Ontologies in the Software Engineering process

Wolfgang Hesse

Fachbereich Mathematik und Informatik, Univ. Marburg,
Hans Meerwein-Str., D-35032 Marburg
hesse@informatik.uni-marburg.de

Abstract: The term *ontology* has become popular in several fields of Informatics like Artificial Intelligence, Agent systems, Database or Web Technology. Deviating from its original philosophical meaning, in the context of Computer Sciences the term *ontology* stands for *a formal explicit specification of a shared conceptualization.* Software Engineering (SE) is a field where conceptualisation plays a major role, e.g. in the early phases of software development, in the definition, use and re-use of software components and as a basis for their integration. Thus, ontologies are likely to invade the SE field as well soon.

In this contribution, conceptual modeling as practiced in SE and Information Systems projects is contrasted with the ontology approach. The corresponding life cycle models for their development are compared. Finally, some perspectives of an *Ontology-based Software Engineering (OBSE)* approach are outlined.

1 Introduction: What is an ontology?

In several fields of Informatics such as Artificial Intelligence, Agent systems, Information systems, Database or Web technology, the term "ontology" has widely been used during the last years. Various languages and techniques for their definition and use have been published and disseminated. Recently, the ontology approach is being considered and debated in parts of the Software Engineering (SE) and the Business Applications communities – mainly for its proximity to the (conceptual and business) modelling fields, to the UML language and the fields of software component re-use and integration.

Before we start to introduce an old term like *ontology* into a new branch of Computer Science it is worthwhile to reflect on its traditional use in the area of philosophy where it is originating from. Originally *ontology* means the "science of being" - more accurately: it deals with the *possibilities* and *conditions* of the being. In other words, philosophers ask what it means "to be", what are the prerequisites and limitations of "being", its origins and future horizons. Such questions lead immediately into further questions on what we might *know about* the being of things and facts and thus ontology is strongly related to *epistemology* and *cognition theory*.

In contrast to most philosophers, computer scientists adopts a naïve, "realistic" approach. Normally, we accept without further questions as "being" what we can

observe, feel, touch or derive form these sensations by reasoning and other mental reflection. However, views and convictions about the same "being object" might deviate from one observer to the next. This makes communication and coming to an agreement often a difficult or even unsolvable task. Our observation of and communication on "being things" is hampered by at least two principal reasons (cf. fig 1):

(1) We cannot be sure that what we observe really "is" or "exists" in the fundamental ontological sense.

(2) We communicate our knowledge on what "is" or "exists" using particular languages (including the so-called "natural" ones) which always are limited in their expressiveness and force us to reduction, incompleteness and imprecision.

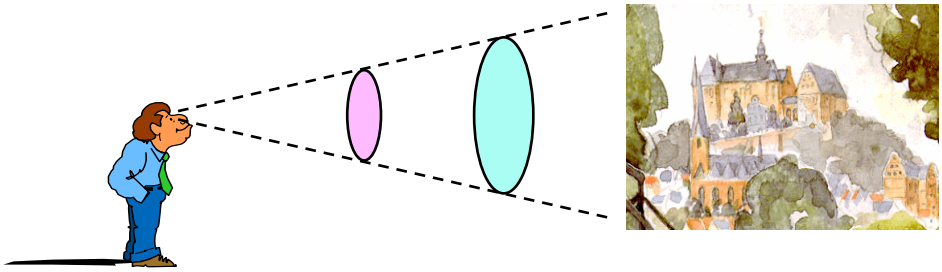

Fig. 1: Filtered view on the "being"

Therefore an ontology cannot express *"what is"* but at most what *we believe to know* about some part of the world and what we are able to *communicate* through the language at our disposal. In Informatics, such a communication takes place at several points: between users and developers of a planned or existing software system while analysing their requirements and (re-) designing or implementing the system functionality, between users and systems or their components while working through a user interface or between components of systems or automated devices while co-operating with each other to solve a given task.

