Knowledge-Based Support for Software Engineering

Dencho Batanov
Department of Computer Science and Engineering
Frederick University, 7 Frederickou Str., Pallouriotisa,
Nicosia 1036, Cyprus, com.bd@fit.ac,cy

Abstract. The existing ambiguity of the notion of software engineering is mainly due to the fact that it is based on and depends on knowledge. The new definition of the term "software engineering", proposed in this paper, encounters that fact. The main subject of discussion in the paper is how three different types of knowledge, namely declarative explicit, declarative structured (ontologies) and tacit can be used for effective support of software engineering as both practice and academic subject. Illustrative examples are shown along with some trends for more intensive use of knowledge for support of software engineering.

Keywords: Knowledge, Software Engineering, Software Development Life Cycle (SDLC), Ontology, Expert System.

1 The Software Engineering Paradigm

Technology has two different but inseparable meanings – as tools and processes [1]. Although the technological tools (products) are more popular and attractive in our everyday life they could not be created and produced without respective processes. Good examples are engineering of any product in general and software engineering in particular. To understand the software engineering paradigm, which we are interested in, it would be helpful to briefly trace back the ambiguity and evolution of the meaning of the term "software engineering" since its inventing at the 1968 NATO Conference on Software Engineering [2]. Numerous definitions of the term have been given over time emphasizing diverse aspects of the notion and more than forty years still there are differences and disagreements. Most of those definitions are centered on the point that the development of software from the initial phase of requirements analysis and specification to the maintenance of software product is strongly linked to the notion of engineering as both academic discipline and profession.

Engineering is a mix of craft and sciences [3] with more dominating and increasing role of sciences in the last centuries because of the demands for more complex functionalities, higher qualities and greater quantities of the products as well as for more complicated management of the process. For some more specific products however crafts, which are characterized by learning by doing, idiosyncratic approach and production of handmade artifacts [4], are the basis of engineering process. Software is without any doubt quite specific product – unique, invisible, getting better over time, flexible and therefore easily modifiable, scaleable in large boundaries, etc., and it is not surprising that engineering of such products requires specific term – "software engineering". There are a lot of publications regarding the nature of this term with speculations, discussions, agreements and disagreements with existing definitions, the place of software engineering in science, practice and education and possible ways of its evolution. The objective of this paper is not to survey those publications but it is worth noting that they vary from emphasizing the absence of fundamental theory [5] through the need of a theory for software engineering [6] and the differences between software engineering and computer science [7] to even looking for some similarities and differences between fashion, politics and software engineering [8]. Central point in all these publications is that software engineering is not a rigorous discipline comparing to the core subjects constituting computer science as area of research, education and practice, such as data structures and algorithms, queuing theory, complexity, languages syntax and semantics, machine learning, etc. Analysis of the nature of those disciplines leads to an interesting thesis, expressed by Chuck Connell in [8]: *Software engineering will never be rigorous discipline with proven results, because it involves human activity*. And even more: *We should stop trying to prove fundamental results in software engineering and accept that the significant advance in this domain will be general*

guidelines. Such a thesis is valid at least to some extent for al engineering disciplines but having in mind the specifics of software products and the existing practice of developing software systems of any size, it could be accepted in full for software engineering although, as the author states, the statements cannot be proved. The problem with this thesis is that the term "human activity" is too broad and vague that obviously cannot be used to clearly define the term of "software engineering". On the other hand the activities which distinguish distinctly the humans from the other forms of life are based on creating and continuous use of knowledge. There are different types of knowledge and different approaches to its classification, for example one is to classify knowledge as static, dynamic, declarative, procedural, heuristic, knowledge of methods and knowledge of equipment and tools; another is to separate knowledge in three different levels – surface, domain and deep; and as another option knowledge in classified in two large groups – explicit (objective) and tacit (subjective). No matter how the knowledge is classified the practice of software engineering shows that knowledge of different types is intensively used in all its phases. This can be expressed by the following definition of the term of software engineering:

Software engineering is a systematic approach to the management and development of software systems based on use of all kinds of knowledge, which is embedded in the final software product.

