An architectural model for software testing lesson learned systems

Javier Andrade [a,1], Juan Ares [a,1], María-Aurora Martínez [b,2], Juan Pazos [c,3], Santiago Rodríguez [a,1], Julio Romera [c,3], Sonia Suárez [a,*]

[a] University of A Coruña, Campus de Elviña, s/n, 15071 A Coruña, Spain
[b] UDIMA-Madrid Open University, Camino de la Fonda, 20, 28400 Collado-Villalba, Madrid, Spain
[c] Technical University of Madrid, Campus de Montegancedo, s/n, 28660 Boadilla del Monte, Madrid, Spain

A B S T R A C T

Context: Software testing is a key aspect of software reliability and quality assurance in a context where software development constantly has to overcome mammoth challenges in a continuously changing environment. One of the characteristics of software testing is that it has a large intellectual capital component and can thus benefit from the use of the experience gained from past projects. Software testing can, then, potentially benefit from solutions provided by the knowledge management discipline. There are in fact a number of proposals concerning effective knowledge management related to several software engineering processes.
Objective: We defend the use of a lesson learned system for software testing. The reason is that such a system is an effective knowledge management resource enabling testers and managers to take advantage of the experience locked away in the brains of the testers. To do this, the experience has to be gathered, disseminated and reused.
Method: After analyzing the proposals for managing software testing experience, significant weaknesses have been detected in the current systems of this type. The architectural model proposed here for lesson learned systems is designed to try to avoid these weaknesses. This model (i) defines the structure of the software testing lessons learned; (ii) sets up procedures for lesson learned management; and (iii) supports the design of software tools to manage the lessons learned.
Results: A different approach, based on the management of the lessons learned that software testing engineers gather from everyday experience, with two basic goals: usefulness and applicability.
Conclusion: The architectural model proposed here lays the groundwork to overcome the obstacles to sharing and reusing experience gained in the software testing and test management. As such, it provides guidance for developing software testing lesson learned systems.

1. Introduction

Software testing is the dynamic verification of actual against expected program behavior on a finite set of test cases, suitable selected from the usually infinite execution domain [1]. Software has become more and more widespread and is now habitually used in critical and complex application domains, making this process increasingly important, critical, costly and complex, and calling for greater quality and reliability. Early studies claimed that the testing process accounted for 50% of total project development costs [2] or even more for highly critical software. The emergence of high-level languages, and the maintenance and upgrading of existing software systems has meant that the proportion of time spent on testing has increased (see, e.g., [3]). Also, the early testing techniques compiled by Myers [4] have been joined by new testing models (model-based testing [5], agile testing [6], etc.) and new testing techniques (machine learning techniques [7], adaptive random techniques [8], etc.). Additionally, software has been applied to new domains and has been output using new development models. All this really does make software testing an increasingly more complex and, above all, knowledge-intensive activity [9].

During testing planning, testing team members mainly select testing strategies, prioritize tests, define regression strategies and select the best testing techniques. Knowledge of existing methods and techniques is necessary but not enough to efficiently perform these tasks. Sound empirical knowledge and experience-based practice criteria are also required to gain a deeper understanding

* Corresponding author. Tel.: +34 981167000; fax: +34 981167160.
E-mail addresses: jag@udc.es (J. Andrade), juanar@udc.es (J. Ares), mariaaurora.martinez@udima.es (M.-A. Martínez), jpazos@fi.upm.es (J. Pazos), santi@udc.es (S. Rodríguez), julio_romera@elcorteingles.es (J. Romera), ssuarez@udc.es (S. Suárez).
[1] Tel.: +34 981167000; fax: +34 981167160.
[2] Tel.: +34 918561699; fax: +34 918561697.
[3] Tel.: +34 913366896; fax: +34 913524819.

of testing technique behavior [9,10]. Testing team members make use of this experience with more or less insight to improve their own job performance. The problem is, though, that the experience gained from different projects is confined to each individual and is not known to or shared by other team members (at least not formally). Testing engineers perform similar tasks and come up against similar problems day in day out as they work on different projects. However, testing teams do not make use or take advantage of the knowledge acquired and the experience gained, as this is confined to each individual. Therefore, the same mistakes are made over again, even though there are individuals in the testing organization with the knowledge and experience required to rule out or stop this. Likewise, successful practices are not repeated. Each testing project, each new technique for use, each new platform is a source of a lot of knowledge and experience. This knowledge and experience can be applied again in the future if the organization is capable of extracting that individual knowledge and making it available to and promoting its use by anyone who has need of it. This is the main aim of the knowledge management (KM) discipline.

This paper encourages the use of a software testing lesson learned (LL) system as a component of organizational learning to improve and promote the dissemination and reuse of individual experience gained from technical and managerial software testing activities. The proposed architectural model for software testing LL systems (i) defines the structure of the LLs, (ii) sets up the procedures for managing LLs, and (iii) supports the design of LL management software tools. In fact, we have developed a prototype software testing LL system implementing the core of the proposed model, that testing teams have used and evaluated. Accordingly, this paper has been structured as follows. Section 2 introduces the KM discipline, and discusses its applicability to software testing, plus the resulting benefits. Section 3 introduces LL systems and outlines the benefits of their use as a KM enabler in software testing, analyzing the latest major initiatives. Section 4 details the proposed architectural model, placing special emphasis on the design of its repository and the LL life-cycle activities. Section 5 details the prototype built and, finally, Section 6 lists the conclusions.

2. KM and software testing

2.1. KM foundations

The term knowledge management was first coined in 1986 [11] by Wiig, and later formalized in 2004 by Holsapple and Joshi in their Knowledge Management Ontology (KMO) [12]. According to Wiig, its main objectives, which are in consonance with the *DKMC1* definition provided in KMO and other works (e.g., [13]), are [14]: (i) *to make the enterprise act as intelligently as possible to secure its viability and overall success*, and (ii) *to otherwise realize the best value of its knowledge assets*.

A key aspect in any KM program is the definition of a corporate memory. A corporate memory specifies and supports the representation and exchange of key corporate knowledge [15–17] (see KMO's knowledge definition in *DKMC6*). Initially strategies from outside KM (see e.g., [18]), primarily knowledge-based and document-based approaches, were used to build corporate memories. Of the knowledge-based strategies, ontologies were significantly exploited. Note, however, that (i) a corporate memory is different from a knowledge-based system [18], and (ii) an ontology, although created to share and reuse knowledge, is concerned not with the dynamic but just the static knowledge [19] of a particular domain [20]. Although paper-based or electronic documents are potentially constitutive of a document-based corporate memory, they are considered as only a first step in the implementation of such repositories [21]. Very often documents are not well-indexed or constitute a specific and abridged bibliography for each employee [18].

As Fig. 1 shows, from the KM perspective, and considering how knowledge can be provided, a corporate memory contains two different repositories [22–24]: (i) a corporate knowledge base, providing direct access to the knowledge made explicit it contains, and (ii) a yellow pages repository, providing the location of and access to relevant knowledge that is not explicit in the above repository.

The corporate knowledge base includes two elements: knowledge and meta-knowledge (i.e., LL). Knowledge refers to the knowledge of a particular organizational environment

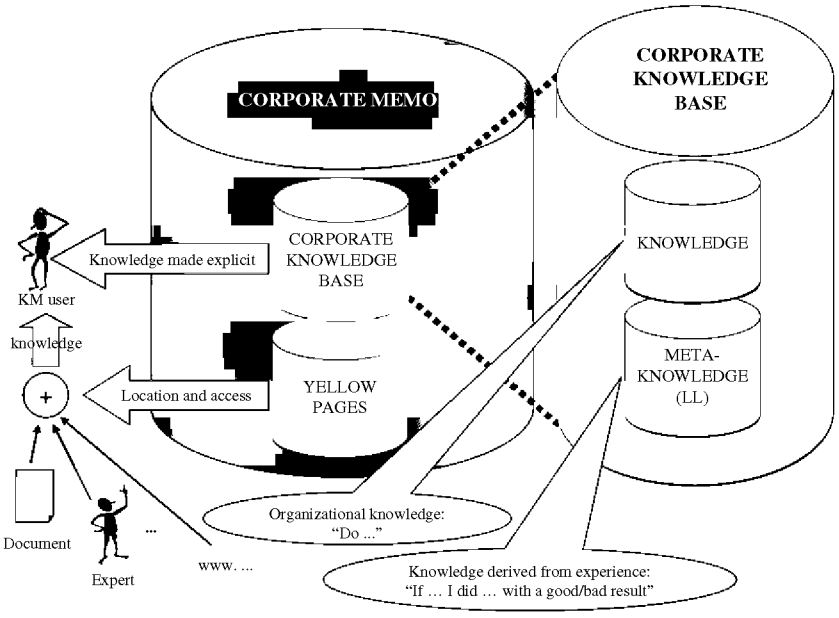

Fig. 1. Corporate memory components.

(e.g., knowledge about how to undertake a particular task). There are several definitions of meta-knowledge, since it has been described as guidelines, tips, or checklists of what went right or wrong in a particular situation [25] (e.g., heuristics about how to undertake a particular task). All these definitions however focus a common aspect: knowledge derived from (gained by) experience. We accept this definition and regard LLs as referring to the knowledge that each person possesses in the shape of experience. Note that a LL (meta-knowledge) can be adopted as knowledge if it generates competitive advantages: it evolves from meta-knowledge into an actual *modus operandi* (see, e.g., [15], which gives an example of this evolution out of the software testing domain). For this reason, often no distinction is made between the two concepts, and they are wrongly taken to mean the same thing.

The second repository, yellow pages, involves publishing the human (e.g., an expert) and non-human (e.g., a web page) sources that have additional knowledge; that is, key knowledge that is not specified in the above repository. Thus, for example, the publication of the contact details of an expert who can provide help with a particular subject is considered within the scope of this repository. Note that a KM program should not try to explicit all the knowledge and LLs that exist in the organization (corporate knowledge) in the corporate knowledge base. This would not be a feasible goal due to the associated costs since every organization has a huge volume of relevant knowledge. This is why this second repository is so useful.

2.2. Software testing as a knowledge-intensive activity

Software engineering is a knowledge-intensive activity [26], simultaneously involving knowledge-intensive subactivities: requirements elicitation, risk management and testing, among others. To perform these activities, a software development organization's key asset is the knowledge, experience and creativity of its developer teams. Project focus, staff turnover, plus the development and release of new platforms, techniques and methods, and the application of software to new domains, all mean, however, that the knowledge and experience acquired by software team members should be disseminated across work groups and not confined to individuals: effective knowledge sharing is a critical success factor, and KM is an enabler of organizational learning [26,27].