In Artificial Intelligence, the notion *ontology* was introduced in the 1970ies - mainly to denote representations of particular areas of knowledge as needed e. g. for automated devices (robots, agents) for their orientation while performing given tasks. Later, this notion was slightly generalized leading (for example) to the following definitions:

> *"Ontologies provide a shared and common understanding of a domain that can be communicated between people and application systems."* (Hensel, [Hen 00])
>
> *"The role of ontologies is to capture domain knowledge in a generic way and to provide a commonly agreed upon understanding of a domain. The common vocabulary of an ontology, defining the meaning of terms and their relations, is usually organised in a taxonomy and contains modelling primitives such as classes, relations, functions, and axioms."* (Mädche et. al, [MSS+00])

Probably the most condensed definition originates from T. Gruber ([Gru 93]):

> *"An ontology is a formal explicit specification of a shared conceptualization"*

In this context, the use of "ontologies" (in plural form) does indeed make sense: At first, there are many parts of the world which can be conceptualised (while one single, universal conceptualisation of the "whole world" is beyond any limits of reasoning and data storage capabilities). Secondly, for any given knowledge domain infinitely many possible conceptualisations might be formed. Thus for most domains several competing ontologies co-exist in practice.

The use of the notion "ontology" in Informatics is rather controversial – some philosophers even consider it a misuse [Jan 01]. Indeed, "*standardised reference model for a certain application domain"* would be more exact and self-explaining – but would it be more practical? At least, there seems to be some congruence between Gruber's definition and the constructivist position in modern philosophy: We cannot know what the world is like but we can construct (and communicate) world views which are to be assessed rather through criteria of *practicability* than by those of *truth*. If we consider software development as a part of "reality construction" [FZB+ 92], ontologies (in this restricted sense) might find their adequate place in it.

2 Ontologies in and for Software Engineering

Which roles might ontologies play in the Software Engineering field? Basically, we see two close connections:

- *Ontologies in the Software Engineering process:* Modularisation, distribution, reuse and integration of software components and systems is among the central SE issues from its first days. The more these tasks are extended and automated, the more important gets the definition and use of ontologies as conceptual basis of such components. Recent software development methodologies such as model-driven development (MDD) [M-M 03] favour models as the central knowledge base from which different implementations may be derived. If systems or components are to exchange knowledge, this will happen more on the modeling than on the implementation level. In the era of coalescing application landscapes a big potential for defining, implementing, disseminating and using ontologies is evolving.
- *Ontologies for the Software Engineering domain:* SE is an own scientific and professional domain with an own structure and terminology. Since it is a rather new domain the task of formulating one (or several?) ontology/ies for this field is a necessary and challenging task. We mention two approaches: the *SWEBoK project* in the English-spoken community [DBA+99] and the German *terminology network* (*"Begriffsnetz"*, [GI 03]).

In the rest of this article I will focus on the first point: *Ontologies in the Software Engineering process***.**

3 Ontologies: just a new label for old conceptual models?

At least since the days when P. Chen invented his Entity/Relationship (E/R-) model and published the corresponding diagrams, *Conceptual Modelling (CM)* is considered a

key activity for database development. In the SE field, the importance of data modelling was recognised and the E/R techniques became popular a few years later. Since then they have strongly influenced most SE analysis and design methodologies, languages and tools. E/R diagrams were diligently modified, re-defined and extended (e.g. by inheritance mechanisms) and they became the central modelling tool for almost every SE analysis and design method.

When the object oriented (OO-) modelling techniques came up in the 1980ies and 1990ies, the former E/R diagrams were enhanced to form so-called *class and object diagrams*. These encompass certain aspects of behaviour modelling which are expressed by operation definitions attached to the entities and their attributes. This development culminated in the Unified Modelling Language (cf. [UML 01]) that offer class structure diagrams for conceptual modelling and other diagram techniques for modelling further aspects like dynamics, states and state transitions, co-operation of components etc.

