BEYOND INFORMATION SILOS – AN OMNIPRESENT APPROACH TO SOFTWARE EVOLUTION

Juergen Rilling

Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
rilling@cse.concordia.ca
www.cse.concordia.ca/~rilling

René Witte

Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
rwitte@cse.concordia.ca

Philipp Schuegerl

Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
p_schueg@cse.concordia.ca

Philippe Charland

System of Systems Section
Defence R&D Canada – Valcartier, Quebec, Quebec, Canada
philippe.charland@drdc-rddc.gc.ca

Nowadays, software development and maintenance are highly distributed processes that involve a multitude of supporting tools and resources. Knowledge relevant for a particular software mainten-ance task is typically dispersed over a wide range of artifacts in different representational formats and at different abstraction levels, resulting in isolated 'information silos'. An increasing number of task-specific software tools aim to support developers, but this often results in additional challenges, as not every project member can be familiar with every tool and its applicability for a given problem. Furthermore, historical knowledge about successfully performed modifications is lost, since only the result is recorded in versioning systems, but not how a developer arrived at the solution. In this re-search, we introduce conceptual models for the software domain that go beyond existing program and tool models, by including maintenance processes and their constituents. The models are sup-ported by a pro-active, ambient, knowledge-based environment that integrates user, tasks, tools, and resources, as well as processes and history-specific information. Given this ambient environment, we demonstrate how maintainers can be supported with contextual guidance during typical maintenance tasks through the use of ontology queries and reasoning services.

1. Introduction

Software is used to implement solutions that are expected to change periodically to adapt to ever-changing environments [1]. The efficient management and execution of these

changes are critical to software quality and evolution [2]. Managing and supporting software maintenance create an ongoing challenge for both the research community and tool developers, due to the variations and interrelationships that exist among software artifacts, tool resources, maintenance processes, and tasks [3, 4, 5, 6].

Software development and maintenance usually take place in a distributed environment, involving a multitude of supporting tools and resources, integrated in complex and often partially defined workflows and processes. On a technical level, software maintainers use numerous tools while working on a certain task to help them understand and modify the involved artifacts [4, 7, 8]. These include classical software development tools such as integrated development environments (IDE), but also knowledge-based tools, from group emails to (largely) static documents, such as library and process descriptions, to dynamic blogs and wiki systems. Identifying knowledge resources that are applicable in a given maintenance context can become a major challenge for software maintainers. However, the required knowledge often exists implicitly due to actions performed by another maintainer within the group. Someone might have solved a similar task using a different tool or a combination of tools, accessed certain resources [7, 9, 10] to help him, and also exchanged emails, wrote documentation, or edited wiki pages. If we can capture, combine, and automatically apply this 'ambient' knowledge, starting from a whole software development group up to a (virtual or physical) organization, we can bring software maintenance to a completely new level: an omnipresent, active software maintenance environment.

It is important to point out that the idea behind our approach has been used successfully in many other domains, like online shopping. Websites such as Amazon.com are not passive blank pages that wait for explicit procurement instructions. Instead, they constantly and pro-actively suggest new products and services, based on the user's current status, history, and the combined actions of other shoppers. We aim to bring a similar experience to the software engineering domain: a pro-active, ambient, knowledge-based environment that maintains users and their tasks, tools and resources, processes and histories, in order to better support developers while working on complex systems.
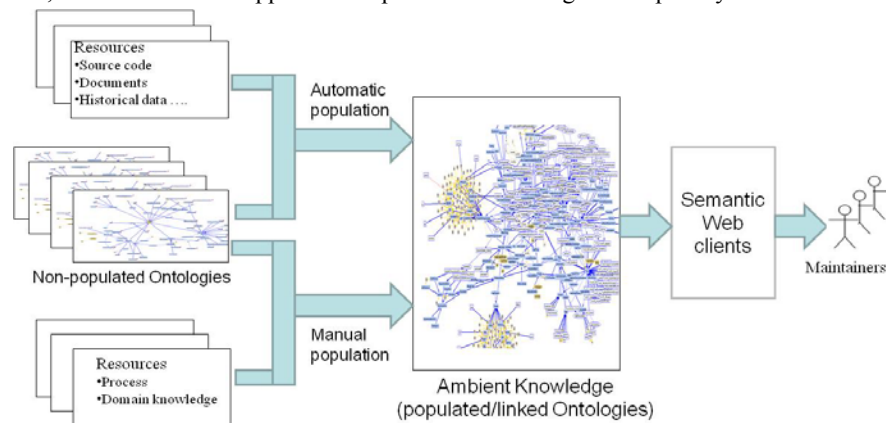

Fig. 1 Overview - Omnipresent process environment

On the conceptual side, our approach is based on ontologies and Semantic Web [11] technologies. We developed ontological models for the software domain that go beyond the typically used program and tool models, by including maintenance processes and their constituents [3, 5, 12, 13, 14]. By automatically capturing information from various artifacts through ontology population, we can build an extensible, distributed knowledge base. This extensible model not only includes existing tools and artifacts (e.g., source code, documents), but also their relevance within a certain program comprehension or maintenance task context and knowledge collected through mining the behavior of developers while solving similar tasks (Figure 1). In this paper, we do not propose the adaptation of a new tool or development process, but rather examine how existing resources can be integrated to implement the next generation of software maintenance environments, which is an important contribution neglected by current research [5, 15].

The remainder of this paper is organized as follows: Section 2 provides a brief review of existing software maintenance research and its limitations. Section 3 introduces our approach of an ambient software maintenance environment. Section 4 describes its implementation, in particular the ontology-based technologies used in our approach. How this approach is applied in practice is described in Section 5. An evaluation, based on a case study, is presented in Section 6. Related work is discussed in Section 7, followed by conclusions and future work in Section 8.

2. Software Engineering Foundations

Before we describe our approach in detail, we first introduce the software engineering foundations relevant for our research: Software maintenance (Section 2.1) and its underlying processes (Section 2.2).

2.1. *Software Maintenance*

As software ages, the task of maintaining it becomes more complex and expensive. Software maintenance, also often referred to as software evolution, constitutes the majority of the total cost occurring during the life span of a software system [16, 17]. With the ever increasing number of computers and their support for business processes, an estimated 250 billion lines of source code were being maintained in 2000, with that number rapidly increasing [17]. The relative cost of maintaining and managing the evolution of this large software base represents now more than 90% of the total cost [65] associated with a software product.

Software maintenance is a multi-faceted domain. Its aspects range from low-level technical aspects (e.g., source code, operating systems and tools such as compilers and editors) over organizational and legal concerns (e.g., prescribed workflows, international standards) to social and cognitive aspects (e.g., communication behavior and cognitive models) [4, 8]. Providing automated support that addresses these concerns is difficult, due to the different representations and interrelationships that exist among software artifacts and knowledge resources [1, 5, 18, 19]. From a maintainer's perspective, exploring [8] and linking them become a key challenge [6], as they have to comprehend a multitude of disconnected artifacts created originally as part of the software development process

[20]. This can be interpreted as a knowledge-driven process, where information is being continually integrated from different sources (including source code repositories, documentation, test case results) and at different abstraction levels (from single variables to complete system architectures). Moreover, for maintainers to be able to perform and complete a particular maintenance task, they typically need to use and interact with various tools and techniques (e.g., parsers, debuggers, source code analyzers, visualization tools). Helping developers in identifying and correctly applying available resources is a challenge for both the individuals and organizations [4, 5, 6, 17]. One approach lies in explicitly describing and abstracting the activities that have to be performed by a developer to successfully perform a particular task. These descriptions are formalized through *process models.*

2.2. *Software Evolution Process Models*

Various process models [3, 5, 13, 14, 17] supporting the evolution of software have been introduced. Historically, they have focused on the software development cycle. However, as pointed out previously, with much of a system's operational lifetime cost occurring during the maintenance phase, this should be reflected in both the development practices and process models supporting maintenance activities. One approach to model software maintenance is to include its aspects as part of the total system life cycle/process model, as suggested for example in [3, 13, 21]. Other approaches include deriving maintenance-specific process models. Among these models are the quick-fix, iterative enhancement, and full-reuse model [3], the staged model [6], the SEI CMMI model [13], and the IEEE Std 14764-2006 – Software Engineering – Software Life Cycle Processes – Maintenance [14].

One of the challenges in applying these models is that various aspects can affect software evolution [5, 7, 22], making it an inherently complex and difficult problem to address. Some of these aspects include the user's ability (e.g., experience, knowledge background), the characteristics of the software system to be maintained (e.g., its application domain, size, and complexity), the maintenance task itself to be performed (e.g., adaptive, corrective, or perfective maintenance), as well as the tools and software artifacts available to support it.

Common to most existing process models, including evolution models, is that they do not specify *how* any available supporting resource (e.g., tool, system, user expertise, software artifact) should be integrated within the process in a given context. Current research in software maintenance focuses mostly on providing conceptual guidance (such as the documented standards) or on developing tool support to address some specific aspects of a software maintenance task. An example of such a process model is the IEEE Std. 14764-2006 [14] that lists and describes activities and their sub-activities, referred to as *task-steps*, as part of the process model.

It is possible to enrich these models by including additional details, by not only describing as part of the process the required activities to be performed, but also the resources that must be employed during these activities. However, deriving such well defined processes requires that the knowledge to be incorporated and used to support them

is known already at the time of their definition. Furthermore, these processes tend to be quite static in their nature, lacking both the ability to dynamically integrate new knowledge resources and the ability to provide guidance that is specific to a given maintenance context. It is out of this observation that our idea was born to create a dynamic, context-aware knowledge base that can support the existing process models with active and adaptive support, based on a formal model of the users (software maintainers), available artifacts (tools, resources) and tasks (process model steps).

3. Ambient Semantic Software Maintenance Support

We now present our contribution, an ambient semantic software maintenance environment. Its goal is to support developers throughout maintenance tasks (as discussed in Section 2.1) by providing a context-sensitive knowledge base that can be queried either directly by a user or indirectly through supporting tools. More precisely, our environment supports maintainers by managing two knowledge-intensive aspects of software evolution:

1. Collecting and maintaining semantic links, i.e., *traceability links,* between software artifacts, in particular those at different abstraction levels, such as source code and its associated documentation.
2. Maintaining knowledge about available tools, software evolution processes, users, and their history of solving tasks with the available artifacts, to provide contextual guidance during complex maintenance tasks.

The first goal is motivated by the observation that maintainers tend to spend a large amount of manual effort on synthesizing and integrating information from various sources in order to establish their connections, i.e., the traceability links [23, 24, 25]. Existing research in software process modeling focuses on reducing the cost associated with this manual effort by developing automatic assistance in establishing and maintaining traceability links among software artifacts [24]. Various software and software evolution artifacts, like requirements, design documents, or bug reports, contain a large amount of information in the form of descriptions and text often written in natural language. These documents combined with source code represent two of the main software artifact types utilized in software evolution [23, 24]. Existing source code/document traceability research [23, 25] mainly focuses on connecting documents and source code using Information Retrieval (IR) techniques. However, these IR approaches ignore structural and semantic information that can be found in both documents and source code, limiting therefore both their precision and applicability. In previous research [26, 27], we already introduced a formal ontological representation that covers source code and documentation and shows how to establish these links through a combination of code analysis and text mining techniques. We address our stated second goal by providing maintainers not only with the ability to explore knowledge relevant to their given task across different artifacts, but also to support an iterative approach to enrich their current understanding of a system. In this paper, we demonstrate some recent additions to the traceability support by

modeling the domain of version control systems (e.g., CVS) and bug report systems (e.g., Bugzilla), for which we added automatic ontology population support.

