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

is no doubt that a detailed knowledge of the particular system that needs to be maintained combined with a familiarity with the domain in which it is used (Anquetil et al., 2007) – both of which have a strong tacit component – is of value, we contend that much of the knowledge needed to maintain systems is, in principle, explicit knowledge. Given the number of studies that seem to indicate the opposite (Brown & Duguid, 1991; Bobrow & Whalen, 2002), we will briefly explain why we believe this to be the case.

Firstly, we need to make clear that we are not arguing that every aspect of maintenance can be reduced to explicit knowledge and stored in a database; there is for example a distinction between the knowledge needed to be a good systems maintainer and the knowledge needed to maintain a specific system, we will focus on the latter. One of the issues we identify above is what might be termed "lost knowledge" such as the rationale for decisions that were taken at some earlier stage in the systems life cycle; it is this type of knowledge that we believe is amenable to the approach we describe here.

Knowledge management strategies built on codification are based on Shannon and Weaver's (1949) work on communications theory that was concerned with how a message could encoded and transmitted via a communications network. Briefly, their work showed that as long as a suitable 'codebook' for encoding and decoding the message existed, then the message could be transmitted without a loss of information. Within KM this same approach is applied to stored item of explicit knowledge, as long as a suitable codebook exists then the meaning of the original item of knowledge can be retrieved (Kimble, 2013).

Cowan et al. (2000) examined the extent to which knowledge could be codified; although Cowan et al. are economists their analysis applies equally to the situation we deal with here. Their analysis centers on the availability of a codebook, which they view as an 'authorized dictionary' that is used to decode messages (e.g. those stored in documents). The codes it contains are not necessarily a representation of some underlying truth but are simply knowledge that has been endorsed by an authority or has gained authority through common consent. They divide the problem into three categories: knowledge that has been recorded in some way and is therefore already codified; knowledge that in principle could be codified but currently is not, and tacit knowledge, which is inarticulable and therefore cannot be codified. The divide the middle case, where knowledge is not yet codified, into two and note, "In one, knowledge is tacit in the normal sense – it has not been recorded either in word or artifact, so no codebook exists. In the other, knowledge may have been recorded, so a codebook exists, but this book may not be referred to by members of the group" (Cowan et al., 2000)

We believe that much of the knowledge needed for systems maintenance lies in the latter category: the knowledge has been made explicit and recorded at some point but the codebook has been 'lost'. To a greater or lesser extent, every systems development methodology ensures that certain aspects of the rationale for the decisions made during the system's development are recorded and documented; returning to the point made by Samuelis (2008) the problem for maintainers is that there is no safe and dependable route back to that original knowledge. We believe that the framework we describe below, which relies upon back-flushing relevant information from the documentation produced in earlier phases of the systems development lifecycle, goes some way towards solving this problem.

4. Knowledge management systems for software engineering

According to the Guide to the Software Engineering Body of Knowledge (ACM/IEEE SWEBOK) (Bourque & Fairley, 2014), Software Engineering is "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software". Research in software engineering presents methods and techniques for an effective, faster, and economical software construction. Effective, (a) meaning problem-solving, faster (b) by minimizing the software development cycle, and economical (c) software development by ensuring the use and reuse of existing software components and optimizing the resource (human and physical) allocation towards the software construction and maintenance.

There are several models like cascade, agile, v-model, spiral, etc. for the Software Development Life Cycle (SDLC); these are approaches that determine the tasks and deliverables for software creation. In general, they are composed by six phases, allowing knowledge workers to specify and change software requirements into the final product or deliverable (Bourque & Fairley, 2014). Requirements specification and Analysis, to interpret and explicit the client's needs; design, to transform requirements (business knowledge) into a plan (technical knowledge); coding, where the plan is implemented; testing, a proof of concept that the software meets the requirements; software deployment, when it is shared and implemented and finally maintenance, to support the software usage. Sponsors, who have the business know-how, interact with the knowledge workers, to whom they transfer business knowledge, so that they can design models to transfer it into technical knowledge, meaning models that describe the software (Camacho, Sanches-Torres, & Galvis-Lista, 2013).

