Ontologies and Software Engineering

3 authors:

Dragan Gasevic
The University of Edinburgh

428 PUBLICATIONS **4,510** CITATIONS

SEE PROFILE

Nima Kaviani
University of British Columbia - Vancouver

34 PUBLICATIONS **284** CITATIONS

SEE PROFILE

Milan Milanović
University of Belgrade

31 PUBLICATIONS **190** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Closing the Loop View project

Project Supporting higher education to integrate learning analytics View project

Ontologies and Software Engineering

Dragan Gašević[1], Nima Kaviani[2], and Milan Milanović[3]

[1] Athabasca University, Canada, `dgasevic@acm.org`
[2] University of British Columbia, Canada, `nimak@ece.ubc.ca`
[3] University of Belgrade, Serbia, `milan@milanovic.org`

1 Introduction

Fast growth of communication and mobile technologies, constant demands for new services, and increased number of computer users, are some of the key reasons of the constantly increasing need for more software. This naturally requires effective methods for engineering software that will be able to respond adequately to the needs for which the software was built, and yet to allow for higher levels of productivity of software engineers. However, today's state of the art and practice demonstrates that both perspectives are still suffering from serious problems. On one hand, the Standish Group published its well-known Chaos Report in 1994 in which it was noted that only 16% of software projects were successful, 31% were failures, and some 53% were challenged. The 2006 report demonstrates a bit better situation where 35% of software projects were successful, 19% were failures, and 46% were challenged [9]. On the other hand, productivity methods are struggling with new challenges such as better methods for software maintenance (e.g., tracing place in the code when adding new or updating present functionalities to the software [69]) or facilitating collaboration of software teams (e.g., mutual understanding between different parties collaborating in requirement engineering, especially in the context of global software development [18]).

While software is a technical category designed to perform specific tasks by using computer hardware, it is also a social category which nowadays is used in almost every aspect of human's life. In fact, software is a knowledge repository where knowledge is largely related to the application domain, and not to software as an entity [4]. So, we need to be able to share and interoperate (application) knowledge stored in software with the knowledge about all relevant aspects surrounding and influencing software (e.g., domain knowledge, new requirements, policies, and contexts, in which people use and interact with software) in order to get software to the more advanced levels. This knowledge sharing and management requires the use of explicit definition of knowledge, as it is a basic need for machines to be able to interpret knowledge.

This is why the software engineering community has recognized ontologies as a promising way to address current software engineering problems [14, 32].

Researchers have so far proposed many different synergies between software engineering and ontologies. For example, ontologies are proposed to be used in requirement engineering [47], software modeling [45], model transformations [42], software maintenance [43], software comprehension [70], software methodologies [30], and software community of practice [1]. Moreover, software engineering technologies are proposed for modeling and reasoning over ontologies. These synergies between ontologies and software engineering have also attracted attention of standardization bodies and have some on-going activities. Ontology-Driven Architecture (ODA) is an effort of the W3C's Software Engineering Best Practices Working Group that tries to develop best practices for using ontologies in software engineering [66]. Probably, the most important result so far is the Ontology Definition Metamodel (ODM) that is proposed to be the Object Management Group (OMG)'s standard [54]. The ODM standard allows for integrating ontology languages (i.e., ontologies) into the software development process based on model-driven engineering principles [7]. Although all of these different efforts demonstrate many benefits to different aspects of software and ontology engineering or give a nice description of the state of the art in the area [14, 32], none of them analyze and evaluate applications of ontologies in different aspects of software engineering by following a comprehensive software lifecycle framework.

In this chapter, we start from defining software engineering as an application context for ontologies, and proceed to defining a framework that identifies places in software lifecycle where ontologies can contribute to improve the current state of software engineering. We consequently have organized the structure of this chapter to use this framework for analyzing the use of ontologies in different phases of software life cycle. Note that the chapter does not discusses Semantic Web rules (see Chapter 5) or upper layers of the Semantic Web cake, but fully focuses on ontologies in software engineering.

2 Software Engineering

The goal of this section is to define software engineering, describe some typical software lifecycle phases, artifacts used and produced in them, participants, their interactions, and relevant domain and application knowledge. Based on this discussion, we define a unified framework for the use of ontologies in software engineering to which we are going to refer in the rest of the chapter.

The most commonly used definition of software engineering is the one given in the IEEE Standard Glossary for Software Engineering [38], where software engineering is defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software." It is obvious that this definition has a very strong foundation on the lifecycle of software, i.e.

how it is built (i.e., development); how it is used (i.e., operation); and how it is updated, and renewed (i.e., maintenance). Therefore, it is natural to discuss about software engineering by focusing on software lifecycle phases. While different methodologies (e.g., Rational Unified Process (RUP) or adaptive methodologies such as agile development) consider different phases for software lifecycle, we use the phases of software lifecycle as defined in [61] given the dominant use of object-oriented paradigm, while the definition of all stages are based on [38]. In Fig. 1, we give an overview of all software lifecycle phases with their parallel activities; used and produced artifacts; types of interactions and collaborations; and participants and their roles. Each of the software lifecycle phases can be defined as follows [38]:

- **Analysis phase** determines what has to be done in a software system. After determining what kind of software is needed to be developed, the *requirements phase* is the first and the most important step. In this phase, the requirements for a software product are defined and documented. This is usually done in collaboration with end-users and domain experts, where the critical point is to establish common understanding of the domain under study. Once requirements are defined, they are formally specified in the form of a legal document. Typically, this document includes functional requirements, performance requirements, interface requirements, design requirements, and development standards; to eliminate all ambiguousness, incompleteness, and contradictions. Modeling approaches are recommended at this stage (e.g., RUP recommends using UML use cases and

