LEVERAGING SOFTWARE REUSE WITH KNOWLEDGE
MANAGEMENT IN SOFTWARE DEVELOPMENT

DIMITRIS PANAGIOTOU* and GREGORIS MENTZAS†

School of Electrical and Computer Engineering
National Technical University of Athens
9 Iroon Polytechniou, Athens, 15780, Greece
*dpana@mail.ntua.gr
†gmentzas@mail.ntua.gr

Received 31 May 2010
Accepted 12 January 2011

Managing knowledge in software development is very important, since software development is a human and knowledge intensive activity. The main asset of a software organization consists of its intellectual capital. In this paper we propose KnowBench, a novel knowledge management system that integrates into the daily software development process and can be used for capturing knowledge and experience as soon as it is generated by providing lightweight tools based on Semantic Web technologies. This approach supports developers during the software development process to produce better quality software. The goal of KnowBench is to support the whole knowledge management process when developers design and implement software by supporting identification, acquisition, development, distribution, preservation, and use of knowledge — the building blocks of a knowledge management system.

Keywords: Software development; knowledge workbench; knowledge management in software development; KnowBench.

1. Introduction

Managing knowledge in software development is very important, since software development is a human and knowledge intensive activity. The main asset of a software organization consists of its intellectual capital. Software development is a “design type process” where every person involved has to make a large number of decisions, each of them with several possible choices [1].

Usually, developers rely on personal knowledge and experience, but as software development projects grow larger it becomes a group activity where individuals need to communicate and coordinate. Individual knowledge has to be shared and leveraged at a project and organization level, and this is exactly what knowledge management proposes.
Cruz [2] notes that knowledge management in software engineering might be exploited to capture software engineering knowledge and experience produced during the execution of projects. Even if software differs from one project to another there is still similar knowledge that can be reused in order to fasten the development process. This can assist in the avoidance of problems based on the experience gathered inside an organization.

Companies developing information systems have failed to learn effective means for problem solving to such an extent that they have learned to fail, according to an article by Lyytinen and Robey [3]. One suggested mean to overcome this problem is an increased focus on knowledge management.

Knowledge has to be collected, organized, stored, and easily retrieved when it needs to be applied. Probst [4] has well described the building blocks of KM systems: identification, acquisition, development, distribution, preservation, and use of knowledge. Many knowledge problems occur because organizations neglect one or more of these building blocks.

More recently, research has shown that semantic web technologies can provide the driving force to better manage knowledge in software development activities — ontology-based knowledge management in software development. Since 2005 the Semantic Web Enabled Software Engineering (SWESE) conference takes place every year with promising results. The Semantic Web Best Practice and Deployment Working Group (SWBPDK) in W3C included a Software Engineering Task Force (SETF) to investigate potential benefits of applying semantics in software engineering processes. As noted by SETF [5], advantages of applying Semantic Web technologies to software engineering include reusability and extensibility of data models, improvements in data quality, enhanced discovery, and automated execution of workflows.

Thus, there is a need for managing knowledge in software development. For this purpose, we designed and implemented the KnowBench (knowledge workbench) system which strives to support software developers mainly in resolving error handling and component reuse problems by providing an ontology-based knowledge management system.

In this paper we describe KnowBench which provides a semantic-based knowledge management system integrated inside a widely used software development environment namely Eclipse IDE a that supports developers during the software development process to produce better quality software. KnowBench aims at exploiting already existing knowledge and experience from past projects and give incentives to developers to articulate and efficiently reuse code or knowledge about solving problems that might have been already addressed inside the organization. Developers are not hindered to do so, as KnowBench is completely integrated into their daily development process thus avoiding extra effort to document tacit knowledge as opposed to existing approaches which realize this in a postmortem.

\[a\text{http://www.eclipse.org/}\]
fashion. The goal of KnowBench is to collect knowledge the developers gain during software development in order to tackle problems easier when reusing this knowledge and leverage successes in future projects.

Although benefits can be derived from individual tools addressing separate software development activities, KnowBench supports the whole knowledge management process. Indeed, it can be considered as integrated collections of services that facilitate software development activities by applying knowledge management paradigm in capturing, storing, disseminating, and reusing knowledge created during the software development process as well as in integrating existing sources.

The rest of this paper is organized as follows. In the next section, we describe our approach towards managing knowledge in software engineering. Afterwards, we describe KnowBench’s architecture and core components. Thereafter, we give the results of its evaluation which draws the guidelines for further optimization and improvements of the system. Afterwards, we outline related work and discuss the system’s benefits. Finally, we conclude the paper.

2. Approach

Experience gained during software development projects is essential to a software organization and a valuable asset which can be exploited for successful implementation of future projects. This knowledge has to be captured in order to be available for reuse by the rest of the organization.

Common techniques for capturing this knowledge are semi-structured interviews [6, 7] or by applying methods such as project postmortem analysis [8], post-project revisions [9], legacy sessions [10], and experience reports and reviews [11].

However, these techniques have a major drawback which hinders capturing knowledge and experience gained during the software development projects. That is, the capture process is not done until a project has finished and thus most of the time overlooked. The capture process requires that the members of project teams are available, which is not possible in most cases [12, 10, 13]. Matturro [14] notes that, if there is a time difference between the experience and its capture, there is the risk of that experience being lost finally. This risk is materialized if the owners of the experience are no longer available, either because they have been reallocated to another project or because they left the organization, taking with them the valuable knowledge and experience acquired.

We approach software development experience capturing in a different way in order to eliminate the above drawback. We propose a novel knowledge management system that integrates into the daily software development process and can be used for capturing knowledge and experience as soon as it is generated by providing lightweight tools based on Semantic Web technologies. In this way, software developers are not hindered to share their knowledge as the effort required to do this is much less in comparison to current techniques.
Software reuse is very important for software development organizations in order to increase software development productivity and quality and decrease costs. Reusing components enables an organization to develop a new system faster than implementing it from scratch. However, software reuse is often uncommon in the software development houses. Therefore, the area of software reuse is still an area of research.

As Shiva [15] notes, there are three major approaches for software reuse: component-based software reuse, metadata-based software reuse and domain engineering. Component-based development (CBD) allows the reuse of large and highly abstract components so fewer new components are needed to build the application. Metadata-based software reuse represents software artefacts and describes several aspects, i.e. how to use them and how they relate to other artefacts. Thus it is easier to locate a software artefact and determine whether it is suitable for usage based on the developer’s requirements. Domain engineering focuses on specific knowledge and artefacts as most organizations work only in a small number of domains. Identifying the common features of existing systems within a particular domain can result in higher efficiency and productivity.