Fig. 1 shows the basic idea of the above definition.

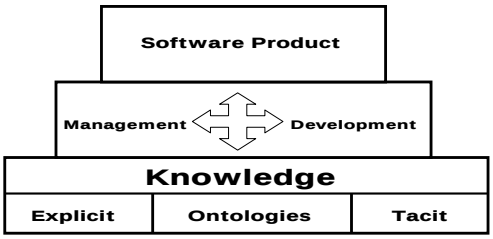

Fig.1. The Software Engineering Pyramid

Central point of the definition is the focus on use of all kinds of knowledge, which in this particular case is classified in three groups: explicit (declarative knowledge), ontologies (declarative structured knowledge) and tacit (individual-related knowledge, which is result of accumulated experience and expressed mainly in a form of rules). It is worth noting at least three interesting properties of the definition: (a) it deals with the two meanings of technology – tools (software products) and processes (management and development); (b) it covers the two major activities in engineering of software systems – management (team organization, choosing the strategy, planning, scheduling, budgeting, cost estimation, maintenance, etc.) and development (use of models, methods, techniques and tools as elements of the chosen methodology), and (c) the quality of the tool (software system) strongly depends on the quality of the processes (management and development).

The software development life cycle (SDLC) consists of well defined phases that developers carry out during the process of software system development and which are subject of research and education. The above definition adds some new aspects of he development, research and education regarding: (a) the necessity of identifying the specific kind of knowledge, which is appropriate for a given phase of SDLS, and (b) use of existing or creating new methods, techniques and tools for gathering, representing and manipulating knowledge for support of the respective phase of SDLC.

The next sections of the paper represent illustrative examples of using the three types of knowledge from the definition for support of software engineering. The use of declarative explicit knowledge in well-known forms as description of models, methods, techniques and tools, manuals, documentation, standards, etc., is briefly mentioned in Section 2. Section 3 is dedicated to one promising way to use structured declarative knowledge in a form of ontology for supporting some of the phases of software

development life cycle. In Section 4 an approach of using tacit knowledge for building expert system for supporting requirements analysis phase is described. Conclusions are outlined and some recommendations for further work are made in Section 5.

2 Use of declarative explicit knowledge

Without doubt this is the kind of knowledge, which is most well known, popular and available. For more than forty years a lot of specialized knowledge has been created, accumulated and disseminated in form of collections of models, methods, techniques and tools, related books, papers, reports on good and bad practices, curricula, training and certification programs with respective teaching and learning materials, manuals, standards and so on. This valuable repository of knowledge is a great and widely used opportunity for practitioners, educators, students and scientists to learn and know more in the field of software engineering. This kind of knowledge, as it is shown in [4] is the basis of transition of the software engineering discipline from craft to profession.

Declarative explicit knowledge, although usually thematically classified, is non-structured in nature. To use such type of knowledge the users need guidance by experienced people to be able to navigate among the numerous sources of related information. That is why this knowledge is used mostly in education, including training and certification. Software engineering education is of primary importance for preparing software developers and is a subject of teaching in all academic institutions all over the world. A good example of accumulated knowledge for creating and applying software engineering curricula for universities is the model, proposed by the Joint Task Force on Computing Curricula of IEEE Computing Society and ACM [2]. The curricula seem to be developed for major of MSc in software engineering but the model can be definitely used for creating the content of related courses at bachelor degree level. We should not forget however that software engineering is based on knowledge and skills from a number of basic courses in Computer Science, such as data structures and algorithms, principles of programming languages, object-oriented programming, databases, etc., which constitute the necessary body of knowledge. In this regard the role of lecturers in software engineering is vital for advising students what to select and read. For example, is not enough to state in the beginning of the course that "software engineering is systematic and disciplined approach to software development" – it is absolutely necessary to support this statement through the entire course with relevant readings, case studies, analyses of good and bad practices and so on. Another good example is to include Software Engineering Project as a separate credited subject in the curriculum giving the students the opportunity to apply their knowledge and skills to development of real software systems. For this purpose they are required to find themselves a lot of additional sources of information about the methods, techniques and tools to be used, analytically compare them and finally to make decision what to be chosen. This is the only way to convert knowledge into practice.