The documents generated by the testing processes (i.e., technical knowledge) contain part of the knowledge used in software testing: test plan, test design specification, test case specification, test incidents, test logs, etc. Another part is, of course, embedded in the testing procedures, techniques and methodologies (i.e., methodological knowledge). But a third type of knowledge is required to implement the above two: meta-knowledge. Etymologically speaking, meta-knowledge is knowledge about knowledge (LLs) [15,27], that is, knowledge about software test applicability, effectiveness for particular testing or software types, risks and real benefits, etc. It is LLs that provide the knowledge required for the effective application of the technical and methodological knowledge. In other words, meta-knowledge is aimed at the application of knowledge.

The biggest problem encountered in this respect is the difficulty for sharing and reusing the experience that each software testing engineer gains from testing at the corporate level. In actual fact, many software testing experiences and skills are taken in by only a few people, and do not become public knowledge [28]. This is probably caused to some extent by the fact that the knowledge associated with software testing has certain peculiarities. First, most testers are self-taught; many have never read a book on the subject [29], and testers seldom receive the lifelong training necessary to effectively do their jobs [30], as there appears to be a belief that testers do not require any specialized training [31]. In actual fact, test jobs are often consolation prizes for people not considered good enough to be recruited as software developers [29]. It is a fact that testers have very limited knowledge of the techniques that are currently at their disposal [32]. On the other hand, testing does not receive as much attention as other software development activities in either research or business practice, meaning that industry testing practices are generally not very sophisticated or effective, and leave a lot to be desired [33]. Technology transfer between research and industry is insufficient, but so is intra-organizational knowledge transfer for such a critical and knowledge-intensive activity [28].

2.3. Usefulness of KM in software testing

To counter the above and institutionalize knowledge, we need a KM program [15]. The aim of a KM program is to enable individuals to solve problems more efficiently. This will be possible due to better decision-making criteria provided especially by either their own or others' past experiences. To be precise, organizational KM has two dimensions: (i) it aims to transfer knowledge existing at the individual level to the organizational level; and (ii) it aims to explicit the knowledge gained from experiences, as a first step for dissemination and later conversion to tacit knowledge. This is what Nonaka et al. term the process of externalization and internalization [34].

KM applied to software engineering has several uses. First, it provides for continuous software process, and thus product, improvement through the modification and adaptation of processes based on practical experience [35]. Second, it helps to rise to the challenges set by new work techniques and methodologies and to share the benefits and outcomes of their use in new application domains [36]. Third, it retains the knowledge in a corporate memory, including everything pertaining to processes, products, domains, techniques, methods, plans, strategies and objectives, as a means for storing past knowledge for reuse [17]. Finally, it institutionalizes best software development practices in the organization [37].

The above advantages of the application of KM to software engineering are, in view of its activities, equally applicable, by inclusion, to testing. Additionally, there are other points on which KM can specifically benefit software testing:

- Selection and application of better suited techniques and methods. There are different testing methods and techniques (manual and automatic), as well as studies that are more or less helpful for selecting which are best suited for the case at hand (see, e.g., [38,39]). Testing techniques and methods are useful, but they have no practical selection and application criteria. Some of these techniques (e.g., ad hoc testing [40], exploratory testing [41], and error-guessing testing [40]) even depend on the knowledge, experience and intuition of testing engineers [42,43]. Experience then plays a key role in testing, and management of past experience will possibly help to effectively tailor the techniques and methods to the ongoing project.
- Test cases selection and test design. Designing tests and selecting test cases involves adopting a strategy to trade off two opposite needs: amplify testing thoroughness, and reduce times and costs [42]. It is not easy to reach a trade-off that successfully selects the set of test cases that maximizes efficiency and minimizes costs, and, as Beer and Ramler [43] found in their survey, tester experience is one of the fundaments for designing test cases and selecting regression tests.

- Running and executing tests. Knowledge and experience of the domain and the product (system under test) is essential for increasing test effectiveness. Thus, for example, as Kaner et al. stated [44], "An experienced tester who knows the product and has been through a release cycle or two is able to test with vastly improved effectiveness".
- Testing by independent groups or outsourcing. Software testing is an activity that lends itself to being performed by independent groups within the organization or to being outsourced. Knowledge (and its management) has logically proved to be a key factor in both cases. In this sense, for example, Karhu et al. [45] studied the relationship of outsourced software testing to KM, concluding that outsourced software testing is more effective when independent testing agencies have enough domain knowledge. This is achieved by making knowledge explicit and, therefore, transferable.

As an illustrative example of how KM can benefit software testing in a particular company, consider that, back in 1978, NEC undertook an initiative for "learning from bugs". This initiative aimed to find the causes of software failures and prevent bugs: program bugs, software development mistakes and failures or stumbling blocks [46]. This initiative was implemented in 1981. It managed to increase software productivity and decrease the defect level, although its biggest achievement was to improve the software development process through the performance of systematic process improvement activities.

3. LLs and software testing

3.1. LL systems

Secchi et al. [47] gave the most comprehensive definition of a LL: "A lesson learned is a knowledge or understanding gained by experience. The experience may be positive, as in a successful test or mission, or negative, as in a mishap or failure. Successes are also considered sources of LLs. A lesson must be significant in that it has a real or assumed impact on operations; valid in that it is factually and technically correct; and applicable in that it identifies a specific design, process, or decision that reduces or eliminates the potential for failures or mishaps, or reinforces a positive result". Unlike other knowledge artifacts (as they are termed in KMO), LLs are rooted in experience, describe both failures and successes and target organizational reuse [25].

LL systems are a KM enabler whose ultimate aim is to convert people's experience-derived individual knowledge into organizational knowledge through reuse. To manage these experiences, LL systems implement the processes supporting the LL life cycle. These processes manage the acquisition/collection, verification, storage, dissemination and reuse of LLs, which are hosted in a corporate memory.

LLs have mainly been used in mission-critical environments, defense-related areas and critical software, where there is a risk of massive human or material losses and it is essential to learn quickly from past experiences (successes or failures). This way, the first LL systems commissioned concerned accident prevention in military, aerospace, energy source management or environmental activities. These are not, however, the only types of activities that can benefit from the advantages of a LL system. Such a knowledge-intensive activity as software testing can also profit from a LL system as a means of managing individual experience gained in testing projects to prevent the same mistakes from being made again and to assure successes are repeated. Martin et al. [48], for example, stress the importance of learning from experience in software testing: "drawing and learning from experience is somehow as important as following a rational approach to testing".

3.2. Usefulness of LL systems in software testing

Let us consider test cases selection, which is a recurrent problem in software testing [38] and one of the factors that most affects testing quality. Test case selection sets out to minimize the number of cases and maximize effectiveness. There are many techniques designed for this purpose, but there is little information about their applicability and suitability for a particular piece of software [39]. Additionally, testers use little information to make the decision on which techniques to use. Also what information they do use is based on intuitive factors and biased criteria. At the end of the day, there is a shortage of experimental information about the results of using these techniques in a particular context. A system managing LLs about testing technique selection and application could definitely help to solve this problem.

On the other hand, the software testing process has to be made more effective, predictable and effortless. To do this, it is necessary to research new approaches like model-checking techniques or search-based approaches for test input generation [49]. These new approaches need empirical and experimental confirmation across different software types. To do this, they have to be applied to real cases to ascertain their effectiveness and applicability by observing the outcomes. A LL system containing experience from having applied these approaches would improve decision making.

Notice also that, although tests of software built using new development models—web-based software, web services, SOA systems, etc.—share the same goals as the software developed with traditional methods, existing testing methods and techniques have to be tailored to the complexities and peculiarities of the new development models. Di Lucca and Fasolini [50] analyze the peculiarities of the web applications testing and conclude that empirical studies need to be conducted to verify and validate the effectiveness and efficiency of existing testing approaches, as well as understand which approach is better suited for identifying each failure type. In the case of software built using new development approaches, we again find that experience in the use and application of existing testing approaches to real testing projects is required and that a LL system would be useful as a container of such experience.

Finally, let us highlight how important learning is in software testing project management. Project-driven testing, test outsourcing and testing by independent teams mean that the tests are run by non-permanent groups (temporary organizations) and that any learning about the steps taken fritters away at the end of the tests. LLs could, however, provide critical input for several project process areas, such as testing plan development, testing scope definition, estimation of the duration of testing activities, testing resource planning, and testing risk identification and analysis [51]. Software testing projects therefore could also improve if management LLs were considered. In fact, looking at Weber et al.'s goal-based classification [25], LLs can be divided into: technical lessons and planning (management) lessons.

In brief, all the testing tasks in which experience is an important factor could benefit from the deployment of a software testing LL system. These tasks include:

- Tasks related to the testing process management (including management LLs):
  - Resource, cost and time estimation.
  - Testing risk identification and evaluation and deployment of preventive and corrective actions.
- Tasks related to the testing strategy (including technical LLs):
  - Identification of aspects to be more thoroughly tested.

- Identification of the order of integration testing.
- Selection of the best testing techniques.
- Selection of the set of test cases that maximizes effectiveness.
- Identification of test case and test suite reuse.
- Identification of testing automation effectiveness criteria.
- Selection of testing automation tools.

- Tasks related to tactical testing issues (including technical LLs):
  - Generation of test data.
  - Prioritization of test cases.
  - Selection of most effective regression tests.
  - Establishment of stopping rules for testing (test coverage).

3.3. Current software tools

Even though, as mentioned above, KM and, particularly, LL systems are highly applicable to the software testing process, there have been few initiatives in this direction. As mentioned in [28], KM has seldom been researched in the software testing field, even though this field has a low knowledge reuse ratio, there are barriers to knowledge delivery, there is a loss of knowledge and software testing knowledge sharing environments are poor. Of the software tools whose functionalities address software testing experience (i.e., meta-knowledge or LLs) management, the following three, which are the latest and most important, are worthy of note.

3.3.1. Wikis for KM in software testing

Lee and Kettinger [52] propose a wiki-based approach to KM in software testing. This collaborative technology supports the creation of a knowledge repository related to several activities. These activities can be generic (e-learning, project management, technical support, collaborations or posting of general information) or specific to software tests. As regards software testing, they propose gathering software test documentation for test planning, the results of unit testing, defect lists and fixes, user acceptance testing reports, and post-analysis reports. Apart from enabling the storage of documents, the proposal also makes provision for agendas, blogs, collaborative and virtual brainstorming spaces. The essence of the system is to create a space for capturing and sharing explicit testing group knowledge with a view to exchanging and converting this explicit knowledge into tacit knowledge held by each of the team members.

This approach stands out as being extremely simple and relatively inexpensive to develop. However, knowledge and experience is retrieved from the wiki using text search tools on documents, blogs and other wiki entries. This is a sizeable obstacle to knowledge searching, dissemination and reuse, as (i) a lot of knowledge embedded in the texts can only be retrieved by examining wiki entries one by one, and (ii) wiki entries do not contain descriptive information about the knowledge that they include (e.g., testing processes, testing techniques, software types, applicable project phase) or the context where the knowledge contained in the wiki is applicable. In this respect, the proposal could be improved by using a knowledge categorization and entry labeling system similar to the one described for the RISE (Reuse in Software Engineering) project [53], where the wiki is enhanced by search technologies using ontologies and user-defined tags to describe knowledge and experiences.