To deliver the context-sensitive software evolution process support, we have to additionally enhance our model of the software domain [28]. That is, we have to establish connections not only between various software artifacts, but also between these artifacts and the process model(s). This involves a number of steps:

- Conceptualizing the software domain, including users and processes (Sections 3.2 and 3.3).
- Creating a knowledge base by automatically populating the sub-ontologies using static code analysis for source code (Section 4.2.1), text mining for documents (Section 4.2.3), and the analysis of CVS and Bugzilla repositories (Section 4.2.2).
- Establishing traceability links between source code, documents, and process ontologies through ontology alignment (Section 4.2.4) and by resolving the inconsistencies (Section 4.2.5).

Before we describe our model in detail, we briefly introduce the foundations used for our formal knowledge representation model, namely ontologies based on description logics.

3.1. *Ontologies and Reasoning*

The term "ontology" originates from philosophy, where it denotes the study of existence. In computer science, the most common definition has been provided by Gruber [29]: "An ontology is an explicit specification of a conceptualization." Ontologies are typically used as a formal and explicit way of specifying the concepts and relationships in a domain of discourse. Ontologies can overcome portability, flexibility, and information sharing problems associated with databases [30]. Compared to relational approaches, which assume complete knowledge (close world assumption), ontologies support the modeling of incomplete knowledge (open world assumption) and extendibility of the ontological model. [29, 31]. Semantic Web technologies allow for machine understandable Web resources that can be shared and processed by both software tools (e.g., search engines) and humans [32]. Ontologies are an important foundation of Semantic Web enabled technologies, as they allow sharing knowledge between different agents and creating common terminologies for understanding [32]. They are also an important step towards enrichment of services and content of the next generation of the Internet ("Web 3.0").

The Web Ontology Language (OWL) [33, 34] is a standard put forward by the W3C. It provides for creating machine understandable information to enable the automatic processing and integration of Web resources. The sub-language OWL-DL, which we use in our approach, is based on Description Logics (DLs) [31]. Using OWL-DL allows us to enrich our platform with *reasoning* services provided by DL-based knowledge representation systems [31]. Unlike many logic programming approaches that cannot guarantee completeness, DL reasoning services are proven to be sound, complete, and terminating. Moreover, DL reasoning is automatic and does not require the development of logic programs to extract the desired inferences. DL reasoning is usually performed on demand and triggered by relevant queries to the knowledge base. OWL ontologies can also simply

be used as a data storage medium, similar to traditional databases. However, the use of DL to define the ontological model allows for a more precise and expressive representation than traditional data semantics [31].

Based on the above discussion, we adopted OWL-DL as the ontological modeling language for our research. OWL-DL representation is applicable to model the dynamically evolving software maintenance domain, due to its support for domain knowledge evolution (concept, role, and instance creation and population). In addition, it does not only provide a more precise and expressive representation of the domain, but also enables automated reasoning to process the content of the represented domain. For a more detailed discussion on OWL ontologies, DL, and reasoning, we refer the reader to [31, 34, 35].

3.2. *Software Artifact Ontology*

As stated earlier, one of the goals of our work is to enable maintainers to be immersed in an omnipresent environment that integrates knowledge resources at different abstraction levels. In particular, we are concerned with "deep links" between individual entities at the process level (tasks, task-steps, source code side - e.g., a class or method) and the corresponding mentions of these entities in a document at the level of individual words or phrases. That is, in contrast to Information Retrieval (IR) approaches [24, 25], we are not only interested in creating links at the level of complete documents or paragraphs, but also much more fine-grained links that precisely show the connections between individual words and code entities. Obviously, these links can only be created between entities that are shared among different knowledge resources. But documents and process level descriptions may also contain many more higher-level concepts that cannot be directly mapped to a single code entity, such as algorithm descriptions, architectures, or requirements. It is here that our ontology-based method provides a strong advantage compared with existing, semantic-poor approaches, as it allows us to explicitly encode knowledge of the software and process domain in a formal language that can be automatically evaluated, e.g., through queries and reasoning. For example, we can model knowledge about change impact (e.g., bug reports, log files) and their relations with other entities (e.g., classes, methods) as an ontology. An end user, such as a software maintainer, can then navigate a single, unified, and formal representation that covers process, source code, and documents. This navigation ability allows for a number of novel use cases, such as the automatic creation of traceability links and the creation of contextual views through both pre- and user defined queries.

Building a formal ontology for software evolution requires the analysis of concepts and relations in this domain of discourse. From a software practitioner's perspective, it is therefore essential that the ontological knowledge base includes and models concepts and roles critical to software evolution processes. Indeed, our conceptualization work has been influenced by other works on software maintenance modeling [18, 23, 36, 37, 38] and the observations of best software maintenance practices. Existing work on ontological modeling of the software engineering domain, including its processes, has focused on conceptualizing the domain [10, 38] to establish a common terminology or to model specific aspects of software engineering processes [10, 38, 40]. In the context of our re-

search, we adopt core parts of these ontologies and further enrich them with new concepts and relationships to (1) more closely reflect the particular needs of our specific modeling goal – establishing traceability links between the process and various knowledge related resources. (2) Provide a design that fully supports and utilizes optimized DL reasoners, such as Racer [35] or Pellet[1], to infer additional knowledge.

3.3. *Software Evolution Process Ontology*

Modeling the software evolution process requires an extension of the artifact ontology with a number of additional concepts. At the upper level, we integrate sub-ontologies through a number of abstract concepts, like *tool, task, artifact,* and *user.* An artifact can be anything involved in the software maintenance process. In particular, emails, source code files, natural language documents, CVS entries, or bug reports are sub-classes of *artifact.* Figure 3 provides a simplified view of the resulting ontology model, focusing only on the major concepts and their roles. In what follows, we briefly summarize these key sub-ontologies.

Task: Describes a unit of work that is triggered by an emerging modification request (MR) or problem report (PR). Information about MRs or PRs, task assignment, tool log (tool used in the task solving process), activity log, etc. is modeled in this sub-ontology. For example, the instances in this sub-ontology might be "Debrief XML decrypt/encrypt component substitution request" (an instance of concept `Request`), "corrective" (an instance for concept `RequestType`), and "successfully finished" (an instance for concept `TaskStatus`).
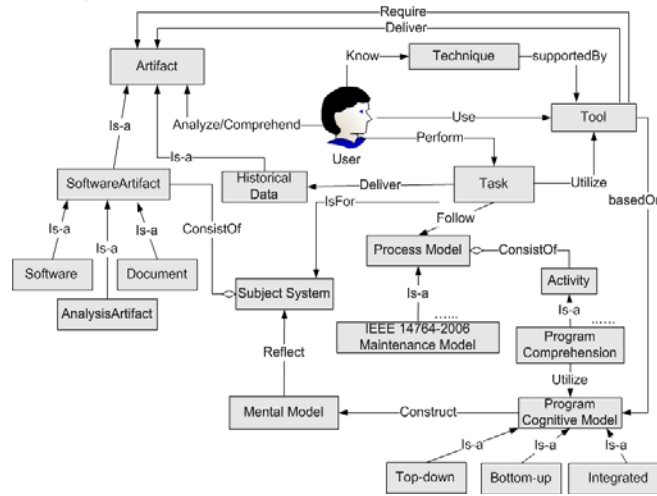

Fig. 2 Software maintenance meta-model

User: Describes maintainers involved in the software maintenance process. Information about the involved roles and their related skills, competencies and responsibilities are modeled. The users involved in the software maintenance can be an individual, an organizational team, or an entire organization. The example instances in this sub-ontology might be "Mike Smith" (an instance of concept `Person`), "SE group" (an instance of concept `Maintainer`), and "MR reporter" (an instance of concept `Role`).

Artifact: Describes artifacts associated with both the software product (e.g., source code, documentation) and the maintenance process (e.g., documents, models) (Figure 3). It has two sub-ontologies: `SoftwareArtifact` and `TaskArtifact`. Software artifacts are described in Section 3.2. Task artifacts are process-related artifacts, such as guidelines, reports.
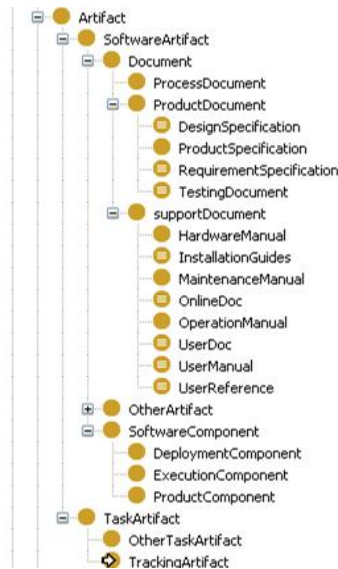

Fig. 3 Partial view of the `Artifact` sub-ontology

Subject system: Describes the software system to be maintained and includes information about the programming language(s) used, application domain, etc.

Process: Describes the interactions and relationships among different sub-activities within a maintenance process model. The example instances in this sub-ontology can be IEEE Std 14764-2006 – Software Engineering – Software Life Cycle Processes – Maintenance" (an instance of concept `ProcessModel`), "5.3 Modification Implementation" (an instance of concept `Activity`), "5.3.2.1 Analysis" (an instance of concept `Task`), and "5.3.2.1 a" (an instance of concept `TaskStep`) [14].

Technique: Describes the software techniques that can be used for supporting software maintenance (e.g., program comprehension technique, source code analysis technique, impact analysis technique). Based on the previous work by Dias et al. [18] and Pressman [41], the techniques supporting the following activities can be identified: requirements elicitation, maintenance support, programming related, testing, configuration management, documentation, and modeling.

Tool: Describes the software tools used to carry out a function or service with the goal to simplify maintenance tasks [19]. At the current stage, our tool sub-ontology provides general categories for commonly used maintenance tools and describes their key features for supporting software maintenance activities. Tools can be classified by their functionality, their role in supporting managers and maintainers, their particular use during the various steps of a software engineering process, their supported environment (hardware and software), or even by their origin or cost. Pressman [41] provides a comprehensive classification of tools by their functions, which includes, among others, analysis, design, programming, software configuration management, testing and documentation tools. In our approach, tools and techniques in the knowledge base (KB) are automatically classified through the use of reasoning. For example, programming tools can be classified based on their support for a particular programming language, operating system, applicability for a specific task, etc.