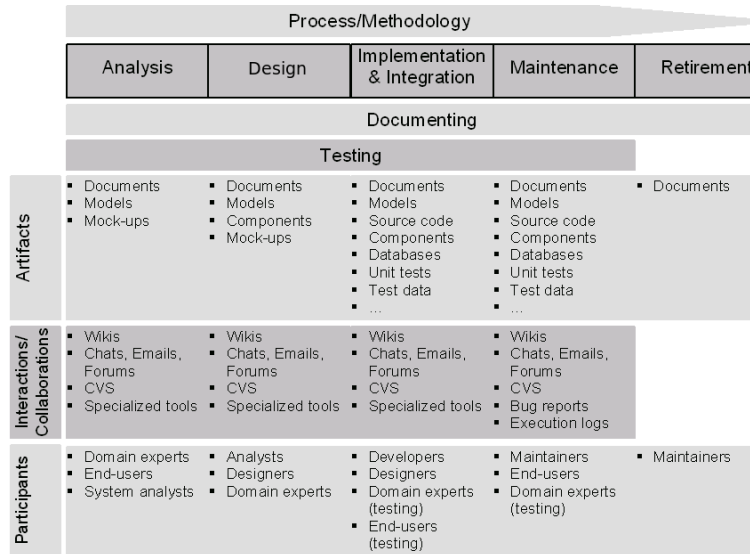| Process/Methodology				
Analysis	Design	Implementation & Integration	Maintenance	Retirement
Documenting				
Testing				
Artifacts				
• Documents • Models • Mock-ups	• Documents • Models • Components • Mock-ups	• Documents • Models • Source code • Components • Databases • Unit tests • Test data • ...	• Documents • Models • Source code • Components • Databases • Unit tests • Test data • ...	• Documents
Interactions/Collaborations				
• Wikis • Chats, Emails, Forums • CVS • Specialized tools	• Wikis • Chats, Emails, Forums • CVS • Specialized tools	• Wikis • Chats, Emails, Forums • CVS • Specialized tools	• Wikis • Chats, Emails, Forums • CVS • Bug reports • Execution logs	
Participants				
• Domain experts • End-users • System analysts	• Analysts • Designers • Domain experts	• Developers • Designers • Domain experts (testing) • End-users (testing)	• Maintainers • End-users • Domain experts (testing)	• Maintainers

Fig. 1. Software development lifecycle: Phases, Artifacts, Interaction and Collaboration, and Participants

class diagrams), while some researchers recommend using even some more formal approaches (e.g., Petri nets [39]).

- **Design phase** defines detailed designs for application domain, architecture, software components, interfaces, and data. Since all the design should be verified and validated to satisfy requirements, usually this phase regards the use of modeling (e.g., UML). The more formal designs are defined, the less potential errors will be, and the more potentials will exist for automatic software implementation (e.g., code generation). Therefore, the software engineering community puts a lot of attention to the discipline called model-driven engineering (MDE) to enable model-driven development (MDD) of software products [24]. Moreover, model transformations (model-to-model; model-to-text; and text-to-model) are the key concepts of MDD which allow for round trip engineering (i.e., forward and reverse engineering) of software.
- **Implementation phase** creates a software product from the design documentation and models. This phase also debugs and documents the software product. This phase assumes the use of programming languages to encode specified designs, and testing techniques (e.g., unit testing) to eliminate any potential bugs. Besides eliminating software bugs, it is also important to be able to check whether implementations are fully valid w.r.t. the models (aka., model-based testing [3]).
- **Integration phase** is the process of combining software components (e.g., Web services), hardware components, or both, into an overall system. This phase is usually done in parallel with the implementation phase. Besides importance of a high-quality and up-to-date documentation, this stage also requires testing, such as acceptance testing (by end-users) and integration testing (i.e., checking the integration with other components).
- **Maintenance phase** is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment, i.e., any change after acceptance of the software by the client. This phase highly depends on the quality of documentation in order to trace parts of software to be changed. Of course, this phase also assumes documenting all changes as well as testing software for its compliance to the initial and newly-defined requirements.
- **Retirement phase** is the period of time in the software life cycle during which support for a software product is stopped. This may happen in cases where a drastic change in design, implementation, or documentation has occurred. This phase also has to be well-documented to explain why a software product is retired.

However, the current software practice suffers from a lack of tractability of all artifacts and elements produced/used in different stages of the lifecycle (e.g., requirement documents and source code), that can substantionally affect software development and especially software maintenance [69]. As already pointed out, software is a socio-technical category which necessitates keeping

track of all relevant human-human and human-software interactions (e.g., chat discussions that may explain why some design decision were made) [1].

It is also very important to mention that every software product strongly relies on the application-specific domain knowledge, standards, and policies related to the software system under study. In addition, every software development process follows some methodologies[4], and it is useful to relate methodology tasks and activities with the software artifacts produced/used in different lifecycle phases. Moreover, each task in the software development lifecycle is important to be assigned to a person (e.g., software programmer) that has competences needed. Very often, such knowledge is not represented explicitly, and thus it is very hard to establish tractability links between such knowledge and produced software artifacts and interactions used in all phases of software lifecycle. Such knowledge can further stimulate social interactions and locate peers that can help in dealing with some specific software development issues. In the rest of the chapter, we present how ontologies can assist in establishing the missing semantic links in the above software lifecycle phases.

3 Analysis

According to the Standish Group report from 1994 [5], the main reasons for software project failures are issues caused by the poor or inappropriate software analysis. The three reasons for software success are user involvement, executive management support, and a clear statement of requirements, while the main reasons for challenged and failed software projects are the lack of user input, incomplete requirements and specifications; and changing requirements and specifications. All these reasons stress the need for mutual understanding between requirement engineers and end-users and the importance of the preciseness of the requirement specification.

3.1 Ontologies as Requirement Engineering Products

The above arguments motivated researchers to look at ontologies as a solution to improve the state of the art in this area. Breitman & Leite argue that ontologies should be sub-products of the requirement engineering phase [10]. It follows the idea of Hendler that on the web we can have many application-oriented ontologies that should be interconnected to facilitate knowledge sharing between different applications [34]. Thus, their requirement engineering process has a particular sub-process for ontology construction. This process is inspired by the layered ontology engineering approach [49], where the main source for creating ontologies is the language's extended lexicon. The lexicon is built by eliciting the important terms from the relevant source documents,

---

[4] Todays methodologies follow incremental and iterative software development.
[5] http://www.spinroot.com/spin/Doc/course/Standish_Survey.htm

and mapping the terms to the appropriate constructs (e.g., classes) of the ontology language used in the application under study. Looking further to some other ontology development technologies, we can also find out that requirement engineering and ontology engineering are even sharing some common methodologies. For example, the DOGMA ontology engineering framework [65] uses a scenario-based approach to engineer ontologies for application domains. The most important thing from both cases is that the ontology is a product of the analysis phase, which means that all parties involved in this process should agree upon the ontology developed. This, in fact, should eliminate the lack of misunderstanding of the users' needs and should further be propagated to the design phase (e.g., by transforming such an ontology to models - cf. Sec. 4). Another benefit is that all documents (e.g., stories) that are used for requirement acquisition could be semantically annotated with the ontologies created from them to represent intelligent content [13]. If such ontologies are further used in the design phase (e.g., models), we can then have traceability between these two software development stages (i.e., analysis and design) and establish mapping relations with other ontologies to provide traceability with other potentially relevant sources of knowledge.

The use of upper-level ontologies is also well-known in software engineering when developing domain models that are usually part of the requirement specification. Typically, an upper-level ontology (e.g., Bunge-Wand-Weber (BWW) model) is used as a definition of the background theory (or the perspective to the world) based on which the domain model is built. Current software development methodologies (e.g, RUP) suggest UML-based domain models as the results of the analysis phase. The current experience demonstrates that if one wants to make such domain models valid w.r.t. the upper level ontology, then a modeling language should be constrained in order to allow the use of only those models that are compliant to the upper ontology. For example, Evermann & Wand [23] constrain the specification of the UML (i.e., UML metamodel) by using the Object Constraint Language (OCL), so that every UML model is fully compliant with the BWW model.