3.3.2. Mobile software system testing

Ong and Tang [54] propose a knowledge management system focusing on mobile telephone system testing. The aim is to raise the quality and reduce the effort and cost of testing by exploiting experience gained earlier in later testing cycles (regression within system testing) or applying this experience to telephone models with similar features. The proposal uses a document management system for testing documents (requirements, testing manuals, and testing procedures), that is, for the more explicit knowledge, and a knowledge-based database to collect knowledge regarding problem-solving tasks, classified and characterized by its attributes (phone model, test features, issues, solutions and contact person).

The proposal covers interviews held to retrieve test engineers' expertise and knowledge and the establishment of a manager in charge of assisting, providing solutions and maintaining knowledge about the issues concerning each test feature. For knowledge reuse and sharing, it proposes a knowledge-based database search and query mechanism and knowledge sharing sessions—by means of which the aim is to achieve knowledge transfer among experts and other members of the testing group (i.e., yellow pages: accessibility for people with the referred knowledge and experience).

In this proposal, the knowledge is categorized and indexed in the knowledge-based database by means of key testing attributes. However, the knowledge stored is neither described nor formalized (context in which it is applicable, structure, types of knowledge artifacts it contains, etc.). Also, the system only focuses on domain knowledge, leaving aside technical and testing process management knowledge.

3.3.3. KM model-oriented software testing process

Xuemei et al. [28] propose the construction of a platform to support knowledge management activities in software testing. The proposed platform is based on a communication site. This site logs the problems raised by staff, the problem-solving process and related documents. Knowledge is contained in a knowledge database, expressed in natural language, and represented by an ontology. The proposal manages a knowledge map, which is also used as a knowledge yellow pages, and knowledge classification trees to classify the documents considered important for solving software testing problems. Finally, the system has a knowledge retrieval engine enabling users to run queries and, using the above knowledge map, filter and access the stored documents that are applicable to their query. If no documents are found, the system puts users in touch with the people who have the knowledge.

This proposal has important features, such as document indexation through knowledge classification trees, ontology-based knowledge representation, the use of search engines and yellow pages to contact the people who have specialized knowledge. In terms of knowledge management, however, this proposal falls down on documentary knowledge formalization and contextualization, and fails to consider less explicit knowledge (knowledge that is part of the experience of the testing groups) for formalization, sharing and reuse.

3.4. Current software tool weaknesses

The above proposals illustrate the fact that software testing knowledge management and, particularly, experience management could be improved, to a greater or lesser extent, by considering the following points:

- Structure and formalization: This is a common weakness of this kind of systems [55] that is an obstacle to the search, dissemination and reuse of knowledge and/or experience: the developed systems are systems targeting the management process rather than the object managed (i.e., the actual knowledge and/or experience). To be

precise, the method used to gather/represent this asset (through knowledge-based and document-based techniques) is not the best, as set out in Section 2.
- Contextualization: This kind of systems does not generally define the environment and the context where the knowledge and/or experience have emerged. This makes it harder to identify the situations where this asset is applicable and, therefore, reusable.
- Integral management: This aspect should be considered from two perspectives: managed object and management process. As regards the managed object, both explicit and implicit knowledge and/or experience must be considered, as should both the technical and management view. Not only should the management process improve knowledge and/or experience storage and search, but it also has to provide procedures to deliver this asset to users and thereby encourage reuse.

4. Proposed architectural model

The first thing to do is to define the scope of the proposed architectural model. Taking into account that our goal is to manage the software testing experience, the scope of this model is confined to this asset (i.e., meta-knowledge or LLs in Fig. 1). The goal therefore is not to manage knowledge about existing methods or techniques (i.e., methodological or technical knowledge) but knowledge derived from (gained by) experience that has the potential for reuse in future testing projects.

Also, the architectural model proposed here is designed to try to avoid the weaknesses identified in Section 3.4 as follows:

- Definition of a representation scheme for LLs that takes into account not only the actual lesson, but also the context in which it emerges and is used. This scheme structures and formalizes all the aspects of the LLs: the context of the lesson, the experience inspiring the lesson, the actual lesson and its reuses.
- Categorization of the descriptors of the LL context and unification of the values of the LL descriptors to improve indexation, access, searching and reuse.
- Integration of LL management processes—acquisition/collection, verification, storage, dissemination and reuse—with software testing activities. Thus, for example, besides the ordinary search facility, the proposed architectural model defines an active procedure for disseminating the lessons throughout all the testing activities. This assures that they can reach their potential users in the context where they are really applicable. This way, we aim to counter the failure of this type of systems to bridge the lesson distribution gap [56], that is to say, they do not bring LLs to the attention of the users when and where they are needed and applicable.

To present the proposed architectural model, Section 4.1 outlines the representation scheme used for the LL repository design, Section 4.2 deals with the processes that support the LL life cycle, and, finally, Section 4.3 shows how to enable the active dissemination of the LLs by the subsystem defined for this purpose. This explanation is merited because this mechanism is more complex to ordinary search facilities.

4.1. Software testing LL repository

As noted in Sections 3.3 and 3.4, the current approaches for managing software testing experience do not provide a formal (software testing) LL representation scheme. We therefore examined the current generic (domain-independent) LL representation approaches in order to find a preliminary approximation for such a scheme. In actual fact, we used, in view of its relevance, the LL representation approach proposed by Weber et al. [25,57] as a starting point. Note, however, that while the generic approaches are able to represent the LLs of any domain, they are not well suited to the inherent features of a specific domain (e.g., software testing). In fact, Weber et al. explicitly identify only the minimum relevant information for a generic LL. This approach should then be particularized–restructured and adapted-for software testing domain LLs.

This particularization process should be carried out bearing in mind that the resulting scheme has to be able to represent useful and applicable LLs by using a proposal closer to the final users. For the sake of usefulness, we have considered the experience and its later generalization and abstraction (see the level of abstraction of the lesson-specificity vs. generality-in, e.g., [57]). To achieve applicability, we have considered the context in which the LL emerges and where it was reused (see the discussion of the source and reuse of LLs in, e.g., [25]), as well as access to the people related to this experience (i.e., authors and reusers) by means, for example, of a yellow pages repository.

As illustrated in Fig. 2, the scheme proposed here has been organized accordingly (restructured as mentioned above) into the following five main blocks: (i) Generic, containing general descriptors identifying and relating the lesson to other lessons; (ii) Experience, containing the description of the events and situations by means of which the knowledge was gained (what happened, what alternatives there were, why things were done and the outcomes); (iii) Lesson Learned, containing the abstraction and generalization of this particular experience with the aim of later reuse; (iv) Context, containing the identification of the environment in which experience was learned or is applicable; and, finally, (v) Reuse, containing annotations regarding each time the lesson was reused. Additionally, the descriptors proposed by Weber et al. have to be adapted. In this respect, remember that Weber et al.'s representation approach considers only a minimum set of information. Taking this into account, we have (i) specialized descriptors in Weber et al.'s proposal as required (e.g., "Suggestion" was tailored as *LL Core*, and "Conditions" was tailored as Context) and (ii) added

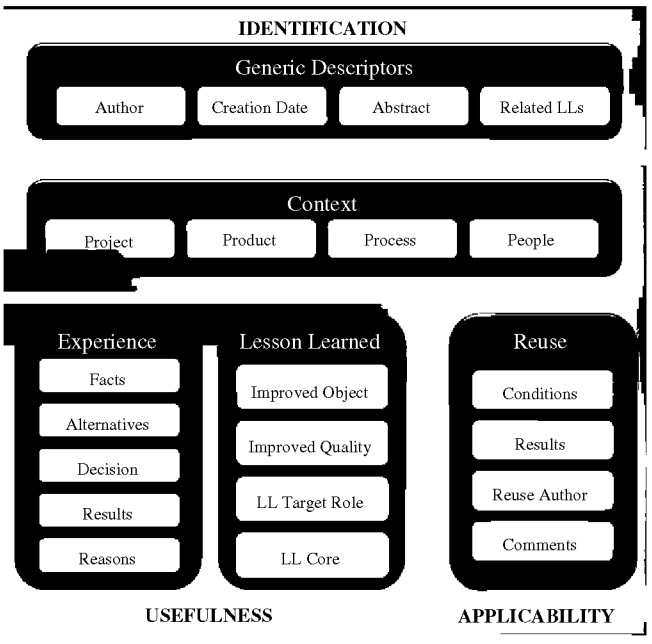

Fig. 2. Software testing LL scheme.

descriptors to extend the representativeness of the minimum set (e.g., descriptors in the Reuse block).

In the following we show the attributes defined for each of the above five blocks. These descriptors have been arranged by topics, subjects and activities that the experience deals with rather than the projects in which it was gained. This was done to improve access to the content of the LLs through a proper categorization of the lesson descriptors. In this respect, Harrison [58] analyses organizational experience, where, after 20 years of post-project reviews, final users have not used the stored reports mainly because of information access obstacles.

Note that we present some possible values of interest for the proposed descriptors (see, e.g., [1,38,39]) from a didactic point of view in the following sections. These are intentionally not exhaustive lists of values, since they are easily expandable to consider each particular situation/organization.

4.1.1. Generic descriptors

This block includes the general descriptors of the lesson used to identify and link the lesson to other related lessons:

- *Author*: Name and e-mail address of the creator.
- *Creation Date*: Date on which the LL was created.
- *Abstract*: Summary of the lesson.
- *Related LLs*: Lessons related to the lesson, stating, for each relation, a link to the related lesson and the type of established relation. The lessons in the repository can be linked by the following relation types: (i) generalization (a lesson in a wider or more general domain or context than the original setting); (ii) particularization (a lesson in a more restricted domain or context than the original setting); (iii) extension (a lesson in a complementary domain to the original setting); (iv) contradiction (lessons with contradictory results); and (v) compilation (a lesson resulting from the combination of several lessons).

4.1.2. Context

The context of a lesson identifies the environment and situation where a lesson emerges. Experience provides a historical perspective from which it is possible to gain an understanding of new situations and events [59]. Thanks to experience, we are able to put the current situations into perspective and recognize relationships between current and past events. There are many possible relation types, but there will always be some degree of similarity or difference between the context and the environment where an experience is gained and the context and environment in which it can be applied. The quantification of the similarity or antagonism between situations is an indicator of the applicability of the LL. For this reason, the lesson necessarily has to contain the characterization of the context where experience was gained.