4. Implementation

In this section, we provide an overview of our implementation and describe its system architecture (Section 4.1) and the various ontology population subsystems (Section 4.2). We also discuss our strategy for managing inconsistencies in the ontological knowledge integration process (Section 4.2.5).

4.1. *System Architecture*

While developing a context-aware system, two key aspects have to be taken in consideration. First, maintainers spend a significant amount of time searching and exploring source code within their Integrated Development Environment (IDE) to comprehend the overall structure of a system [4, 5, 22]. Second, context changes caused by the need to switch between applications, views, etc. should be limited to ensure that a user's comprehension context is not lost. For example, in modern IDEs, contextually relevant information with respect to the currently edited or analyzed source code is automatically made available (e.g., methods available, auto-completion of partially typed source code). Similarly, we have integrated our ambient knowledge environment through Eclipse by taking advantage of the extensibility provided by the Eclipse plug-in architecture.

We selected a client-server architecture to support a centralized management facility, shown in Figure 4. Its key components are an Eclipse plug-in and a web server. The Eclipse plug-in is responsible for collecting context information, forwarding requests to the server and displaying results. The server, on the other hand, provides persistent ontology support, aligns the sub-ontologies and provides ontology management services, including

querying/reasoning and data mining services for the clients. The server side utilizes the Protégé API and the Jena Semantic Web Framework[2] to access ontologies. Jena provides the backend repository for both concepts (TBox) and instance data (ABox). Leveraging Jena's persistent storage support, the model storage can be based on files or on a relational database. TBox management is centralized on the server to ensure consistency, quality of the ontology design, and standardization of the KB. Reasoning services are provided by Pellet, which provides optimized TBox and ABox reasoning capabilities.
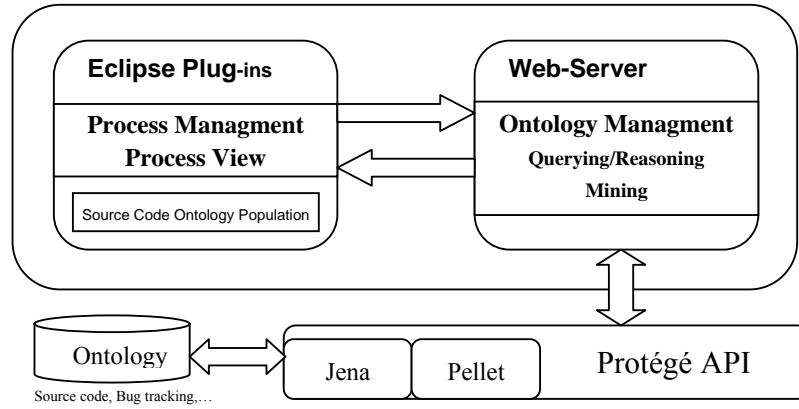

Fig. 4 System architecture overview

A successful context-aware support system is based on the premise that the right information should be made available at the right time in the right format. From a software maintainer's perspective, the challenge is specifically the need to adjust both information and context in real time to the current process task-step being performed. Our implementation thereby follows two strategies. First, process information is made explicit and visible to the current developer. This is achieved by developing an Eclipse *Cheat Sheet* plug-in to provide extended process information. Secondly, information from the current Eclipse workspace is maintained and process context sensitive information is provided by querying the underlying KB.

The client provides the foundation for our visual integration, supporting the interaction and linking between the maintenance process activities and the relevant knowledge resources. Similar to more traditional database applications, ABox management is provided through the client.

4.2. *Ontology Population*

One of the major challenges for software maintainers is the large amount of information that has to be explored and analyzed as part of typical maintenance activities. Therefore, support for automatic ontology population is essential for the successful adoption of Semantic Web technologies in supporting software maintenance. In this section, we de-

---

[2]   http://jena.sourceforge.net/

scribe the automatic population of two of the sub-ontologies from existing artifacts, like source code, Bugzilla, SVN and documents.

4.2.1. *Source Code Ontology Population*

Concepts in the source code ontology typically have a direct mapping to source code entities, allowing instances of these concepts to be automatically recognized by our source code ontology population subsystem. Within our implementation, we utilize the Eclipse JDT compiler in order to read the source code, perform common tokenization and syntax analysis to produce an *Abstract Syntax Tree* (AST). Our population subsystem traverses the AST created by the JDT compiler to identify concept instances and their relations, which are then passed to an OWL generator for ontology population (Figure 5). Furthermore, it can also identify instances of roles (i.e., relations between source code entities) by statically analyzing the source code.

The number of instances and relations identified by our system depends on the complexity of the ontology and the size of the source code to be analyzed. At the current stage of our research, we limit the population of the source code ontology to 38 of the higher-level concepts (classes) and 41 types of relations (ObjectProperties). We restrict the ontology population currently to these high-level concepts (e.g., package, class, method) due to the application domain of the ontology. Our objective is to support system evolution at a higher level of abstraction, rather than focusing on specific, low-level source code analysis (e.g., at the expression level).

4.2.2. *CVS and Bugzilla Ontology Population*

CVS and Bugzilla are two typical examples of bug tracking and version management systems that are used in both open source and industrial applications, containing typically large amounts of software maintenance relevant information. We use these existing knowledge bases to populate our CVS and Bugzilla sub-ontologies. We use Mylyn[3], an Eclipse plug-in that provides connectors to Bugzilla and CVS as well as supports the data export from these repositories. Both the version management ontology (consisting of concepts `File`, `Revision`, `Release`) and the bug tracking ontology (consisting of concepts `Issue`, `Milestone`, `Person`, `Product`, `Activity`, `Comment`, `Attachment`, `Component`, `Computersystem`) have been integrated using an existing ontology model [42]. Through the use of Mylyn, we can now populate our CVS and Bugzilla sub-ontologies and make this knowledge an integral part of our ambient knowledge base.

4.2.3. *Documentation Ontology Population*

We developed a custom text mining system to extract knowledge from software documents and populate the corresponding sub-ontology [28, 43]. The system reads software documents and uses a number of Natural Language Processing (NLP) techniques to identify *named entities* that correspond to one of the concepts defined in the software ontolo-

---

[3]   http://www.eclipse.org/mylyn/

gy, like *person, class, method, software architecture,* or *design pattern*. A particular feature of our system is that it can use the results from the automatic code analysis described in Section 4.2.1 to facilitate the detection of corresponding entities in natural language documents. For further details on our software text mining system, we refer the reader to [28].

4.2.4. *Ontology Alignment and Linking*

After having populated and represented the various software artifacts in the form of ontologies, traceability links are established through ontology alignment [11, 44]. Traceability links between source code and documentation are created based on concept and instance information. Since our documentation, Bugzilla, and source code ontologies share many concepts from the programming language domain such as Class or Method, we apply in this research an instance matching approach, based on common names and properties. At the current stage of this research, we focus more on matching instances that have been discovered from source analysis and text mining. Our text mining system can additionally take the results of the source code analysis and bug report analysis as semantic input when detecting named entities. This allows us to directly connect instances from the source code, Bugzilla and document ontologies. For example, our source code analysis tool may identify c_1 and c_2 as classes, and this information is used by the text mining system to identify the named entities c'_1, c'_2, and their associated information in the documents. As a result (exemplified in Figure 5), source code entities c_1 and c_2 are now linked to their occurrences in the documents (c'_1 and c'_2), as well as other information about the two entities mentioned in the document, such as design patterns, architectures, etc.
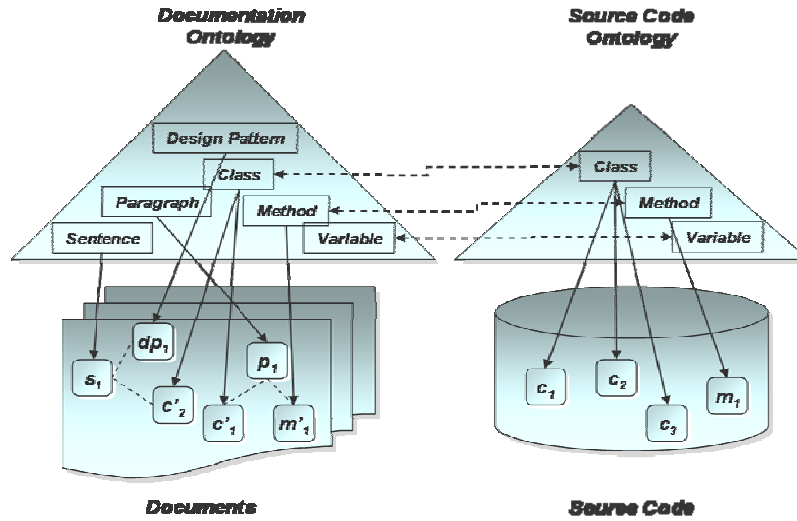

Fig. 5 Linking instances from source code and documents

After the source code and documentation ontology have been linked, users can then perform ontological queries on either documents or source code, regarding properties of c_1 or c_2. For example, retrieve document passages that describe both c_1 and c_2, or retrieve design pattern descriptions referring to the class that contains the class currently analyzed. Note that the alignment process might also identify inconsistencies – the documentation might list a method for a different class, for example – which are detected through the alignment process and registered for further review by the user.
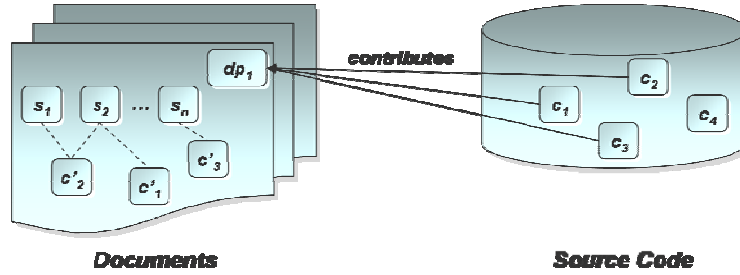

Fig.6 Retrieve implicit information from documents

In addition, users can always manually define new concepts/instances and relationships in both ontologies to establish the links that cannot be detected by the automated alignment. For example, as Figure 7 shows, the text mining system may detect an instance of DesignPattern – dp_1 and users can then create the relationships between the pattern and classes that are contributed the pattern (e.g., c_1, c_2, and c_3) through our query interface. The newly created links then become an integrated part of the ontology, and can be used to, for example, retrieve all documents related to the pattern (i.e. s_1, s_2, …, s_n).

Furthermore, documents can not only be linked to source code, but also to design-level concepts that relate to particular software maintenance tasks. For example, in contrast to the serialized view of software documents, i.e., sentence by sentence, or paragraph by paragraph, our formal ontological representation of software documentation also provides the ability to create hierarchical documentation views. Using the classification service of the ontology reasoner, one can classify document pieces that related to a specific concept or a set of concepts.