3.2 Requirement Engineering Approaches

Requirement engineering phase assumes the use of many different sources, which are not only end-users and domain experts, but also policies and standards. Requirement engineering also implies the use of different methodologies such as goal-driven, viewpoints-oriented, and scenario-based approaches, or their combinations [47]. None of these approaches usually allow for using different approaches collaboratively, since they are mainly constrained by the tools they use. Recognizing this problem, Lee & Gandhi proposed an ontology-based framework, aka Onto-ActRE, which promotes cohesiveness between the artifacts generated from different modeling techniques and creates a shared understanding from multiple dimensions [47]. The central point of this solution is a Problem Domain Ontology that integrates 1) goal-driven scenario

composition; 2) requirements domain model; 3) viewpoints hierarchy, and 4) other domain specific taxonomies. Leveraging PDO represented in OWL and the Jena Semantic Web framework, they developed the GENeric Object Model (GenOM) tool that, for example, allows requirement engineers to utilize the requirements domain model along with the goals from the goal hierarchy and the associated stakeholders in a viewpoints hierarchy. Although not suggested by Lee & Gandh, the requirements domain model can be obtained from a domain ontology developed by some of the approaches discussed earlier.

3.3 Requirement Engineering Collaboration

Collaboration appears to be the crucial activity in successful requirements engineering, especially in the current global software development landscape. The main challenges to be addressed are [16]: (i) knowledge acquisition and sharing; (ii) effective communication and coordination; and (iii) aligning RE processes and tools. We have already commented on how ontologies can address (i), but there is a need to combine it with (ii) to facilitate efficient collaboration and coordination of involved parties. The use of Wikis appears to be a promising solution to this task. Wikis demonstrate the use of ontologies to define the structure (e.g., concepts such as Use Case, and Actor) and types of documents used in the requirements engineering phase based on the story telling approach [17]. Software engineering can benefit form semantic Wikis as frameworks for (application) ontology engineering by using collective intelligence [64]. Collaborative results produced in semantic Wikis can directly be translated to models used in the design phase (cf. Sec. 4 for details).

Not only are Wikis means of collaboration in requirements engineering, but stakeholders also communicate by other communication channels and tools such as chats, discussion forums, etc. [63]. It would definitely be an important research challenge to leverage ontologies for managing knowledge contained in all these channels (e.g., semantically annotating discussion messages [67] to represent contextual knowledge about why and how some decisions were made). Finally, for a successful collaboration of distributed stakeholders, it is also important that they fully understand the different cultural, geographical, and organizational boundaries. For example, a common problem in collaboration could be a misunderstanding of different requirements engineering tools, methodologies they are based on, and levels of details of the requirement specification requested [18]. While ontologies like the Problem Domain Ontology can certainly harmonize different ontology engineering approaches, there are some open opportunities for applications of ontologies such as to describe requirement engineering tools and methodologies (and connect them with general software engineering development methodologies [30]) or to harmonize communication among stakeholders with different cultural and technical origins.

3.4 Requirements Verification

Testing of identified and specified requirements is a critical activity of the analysis phase, as it is very important to make sure that all involved stakeholders with different bakgrounds and levels of knowledge agree upon the requirements specification. Probably, the most effective way is to use formal model-based animations (e.g. UML use-cases and classes) that present defined requirements. However, as UML does not have formally defined semantics, it is very hard to run simulations that formally analyze the models defined [26]. Although development of methods for formal analysis of models is set as one of the main challenges in the area of model-driven engineering [27], there are already some proven formal languages that have successfully been used for verification of requirement specifications. For example, Jorgensen & Bossen suggest the use of Petri nets for defining executable use-cases [39]; demonstrating potentials of Petri net analysis for requirement engineering.

However, Petri nets are a formalism for modeling processes rather than for modeling a structure (e.g., domain model) of a system under study. The question is then how to combine domain ontologies developed in some of the abovementioned ways and process formalisms such as Petri nets? Brockmans et al. proposed a mechanism for semantic annotation of Petri nets by using concepts from domain ontologies [12]. Taking a similar approach, [28] demonstrates that ontology alignment techniques can assist in the automatic business process integration. This example can stimulate some other applications of ontologies to semantically enrich requirement engineering and even improve tractability of all artifacts produced in this phase to be used in other software lifecycle phases. For example, one could trace requirement document from Petri net models by using ontology concepts annotating Petri net elements. Moreover, these semantic links between requirement documents and Petri net models via the domain ontology could further increase the degree of "intelligence" of content [13]. This ontology annotation of Petri nets can also serve as an interesting direction for further integration of ontologies and models to develop mechanisms to semantically annotate models used in the design phase (e.g., [46] investigates workflow and composition languages). Such semantically annotated artifacts will further be interlinked with the artifact of the implementation and integration phases (e.g., Semantic Web services – Chapter 28).

4 Design

As already mentioned, the design phase assumes a comprehensive definition of the software system under study. As a result, this phase heavily relies on the use of modeling principles and best software practices such as software patterns. Due to the importance of modeling in this phase, in this section, we first introduce model-driven engineering (MDE) as a software engineering

discipline that promotes software development fully based on modeling principles. Then, we discuss how MDE helps to integrate ontologies into software design, and finally, we conclude this section by discussing how ontologies can be applied to improve the use of design patterns.

4.1 Model-Driven Engineering (MDE)

Model Driven Engineering (MDE) is a new software engineering discipline in which the process heavily relies on the use of models [7]. Models are the central MDE concepts and are specified by using modeling languages (e.g., UML or ODM), while modeling languages are defined by metamodels. A metamodel is a model of a modeling language. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language [62]. The core idea of MDE is to increase the productivity of software developers by increasing level of abstraction when developing some software. Once models have been developed, they can be translated to different platform specific implementations (e.g., Java or C#). The OMG's Model Driven Architecture (MDA) is a possible architecture for MDE [50].

MDA consists of three layers, namely: M1 (model) for defining models of systems under study; M2 (metamodel) for defining modeling languages (e.g., UML and Common Warehouse Metamodel (CWM)); and M3 (metametamodel) where only one metamodeling language is defined (i.e. MOF) [53]. The relations between different MDA layers can be considered as instance-of or conformant-to, which means that a model is an instance of a metamodel, and a metamodel is an instance of a metametamodel. Besides MOF, MDA also includes the Object Constraint Language (OCL) to define (more formal) constraints over MOF-defined MDA layers, so that more precise model definitions can formally be verified. OCL is also defined by a MOF-based metamodel resided on the M2 level. Another MDE architecture is Eclipse Modeling Framework (EMF), which is different from MDA just in using Ecore on the M3 layer instead of MOF (cf. Sec. 4.5).