We approach software reuse by combining the three above approaches: we use metadata-based modelling of reusable components and provide tools for interlinking them when these are in a specific domain. This is achieved by an ontology-based system which is supported by a semantic wiki environment. The semantic wiki allows for the easy creation of metadata and of semantic links between several components in the same domain.

Furthermore, we base this system upon the knowledge management lifecycle which is embedded in the software development environment in order to provide holistic support for managing software development knowledge. The integration of the system to the software development environment facilitates the developer to share knowledge and reuse this as there is no need to search outside the working environment.

3. KnowBench

The KnowBench system can be used to capture the knowledge and experience generated during the software development process. Although every software development project is unique in some sense, sharing similar experiences can assist developers to perform their activities in a better way. Moreover, reusing knowledge can prevent the repetition of past failures and guide the solution of recurrent problems.

The KnowBench is designed and implemented in order to assist software development work inside the Eclipse IDE exploiting the power of semantic-based technologies so as to provide a contemporary knowledge management system in software development.
KnowBench provides functionality that can be used for articulating and visualizing formal descriptions of software development related knowledge in a flexible and lightweight manner. This knowledge is then retrieved and used in a productive manner by a semantic search engine and a P2P metadata infrastructure — namely GridVine [16]. Thus, the collaboration of dispersed software developers is achieved who can benefit from each others’ knowledge about specific problems or the way to use specific source code while developing software systems.

Knowledge management requires both a shared language and a good fit with concepts that already exist in the organization. In the context of KnowBench, the KnowBench ontologies define the shared vocabulary that will be used to facilitate communication, search, storage, and representation. The ontology-background of the system enables to move from a document-oriented view of knowledge management to a content-oriented view, where knowledge artefacts are interlinked, combined, and used. Furthermore, the ontologies constitute the glue that binds the KnowBench components together. Since all knowledge handled has a formal meaning (semantics) associated, it is accessible not only to human developers, but also to automated components/tools. In KnowBench’s knowledge management approach, ontologies are used to structure the metadata store, as well as to support the main knowledge services, such as acquisition, development, distribution, preservation, and use of knowledge artefacts.

3.1. Support areas

Below we describe two typical scenarios which briefly depict the main areas that KnowBench aims to support, namely run-time error and component reuse.

3.1.1. Run-time error

Dave is working on a Java software development project in a KnowBench-powered Eclipse environment. The project uses the external component “Lucene” in Version 1.9. He discovers a new release of “Lucene” (Version 2.0) and replaces the old version in the Java Build Path setting of the project. He then tries to run the program and gets a run-time exception. He decides to search for workarounds using KnowBench which in turn suggests (after searching the global metadata store) modifying a call to the Lucene API, which was responsible for a similar error for Peer-Developer Peter. Dave opens the solution in DevWiki [17] and reviews it. He applies the solution and runs the program again. The error does not re-occur. Finally, Dave documents the change applied in his code using DevWiki and KnowBench stores it into the repository thus growing the collective knowledge inside the P2P network.

bhttp://lucene.apache.org/
3.1.2. Component reuse

Dave is developing a groupware application. He downloads a recent version of the OpenXChange software and adds it to the Java Build Path setting of the project. Then he imports the package in a source file of the groupware application he is currently editing. He uses KnowBench to get a list of other projects which include OpenXChange. KnowBench finds that Peter has some code and documents about OpenXChange. KnowBench displays the resources to Dave. He takes a look at the information provided by Peter and recognizes some interesting documentation on the OpenXChange API and some example code from Peter’s project. Then, he uses the API inside his source code by reusing Peter’s code. Furthermore, Dave decides to contribute knowledge about the use of the API in his source code and the particular domain he is using it. Thus, he uses the semi-automatic annotation mechanism in order to semantically annotate the source code and he shares it in the P2P network. Thus, the source code can be found easier since it is annotated semantically and can be retrieved in the network by other developers by using semantic or structured search.

3.2. Ontologies

We have deployed software development ontologies [18] inside the KnowBench system in order to describe and capture knowledge related to software artefacts. The system architecture allows for the extension or the use of different software development ontologies. The set of the deployed ontologies includes three main ontologies (knowledge artefact, problem/solution and annotation) which interact among each other and are all under a common parent ontology (KnowBench ontology) which binds them. These ontologies have been written in the Web Ontology Language (OWL).

Knowledge Artefact Ontology

The knowledge artefact ontology describes different types of knowledge artefacts such as the structure of the project source code, reusable components, software documentation, knowledge already existing in some tools, etc. These knowledge artefacts typically contain knowledge that is rich in both structural and semantic information. Providing a uniform ontological representation for various knowledge artefacts enables us to utilize semantic information conveyed by these artefacts and to establish their traceability links at the semantic level.

A knowledge artefact is an object that conveys or holds usable representations of different types of software knowledge. It is modeled through the top level class Knowledge Artefact.

\(^{c}\text{http://www.open-xchange.com/}\)
\(^{d}\text{http://www.w3.org/TR/owl-features/}\)
There are some common properties for all types of artefacts such as:

- **hasID** — which identifies an artefact in a unique manner. Recommended best practice is to identify the artefacts by means of a string conforming to a formal identification system;
- **hasTitle** — which defines a title of an artefact (e.g., class name or document title);
- **hasDescription** — which describes what an artefact is about;
- **createdDate** — which models date of creation of an artefact;
- **usefulness** — which models how frequently an artefact is used as a useful solution.

All previously mentioned properties are data properties and are modelled in the Knowledge Artefact ontology. However, many additional properties can be defined for a knowledge artefact that link it to the entities defined in the other KnowBench ontologies (e.g., the Annotation ontology).

Knowledge artefacts other than source code, such as documentation or entries in issue tracking systems, contain rich semantic information that is not used by existing tools for knowledge management in software engineering. Introducing an ontological representation for software documentation, issue-tracking systems, etc. enables us to “understand” parts of the semantics conveyed by these artefacts and to establish additional traceability links among these artefacts and the software artefacts. As shown in Fig. 1, we consider different specializations of the KnowledgeArtefact class, namely SoftwareArtefact, KnowledgeDocument and Tool-embeddedKnowledge. The reason for that is that knowledge needed for software development tasks is contained in not only software code or components but also in some documents (cf. knowledge document in Fig. 1) used during software development process as well as in tools (cf. tool-embedded knowledge in Fig. 1) that support the software development process. Knowledge documents represent informal knowledge produced by a human usually as a text document. On the other hand, tool-embedded knowledge is knowledge collected by a tool in a semi-structured form. This structure of the
KnowledgeArtefact class enables very easy extension of the model by preserving all functionality. For example, if someone finds another software-development supporting tool which can be used as a useful knowledge source, its knowledge should be modeled as an additional subclass of the Tool-embeddedKnowledge class.