4.2.5. *Inconsistency Management*

The problem of ontology change is far from trivial. The large size of modern day ontologies makes this problem even more complicated [75]. Furthermore, the Semantic Web is characterized by decentralization, heterogeneity and lack of central control or authority. These features have greatly contributed to the success of the web but they have also introduced several new, challenging and interesting problems, not previously found in AI. Given these heterogeneous environments, knowledge integration, as we perform it for the software domain, becomes also the management of inconsistent information. It is not realistic to expect all the sources to share a single, consistent view at all times. Rather, we

expect disagreements between individual users and tools during an analysis. An elegant model for managing (possibly conflicting) information from different sources has been proposed by [45]: Knowledge is structured into *viewpoints* and *topics*. Viewpoints are environments that represent a particular point of view (e.g., information stemming from a particular *tool* or entered by a *user*). Topics are environments that contain knowledge that is relevant to a given subject (e.g., design patterns, architectural recovery). These environments are *nested* within each other: viewpoints can contain either other viewpoints or topics. A topic can contain knowledge pertaining to its subject, but also other viewpoints, e.g., when the subject is another user.

Through this approach, we can explicitly distinguish between knowledge a particular resource has about a certain topic and at the same time, manage possible conflicting knowledge about what a resource *believes other resources may believe* about the same topic, since this information is contained within different, nested viewpoints. These viewpoints create *spaces* within which to do reasoning: consistency can be maintained within a topic or a viewpoint, but at the same time, conflicting information about the same topic can be stored in another viewpoint. This allows us to collect and maintain as much knowledge as possible, attributing it to its sources, without having to decide on a "correct" set of information, thereby losing information prematurely. For example, a user might believe that a certain set of classes form the *bridge design pattern*, while the documentation states they belong to an *architectural layer*, and the design pattern analysis tool identifies them as a *factory pattern* (Figure 8).
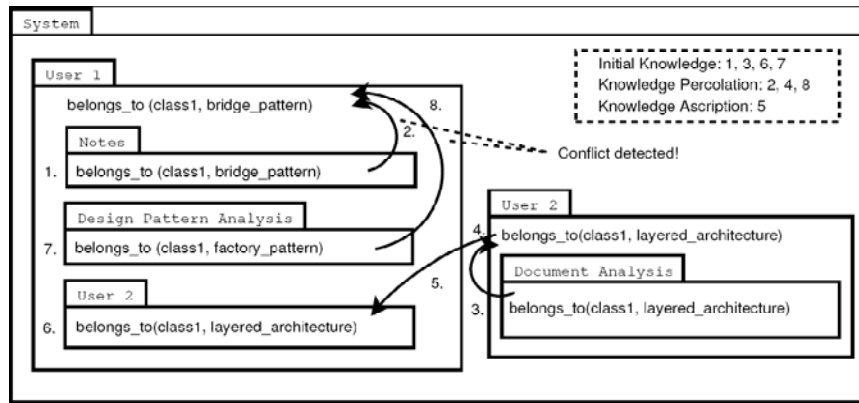

Fig. 7 Knowledge merging using ascription and percolation

Viewpoints can be constructed as well as destructed through the processes of ascription and percolation. Stated briefly, the process of ascription allows incorporating knowledge from other viewpoints (users, tools) unless there is already conflicting information on the same topic. The mechanism of percolation is introduced for the deconstruction of nested knowledge. Here, some (assumed) held knowledge on a topic contained in a nested viewpoint may be percolated into its outer environments, up to the top-level viewpoint of a user (or the main environment) and be thus acquired as knowledge.

Several philosophical problems related to the adaptation of knowledge in general have been identified in the research area of belief change (also known as belief revision) [72, 73, 74]. These have also been applied to knowledge represented in ontologies [66, 78]. One such work, at a preliminary stage, is [76], where the authors propose the use of the AGM theory [39] for ontology evolution. Only some informal ideas regarding the connection of the AGM theory with ontology evolution are provided. The focus lies on the operations of contraction and revision, whereby, following the lead of the AGM postulates [39], certain properties that should hold in a rational contraction and revision operation are presented. However, these properties are not directly applicable to many DLs, for the same reasons that the AGM theory is not directly applicable to such DLs.

An interesting extension of a belief change algorithm appears in [66], where the authors attempt to recast the maxi-adjustment algorithm, originally introduced for propositional knowledge integration, to the context of DLs. This algorithm allows the elimination of any inconsistencies that could arise in a stratified KB after its expansion with a new proposition, which, in turn, allows the development of a revision algorithm. The authors propose a basic and a refined algorithm which apply the ideas of maxi-adjustment to DLs. Unfortunately, for their method to be applicable, the disjunction of DL axioms needs to be definable, which is not possible in standard DLs.

In our approach, we essentially decouple the strategy for revising the knowledge base (such as belief revision) from the representation of different beliefs (using viewpoints and topics). The percolation and ascription operators can be implemented using different strategies, for example, using an adapted version of fuzzy belief revision [79] as recently demonstrated in the Fuzzy Believer system [80].

5. Application

We can now discuss how to apply our environment in practice. In particular, we demonstrate that our system is not limited to static knowledge-based information retrieval, but also allows for a dynamic and context-aware guidance in which implicit as well as explicit information is processed. Thereby, developers and maintainers are provided with information beneficial in fulfilling their work efficiently. As pointed out before, our research focus is not on the development of a new software tool or technique in attempt to support a specific maintenance activity. Rather, our goal is to automatically integrate various existing tools and knowledge resources through the use of queries and automated reasoning within a maintenance process.

The application is discussed in three steps: In Section 5.1, we illustrate the use of ontologies in supporting software evolution. We describe the use of ontologies to support different comprehension strategies used to describe the cognitive process applied by maintainers to integrate various knowledge abstractions and model them. We also illustrate how questions arising during a maintenance process can be encoded in ontology queries

and answered by our system. Section 5.2 introduces the context-aware guidance layered on top of these low-level queries.

5.1. *Software Evolution Support*

Through the use of a standard ontology model, our environment provides built-in support for ontology queries. Thus, the central paradigm of our approach is to encode a higher-level question into an ontology query through a query language such as SPARQL[4]. The answer retrieved from the knowledge base can then be either used directly, or applied for further queries in order to provide a solution to the user.

5.1.1. *Software Comprehension Support*

In what follows, we describe the use of ontologies in supporting program comprehension as an essential part of any software evolution process, as well as their support for strategies on how to support cognitive models used in program comprehension. These comprehension strategies provide the essential guidance for programmers in understanding software systems and involve ontology exploration and automated reasoning. We now describe when and how different comprehension activities may be invoked in the context of specific comprehension strategies.

Bottom-up Comprehension Strategy

Using a bottom-up comprehension strategy, programmers understand a program based on *chunking* [67]. Chunks are portions of an artifact that the programmer recognizes. Larger chunks contain smaller chunks nested within them.

- The programmer pieces together his/her understanding by combining chunks into increasingly larger chunks. In the perspective of ontology-based comprehension, chunks are concept instances. Smaller chunks or larger chunks denote concept instances at different granularity levels, such as statement, method, class, or package levels. Therefore, recognizing smaller chunks, and formulating larger chunks are a set of activities called *concept identification and specification*. More specifically, in this context, concept identification refers to an activity by which programmers retrieve information related to a specific portion of a software artifact. The software ontology contains rich concept instances at different granularity levels, as well as the relationships among these instances. The ontology therefore can facilitate programmers in identifying concepts and relationships that are related to a specific portion of the code, e.g., a method he/she is currently reading. For example, during the reading of a smaller chunk – a variable usage, a programmer may query the ontology about the type of this variable, or the place where this variable is defined.
- In order to understand a larger chunk, such as a class, a programmer may first retrieve smaller chunks that are contained within the larger chunk, such as me-

---

thods defined in the class, or variables used in the class. Only once all retrieved smaller chunks have been comprehended, the programmer is able to obtain a complete understanding of the larger chunk. After such a localized understanding of a program chunk has been obtained, programmers often continue the comprehension process by introducing additional concepts or relationships with regards to their specific maintenance tasks, or specifying understood chunks as instances of existing concepts/relationships.

- The newly added knowledge then becomes an integrated part of the ontology and can be reused in other comprehension tasks. For example, in order to comprehend data structures implemented in a software system, a programmer may define new concepts such as `Array`, `LinkedList`, `Stack`, or `Heap` as part of the ontology. He/she thus can use these new concepts to specify recognized chunks, e.g., a class is an instance of `LinkedList`. This information can then be further reused by the programmer to identify larger chunks. For example, a `Stack` is implemented by a `LinkedList`.

Top-down Comprehension Strategy

In situations where a top-down comprehension strategy is adopted, programmers start reading source code with certain hypotheses and then confirm or reject these hypothesis based on identified evidences, the so-called *beacons* [68]. Within the ontology-based program comprehension model, a hypothesis can be regarded as a concept description or a query. When the concept description or the query returns one or more concrete instances, the hypothesis is confirmed and retained, becoming part of their understanding. Otherwise, the hypothesis is discarded.

- Programmers typically start the comprehension process by formulating hypotheses concerning properties of the program. The ontology language can help programmers to specify their hypotheses. For example, a programmer may suspect that a class implements a specific design pattern [69]. In a next step, an ontology query is composed and performed to validate if any of the software documents describe this class in terms of design patterns. Another example is that a programmer may hypothesize a software package is a self-contained component, i.e., there exists no dependencies to other classes/packages outside the package it is implemented. A simple query can help the programmer to verify this hypothesis by retrieving all classes that are used by this package, but not defined within it.
- The software ontology can answer queries through automated reasoning, and thus confirm/refute these hypotheses. Through each confirmation or refutation, the programmer obtains a better understanding of the program. In the above examples, the documentation ontology was used to identify whether any design patterns are related to the class; and the source code ontology returned the number of classes that are used by the package. These provided answers can help

programmers verify their hypothesis, and thus accelerate the comprehension process.

As-needed Comprehension Strategy

As the size and complexity of a software system increases, a complete understanding of the whole system is less feasible [70]. Instead, programmers often adopt an *as-need* strategy [71]. For this reason, programmers tend only to comprehend these portions of the program that are relevant for a specific software evolution task.

Adopting an as-needed strategy as part of the ontology-based program comprehension approach requires a programmer to start the comprehension process with an initial mental ontology. This mental ontology consists of a set of concepts/relationships that are considered essential for a maintenance task.

- Through reading different artifacts, instances of the initial ontology are comprehended in the first step (i.e., ontology populating). In the case more concepts/relationships are necessary, the ontology has to be extended, and more effort is required to comprehend instances of these newly introduced concepts/relationships. Therefore, the as-need comprehension process can also be regarded as an iterative process of concept recognition and relationship discovery, except that the comprehension goal is not to construct a complete mental model, but rather a minimum subset that is required for the completion of a particular comprehension task. The software ontology contains detailed information with regard to source code and documents.

- Programmers can identify certain subset of the ontology as their starting point. By exploring and querying the ontology, they can not only investigate specific properties of a concept/concept instance, but also enlarge the subset by utilizing existing or newly defined concepts/relationships. The process finishes once sufficient confidence was established that their understanding is sufficient to complete the particular task.

