The Journal of Systems and Software 85 (2012) 1682–1698

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

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

Coding-error based defects in enterprise resource planning software: Prevention, discovery, elimination and mitigation

Isaac Woungang[a], Felix O. Akinladejo[b], David W. White[b], Mohammad S. Obaidat[c,*], Fellow of IEEE, Fellow of CS

[a] Department of Computer Science, Ryerson University, Toronto, ON, Canada
[b] Faculty of Engineering and Computing, University of Technology, Jamaica
[c] Department of Computer Science and Software Engineering, Monmouth University, West Long Branch, USA

ARTICLE INFO

Article history:
Received 8 August 2011
Received in revised form 14 February 2012
Accepted 15 February 2012
Available online 10 March 2012

Keywords:
Defect density
Coding defects
ERP
Software development
Defect reduction
Code auditing
Static code analysis
Software testing

ABSTRACT

Software defects due to coding errors continue to plague the industry with disastrous impact, especially in the enterprise application software category. Identifying how much of these defects are specifically due to coding errors is a challenging problem. In this paper, we investigate the best methods for preventing new coding defects in enterprise resource planning (ERP) software, and discovering and fixing existing coding defects. A large-scale survey-based ex-post-facto study coupled with experiments involving static code analysis tools on both sample code and real-life million lines of code open-source ERP software were conducted for such purpose. The survey-based methodology consisted of respondents who had experience developing ERP software. This research sought to determine if software defects could be merely mitigated or totally eliminated, and what supporting policies, procedures and infrastructure were needed to remedy the problem. In this paper, we introduce a hypothetical framework developed to address our research questions, the hypotheses we have conjectured, the research methodology we have used, and the data analysis methods used to validate the stated hypotheses. Our study revealed that: (a) the best way for ERP developers to discover coding-error based defects in existing programs is to choose an appropriate programming language; perform a combination of manual and automated code auditing, static code analysis, and formal test case design, execution and analysis, (b) the most effective ways to mitigate defects in an ERP system is to track the defect densities in the ERP software, fix the defects found, perform regression testing, and update the resulting defect density statistics, and (c) the impact of epistemological and legal commitments on the defect densities of ERP systems is inconclusive.

We feel that our proposed model has the potential to vastly improve the quality of ERP and other similar software by reducing the coding-error defects, and recommend that future research aimed at testing the model in actual production environments.

1. Introduction

Enterprise resource planning software (ERP) belongs to a class of large-scale software called enterprise application software. Enterprise applications are complex software systems that are expensive to build, debug and maintain (Rettig, 2007). They were first conceptualized during the decade of the 1970s but could not be built because of the limitations in the hardware and programming languages of that era. By the decade of the 1980s however, four-tenths of all companies earning over one billion United States dollars had implemented some sort of enterprise application (Markus and Tanis, 2011). In 2006, the enterprise application software market was valued at US$100 billion, but software failures were estimated to cost the industry between US$20 billion and US$50 billion. These failures were not limited to coding defects, but coding defects nevertheless are a significant concern for enterprise software in general. On one hand, efforts have been made to find practical ways to verify that ERP and other types of software are free of defects. On the other hand, full software verification for software with large code bases such as ERP software has been proven to belong to the class of problems considered undecidable and theoretically intractable.

In this paper, we attempt to develop a practical model for preventing, discovering, fixing and reducing defects in enterprise software. Though our recommendations are applicable to enterprise software in general, we focus on ERP software. The overall

* Corresponding author.
E-mail addresses: iwoungan@scs.ryerson.ca (I. Woungang), fakinladejo@utech.edu.jm (F.O. Akinladejo), dwwhite@utech.edu.jm (D.W. White), obaidat@monmouth.edu, m_obaidat@yahoo.com (M.S. Obaidat).

goal of the paper is to improve the understanding of preventing, discovering, fixing and reducing defects in enterprise software. Specifically, we have attempted to address the following research questions (RQs):

- *RQ1. What are the key programming techniques that exist, which prevent software defects in ERP?*
- *RQ2. What are the best methodologies existing to discover software defects in ERP?*
- *RQ3. What are the most effective ways to mitigate software defects once they are discovered in ERP?*
- *RQ4. To what extent can software defects be eliminated in ERP?*
- *RQ5. How do epistemological and legal commitments impact the level of defects in ERP?*

We have conducted a review of the literature to determine the attributes that could affect the defect density of ERP software, and used these attributes to formulate a set of hypotheses and develop a theoretical framework. We then conducted a survey-based statistical study of ERP software developers internationally and a series of experiments involving static code analysis tools on both sample code and real-life million lines of code open-source ERP software to determine if each of our hypotheses should be rejected or accepted. We used the results from these to revise our model highlighting the main factors affecting ERP software defect density.