The previously mentioned subclasses of the class KnowledgeArtefact are also defined in the Knowledge Artefact ontology and they inherit all properties defined for the class KnowledgeArtefact. The SoftwareArtefact class is further split into SourceFile and Component classes. There are also two specializations of the Tool-embeddedKnowledge class, namely the Issue class and the ChangeSet class.

Some additional properties are defined between the subclasses of the class KnowledgeArtefact, namely:

- hasRelatedSoftwareArtefact — property defined between classes KnowledgeDocument and Tool-embeddedKnowledge and the class SoftwareArtefact that models that there is a related software artefact for either a knowledge document or a tool-embedded knowledge entry;
- hasRelatedKnowledgeDocument — inverse property for the hasRelatedSoftwareArtefact property defined between the class SoftwareArtefact and class KnowledgeDocument (and consequently all its specializations) that models that there is a related knowledge document for a software artefact;
- hasRelatedToolEmbeddedKnowledge — inverse property for the hasRelatedSoftwareArtefact property defined between the class SoftwareArtefact and class Tool-embeddedKnowledge that models that there is a related tool-embedded knowledge item for a software artefact;
- usesComponent — property defined between the class SourceFile and class Component that models that a component is used in a source code;
- isUsedInCode — inverse property for the usesComponent property;
- isSourceFor — property defined between the class SourceFile and class Component that models that a source code file is source for a component.

Problem/solution ontology

The problem/solution ontology models the problems occurring during the software development process as well as how these problems can be resolved. This ontology is essential to the KnowBench system, as source code can be documented when a certain problem is met and the respective solution to it is described.

Problem

A problem is an obstacle which makes it difficult to achieve a desired goal, objective or purpose. It is modelled through the class Problem and its properties:

- hasID — a problem is identified by its identifier;
- hasTitle — a summary of a problem;
- hasDescription — a textual description of a problem;
- hasSeverity — it models how severe the problem is;
hasSolution — every problem asks for a solution. This is the inverse property for the property isRelatedToProblem defined for the class Solution;

hasSimilarProblem — it models similar problems.

There are different types of problems that are modelled as subclasses of the class Problem: Error, InformationNeedProblem, Issue.

Each of these classes has its own properties. For example, error codes are enumerated messages that correspond to errors. They are typically used to identify faulty hardware, software, or incorrect user input. Error codes are typically identified by number, each indicating a specific error condition.

Solution
A solution is a statement that solves a problem or explains how to solve the problem. It is represented through the solution class and its properties:

- hasID — an unambiguous reference to a solution;
- isRelatedToProblem — a solution is defined as the means to solve a problem;
- hasCreationDate — date of creation of a solution;
- hasDescription — a textual description of a solution;
- hasPrerequisite — a solution requires a level of expertise in order to be used;
- hasComment — a solution can be commented by the people that used it;
- suggestsKnowledgeArtefact — a solution recommends using a knowledge artefact in order to resolve a problem the solution is defined for;
- isAbout — a solution can be annotated with domain entities;
- isRelatedToSEEEntity — a solution can be annotated with the general knowledge about software engineering domain.

Annotation ontology
The annotation ontology describes general software development terminology as well as domain specific knowledge. This ontology provides a unified vocabulary that ensures unambiguous communication within a heterogeneous community. This vocabulary can be used for the annotation of the artefacts. We distinguish two different types of annotations: (a) domain annotation — software providers in different domains should classify their software using a common vocabulary for each domain. Common vocabulary is important because it makes both the users and providers to talk to each other in the same language and (b) software engineering annotation — general knowledge about the software domain including software development methodologies, design patterns, programming languages, etc.

By supporting different types of annotation it is possible to consider information about several different aspects of the knowledge artefacts.

We note here that the vocabulary is used for the annotation of the knowledge artefacts, e.g. code, knowledge documents, etc. as well as solutions of problems. These annotations will be used to establish the links between these different knowledge artefacts.
Domain ontology
Based on the description of the ontologies we gave, it can be concluded that, these ontologies are general enough to be applied and model software developed for any domain. On the other hand, the entities from the domain ontology are domain specific and cannot be easily generalized to other software projects. Consequently, the domain ontology has to be developed for each software project separately.

The content of knowledge artefacts themselves is essential toward determining particular topical interests and understanding the relationships between knowledge artefacts. The domain ontology should consist of concepts and relations modeling meaning of content of a knowledge artefact. This may include already existing categorisation of services (e.g. content management) as well as typical community terminology (e.g. types of existing documents such as invoice, receipt, etc.).

Construction of this ontology can be a very time consuming and error-prone task for the domain experts. In order to get a feeling on what are the relevant topics for knowledge artefacts, what are the relations between them and at the end to assign each knowledge artefact to some certain domain entities, the domain expert has to read all the knowledge artefacts and to understand them. This can be overcome by building a special tool based on NLP methods (see Sec. 3.4.4) applied on the content of knowledge artefacts which helps the domain expert by suggesting the topics, showing the importance of topics so far, and putting them into the right place into the ontology (including the relationships with existing entities).

Software Engineering Ontology
The Software Engineering ontology consists of a large body of concepts that are expected to be discovered in software documents or in the comments in code. These concepts are based on various programming domains, including programming languages, algorithms, data structures, and design decisions such as design patterns and software architectures. This ontology is used for the annotation of different knowledge artefacts (e.g. code). It should be noted that the software engineering ontology is used for the annotation of knowledge artefacts as well as solutions.

Dependencies between KnowBench ontologies
The KnowBench ontologies are interlinked: For example, software artefacts modelled in the Knowledge Artefact ontology cause problems that are modelled in the Problem-Solution ontology. Each problem occurs in a context, i.e. it refers to a knowledge artefact. The resolution of a problem has to be applied on the knowledge artefacts.

Additionally, each knowledge artefact can be annotated with the entities from the domain ontology in order to specify what the software artefact is about as well as with the entities for the general software engineering ontology in order to clarify the methodology/approach/design patterns that have been used. Code entities as well as solutions of problems are also annotated with the domain and software engineering entities. Therefore, the properties isAbout and isRelatedToSEEntity are modeled. The range of these properties are classes DomainEntity and
Software Engineering Artefacts that are defined in the Domain ontology and Software Engineering ontology respectively.

A further crosslink among the ontologies is the property hasComment which allows to attach descriptions to knowledge artefacts. The domain is thus the class Comment, with the following properties:

- hasCommentDescription — a textual description of a comment;
- hasAuthor — a comment is defined by a person;
- hasUsefulnessLevel — a comment has usefulness level in range 1 to 10.