Both approaches have in common the goal of representing the relevant "things" of an application area and their interrelationships. In principle, for defining an ontology the same concepts and languages may be used. In both fields we find a "syntactical" and a "semantical" region: The first encompasses the naming, classification and structural description of "things" (*entities*) and their associations (*relationships*), including their behaviour expressed by operations with their names, parameter and result types. Parts of the "semantics" are further conditions, constraints, connections and "meaning" – background information which often is not (conveniently) expressible by (foreground) diagrams or by formal language. A conceptual model is the more "semantic", the more it is able to cover such phenomena in a formalised way.

However, there are differences as well: First, we have to deal with different scopes. Traditional CM is normally used for one particular project (or at least for a given, known, limited family of projects), its concepts and definitions do normally not refer to areas beyond the project scope. Requirements on the commitment and stability of the model are limited to the project or project family scope, i.e. a conceptual model can be changed whenever the involved project members are informed and agree on it.

This is quite different for an ontology: The group of people (or even automata) who share its contents (in the sense of Gruber's Definition, cf. above) is much bigger and not limited to a specific project. It encompasses future projects and developments within the same domain including potential, possibly still unknown ontology use(r)s. The increased number of people from which an *ontological commitment* [G-L 02] is expected and the extension of such commitments to non-human devices are main reasons to demand for more precise, exchangeable and negotiable ontologies.

Spyns et al. contrast essential properties of data models and ontologies [SMJ 02]:

- *Operational levels:* While data models contain domain rules expressed on a rather low, implementation-oriented level (e.g. concerning null values or primary keys) ontologies should be implementation-independent.
- *Expresssive power:* According to the different operational levels different sorts of languages and description tools are appropriate. SQL is a good language for data modelling but surely not for ontologies where the query aspect is less important than e.g. structural and taxonomical issues.

- *Users, purpose and goals:* Here the differences follow directly from the different scopes: Data models are defined for a specific application in a limited domain, while ontologies should span the borderline to adjacent domains and be usable for many applications (including unknown, future ones).
- *Extensibility:* With respect to this point the requirements on ontologies are much more challenging: When a data model is defined, likely extensions can be foreseen and taken into account. Other for an ontology: ideally it should anticipate non-foreseen uses and be open for any kind of extensions.

The authors highlight the apparent conflict between genericity and effectiveness: The more generic an ontology is conceived, the broader is its potential scope – but being less specific normally implies to abandon specific domain rules that are needed for effective interoperability (cf. [SMJ 02], p.3).

4 Approaches to an ontology-based software process

If we want to characterise the specifics of *Ontology-based Software Engineering (OBSE)*, many questions come into play. First we try to list and answer a few principal ones of these and then discuss some alternatives for founding an OBSE process.

Q.: For which kind of projects could an ontology-based approach be adequate?

A.: Whenever a software development project is not just concerned with a specific application but is part of a landscape of projects located in the same domain, an ontology-based approach might be worth considering and even be advantageous to a traditional approach.

Q.: Which are the specific goals of OBSE (on contrast to traditional approaches)?

A.: One principal goal is to extend the idea of reuse from the implementation to the modelling level. I.e. instead of building systems from ready-made components which are "plugged together" like hardware modules, ontologies are *reusable model components* from which particular implementations can be derived for specific platforms, according to specific interfaces and constraints of application development projects.

Q.: Which concepts and paradigms of recent software technology do fit better or worse to an OBSE approach?

A.: Well-known recent paradigms are the *Model-Driven Development (MDD)* and the *Agile Software Development (ASD)* approach. MDD puts the model in the centre of the development process and thus encourages the reuse of models – as is required for OBSE. On the other hand, agile methods do not emphasise modelling activities or specifications but prefer quick implementation cycles and code refactoring. Therefore, in OBSE their place (and use) might be restricted to particular tasks like building prototype applications or implementations of ontologies.

Q.: Which languages and description tools are to be preferred for OBSE?