4.2 MDE and Ontologies

Cranefield was the first to explore the synergy of software modeling languages and ontologies [15]. He started from the assumption that there are similarities between the standard concepts of UML and those of ontologies (e.g., classes, relations, and inheritance). Having this in mind, he proposed the use of UML for modeling ontologies due to the wide-acceptance of UML by software engineers and many already-developed UML models, which would facilitate adoption of ontologies by software practitioners. Moreover, software engineers can also benefit from the use of ontology reasoning services (e.g., consistency checking) to reason over UML models. In this way, one can connect software design and ontology development. This motivated several other researchers to look into the problem of similarities and differences between ontology and

software modeling languages (mainly UML). The details about findings could be found in [29].

The abovementioned activities initiated a standardization process at the Object Management Group (OMG) to issue a request for proposals for the Ontology Definition Metamodel (ODM) in 2003. The aim was to define a MOF-based metamodel for the OWL (cf. Chapter 3) and RDF(S) (cf. Chapter 4) ontology languages (i.e., ODM), corresponding ontology UML profile (to use standard UML tools for modeling ontologies), and transformations between ODM and other relevant ontology and modeling languages. These activities resulted in the OMG's ODM specification [54] that defines MOF-based metamodels for Semantic Web ontology languages, RDF(S) and OWL as well as metamodels of Common Logic, Entity-Relationship models, and Topic Maps. The ODM specification also specifies model transformations (by using the OMG's Query/View/Transformations (QVT) standard transformation language [55]) of the ODM and RDFS metamodels with the metamodels of the following languages: UML, Common Logics, Topic Maps, and Entity-Relationship. IBM's tool Integrated Ontology Development Toolkit (IODT) is the most complete implementation of ODM [57].

Note also that application (and domain) ontologies can also be used in the design of software architectures. Grønmo et al. demonstrated how MDE principles can be used to model Semantic Web services (i.e., OWL-S) by extending UML activity diagrams [31]. This approach reminds of the approach for semantic annotation of Petri nets [12]. This demonstrates the importance of further exploration of how ontologies can be integrated into custom modeling languages (e.g., Business Process Modeling Notation - BPMN). This effort can have several contributions to software engineering such as (i) improved tractability of software models when maintaining software and (ii) improved software integration capacity, especially, in the context of service-oriented architectures.

4.3 Software Models and Business Vocabularies

Not always should domain ontologies be defined in the analysis phase, but the requirements specification can be only in the form of documents written in natural language. This implies that we should define our domain models (i.e., ontologies) in the design phase from scratch. So, for this task it will be useful to have an automatic or a semi-automatic approach to produce ontologies for requirement documents (see Chapter 15). Moreover, we should also be able to update textual requirement documents automatically with the changes of ontologies.

Semantics of Business Vocabulary and Business Rules (SBVR) is a promising solution to the above problem [56]. SBVR is the OMG specification that defines a metamodel for capturing expressions in a controlled natural language and representing them in formal logic structures. The SBVR metamodel is compatible with Common Logic. Given that the ODM specification defines

the mappings between the Common Logics metamodel and the metamodels of both OWL and RDF(S), the ODM specification provides a bridge to transform SBVR to OWL, RDF(S), UML, Topic Maps, Entity-Relationship models, and Description Logics.

4.4 Ontologies and Model Reasoning

Software modeling tools are usually very intuitive and allow software designers to use a visual notation of modeling languages (e.g., UML). However, today's software modeling tools lack the support for formal validation of software models, and discovering some potentially hidden implications of such models (e.g., inconsistencies and redundancies), which may impact the overall quality of software designs [27]. Trying to address these issues, Berardi et al. explored the use of description logics to enable reasoning over UML class models [6]. The main finding of their research is that UML class diagrams are EXPTIME-hard, even under restrictive assumptions including only binary associations, only minimal multiplicity constraints, and generalizations (between classes and associations) with disjointness and completeness constraints. They also demonstrated how reasoning over UML class models can become EXPTIME-complete by disabling the arbitrary use of first order OCL predicates, but still allowing disjointness constraints on the generalization hierarchies. A practical contribution is a reasoner that allows for reasoning over UML class models. There are similar on-going research activities in the MDE community to provide formal semantics for UML [27]. The ontology community also considers the use of some UML features (e.g., composite structures) in the future OWL extensions (e.g., OWL 1.1) [58].

4.5 Ontologies and Model Transformations

Model transformation plays an important role and represents the central operation for handling models in MDE [7]. Model transformation is the process of producing one model from another model of the same system [50]. Model transformations are usually defined between different modeling languages that are defined by different metamodels, and hence the process of transformation is usually called metamodel-driven model transformation. The OMG adopted the MOF2 Query View Transformation (QVT) specification [55] to address this need. One of the most commonly used QVT implementations is ATLAS Transformation Language (ATL) [5], which is the official Eclipse recommendation for model-to-model transformations, and yet is an open-source solution. Based on the previous discussions on the similarities between ontologies and models, there have been several approaches that propose the use of ontology alignment techniques to attack the problem of model transformation. The ModelCVS system addresses this problem by transforming (i.e., lifting) metamodels into ontologies (i.e., transforming Ecore to ODM) [42]). Then, such obtained ontologies are further refactored to represent explicitly some

hidden concepts that are usually not precisely represented in metamodels, but should be placed in ontologies. Finally, ontology matching algorithms are executed over such ontologies (e.g., COMA++), and discovered mapping relations are encoded into the ATL transformations. The ontology-based model transformation (ontMT) approach in an attempt that semantically annotates metamodels with the concepts from a reference ontology of a domain [60]. ontMT makes use of such semantic annotations to reason over concepts of the metamodels being mapped and generates model transformations (i.e., ATL) from inferred mapping relations.

Both of these applications of ontologies to model transformations have been recognized as valuable contributions to the MDE area. However, there are many important research questions that should be solved such as: combinations of both approaches to make the process of model transformation more effective; applying ontologies and ontology matching at the model (M1) level of MDA, for example, to improve software refactoring; and applications of ontology matching to contribute round-trip engineering (i.e., code generation and reverse engineering) by complementing the efforts for model-to-text and text-to-model transformations [40].

4.6 Ontologies and Software Patterns

Using the experience form the urban architecture, software engineering adopted the concept of software patterns as an attempt to describe successful solutions to common software problems. The pattern, in short, is a thing, which happens in the world, and the rule which tells us how and when to create it. A pattern language is a network of multiple patterns, with links between related patterns. The most known type of software patterns are design patterns which nowadays are used in almost all applications. Patterns are, in fact, shared knowledge of software engineering, and represent a way for common understanding of software designs. Patterns are described in literary forms, such as sonnets. This works fine if patterns are intended to be understood by software engineers, but if they need to be interpreted by tools, there is a need for a formal representation of patterns [20]. For example, the BORE tool leverages the ontologies to encode the pattern language for usability design patterns [36]. BORE does not automate user interface design, as for effective user-interface design, talent and creativity of the software designer is very importnat. However, the design pattern ontology helps designers improve their knowledge about patterns and share the design experience with other designers easier. As suggested for using semantic annotations of models with ontologies, semantic annotations of design patterns and artifacts can also improve the maintenance, so that one can trace the knowledge on which the design was based [20].