Finally, an issue can be related to a problem and knowledge captured in issue tracking systems or content management systems may document solutions for some problems. These dependencies are shown in Fig. 2.

Figure 3 depicts software engineering and domain entities which annotate several artefacts which are contained in physical resources (source code). The annotated artefacts can be used in order to find source code that is pertinent to specific concepts.

More specifically, in Fig. 3 the MetadataStore source code file is written for a bank application and relates to a software engineering entity “DataStore”. The MetadataStore.allTBoxModelsAreCreated method is written in Java 6. The MetadataStore.initKBModel method is a repository model and in general contains code

![Fig. 2. Relations between KnowBench ontologies.](image)
for a repository. The concept “Resource” indicates where source code is stored as a physical resource (path of the file).

This knowledge base can be used by a developer in order to locate (by performing a query) source code that is developed for a bank application or is used to build a repository model. Furthermore, the developer can locate methods that are written in Java6 or that are written for implementing a repository or a data store. Finally, the physical resources of this source code can be found in order to retrieve the actual file containing it.

### 3.3. Architecture

Figure 4 depicts the overall KnowBench architecture. The core components delivering KnowBench’s functionality are: semantic search, global metadata store, software development semantic wiki (DevWiki) and (manual/semi-automatic) semantic annotation of source code. Below, we give more details about their actual interaction.

The above components are based on these APIs: semantic annotation API, shallow NLP and IE (information extraction) API (wrapping up and customizing Text2Onto [19]), the semantic search API and the global metadata store.
The semantic annotation API is used for both manual and semi-automatic semantic annotation of source code and interacts with the metadata store to perform operations on the knowledge base. It is responsible for automated generation of annotation related metadata, when it is provided with an annotation concept and a respective source code element. Manual annotation is triggered by the software developer via his/her KnowBench using its source code editor to annotate a specific source code fragment.

On the other hand, semi-automatic annotation uses a source code corpora as input in order to extract new annotation concepts and if feasible their classification inside the annotation ontology hierarchy tree. The output of the semi-automatic annotation is then passed to the semantic annotation API to add the actual annotation metadata inside the knowledge base via the global metadata store in the same manner as in the manual annotation process. The shallow natural language processing (NLP) and information extraction (IE) API is used to achieve the purpose of annotating source code corpora in a semi-automatic supervised by the software developer procedure.

DevWiki (semantic wiki used for documenting and retrieving software development knowledge) communicates and operates directly on the knowledge base (via the global metadata store API). DevWiki supports operations such as creating, updating/modifying and removing knowledge base items. Furthermore, DevWiki
uses services from the global metadata store for browsing the ontologies and the knowledge base. Finally, the P2P MDS is used to publish local knowledge.

DevWiki uses the semantic search engine in order to execute (semantic) queries, displaying a list of (semantic) search results, refining queries, etc. by using support from the global metadata store. It uses this semantic search engine to retrieve knowledge stored either in the local or in the P2P repository. Since the developers will be able to search for knowledge, semantic search services such as query (keyword), query (structured query), getQueryRefinements and getQueryRelaxations are called by DevWiki.

The global metadata store API provides an abstract layer on top of the local and P2P repositories through customized APIs (Jena [20] and GridVine-based APIs [16]).

3.4. Core components

Below we outline the core components which constitute the KnowBench system.

3.4.1. Semantic search

KnowBench supports advanced methods for knowledge search through a semantic search engine [21] by taking into account three different types of search, namely keyword, structured and semantic search.

Keyword search component realizes a Lucene-based traditional paradigm of search, with an advanced functionality of indexing different sources of relevant information for software developers, such as SVN repositories or JIRA issues.

While keyword-based search represents the standard model for search interfaces, structured search allows more precise queries which in turn yield more precise results. “Structured search” is the search defined through ontology entities. In the structured search, users do not need to have any background knowledge on the ontology as they can directly define through the KnowBench GUI the specific classes and the properties that they are looking for. For example, one can choose to search only for the problems which have been identified by the user John.

Structured search promises to provide more accurate results than present-day keyword search.

Semantic search in contrast is the search which automatically translates keyword queries into formal logic queries so that developers can use familiar keywords to perform structured search without having to navigate in advance through the ontology tree. The translation consists of three major steps: keyword mapping, graph exploration and query ranking. The generic graph-based approach is followed to explore the connections between ontology nodes that correspond to keywords in the query. In this way, all interpretations that can be derived from the underlying RDF graph can be computed.
The ontologies are used in the process of answering queries. Thereby, implicit knowledge (e.g. stemming from information about hierarchical relations or inferred by rules) is taken into account for matching results. However, this does not naturally provide any indication about different degrees of relevancy of a match — i.e. a knowledge artefact is either a result or it is not. In order to rank the results of a query, context-based information is used to capture the meaning intended by the users and thus, the search space is further constrained.

Moreover, to deal with information overload (too many results) as well as no answers to problems, the engine creates suggestions for query reformulation that could reduce the long time it takes to find information. Several possibilities are proposed, including creation of semantic index from semi-structured sources, according to the ontologies. Additionally, semantic relations between different artefacts are drawn based on implicit mutual dependencies.

3.4.2. Global metadata store

The global metadata store consists of two components (APIs) — the LocalMDS and P2PMDS and two managers — KeyStore Manager and Policy Manager. It provides an abstract layer for handling these and its purpose is to manage knowledge stored either locally or in the P2P network.

The P2PMDS is based on the GridVine/P-Grid system [22] and is customized for KnowBench in order to realize an access control aware P2P metadata store using the services of KeyStore and Policy managers.

The KeyStore Manager is responsible for providing up-to-date status of various P2P groups and the corresponding credentials. The P-Grid secured communication infrastructure needs a set of Certificate Authority’s credentials to process various certificates passed to peers in regular communication with other peers.

The Policy Manager provides services to configure and update the access control policy. The developer defines what is shared to whom through the P2P repository by assigning certain policies to different types of knowledge using criteria and the desired developer groups where this knowledge is to be shared. Examples of these criteria could be: “problems that are written by John”, “solutions that are solving problem X”, etc. The criteria are built by constraining properties of ontology concepts to certain values, thus allowing the possibility to create complex rules about what stays inside the local repository or is shared to other peers.

The LocalMDS is built on top of the well known open-source Jena framework in order to provide a handy API for handling the local knowledge base. The main reason for abstracting Jena to KnowBench and building a new API on top of it is that we needed bulk operation (executing multiple knowledge base transactions at once) and easier/more powerful handling as well as aggregated operations that are not provided by Jena (Jena methods are of high granularity for the purposes of KnowBench — the encapsulation of several Jena methods was necessary in order to provide a higher level of functionality to KnowBench API calls).
3.4.3. **Software development semantic wiki (DevWiki)**

KnowBench utilizes the DevWiki system [17] in order to assist software developers in the articulation and navigation of software development related knowledge. DevWiki uses a lightweight and flexible editor with auto-completion and popup support. Browsing through knowledge is done like surfing through a conventional wiki using the semantic links between different knowledge artefacts. This browser is available inside the Eclipse IDE so that the software developer does not have to switch to another external browser.