A.: This is still a very controversial question. Well-known ontology implementation languages like XML+RDF or DAML+OIL are on the wrong operational level – maybe good for automated agents but inappropriate for software engineers while

modelling. An ontology language in the SE context should primarily be made for humans - but hopefully be translatable for automata. Of course, UML is a serious candidate for an OBSE language. (cf. e.g. [Bac+ 02], [Cra 01]). However, it can be argued that it is too wide on the one hand side and too narrow on the other.

Too wide - since many of its constructs like sequence or deployment diagrams are out of scope or at least marginal from an ontological point of view. Too narrow - since the UML class structure is rather rigid and, for example, attributes, operations and associations are no first-class constructs, i.e. can only be used in connection with class definitions. This raises a further, more fundamental and well-known question concerning *granularity* in CM: Should one formally distinguish entities (classes/ objects) from features (attributes, operations) - as do the E/R- and UML approaches – or treat them all the same way, i.e. follow an ORM-like approach (cf. [Hal 01], [SMJ 02])?

To answer these and further questions in detail would require much more space (and work) than can be spent for this paper. Instead, I shall concentrate on issues regarding the *OBSE process*. If we start from the traditional SE life cycle, which of its phases are mostly affected by the OBSE goals? Of course, all those that are particularly concerned with the application domain, i.e. the early phases of *system and requirements analysis* and of *domain modelling* and the late phases of *integration, deployment, use and revision* of the released software.

Following the now popular, UML-based process models, software development starts with analysing its requirements based on a use case structure *(use case analysis)*. Use cases are mainly analysed and documented on a verbal, informal level using natural language (cf. [JBR 99]). In an ontology development project, the role of use cases might be played by *ontology application cases*, i.e. potential uses of the ontology by (present or future) application projects.

According to Jacobson, use cases can be exploited to find objects, list them in an *object list* and, based on that list, define the *analysis model*, i.e. a UML class structure diagram (or set of diagrams) complemented with the corresponding attribute and operation definitions. Such a procedure – which is already somehow weak for a "normal" development project – is not appropriate for OBSE. A promising approach to derive concepts from use case descriptions employs Formal Concept Analysis [G-W 98], [D-H 00]

The main purpose of an ontology is just to replace restricted, unstable object lists and analysis models by a more stable, well-founded and approved conceptual framework. Here we have to distinguish projects which imply developing a *new ontology* or at least enhancing it by new parts from those building *on an already existing* ontology. In the first case all relevant concepts, properties, relationships, states etc. of the concerned domain have to be defined in a common taxonomical framework. Normally, the scope and generality of the framework go beyond the immediate project requirements. In order to be general enough for further use, the ontology has to be negotiated not only with the immediate customer(s) but has to reflect previous work in the domain and to be confirmed by institutions responsible for that domain such as standardisation committees or taxonomists.

In the second case the project can (hopefully) profit from the OBSE approach. The existing ontology is used as basis for building the analysis model. Classes and objects of this model can be derived from and have to be anchored in the existing ontology. Relevant definitions of the ontology are specialised, adapted or extended according to the particular project requirements. New definitions have to be compatible with the ontology and be prepared for acceptance to it (provided they are general enough). This is the kind of *ontological commitment* to be expected from software projects in a domain covered by an ontology.

Which form is most appropriate for defining and communicating ontologies in software engineering projects? We have already briefly discussed the language level and benefits and shortcomings of a UML-like approach. The main arguments in favour of UML are its diffusion and diversity, opposed by the counter-arguments of wrong granularity, missing flexibility and logical rigidity (cf. above).

Another promising approach builds on glossaries: In the KCPM methodology, a *glossary* is employed as the central knowledge base for gathering, storing and communicating domain knowledge during the requirements capture and modelling phases of software projects [K-M 98]. The glossary is built up by two kinds of (table-like) type descriptions: *thing types* and *connection types*. In order to support the glossary building task, linguistic techniques such as natural language text analysis are employed and supported by corresponding tools [FKM+00]. Glossaries may be transformed into conceptual models or UML-like class structure diagrams according to a set of laws and transformation rules in a semi-automated way.

