The Journal of Systems and Software 85 (2012) 132–151

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Controlling software architecture erosion: A survey

Lakshitha de Silva*, Dharini Balasubramaniam

School of Computer Science, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SX, UK

ARTICLE INFO

Article history:
Received 30 June 2010
Received in revised form 20 July 2011
Accepted 21 July 2011
Available online 27 July 2011

Keywords:
Software architecture
Architecture erosion
Design erosion
Software decay
Controlling architecture erosion
Survey

ABSTRACT

Software architectures capture the most significant properties and design constraints of software systems. Thus, modifications to a system that violate its architectural principles can degrade system performance and shorten its useful lifetime. As the potential frequency and scale of software adaptations increase to meet rapidly changing requirements and business conditions, controlling such architecture erosion becomes an important concern for software architects and developers. This paper presents a survey of techniques and technologies that have been proposed over the years either to prevent architecture erosion or to detect and restore architectures that have been eroded. These approaches, which include tools, techniques and processes, are primarily classified into three generic categories that attempt to *minimise*, *prevent* and *repair* architecture erosion. Within these broad categories, each approach is further broken down reflecting the high-level strategies adopted to tackle erosion. These are: process-oriented architecture conformance, architecture evolution management, architecture design enforcement, architecture to implementation linkage, self-adaptation and architecture restoration techniques consisting of recovery, discovery and reconciliation. Some of these strategies contain sub-categories under which survey results are presented.

We discuss the merits and weaknesses of each strategy and argue that no single strategy can address the problem of erosion. Further, we explore the possibility of combining strategies and present a case for further work in developing a holistic framework for controlling architecture erosion.

1. Introduction

Software systems are under constant pressure to adapt to changing requirements, technologies and social landscapes. At the same time these systems must continue to deliver acceptable levels of performance to users. Often, modifications made to a software system over a period of time damage its structural integrity and violate its design principles. As a result the system may exhibit a tendency towards diminishing returns as further enhancements are made. Such software is no longer useful for its intended purpose nor is it economically viable to maintain. Eroded software often goes through a process of re-engineering, though this may not always yield the expected benefits. The alternative is to build a replacement system from scratch, which clearly would require a sizeable investment. Moreover, software have become key assets in organisations that sell them as well as in those which use them. Ensuring these systems continue to perform as expected over long periods of time is vital for the sustainability of these organisations.

Software erosion is not a new concept. Parnas (1992) argues that software aging is inevitable but nevertheless can be controlled or even reversed. He highlights the causes of software aging as obsolescence, incompetent maintenance engineering work and effects of residual bugs in long running systems. However, later works in this area such as those carried out by Huang et al. (1995) and Grottke et al. (2008) define software aging as the gradual degradation of performance in executing software processes due to changes in the runtime state (e.g. memory leaks). In this paper we regard erosion as the overall deterioration of the *engineering quality* of a software system during its evolution (see Section 2.2). Erosion is often a contributory factor to the kind of software aging studied by Huang et al. and Grottke et al.

The impact of erosion is profound when the damage affects the architecture of a software system. Software architecture (Perry and Wolf, 1992; Shaw and Garlan, 1996) establishes a crucial foundation for the systematic development and evolution of software and forms a cycle of influence with the organisation to which the system belongs (Bass et al., 2003). It provides a high level model of the structure and behaviour of a system in terms of its constituent elements and their interactions with one another as well as with their operating environment. Architecture also encompasses rationale, which forms the basis for the reasoning and intent of its designers (Perry and Wolf, 1992).

* Corresponding author. Tel: +44 1334 463253; fax: +44 1334 463253.
E-mail addresses: lakshitha.desilva@acm.org (L. de Silva), dharini@cs.st-andrews.ac.uk (D. Balasubramaniam).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.07.036

In this work we focus on *architecture erosion* which possibly plays the biggest role in accelerating software erosion. Architecture erosion is usually the result of modifications to a system that disregard its fundamental architectural rules. Although a single violation is unlikely to produce an adverse effect on the system, the accumulation of such changes over time can eventually create a mismatch between the implemented software and its architecture. The effects of architecture erosion tend to be systemwide and, therefore, harder to rectify. Other forms of software degeneration such as inferior quality code are typically easier to repair.

Architecture erosion and its effects are widely discussed in literature. Perry and Wolf (1992) differentiate *architecture erosion* from *architecture drift* as follows: erosion results from violating architectural principles while drift is caused by insensitivity to the architecture. As the underlying causes for both are the same, we will not consider this difference for the purpose of our survey. Additionally, the notion of software architecture erosion is discussed using a number of different terms such as architectural degeneration (Hochstein and Lindvall, 2005), software erosion (Dalgarno, 2009), design erosion (van Gurp and Bosch, 2002), architectural decay (Riaz et al., 2009), design decay (Izurieta and Bieman, 2007), code decay (Eick et al., 2001; Stringfellow et al., 2006) and software entropy (Jacobson, 1992). Although some of these terms imply that erosion occurs at different levels of abstraction (for instance code decay may potentially be considered insignificant at the architectural level), the underlying view in each discussion is that software degeneration is a consequence of changes that violate design principles.