Any software development *P*roject and, by inclusion, software testing, involves *P*eople carrying out engineering *P*rocess activities in order to produce software *P*roducts. For this reason, the definition of the context of a LL was driven by these 4 Ps. Table 1 shows the descriptors that formalize the context of a LL depending on the 4 Ps, and their possible values. Both the descriptors and the possible (Testing) *P*rocess values are extracted from the chapter on the software testing knowledge area in the *Guide to the Software Engineering Body of Knowledge* (*SWEBOK*) [1]. On the other hand, the descriptor Tester Independence within *P*eople (Testing Team) and its possible values complies with the *IEEE Standard for Software and System Test Documentation* [60]. Finally, the descriptors defined for *P*roject and *P*roduct are inspired on the characterization scheme

Table 1
Software testing LL context.

Descriptor type	Descriptor	Possible values
Project	Development model	Waterfall, spiral, incremental, model-driven, RAD, agile, extreme, scrum, test-driven
	Project size	<30, 30–300, 300–600, >600 function-points
	Project cost	<25,000€, 25,000–250,000€, 250,000–1000,000€, >1000,000€
	Business area	Scientific, retail, military, transport, engineering, communications
	Project goal	New development (adaptive, perfective, corrective, preventive), maintenance, platform migration
Product	Software architecture	J2EE, client–server, concurrent system, large platform, distributed
	Programming language	VBasic, C, Java, Perl, Prolog
	Hw–Sw integration	Embedded, integrated, independent
	Software type	Firmware, expert system, real-time software, management software, operating system
	Software acquisition	Tailored, commercial off-the-shelf software, outsourced package, customized standard product
Process	Testing activity	Risk management, cost management, schedule/planning management, resources management, configuration management test planning, test case generation, test environment development, test execution, test result evaluation, problem reporting, defect tracking
	Testing strategy	Specification based, code based, fault based, usage based, application type based, tester intuition based
	Testing technique	Ad-hoc, exploratory, equivalence partitioning, boundary-value analysis, decision table, finite state machine, random, control flow based, data flow based, error guessing, mutation, web based, GUI testing, object-oriented testing, component-based testing
	Testing level-target	Unit, integration, system
	Testing objective	Acceptance, installation, alpha, beta, functional, reliability, regression, performance, stress, recovery, configuration, usability
	Testing automation level	Automated, manual, mixed
People	Experience level	High (over 2 years' software testing experience), medium (from 6 months' to 2 years' experience), low (less than 6 months' experience)
	Tester independence	Embedded, internal, integrated, modified, classical

presented in [38,39], although the scope of this scheme—testing technique selection improvement—is narrow.

4.1.3. Experience

The process for obtaining the LLs from experience is far from simple, as it is troublesome to identify the real reasons why something works or has not worked well. Thus, according to Wheelwright and Clark [61], "the connection between cause and effect may be separated significantly in time and place. In some instances, for example, the outcomes of interest are only evident at the conclusion of the project. Thus, while symptoms and potential causes may be observed along the development path, systematic investigation requires annotation of the outcomes, followed by an analysis that looks back to find the underlying causes". On the other hand, even if the experience is analyzed and a finding (LL) is output, the LL is not always straightforward to apply, as its applicability has to be evaluated for a context that is not 100% identical. To be able to assess this applicability it is necessary to evaluate the context and to reproduce the thought process by which the finding (lesson) was reached, as a means of verifying the applicability to a new situation (see, e.g., the process-oriented approach to decision-making in [62]). It is important then to understand what happened, what problem emerged, what alternatives were assessed, what decisions were taken and what were the results and then proceed accordingly (see, e.g., steps in the Decision-Making Procedure [63]). In summary, it is necessary to represent the experience taking into account the following descriptors:

- *Facts*: Events that occurred or stated problems.
- *Alternatives*: Options taken into account to solve the above problem.
- *Decision*: Decision made as regards what alternative was chosen and why.
- *Results*: Outcomes.
- *Reasons*: Causes that led to the *Results*.

4.1.4. Lesson Learned

As mentioned above, a LL is obtained from the analysis of experience. This is the result of a process of generalization and abstraction and is represented here by three aspects: (i) the improved factor (that is, what testing process or task the application of this experience improves—*Improved Object* descriptor—and which quality of the testing process is improved—*Improved Quality* descriptor); (ii) the role that can exploit the LL (*LL Target Role* descriptor); and, finally, (iii) the LL core (*LL Core* descriptor). Table 2 outlines the defined LL descriptors.

The representation of the *LL Core* is based on the scheme presented in [64] for specifying experience packages by means of quality patterns. In these experience containers, the experience core is represented by means of ⟨problem, solution⟩ pairs. In our system, however, it will be represented by means of different characterizations or patterns taken from this initial scheme and tailored to the software testing field. Taking into account the key problems arising in the software testing field, we have established six pattern types:

- *Problem/risk prevention*: actions to be taken to prevent a problem or lessen a testing process risk.
- *Problem solution/risk contingency*: actions taken to solve a problem or minimize the impact of a materialized risk.
- *Technical impact*: consequences of using a particular testing tool, technique or approach (i.e., knowledge extracted after using the element).
- *Application criteria*: points to be taken into account about how to apply a testing tool, technique or method (i.e., knowledge to be taken into account before using the element).
- *Testing process improvement*: organizational and procedural modifications or adaptations of the testing processes, activities or tasks with the aim of achieving better practice.
- *Operating guideline*: criteria or recommendations on any of the activities within the testing process. It should set out how to organize, plan or materialize an activity. It can refer to a technical activity, such as the generation of test data or management data, or to test planning.

Of course, the representation of the LL core is completely extensible, providing for the inclusion of other patterns that can represent the LL core.

4.1.5. Reuse

The LL repository will contain one annotation for every time that a LL is reused. The aim is to be able to find out the contexts in which it was reused and the results of this reuse. The descriptors established for this purpose are as follows:

- *Conditions*: Environment and context in which the lesson was reused.
- *Results*: Positive, negative, neutral result and explanation of the result.
- *Reuse author*: Name and e-mail address.
- *Comments*: Opinions or suggestions for the reuse of the lesson or its improvement.

4.1.6. An example of a LL

With the aim of clarifying the descriptors defined in the above sections, Table 3 shows an example of a LL represented following the proposed representation scheme. Please note that we use the *problem/risk prevention* pattern for the *LL Core* descriptor. This

Table 2
Software testing LL descriptors.

Descriptor	Possible values/patterns	Pattern components
Improved object	Testing risk management, cost management, schedule/planning management, resources management, test reuse, test patterns generation, testing automation, testing execution, testing documentation	
Improved quality	Effectiveness, efficiency, optimization, adaptability, reliability, usability, comprehensibility, correctness, coherence, consistency, traceability	
LL target role	Project manager, testing analyst, testing designer, tester, development engineer	
LL core	Problem/risk prevention	Problem/risk + preventive actions
	Problem solution/risk contingency	Problem/risk + corrective actions
	Technical impact	Testing technique + consequences
	Application criteria	Testing technique + application criterion
	Testing process improvement	Activity + improvement
	Operating guidelines/procedure	Activity + guidelines

Table 3
Software testing LL example.

Generic	Author	*Author, Author@e-mail.com*		
	Creation date	*yyyy/mm/dd*		
	Abstract	Performance testing of a system with a three-tier architecture should be run starting with the bottom layer (data access), continuing with the business logic layer and ending with the presentation layer. This makes it easier to identify the source of the failures by inadequate response times		
	Related LLs	*NA*		
Context	Project	Development model	Incremental	
		Project size	Greater than 600 function-points	
		Project cost	250,000–1,000,000€	
		Business area	Retail	
		Project goal	New development	
	Product	Software architecture	J2EE	
		Programming language	Java	
		Hw–Sw integration	Independent	
		Software type	Management software	
		Software acquisition	Tailored	
	Process	Testing activity	Test execution	
		Testing strategy	Specification based	
		Testing technique	Web based	
		Testing level-target	System	
		Testing objective	Performance	
		Testing automation level	Mixed	
	People	Experience level	Medium	
		Tester independence	Integrated	
Experience	Facts	Performance testing of a three-tier system is divided into three testing tasks, one for each layer: data access, business logic and presentation There are two automated testing tools: the first monitors database management system access times and the second monitors application response times, as well as examining stress and concurrency		
	Alternatives	A priori, any of the six order combinations is feasible, although intuitively there are two reasonable combinations: 1st: data access, 2nd: business logic, 3rd: presentation 1st: presentation, 2nd: business logic, 3rd: data access		
	Decision	To prevent stoppages in view of testing team resource availability, the preferred option was to start with the presentation layer and finish with the data access layer		
	Results	The causes of poor response time were hard to identify during presentation layer testing		
	Reasons	As the code has not been debugged, it is not possible to single out the causes of the poor response times during testing		
Lesson learned	Improved object	Testing execution		
	Improved quality	Efficiency		
	LL target role	Project manager, Testing analyst		
	LL core	Problem/risk prevention	Problem/risk	Determine the optimal sequence for running performance tests ruling out the problems for identifying the source of the detected failures and the possible unnecessary repetition of tests
			Preventive actions	Run tests in the following order: 1st: data access, 2nd: business logic and 3rd: presentation.

lesson was collected during the prototype evaluation process described later in Section 5, which has not yet been reused (i.e., the descriptors of the Reuse block are empty).

4.2. Software testing LL management

LLs are managed through subprocesses that are related to each other forming what is known as the LL life cycle. These subprocesses are acquisition (or collection), verification, storage, dissemination and reuse [25,27]. They are described in the following considering how they are integrated into the testing process.

4.2.1. Acquisition/collection

The acquisition or collection of software testing LLs is the first subprocess to be considered. Its goal is to capture and then represent experience about software testing processes with a view to reuse.

Possible LLs can be identified in the testing process as follows:

- *Everyday testing tasks*: The performance of everyday testing activities is a source of experiences from which lessons can be learned.
- *Lessons learned sessions*: Meetings resembling post-project reviews can be held at particular, important times in the testing process, especially at the end of each testing

phase, where the goal is to identify, share and discuss experiences collectively in open sessions while they are still fresh in the participants' minds [58]. The milestones at which these sessions are held will depend on the testing model, where the usual procedure will be for them to be held at the end of each testing cycle, at software installation time and at the end of the testing project.
- *Software failures in the operational system*: A software failure in an operational system indicates that the failure was not detected during the testing process. An analysis of the reason for this can help to identify preventive and/or corrective actions to assure this does not happen again.

Irrespective of when LLs are identified, the collection process can be active or passive [17]. Active collection means that some organizational mechanism scans communication processes in order to detect LLs. Passive collection means that workers believe that an experience merits being considered as a LL. In any case, and whichever the collection method used, the storage of LLs in the repository could be supervised or unsupervised [17]. That is to say, the LLs can be previously verified before storage, as in the subprocess outlined below, or stored directly by the LL authors.