4.1. Software based knowledge intensive organizations

Dependency on technology grows each day, increasing the costs for developing software, making it a larger piece of the "companies cake" called budget as well as boosting the amount and diversity of information that organizations must deal with (Natali & Falbo, 2005; Rus & Lindvall, 2002). A software-based organization can be viewed as a knowledge intensive organization. The organization's software development practices are based on the knowledge and competencies (mainly their experience and heuristics) of its software developers and stakeholders (Vasconcelos, Kimble, Miranda, & Henriques, 2009).

In an organization individuals create documents, deal with software legacy applications, emails and many other tools where their everyday work is documented. Additionally, there is an informal or formal exchange of ideas with their peers to clarify doubts or discuss certain topics. All of these are contents for the three levels of refinement to knowledge, which are data; information; and knowledge (Davenport 2010; Rus & Lindvall, 2002). Knowledge is not a recipe; it is a set of procedures for dealing with a concrete situation, a routine. Knowledge should allow organizations to cope with different situations, anticipate implications, and assess its effects (Cascão, 2014).

Knowledge Management aims at the individuals, their knowledge, and the flows between them. Knowledge life cycle, defines the phases of organizational knowledge (Rus & Lindvall, 2002). These phases are: (1) Creation, the origin of the cycle, occurs by informal or formal exchange of ideas and information either from internal or external sources; (2) Capturing, follows creation, it is the moment when it becomes explicit/documented; and (3) Transforming knowledge, which means organizing, mapping and converting it into an interpreted form. After it has been documented and organized, it can be deployed, meaning it can be accessed. Accessibility does not mean it reaches individuals; therefore sharing occurs is when knowledge is used. The last phase is the goal of KM, which is the usage, allowing individuals to create more knowledge and therefore closing the cycle.

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

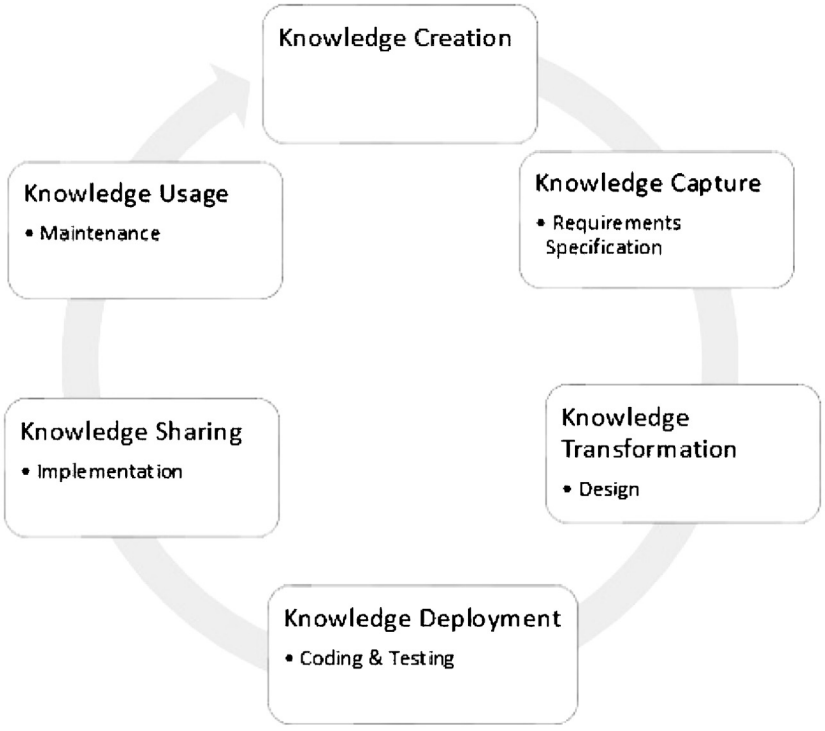

Fig. 1. Mapping KLC processes versus SDLC activities.

There is a significant amount of documents and artifacts produced during the phases of the SDLC, recording knowledge gathered along each sprint or project. Dealing with their usage is a challenge for the organizations. How to access it, along with ways to research what is really needed in a particular context, makes all this documentation unusable and difficult to share. Moreover, this knowledge so dispersed that makes it hard to update (Rus & Lindvall, 2002).

