International Journal of Information Management xxx (2016) xxx–xxx

Contents lists available at ScienceDirect

International Journal of Information Management

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

The application of knowledge management to software evolution

José Braga de Vasconcelos [a,*], Chris Kimble [b], Paulo Carreteiro [a], Álvaro Rocha [c]

[a] Knowledge Management and Software Engineering Research Group, Universidade Atlântica, Fábrica da Pólvora de Barcarena, 2730-036 Barcarena, Portugal
[b] KEDGE Business School Rue Antoine Bourdelle, Domaine de Luminy, BP 921 13288 Marseille, Cedex 9, France
[c] Department of Informatics Engineering, Universidade de Coimbra, Pólo II—Pinhal de Marrocos, 3030-290 Coimbra, Portugal

ARTICLE INFO

Article history:
Received 7 February 2016
Received in revised form 1 April 2016
Accepted 23 April 2016
Available online xxx

Keywords:
Software engineering
Knowledge management
Collaborative work
Software maintenance
Software development process

ABSTRACT

In complex software development projects, consistent planning and communication between the stakeholders is crucial for effective collaboration across the different stages in software construction. Taking the view of software development and maintenance as being part of the broader phenomenon of software evolution, this paper argues that the adoption of knowledge management practices in software engineering would improve both software construction and more particularly software maintenance. The research work presents a guidance model for both areas: knowledge management and software engineering, combining insights across corporate software projects as a means of evaluating the effects on people and organization, technology, workflows and processes.

1. Introduction

The "software crisis" (the claim that software development projects are always over budget, behind schedule, and unreliable) has been a feature of software engineering since the early 1970s. The cost of maintaining software has historically, and continues to be, seen as a major component of the cost of software engineering projects, with estimates ranging from 50% to 90% of the total cost of a project (Koskinen, 2003). In this article, we will look at the application of knowledge management (KM) techniques to the problems of producing reliable, cost efficient software in general and the problems of software maintenance in particular. To do this we adopt the position of Lehman (Lehman, 1979, 1996; Lehman & Ramil, 2003) that software maintenance simply one aspect of the broader and inescapable phenomenon of software evolution. Consequently, in line with Anquetil et al. (2007), we do not claim that the knowledge requirements for software maintenance are significantly different from those of software development, but we do argue that the nature of software maintenance poses certain difficulties for the management of that knowledge.

Software maintenance is an activity based around maintaining what are essentially legacy systems, with all that entails. Software development may take several months, but software maintenance may last for many years. Similarly, the working environment of maintenance engineers (e.g. the programming language, the database management system, the data model, the system architecture) have all been dictated by decisions that were taken based on constraints that may have changed radically, forcing them to work with obsolete tools and techniques and to deal with various forms of undocumented fixes and work-arounds. While a software development a developer will have immediate access to the design requirements of the system, maintenance engineers may only have a vague knowledge of those requirements (Anquetil, de Oliveira, de Sousa, & Dias, 2007).

From the description above it might appear that much of the knowledge needed to maintain systems, such as a deep understanding of the domain and the problems that are encountered there, could best be described as tacit knowledge, which is notoriously difficult to capture and store (Nonaka & von Krogh, 2009). Nevertheless, the approach we propose in this article is one based on the storage and retrieval of codified knowledge from a database. Our assertion is that he problem essentially that of a "lost codebook" (Cowan, David, & Foray, 2000) which states that if knowledge has already been articulated and has been recorded in some form then, even although the knowledge that underpins the original categorization has been 'lost', it should, in principle, be possible to recover it. The framework we describe here uses this approach by back-flushing relevant information from the documentation of the earlier segments of the systems development lifecycle and

* Corresponding author.
E-mail addresses: jose.braga.vasconcelos@uatlantica.pt (J.B. de Vasconcelos), chris.kimble@kedgebs.com (C. Kimble), pcarreteiro@uatlantica.pt (P. Carreteiro), amrocha@dei.uc.pt (Á. Rocha).

http://dx.doi.org/10.1016/j.ijinfomgt.2016.05.005
0268-4012/

making it available to those who are later tasked with maintaining the system.

The rest of the article is structured as follows. We begin by examining the software engineering and the nature of software maintenance in greater detail. We do this mainly by drawing on the work of Lehman (Lehman, 1979, 1996; Lehman & Ramil, 2003) on software evolution and examining its implications for KM. We follow this by examining the KM strategies that could be used to address this problem. Here we draw a distinction between the type of knowledge needed to be an effective systems maintenance engineer and the knowledge needed to maintain specific systems, and focus on the issue of how a strategy of codification could best be used to deal with the latter. Then we examine the specific needs of knowledge management systems (KMS) and ask, "What kind of KMS is needed to help software maintenance knowledge workers?T̈ollowing this, we present the MIMIR Framework and end with some conclusions regarding our research's key focus and related contributions.

