Information and Software Technology 52 (2010) 31–51

Contents lists available at ScienceDirect

Information and Software Technology

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

Characterizing software architecture changes: A systematic review

Byron J. Williams [a], Jeffrey C. Carver [b,*]

[a] Department of Computer Science and Engineering, Mississippi State University, United States
[b] Department of Computer Science, University of Alabama, Box 870290, 101 Houser Hall, Tuscaloosa, AL 35487-0290, United States

A R T I C L E I N F O

Article history:
Received 10 December 2008
Received in revised form 9 July 2009
Accepted 13 July 2009
Available online 18 July 2009

Keywords:
Software architecture
Software maintenance
Change characterization
Software evolution
Systematic review
Software changes

A B S T R A C T

With today's ever increasing demands on software, software developers must produce software that can be changed without the risk of degrading the software architecture. One way to address software changes is to characterize their causes and effects. A software change characterization mechanism allows developers to characterize the effects of a change using different criteria, e.g. the cause of the change, the type of change that needs to be made, and the part of the system where the change must take place. This information then can be used to illustrate the potential impact of the change. This paper presents a systematic literature review of software architecture change characteristics. The results of this systematic review were used to create the Software Architecture Change Characterization Scheme (SACCS). This report addresses key areas involved in making changes to software architecture. SACCS's purpose is to identify the characteristics of a software change that will have an impact on the high-level software architecture.

Contents

* Corresponding author. Tel.: +1 205 348 9829; fax: +1 250 348 0219.
E-mail addresses: bjw1@cse.msstate.edu (B.J. Williams), carver@cs.ua.edu (J.C. Carver).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.07.002

1. Introduction

Software change is inevitable. All software systems must evolve to meet the ever-expanding needs of its users. Therefore, it is vital for organizations to perform software maintenance in such a way as to reduce complications arising from changes and the potential for new bugs to be introduced by the change. Software developers need a comprehensive solution that helps them understand changes and their impact. This understanding is important because, as changes are made architectural complexity tends to increase, which will likely result in an increase in the number of bugs introduced [50,65,114]. *Architectural complexity* measures the extent to which the behavior of one component can affect the behavior of other components from an architectural standpoint [16]. A complex system is potentially less understandable for developers resulting in decreased quality and a system that is more difficult to maintain [13]. *Software quality* is the degree to which software possesses a desired combination of quality attributes [1]. Due to the number and frequency of changes that mature systems undergo, software maintenance has been regarded as the most expensive phase of the software lifecycle.

Late-lifecycle changes are changes that occur after at least one cycle of the development process has been completed and a working version of the system exists. These unavoidable changes pose an especially high risk for developers. Understanding late-lifecycle changes is important because of their high cost, both in money and effort, especially when they are due to requirements changes. Furthermore, these late-lifecycle changes tend to be the most crucial changes because they are the result of better understood customer and end-user needs. As these late changes are made, system complexity tends to increase. Different names have been given to this phenomenon of increasing complexity. Eick, et al., called the problem *code decay*. They examined a 15-year old system and found that it became much harder to change over time. One cause of this decay was the violation of the original architectural design of the system [50]. Lindvall, et al., called the problem *architectural degeneration*. This term is used to describe the deviation of the architecture They found that even for small systems the architecture must be restructured when the difficulty of making a change becomes disproportionately large relative to its size [93]. Parnas used the term *software aging* to identify the increased complexity and degraded structure of a system. He noted that a degraded structure or architecture would increase the number of bugs introduced during incremental changes [114]. And finally, Brooks stated that "all repairs tend to destroy the structure, to increase the entropy and disorder of the system…more and more time is spent on fixing flaws introduced by earlier fixes" [35].

Flexibility is a quality property of the system that defines the extent in which the system allows for unplanned modifications [116]. Flexibility is reduced when late changes draw the system away from its original design. There are many sources of late-lifecycle changes including: defect repair, adapting to changing market conditions or software environments, and evolving user requirements. Due to the time pressure resulting from these crucial late-lifecycle changes, developers often cannot fully evaluate the architectural impact of each change. As a result, the architecture degrades (i.e., becomes increasingly difficult to change), escalating the likelihood of faults and the difficulty of making future changes [50,84,114].

When dealing with late-lifecycle changes, it is important to focus on the *software architecture*, a high-level representation that defines the major structure and interactions of the internal components of a system and the interactions between the system and its environment [59]. When a change affects the architecture, the original architectural model must be updated to ensure that the system remains flexible and continues to function as originally designed. When an architectural change causes the interactions to become more complex, which in turn causes the system to be more difficult to change, the architecture is degenerating. *Architectural degeneration* is a mismatch between the actual functions of the system and its original design. Because architectural degeneration is confusing for developers, the system must undergo either a major reengineering effort or face early retirement [65].

To address these problems, developers need a way to better understand the effects of a change prior to making it [100]. The high-level goal of this research is to:

Identify and characterize the types of changes that affect software and develop a framework for analysis and understanding of change requests.

This paper presents a systematic literature review of software change. The goal of the review was to identify and characterize software architecture changes to determine the types of changes that impact software architectures. A systematic literature review is a formalized, repeatable process in which researchers systematically search a body of literature to document the state of knowledge on a particular subject. A systematic review provides the researchers with more confidence in their conclusions compared with an ad hoc review. A needs assessment conducted prior to the review indicated several key areas that must be addressed to improve the software change process.

- *Change understanding and architecture analysis*: Prior to making a change, it is important for a software developer to understand how it will impact the architecture. A change analysis tool