While some knowledge is recorded, some remains in the individuals' mind (Natali & Falbo, 2005), meaning that at the end of the day knowledge leaves the organization. Heuristics that individuals attain over several years cannot be transmitted within months to others and when they leave the company, a knowledge gap is created, sometimes with complex and serious consequences for the organization.

Combining distinct approaches, such as KM and SE, the key challenge of our research is to answer the following question: "What kind of KM framework is needed to help software maintenance knowledge workers?T̈he following section presents an overview of some related work; this is followed by an approach for the application of KM to software engineering, identifying the challenges organizations face to apply KM practices.

4.2. Related work

We reviewed other authors' research in KM for SE especially through the software maintenance approach as means of reference and guidance to our work. The following table (Table 1) resumes each one regarding their focus on knowledge management, software development process and software quality dimensions.

According to Andrade et al. (2006), the focus is essentially on knowledge capture in the software requirements phase, enhancing the corporate memory as the main area of intervention, and it has left out the software maintenance phase and the related problems. On the other hand, Rodriguez et al. (2014) and Talib et al. (2010) have the main focus on the software maintenance phase, its problems for the software development process dimension and for the KM dimension emphasize the knowledge share, and the reuse of lessons learned. In this research, the previous phases of the software development life cycle were left aside.

Anquetil et al. (2007) work is the most in-depth research paper in this area. However, has no focus on quality control and knowledge corrective measures. Although the primary focus is the software maintenance phase, it does not neglect the previous phases in the overall software development process. Knowledge sharing has a minor attention and knowledge capture has a mild focus.

Anquetil et al. (2007) is definitely a reference and guidance for future research work because it has a well-defined ontology based on the maintenance problems to support other dimensions. Additionally, this research has reached the perception that the application of KM in software maintenance promotes the creation of a knowledge management culture (and approach) within the organization, facilitating the implementation of KM initiatives to enhance SE activities.

4.3. KM practices for software engineering

Individuals that work in software development projects, also known as 'knowledge workers' (Davenport 2010) must be able to interpret the sponsor's needs and transform them into coded language, thus SE is a discipline where one has to master both social and technical skills (Alawneh, Hattab, & Al-Ahmad, 2008). Although every project is unique and each model for software development has its specific methodology, all of them bare in common a complex set of tasks to achieve the final goal. The SDLC is a knowledge-intensive environment where KM processes fit like a glove. It can function as a complement to support individuals during the SDLC phases to enhance productivity and quality (Natali & Falbo, 2005). The map we present in Fig. 1 evidences the proximity between the Knowledge Life Cycle processes and SDLC activities. SDLC activities transform tacit knowledge into explicit (Aurum & Ward, 2004), and

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

Table 1
Analogous projects review.

Dimensions		Projects				
		P1	P2	P3	P4	MIMIR
		A Reference Model for Knowledge Management in Software Engineering	Applying Agents to Knowledge Management in Software Maintenance Organizations	MASK-SM: Multi-Agent System Based Knowledge Management System to Support Knowledge Sharing of Software Maintenance Knowledge Environment	Software maintenance seen as a knowledge management issue	A Knowledge Management Approach for Software Engineering Projects Development
Knowledge Management	Knowledge Capture	●●●	○	○	●	●●
	Knowledge Transformation	●●	○	○	●	●●
	Knowledge Sharing	●	●●●	●●●	●●	●●
	Knowledge Usage	●	●●●	●●●	●●●	●●●
Quality	Knowledge Quality Control	○	●●●	○	○	●●
Software Development Life Cycle	Requirements Definitions	●●●	○	○	●●	●●●
	Software Development	●	○	○	●●	●●●
	Software Maintenance	○	●●●	●●●	●●●	●●●

Legend: ○ No focus.
● Little focus.
●● Mild focus.
●●● Large focus.

when properly documented, they are organizational knowledge reification activities.