5.1.2. *Process support*

What makes the query paradigm so useful in our approach is the fact that the knowledge base integrates knowledge from different sources and at very different levels of abstraction. Thereby, even a seemingly simple query can provide insights to a software engineer that would otherwise require a significant amount of manual research. Furthermore, through the integration of the process model, questions that relate to tasks can be asked as well. For example:

- Which source code and documentation artifacts related to my current modification requests can support an early impact analysis as requested in process activity 5.3.1?
- What are the document passages (e.g., sentences, paragraphs) describing the source code entity (e.g., class, method) I am currently inspecting in my editor?

- Given a modification to a method implementation, which requirements documents would be affected by a change and who has the most expertise with making this change?

Note that these queries cross conceptual boundaries between process, code, and natural language document ontologies, which is not possible with other approaches that lack the ability to model these artifacts in a common formal representation.

In terms of the design-level links, OWL-DL axioms or concepts can be used to establish the links between the process model and the modeled knowledge resources. Relationships and properties, specified as OWL axioms can be used to establish implicit links as well. For example, in the IEEE maintenance process task-step *"5.3.2.1 a) Identify the elements to be modified in the existing system"* [14], the `supportedBy` property can be utilized to automatically link the task-step with supporting tool resources. The `supportedBy` relation in Figure 8 is defined as a transitive property used to link *Task Step* with *Technique* (1) as well as for linking *Technique* with *Tool* (2). Then the links between *TaskStep* and *Tool* can be automatically derived through the inferred link (3). As a result, each task-step in the process ontology can automatically be linked through this transitive `supportedBy` property to the sub-ontologies of *Technique* and *Tool*. Through further traversing of the transitive closure across the subontologies, other knowledge resources can be identified as well.
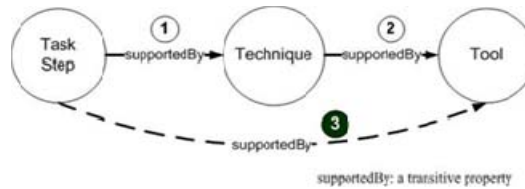

Fig. 8.Transitive design level links

In addition to these design-level links, logical links between the process and the knowledge resources can be established through predefined queries for each task-step.

Figure 9 illustrates the use of a set of predefined queries to support different process task-steps. Users can also define their own supporting queries based on their experience to extract, explore, and reason upon the information stored in the KB.
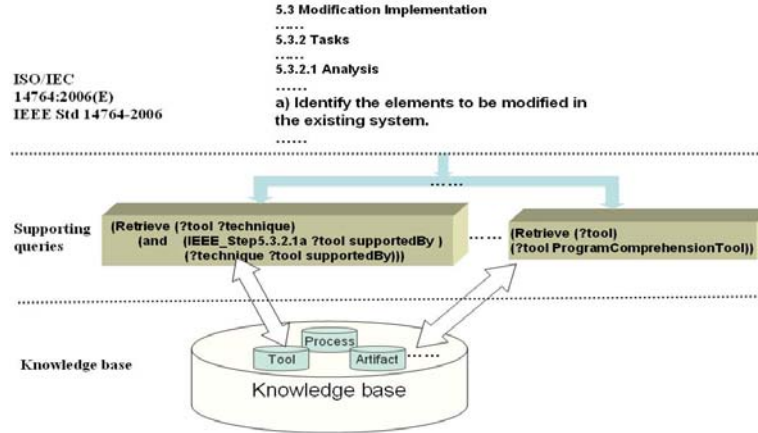
Fig. 9 Task supporting queries

These predefined queries supporting the different task-steps allow users to retrieve explicitly modeled knowledge, as well as inferring implicit knowledge across the KB to support a particular task-step.

5.1.3. *Eclipse Integration*

A successful process-centered support system has to continuously provide information to the user taking in consideration the current context. Thereby, contextual information originates from both a process perspective and the IDE's perspective. Achieving this premise depends greatly on the available knowledge resources and their accessibility. Our process environment explicitly supports such connections between the involved process steps and other supporting resources (e.g., artifacts, maintainers, techniques, and tools).

In what follows, we describe a general application scenario as supported by our environment. In a first step, a process and its activities are defined by either creating an organization specific process or by adopting and tailoring an existing process model and its workflow. In this example, we customize the IEEE maintenance standard [14], by specifying which of the IEEE process activities, tasks, and task-steps are required. The process model is defined through the use of the Eclipse Cheat Sheet mechanism, which also provides the option to assign Eclipse commands, ontology queries, and additional resources (such as help pages) to the various activities. Cheat sheets were originally contributed by Eclipse as an important productivity tool that lets users view interactive tutorials from within the Eclipse Workbench [46]. We have extended the Cheat sheets plug-in using the Eclipse's plug-in mechanism to integrate and support ontology queries.

Task and Task-Steps can be made optional or non-optional depending on the needs of the process to be defined. After completing the process definition, the information can be stored centrally on the server (as an Eclipse XML Cheat Sheet). Figure 10 shows the adopted cheat sheet editor.
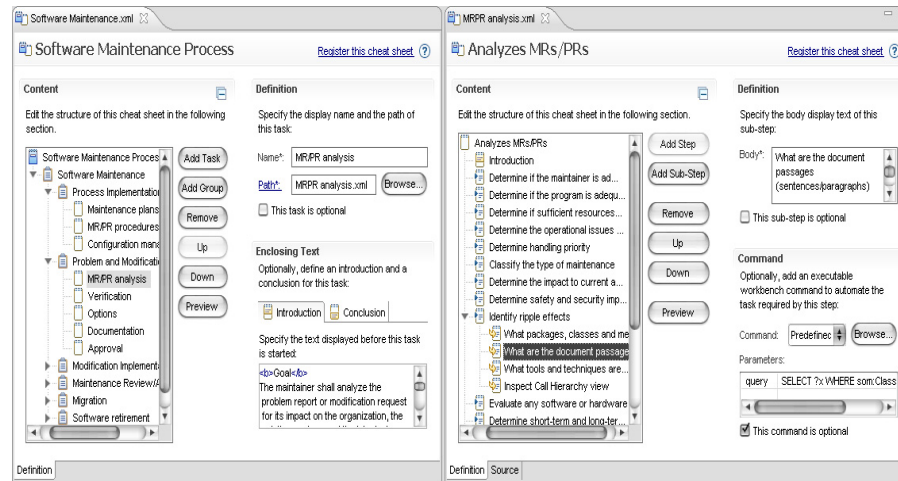
Fig. 10 Creating and managing processes

Eclipse commands: By enriching the process definition with commands that directly interact with the Eclipse IDE, some activities can be further explained or even enforced. An example for such Eclipse commands could be to enforce specific refactoring methods of the source code after a modification request was encoded, e.g., "Encapsulate Field" or "Extract Method" (Figure 11).

Ontology queries: Query templates are managed centrally by the server and can be integrated into cheat sheets as Eclipse commands. Figure 11 shows the dialog to assign a query to a task or task-step. For example, given is the following query to support the initial impact analysis in the IEEE process (*Activity 5.2: Problem and Modification Analysis*):

Given is a modification involving a modification to a method implementation, which requirement documents would be affected by a change and who has the most expertise with making this change?

The query uses the current method name given in the editing window as an input parameter and queries across the source code (the method, its class, and parameters), document (connections between source code elements and requirement documents) and bug tracking (e.g., users who have previously worked on the same method) ontologies.
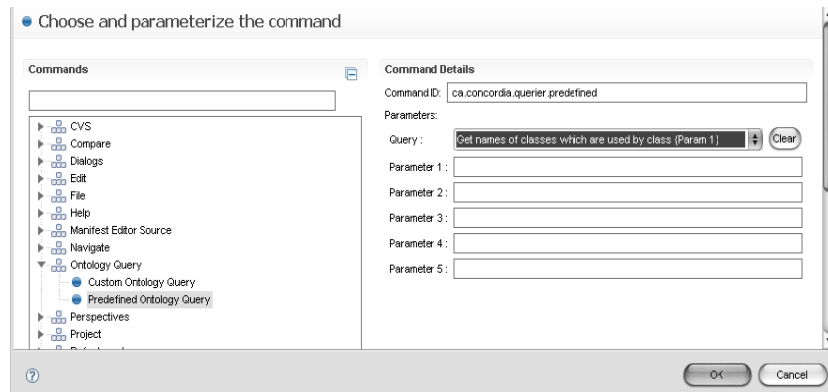
Fig. 11

Query management

Maintainers include the existing process templates in their own IDE by loading the process definition file from the server. They are provided with a simple and integrated textual description of the process (shown in Figure 12), providing them with step by step guidance through the pre-defined process. Contextual information is gathered while they are working and is used to complete query parameters or commands available at the current process step. Such contextual information might be extracted from the current method field being analyzed or edited by the maintainer. Ontology queries themselves are executed by the web server and results are shown in a web browser (integrated as part of the Eclipse IDE). Additional ontology exploration can be performed by selecting parts of the query results to return additional properties related them.
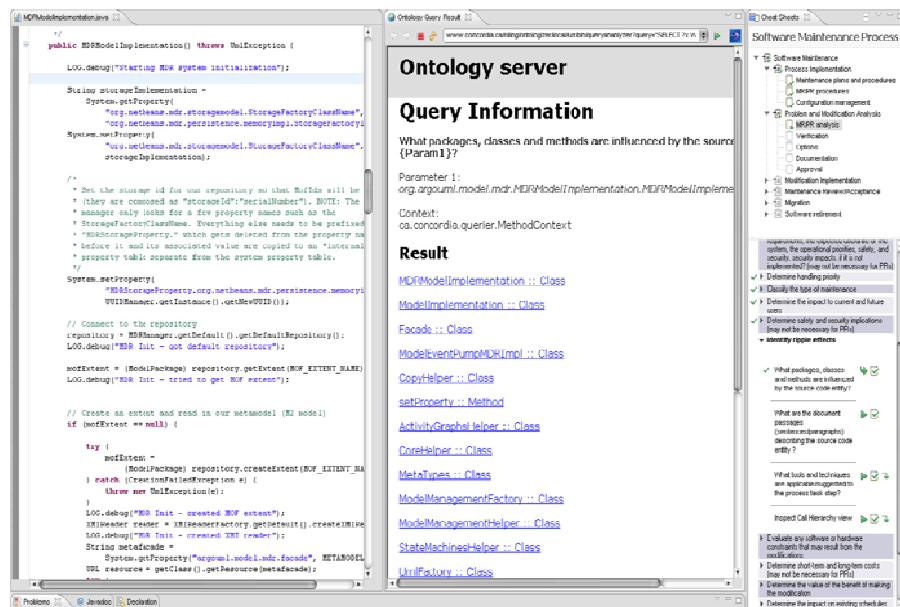

Fig. 12 Ambient Software Evolution environment

The process state is stored within the Eclipse project information, to allow for interruption and continuation of processes. Modifications to the knowledge (such as changed source code) are automatically transferred to the server which incrementally updates the ontology.

In what follows, we introduce different context sensitivity levels that are supported through customizable views through pre-defined queries that establish the link to the KB. For a specific task-step, we also support context switching to the advisor views that provide suggestions on what relevant knowledge resources and other related task-steps might be applicable in the given context.