DevWiki provides assistance to software developers by shortening their total effort in terms of time. In order to achieve this goal, we approach software development documentation and related problem solving by combining lightweight yet powerful wiki technologies with Semantic Web standards.

3.4.4. **Manual/semi-automatic semantic annotation of source code**

An important aspect of the KnowBench is the ability to annotate semantically source code. We have extended the standard Eclipse JDT editor to add this possibility. The software developer is able to annotate source code with semantic annotation tags that are available or define new tags and extend the used annotation ontology.

The manual semantic annotation of source code provides granularity regarding the respective source code fragment to be annotated. This granularity level is restricted by the underlying Eclipse platform itself as the IJavaElement interface is exploited to map between source code fragments and metadata. This limits the selected source code fragments to be annotated in the nearest Java elements that surround the source code fragment at hand (e.g. package/class/method/variable etc.). Additionally, by using IJavaElement the semantic annotation mechanism is capable of managing annotations of several objects, such as Java packages, classes, methods, etc.

When a manual semantic annotation (as well as semi-automatic — see below) is added auto-generated metadata are added to the knowledge base: (a) code entity instance — the type of the code entity, (b) source file instance — the source file that contains the code entity, (c) file instance — the physical file that a source file is located and (d) annotation class — in the case that the software developer chooses to extend the annotation ontology and add a new concept.

We note here that it is possible to annotate the same code entity with multiple annotations.

On the other hand, the shallow natural language processing (NLP) and information extraction (IE) API is used to semi-automatically annotate source code corpora. The API is built on top of the Text2Onto API and provides customizations suitable for source code (e.g. java keywords are ignored such as import, package for, while, etc.).

This process is supervised by the user and results in new concepts and sometimes in their classification inside the annotation ontology hierarchy tree. These are then
used to annotate semantically the source code. Thus, the time consuming task of manually annotating source code can be significantly decreased.

In order to achieve this goal in KnowBench we exploited ontology learning [23] and information extraction techniques [24]. Ontology learning is needed in order to extract ontology concepts from source code corpora that can be used for annotating it. On the other hand, information extraction is needed in order to instantiate the annotation ontology with individuals (like in manual semantic annotation). A framework exploiting both technologies is Text2Onto [19] which we adopted and extended in order to implement the desired functionality.

The architecture of Text2Onto is based on the Probabilistic Ontology Model (POM) which stores the results of the different ontology learning algorithms. A POM as used by Text2Onto is a collection of instantiated modeling primitives which are independent of a concrete ontology representation language.

KnowBench takes advantage of two modeling primitives, i.e. concepts and concept inheritance. In the respective modelling primitives KnowBench employs different Text2Onto algorithms in order to calculate the probability for an instantiated modeling primitive. In particular the following measures are calculated:

Concepts are identified using the following algorithms: Relative Term Frequency (RTF), Term Frequency Inverted Document Frequency (TFIDF), Entropy and the C-value/NC-value method in Ref. 25. For each term, the values of these measures are normalized into the interval $\frac{0}{C138}$, after taking the average probability of the algorithms output combination which is used as the corresponding probability in the POM.

Regarding concept inheritance, KnowBench combines Text2Onto algorithms such as matching Hearst patterns [26] in the corpus and applying linguistic heuristics mentioned in Ref. 27.

3.5. Implementation details

The above components communicate with each other and with Eclipse via OSGi\textsuperscript{e} (in Eclipse world Equinox\textsuperscript{f}). They are all implemented using the Eclipse PDE\textsuperscript{g} and built as autonomous bundles (plug-ins) in order to ease the process of extending the system by adding new components. This is achieved through a thin core KnowBench plug-in manager which instruments the interactions between the various bundles (which have to comply with some minimum interfaces defined in this manager).

Furthermore, we based KnowBench on Eclipse plug-in/OSGi mechanisms in order to enable smooth integration of the several components. The main reason for that is that OSGi allows either a direct communication of components or an indirect

\textsuperscript{e}http://www.eclipse.org/osgi/
\textsuperscript{f}http://www.eclipse.org/eqinox/
\textsuperscript{g}http://www.eclipse.org/pde/
communication. The latter is based on the OSGi event mechanisms and provides mechanisms for synchronous as well as asynchronous communication.

Asynchronous communication using the OSGi event service is supported. The main reason for this is to maximize the openness of communication, since several plug-ins might be interested in communication, which is not anticipated at design time.

On the other hand, when in the design time is already known that the interaction between the components is required, the direct communication is enabled due to performance reasons. For example, all communication with the local metadata store has been done by direct method calls.

4. Knowledge Management Support with KnowBench

We have conducted detailed evaluation of KnowBench in small groups of software developers (in a total of 16 developers) in the following organizations: (1) a multinational Brussels-based company specializing in the field of Information and Communication Technology (ICT) services (Intrasoft International S.A. — 4 developers), (2) a leading Hungarian association dealing with open source software at corporate level (Linux Industrial Association — 4 developers), (3) an Italian company which operates in the Information Technology market, focusing on business applications (TXT e-Solutions — 4 developers) and (4) the corporate research laboratory of the Thales group — a global electronics company serving Aerospace, Defence, and Information Technology markets worldwide (Thales Research & Technology — 4 developers). See Ref. 28 for a comprehensive report on the KnowBench evaluation.

4.1. Evaluation method — GQM

The evaluation of the KnowBench system was based entirely on the use of the Goal-Question-Metric (GQM) method [29]. According to this method, the efficacy of the KnowBench system is derived by comparison to a number of stated goals. GQM provides a process framework for evaluating software systems and defines a measurement model on three levels:

- Conceptual level (goal): A goal is defined for an object for a variety of reasons, with respect to various models of quality, from various points of view and relative to a particular environment.
- Operational level (question): A set of questions is used to define models of the object of study and then focuses on that object to characterize the assessment or achievement of a specific goal.
- Quantitative level (metric): A set of metrics, based on the models, is associated with every question in order to answer it in a measurable way.
The GQM process consists of four phases: (a) the project plan, (b) definition of goals, questions and metrics, (c) data collection and (d) data analysis under consideration of goals and questions.

The areas of evaluation (goals) were the following two:

- The knowledge empowerments in the software development practice of the organization, in order to assess the benefits of using KnowBench for the support of pertinent knowledge processes and
- The KnowBench system itself, since its successful application depends on its endorsement by the members of the organization.