Knowledge in software development projects is diverse and growing in proportions. Organizations have problems identifying the contents, location, and the best way to make use of it (Rus & Lindvall, 2002). An important step towards a knowledge management project is the characterization of the knowledge assets. The domain knowledge includes business processes, decision-making, entrepreneurial, declarative, and procedural knowledge, heuristics and informal knowledge (Vasconcelos et al., 2009). For this paper, the activities of SDLC were segmented in three major process areas derived from Fig. 1, regarding their outputs combined with their needs. They are Requirements Definition (RD), Software Development (SD), and Software Maintenance (SM), as seen in Fig. 2. With this, it is possible to identify all the documents created during the SDLC and determine within each one the more significant information.

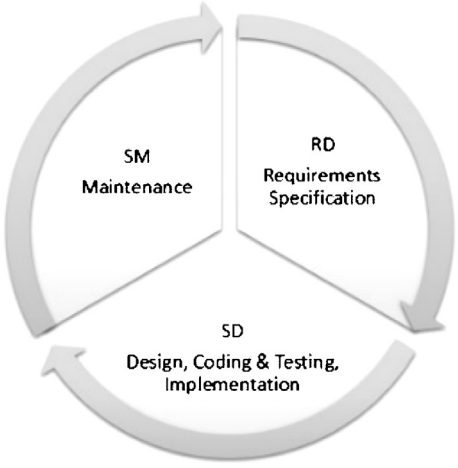

Fig. 2. SDLC segmentation accordingly to their outputs and needs.

Our main focus is the SM area, since in general KM processes have been less explored (Alawneh et al., 2008; Bourque & Fairley, 2014; Isotani, Dermeval, Bittencourt, & Barbosa, 2015; Natali & Falbo, 2005; Rus & Lindvall, 2002; Vasconcelos et al., 2009). It turns out that the documents produced in previous phases of SDLC are rarely used to aid in solving maintenance problems. Maintainers end up mainly resorting to read the legacy software code or to an informal exchange of ideas with their peers; 40% to 60% of the effort made in SM is spent understanding the software that is being maintained (Anquetil et al., 2007). SM deals with real problems and its activity can be used to evaluate knowledge documented in the previous phases.

There is also a set of KM practices which can be applied to software engineering activities, such as (a) knowledge elicitation and acquisition practices during the software requirements stage; (a) the application of information management and visualization tools during the software design stage; (c) and collaborative work practices and procedures during the software construction (programming tasks) and maintenance stage (Table 2).

5. MIMIR framework

Software construction and maintenance is a demanding and complex activity. Software maintainers need to master programming languages, the system architecture, understand data models, have procedural knowledge, software updates, and their impacts and often this is done for several applications. Experienced individuals are chosen for these tasks, seniority and heuristics makes

Table 2
Knowledge management practices for software engineering activities.

Software Engineering Stage	Knowledge Management Practices
Requirements engineering activities	–Knowledge elicitation –Knowledge acquisition
Software design	–Information management and visualization
Software construction and maintenance	–Collaborative work