5 Implementation and Integration

The design of software products should specify how the system should finally be implemented and integrated with other software systems, so that the software product eventually accomplishes the requirements initially set. This phase usually looks at lower computer-specific details and is done by using programming languages. Although the goal of MDE is to allow for automatically generating as much implementation code as possible along with many promising results, the current state of the art indicates that many implementation details should still be done manually. This section explores the potentials of using ontologies in the implementation and integration phases.

5.1 Implementation

In this section, we distinguish between three different approaches to the use of ontologies in software implementation.

First, as already indicated, some approaches claim that ontologies could be used in the same manner as models in MDE. Thus, we should be able to generate the implementation of a software system from an ontology, possibly the domain ontology that we created in the analysis phase and refined in the design phase. Following this approach, Cranefield created transformations of UML models to Java code (e.g., classes) besides the RDF(S) ontologies [15], and thus provided a complementary Java implementation for an RDF(S) ontology. However, this approach did not provide mechanisms for preservation of the semantic definitions in ontologies (e.g., OWL restrictions). RDFReactor is a more recent approach that allows for mapping RDF(S) ontologies to Java. Although it does not support OWL, it improves the previous work by eliminating some non-safe type usages (e.g., Java.util.List for properties) by the use of domain specific classes generated from the ontology and leverages the use of static semantic analysis used by compilers of programming languages.

Second, given the AI origins of ontologies, ontologies can also be used in the implementation of software systems in a more declarative way, but yet to use conventional object-oriented programming languages (e.g., Java). HP's Jena Semantic Web framework offers a Java API for handling RDF(S) and OWL ontologies. Examples of alternatives for Jena are the Protégé OWL API and Protégé-Frames API [45]. In this case, we can say that ontologies are not used only for code-generation (like it is the case with MDE and approaches such as [41] and [68]), but ontologies are also a part of the run-time software behavior. A good aspect of implementations based on generic ontology APIs is that they are more dynamic in terms of allowing for on-the-fly ontology changes and updates. However, in these implementations, software developers can not resolve run-time issues by using static semantic analysis. Such run-time issues can hardly be handled with standard exception mechanisms of programming language [45]. In addition, it is not possible to benefit from widely-adopted techniques for software testing, such as JUnit. This group of

implementations can also benefit from the ODM specification, as the OWL and RDF(S) metamodels can also be programmatically managed by using model handlers (i.e., their APIs) such as EMF and Java Metadata Interface (JMI, http://java.sun.com/products/jmi/).

Third, ontologies can be used as a part of the implementation logic in software systems that are implemented by using rule-based languages (e.g., Jess or JBoss Rules). This is the most flexible software implementation approach, as it not only allows for dynamically changing ontologies, but also rules. Then, an inference engine is responsible for execution of rules. Given that most of rule languages define rules over vocabularies and ontologies, this implementation technique can nicely be applied to ontologies [33]. However, rule-based languages are not the widely adopted implementation approach in software engineering and this approach is mainly used for implementation of smaller specialized components with high degree of dynamicity (e.g., e-Negotiations).

Besides the abovementioned approaches, the use of ontology-based semantic annotations can additionally improve software development lifecycle. For example, Java annotation mechanism can be used to semantically interconnect parts of Java code and ontology conceptualization. Not does this can only be useful for JavaBeans[6] to perform some advanced reasoning (e.g., consistency checking), but it can also produce some benefits for the overall software maintenance (e.g., license ontologies can be useful to apply different license policies to different parts of source code). Finally, the potential text mining and ontology-based analysis of the code can be interesting to provide (semi-)automatic approaches to verify some implementation requirements and their designs, similar to the use of ontologies for detection of design errors [37].

5.2 Integration

The most important contribution of ontologies to software integration is semantic Web services. Semantic Web services, as the augmentation of Web service descriptions through Semantic Web annotations, facilitate the higher automation of service discovery, composition, invocation, and monitoring on the Web. In this section, we focus on a relevant topic: *semantic middleware.*

The concept of middleware is applied to managing heterogeneity of various software components and technologies used in a distribute software system. However, it is very important to have environments for developing such middleware-based distributed systems. Application servers are component-based middleware platforms that provide functionalities for developing distributed systems which can use the components developed the developers or third parties. The current management of the functionalities of application severs is based on the use of administrative tools and XML configuration. While this brings a lot of flexibility, there are still many complexity management issues for developers and administrators. These issues are chiefly cased

---

[6] http://blogs.sun.com/bblfish/entry/java_annotations_the_semantic_web

by the lack of an explicit representation of the data in configuration files, or having no commitment to any abstract model that can improve the interpretation of data when developing and analyzing distributed systems [51].

Studying the above issues, Oberle [51] identified the typical challenges at development time as: component dependencies and versioning, licensing, capability descriptions, service classification and discovery, semantics of parameters, and automatic generation of component and service metadata. In addition, typical run time use-cases, requiring more advanced complexity management approaches are: access rights management, error handling, transactional settings, and secure communication. Trying to provide a more generic solution that can be independent of a particular application domain as much as possible, Oberle et al. proposed a stack of the ontologies based on the *DOLCE* (Descriptive Ontology for Linguistic and Cognitive Engineering) generic ontology and its sub-module for descriptions and situations that define patterns for (re)structuring domain ontologies. At the top, the core ontologies of components (typical concepts characterizing components in application servers) and of services (typical concepts characterizing services) are defined (see http://cos.ontoware.org and Chapter 18). These two ontologies are then specialized in domain ontologies by adding concepts specific for a domain of discourse. These ontologies are leveraged in KAON SERVER, a semantic application server that is implemented as an extension of the open-source JBoss application server. Thanks to the ontological description of components, KAON SERVER can perform more advanced analysis of the components used in a distributed system by making use of ontology reasoning and query languages, and thus helps developers and administrators with more contextualized feedback (e.g., who can access a particular component).

The organization of ontologies on which KAON SERVER is based, indicates why it is important to ground domain ontologies in upper-level ontologies (e.g., DOLCE) in the early software lifecycle development phases (e.g., analysis and design). There are many potential benefits for this approach. For example, if our requirement domain models are based on upper-level ontologies, requirement engineers will be able to search for suitable components in the analysis phase. Moreover, the implementation of such systems can later be capable of more flexible integrations with software systems. Indeed, a similar approach for integration of business processes based on the use of Semantic Web services have already been proposed [22], However, the future research should define methodologies that can guide the use of generic ontologies used and refined in all software lifecycle phases (see Chapters 16 and 18 for more information).