Another useful example of relying on explicit declarative knowledge is the so called codified body of knowledge [4], represented by two significant projects – SEEKA (Software Engineering Education Knowledge Areas) and SWEBOK (SoftWare Engineering Body of Knowledge). SEEKA is more oriented to the knowledge areas that should be covered in an undergraduate curriculum in software engineering while SWEBOK concerns knowledge and practices, which can be applied to most projects most of the time. Although the slight differences both of them offer extremely helpful information about software engineering as a subject of learning and practice.

Software engineering in practice is different from software engineering in education. The differences are in the size and complexity of the projects, the number of people involved, the organization and management of the teams, the required quality of the software product and related compliance with standards, the time and budget constraints, etc. Accordingly, along with the traditional descriptions and manuals of models, methods, techniques and tools, there are additional sources of explicit declarative knowledge, which support the work of practitioners in software engineering. Examples of such sources are the large number of approved and working standards for software quality assurance, the regular publications of professional societies like IEEE and ACM and their special interest groups in software engineering, specialized journals and

proceedings of conferences and workshops, analytical reviews of good and bad practices, project reports and so on. Another well known for specialists example is the so-called "Capability Maturity Model" (CMM) [4], created in the Software Engineering Institute at Carnegie Mellon University. The model helps evaluating the software products in a standardized way, which contributes to improving the working processes and the quality of the product as a whole.

In fact all necessary sources of explicit declarative knowledge are available somewhere in the world repository. Is it enough for more effective and efficient software product development? Not, of course, simply because the access to those resources is difficult and time consuming especially when the developers need the information online. Recent Web technologies however offer solution of this problem through Web services and currently emerging cloud computing. This can be considered as a challenging research and implementation topics for knowledge-based support of software engineering.

3 Use of declarative structured knowledge (ontologies)

Ontology here is defined as declarative structured knowledge because it can be derived from the structural representation of concepts (entities, objects, classes) linked through existing or established relationships in a given problem domain. This is another interpretation of the definition of ontology as a specification of a representational vocabulary for a shared domain of discourse: definitions of classes, relations, functions, and other objects [9] or, more generally, a specification of conceptualization [10]. In this section I will show as illustrative example how ontologies as form of knowledge can be used to support some of the most difficult phases of SDLC in object-oriented software engineering. The readers who would be interested in this example as complete representation can find more detailed description of respective models, methods and techniques in [11], [12].

The motto of classical object-oriented software development may be formulated in different ways, but its essence can be stated simply: "Identify and concentrate on objects in the problem domain description first. Think about the system function later." At the initial analysis phase, however, identifying the right objects, which are vital to the system's functionality, seems to be the most difficult task in the whole development process from both theoretical and practical point of view. Object-oriented software development is well supported by a huge number of working methods, techniques, and tools, except for this starting point - object identification and building the related system object model. Converting the text description of system problem domain and respective functional requirements into an object model is usually left to the intuition and experience of developers. One commonly accepted rule of thumb is, "If an object fits within the context of the system's responsibilities, then include it in the system." However, since the members of the development team are likely to have different views on many points, serious communication problems may occur during the later phases of the software development process. Here is the place where the knowledge represented by ontologies can help. It is worth noting that an ontology is either built already for a given problem domain or, if not, can be created using respective methods, techniques and tools (languages) specific for the field of ontology development, which is not objective of this paper. Fig. 2 illustrates the way in which ontology can be used as a supporting tool for the process of building the object model of the software system and converting its elements (objects) into abstract data types (ADTs). As far as the implementation of ADTs are classes and they are the basic building modules of object-oriented software, it becomes clear that actually the ontologies can help the entire analysis and design phases of SDLC.