4.2.2. Verification

The goal of this subprocess is basically to check that a LL is correct, consistent, relevant, and not redundant before it is stored in the repository (see, e.g., [25,47,59]). The result of this subprocess—which will be carried out by an expert, authorized person, or a team (e.g., a team of expert testers and supervised by a KM Manager)—will be to accept, modify or reject the LL [17,59].

Obviously, this additional verification work should be taken into account as part of their job responsibilities, and it should not imply an extra/unrewarded workload. Otherwise, there is a danger of this additional work not being done properly. Note that the omission of this subprocess makes the LLs easier/cheaper to incorporate, as no dedicated resources are required. In return, though, there is no guarantee of LL correctness, consistency, relevance or non-redundancy, with all that this implies.

4.2.3. Storage

This subprocess addresses issues related to the lesson representation and indexation, formatting, and repository framework [25,57]. In this sense, the descriptors of the proposed representation scheme used for the software testing LL repository defined in Section 4.1 precisely support the representation/storage of the lesson and relate it to others. This last point enables a (verified) LL to establish a network of relations to other LLs in the repository (that is to say, it enables "navigation" among LLs).

4.2.4. Dissemination

The ultimate aim of a LL repository is reuse, and repository accessibility and lesson dissemination is a key factor. This aspect is not, however, always effectively resolved. Harrison [58], for example, analyzed the initiatives for capturing software engineering LLs by holding post-project review meetings and concluded that post-project review reports were seldom accessed after the end of the project for two main reasons: (i) non-existence of formal mechanisms to assure the dissemination of the information beyond the review meeting participants, and (ii) inaccessibility of the information contained in the reports (i.e., unsatisfactory indexation of the contents).

We propose two different strategies for disseminating software testing lessons. The first is active dissemination [17] or selective casting, where a software testing LL system disseminates the LLs according to a number of preset parameters (e.g., dissemination of potentially useful LLs at the start of each testing activity by means of the Context block Testing Activity descriptor). The second is passive dissemination [17] or on-demand searching, where the user is responsible for communicating with a software testing LL system and requesting the delivery of LLs. Both strategies will be described in more detail below:

- *Selective casting*: A software testing LL system using this strategy sends LLs (e.g., via e-mail) to all the individuals that meet the preset criteria and are considered potential users of those LLs. Depending on the criterion applied to determine the recipients, selective casting will be either:
  - *Personal casting*: When a project kicks off or changes, potentially applicable LLs are broadcast to the human resources assigned to the project, considering: (i) the project and product descriptors defined in the repository (project/product-based casting), and (ii) the profiles of potential users (*LL Target Role* descriptor). Due to its complexity, Section 4.3 proposes a specific architecture for this kind of dissemination.
  - *Narrow casting*: Users select the types of LLs about which they would like to be informed if they are entered or modified in the LL repository. Users subscribe in compliance with the descriptors that are part of the repository indexes.
- *On-demand searching*: The user activates the communication process by sending a request to a search engine. The search engine will query the repository and return a report that the user will be able to filter according to the defined search indexes. Depending on search parameter specificity, the search engine can work to three separate on-demand searching methods:
  - *Descriptor-based search*: Users define the search values for the selected descriptors, generating a report containing the lessons that match the search term.
  - *Dynamic search*: This is based on the method described in [65]. It involves displaying the LL repository content by dynamically and incrementally selecting and filtering index descriptors. Search criteria are gradually defined dynamically, and the results are represented spatially so that the user can refine the search.
  - *Similarity search*: The query is specified by assigning search values to any of the repository index descriptors and an impact factor or relative search importance to each one. The search is run by a similarity function weighted by the impact factors and based on the conceptual proximity of the query descriptor values and the existing lessons. The conceptual proximity value is based on Tversky's contrast model [66]. The resulting set is compared with predefined similarity thresholds that the user can modify depending on the candidate lessons for evaluation.

There are other dissemination strategies (e.g., broadcasting), but they are not usually recommended (e.g., they have very low hit rates and generate too many useless messages).

4.2.5. Reuse

Reuse is the ultimate goal of any LL system because it is the subprocess around which all such components revolve. The goal is to apply one or more LLs in the project at hand, and the benefit of lesson reuse will be directly related to the user-perceived utility of the LL system [67]. In the proposed architectural model (see Section 4.1.5), the repository contains an annotation for every time the lesson is reused. This information is valuable for both ascertaining the applicability of the lesson and putting all

the individuals with experience in a particular subject into touch.

Specifically, LLs in software testing can be reused at three different levels:

- *Testing project level*: The LLs from product testing are applied in the same project, in regression testing or in different testing iterations.
- *Product level*: The lessons are applied throughout the whole product life cycle (i.e., including maintenance).
- *Organizational level*: Lessons are applied to projects or products with similar features.

4.3. Architecture for the proposed active LL dissemination subsystem

By monitoring software testing processes and the state of the LL repository, potentially relevant lessons can be proactively notified to the testing team. Thanks to this integration, the experiences stored in the LL repository are delivered to the testers in the context where they could really be useful. Specifically, an active LL dissemination subsystem acts as an assistant that proactively suggests LLs for reuse. This is a very effective approach, as the user does not need to know about or even how to use the search system efficiently. Although this approach is essential for bridging the "lesson distribution gap" [56] (one of the main difficulties with deploying LL systems in any domain), very few existing systems use active dissemination [68].

Fig. 3 presents the architecture for the proposed active LL dissemination subsystem. In this proposal, the Testing Projects Map Builder dynamically forms an image (a project map) of each testing project from the information contained in the Testing Projects Management System. This project map contains the context descriptors defined in previous sections, the project planning and state, and human resources allocation and their roles. Note that this project map may contain incomplete information, as the information concerning the testing project context grows and changes incrementally as it advances (Project Update Message).

These project maps are used to search the LL repository by the context descriptors defined in Table 1, where lesson applicability to the project at hand is evaluated considering the previously defined similarity search method (Similarity Search Server). The Lesson Learned Router will route the lessons that are above the applicability threshold to the testing engineers concerned.

On the other hand, the Repository Monitoring Module will detect and report the modification of the repository (e.g., creation or modification of content or context) to the Applicability Evaluation Module. This module will identify the projects to which the lesson is applicable through the Similarity Search Server, and the Lesson Learned Router will distribute it to the appropriate engineers (potential users).

5. Prototype

5.1. Features

Based on the proposed architectural model defined in Section 4, we have designed and built a web-based software testing LL system prototype with the aim of evaluating the strengths and weaknesses of the core of the proposed model.

This core addresses: (i) the LL representation scheme proposed in Section 4.1, and (ii) the basic subprocesses of the software testing LL management (LL life cycle in Section 4.2). The reason behind this decision is that the core of any information system (and LL systems are a particular case) is the information (scheme) in itself and the essential processes managing this information (scheme). It makes no sense to add other proposals, subsystems, and/or functionalities if users do not agree with the LL representation scheme and/or its essential management subprocesses, since they would work upon that groundwork. This prototype should therefore be evaluated before considering adding additional aspects (e.g., the LL dissemination subsystem presented in Section 4.3).

Taking the above into account, the features of the prototype developed are as follows:

- *Storage according to the proposed LL representation scheme*: The LLs are stored in a relational database whose conceptual scheme is directly derived from the proposal outlined

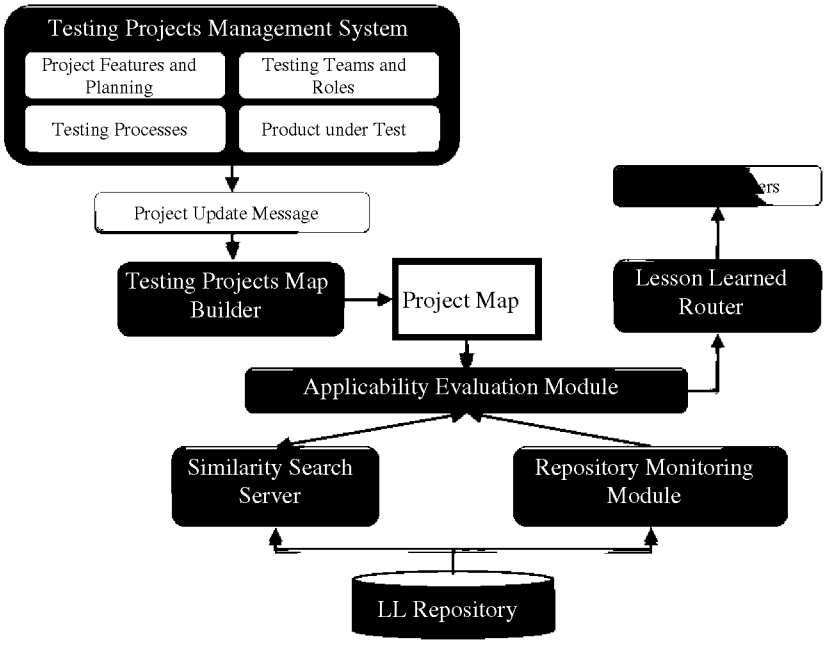

Fig. 3. Proposed active LL dissemination subsystem.

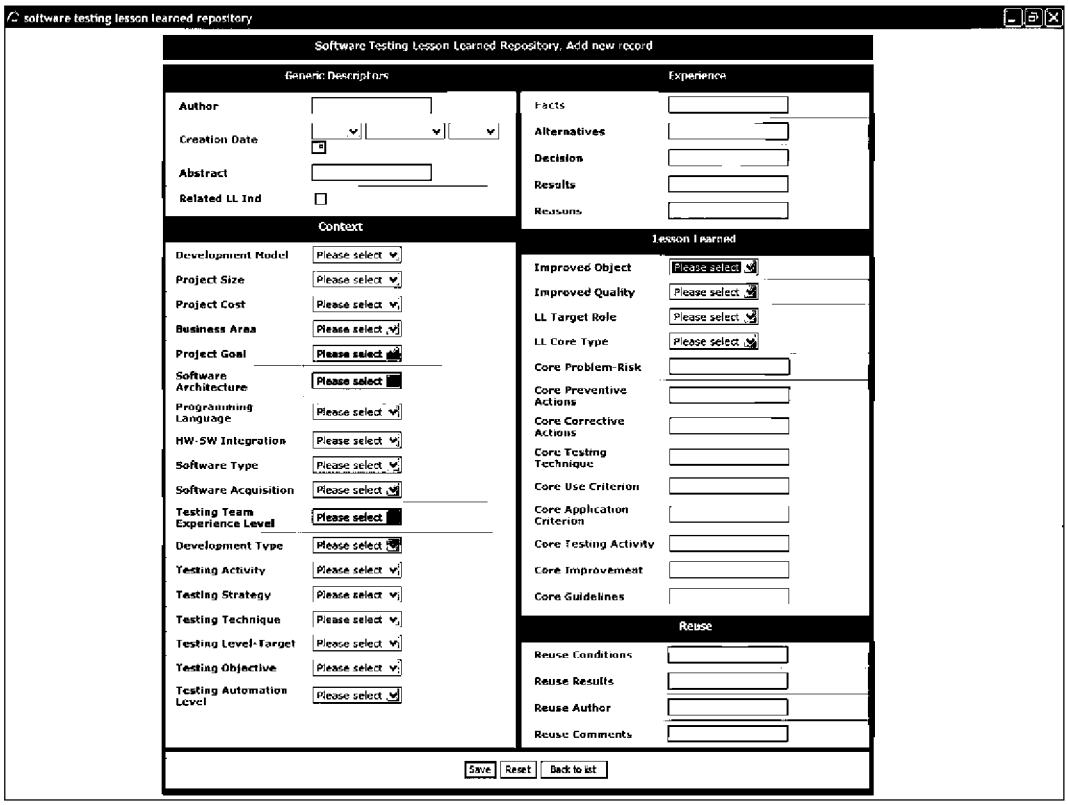

Fig. 4. LLs submission web form.

in Section 4.1. This enables textual and descriptor-based searches (see comments in Section 4 about the searching and indexing problems stated by Harrison [58]).