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

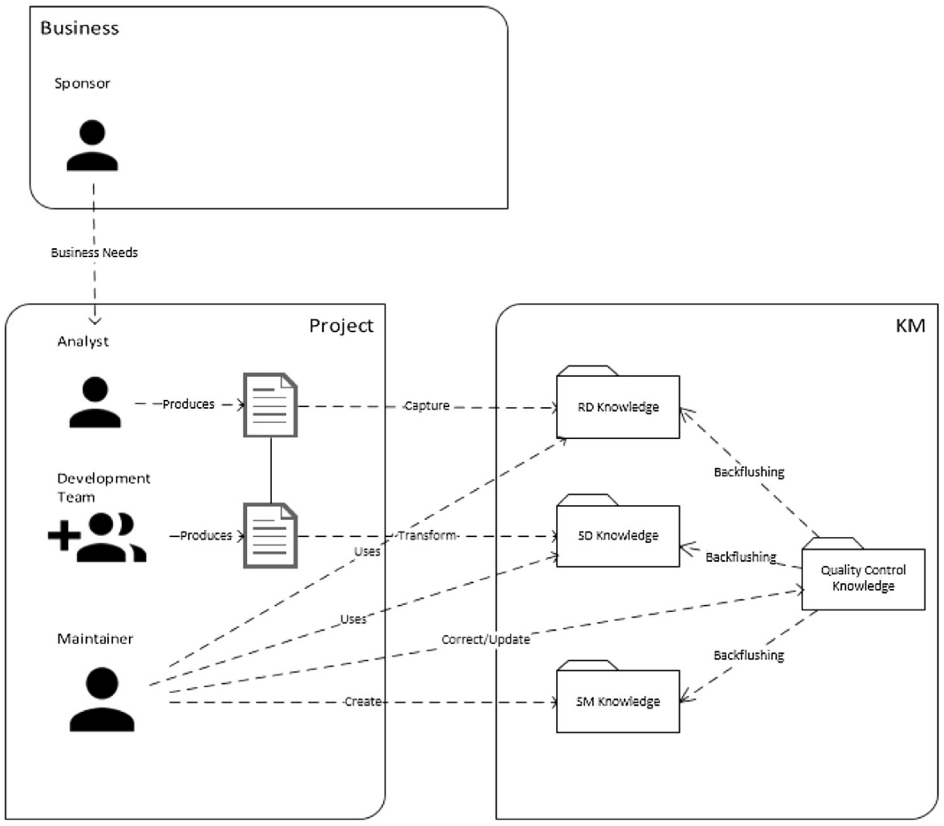

Fig. 3. MIMIR model overview.

them the manager's top choice, thus highlighting that these are tasks that require a high level of organizational knowledge, sensibility, and experience, due to their criticality and urgency. The MIMIR Model (Carreteiro, Vasconcelos, Barão, & Rocha, 2016) presented next, named after the Norse mythology god who guards the "Well of the Highest Wisdom", is an approach to a framework mapping information based on documents created in the SDLC phases to retrieve from them the utmost relevant knowledge for SM.

5.1. MIMIR model overview

This model was driven by the fact that maintainers deal with problems, therefore accessing relevant knowledge to aid them in the research is extremely important. Additionally, they can validate, correct or update the accessed knowledge, for which back flushing is applied, a technique used to correct data in the origin during the Extract, Transform, and Load (ETL) process in Data Warehousing.

MIMIR Model performs a preliminary approach to the SDLC phases in organizations based in three segmentation areas as shown in Fig. 2. Thus, as shown in Fig. 3, several roles were defined as Unified Modeling Language (UML) stereotypes for participants, namely: Sponsor, Analyst, Development Team, and Maintainer. Sponsor role is to define the needs for project and validator for the requirements quality; they can be external or internal to the organization. Analyst is the expert for the model, providing the requirements elicitation, application analysis, and main source of knowledge in the project area. Development Team involves the individuals that participate in the project mainly in the coding phase. In smaller projects, the Analyst can participate in the development team. Maintainer is the individual that provides support for the usage of the system, if the analyst has exclusive business knowledge he does not play the maintainer role, on other hand, if he is an Application Analyst he will surely be a Maintainer. RD Knowledge will retain all significant information regarding requirements elicitation. SD Knowledge will gather information from development phases of the project like technical plans or software design specification. SM Knowledge gathers maintainers' most frequently problem solutions, batch processing sequences, or application navigational sequences. Quality Control Knowledge is where the corrections and updates to previous knowledge remain until approval from experts like analysts to initiate the back flushing process.

5.2. MIMIR requirements capture model

MIMIR is centered in the extraction of knowledge created in documents produced by the knowledge workers throughout the SDLC phases using the CMMI-DEV model base. In this paper, in Fig. 4 we only present the main concepts of the RD Knowledge Base feed in the requirements phase.

The elicitation of Sponsor needs from the Analyst will first produce the Business Requirements Specifications (BRS) document; a high-level business needs document and kick-off for the SDLC phases. The project ID goals and scope are important information that the MIMIR-Extractor module will acquire from the document. These will permit to create the project in the MIMIR RD Database as well as the first tags will be added to help in the search queries and the knowledge flow for the individuals, specially maintainers.

Also produced by the Analyst is the Project Requirements Specifications (PRS), in which the requisites for the development team will be described, each one with a specific ID, type and description and all referring to a functional area of the application or system.