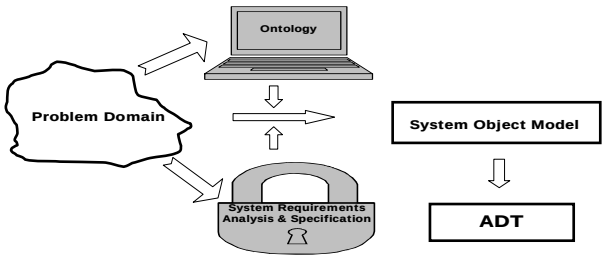

Fig. 2. Ontologies and Building Object Model (ADT)

Models are inseparable and one of the most significant parts of any methodology. They help developers to better understand complex tasks and represent in a simpler way the work they should do to solve those tasks. Fig. 3 shows the models, which we use to transform requirements specification to object model of the system. The starting point of transformation is the text model (T-model), which represents a concise description of the problem domain, where the software system under development will work, written in a natural language, usually English. If not available, the T-model should be created by the developer describing the general user requirements for the system functionality. The presumption is that this problem domain description contains the main objects, which will participate in ensuring the system's functionality. Of course, at this level the objects are represented by their natural names only and as such are very far from the form we need to reach - represented as ADTs.

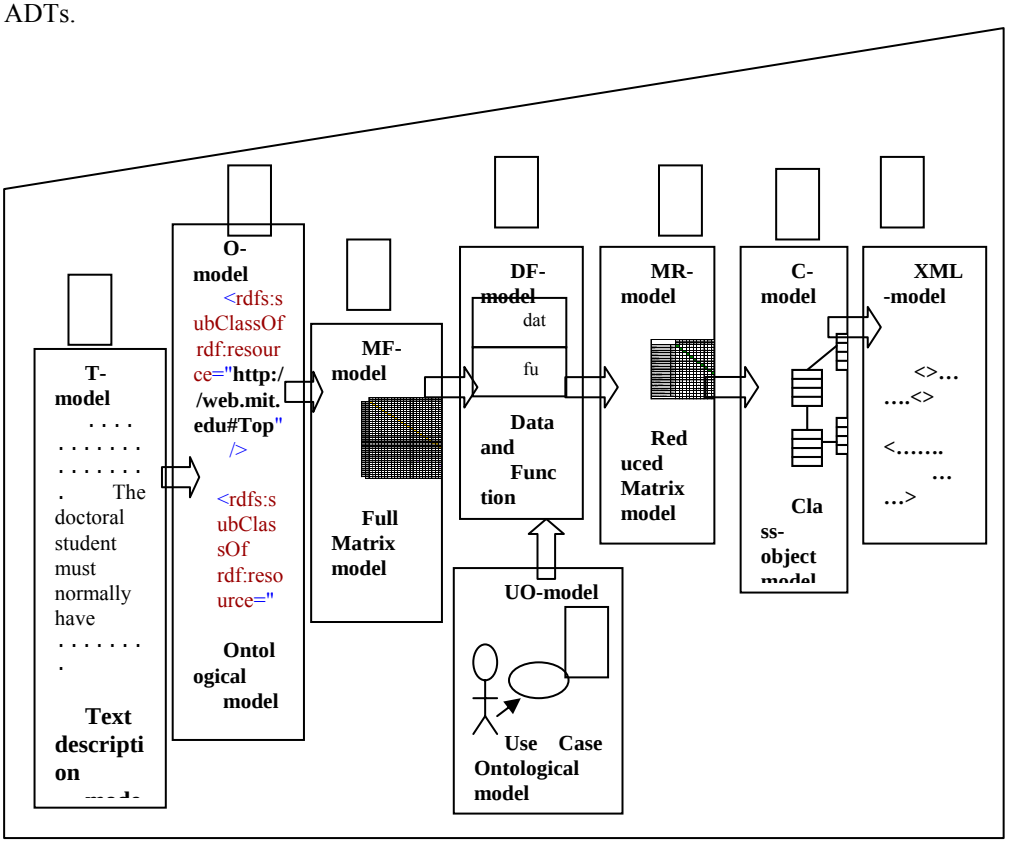

Fig.3. Models for converting a text description into an object model

To help this process we refer to a tool of conceptualization - an ontological engine, which applied to the T-model, generates an ontology model (O-model) of the problem domain. The O-model is a straightforward and practically useful source of information for identifying the participating objects. We use this information to build a so-called Full Matrix

model (MF-model), which represents in a simple form objects along with the linkages (relationships) between them. However, we should say that the processing of the MF-model is semi-formal in nature. This means that at this phase the developer should take important decisions about which objects could be considered as basic ADTs and which, and where, could play a role of attributes of other ADTs. The idea is simple but not very easy for implementation - to reduce the full object matrix to a reduced matrix (we call this model MR-model), which contains only the basic objects represented later as ADTs containing other ADTs as attributes. The implementation is not very easy because we need more information here, which relates to expected functionality of participating objects. This information, however, is available or can be extracted from the Use Case model of the system under development. Note that at this phase we can also use the problem domain ontology. Along with showing the concepts hierarchy (possible objects in the system) the ontologies also analyze the verbs linking those concepts, which can be considered as functions (operations) belonging to respective objects. We actually use the text descriptions of different Use Cases to extract different functionality of the system by the ontological engine and as a result we get the so-called Use Case Ontological model (UO-model). The functionality, expressed by the UO-model, can be used at this phase along with the ontological information about the objects in the MF-model to create the Data and Function model (DF-model). As a matter of principle DF-model can be used for each of the objects in the DF-model but this would lead to a high degree of redundancy and quite