2. Software engineering and software maintenance

The origin of software engineering lies in the search for solutions to the problems associated with software development that began to emerge in the 1960s. The term 'software crisis', which became a shorthand for the observation that software seemed to take too long to develop and required extensive (and expensive) modifications after delivery, was coined in first NATO Software Engineering Conference in 1968 (Bryant, 2000). In time, techniques and technologies, such as high-level programming languages and structured design methodologies, were developed to make the production of software more efficient and less error prone but, despite this, maintenance cost seemed to remain stubbornly high. Lehman (1980) claims that in 1977, 70% of the total cost of a system was accounted for by maintenance, almost 20 years later Pigoski (1996) was still able claim that up to 80% of the cost of information systems was down to maintenance. Although there is considerable scope for argument about exactly how costs should be measured, Lehman and others argue that, to some extent, the concern about maintenance cost is misplaced and that the whole notion of maintenance as it is applied to software needs to be re-evaluated.

Lehman (1980) observes that in mechanical systems, the term maintenance is generally used to describe the restoration of something to its former state: deterioration has occurred due to wear and tear, and is corrected by the repair or replacement of a component. However, software does not suffer from wear and tear and continues to function in the same way until it is changed; maintenance therefore involves a change away from the previous state rather than a restoration of it. Similarly, in mechanical systems, major changes to a product are only achieved by redesign, retooling, and the construction of an entirely new model, whereas with programs changes can be superimposed on an existing system without the need to redesign the system as a whole. Software only changes when people decide that the current behavior is wrong or inappropriate; furthermore, such problems can be identified and corrected in any phase of the life cycle, not only in the so called maintenance phase. Change is intrinsic to the nature of software and consequently it more accurate to talk of 'software evolution' than of software development or maintenance. This of course is not an argument that software maintenance costs should be ignored but rather that the focus should be on reducing the unit cost of change and minimizing the rate of increase as the system ages.

Although many of Lehman's arguments are abstract and theoretical, much of his work is based on first hand observation of software projects; this becomes particularly relevant when considering how development and maintenance are actually carried out. For Lehman software evolution is the process of keeping software synchronized with its social, legal, and organizational environment, consequently, the impact of managerial and organizational artefacts such as the partitioning of software development into the phases of the systems lifecycle assume a crucial importance.

Managers typically tend to concentrate on the successful completion of their current projects, as their success is usually judged by immediately observable results such as cost and timeliness. Managerial strategies will therefore inevitably be dominated by a desire to achieve a local outcome with visible short-term benefits; they will not take into account long-term effects which cannot be easily predicted and whose cost cannot be accurately assessed. Thus, the temptation is to make changes in an ad-hoc fashion, one upon another, rather than group them together and implement them in coherent manner. During development, this tendency is counteracted by partitioning the work into distinct phases, but systems maintenance is often event driven and reactive; changes may be localized and tailored to meet specific needs, while recommended patches and system updates may be ignored or only partially implemented leading to unforeseen problems later on.

The effect of these observations in terms of KM is three fold.

Firstly, the apparent compatibility between the types of knowledge used in systems development and systems maintenance appears to bode well for KM as it implies that one knowledge schema might be able to be used for both activities, however, the practice of systems development and maintenance tends to undermine this view.

Secondly, the piecemeal, reactive, and sometimes chaotic nature of software maintenance is problematical for an approach based on the systematic capture, storage, and reuse of knowledge. While systems development methodologies provide a centralized discipline to structure and aid the documentation what happened during the creation of the software, maintenance is often carried out independently, and under pressure, at the local level; capturing knowledge for later reuse is likely to be seen as a low priority.

Finally, from a more theoretical viewpoint, while it is undoubtedly of value to the effective management of systems development, dividing systems development into phases inevitably introduces a discontinuity into the process. Systems development becomes a progressive series of mappings of needs in the real world onto a set of specifications for actions to be performed by a machine (Blum & Sigillito, 1985; Lyytinen, 1987). As Samuelis (2008) notes, each discontinuity represents a new level of abstraction, and with each level of abstraction, knowledge is lost: design decisions that were taken at higher levels cannot be reconstructed or predicted from the abstractions that exist at lower levels. This problem exists regardless of approach taken: structured methods such as SSADM tend to abstract away process specific details, while object oriented methods tend to abstract away implementation specific details (King & Kimble, 2004).

3. Knowledge management for software maintenance

From the forgoing discussion, we can see that using KM for software maintenance appear to be viable as long as (a) the exigencies of the task of software maintenance are respected and (b) the knowledge relating to previous phases in the systems development can be made available. How might this be achieved?

The literature on KM tends to divide along two lines; the first focuses on capturing 'explicit' knowledge and storing it in repositories for later reuse, whereas the second focuses on people and communities as sources of 'tacit' knowledge. The distinction between these two approaches and problematical nature of the relation between tacit and explicit knowledge has been explored elsewhere (Hildreth & Kimble, 2002; Kimble, 2013). Although there

Please cite this article in press as: de Vasconcelos, J. B., et al. The application of knowledge management to software evolution. *International Journal of Information Management* (2016), http://dx.doi.org/10.1016/j.ijinfomgt.2016.05.005