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

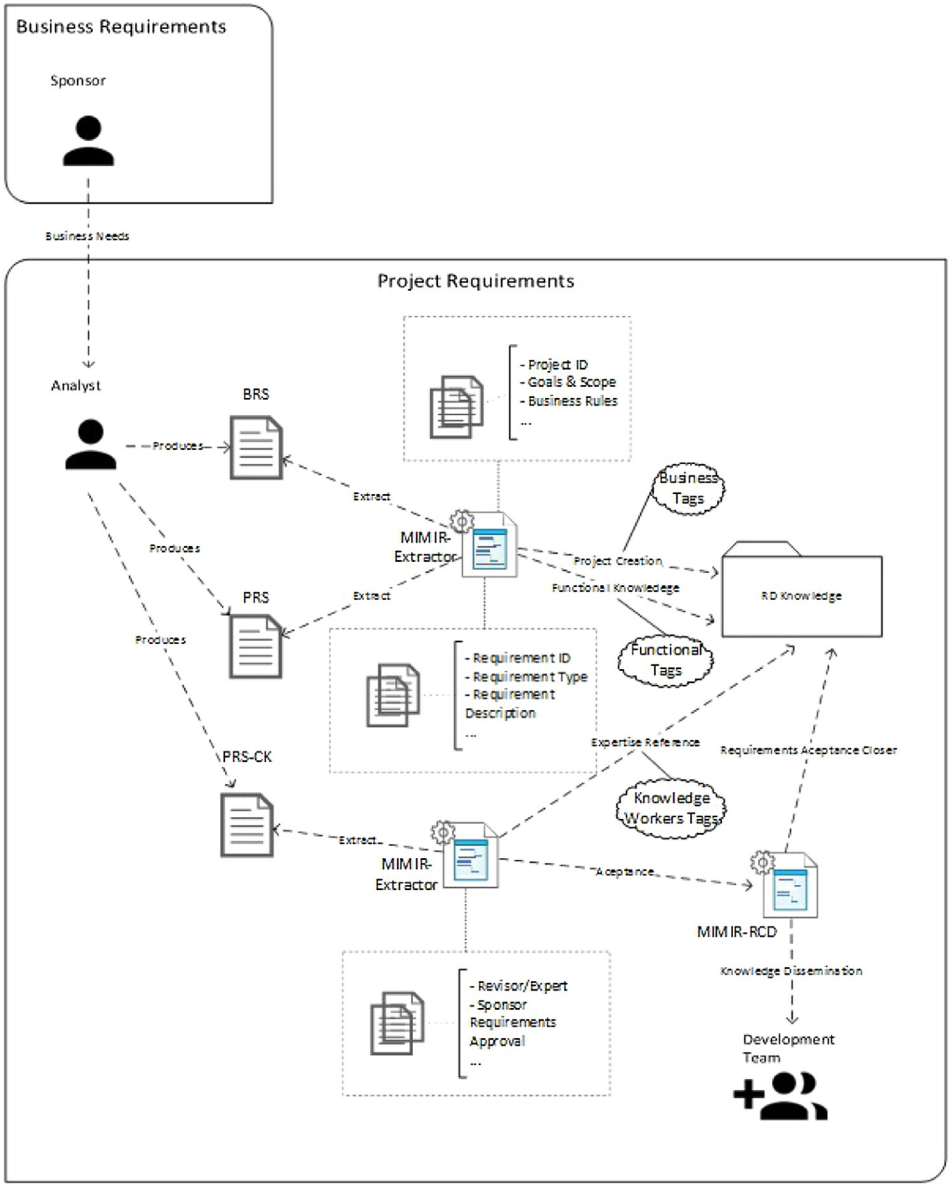

Fig. 4. MIMIR requirements capture model.

This information will be added to the RD Database project created previously. The requirement description will provide important knowledge in a natural language, easy and quick to interpret by the individuals specially Maintainers. Using the requirement ID as a key, MIMIR will map these descriptions to the list of artifacts that are affected in the project, registered later in the SD Knowledge Database.