Mechanisms for controlling architecture erosion have been traditionally centred on architecture repair as evident from the large body of published work (e.g. Harris et al., 1995; Bellay and Gall, 1997; Gannod and Cheng, 1999). Architecture repair typically involves using reverse engineering techniques to extract the implemented architecture from source artefacts (recovery), hypothesising its intended architecture (discovery) and applying fixes to the eroded parts of the implementation (reconciliation). Subsequent research in this area focuses more on erosion prevention schemes (e.g. Mae (Roshandel et al., 2004) and ArchJava (Aldrich et al., 2002)) and explores concepts from other areas of computer science such as artificial intelligence to increase the accuracy of architecture discovery and recovery techniques (e.g. Bayesian learning-based recovery (Maqbool and Babri, 2007)).

This paper presents a classification of strategies for controlling architecture erosion and a survey of currently available approaches under each category in the classification. Section 2 of the paper describes architecture erosion with the aid of a few industrial examples and briefly discusses related work in the form of other surveys with a similar aim. Section 3 introduces the classification scheme while Sections 4–9 present the survey results under this classification. For each approach, we provide evidence of adoption where available, and a discussion of efficacy and cost–benefit analysis based on our own experience and material from literature. In Section 10 we discuss some of the factors that may influence the selection of an erosion control strategy along with possibilities for using strategies in combination with one another. In conclusion, Section 11 outlines the state-of-the-art with respect to current strategies and presents some thoughts on future work.

2. Background

In this section we provide the context and motivation for the rest of the paper. Based on industrial case studies, we recognise that architecture erosion is often an inevitable outcome of the complexity of modern software systems and current software engineering practices, requiring approaches for controlling erosion at different stages of the software life cycle. We also introduce the terminology used in the remainder of the article and build a case for this survey by highlighting the strengths and drawbacks of previous survey attempts.

2.1. Context

The deterioration of software systems over time has been widely discussed since the late 60s debate on the "software crisis" (Naur and Randell, 1969; Dijkstra, 1972). Evolving software systems gradually become more complex and harder to maintain unless deliberate attempts are made to reduce this complexity (Lehman, 1996). At the same time, these software systems have to be continually upgraded to adapt to changing domain models, accommodate new user requirements and maintain acceptable levels of performance (Lehman, 1996). Therefore, complexity becomes a necessary evil to prevent software from becoming obsolete too soon.

Complexity, however, makes it harder to understand and change a design, leading to programmers making engineering decisions that damage the architectural integrity of the system. Eventually, the accumulation of architectural violations can make the software completely untenable. Lack of rigorous design documentation and poor understanding of design fundamentals make a complex system even harder to maintain (Parnas, 1992). Furthermore, software architectures that have not been designed to accommodate change tend to erode sooner (Parnas, 1992).

Architecture erosion can also result from modern software engineering practices. Architectural mismatches can arise in component-based software engineering (CBSE) due to assumptions that reusable components make about their host environment (Garlan et al., 1995). New challenges in CBSE like trust, reconfigurability and dependability create enormous demands on the architectures of evolving software systems (Garlan et al., 2009). In addition, modern iterative software development processes (such as agile programming methods) may cause the occurrence of architecture erosion sooner rather than later because they place less emphasis on upfront architectural design (van Gurp and Bosch, 2002).

A number of case studies indicate that architecture erosion is widespread in the industry. Eick et al. (2001) present a study of a large, 15-year old telecommunication software system developed in C/C++. They derive a set of indices from change request data to measure the extent of erosion and its impact on the system. Although termed *code decay*, the module level changes have architectural level impact. The study shows a clear relationship between erosion and an increased effort to implement changes, an increased number of induced defects during changes, increased coupling and reduced modularity. In another commonly cited example of architecture erosion, Godfrey and Lee (2000) describe their analysis of the extracted architectures of the Mozilla web browser (which subsequently evolved into Firefox) and the VIM text editor. Both these software products showed a large number of undesirable interdependencies among their core subsystems. In fact, the badly eroded architecture of Mozilla caused significant delays in the release of the product and forced developers to rewrite some of its core modules from scratch (van Gurp and Bosch, 2002; van Gurp et al., 2005). Other similar findings have been reported on popular open source projects such as FindBugs (Sutton, 2008), Ant (Dalgarno, 2009) and version 2.4 of the Linux kernel (van Gurp and Bosch, 2002).

The impact of architecture erosion is far reaching and has an associated cost. In the worst case an eroded software system that is not salvageable requires complete re-development. Even if a system does not become unusable, erosion makes software more susceptible to defects, incurs high maintenance costs, degrades performance and, of course, leads to more erosion. Consequently, the system may lose its value, usefulness, technical dominance and