and the domain this particular company is operating in affect our conclusions. However, based on our experience with case-studies in other companies, the corporate culture in this company is representative for many software developing companies. In addition, despite coming from the same company, the two cases we selected are dissimilar, so, any conclusions that can be generalized for these two cases may be applicable to other domains as well.
- **Quantitative data.** As explained earlier, we use a (mostly) qualitative approach. Complementing our data with quantitative metrics would certainly strengthen our conclusions. However, there are a few reasons why this study does so only to a limited extent. First of all, many relevant metrics that would need to be collected are generally considered as sensitive information in software development organizations. Consequently, we did not have access to raw quantitative data. However, the company does collect metrics and provided some qualitative information regarding e.g. defects to us that was based on this data. Additionally, this is an exploratory study. A quantitative study requires a more precise formulation of hypotheses, relevant quantifiable parameters and a model for the interpretation of values for these parameters. A study such as presented here may provide the necessary input formulating hypotheses and parameters for future quantitative studies.
- **Cases are not comparable.** We have deliberately chosen to research two cases from different domains to show that identification, resolution and prevention of design erosion works the same across domains. Therefore, both cases use different types of technology and involve people with different skills and training. On the other hand, both teams operate in a centrally managed release development project to design and build the sub-systems as part of one product. This makes it possible to compare the results of both case studies, notwithstanding some limitations.

4. Results & observations

In this section, we present the answers we found to the five research questions in the introduction in both our case studies. While we cannot go into much detail on either of the case studies, it is worthwhile to outline them in an abstract fashion.

- Case 1 examined the evolution of an infrastructure component that had evolved in a number of versions. In each version, significant architectural changes were made to this component. Recently, based on an internal evaluation it was decided to replace this component with a new component because the old one had eroded so much that repairing it and adding new features was no longer feasible.

- Case 2 concerns an application module that was originally designed at the request of a particular customer. After an initial design project, the realization phase was handled by a relatively inexperienced development team. However, the resulting software had all sorts of problems. Eventually, the development was transferred to a more experienced team. This team subsequently decided to refactor and restructure the software.

As mentioned in the introduction, a full paper with much more detail is pending. In the remainder of this section, we will simply refer to them as case 1 and case 2.

4.1 Symptoms

A first step in preserving the design of a software system is to recognize the symptoms of an eroding system. Both cases we examined, exhibited similar symptoms of deterioration:

- **Low quality code.** In both cases, the developers working with the system were unhappy with the quality of the source code. They complained about misuse of language constructs, the lack of structure, inconsistent use of code standards, etc.

- **Uncertainty about specifications.** There was a great deal of uncertainty about the specification of the system in both cases. The designs were sketchy and incomplete. In the case 1, application developers actually depended on unspecified and even incorrect behavior of the infrastructure component. In case 2, changes were not properly documented (as prescribed in the companies development processes), effectively making the existing designs obsolete.

- **Regressions.** In both cases, fixes for defects often introduced new problems. Particularly in case 1, where at one point there were about 100 known defects, this was an important reason for discarding the old software. The estimated cost of fixing these 100 defects in combination with the near certainty of additional defects provided enough motivation for doing so.

- **Deployment problems.** In both cases, there were problems with respect to the usage of the system. In case 1, developers of application modules were relying on the unspecified, arguably incorrect, behavior of the component whereas in case 2, the functional design was no longer accurate because design changes were not documented.

- **Defect rates & cost.** An interesting aspect about the development process in the company is that it includes a fine-grained process for measuring defect rates and relating defects to particular development artifacts. In both cases, the developers we interviewed indicated that the amount of defects that needed to be fixed was substantially higher than in comparable systems.