- *Characterization of descriptors*: All the LL context descriptors (see Table 1), and the Improved Object, Improved Quality and LL Target Role descriptors (see Table 2) are defined as user-customizable entities. This makes the prototype easily adaptable to any testing environment and assures the standard use of the attribute values. The values for our prototype are presented in Tables 1 and 2.
- *Passive collection*: The user uses a web form (see Fig. 4) to enter the LL in the system specifying the values of the defined descriptors. Additionally, users can link new to existing lessons using the relations specified in Section 4.1.
- *Parameterizable verification*: The submitted lessons can be entered in the repository directly (i.e., storage subprocess) or subject to acceptance by an expert or authorized person (i.e., verification subprocess). In our case study, this lesson acceptance condition is parameterizable and was verified by a team composed of expert testers and supervised by a KM Manager. Note that if LLs are to be verified before storage, the result of completing the web form shown in Fig. 4 will be to merely report the LLs to the person or group responsible for verification. This person or group will then be responsible for storing the approved LLs.
- *Passive dissemination by descriptor-based search*: Repositories are searched using the descriptors that have been defined to represent the LLs, thus deploying the descriptor-based search method mentioned in Section 4.2.4. The searches are text-based for textual descriptors, and value-based for descriptors that have preset values (i.e., context descriptors *Improved Object*, *Improved Quality*, *LL Target Role*, in Table 1, and LL core pattern, in Table 2). Fig. 5 shows the search web form.
- *Reuse*: User feedback about the reused lesson can be entered in each lesson in the repository. This way, after reusing a lesson, a user enters the reuse conditions, the outcome and any comments or suggestions that are considered interesting for later lesson reuse into the system.
- *Knowledge directory*: Finally, the prototype additionally includes a yellow pages repository (cf. Section 4.1) structured on the basis of the context of the lessons and on the improvements they provide. This way, the context descriptors and the improvement descriptors (Improved Object and Improved Quality) are retrieved for each stored lesson and associated with the testing engineers that have created or reused the lesson. Thus, as Fig. 6 shows, software testing knowledge is related to the person who has and/or uses the knowledge. This knowledge directory serves as support for promoting a knowledge personalization strategy [69], which improves the dissemination of the more tacit knowledge.

Finally, from a technical point of view, the prototype was built using WAMP. WAMP is an open source software suite comprising Apache server, MySQL database and PHP programming for Windows. WAMP was selected mainly on the grounds of the low cost and effort associated with the development process.

5.2. Evaluation

5.2.1. Scope and environment

Taking into account the core established in the beginning of Section 5.1, the scope of the evaluation carried out considering

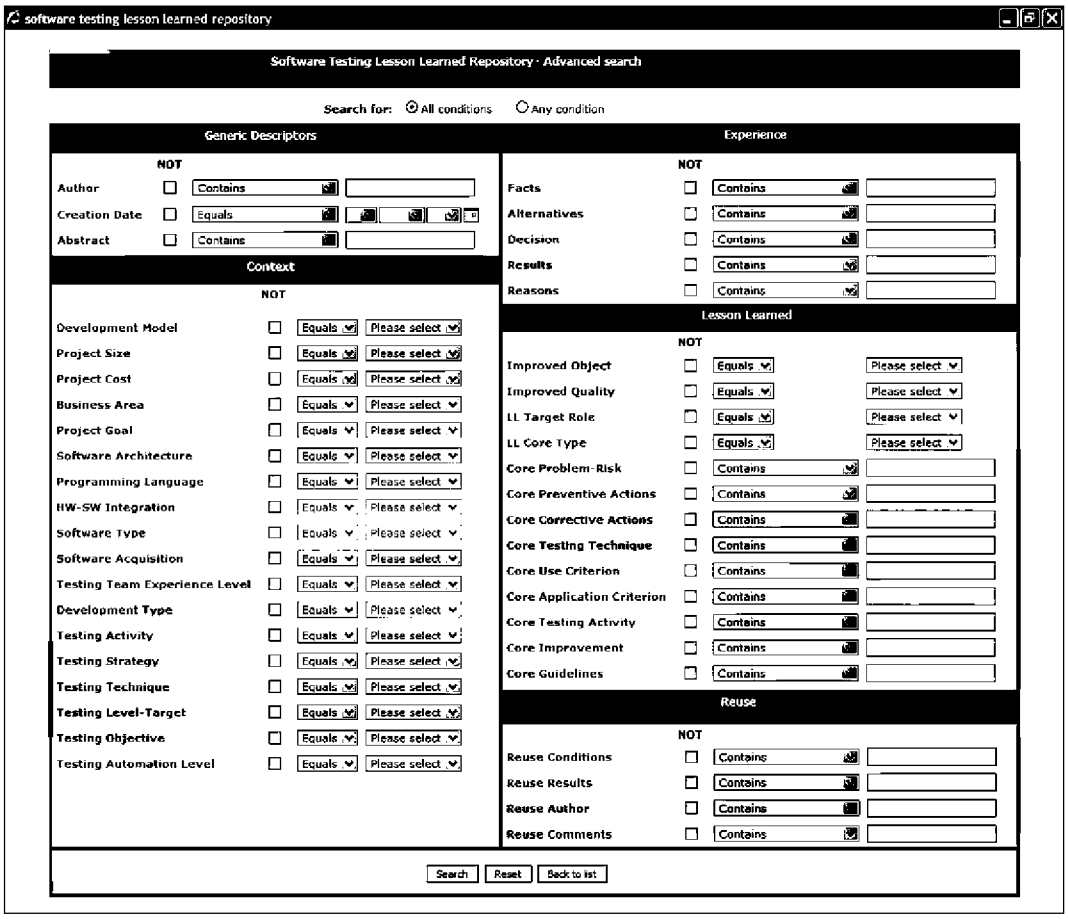

Fig. 5. LLs descriptor search form.

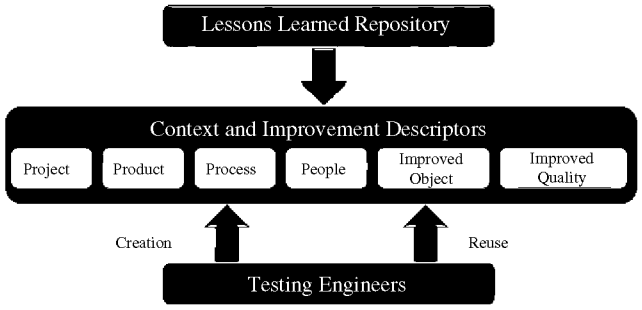

Fig. 6. Knowledge directory.

the developed prototype includes the adequacy of the LL representation scheme, and the support for the basic subprocesses of the software testing LL management (i.e., feasibility to support the representation/storage of a LL and the basic management activities). Note, in this respect, that a software tool can never guarantee an effective organizational deployment of the management subprocesses, as it is concerned with only the provision not the effective use of functionalities. That is to say, we have evaluated how our prototype supports the core aspects considered—not their organizational institutionalization—in order to draw conclusions about the proposed architectural model. In fact, institutionalization and its benefits (e.g., from the advantages of reuse) would require a medium- to long-term evaluation, which is obviously beyond the scope of this paper.

The evaluation has been done during the performance and stress tests of two web environment management software projects within a Spanish software development company. The main features of these projects are as follows:

- Project A:
  - Summary: Intranet-based information system supporting the management and control of order preparation and delivery to customers of a distribution enterprise. The users employ workstations and mobile devices to control stock.
  - Development aspects: The incremental development model was used with the J2EE standard and the model-view-controller pattern. It was a new development project with a total size of 1014 function points and a total final cost of 817,211€.
  - Testing considerations: Manual code inspections were carried out for the most critical components, and performance monitoring tools were used to evaluate resource use in the DBMS and the applications server. The main testing goals were the evaluation of the system resource consumption and the verification of response time, as well as the evaluation of the requirements related to massive data management.
- Project B:
  - Summary: Internet-based information system supporting the acquisition of goods on offer and the distribution of information by electronic media to customers (e-mail, SMS, etc.).

- Development aspects: This new development project used the incremental development model. J2EE standard and the model-view-controller pattern were used for this project, whose total size was 376 function points and total cost 286,000€.
- Testing considerations: The most critical aspect in this project was to evaluate the massive electronic communications delivery.

From the above description, it follows that these two projects have similar contexts. We explicitly looked for this similarity in order to retrieve LLs (non-empty searches) using passive dissemination (descriptor-based search) of the prototype mainly to promote reuse between the two projects.

The testing team was composed of seven members for Project A and four members for Project B, with different profiles and workloads (see Table 4 for details). Team members had medium and high experience in testing activities but no experience in the use of LL systems (and KM systems in general).

5.2.2. Process

The evaluation process began with the performance and stress testing activities in Project A (3.5 months) followed by B activities (2 months). For both projects, the members of the testing team identified LLs both during the everyday testing tasks within these periods and in the LL sessions scheduled at the end of each testing phase. Each identified LL was verified (in this case, by a team composed of expert testers and supervised by a KM Manager) before its storage in the LL repository. As a result, 51 non-redundant LLs were identified. Tables 5 and 6 summarize, respectively, these lessons classified according to their type (technical or planning LLs) and when LLs were identified (everyday testing tasks or LL sessions).

Table 7 shows the approximate times taken to register LLs during the evaluation process. We find that the time/lesson ratio is much greater for lessons elicited from LL sessions than for lessons elicited from everyday testing tasks. This is due to differences in the lesson identification process. There is not usually a planned LL search, analyzing facts, decisions, results and causes, to elicit lessons from everyday testing tasks; they are often identified more or less on the fly without any organized deliberation. By definition, LL sessions involve an organized analysis of potential LLs, which are shared with the other members of the team, discussed and agreed upon by participants. All these are time-consuming activities and, therefore, it takes on average longer to register the lessons.