complicated matrix presentation even for relatively simple T-models. To avoid this we propose to use so called business object patterns. The representation of the C-model is significantly different from MR-model however, as far as the former shows not only the object hierarchy but the objects' structure as well. In other words, the C-model is a model representing ADTs. The last model, the XML-model is optional but can be very important in practice because it allows the C-model to be published on the Web in a unified (XML-based) format supporting in this way the collaborative work, which is a commonly accepted technology nowadays.

The shown models and the process of their transformation can help developers of complex object-oriented software systems to: (a) transform user requirements (represented as text description) into an object model of the software system based on the use of ontologies; (b) improve the existing methods and techniques for creating a specific ontology from a text description of the system problem domain; (c) work out implementation techniques and tools for semi-automated or automated generating and editing of ADTs for object-oriented application software development, and (d) improve the effectiveness and efficiency of the existing methodology for high-level system analysis in object-oriented software engineering.

4 Use of tacit knowledge

For all phases of SDLC without any exceptions the developers are forced to make decisions, which are vital for the quality of the final product. Give one and the same user requirements and specifications to, let us say ten different teams, and you will get certainly ten absolutely different systems as result. Not different in required functionality but different as user acceptance, performance, cost, reliability, etc., generally speaking different in quality. This is not quite normal for other conventional products but for software the opposite would be not normal. This is the uniqueness of software product. And this is because the people who make decisions at some points of SDLC are different – as background, qualification, experience or as knowledge and skills in the field of software engineering. Practically most of this knowledge is tacit in nature – knowledge, which is hidden, difficult to express, explain, share with others and formalize. Tacit knowledge exists usually in two forms [14]: (1) knowledge embodied in people and social networks, and (2) knowledge embedded in the processes and products that people create. We are interested here in the second form. Because the people develop and use tacit knowledge before they are able to formalize or codify it the problem is how to extract this knowledge from those who possess it and after that to represent and process it in a computerized environment in order to implicitly embed it in the software product. The artificial intelligence offers different representation schemes and respective methods and techniques for manipulating such type of knowledge but the most popular and relatively easy for implementation way remains the use of rule-based expert systems. Unfortunately, especially in the field of software engineering there are only a few examples of such systems, which are attempts to support some of the phases of SDLC. We used one of