4.2 Identification

In order to repair an eroded system, it has to be recognized first that the system is eroded and that it is worthwhile to undertake an effort to repair it. Obviously, in the systems we examined, the developers came to this conclusion. A number of factors may play a role in identifying erosion:

- **Evaluation**. In both cases, the decision to redevelop/redesign the system was taken after an internal evaluation of the software. In both cases these evaluations were prompted by problems with the existing software and a general feeling the software was not in a good condition (e.g. because of the symptoms outlined above). Additionally, in both cases, the defect rates that are routinely collected within this company were abnormally high, which provided additional evidence that both software systems had quality problems.
- **New requirements**. New requirements may call for enhancements that, given the quality of the system at that point, are infeasible. In both cases, it was the case that there were new requirements that were proving to be hard to realize in the existing systems.
- **Change of staff**. Developers, like most human beings, may be reluctant in admitting their own faults. In both cases, the developers that identified the erosion and took the initiative for the redevelopment of the software had not been involved in the original development of the software.
- **Defect Metrics.** In both cases, defect metrics played an important role. The development process includes a fine-grained process for collecting such metrics and the decision to redesign (case 1) or refactor (case 2) was partially based on these metrics.

4.3 Causes

In order to effectively repair an eroded system, the causes of the issues that are responsible for the erosion need to be understood. . We have found that both cases had a number of common issues. Consequently, these issues are also likely to share the same causes:

- **Vaporized design decisions**. In both cases, all or most of the original developers were either no longer working on the system or had left the company entirely. Consequently, many of the design decisions taken early in the evolution of both systems were poorly understood. Particularly the maintenance of case 1 became more problematic after the person who designed this component left the company. In the other case, the designers were on a different continent than the people who were involved in the realization phase.

- **Too little attention to design during evolution**. During the evolution of a system, changes may occur that require that the software design is altered. In both cases, we found that little attention to the design was paid during the evolution. In case 1 several, major design changes had taken place during its evolution. The resulting software had become extremely complex. In case 2, time-pressure had caused developers to bypass the proper process for defect fixing (which includes documenting the changes and designing a fix).
- **Quick fixes**. During the evolution of a system, defects are found and fixed (in [8] this is called corrective maintenance). The proper way to fix a defect is to analyze the defect, design a solution, implement and test the solution. Unfortunately, time-pressure or cost considerations may prevent developers to properly follow this process. Often this results in quick fixes that addresses the issues but that may also introduce additional issues. Especially in case 2, it was identified that the existing process for processing change requests (which is the common way for fixing defects) had not always been followed. In case 1 the design was so out of date that developers did not bother to update it anymore.
- **Experience.** An interesting aspect in case 2 was that the development of the software was transferred a number of times. One of the first development groups was relatively inexperienced and consequently, the quality of their work was relatively low. The lack of experience with development and the internal development processes probably was an important reason for the problems that surfaced once the development was transferred to a more experienced team.
- **Time pressure.** In the two cases we examined, two components of the same software product. While the components of this product are developed separately, their development must be synchronized with the release cycles of the product. Consequently, if a particular change cannot be realized in the timeframe between two product releases, problems may arise. The time-pressure associated with these releases was an important factor in the initial development of case 2. In order to make the release, certain things were rushed an parts of the code were incomplete.

4.4 Resolution

Once it has been determined that a system is eroded, and once causes have been identified, an attempt can be made to repair the system and prevent further damage. The obvious things that can be done and that we have observed in both cases are:

- **Redevelopment**. Redevelopment of the software is often the only real option in fixing an eroded system. This approach was chosen in case 1. Interestingly this decision was taken based on an estimate of the cost of fixing all the known defects (about 100).
- **Restructuring**. The people working on case 2, on the other hand, chose to restructure the existing system and reserved a significant amount of time for it. As in case 1, this decision was based on a cost estimation.
- **Strong focus on design**. As pointed out earlier, the lack of up to date designs is usually one of the problems with eroded systems. In both cases, recovering/updating the designs was an integral part of the attempt to address the problems and key to the success of the whole operation.
- **Modularization and object orientation**. In both cases, the developers complained about the fact that the source code was in bad shape and that there were many dependencies between the various modules and components in the system. In both cases object oriented like mechanisms such as encapsulation, information hiding and delegation were applied to improve the structure of the system.
- **Take product release cycles into account.** As argued earlier, the development of individual subsystems, such as the two cases we are dealing with, must be synchronized with the product release cycles. Typically, changes are projected at a particular release and there is little room for delays. Consequently, it must be possible to make the necessary changes within that timeframe. If not, an option is to break down the work. This happened in case 1 where the new component was planned and delivered in two releases. In case 2, one of the problems was that the developers adopted some quick fixes in order to be able to integrate their software in the product in time for the product release.