Table 5
LL distribution by type.

	Project A	Project B	Total
Technical LLs	24	15	39
Planning LLs	4	8	12
Total	28	23	51

Table 6
LL distribution by acquisition/collection time.

	Project A	Project B	Total
Everyday testing tasks	18	14	32
LL sessions	10	9	19
Total	28	23	51

Table 7
Distribution of LL registration times depending on acquisition/collection time.

	[0–15 min]	[15–30 min]	[30 min–1 h]	[>1 h]
Everyday testing tasks	10	15	6	1
LL sessions	6	3	5	5
Total	16	18	11	6

Note that the testing team members were expressly encouraged at the beginning of Project B to search LLs in the repository throughout the project. This way, they put into practice the passive dissemination functionality by retrieving previously discovered LLs (from Project A) that could be useful in this project. Generally, the stakeholders found this facility to be interesting, although, as discussed in Section 5.2.3, they discovered that its use had some limitations.

5.2.3. Results

At the end of the case study, the most interesting aspects referred by stakeholders were:

- The first suggestion was to identify the project in which the lessons were learned within the LL representation scheme to improve the search for lessons that are reusable in the same project (e.g., later on in the same project: regression testing).

Table 4
Stakeholders involved in prototype evaluation.

Project	Human resources	Role	Function
Project A	1 DB2 (DBMS) and System Z (central host) environment specialist	Testing analyst, testing designer	Central system (automatic and manual) test analysis and design
	1 DB2 (DBMS) and System Z (central host) environment specialist	Tester	Central system (automatic and manual) testing
	1 WAS (applications server) specialist	Testing analyst, testing designer	Applications server (automatic and manual) test analysis and design
	1 WAS (applications server) specialist	Tester	Applications server (automatic and manual) testing
	1 DB2/System Z environment tester	Tester	Manual DBMS DB2 interface code inspection
	1 WAS environment tester	Tester	Manual applications server code inspection
	1 Testing manager/project manager (part time)	Project manager	Project management
Project B	1 DB2 (DBMS) and System Z (central host) environment specialist	Testing analyst, testing designer, tester	Central system (automatic and manual) test analysis, design and implementation
	1 WAS (applications server) specialist	Testing analyst, testing designer, tester	Applications server (automatic and manual) test analysis, design and implementation
	1 Communications environment specialist	Testing analyst, testing designer, tester	Communications test analysis, design and implementation
	1 Testing manager/project manager (part time)	Project manager	Project management

- Although the *LL related* descriptor was a positively rated concept, stakeholders pointed out that it was hard to use in a real LL system: it would be difficult to identify the related LLs for a given LL in a repository managing a huge number of LLs.
- The lessons identified during testing were more relevant and precise than those identified in the sessions at the end of each testing phase. Although the testing time frame in both projects was not long (2 and 3.5 months), time was found to dull the perception of experience and was an obstacle to the characterization and generalization processes. Therefore, engineers should be encouraged to start up the process of gathering LLs as early on as possible.
- LL repository coherence and consistency maintenance proves to be a really tiresome, albeit indispensable, task for maintaining system reliability and effectiveness. During the verification subprocess, 12 redundant LLs were identified, even though the two projects were short. Due to the likelihood of redundancy being high, LLs should be verified before storage in the repository.
- With the aim of making the dissemination more efficient, stakeholders suggested that any lesson that could be of interest to the project should be distributed to the testing team members considering: (i) project-specific features and (ii) their profile. This proposal fits the personal casting method (i.e., active dissemination) introduced in Section 4.2.4 and detailed in Section 4.3, and not implemented in the prototype. In short, the proposal lays the emphasis on a reduction of the workload associated with the search for applicable LLs, but does not rule out passive dissemination, which, for example, is necessary as a complement to personal casting as the project advances.
- Stakeholders pointed out the potential system acceptance-related risk of deploying a system such as the above, owing mainly to the negative connotations of having to post an error made (i.e., negative LL). Note, in any case, that: (i) one of the key points of a KM program (part of which involves the deployment of a system such as the above) is the introduction of mechanisms that guarantee full employee commitment [70], that is, a company culture based on exchange and collaboration; and (ii) not all lessons are derived from negative experiences, and the publication of negative experiences should be praised as part of an effort to prevent mistakes from being made within the organization again in the future.
- Another factor mentioned as a potential risk in commissioning a LL system is the shortage of time or unavailability to both enter (i.e. verification and generalization of experiences and their transformation into actual lessons) and retrieve (passive dissemination in the prototype) the lessons. With regard to the entry of lessons, again note that the management must be committed to the KM program, seeing it as a competitive investment and providing organizational mechanisms to support the workload required to enter the lessons to the repository. With regard to the retrieval of the lessons from the repository, active dissemination could reduce this risk.
- Testing team members commented on the workload required to register LLs in the prototype. Most of these comments signaled that it took too long to register LLs elicited from LL sessions, especially compared with the time taken to register LLs from everyday testing tasks. But, only two of the eleven team members stated that the time taken was clearly unacceptable. Opinions on the cost-effectiveness (potential benefit/cost ratio) of the process were more divergent. There were contradictory opinions on this point, which we believe to be justified by the fact that one of the variables (potential benefit) is an understandably personal appraisal that is neither objective nor verifiable (especially by means of an evaluation such as is conducted here). Again two team members (the same two as above) stated that it was clearly not a cost-effective process. The other team members considered it to be cost effective to varying degrees, where on average they were inclined to think that it could turn out to be quite cost effective.

6. Conclusions

As discussed earlier in the paper and in referenced works, managing experience derived from software testing activities is an important factor that could benefit software development. However, significant weaknesses have been detected in the current systems addressing software testing experience management, primarily: (i) unstructured and unformalized LLs, (ii) uncontextualized LLs, and (iii) failure to integrate the LL management processes with the software testing activities.

In this paper we propose an architectural model for software testing LL systems specifically designed to avoid the above weaknesses and to take into account two basic goals: LL usefulness and applicability. The key features of this model are:

- It defines a representation scheme enabling a structured formalization of the LL repository. Many of the descriptors in this scheme take their values from pre-established sets, and even the descriptors with textual values reflect a very pronounced structure. The transformation of experience into lessons involves a process of representation/coding, which can, of course, end up leading to a loss of the most tacit part of the knowledge. This is, however, an acceptable loss and is the way to get an accessible and organized repository [71]. Otherwise, what we would have is a pile of information containing inscrutable experience.
- It includes the LL context in the above-mentioned scheme, since the similarity or antagonism between the context where an experience is gained and the context in which it can be applied is an indicator of the applicability of the LL. For this reason, the lesson representation necessarily has to contain the characterization of the context where the experience was gained, which, in this proposal, was done using the 4 Ps: Project, Product, Process and People.
- It integrates LL management processes—acquisition/collection, verification, storage, dissemination and reuse—with software testing activities. Thus, for example, an active LL dissemination subsystem was proposed to assure that LLs can reach their potential users in the testing context where they are really applicable.

The evaluation of the developed prototype focuses on the adequacy of the LL representation scheme, and the support for the basic subprocesses of software testing LL management (i.e., feasibility to support the representation/storage of a LL and the basic management activities), thus considering the core of the proposed architectural model.

The most interesting findings pointed out by the stakeholders taking part in the evaluation process refer to both the representation scheme and integration. Their main proposal concerning the representation scheme was to add a new descriptor for directly and explicitly stating the project in which the lessons were learned

(e.g., *Project Id* for Project in the Context block). Stakeholder suggestions on how to improve integration referred to the following subprocesses: acquisition/collection (early testing LL identification was strongly recommended), verification (testing lessons should be verified before they are entered in the repository), and, finally, dissemination (they all thought that active testing LL dissemination, defined in the model but not implemented in the prototype, was an interesting option for reducing the search-related workload).

Finally, the evaluation of the developed prototype corroborated some of the problems of deploying this kind of systems echoed in the current literature: misgivings about a system that publishes errors and the extra workload required to enter and retrieve experiences. Although there are potential technical solutions for allaying some of these problems (see, e.g., experience retrieval through active dissemination in Section 4.3), KM program success depends on more than just acquiring tools (see, e.g., [70]). Employees are normally reluctant to share their knowledge, as it is their advantage. If, on top of this, knowledge sharing increases the employee workload, the KM program will definitely fail. Therefore, management involvement and backing is essential [70]. Management will have to endorse their involvement and backing by means of policies aimed at promoting experience sharing and reuse (e.g., employee incentive schemes, public recognition, etc.).