The rest of the paper is organized as follows. Section 2 summarizes some related works, the nature of ERP software; techniques used to discover, track and reduce defects; testing methodologies and the use of static code analysis tools. In Section 3, we hypothesize various factors that affect ERP software. Section 4 describes our research methods. Section 5 describes the results of the survey and experiments. Section 6 discusses our research findings and highlights some future research directions.

2. Related work

2.1. History of ERP software

Markus and Tanis (2011) theorized that in the decade of the 1970s, although there was a vision to create enterprise application software to integrate individual information systems within organizations, computing capacity and programming language capabilities were too limited to cause such applications to be built. Instead, organizations constructed “islands of automation”, meaning that a new application was built for each individual information processing need within the same organization. By the 1980s however, the vision of enterprise applications begun to be realized, although somewhat slowly, as organizations were able to interface the individual information systems into a homogeneous one. Markus and Tanis (2011) also indicated that the mid-1990s was a turning point for the development and adoption of enterprise application software since more and more organizations embraced the client-server architecture over the traditional and comparatively expensive mainframe architecture. They also stated that by the year 1998, approximately 40% of all organizations with annual revenues of more than one billion dollars had implemented some form of enterprise application software. In Pearlman (2007) and O’Leary (2000), it was stated that enterprise application software were complex software applications. In Currie (2006), it was further noticed that enterprise application software were not only complex, but also prone to high defect rates.

2.2. Programming languages

Application software such as ERP were developed using computer programming languages. In Louden (2003), programming languages were described as notational systems for describing computation in both machine-readable and human-readable form. The history of programming languages was explored in Beer (2004), where the author traced the development of the first programming languages in the early 1950s up to the objected-oriented and scripting languages of the early 2000s. Both Louden (2003) and Stegmann (1979) described the drawback faced by languages based on the Von Newman architecture (so-called the Von Neumann bottleneck). In TIOBE Software (2011), programming languages were generally categorized into statically and dynamically typed, and further, into object-oriented, procedural, functional and logical. Also highlighted was their correlation to mathematical theory which allows the behavior of a computer programs to be described abstractly and precisely, enabling a determination of whether or not it would run correctly, and allow very concise code to be written for very complex tasks. Louden (2003) stated that the object-oriented programming had been very successful in allowing programs to be written using reusable and extensible code which mimics the behavior of objects in the real world, however, such programs were harder to understand and describe abstractly making it harder to predict their behavior and determine their correctness. Despite this, Louden (2003) stated that object-oriented programming languages were becoming the languages of choice in developing application software. The language popularity index in TIOBE Software (2011) revealed that the top five languages (in the order of popularity) as of November 2010 were: Java, C, C++, PHP, and C#, three of which were object-oriented. The alternative language popularity index in SourceForge (2011) lists the world’s top five languages as of February, 2011 as: C, Java, PHP, BASIC, and C#. By correlating both lists, it was determined that the world’s most popular programming languages were Java, C, PHP, and C#, not necessarily in that order.

Louden (2003) and Beer (2004) seemed to suggest that there might be a relationship between the choice of programming language and the possibility of bugs being present in the software being developed. According to (Phipps, 1999), a normal C++ program can have between two to three times as many bugs per line of code as a normal Java program, can generate between 15% and 50% more defects per line of code, and could take more time to debug compared to the normal Java program (in average $6t$ where t is the time taken by a normal Java program). However, there were other authors (Candea et al., 2004; Kaelin, 2011) who believed that no matter what programming language was chosen, software bugs or errors were inevitable.

2.3. Complexity and customization

In Fowler (2003), it was argued that the complexity of software makes it harder to change. During the Turing Award Lecture in 1980, C. A. R. Hoare, co-developer of a number of programming languages, opined that there were two ways to construct software: either making it so simple that there is obviously no defect or making it so complex that there are no obvious defects (Hoare, 1981). In 1976, McCabe introduced a discourse on measuring and reducing the complexity of programs, where graph theory was used to map, analyze and reduce program complexity. It was stated that reducing the complexity of a program could not guarantee a program to be completely bug-free, but would instead “surface more bugs and improve the quality of the software.” Despite McCabe’s arguments (McCabe, 1976) and the wide adoption of his complexity measure, Fenton and Melton (1990) presented a mathematical formulation in which it was not clear that one module X, with a higher McCabe