Spyns et al. follow a similar approach based on the ORM paradigma (*Objects with Roles-Model*, [Hal 01]). The authors distinguish a set of generally mandatory definitions called ontology base and packages of domain-specific rules called ontological commitments (cf. [SMJ 02]). The *ontology base* consists of a collection of fact declarations, the so-called *lexons*, which basically express n-ary associations between terms (representing things or other associations). A *commitment* consists of a set of semi-formal rules – i.e. rules partly given in natural language, partly in a formal language. They mediate between the ontology base and its applications. For example, it is determined by such rules, which parts of the ontology are visible for the specific application and which are not or which additional conditions or constraints apply to them. In the course of application development projects, these rules can step-by-step be further formalised. Thus glossary-based modelling work is always performed in two steps:

(1) analysing and further refining the ontology base and

(2) adapting this base to the own project and to the specific application by ontological commitment, i.e. determining rules for their use, constraints or extensions.

To sum up, we can state: An ontology can well be used to facilitate software development processes in the long term horizon and serve for more homogeneous systems – in particular with respect to their interoperability and (partial) re-use. An ideal ontology should comprise domain knowledge as detailed and accurate as possible, but on the other hand it should be free from implementation-dependent details which might constrain the space of possible solutions in an undue way.

5 The ontology life cycle

Now we are ready to look for the ontology development process against the background of SE processes. First we consider an ontology as a special piece of "software" which has its own – mostly rather long-term life cycle. This life cycle is intertwined with the life cycles of software application development projects that are connected to the ontology by the same application domain. Both life cycles are maintained by different roles: the software engineer and the ontology engineer. Nevertheless Fernandez et al. have emphasised a certain analogy between the two processes and have investigated well-known SE life cycle models as potential paradigms for ontology development (cf. [Fer 99]). Among these are:

- Waterfall,
- incremental development,
- evolutionary development.

Obviously, waterfall-like models are inappropriate: In general, ontology development does not follow a sequential plan with consecutive phases, intermediate results, milestones etc – whereas particular subjects (*ontology components*) might be developed in such a way. More appropriate seems an incremental approach: starting with a kernel ontology, which is stepwise extended by "ontology increments" i.e. partial formalisations of particular subdomains. However, incremental development has to follow an overall plan as well, triggered by the "milestones" of released increments.

For ontology development, an evolutionary approach seems to be most appropriate: It might start with an "ontology prototype" comprising some basic definitions – maybe stated as an initial glossary. Then this is refined and enhanced step by step through a series of application development projects. This means: Each application project initiates a new ontology evolution cycle. For modelling evolution cycles, the EOS model (for *E*volutionary, *O*bject-oriented *S*oftware development, cf. [Hes 96], [Hes 03]) may be helpful. Among its key concepts are:

- *Component-based structure:* Complex systems are viewed as hierarchical compositions of parts called components and modules. Correspondingly, we decompose a (sufficiently complex) ontology into sub-ontologies which can be developed (relatively) independent from each other but have to be integrated and adapted to the sharing partners afterwards.

- *"Fractal" development cycles following the system architecture:* Unlike traditional life cycle models, EOS links development cycles with the system structure (and not vice versa): every component has its own cycle. Typically, these are performed concurrently and have to be synchronised – leading to a "fractal" process structure (cf. fig. 3, right part). Accordingly, ontology development can be viewed as a complex process structured in a "fractal" way, where single components may be worked out as parts of particular application development projects.