these examples - CASSANDRA to support the phase of requirements analysis and specification. This phase is one of the most difficult and ambiguous and at the same time of vital importance for the success of the project. As it is stated in [13] "*Research indicates that nearly 50% of all software project defects originate in the requirements gathering process and that 60% to 80% of project failures can be attributed directly to poor requirements gathering*." Obviously any guidance during this phase would be extremely useful.

CASSANDRA [15] is an ambitious project for developing an automated software engineering coach and the name stands for **C**assandra – an **A**ssistant for **S**ystem **S**pecification **AND R**equirements **A**nalysis. The idea is the support to be in a style that resembles a human coach – proactive, asks the user questions, gives advices and recommendations and explains them in the case of users' request. Everything in CASSANDRA is implemented in Prolog – the user interface, knowledge base, persistency and CASE tool access. Having the framework however, the contents of the inserted facts and questions asked can be easily changed, which allows for adjusting the expert system to the needs of different categories of users. We, for example, developed an expert tool for novice and junior developers with the idea to apply it to the educational process. In addition we changed completely the user interface implementing it in C# in .NET environment, which has connectivity to Prolog, giving in this way an opportunity for more attractive and convenient interaction between the user and system.

The architecture of CASSANDRA is quite complicated and it is not the aim of this paper to consider it in more details. The generation of recommendations however follows the classical mechanism of rule-based expert systems. All questions, facts and recommendations, which are the basic elements of knowledge base, are organized in functionally well-defined groups, such as goals of the system, system users, system constraints, system architecture, functionality definitions, design, performance, maintenance and support. Below are two examples of related fact and recommendation as illustration:

> ***fact (es1,no):-*** *fact ('present software', no), recommend ('The fact that the company is not using any present software at the moment has both its advantages and its disadvantages. That is, you are going to be the first one who will try to find out all the needs of the company therefore more work should be done. On the other hand, it means that you have no competitor to compete with on the functionality of the software.').*
>
> ***recommendation (existing_software):-*** *fact(es1,no), recommend ('You need to develop very careful strategies from which you will retrieve information about how the company is working and for what reason it needs the software for in order to be able to provide the best possible software solution for it. Some of those methods and techniques you can find out in the book of ..... on page .....').*

The inference engine first calls the *recommendation* rule which on its turn calls the *fact* rule in order to be satisfied. The *fact* rule then starts the process to satisfy itself. That is, it starts calling all the facts that are included in its rule.

There are a good number of environments for development of rule-based expert systems. It seems to me however that the problem with having so small number of working examples of expert systems for support of software engineering is to find experts for different phases of SDLC ready for sharing their tacit knowledge. On the other hand the process of creating a solid knowledge base is costly and very time consuming. The decision maybe is the leading organizations in computing as IEEE and ACM to start coordinating this hard work as well as collaboration between universities and respective funding at national level.

5 Conclusion and recommendations

The long time existing ambiguity of the notion of software engineering is mainly due to the fact that it is based on and depends on knowledge. A new definition of the term is proposed in this paper, which states: "*Software engineering is a systematic approach to the management and development of software systems based on use of all kinds of knowledge, which is embedded in the final software product*". More specifically, three types of knowledge are identified as basic for support of developers' activities in software engineering: explicit declarative knowledge, which are unstructured in nature, ontologies as representatives of declarative structured knowledge and tacit knowledge. The main body of the paper is

dedicated to illustrative examples of and respective comments on the use of each one of those kinds of knowledge in the education and practice of software engineering.