6 Maintenance

Any change ever since the client accepts the software, is related to the maintenance software development lifecycle phase. When developing software, soft-

ware engineers need a lot of knowledge about application domain, technologies used, algorithms applied, software testing, and past and new requirements. However, this knowledge is usually not recorded and for software maintainers (which are not necessarily the original software developers) it is very hard to fully understand the system being maintained. It is then not surprising why software maintainers spend 40-60% of their time just to understand the system being maintained [59]. The current software development practice tries to address this problem by requesting software developers and maintainers to document as much of this knowledge as possible. However, documenting software is usually not enough, as software maintainers need an easy assess to the knowledge relevant to the given context of software maintenance. Tractability links between various software artifacts are needed, to make this process more efficient. This is why some researchers argue that software maintenance is a knowledge management task where ontologies play a critical role [19].

To enable the support for managing knowledge of software maintenance, Anquetil et al. developed a comprehensive ontology for software maintenance consisting of five sub-ontologies [2]: the software system ontology with concepts such as software system, users, and documentation; the computer science skills ontology with concepts such as computer science technologies and modeling languages; the modification process ontology with concepts such as modification request and maintenance activity; the organizational structure ontology with concepts such as organizational unit and directive; and the application domain ontology that associates domain concepts with tasks to be performed. Applying Post-Mortem Analysis (a method to elicit knowledge in software engineering), they developed a methodology that allows for explicit representation of knowledge of different stages of the ISO/IEC 14764 maintenance process (i.e., after modification analysis, after implementation of the modification, and at the end of the project) by using their software maintenance ontology. However, this approach does not consider the problem of establishing tractability links with the already developed artifacts in the previous phases. To do so, the knowledge management process requires (semi-)automatic approaches to capture the knowledge encoded in legacy systems.

Witte et al. address the above problems by developing two ontologies, namely, the source code ontology (i.e., an ontology of major concepts of object-oriented programming languages) and documentation ontology (i.e., an ontology of different concepts that may appear in documents related to programming, such as programming languages and data structures) [70]. This experiment demonstrated that these two ontologies allow for establishing tractability links between software documentation (i.e., JavaDoc) and source code. Moreover, such links can help in the validation of, for example, documentation, by checking whether relations described in the documentation (e.g., between a class and a method) actually exist, and whether the documentation reflects the state of the implementation in the source code. This approach allows for even more advanced software maintenance use cases, including, identification of security concerns in source code (e.g., checking whether public and

non-final attributes can be updated outside of the class they belong to) and architectural recovery and restructuring (e.g., checking whether documented architecture such as layered architecture is actually implemented).

Other authors demonstrate that it is possible to perform even more advanced software analysis by using ontologies [43]. Besides using software ontology model (FAMIX-based and language-independent ontology of object-oriented code), this approach uses a bug ontology (inspired by Bugzilla) and a version ontology. These ontologies are first populated by parsing a source code extracted from a CVS. Then, the ontologies are queried by using iSPARQL, an extension of SPARQL that adopts the concept of virtual triples. The three ontologies and iSPARQL can assist software maintainers in use cases such as: code evolution visualization (e.g., how a class evolved in different revisions); detection of code smells (e.g., long parameter list); application of code metrics (e.g., big classes with many methods and attributes and their correlation with bug reports); and ontology reasoning (e.g., methods that are not invoked, aka orphan methods). This project also reports on the scalability issues of today's Semantic Web technologies (e.g., reasoners) which can be another stimulus for the great interest of software engineering in the future research on integrating searching and reasoning approaches on the Web [25].

For software maintainers it can also be important to know what designs are implemented in the maintained source code [20]. An ontology of design patterns can be used to analyze source code and discover design patterns implemented. In addition, this ontology can assist in providing common understanding between software developers and software maintainers.

As we initially indicated, software is a social category, and so is software maintenance. Thus, it is also important to allow for capturing other relevant knowledge related to software maintenance (e.g., exchange of experiences on discussion forums). Capturing such type of knowledge facilitates communication between developers and helps with locating the developers with the most suitable skills, experiences, and reputations. The Dhruv system addresses this problem and facilitates connecting three different types of knowledge, i.e., content, interaction, and community. There are certainly a lot of potentials to experiment with the use of ontologies for social networking (e.g., FOAF) to build networks of software developers. Additionally, this can also be applied to analyze the trustworthiness of software based on the level of its developers reputation. A good example in the line of this research direction is the Baetle ontology (http://code.google.com/p/baetle/) that combines a bug ontology with several other ones (e.g., atom, workflow, and description of a project's ontologies). In addition, software maintenance can benefit from the use of domain ontologies that were built during the software development (as described in the previous sections) or extracted from already developed artifacts [8]. Moreover, there is a potential to use ontologies to semantically annotate the logs of software behaviors, which can be useful for software maintenance (e.g., to synthesize models of behavior [44] and compare them with models defined in the analysis and design phases [11]).

7 Conclusions

To the best of our knowledge, there has been no approach addressing the issues of the retirement phase. Although retirement usually means the end of the use of a software product, it could be very important for software developers to be able to create repositories of retired software, as each retired software system contains a lot of knowledge encoded in its implementation [48]. Thus, we need methods to extract knowledge out of retired software systems, especially those that are implemented using legacy technologies. Some of the ontology-based approaches to software maintenance [20, 43] could be used as good directions for (re-)using knowledge from retired software systems. Ontologies could also play an important role in the on-going effort of the OMG for Architecture-Driven Modernization (ADM) and their metamodel for Knowledge Discovery Metamodel (KDM) [52].

To sum up the current state of the use of ontologies, we refer back to Sec. 2 and analyze the orthogonal dimensions to the software lifecycle phases (i.e., documenting, testing, artifacts, interaction/collaboration, and participants). Documentation is probably the most commonly analyzed application of ontologies with ontologies used in all software lifecycle phases. Domain, upper-level, and document structure ontologies are chiefly used to improve documentation. Still, the documentation activity could additionally benefit from ontologies by developing intelligent tools for software annotation that will for example have features for checking validity of documentation statements w.r.t. the software artifacts [70]. Ontologies also help to have clear semantic relations between different software artifacts and documentations (e.g., models and documents), and thus building software documentations as intelligent contents [13]. This research also indicates that there is a need for developing standard ontologies of documentation structure and types.