- *Homogeneous concurrent development cycles:* All development cycles (be it on the system, component or sub-component level) have the same structure consisting of the four main phases: *analysis, design, implementation* and *operational use*. In the context of ontology development, these phases can be characterised as follows:

Analysis: In case of building a new ontology, this first phase serves for starting the overall ontology development process. The universe of discourse, goals and purposes are determined, possible sources of knowledge are identified. Interfaces and references to other existing, relevant ontologies or components (prior, ancestor, descendent, neighbour, competing, conflicting ones ...) are investigated. The role of "use cases" is taken by existing or prospective applications, i.e. projects which (might) use the ontology presently or in the future. The level of formality, possible description languages, tools etc. are determined. A (provisional) glossary is built – e.g. following the KPCM or ORM approach (cf. above). An overall ontology development plan and a strategy for its further development and integration are defined. In case of re-working an existing ontology, analysis starts with the already existing definitions and builds on them.

Design: The structure and hierarchy, possible sub-ontologies, interfaces and rules for commitment are determined in this phase. The ontology domain is conceptualised, the glossary is (further) filled, enhanced, and extended, explanations and cross-references are given. Mappings and translations to "dialects" like E/R- or UML-diagrams or formal ontology languages are provided. Completeness and consistency of definitions are checked.

Implementation: The ontology is translated into a concrete ontology or programming language (e.g. DAML+OIL, OWL, XML+RDF, DL, Prolog, Java classes) according to the requirements. Results from sub-development cycles (regarding sub-ontologies) are integrated. The ontology definitions are published for (potential) users.

Operational use: The integration process is completed with ancestor and neighbour ontologies (cf. also [FGJ 97], p. 2). Integration units are checked for overlapping parts, inconsistencies, incompleteness etc. Feedback from application projects, organisations, users, etc. is gathered and analysed. If justified by new requirements and the administrative prerequisites are given, a revision cycle may be started.

Now we are ready to compare the Software Engineering and Ontology Engineering life cycles. In the following table, some of their characteristics are contrasted:

	Software Engineering	**Ontology Engineering**
Duration	• determined, limited for one project	• undetermined, unlimited
Structure	• phases, grouped in iterations and activities	• evolution cycles, grouped in phases and activities
Sub-processes	• for components, increments or special tasks (e.g. QA)	• for developing or revising subdomains
Paradigms	• waterfall • incremental • component-based	• incremental • component-based • evolutionary

	• prototyping, spiral-like	
Activities and artefacts	• find and describe use cases • elicit requirements • build conceptual model / class structure • build system architecture, specify components and modules • code: use/generate concrete programming language • test and debug • integrate, test subsystems • deploy system • validate, adapt to environment • get feedback form users • start revision (if necessary)	• identify potential applications • analyse terminology • build taxonomy • build and fill glossary • define facts and rules • check with other glossaries, solve terminological conflicts • formalise: translate into formal ontology language • integrate: link and validate sub-ontologies, adapt to super-/ neighbour ontology • publish ontology • validate, receive feedback • elicit requirements for revision
Languages/ description tools	• use case diagrams • natural language • E/R diagrams, UML • pseudo code • programming language(s)	• natural language • E/R diagrams, UML • glossaries, tables • semantic networks • topic maps • logic languages • conceptual graphs • frames, DAML+OIL, OWL
Target groups	• (other) Developers • Users • Customers	• Domain experts • Developers • Agents of other systems
Results and products	• project-specific • short-term oriented • (mostly) not re-usable • isolated, restricted for particular application	• spanning many projects • long-term oriented • re-usable • "sharable" among many organisations and projects

Fig. 2: Software Engineering vs. Ontology engineering

6 Outlook: Perspectives of *Ontology-based Software Engineering*

In this last section, I want to resume my vision on *Ontology-based Software Engineering (OBSE)* in the form of a few theses:

- Ontology development and software development have their own, concurrent, intertwined cycles which have something in common but also differ in their goals, responsibilities and time horizons. Ontology development is more long-term oriented and has a wider scope. Normally it is linked to many software projects and managed by project-spanning teams and organisations.
- If there exists already an ontology for the application domain of a software project, this is a candidate for an OBSE project. During the early phases it can "dock" at the ontology development and profit from earlier work done there. In the late phases, there is a backward knowledge transfer of the project results to the ontology – as far as they are relevant for it. The EOS model can be used as a basis for a common, co-evolutionary model (cf. fig. 3)