5.2. *Establishing Contextual Guidance*

We adopt the following informal definition of context awareness in our research. Context-aware refers to the capability of an environment to identify the knowledge resources relevant to the interactions among user, process, and resources in a given task setting. We consider a process model to be task context-sensitive if it provides guidance to a user's state while performing a particular task. A task context can be established by constraining the scope of available information and by inferring additional knowledge from an existing KB. We introduce three levels of context sensitivity, referred to as context level 0-2 (Table 1). These levels are used to define the scope and knowledge inference applied at each level. The context levels range from level 0 (all available information resources are considered without any constraints or additional knowledge inference), to level 2 (not only does it constrain the available resources to the specific task, organization and user, but it also considers historical data and infers additional knowledge from the KB to support this context level).

Table 1 Context level description

Context	Characteristics
Level 0	Main focus is on instance retrieval. No or only very limited use of constraints and reasoning services is made.
Level 1	Constraints are added by considering direct and indirect dependencies among sub-ontologies and their concepts. Additional use of constraints, parameters, and reasoning services.
Level 2	Provides an additional refinement to the context level 1 by using historical data collected from previous tasks as well as by applying data mining and other analysis techniques.

In what follows, we introduce a use case to describe in more detail these different contextual levels.

Use Case: "Identify the elements to be modified in the existing system" [14].

This use case corresponds directly to the IEEE maintenance standard task-step 5.3.2.1a. During the modification implementation phase, maintainers will perform impact

analysis to determine the affected parts of a system. Maintainers will have to decide on resources such as analysis techniques, tools, and artifacts that are available or should be utilized to perform the analysis.

5.2.1. *Context Level 0*

At this context level, the environment would not define any specific constraints with respect to the scope of the knowledge retrieval. The information retrieved will focus mainly on instance retrieval. The following SPARQL query example provides contextual support at level 0 for the defined use case.

Query: Show all programmers and maintainers working on the current project?

```
PREFIX vom: <http://ca.concordia.ontoweb/son>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
SELECT DISTINCT ?user
WHERE {
?R vom:author ?user.
}
```

The query returns all distinct programmers/maintainers who have previously worked on the current project. This query provides knowledge about users who have previous project related knowledge and might be good candidates for the current maintenance task.

5.2.2. *Context Level 1*

From an organizational viewpoint, one of the major challenges in adopting a process model is the need to adopt and integrate the new process within an existing organizational structure. Processes that will allow/support the reuse of existing resources and infrastructure can ease both the integration and acceptance of such a new process.

This context level constrains the query results with respect to a user's expertise level, the specific task setting (MR), and available resources within an organization by including additional constraints and dependencies.

Query: Show all classes affected by changes in method {Parameter %1}

```
PREFIX vom: < http://ca.concordia.ontoweb/son>
SELECT DISTINCT ?class
WHERE {
{?method vom:invokes %1.
 ?method vom:isMethodOf ?class}
UNION
{?method vom:isInvokedBy %1.
 ?method vom:isMethodOf ?class}
}
```

In this example, we constraint the query results through the parameter to the specific context the user is working on. The query parameter is provided either through the user or

by the system itself (e.g., current editing context in the source code). The query returns all classes that are potentially affected by a modification in the method name provided by the parameter.

5.2.3. *Context Level 2*

Common to most process models is the need to collect and analyze previous process-related data to improve and optimize the process. Within our unified process model, we also collect such data, namely the ones related to the resources utilized to complete a modification request (MR) and the applicability of these resources in completed MRs. This historical data enriches our knowledge base and allows for a further refinement of the context according to similarities between the current MR and previously performed MRs. At this level, simple statistical processing, data mining techniques, or other advanced analysis techniques can be applied to enrich the results obtained by queries at this context level.

Query: Show all classes affected by changes in method {Parameter %1}?

```
PREFIX sem: < http://ca.concordia.ontoweb/son>
SELECT DISTINCT ?class
WHERE {
{ ?class sem:isSubclassOf %1 }
UNION
{ ?class sem:hasSubclass %1}
UNION
{?method sem:isMethodOf %1.
 ?method1 sem:invokes ?method.
 ?method1 sem:isMethodOf ?class}
UNION
{?method sem:isMethodOf %1.
 ?method1 sem:isInvokedBy ?method.
 ?method1 sem:isMethodOf ?class}
UNION{
?attr sem:isAttributeOf %1.
?attr sem:hasDeclaredClass ?class}
UNION{
?method sem:isMethodOf %1.
?LV sem:isLocalVariableOf ?method.
?LV sem:hasDeclaredClass ?class}
UNION{
?method sem:isMethodOf %1.
?FP sem:isFormalParameterOf ?method.
?FP sem:hasDeclaredClass ?class}
UNION{
```

```
?attr1 sem:hasDeclaredClass %1.
?attr1 sem:isAttributeOf ?class.}
UNION{
?LV1 sem:hasDeclaredClass %1.
?LV1 sem:isLocalVariableOf ?class.}
UNION{
?FP1 sem:hasDeclaredClass %1.
?FP1 sem:isFormalParameterOf ?class.}
}
```

This query performs an impact analysis for the class specified in *Parameter %1*. It takes advantage of transitive closure relations to identify also the indirect call dependencies that exist among the class *Parameter %1* and the rest of the system.

6. Case study

As discussed throughout the article, software evolution requires the integration of knowledge from all aspects of the software life cycle, through the use of Semantic Web technologies. In order to validate our approach, we have performed two successive case studies in a graduate software maintenance course setting, with a total of 18 students participating in each study. Student expertise varied significantly, from recent graduate students with no or very limited industrial experience, to students with several years of industrial programming/software maintenance experience.

The course material covered in the course was based on the IEEE standard [14], including the major activities specified by the standard's document, as well as techniques and tools supporting the process. For the case study, students had to apply and follow the IEEE maintenance process standard workflow on two medium-size open source software systems, developed in Java. The first application, Debrief[5], is an analysis and reporting software for tracking maritime exercises. The second application, Lobo[6], is a Java-based Web browser. For each application, we defined two maintenance tasks. The first task category involved a component substitution that required the replacement of an existing component with a functionally equivalent one. The second task category corresponded to a typical corrective maintenance activity, i.e., fixing a bug (which was seeded in our case). Each student was randomly assigned two maintenance tasks. Table 2 provides a more detailed description of the tasks performed.

---

[5]   http://www.debrief.info/index.php
[6]   http://www.lobobrowser.org

Table 2 Case Studies

Software System	Maintenance Task	Description	Number of Participants in the Case Study	
			#1	#2
Debrief	Component Substitution	Non-encrypted XML files component had to be substituted with a component that implements XML encryption.	5	4
	Bug Fixing	A seeded error in Debrief affecting the saving of plot data had to be corrected.	4	5
Lobo	Component Substitution	The standard Java SSL libraries to communicate with secure websites had to be replaced with a more secure SSL component.	5	4
	Bug Fixing	A bug in one of the menu options resulting in undesired system behavior had to be corrected.	4	5

6.1. *Case study settings*

The two case studies mainly differ in the amount of knowledge and level of detail made available through our environment. The tool support for the first case study can be best described by its limited KB and as a result, limited process/context awareness. For this case study, we only modeled and populated the following sub-ontologies:

- The process ontology was populated with a concrete instance of the IEEE maintenance standard and its activities [14].
- We also populated the tools and techniques sub-ontologies to provide users with some guidance in selecting applicable tools/techniques for the different maintenance process activities and tasks specified in the process model. For this reason, we introduced pre-defined queries at the context level 0 that focused mainly on instance retrieval from the KB.

For the second case study, we significantly enriched the KB with the following additional information:

- We added a source code ontology to the KB and populated it with instances from the Debrief and Lobo source code.
- We made historical data collected during the first study available. This historical information included, among others, tools and techniques utilized by the participants during the first case study, CVS information with respect to source code modification, and who performed the actual change.
- Additional pre-defined queries were created to provide not only contextual views across the different sub-ontologies, but also to use the now available historical data and additional reasoning services to support contextual queries at different context levels (0-2).

Initial tool usage was recommended to the users, but they were under no obligation to use the tool throughout the case studies.

6.2. *Results and Discussion*

Data collection for the evaluation was performed through a log file created automatically by our environment, capturing the process activities, task steps and queries performed, as well as through an online survey collecting more specific user feedback.

Tool acceptance: As mentioned previously, students were only encouraged to use our environment throughout the case study if they felt the environment would provide them with some added benefit in completing their software evolution tasks. The acceptance rate (Figure 13) corresponds to users who utilized our environment throughout a complete case study. A comparison of the results from the first and second case study shows that the acceptance for our environment increased from 71% to 94% in the second case study.
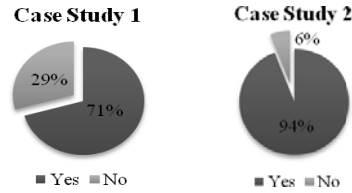

Fig. 13 Overall user acceptance rate

We further analyzed the collected feedback to identify which factors in particular contributed to the improved tool acceptance. From the analysis, we identified two major factors, perceived tool usefulness and the availability of additional contextual queries. The perceived usability/helpfulness of the tool reached 82% (Figure 14). Almost twice as many students found the tool environment useful during the second case study. With respect to the contextual queries, 65% (compared to the first case study) of students indicate that the contextual queries and the additional knowledge provided by these queries helped them during the completion of their assigned maintenance task.
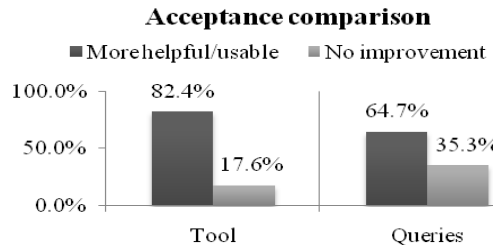

Fig.14 Comparison of tool and query applicability

This initial case study shows that through the use of the unified ontological representation and pre-defined queries, we were able to provide users with a context-sensitive ubiquitous knowledge environment by hiding language specific aspects of the underlying knowledge resources. We also observed that providing additional contextual queries does

not only improve the acceptance of the tool, but also enhances the use and perceived benefits of knowledge provided by the environment.

Given the results and feedbacks collected from the second case study, we are currently in the process of further extending and refining our ontological model to further enhance its ability to support additional tasks and context specific queries. The two case studies provided us with a proof of concept that our approach is capable of providing additional benefits to maintainers. It was in particular the integration of process, task and resource relevant information that was well received. Furthermore, we demonstrated the use of automatic ontology population to provide a populated knowledge base to be able to deliver context-sensitive guidance to maintainers during typical maintenance tasks. However, given the complexity of the software evolution domain (including the variations of artifacts, knowledge resources, user expertise, etc) er contexts, etc., additional larger user studies are needed to further evaluate our approach.