The GQM method is based on the idea of goal attainment. This is determined through the summative account of the collected replies for the set of chosen questions. Below we describe the metrics adopted for determining the positive and the negative contributions towards the attainment of the goals as well as the method for aggregating results within each goal context.

The questionnaire adopted for the elicitation of the respondents’ feedback was implemented on the basis of three types of questions:

(a) Questions of 1..5 Likert scale (negative to positive orientation),
(b) Questions of “Yes” or “No”, and
(c) Questions of open end.

The first two types of questions were subject of our quantitative analysis, while the latter group was used to formulate the ideas for the improvement of KnowBench. For the Likert scale questions, we consider answers rated with 1 and 2 as negatively while those rated with 3–5 as positively contributing to the attainment of a goal. The measurement of the goal is derived after the aggregation of the average of all positively and negatively-contributing answers.

KnowBench is proposing a holistic treatment of Knowledge Management for software engineering that is based on a lifecycle perspective. As already mentioned, the basic building blocks of the KM lifecycle that KnowBench realizes refer to the activities of identification, acquisition, development, sharing, usage, and preservation of knowledge. In order to conduct the evaluation, we used questionnaires which aim to measure the effectiveness of KnowBench in supporting the KM lifecycle.

Evaluation goals for KnowBench itself are primarily formulated based on software quality attributes. We used a typical classification to set the goals — the FURPS approach [30] considers functional as well as non-functional requirements (usability, reliability, performance, supportability).

### 4.2. Evaluation according to the KM building blocks

We grouped the evaluation results according to the KM building blocks depicted in Sec. 1. These results will give us the right guidelines in order to further optimize the
system and improve the support in various aspects that KnowBench provides towards managing software development knowledge.

Please note that the knowledge identification block is not described below as the KnowBench ontologies serve this purpose.

Questionnaires were used in order to measure KnowBench performance in the various aspects of its evaluation. An example questionnaire that was used for measuring the performance in knowledge development (Sec. 4.2.2) is given in Table 1 below:

### 4.2.1. Knowledge acquisition

According to the respondents, KnowBench achieves a good score (73% are positive) for its support in acquiring existing knowledge.

Even though it was easy for 92% of respondents to identify which information should be provided in order to crawl a repository, 67% of the respondents were satisfied with the support for consistency checking of the provided input.

<table>
<thead>
<tr>
<th>Question</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>How understandable is the meaning of knowledge items?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How many knowledge items did you create?</td>
<td>0, &lt;10, &lt;30, &lt;50, a lot</td>
</tr>
<tr>
<td>Does the creation of knowledge items disturb you in your daily work?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How easily can you understand how to create a new knowledge item using DevWiki?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How easily can you understand what is required as input data for creating a knowledge item?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How much effort is required on defining a new knowledge item using DevWiki?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>What is the time taken to create a knowledge item using DevWiki?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How much guidance is provided in creating knowledge items using DevWiki?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>Does KnowBench allow for timely access to information?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How clear is the meaning of annotation?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>Is the granularity level of source code that can be annotated sufficient?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How understandable is the manual approach for annotation creation?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How much effort is required for manual annotation creation?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How understandable is the semi-automatic approach for annotation creation?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How much effort is required for semi-automatic creation of annotations?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How much are you satisfied with the suggestions for semi-automatic creation of annotations?</td>
<td>Likert (1..5)</td>
</tr>
<tr>
<td>How many annotations proposed by the semi-automatic approach have you chosen?</td>
<td>0, &lt;3, &lt;10, &lt;20, a lot</td>
</tr>
<tr>
<td>How many annotations did you create?</td>
<td>0, &lt;10, &lt;30, &lt;50, a lot</td>
</tr>
<tr>
<td>Does KnowBench provide friendly and easy-to-use GUI for creating annotations?</td>
<td>Yes, No</td>
</tr>
<tr>
<td>What would be your suggestion for the improvement of knowledge development in KnowBench?</td>
<td>Open ended question</td>
</tr>
<tr>
<td>What would be your suggestion for the improvement of annotation creation in KnowBench?</td>
<td>Open ended question</td>
</tr>
</tbody>
</table>
Regarding the supported types of knowledge sources, 85% of the respondents were satisfied with the support; 23% found that additional types of knowledge sources relevant for coding should be supported. However, there is no respondent who used either all available type of knowledge sources or suggested that the crawler has missing any.

In general, the respondents encountered some minor problems during the evaluation phase of the system. 23% of the respondents mentioned that the system presented occasional failures of not sending notification when the crawling is completed.

As far as the system's response time, although 64% of the respondents found it quite fast — some further optimization of the system would be useful.

4.2.2. Knowledge development

All respondents found the knowledge development support in KnowBench clear and easy to follow and agree that the system provides support at an adequate level (76%).

The meaning of knowledge items is understandable for 86% of the respondents. DevWiki meets the expectations of respondents. Creation of knowledge items was understandable and did not disturb developers during daily work.

Regarding annotations, their meaning and purpose were clear for 86% of the respondents. 79% of the respondents found the granularity level of source code that can be annotated sufficient.

Respondents agreed that manual creation of annotations required more effort in comparison with automatic annotations. This is expected as they had to analyze the source code and to assign tags to it. However, 92% of the respondents found that KnowBench provides friendly and easy-to-use forms for creating annotations and only 36% of them considered the manual annotation as effortful activity.

On the other hand, in semi-automatic annotation, 42% of proposed annotations were chosen, thus 82% of the respondents were satisfied with the suggestions.

4.2.3. Knowledge sharing

Knowledge sharing in KnowBench meets the expectation of 69% of respondents.

Even though all aspects of knowledge sharing in KnowBench are above the threshold, only the level of details to be specified in order to share knowledge and the usefulness of shared information received high marks (greater than 90%). Other aspects received an average score. For example, the meaning of policy rules for knowledge sharing is understandable for the 71% of respondents.

4.2.4. Knowledge usage

The search functionality received an average score (57%). The respondents used this functionality very often and had totally different opinions about it. All kinds of answers were given to all questions related to search functionality.
The respondents were satisfied with the quantity and quality of search results. As far as quantity of search results is concerned, 91% of the respondents found the number of results optimal. For 73% of the respondents the list of results did not contain any irrelevant result. As regards the quality of search results, 62% of the respondents confirmed that the search results satisfy their information needs more than average.

4.2.5. Knowledge preservation

The lifecycle of the knowledge items, i.e. creation, update, deletion in KnowBench seems to be supported well (expectation of 67% of the respondents). Modification of knowledge is not a time-consuming function for 75% of the respondents and can be done very easily by 73% of the developers.