Using explicit declarative knowledge, accumulated and disseminated in form of collections of models, methods, techniques and tools, related books, papers, reports on good and bad practices, curricula, training and certification programs with respective teaching and learning materials, manuals, standards and so on, is still dominant. Nowadays Web technologies and more specifically Web services and currently emerging cloud computing can open new opportunities for more efficient and effective use of world repository in the field of software engineering. This can be considered as a challenging research and implementation topics for knowledge-based support of software engineering.

Most interesting, challenging and promising approach to knowledge-based support of practically all phases of SDLC is using ontologies. They either exist or can be created for different specific problem domains and offer good opportunities for merging with software engineering. The example shown and briefly discussed in the paper is dedicated to creating the object model of the software system from a given textual description of the problem domain and system functionality. It is expected more research to be carried out in this promising area.

Finally, an example of using tacit knowledge in an expert system serving as an automated coach in software engineering, and particularly in the requirement analysis phase, and built on the basis of CASSANDRA environment, is shown and discussed. Such type of expert systems could be very helpful for both education and practice in software engineering but their development requires collaborative work and availability of solid resources.

Acknowledgements. My thanks go to the hundreds of my students, who during the years have always inspired me to teach and supervise them better in the fields of software engineering and knowledge-based systems.

References

1. Dencho Batanov, Eero Eloranta, Advanced Web technologies for industrial applications, Guest Editorial, in Computers in Industry, Volume 50 (Special Issue), Number 2, pp.123--125 (2003)
2. Computing Curriculum – Software Engineering, Public Draft 1, The Joint Task Force on Computing Curricula, IEEE Computer Society, ACM (2003)
3. Gary Shute, The Nature of Software Engineering, http://www.d.umn.edu/~gshute/softeng/nature.html (2007)
4. Richard E. (Dick) Fairley, Leonard L. Tripp, Software Engineering: from Craft to Profession, http://cs.wm.edu/~coppit/.../papers/CraftToProfession.pdf (2002)
5. Phillipe Kruchten, The Nature of Software: What's So Special About Software Engineering?, http://www.ibm.com/developerworks/rational/library/4700.html (2004)
6. Ivar Jacobson and Ian Spence, Why We Need A Theory for Software Engineering, Dr. Dobb's Digest, October (2009)
7. Chuck Connel, Software Engineering ≠ Computer Science, Dr. Dobb's Digest, June (2009), Ivar Jacobson and Bertrand Meyer, Dr. Dobb's Digest, August (2009)
8. Gruber T.R., A translation approach to portable ontology specifications. Knowledge Acquisition 5, pp. 199-220 (1993)
9. Gruber T.R., Towards Principles for the Design of Ontologies Use for Knowledge Sharing. In Proceedings of IJHCS-1994, 5 (6), pp. 907--928 (1994)
10. Dencho N. Batanov, Merging ontologies and object-oriented technologies for software development, Proceedings of the 20th International Conference SAER-2006, Plenary paper, 23-24 September, Varna, Bulgaria (2006)
11. Dencho N. Batanov and Waralak Vongdoiwang, Using Ontologies to Create Object Model for Object-Oriented Software Engineering, Chapter 16, Part 3 in "Ontologies. A Handbook of Principles, Concepts and Applications in Information Systems", Editors Raj Charman, Rajiv Kishore and Ram Ramesh, ISBN: 978-0-387-37019-4 and 978-0-387-37022-4 (online), Springer US, pp. 461-487 (2007)
12. A Practical Guide to Effective Requirements Development, SearchSoftwareQuality.com, E-guide (2007)
13. Joseph A. Horvath, http://providersedge.com/docs/km_articles/, (2000)
14. Marcus. Schaher, CASSANDRA: An Automated Software Engineering Coach, KnowGravity Inc, (2001)