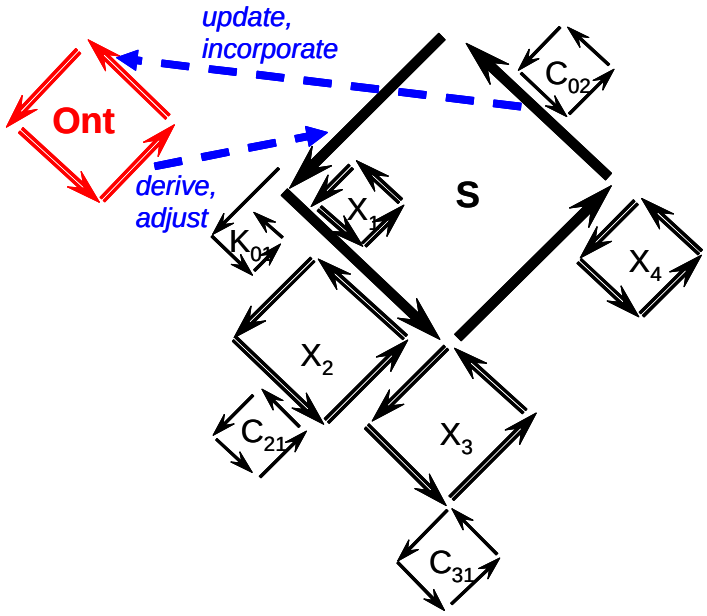

Fig. 3: Co-evolution of ontology and software development

- In a first attempt, the ontology might be viewed as a special (pre-fabricated) *component* in the software development process. However, if it comprises the whole application domain it is more likely to represent knowledge to be used in many components of the system to be developed. Thus the. During the project, it is further developed and parts of it influence the development of particular components of the system.

- Ontology and software development are intertwined in manifold respects. Particularly important are the connections in the early and late phases:

  - *System analysis* implies the *investigation* of existing ontologies and the transfer of codified knowledge for the application domain being considered.

  - *System design* builds on ontological definitions and *commitments* for the current project.

  - *System implementation and operational use* imply *feedback* of the project *results* and *experiences* to the ontology developers/maintainers and have to be incorporated there in order to keep the ontology a living organism.

- Ontologies are a promising instrument for *knowledge transfer* from project to project in a certain application domain and from one development cycle of a project to the next. In the mid- and long-term future, OBSE might become an attractive software engineering paradigm which serves for closer co-operation, better compatible models, more re-usable components and fewer costs in the software development field.

References:

[Bac+02] K. Baclawski et al.: Extending the Unified Modeling Language for ontology development. Software and Systems Modeling - SoSym 1 (2), pp. 142-156 (2002)

[Cra 01] S. Cranefield: UML and the Semantic Web, Sem. Web Working Symposium, Stanford 2001 http://www.semanticweb.org/SWWS/program/full/paper1.pdf

[D-H 00] S. Düwel, W. Hesse: Bridging the gap between Use Case Analysis and Class Structure Design by Formal Concept Analysis. In: J. Ebert, U. Frank (Hrsg.): Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proc. "Modellierung 2000", pp. 27-40, Fölbach-Verlag, Koblenz 2000

[DBA+99] R. Dupuis, P. Bourque, A. Abran, J. W. Moore, and L. L. Tripp :The SWEBOK Project: Guide to the Software Engineering Body of Knowledge Presented at ICSSEA '99 Paris, France December 8-10, 1999.