References

[1] IEEE Computer Society, SWEBOK, A Guide to the Software Engineering Body of Knowledge, 2004. <http://www.se.sjtu.edu.cn/sites/se/gb/CCSE/Swebok_Ironman_June_23_%202004.pdf> (accessed 12.05.11).
[2] B. Beizer, Software Testing Techniques, second ed., Van Nostrand Reinhold, New York, 1990.
[3] J.J. Marciniak (Ed.), Encyclopaedia of Software Engineering, second ed., John Wiley & Sons, New York, 2002.
[4] G.J. Myers, The Art of Software Testing, John Wiley and Sons, New York, 1979.
[5] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan-Kauffman, Amsterdam, 2007.
[6] L. Crispin, J. Gregory, Agile Testing: A Practical Guide for Testers and Agile Teams, Addison-Wesley, Boston, 2009.
[7] L.C. Briand, Novel applications of machine learning in software testing, in: Proceedings of the 8th Conference on Quality Software, QSIC'08, United Kingdom, 2008, pp. 3–10.
[8] T.Y. Chen, H. Leung, I.K. Mak, Adaptive random testing, in: Proceedings of the 8th Conference on Quality Software, QSIC'08, United Kingdom, 2008, pp. 320–329.
[9] A.C.C. Natali, A.R.C. Rocha, G.H. Travassos, P.G. Mian, Integrating verification and validation techniques knowledge into software engineering environments, in: Proceedings of 4as Jornadas Ibeoamericanas de Ingeniería del Software e Ingeniería del Conocimiento, JIISIC'04, Spain, 2004, pp. 419–430.
[10] N. Juristo, A.M. Moreno, S. Vegas, Towards building a solid empirical body of knowledge in testing techniques, ACM SIGSOFT Software Engineering Notes 29 (2004) 1–4.
[11] J. Liebowitz, Knowledge Management Handbook, CRC Press, Florida, 1999.
[12] C.W. Holsapple, K.D. Joshi, A formal knowledge management ontology: conduct, activities, resources, and influences, Journal of the American Society for Information Science and Technology 55 (2004) 593–612.
[13] D. Liu, I. Wu, Collaborative relevance assessment for task-based knowledge support, Decision Support Systems 44 (2008) 524–543.
[14] K.M. Wiig, Knowledge management: where did it come from and where will it go?, Expert Systems with Applications 13 (1997) 1–14
[15] J. Andrade, J. Ares, R. García, J. Pazos, S. Rodríguez, A. Silva, Formal conceptualisation as a basis for a more procedural knowledge management, Decision Support Systems 45 (2008) 164–179.
[16] A. Abecker, A. Bernardi, K. Hinkelmann, O. Kuhn, M. Sintek, Towards a well-founded technology for organizational memories, in: B. Gaines (Ed.), Artificial Intelligence in Knowledge Management, Papers from the 1997 AAAI Spring Symposium, AAAI, Stanford, 1997, pp 1–7.
[17] G. van Heijst, R. van der Spek, E. Kruizinga, Corporate memories as a tool for knowledge management, Expert Systems with Applications 13 (1997) 41–54.
[18] R. Dieng, O. Corby, A. Giboin, M. Ribière, Methods and tools for corporate knowledge management, International Journal of Human–Computer Studies 51 (1999) 567–598.
[19] A. Gómez-Pérez, M. Fernández-López, O. Corcho, Ontological Engineering with Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web, Springer-Verlag, London, 2004.
[20] M. Uschold, M. Grüninger, Ontologies: Principles, Methods and applications, Knowledge Engineering Review 11 (1996) 93–155.
[21] G. Simon, Knowledge acquisition and modeling for corporate memory: lessons learnt from experience, in: Proceedings of 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, KAW'96, Canada, 1996, pp. 41.1–41.18.
[22] O. Kühn, A. Abecker, Corporate memories for knowledge management in industrial practice: prospects and challenges, Journal of Universal Computer Science 3 (1997) 923–954.
[23] J. Andrade, J. Ares, R. García, S. Rodríguez, S. Suárez, Lessons learned for the knowledge management systems development, in: Proceedings of the 2003 IEEE International Conference on Information Reuse and Integration, IRI 2003, USA, 2003, pp. 471–477.
[24] J. Andrade, J. Ares, R. García, S. Rodríguez, M. Seoane, S. Suárez, Guidelines for the development of e-learning systems by means of proactive questions, Computers and Education 51 (2008) 1510–1522.
[25] R. Weber, D.W. Aha, I. Becerra-Fernández, Intelligent lessons learned systems, Expert System with Applications 20 (2001) 17–34.
[26] A. Aurum, F. Daneshgar, J. Ward, Investigating knowledge management practices in software development organisations – an Australian experience, Information and Software Technology 50 (2008) 511–533.
[27] A. del Moral, J. Pazos, E. Rodríguez, A. Rodríguez-Patón, S. Suárez, Gestión del Conocimiento, Thomson-Paraninfo, Madrid, 2007.
[28] L. Xuemei, G. Guochang, L. Yongpo, W. Ji, Research and application of knowledge management model oriented software testing process, in: Proceedings of the 11th Joint Conference on Information Sciences, JCIS 2008, China, 2008.
[29] R.L. Glass, R. Collard, A. Bertolino, J. Bach, C. Kaner, Software testing and industry needs, IEEE Software 23 (2006) 55–57.
[30] R. Jain, S. Richardson, Knowledge partitioning and knowledge transfer mechanisms in software testing: an empirical investigation, in: Proceedings of the 1st Workshop on Advances and Innovations in Systems Testing, USA, 2007.
[31] E.V. Berard, Bringing testing into the fold, I EEE Software 13 (1996) 91–92. 95.
[32] S. Vegas, N. Juristo, V.R. Basili, Identifying Relevant Information for Testing Technique Selection. An Instantiated Characterization Schema, Kluwer Academic Publishers, Boston, 2003.
[33] N. Juristo, A.M. Moreno, W. Strigel, Software testing practices in industry, IEEE Software 23 (2006) 19–21.
[34] I. Nonaka, H. Takeuchi, The Knowledge-Creating Company. How Japanese Companies Create the Dynamics of Innovation, Oxford University Press, Oxford, 1999.
[35] R. Almeida, L.S. Mota, F.F. Rosa, Using knowledge management to improve software process performance in a CMM level 3 organization, in: Proceedings of the 4th International Conference on Quality Software, QSIC'04, Germany, 2004, pp. 162–169.
[36] L.C. Briand, On the many ways software engineering can benefit from knowledge engineering, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE 2002, Italy, 2002, pp. 3–6.
[37] J. Arent, M.H. Pedersen, J. Nørbjerg, Strategies for organizational learning in SPI, in: L. Mathiassen, J. Pries-Heje, O. Ngwenyama (Eds.), Improving Software Organizations. From Principles to Practice, Addison-Wesley Professional, Boston, 2001, pp. 235–253.
[38] S. Vegas, N. Juristo, V. Basili, Packaging experiences for improving testing technique selection, Journal of Systems and Software 79 (2006) 1606–1618.
[39] S. Vegas, V. Basili, A characterization schema for software testing techniques, Empirical Software Engineering 10 (2005) 437–466.
[40] C. Kaner, J. Falk, H.Q. Nguyen, Testing Computer Software, second ed., John Willey & Sons, New York, 1999.
[41] J. Bach, Exploratory testing explained, in: E. van Veenendaal (Ed.), The Testing Practitioner, UTN (Uitgeverij Tutein Nolthenius) Publishers, Den Bosch, 2002, pp 209–221.
[42] A. Bertolino, E. Marchetti, A brief essay on software testing, in: R.H. Thayer, M.J. Christensen (Eds.), Software Engineering, The Development Process, Wiley Interscience-IEEE Computer Society Press, New Jersey, 2005, pp. 393–411.
[43] A. Beer, R. Ramler, The role of experience in software testing practice, in: Proceedings of the 34th Euromicro Conference Software Engineering and Advanced Applications, SEAA 2008, Italy, 2008, pp. 258–265.
[44] C. Kaner, J. Bach, B. Pettichord, Lessons Learned in Software Testing. A Context-Driven Approach, John Willey & Sons, New York, 2002.
[45] K. Karhu, O. Taipale, K. Smolander, Outsourcing and knowledge management in software testing, in: Proceedings of the 11th Conference on Evaluation and Assessment in Software Engineering, EASE 2007, United Kingdom, 2007, pp. 53–63.
[46] J. Kajihara, G. Anamiya, T. Saya, Learning from bugs, IEEE Software 10 (1993) 46–54.

[47] P. Secchi, R. Ciaschi, D. Spence, A concept for an ESA lessons learned system, in: Proceedings of Alerts and LL: An Effective Way to Prevent Failures and Problems, The Netherlands, 1999, pp. 57–61.
[48] D. Martin, J. Rooksby, M. Rouncefiel, I. Sommerville, 'Good' organisational reasons for 'bad' software testing: an ethnographic study of testing in a small software company, in: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, USA, 2007, pp. 602–611.
[49] A. Bertolino, Software testing research: achievements, challenges, dreams, in: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, USA, 2007, pp. 85–103.
[50] G.A. Di Lucca, A.R. Fasolino, Testing web-based applications: the state of the art and future trends, Information and Software Technology 48 (2006) 1172–1186.
[51] F.T. Anbari, E.G. Carayannis, R.J. Voetsch, Post-project reviews as a key project management competence, Technovation 28 (2008) 633–644.
[52] T.E. Lee, B. Kettinger, Wikis for KM in software testing: benefits and success factors, in: Proceedings of the 2nd International Research Workshop on Advances and Innovations in System Testing, USA, 2008, pp. 137–144.
[53] J. Rech, C. Bogner, V. Haas, Using wikis to tackle reuse in software projects, IEEE Software 24 (2007) 99–104.
[54] K.W. Ong, M.Y. Tang, Knowledge management approach in mobile software system testing, in: Proceedings of the IEEE International Conference on Industrial Engineering and Engineering Management, IEEE IEEM 2007, Singapore, 2007, pp. 2120–2123.
[55] J. Andrade, J. Ares, R. García, S. Rodríguez, S. Suárez, A reference model for knowledge management in software engineering, Engineering Letters 13 (2006) 159–166.
[56] D.W. Aha, R. Weber, H. Muñoz-Ávila, L.A. Breslow, K.M. Gupta, Bridging the lesson distribution gap, in: Proceedings of the 17th International Joint Conference on Artificial Intelligence, IJCAI 2001, USA, 2001, pp. 987–992.
[57] R. Weber, D.W. Aha, I. Becerra-Fernandez, Categorizing intelligent lessons learned systems, in: Proceedings of the Workshop on Intelligent Lessons Learned Systems of the Seventeenth National Conference on Artificial Intelligence, AAAI 2000, USA, 2000, pp. 63–67.
[58] W. Harrison, A software engineering lessons learned repository, in: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop, SEW'02, USA, 2002, pp. 139–143.
[59] J. Andrade, J. Ares, R. García, J. Pazos, S. Rodríguez, A. Rodríguez-Patón, A. Silva, Towards a lessons learned system for critical software, Reliability Engineering and System Safety 92 (2007) 902–913.
[60] IEEE Computer Society, IEEE Standard for Software and System Test Documentation, 2008.
[61] S.C. Wheelwright, K.B. Clark, Revolutionalizing Product Development: Quantum Leaps in Speed, Efficiency and Quality, The Free Press, New York, 1992.
[62] M. Zeleny, Multiple Criteria Decision Making, McGraw-Hill, New York, 1982.
[63] J.S. Hammond, R.L. Keeney, H. Raiffa, Smart Choices: A Practical Guide to Making Better Decisions, Harvard Business School Press, Boston, 1999.
[64] F. Houdek, H. Kempter, Quality patterns – an approach to packaging software engineering experience, in: Proceedings of the Symposium on Software Reusability, USA, 1997, pp. 81–88.
[65] B. Shneiderman, Dynamic queries for visual information seeking, IEEE Software 11 (1994) 70–77.
[66] A. Tversky, Features of similarity, Psychological Review 84 (1997) 327–352.
[67] K.D. Althoff, M. Nick, C. Tautz, Improving organizational memories through user feedback, in: Proceedings of the Learning Software Organizations Workshop, LSO'99, Germany, 1999, pp. 27–44.
[68] R.O. Weber, D.W. Aha, H. Muñoz-Ávila, L.A. Breslow, Active delivery for lessons learned systems, in: Proceedings of the 5th European Workshop on Advances in Case-Based Reasoning, EWCBR 2000, United Kingdom, 2000, pp. 322–334.
[69] M.T. Hansen, N. Nohria, T. Tierney, What's your strategy for managing knowledge?, Harvard Business Review 77 (1999) 106–116
[70] T.H. Davenport, L. Prusak, Working Knowledge: How Organizations Manage What They Know, Harvard Business School Press, Boston, 2000.
[71] B. Johnson, E. Lorenz, B. Lundvall, Why all this fuss about codified and tacit knowledge?, Industrial and Corporate Change 11 (2002) 245–263