The Project Requirements Specifications Check-List (PRS-CK) will close the requirements phase and consolidate the RD Database information from this document and MIMIR-Extractor will retrieve information regarding the Analyst involved in the requirements elicitation, for previous expertise referral. The last step of the requirements phase is determined by requirements acceptance from the Sponsor, which will provide the trigger for the Requirements Close and Dissemination module (MIMIR-RCD) to make the knowledge from the MIMIR RD Database visible and distribute it to individuals in the Development Team.

6. Conclusions

Effective software engineering practices aim to reduce existing cases of software development projects that miss schedule, exceeds budget previously defined in the requirements engineering stage, and deliver software applications with reduced quality. In complex and concurrent software construction projects, a consistent planning and communication between the stakeholders becomes crucial for effective collaboration across the different software construction stages. In this paper, we have argued that the adoption of knowledge management practices in software engineering activities would improve software construction and maintenance tasks.

Aiming to emphasize the effect of knowledge management practices during software development projects, this paper presented a first approach to cope with knowledge management and engineering practices across software development projects. The main goal was to present a knowledge management roadmap for software development process tasks during a typical software project development. The research applies an architectural case

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

study using software maintenance tasks as a means to enhance the knowledge flows within the organization. Software maintainers validate, correct, and update knowledge from previous phases of software development life cycle through the application of back flushing technique at the software data warehouse and repository.

Software development projects are knowledge-intensive and software development activities are the reification of the organizational knowledge. A lot of knowledge is documented but a lot remains in the individuals' mind, therefore creating a potential (organizational) knowledge gap. How to use and reuse documented knowledge and how to document the software development process effectively are challenges for the organizations. Software development activities proximity to knowledge management practices can work as a facilitator to implement knowledge management projects in software based organizations. However, knowledge management approaches to software engineering are mainly connected to the early phases of a typical software development process and especially in the requirements specification, leaving the software maintenance activities to a second plan. A significant amount of the time spent in maintenance activities is reacquiring knowledge and this is the main driver for our knowledge management approach to software construction and maintenance.

The aforementioned MIMIR architectural approach applies documents provided from previous software engineering phases to capture knowledge from each software project and has a particular concern to use the maintainers as a means to validate and adjust the knowledge gradually inferred. The result is a KM software project charter combining a set of knowledge acquisition tasks across the SDLC framework.

As future work, our main objective is to present a more detailed MIMIR model, methodology and tool, based on software development organization, combining insights across corporate software projects as a means of evaluating effects on people and organization, technology, workflows and processes, and formally applying and evaluation knowledge management and engineering techniques in software development (construction and maintenance) processes.

References

Alawneh, A. A., Hattab, E., & Al-Ahmad, W. (2008). An extended knowledge management framework during the software development life cycle. *International Technology Management Review*, *1*(2), 44–62.

Andrade, J., Ares, J., Garcia, R., Rodriguez, S., & Suarez, S. (2006). A reference model for knowledge management in software engineering. *Engineering Letters*, *13*(2), 2006.

Anquetil, N., de Oliveira, K. M., de Sousa, K. D., & Dias, M. G. B. (2007). Software maintenance seen as a knowledge management issue. *Information and Software Technology*, *49*(5), 515–529.

Aurum, A., & Ward, J. (2004). Knowledge management in software engineering – describing the process. *2004 Australian software engineering conference*.

Blum, B. I., & Sigillito, V. G. (1985). Some philosophic foundations for an environment for system building. In *Paper presented at the Proceedings of the 1985 ACM annual conference on the range of computing: mid-80's perspective* (pp. 516–523).

Bobrow, D., & Whalen, J. (2002). Community knowledge sharing in practice: the Eureka story. *Reflections*, *4*(2), 47–59.

Bourque, P., & Fairley, R. E. (2014). *SWEBOK v3.0 – guide to the software engineering body of knowledge*. New Jersey: IEEE Computer Society.

Brown, J. S., & Duguid, P. (1991). Organizational learning and communities of practice: toward a unified view of working, learning, and innovation. *Organization Science*, *2*(1), 40–57.

Bryant, A. (2000). It's engineering Jim. . . but not as we know it: software engineering—solution to the software crisis, or part of the problem? In *Paper presented at the Proceedings of the 22nd international conference on Software engineering* (pp. 78–87).