54% of the respondents found that the knowledge in the P2P network is not always up-to-date. Once published, knowledge cannot be revoked/updated whenever “source knowledge” is deleted/modified. The respondents also noticed delays in the knowledge sharing process. This is normal behavior — when knowledge is shared, time is needed for having P2P knowledge to another peer, especially when there is high traffic on the network.

4.3. Evaluation based on the FURPS approach

In this section we outline the evaluation results of the KnowBench system itself. The evaluation was conducted using the FURPS approach. The goals are slightly modified and include usability, reliability, scalability and availability, security and deployment.

4.3.1. Usability

Overall, the usability of KnowBench has been well appreciated by 79% of the respondents. More than 80% of respondents agree that the user knows where he is at all times, how he got there, and how to get back to the point from which he started. They also found that the system is simple and that the GUI behaves as the users expect. The navigation through the system’s functions was found to be very good.

The appearance of the system pleased 73% of the respondents.

Although the overall interaction with the system is quite comprehensive, some improvements are recommended, since 31% of respondents occasionally perceived the interactions as awkward or redundant.

Even though KnowBench achieves a good score for its usability, some additional work should be done so that it becomes more user-friendly and intuitive. Indeed, the 67% of respondents found that easiness and intuitiveness of the GUI could be further improved.
4.3.2. **Reliability**

KnowBench meets the expectations of the users as regards its reliability. More specifically, 67% of respondents did not encounter problems during the evaluation of KnowBench.

4.3.3. **Scalability and availability**

The scalability and availability of KnowBench meets the expectation of 65% of the respondents. The respondents agree that KnowBench behaves normally when the size of the knowledge base is increased.

As far as the system’s response time, 40% of the respondents found it quite slow; especially the P2P network should be faster. Thus, some improvements in the system’s response time are needed in order for KnowBench to become more agile. 29% of the respondents answered that the system requires a lot of memory. The respondents also mentioned the need for further optimisation of the system.

On the other hand, KnowBench does not occupy a significant size of hard disk space. It also does not introduce an increase raise in network traffic.

4.3.4. **Security**

The respondents agree (88%) that KnowBench provides high level of security through restricted access to shared information to users with appropriate rights and provision of adequate protection of shared knowledge (e.g. encryption).

4.3.5. **Deployment**

The deployment of KnowBench meets the expectation of 80% of the respondents. The respondents found that the installation procedure is easy to follow. They also agreed on the friendliness and ease of use of the uninstall procedure. Repeat setup operation in the case that an error occurred could be further simplified in order to satisfy more than 73% of respondents.

4.4. **Evaluation results synopsis**

KnowBench has been well appreciated by 62% of the respondents. All respondents found that its functionalities seem to integrate well with Eclipse without overloading developers with information or encumbering them with time-consuming functions. 77% of respondents agreed that KnowBench is capable of improving their overall working experience. They found that the KnowBench concept is good. However at its current prototypical implementation phase, its potential could not be fully exploited. Figure 5 summarizes the percentages of positive and negative responses of the developers concerning the knowledge management lifecycle support. Likewise Fig. 6 summarizes the responses concerning the various aspects of the FURPS evaluation.
There are several approaches for managing knowledge in software engineering. Holz [31] presents a detailed life-cycle model for POKM (Process-Oriented Knowledge Management) that is specific to software development processes. This life-cycle model for Software Engineering Process-Oriented KM (SEPOKM) is integrated into the life-cycle model performed by the organisation’s Process Group and becomes an essential part of a continuous organizational learning process.

Cruz [2] presents an infrastructure to deal with knowledge management in software engineering environments (SEE). This infrastructure is applied to manage product software quality knowledge in ODE, an ontology-based SEE.

Additionally, research has been conducted in experience management. Feldmann and Althoff [32] have defined a “learning software organization” as an organization that has to “create a culture that promotes continuous learning and fosters the
exchange of experience”. Dybå [33] places more emphasis on action in his definition: “A software organization that promotes improved actions through better knowledge and understanding”.

In software engineering, reusing life cycle experience, processes and products for software development is often referred to as having an “Experience Factory” [34]. In this framework, experience is collected from software development projects, and is packaged and stored in an experience base.

Borges [35] stores and shares the experience obtained in software process definition. In order to achieve this, they built ProKnowHow, a tool that supports the standard software process tailoring procedure for each project, providing KM support.

Furthermore, research has been conducted on aspects of knowledge management that are related to software engineering [36−39]. Bjørnson and Dingsøyr [40] present a systematic review of findings and research methods used in knowledge management in software engineering.

On the other hand, there are emerging efforts in research towards exploiting Semantic Web technologies in software engineering in order to manage software development knowledge [41−43]. For example, SemIDE [44] is a framework which is based on meta-models in order to describe semantically software engineering processes. OSEE [45] is a tool which aids the construction of a knowledge base which captures software related artefacts.

Ankolekar et al. [46] provide an ontology to model software, developers, and bugs. Happel et al. [47] present an approach addressing the component reuse issue of software development by storing descriptions of components in a Semantic Web repository, which can then be queried for existing components.

There have also been many previous attempts to design a Relational database schema to store RDF/Owl data [48, 49]. ScRiDA [50] combines object-oriented programming and relational databases, with semantic models. Another approach [51] relies on manual annotation by a user to explicitly associate instance elements in software artefacts with concepts in the ontology. However, this technique implies a considerable amount of effort since there is a significant number of software engineering elements.

On the other hand there are also attempts to use semantic wikis in software engineering. The RISE (Reuse in Software Engineering) project [52] introduces the Riki Wiki combined with the Wikitology paradigm [53, 54]. With Wikitology, it is possible to derive semi-automatically an ontology needed for information retrieval from the pages in the Riki itself. Such a Wikitology automatically updates “itself”, i.e. it reflects the changes of contained knowledge, changed views of the users accounts, new projects, customers, rules, trends, and ideas inside the Wiki. By considering the Riki as the ontology, the ontology will be always collectively edited up-to-date and will automatically reflect all changes.

Hyena [55] aims at providing a platform that makes RDF’s advantages available to software engineering without diverging too far from the concepts and tools that
are familiar in that domain. It provides editing, GUI and publishing services for implementing editors of RDF-encoded custom models and it is implemented as an Eclipse plug-in. It includes some built-in ontologies for many basic software engineering tasks. These ontologies have themselves become part of the platform, because they provide services that will be useful for other ontologies.

Our approach differs from related work as we focus mainly in problem solving when coding and systematic software reuse. KnowBench is integrated into the daily software development process and can efficiently manage experience and knowledge as this is captured as soon as it is generated with the support of lightweight tools based on Semantic Web technologies. In this way, software developers are given incentives to share their experiences as the effort required to do this is less in comparison to the state-of-the-art.