4.5 Prevention

The developers of the systems we examined in this paper have experienced first hand what it takes to recover a deteriorated system. Naturally, they made an effort to learn from the experience to adapt the way they develop software in such away that future problems can be avoided. In the cases we examined a number of practices were adopted that appear to be successful:

- **Automatic regression testing**. In order to prevent that new defects are introduced during defect fixing, automated tests can be used to verify that the system still works. Regressions were particularly a problem in case 1. Therefore, the developers adopted the practice of creating automatic tests while they were redeveloping their component. By the time this component was finished, a test suite of 800 tests was available. Also, the defect metrics showed that there were almost no regressions during the maintenance of the new system.
- **No undocumented fixes**. Both cases shared the problem that in the past there had been undocumented changes. This both makes it hard to test the software and to use it correctly (this was a problem in case 1). To address this, all changes are now documented properly. Also, in case 1, test cases are made to ensure that the software works as advertised in the documentation. Any deviation from the specified behavior is considered as a defect now.
- **Stronger focus on process**. Part of the problems in case 2, and to a lesser extent case 1, can be attributed to the fact that the existing development process was not enforced. This caused all sorts of problems the processes were designed to prevent.

5. Concluding remarks

In this position paper, we have briefly discussed the results of two case-studies. As discussed in the introduction, a full paper including details on how and where the results outlined here were obtained is pending.

An important conclusion of our earlier work was that design erosion is inevitable. Consequently, our case study did not focus on how to prevent design erosion but on effective strategies for dealing with design erosion. Both software systems in the two cases we discuss in this paper are part of a software product, which has existed in several versions. The company involved identified that there were problems with these subsystems and successfully addressed these issues without causing any delays in the product release schedule. In other words, the process of identifying and resolving design erosion works reasonably well in this company.

An interesting aspect of this case study is that, in addition to the technical factors identified in our earlier study [3], there are also a number of non-technical factors that contribute to design erosion. For example, in case 2, an important factor was that the existing development process was not enforced. Consequently, any measures for resolving or preventing design erosion also have to consider these non-technical factors.

Based on what we have seen in this case study and in other software systems, we are strengthened in our belief that design erosion is inevitable. Software developing organizations should not be judged by how effective they are in preventing design erosion but in how effective they are in identifying and resolving eroded software components.

In future work we will present more details about the case study presented here. Additionally, we intend to write a ‘best practices’ paper.

6. References

[1] V. Basili, “Editorial”, Journal of Empirical Software Engineering, Vol 1. no. 2, 1996.

[2] V. Basili, F. E. McGarry, R. Pajerski, M. V. Zelkowitz, “Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Lab“, proceedings of ICSE 2002, pp. 69-79, 2002.

[3] J. van Gurp, J. Bosch, “Design Erosion: Problems & Causes”, Journal of Systems & Software, 61(2), pp. 105-119, Elsevier, March 2002.

[4] C. B. Jaktman, J. Leaney, M. Liu, “Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study”, in Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), 1999.

[5] D. L. Parnas, “Software Aging”, in Proceedings of ICSE 1994, 1994.

[6] D.E. Perry, A. L. Wolf, “Foundations for the Study of Software Architecture”, in ACM SIGSOFT Software Engineering Notes, vol 17 no 4, 1992..

[7] C.B. Seaman, “Qualitative Methods in Empirical Studies of Software Engineering“, IEEE Transactions of Software Engineering, 25(4), pp. 557-572, 1999.

[8] E. B. Swanson, “The dimensions of maintenance“, proceedings of the 2nd international conference on software engineering, pp. 492-497, IEEE Computer Society Press, Los Alamitos 1976.