[Fer 99] M. Fernandez, A. Gomez-Perez, N. Juristo: Methontology: From ontological art towards ontological engineering, Symp. on Ontological Engineering of AAAI, Stanford Ca. (1997)

[FKM+00] G. Fliedl, Ch. Kop, H. C. Mayr, W. Mayerthaler, Ch. Winkler: Linguistically based requirements engineering - The NIBA project. In: Data & Knowledge Engineering, Vol. 35, 2000, pp. 111 - 120

[FZB+92] Ch. Floyd, H. Züllighoven, R. Budde, R. Keil-Slawik: Software Development and Reality Construction, Springer-Verlag 1992

[G-W 98] B. Ganter, R. Wille: Formal Concept Analysis, Mathematical Foundations, Springer 1998

[Gua 98] N. Guarino: Formal Ontology and Information Systems. In: Proc. FOIS '98, Trento (Italy) June 1998, Amsterdam IOS Press pp 3-15

[GI 03] Gesellschaft für Informatik (GI)-Arbeitskreise "Terminologie der Softwaretechnik" und Begriffe für Vorgehensmodelle": Informatik-Begriffsnetz. http://www.tfh-berlin.de/%7Egiak/

[Gru 93] T. Gruber: A translation approach to portable ontologies. Knowledge Acquisition, 5(2), pp. 199-220 (1993), also: *What is an Ontology?* http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[G-L 02] M. Gruninger, J. Lee: Ontology - Applications and Design. CACM 45.2, pp. 39-41 (Feb. 2002)

[Hal 01] T. Halpin: Information Modeling and Relational Databases: from conceptual analysis to logical design. Morgan-Kaufmann 2001

[Hen 00] D. Hensel: *Relating Ontology Languages and Web Standards.* In: J. Ebert, U. Frank (Hrsg.): *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proc. "Modellierung 2000"*, Fölbach-Verlag, Koblenz 2000, pp. 111-128

[Hes 96] W. Hesse: Theory and practice of the software process - a field study and its implications for project management; in: C. Montangero (Ed.): Software Process Technology, 5th European Workshop, EWSPT 96, Springer LNCS 1149, pp. 241-256 (1996)

[Hes 02] W. Hesse: Das aktuelle Schlagwort: Ontologie(n). in: Informatik Spektrum, Band 25.6 (Dez. 2002)

[Hes 03] W. Hesse: Dinosaur Meets Archaeopteryx? or: Is there an Alternative for Rational's Unified Process? Software and Systems Modeling (SoSyM) Vol. 2. No. 4, pp. 240-247 (2003)

[JBR 99] I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process. Addison-Wesley 1999

[Jan 01] P. Janich: Wozu Ontologie für Informatiker? Objektbezug durch Sprachkritik. In: K. Bauknecht et al. (eds.): Informatik 2001 - Tagungsband der GI/OCG-Jahrestagung, Bd. II, pp. 765-769. books_372ocg.at; Bd. 157, Österr. Computer-Gesellschaft 2001

[M-K 03] H.C. Mayr , Ch. Kop: A User Centered Approach to Requirements Modeling. In: M. Glinz, G. Müller-Luschnat (Hrsg.): Modellierung 2002 - Modellierung in der Praxis - Modellierung für die Praxis, pp. 75-86, Springer LNI P-12 (2003)

[M-M 03] J. Miller, J. Mukerji: MDA Guide. Version 1.1.1, Object Management Group 2003.

[MSS+00] A. Mädche, H.-P. Schnurr, S. Staab, R. Studer: Representation-Language-Neutral Modelling of Ontologies, in: J. Ebert, U. Frank (Hrsg.): *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proc. "Modellierung 2000"*, pp. 143-150, Fölbach-Verlag, Koblenz 2000

[SMJ 02] P. Spyns, R. Meersman, M. Jarrar: Data modelling versus Ontology engineering, SIGMOD Record 31 (4), Dec. 2002

[UML 01] Unified Modeling Language (UML) 1.5 Documentation. OMG document ad/99/06-09. Rational Software Corp., Santa Clara, CA 2001. http://www.rational.com/uml/resources/ documentation