7. Related Work

Most of the existing work on ambient software development [47, 48, 49] has focused on defining what constitutes an ambient environment and formalizing the services provided by such environments [48, 49, 50]. Visualization aspects of ambient software development were addressed for example in [52], in which the visualization of software artifacts from the perspective of supporting teams in coordinating efforts are discussed. The development of environments that hide external resources (e.g., sensors) from end-users by allowing them to be immersed without having direct exposure to the sensor data was presented in [50, 51].

In our approach, the focus is on a particular user group, i.e., software maintainers, and the formal model required for supporting their work through context-sensitive, process-oriented guidance. It is ambient in the sense that we integrate numerous sources to semi-automatically build the knowledge base surrounding a (physical or virtual) group of developers. We see the combination of our KB with "hard" sensors providing further feedback on the actions of a human user or group of users as a promising extension of our work.

As a knowledge representation language, OWL has already been applied in many applications of the software engineering domain, such as model-driven software development [44], a CMMI-SW model representation and reasoning for classifying organizational maturity levels [53], reverse engineering tool integration [15], component reuse [54] and open source bug tracking [55]. However, there exists only limited research in modeling software maintenance and evolution using ontologies. Ruiz et al. [37] present a semi-formal ontology based on REFSENO for managing software maintenance projects. They consider both the static and dynamic aspects such as workflow in software maintenance processes. The ontology was constructed using KM-MANTIS, a knowledge management system with the goal of improving the management of maintenance projects. However, no implementation or usage details are provided. Kitchenham et al. [36] designed a UML-based ontology for software maintenance to identify and model factors that affect the results of empirical studies. Their goal differs from ours by utilizing ontologies to

establish a common understanding context for empirical studies. Furthermore, the resulting ontology was not formally modeled and no reasoning services were used to infer implicit knowledge. Dias et al. [18] extended the work by Kitchenham by applying a first order logic to formalize knowledge involved in software maintenance. Although they stated that it is worthwhile to provide a knowledge base, they only identified knowledge relevant to the software maintenance domain without actually providing a concrete KB implementation. González-Pérez and Henderson-Sellers present a comprehensive ontology for software development [56] that includes a process sub-ontology modeling, among others, techniques, tasks, and workflows. The ontology is presented using UML diagrams. No implementation is given, implying that the authors also have not examined the integration of their ontology in an actual software development process.

There exists relevant work on conceptualizing the software engineering domain to support teaching of software engineering, including establishing a common software engineering terminology [40, 57] or defining common concepts found in process models [14]. Petrenco et al. [58] used open source software systems in teaching software evolution. Their experience showed that by integrating CVS into course projects and by collecting feedback, it was possible to enhance the assessment of student performance. Falbo et al. [59] reported on shared conceptualizations for integrated tool development, and Deridder et al. [60] have used ontologies for linking artifacts at several phases of the development process. The SWEBOK project [40] applies ontologies in SE to provide pointers to relevant literature on each of its concepts. Current Web–based learning approaches based on concept of *learning object* [61] focus on reusability in their content design.

Common to all of these approaches is that their main intent is to support in one form or another the conceptualization of knowledge, mainly by standardizing the terminology and to support knowledge sharing based on a common understanding. These approaches typically fall short on adopting and formalizing a process model that supports connecting knowledge resources in the KB with process activities. They also lack the use of querying and reasoning services to infer implicit knowledge that provides contextual guidance. Compared to these approaches, our work represents a first step towards the notion of "Web 3.0" [62]. We do not only support the formal conceptualization and integration of knowledge, but also the ability to query and to infer knowledge from the existing KB by utilizing reasoning services.

The collaborative nature of software engineering has more recently been addressed by introducing Wiki systems into the SE process. Semantic Wiki extensions like Semantic MediaWiki [63] or IkeWiki [64] add formal structuring and querying extensions based on RDF/OWL metadata. This work can be seen as complementary to our approach, in that they can support the creation and visualization of the developed knowledge base. However, by themselves they do not address the main concern covered by our approach, i.e, delivering active and context-sensitive guidance to an individual developer for his current task, based on knowledge both explicitly encoded and implicitly derived by actions performed by other developers.

Compared to other modeling approaches such as Model Driven Architecture (MDA) and related modeling standards (e.g., UML/MOF [65] or the Eclipse Modeling Framework (EMF) [46]) our approach differs fundamentally in its objective. (1) These model-

ing approaches focus on forward engineering, i.e., produce code from abstract, human-elaborated specifications, following a strict modeling process. We, on the other hand, focus on software evolution, where one has to extract, model, integrate and utilize knowledge from various and often not well defined resources. (2) MDA and EMF both focus on modeling well defined processes, in an attempt to derive complete models. Our approach on the other hand is based on an open world assumption, focusing on the integration of existing and newly gained knowledge and by making this knowledge available to the end-user.

With regard to tool support for process guidance, recent work by IBM Rational on its Method Composer (RMC) [17] and Process Advisor [9] are the closest related approaches to our research. RMC is a process management tool for authoring, configuring, and publishing development process like the Rational Unified Process (RUP). The IBM Rational Process Advisor integrates RUP process guidance within the context of software development tools [9]. Due to the required tool customization, the contextual guidance through the Process Advisor is currently only supported within a selected set of Rational tools. Our approach differs in its overall motivation as we focus on knowledge integration rather than tool integration, by providing a flexible and dynamic knowledge management that allows newly gained knowledge (process and/or resources) to become an integrated part of our model. Furthermore, we support reasoning services that can infer implicit knowledge not explicitly modeled in the KB to provide contextual guidance.

8. Conclusions and Future Work

Software evolution is a major part in terms of effort and cost involved in any software life cycle. Software maintainers have an ever-growing number of tools and resources at their disposal, but this also results in a form of information and tool overload, as they are all "passive" in today's environments. The user has to manually identify, retrieve, and activate the tools with the proper resources before they can help him in the completion of a given maintenance task. In addition, implicit knowledge is often lost, since it is not captured by today's environments. A similar task might have been successfully performed by a different developer using a tool the user is unaware of, but if this knowledge is not captured, stored, and delivered in a context-sensitive manner to the end user, it cannot serve as a "corporate memory" for future problem-solving. Our research evolves common software engineering environments towards ambient, knowledge-managing platforms. These integrate existing tools and resources using ontological representation and reasoning models, enabling them for the first time to deliver context-sensitive guidance, based on a formal model of existing maintenance process standards.

Our work promotes the use of formal ontology formats and automated reasoning in software evolution. We demonstrate their applicability by providing a formal OWL-DL-based ontological representation that models both the software evolution processes and the resources relevant to support them. A contemporary, industry-relevant IEEE standard for software maintenance has been successfully integrated to model the software evolution processes. The resulting, unified ontological representation allows for the integration of different knowledge sources. The flexibility and extensibility provided by Web ontol-

ogies also enable the evolution and enrichment of the knowledge base. A case study demonstrates the user acceptance and usefulness of our ideas and their initial implementation.

As future work, we aim to integrate additional clients and resources into our environment, such as mailing list and blogs, to extend the scope of collecting and exchanging data, thereby enhancing the guidance. We also plan to investigate the use of other queries languages like iSPARQL[7] or SPARQL-DL that might be better suited for our application purpose. Additionally, we plan to use rule languages like SWRL[8] to support more complex queries. Also, more user studies involving a larger number of human subjects are required to be able to claim the significance of the results.

Acknowledgements

References

[1] Bohner, S. A. and Arnold, R. S.: An Introduction to Software Change Impact Analysis. Software Change Impact Analysis, pp. 1–26, 1996.

[2] Breitman, K., and Leite, J.: Scenario Evolution: A Closer View on Relationships. Proc.of the 4th Intl Conf.on Requirements Eng. (ICRE 2000), pp. 95-105,

[3] Basili, V.R., "*Viewing Maintenance as Reuse Oriented Software Development*", IEEE Software, pp.19-25, 1990.

[4] Mayhauser A.V., Vans A.M., "*Program Comprehension during Software Maintenance and Evolution*", IEEE Computer, pp44-55, Aug.1995.

[5] M. M. Lehman , L. A. Belady, Program evolution: processes of software change, Academic Press Professional, Inc., San Diego, CA, 1985

[6] Storey, M.-A. D., Sim, S.E., and Wong K., "*A Collaborative Demonstration of Reverse Engineering Tools*", ACM SIGAPP Applied Computing Review, Vol. 10, Issue 1, 18-25, 2002.

[7] M.-A. D. Storey , K. Wong , H. A. Müller, How do program understanding tools affect how programmers understand programs?, Science of Computer Programming, v.36 n.2-3, p.183-207, March 2000.

[8] Brooks, R., "*Towards a theory of the comprehension of computer programs*", International Journal of Man-Machine Studies, 18:543-554, 1983.

[9] Smith, J., Popescu, D., and Bencomo, A., "*IBM Rational Process Advisor: Integrating the Software Development Process with IBM Rational Developer and Tester V7 tools*", IBM report, December, 2006.

[10] K. Bontcheva, M. Sabou, Learning Ontologies from Software Artifacts: Exploring and Combining Multiple Sources, In *Proceedings of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006).*

[11] Berners-Lee T., Hendler J., and Lassila O., "*The Semantic Web*", Scientific American, 284 (5):34-43, 2001.

[12] Bennett, K.H. and Rajlich, V.T., "*Software Maintenance and Evolution: a Roadmap*", Proceedings. of the Conference on the Future of Software, 73-87, 2000.

[13] *CMMI for Development*. Version 1.2, Technical Report CMU/SEI-2006-TR-008, Carnegie Mellon, Software Engineering Institute, USA, 2006.

---

[7] http://www.ifi.uzh.ch/ddis/isparql.html

[8] http://www.w3.org/Submission/SWRL/

[14] International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering, Software Life Cycle Processes, Maintenance, ISBN: 0-7381-4961-6, 2006.

[15] Jin, D. and Cordy, J. R., "Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology", 21st IEEE International Conference on Software Engineering (ICSM), 2005.

[16] Sommerville, I.: Software Engineering. 6th edn. Addison-Wesley (2000)

[17] Seacord, R., Plakosh, D., Lewis, G.: Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. SEI Series in SE. Addison-Wesley (2003).

[18] Dias, M. G. B., Anquetil, N., and Oliveira, K. M. d.: Organizing the Knowledge Used in Software Maintenance, J.of Universal C.S., 9(7), pp. 641-658, 2003.

[19] Jens H. Jahnke and Andrew Walenstein "Reverse Engineering Tools as Media for Imperfect Knowledge,", IEEE Working Conference on Reverse Engineering, (WCRE'2000), pp 22-31.

[20] Kroll, P., Kruchten, P.,*"The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP"*,Addison Wesley, 2003.

[21] Aversno, L., Canfora, G., Lucia, A.D. and Stefanucci, S.: Automating the Management of Software Maintenance Workflows in a Large Software Enterprise: a Case Study", Journal of software maintenance and evolution: research and practice, 2002:14:229-255

[22] Storey, M.-A.: Theories, Methods and Tools in Program Comprehension: Past, Present, and Future. 13th IWPC, pp.181-191, 2005.