Camacho, J. J., Sanches-Torres, J. M., & Galvis-Lista, E. (2013). A systematic literature review understanding the process of knowledge transfer in software engineering. *The International Journal of Soft Computing and Software Engineering*, *3*(3), 219–229.

Carreteiro, P., Vasconcelos, J. B., Barão, A., Rocha, A., et al. (2016). A knowledge management approach for software engineering projects development. In Rocha (Ed.), *Proceedings of the world conference on information systems (WorldCist2016)*.

Cascão, F. (2014). *Gestão de Competências, do conhecimento e do talento*. Lisboa: Edições Sílabo.

Cowan, R., David, P. A., & Foray, D. (2000). The explicit economics of knowledge codification and tacitness. *Industrial and Corporate Change*, *9*(2), 211–253.

Davenport, T. H. (2010). *Process management for knowledge work: handbook on business process management*. Springer.

Hildreth, P. M., & Kimble, C. (2002). The duality of knowledge [Electronic version]. *Information Research*, *8*(1). Retrieved 12 October 2012 from: http://informationr.net/ir/8-1/paper142.html

Isotani, S., Dermeval, D., Bittencourt, I., & Barbosa, E. (2015). Ontology driven software engineering: a review of challenges and opportunities. *IEEE Latin America Transactions*, *13*(3), 863–869.

Kimble, C. (2013). Knowledge management, codification and tacit knowledge [Electronic Version]. *Information Research*, *8*(2). Retrieved 19 June 2013 from: http://informationr.net/ir/18-2/paper577.html

King, D., & Kimble, C. (2004). Notions of equivalence in software design. In *Paper presented at the 9e colloque de l'AIM*.

Koskinen, J. (2003). *Software maintenance costs*.. Retrieved March, 2016, from: https://archive.is/o/oBIIr/http://www.cs.jyu.fi/~koskinen/smcosts.htm

Lehman, M. M., & Ramil, J. F. (2003). Software evolution-background, theory, practice. *Information Processing Letters*, *88*(1), 33–44.

Lehman, M. M. (1979). On understanding laws, evolution: and conservation in the large-program life cycle. *Journal of Systems and Software*, *1*, 213–221.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 1060–1076.

Lehman, M. M. (1996). Laws of software evolution revisited. In *Software process technology*. pp. 108–124. Springer.

Lyytinen, K. (1987). Different perspectives on information systems: problems and solutions. *ACM Computing Surveys*, *19*(1), 5–46.

Natali, A. C., & Falbo, R. d. (2005). Knowledge management in software engineering environments. Proceedings of the XVI Brazilian Symposium on Software Engineering, Vitoria. pp. 238–253.

Nonaka, I., & von Krogh, G. (2009). Perspective – tacit knowledge and knowledge conversion: controversy and advancement in organizational knowledge creation theory. *Organization Science*, *20*(3), 635–652.

Pigoski, T. M. (1996). *Practical software maintenance best practices for managing your software investment*. New York: John Wiley & Sons.

Rodriguez, O. M., Vizcaino, A., Martinez, A. I., Piattini, M., & Favela, J. (2014). Applying Agents to Knowledge Management in Software Maintenance Organizations. https://www.researchgate.net/publication/267701359_Applying_Agents_to_Knowledge_Management_in_Software_Maintenance_Organizations.

Rus, I., & Lindvall, M. (2002). Knowledge management in software engineering. *IEEE Software*, *19*(3), 26–38.

Samuelis, L. (2008). Notes on the emerging science of software evolution. In *Handbook of research on modern systems analysis and design technologies and applications*. pp. 161–167. Hershey: Information Science Reference.

Shannon, C. E., & Weaver, W. (1949). *The mathematical theory of communication*. Urbana, IL: The University of Illinois: Press.

Talib, A. M., Abdullah, R., Atan, R., & Murad, M. A. (2010). MASK-SM: multi-Agent system based knowledge management system to support knowledge sharing of software maintenance knowledge environment. *Computer and Information Science*, *3*(2), 52–78.

Vasconcelos, J. B., Kimble, C., Miranda, H., & Henriques, V. (2009). A knowledge-Engine architecture for a competence management information system. In *Proceedings of the UKAIS conference*

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