6. Discussion
KnowBench models software artefacts and exploits the resulting knowledge base in order to provide knowledge for solving problems by offering a semantic search mechanism and taking into account previous local experience or experience by other colleagues. The experience gained by past projects is captured in parallel as the projects grow and the effort required is minimized. This approach assists developers to capitalize collectively tacit knowledge, thus enabling an organization to leverage and take advantage of this knowledge (mostly not captured and neglected in traditional systems), in order to produce better quality software by developing software based on past experiences, lessons learned and guidelines. When a significant amount of this knowledge is stored in KnowBench, developers can benefit from each others’ experience in order to avoid delays and adhere to standard approaches when developing source code.

The goal of KnowBench is to provide a knowledge management system that supports software developers in the coding phase of the software development process. KnowBench makes a contribution to the solution of concrete problems in coding, namely error handling and component reuse. It enables distributed teams to become more effective by learning from each others’ experiences, e.g. on reusing specific components.

KnowBench offers an environment to facilitate knowledge articulation pertinent to software development as well as it provides means to annotate manually or semi-automatically this kind of knowledge in order to foster easier knowledge acquisition and sharing with the usage of semantic search and a P2P metadata infrastructure. The semi-automatic annotation mechanism provides a way to classify source code in a quick manner, thus facilitating code reuse when a developer searches for code that is related to a software engineering or domain concept. For example, the developer might be interested in finding code that is written in Java for bank applications. Furthermore, KnowBench provides intuitive visualization and navigation through this knowledge exploiting the wiki paradigm.
KnowBench supports many advanced methods for knowledge search by (i) taking into account three different types of search, namely keyword, structured and semantic search; (ii) integrating results from different repositories (i.e. local and P2P storage); and (iii) including user feedback and similarity in the search process to refine information needs. While keyword-based search represents the standard model for search interfaces, structured search allows more precise queries which in turn yield more precise results. Novelty of the search lies in the combination of keyword and structured queries and in the usage of the context information to restrict the search space. Structured search provides more accurate result than present-day keyword search. KnowBench generates formal queries based on keywords. To realize this, a novel approach is used for adapting keywords to structured queries: the approach automatically translates keyword queries into formal logic queries so that developers can use familiar keywords to perform structured search.

KnowBench is seamlessly integrated into the Eclipse environment and thus gives software developers the incentives to develop or explore knowledge without bothering to open an external tool or web browser to accomplish this task which has the advantage of leaving the user in his/her current work context and not distracting him/her from the rest of his/her tasks. For example, he/she can develop source code and at the same time document problems that he/she encounters or provide guidelines and workarounds for a specific problem in source code that is commonly known to his/her organization by simply choosing to open the DevWiki editor next to the source code editor.

On the other hand, KnowBench has some limitations. It does not provide support for the whole software engineering lifecycle (e.g. design and requirements engineering). It focuses in one phase of the lifecycle — software development.

One issue that is not trivial to solve is that the knowledge in the P2P network is not always up-to-date. Once published, knowledge cannot be revoked/updated whenever “source knowledge” is deleted/modified.

The semantic annotation of source code provides granularity which is restricted by the underlying Eclipse platform itself as the IJavaElement interface is exploited to map between source code fragments and metadata. This limits the selected source code fragments to be annotated in the nearest Java elements that surround the source code fragment at hand (e.g. package/class/method/variable etc.).

Finally, KnowBench does not provide an ontology editor to alter the underlying ontologies. An external tool has to be used for that purpose.

7. Conclusions

In this paper we presented the KnowBench system — a novel knowledge management system that integrates into the daily software development process (Eclipse IDE) and can be used for capturing knowledge and experience as soon as it is generated by providing lightweight tools based on Semantic Web technologies. Semantic web technologies provide the driving force to better manage knowledge in
software development activities inside KnowBench. KnowBench offers an easy to use environment to facilitate knowledge articulation pertinent to software development. Additionally, it provides means to annotate manually or semi-automatically this kind of knowledge in order to foster easier knowledge acquisition and sharing by exploiting a semantic search engine and a P2P metadata infrastructure. Thus, better and more flexible collaboration among software developers scattered across the globe is facilitated.

Since manual annotation of source code may be very time-consuming and error-prone activity, the KnowBench system takes advantage of ontology learning and information extraction techniques to propose annotations. Both manual and semi-automatic semantic annotation of source code is facilitated by automated generation of corresponding metadata that assist also in describing and mapping Eclipse resources to the knowledge base.

The semi-automatic annotation mechanism provides a way to classify source code in a quick manner, thus facilitating code reuse when a developer searches for code that is related to a software engineering or domain concept. For example, the developer might be interested in finding code that is written in Java for bank applications. Furthermore, KnowBench provides intuitive visualization and navigation through this knowledge exploiting the wiki paradigm and the graph metaphor. All functionality is integrated into the Eclipse environment which is convenient for the software developer as he/she does not have to switch environment in order to articulate or use knowledge.

The evaluation of the system assisted us in determining what improvements may further optimize it. Evaluators’ responses provided us with valuable feedback. Some examples of improvements that would be nice to have are: (a) some parts of the ontologies should be more detailed and more structured, (b) the wiki language should be easier to understand and learn, (c) it would be nice to keep a history of the changes of each knowledge item to see who changed what, (d) visualization of all knowledge items shared by a specific user (the owner or a colleague), (e) the P2P search should be faster, (f) include personal tags to some of the knowledge items, (g) faster response time with respect to searching/visualization of shared knowledge items, (h) improvement of the install and uninstall procedure and (i) KnowBench is limited to work with Java 6; some evaluators expressed the need to work with Java 5 as well.

Furthermore, we are planning to enhance the semi-automatic semantic annotation mechanism in order to derive more precise annotations by introducing JAPE grammar rules [24] of the GATE system that will be customized for source code. Additionally, we want to further extend the manual semantic annotation in order to add the capability of annotating arbitrary lines of code regardless of the surrounding Java elements. However, this is a hard to solve and non-trivial issue.

During the evaluation of KnowBench we collected enough feedback in order to improve the system. The next phase would be inevitably to improve the system by incorporating the suggestions and the lessons learned during the evaluation of the
system. Enough feedback and expertise exists already for turning KnowBench into a more efficient and effective platform for KM in software development.

Acknowledgments

This work was partly supported by the TEAM project, which is funded by the EU-IST program under grant FP6-35111. The authors are responsible for the content of this publication.

References


38. T. Dingsøyr, Knowledge Management in Medium-Sized Software Consulting Companies, PhD thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, 2002.