[23] J. Ratzinger, T. Sigmund, P. Vorburger, H. Gall, "Mining Software Evolution to Predict Refactoring", *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, IEEE Computer Society, p. 10

[24] P. Arkley, P. Mason, and S. Riddle, "Position Paper: Enabling Traceability," Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, Scotland (September 2002), pp. 61–65.

[25] A. Marcus, J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing". In Proceedings of 25th International Conference on Software Engineering, 2002.

[26] Yonggang Zhang, René Witte, Juergen Rilling, and Volker Haarslev. An Ontology-based Approach for the Recovery of Traceability Links. In: 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006). October 1st, Genoa, Italy. 201

[27] René Witte, Qiangqiang Li, Yonggang Zhang, and Juergen Rilling. Text Mining and Software Engineering: An Integrated Source Code and Document Analysis Approach. IET Software Journal, Special Section on Natural Language in Software Development, 2008.

[28] Wen Jun Meng, Juergen Rilling, Yonggang Zhang, René Witte, and Philippe Charland. An Ontological Software Comprehension Process Model. In: 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006). October 1st, Genoa, Italy

[29] Gruber, T. R.: A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5(2):199-220, 1993.

[30] H. Wache, T. Vgele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. H ubner. "Ontologybased integration of information - a survey of existing approaches", IJCAI-01 Workshop: Ontologies and Information Sharing, pp 108--117, 2001.

[31] F. Baader et al., "The Description Logic Handbook", Cambridge Univ. Press, 2003.

[32] Baader F., Horrocks I., and Sattler U.: Description logics as ontology languages for the semantic web. In Dieter Hutter and Werner Stephan, editors, number 2605 in Lecture Notes in Artificial Intelligence, pages 228-248. Springer, 2005.

[33] Web Ontology Language, http://www.w3.org/2004/OWL/, (accessed: June, 2007).

[34] OWL Web Ontology Language Reference, W3C Recommendation, http://www.w3.org/TR/owl-ref (accessed June 2007).

[35] Haarslev, V. and Möller, R., *"RACER System Description"*, In Proc. of Int. Joint Conference on Automated Reasoning (IJCAR'2001), Springer-Verlag, 701-705, 2001.

[36] Kitchenham, B., Travassos, G.H., Mayrhauser, A.V., Niessink, F., Schneidewind, N.F., Singer, J., Takada S., Vehvilainen R., and Yang H., *"Towards an Ontology of Software Maintenance"*, Journal of Software Maintenance and Practice, 11(6), 365-389, 1999.

[37] Ruiz, F., Vizcaíno, A., Piattini, M., and García F., *"An Ontology for the Management of Software Maintenance Projects"*, International Journal of Software Engineering and Knowledge Engineering, 14(3), 323-349, 2004.

[38] P. Wongthongtham, E. Chang, T.S. Dillon, I. Sommerville (2007) 'Software Engineering Ontology – Instance Knowledge Part I', International Journal of Computer Science and Network Security, USA.

[39] C. Alchourron, P. Gärdenfors, D. Makinson. On the Logic of Theory Change: Partial Meet Contraction and Revision Functions. Journal of Symbolic Logic, 50:510-530, 1985.

[40] Software Engineering Coordinating Committee. SWEBOK, version 2004, http://www.swebok.org.

[41] R. Pressman, " Software Engineering: A Practioner's Approach, 6th Edition, McGraw Hill College, ISBN-10:007301933X, 2005

[42] C. Kiefer, A. Bernstein, J. Tappolet, "Mining Software Repositories with iSPARQL and a Software Evolution Ontology", Proceedings of the 29th International Conference on Software Engineering Workshops (ICSEW),2007, pp.10

[43] René Witte, Qiangqiang Li, Yonggang Zhang, and Juergen Rilling. Ontological Text Mining of Software Documents. 12th International Conference on Applications of Natural Language to Information Systems (NLDB 2007), June 27-29, 2007, CNAM, Paris, France. Springer LNCS 4592, pp.168-180.

[44] Ontological Driven Architectures and Potential Uses of the Semantic Web in Systems and SE, www.w3.org/2001/sw/BestPractices/SE/ODA/, (accessed Feb, 2008).

[45] Ballim, A. and Wilks, Y., "*Artificial Believers: The Ascription of Belief*", Lawrence Erlbaum, 1991.

[46] Eclipse and Open Development Platform, www.eclipse.org (last accessed 2/2008)

[47] Emile Aarts and José Encarnação, True Visions: The Emergence of Ambient Intelligence, Springer, 2006

[48] José Encarnação (Ed.) et al.: Ambient Intelligence - The New Paradigm for Computer Science and for Information Technolgy. Special Topic of Journal it - Information Technology, Oldenbourg Verlag, Munich. Vol.50(2008) Issue 1.

[49] Giuseppe Riva et al. Ambient Intelligence: The Evolution of Technology, Communication and Cognition towards the Future of Human-Computer Interaction, IOS, Amsterdam, The Netherlands, 2005

[50] Adam Greenfield, "Everyware: The Dawning Age of Ubiquitous Computing", New Riders Publishing, 2006

[51] Wright, David, Serge Gutwirth, Michael Friedewald et al., Safeguards in a World of Ambient Intelligence, Springer, Dordrecht, 2008.

[52] Parnin, C. Gorg, C." Design Guidelines for Ambient Software Visua-lization in the Workplace", Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on Publication Date: 24-25 June 2007

[53] Soydan, G. H. and Kokar, M., "An OWL Ontology for Representing the CMMI-SW Model", Workshop on Semantic Web Enabled Software Engineering (SWESE), 2006.

[54] Happel, H.-J., Korthaus, A., Seedorf, S., and P.Tomczyk, *"KOntoR: An Ontology-enabled Approach to Software Reuse"*, 18th International. Conference on Software Engineering and Knowledge Engineering (SEKE), San Francisco, July, 2006.

[55] Ankolekar, A., "*Supporting Online Problem –Solving Communities with the Semantic Web*", Ph.D.Thesis, Carnegie Mellon University, Pittsburgh, PA, 2005.

[56] C. Gonzalez-Perez, B. Henderson-Sellers,"Modelling software development methodologies: A conceptual foundation", Journal of Systems and Software 80(11): 1778-1796, 2007

[57] Althoff, K.-D., Birk, A., Hartkopf, S., M¨uller, W., Nick, M., Surmann, D. and Tautz, C. (2000). Systematic Population, Utilization, and Maintenance of a Repository for Comprehensive Reuse. In G. Ruhe and F. Bomarius eds.), *Learning Software Organizations - Methodology and Applications*, Springer Verlag, Lecture Notes in Computer Science, LNCS 1756, 25-50.

[58] M. Petrenko, D. Poshyvanyk, V. Rajlich, J. Buchta, "Teaching Software Evolution in Open Source," Computer, vol. 40, no. 11, pp. 25-31, Nov., 2007.

[59] Falbo, R.A., Natali, A. C. C., Mian, P. G., Bertollo, G., Ruy, F. B. (2003). ODE: Ontology-based software Development Environment. In Proceedings of the "IX Congreso Argentino de Ciencias de la Computaci´on", 1124-1135.

[60] Deridder, D., Wouters, B., Lybaert, W. (2000). The Use of an Ontology to Support a Coupling between Software Models and Implementation. In Proc. of the Int. Workshop on Model Eng., 14th European Conf. on Object-Oriented Prog. (ECOOP), 2000.

[61] Polsani, P. R. (2003). Use and Abuse of Reusable Learning Objects. *Journal of Digital information*, 3(4).

[62] http://www.w3.org/2007/Talks/0123-sb-W3CEmergingTech/Overviewp.pdf (last accessed Dec. 2007)

[63] M. Krötzsch, D. Vrandečić, and M. Völkel. Semantic MediaWiki. In I. Cruz, S. Decker, D. Allemang, C. Preist,D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, The Semantic Web – ISWC 2006, volume 4273 of LNCS, pages 935–942. Springer, 2006.

[64] S. Schaffert. IkeWiki: A Semantic Wiki for Collaborative Knowledge Management. In WE-TICE, pages 388–396. IEEE Computer Society, 2006.

[65] Open Management Group (OMG) – Unified Modeling Language, www.uml.org (last accessed 2/2008)

[66] T. Meyer, K. Lee, R. Booth. Knowledge Integration for Description Logics. In Proceedings of the 7th International Symposium on Logical Formalizations of Commonsense Reasoning, 2005.

[67] Letovsky, S. (1986). *Cognitive Processes in Program Comprehension*. Ablex Publishing Corp.

[68] Brooks, F. P. (1987). No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer,* 20(4), pp. 10-20.

[69] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns : elements of reusable object-oriented software. Reading, MA, Addison Wesley Professional.

[70] Rajlich, V., & Wilde, N. (2002). The Role of Concepts in Program Comprehension, Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC'02), pp. 271-278.

[71] Koenemann, J., & Robertson, S. (1991). Expert Problem Solving Strategies for Program Comprehension, Human Factors in Computing Systems, ACM Press.

[72] P. Gärdenfors. Belief Revision: An Introduction. In P. Gärdenfors (ed), Belief Revision, pp. 1-20, Cambridge University Press, 1992.

[73] P. Gärdenfors. The Dynamics of Belief Systems: Foundations vs Coherence Theories. Revue Internationale de Philosophie 44, pp. 24-46. Reprinted in C. Bicchieri, M. L. Dalla Chiara (eds). Knowledge, Belief and Strategic Interaction, pp. 377-396, Cambridge University Press, 1992.

[74] H. Katsuno, A.O. Mendelzon. On the Difference Between Updating a Knowledge Base and Revising It. Technical Report on Knowledge Representation and Reasoning, University of Toronto, Canada, KRR-TR-90-6, 1990.

[75] D. McGuiness, R. Fikes, J. Rice, S. Wilder. An Environment for Merging and Testing Large Ontologies. In Proceedings of the 7 International Conference on Principles of Knowledge

Representation and Reasoning (KR-00), Technical Report KSL-00-16, Knowledge Systems Laboratory, Stanford University, 2000.

[76] S.H. Kang, S.K. Lau. Ontology Revision Using the Concept of Belief Revision. In Proceedings of the 8th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES-04), part III, pp. 8-15, 2004.

[77] K. Lee, T. Meyer. A Classification of Ontology Modification. In Proceedings of the 17th Australian Joint Conference on Artificial Intelligence (AI-04), pp. 248-258, 2004.

[78] P. Liberatore. The Complexity of Iterated Belief Revision. In Proceedings of the 6th International Conference on Database Theory (ICDT-97), pp. 276-290, 1997.

[79] René Witte, "Fuzzy Belief Revision". Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'02), pp. 311-320, April 19-21, 2002, Toulouse, France.

[80] Ralf Krestel, René Witte and Sabine Bergler. Processing of Beliefs extracted from Reported Speech in Newspaper Articles. International Conference on Recent Advances in Natural Language Processing (RANLP 2007), September 27-29, 2007, Borovets, Bulgaria.