Using ontologies for software testing is probably the least explored aspect of software engineering. In fact, we have seen that (upper-level) ontologies are only used to validate requirements and detect design errors [37]. Given a lot of attention to model-based software testing, ontologies are definitely a promising technology (as discussed in Sect. 6) to even outperform MDE-based approaches (e.g., UML) thanks to their strong formal and reasoning foundation. Therefore, further research topics such as semantic annotation of logs of software behaviors for intelligent monitoring, semantic annotations of unit and integration tests, ontology-based reverse engineering, and ontology-based software metrics can bring many potential benefits to software engineering.

The use of ontologies for various software artifacts is probably one of the areas that has attracted a lot of attention so far. Domain and upper-level ontologies, ontologies for documentation, source code, bugs, ontology-based models, model transformations, requirements, and design patterns, are just some examples that are used for important software engineering tasks such as adding more semantics to the artifacts, improving tractability links, consistency checking of models, generating model transformations, and software

metrics. While all these attempts are well-recognized by both the Semantic Web and software engineering communities, further exploration of semantic annotation mechanisms of software models and implementation code, integration of ontologies and metamodeling architectures, and a comprehensive tractability model of software artifacts, are some of the biggest challenges concerning the aspects of software knowledge artifacts.

Interaction and collaboration are fundamental requirements for successful software engineering. The current efforts already demonstrate some interesting results for some of the software lifecycle phases (e.g., facilitating mutual understanding of stakeholders and semantic Wikis for requirement acquisition). However, social aspects of design, implementation, integration, and maintenance phases are almost the dark side of ontologies [35]. Investigating the use of collaborative tagging and folksonomies to improve collaborative experience when designing, implementing and integrating; leveraging social networking ontologies (e.g., FOAF) for annotating software artifacts; and multi-cultural understanding; are some of the potential applications where ontologies can improve interaction capturing and facilitate better collaboration in software engineering [21]. Ontologies can also be a suitable technology for integration of software development environments and collaborative tools (e.g., adding chats like GTalk chats in Gmail). In addition, competence ontologies can help locate software engineers with competencies needed for particular projects, which is one of the most common issues in today's software knowledge management, especially in the domain of global software development [18].

Another important area is to describe software processes and methodologies. Not only do ontologies of methodologies have potentials to be related with modeling languages [30], but they can actually be used to semantically interlink, for example, particular project tasks and activities with all different artifacts produced/used, participants responsible, and interactions done.

References

1. Anupriya Ankolekar, Katia Sycara, James Herbsleb, Robert Kraut, and Chris Welty. Supporting Online Problem-solving Communities with the Semantic Web. In *Proc. of the 15th Int'l conf. on WWW*, pages 575–584, 2006.
2. Nicolas Anquetil, Káthia M. de Oliveira, Kleiber D. de Sousa, and Márcio G. Batista Dias. Software Maintenance seen as a Knowledge Management Issue. *Inf. Softw. Technol.*, 49(5):515–529, 2007.
3. L. Apfelbaum. Model Based Testing. In *Soft. Quality Week 1997 Conf.*, 1997.
4. Phillip G. Armour. Software: hard data. *Commun. ACM*, 49(9):15–17, 2006.
5. ATLAS Transf. Language, 2006. http://www.sciences.univ-nantes.fr/lina/atl.
6. Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artif. Intell.*, 168(1):70–118, 2005.
7. Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.
8. Kalina Bontcheva and Marta Sabou. On Self-Validating Rule Bases. In *Proc. of the 2nd Int'l WSh on Semantic Web Enabled Software Eng.*, 2006.

9. Grady Booch. The Irrelevance of Architecture. *IEEE Soft.*, 24(3):10–11, 2007.

10. Karin Breitman and Julio Cesar Sampaio do Prado Leite. Ontology as a Requirements Engineering Product. In *11th IEEE Int'l Requirements Eng. Conf.*, pages 309–319, 2003.

11. Saartje Brockmans, Robert M. Colomb, Elisa F. Kendall, Evan Wallace, Christopher Welty, Guo Tong Xie, and Peter Haase. A Model Driven Approach for Building OWL DL and OWL Full Ontologies. In *Proc. of the 5th Int'l Semantic Web Conf.*, pages 187–200, 2006.

12. Saartje Brockmans, Marc Ehrig, Agnes Koschmider, Andreas Oberweis, and Rudi Studer. Semantic Alignment of Business Processes. In *Proc. of the 8th Int'l Conf. on Enterprise Info. Sys.*, pages 191–196, 2006.

13. Tobias Bürger. Putting Business Intelligence into Documents. In *Proc. of the WSh. on Semantic Business Process and Product Lifecycle Management*, 2007.

14. Calero Coral, Ruiz Francisco, and Piattini Mario. *Ontologies for Software Engineering and Software Technology*. Springer, Berlin, Heidelberg, 2006.

15. Stephen Cranefield. UML and the Semantic Web. In *Proceedings of the Semantic Web Working Symposium*, pages 113–130, 2001.

16. Daniela Damian. Stakeholders in Global Requirements Engineering: Lessons Learned from Practice. *IEEE Software*, 24(2):21–27, 2007.

17. Björn Decker, Eric Ras, Jörg Rech, Pascal Jaubert, and Marco Rieth. Wiki-Based Stakeholder Participation in Requirements Engineering. *IEEE Software*, 24(2):28–35, 2007.

18. K. C. Desouza, Y. Awazu, and P. Baloh. Managing Knowledge in Global Software Development Efforts: Issues and Practices. *IEEE Soft.*, 23(5):30–37, 2006.

19. Márcio Dias, Nicolas Anquetil, and Kàthia de Oliveira. Organizing the Knowledge Used in Software Maintenance. *J. UCS*, 9(7):641–658, 2003.

20. Jens Dietrich and Chris Elgar. Towards a web of patterns. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(5):108–116, 2007.

21. Jens Dietrich and Nathan Jones. Using Social Networking and Semantic Web Technology in Software Engineering–Use Cases, Patterns, and a Case Study. In *Proce. of the 2007 Australian Software Eng. Conf.*, volume 0, pages 129–136, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

22. Stefan Dietze, Alessio Gugliotta, and John Domingue. A Semantic Web Services-based Infrastructure for Context-Adaptive Process Support. In *Proc. of the International Conf. on Web Services*, pages 537–543, 2007.

23. Joerg Evermann and Yair Wand. Toward Formalizing Domain Modeling Semantics in Language Syntax. *IEEE Trans. Software Eng.*, 31(1):21–37, 2005.

24. Jean-Marie Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon.

25. D. Fensel and F. van Harmelen. Unifying Reasoning and Search to Web Scale. *IEEE Internet Computing*, 11(2):96–95, 2007.

26. R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using UML 2.0: promises and pitfalls. *Computer*, 39(2):59–66, 2006.

27. Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proc. of 28th Int'l Conf. on Software Eng. - Future of Software Engineering* , pages 37–54, 2007.

28. Dragan Gašević and Vladan Devedzić. Petri net ontology. *Knowl.-Based Syst.*, 19(4):220–234, 2006.

29. Dragan Gašević, Dragan Djurić, and Vladan Devedzić. *Model Driven Architecture and Ontology Development*. Springer, 2006.

30. César Gonzàlez-Pèrez and Brian Henderson-Sellers. *An Ontology for Software Development Methodologies and Endeavours*, volume Ontologies for Software Engineering and Software Technology, pages 123–151. Springer, 2006.
31. Roy Grønmo, Michael C. Jaeger, and Hjørdis Hoff. Transformations Between UML and OWL-S. In *Proc. of the 1st European Conference Model Driven Architecture - Foundations and Applications*, pages 269–283, 2005.
32. Hans-Jörg Happel and Stefan Seedorf. Applications of Ontologies in Software Engineering. In *Proc. of the Int'l WSh. on Semantic Web Enabled Software Engineering*, 2006.
33. Marek Hatala, Ron Wakkary, and Leila Kalantari. Rules and ontologies in support of real-time ubiquitous application. *J. Web Sem.*, 3(1):5–22, 2005.
34. J. Hendler. Agents and the Semantic Web. *IEEE Int. Sys.*, 16(2):30–37, 2001.
35. J. Hendler. The Dark Side of the Semantic Web. *IEEE Int. Sys.*, 22(1):2–4, 2007.
36. Scott Henninger and Padmapriya Ashokkumar. An Ontology-Based Infrastructure for Usability Design Patterns. In *Proc. of the Int'l WSh. on Semantic Web Enabled Software Engineering*, pages 41–55, 2005.
37. Allyson Hoss. *Ontology-Based Methodology for Error Detection in Software Design*. PhD thesis, Louisiana State University Graduate School, 2006.
38. IEEE Standard Glossary of Software Engineering Terminology-Description, 1990. http://ieeexplore.ieee.org/servlet/opac?punumber=2238.
39. J.B. Jorgensen and C. Bossen. Executable use cases: requirements for a pervasive health care system. *IEEE Software*, 21(2):34–41, Mar-Apr 2007.
40. Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS:: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proc. of the 5th Int'l Conf. on Generative Prog. and Component Eng.*, pages 249–254, 2006.
41. Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A. Padget. Automatic Mapping of OWL Ontologies into Java. In *Proc. of the 16th Int'l Conf. on Software Eng. and Knowledge Eng.*, pages 98–103, 2004.
42. Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In *Proc. of the ACM/IEEE 9th Int'l Conf. on Model Driven Eng. Languages and Sys.*, pages 528–542, 2006.
43. Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. Analyzing Software with iSPARQL. In *Proc. the 3rd ESWC Int'l WSh. on Semantic Web Enabled Software Eng.*, 2007.
44. Ekkart Kindler, Vladimir Rubin, and Wilhelm Schäfer. Process Mining and Petri Net Synthesis. In *Proc. of Business Process Management WSh.*, pages 105–116, 2006.
45. Holger Knublauch. Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protege/OWL. In *Proc. of 1st Int'l WSh on the Model-Driven Semantic Web*, 2004.
46. Florian Lautenbacher and Bernhard Bauer. A Survey on Workflow Annotation and Composition Approaches. In *Proc. of the Wsh. on Semantic Business Process and Product Lifecycle Management*, 2007.
47. Seok Won Lee and Robin A. Gandhi. Ontology-based Active Requirements Engineering Framework. In *Proc. of the 12th Asia-Pacific Software Eng. Conf.*, pages 481–490, 2005.

48. Raghavendra Rao Loka. Software Development: What Is the Problem? *IEEE Computer*, 40(2):112–111, Feb 2007.
49. Alexander Maedche. *Ontology Learning for the Semantic Web*. Kluwer Academic Publishing, Boston, 2002.
50. J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.
51. Daniel Oberle. *The Semantic management of middleware*. Springer, 2006.
52. OMG KDM. Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model, 2007. http://www.omg.org/cgi-bin/apps/doc?ptc/07-03-15.pdf.
53. OMG MOF. Meta Object Facility (MOF) Core, v2.0, 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.
54. OMG ODM. Ontology Definition Metamodel, 2006. http://www.omg.org/cgi-bin/doc?ad/06-05-01.pdf.
55. OMG QVT. MOF QVT Final Adopted Specification, 2005. http://www.omg.org/docs/ptc/05-11-01.pdf.
56. OMG SBVR. Semantics of Business Vocabulary and Business Rules, 2005. http://www.omg.org/docs/bei/05-08-01.pdf.
57. Yue Pan, GuoTong Xie, Li Ma, Yang Yang, Zhaoming Qiu, and Juhnyoung Lee. Model-Driven Ontology Engineering. *Journal of Data Semantics VII*, pages 57–78, 2006.
58. P. F. Patel-Schneider and I. Horrocks. OWL 1.1 Web Ontology Language: Overview, 2006. http://www.w3.org/Submission/owl11-overview/.
59. S.L. Pfleeger. *Software engineering: theory and practice*. Prentice-Hall, 1998.
60. Stephan Roser and Bernhard Bauer. An Approach to Automatically Generated Model Transformations Using Ontology Engineering Space. In *Proc. of the 2nd Int'l WSh. on Semantic Web Enabled Software Eng.*, 2006.
61. Stephen R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 2006.
62. E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.
63. Vibha Sinha, Bikram Sengupta, and Satish Chandra. Enabling Collaboration in Distributed Requirements Management. *IEEE Software*, 23(5):52–61, 2006.
64. Katharina Siorpaes and Martin Hepp. myOntology: The Marriage of Ontology Engineering and Collective Intelligence. In *Proc. of the Wsh. on Bridging the Gep between Semantic Web and Web 2.0*, pages 127–138, 2007.
65. P. Spyns, Y. Tang, and R. Meersman. A model theory inspired collaborative ontology engineering methodology. *Journal of Applied Ontology*. submitted.
66. P. Tetlow, J. Z. Pan, D. Oberle, E. Wallace, M. Uschold, and E. Kendall. Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering. W3C Working Draft, 2006.
67. V. Uren, P. Cimiano, J. Iria, S. Handschuh, M. Vargas-Vera, E. Motta, and F. Ciravegna. Semantic Annotation for Knowledge Management: Requirements and a Survey of the state of the art. *J. of Web Semantics*, 4(1):14–28, 2006.
68. Max Völkel and York Sure. RDFReactor – From Ontologies to Programmatic Data Access. In *Poster Proc. of the 4th Int'l Semantic Web Conf.*, 2005.
69. Christopher A. Welty. Software Engineering. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *Description Logic Handbook*, pages 373–387. Cambridge University Press, 2003.
70. René Witte, Yonggang Zhang, and Juergen Rilling. Empowering Software Maintainers with Semantic Web Technologies. In *Proc. of the 4th European Semantic Web Conference*, pages 37–52. Springer, 2007.