A Software Engineering Ontology as Software Engineering Knowledge Representation

P. Wongthongtham, N. Kasisopha, E. Chang, T. Dillon
Digital Ecosystems and Business Intelligence Institute,
Curtin University of Technology, Perth, Australia
E-mail: {p.wongthongtham, n.kasisopha, e.chang, t.dillon}@curtin.edu.au

Abstract

This paper aims to present software engineering ontology as software engineering knowledge representation for a multi-site software development. It will not only facilitate the capturing of software engineering knowledge but also enhance the sharing of software engineering knowledge across geographically multiple software development sites. The software engineering ontology assists in defining information for the exchange of semantic project data and is used as a communication framework. Its end users are software engineers sharing software engineering domain knowledge as well as software engineering project data.

1. Introduction

With the advent of the Internet, software development has increasingly focused on the Internet which enables a multi-site environment that allows multiple teams residing across cities, regions, or countries to work together in a networked distributed fashion to develop the software. A realisation of the advantages of multi-site software development has led to major corporations moving their software development to countries where employees are on comparatively lower wages. It is this imperative of financial gain that drives people and businesses to multi-site development and the Internet which facilitates it.

However, the globalization of software development means that the problems of multi-site development are increasing. Team members who carry out the tasks and activities, team leaders who control the tasks and activities, and managers who manage the project and leaders, may or may not be at the same site in a multi-site environment. These people often have never met face-to-face, have different cultural and educational backgrounds, interpret methods in different ways, etc.

Additionally, software engineering training and practice are quite different between cities and countries. It can be difficult to communicate between teams and among team members, if strict software engineering principles and discipline are not understood and followed. The inconsistency in presentation, documentation, design and diagrams could prevent access by other teams or members. Sometimes, these issues (such as a diagram using non-standard notation) are ignored because they are not understood and no-one asks for clarification.

Despite this, software engineering has a commonly understood body of knowledge and is an easily learnt subject that includes some of the latest technology and methodology which is easily adopted. However, different teams could be referring to different texts on software engineering. Teams or team members use a particular text as their own individual guide, and when they communicate, their own knowledge base and terminology is different from others. Often, the issues raised or debated are related to inconsistency in understanding software engineering theories and practice.

Consequently, several practical problems arise and underlying issues need to be explored. Communication is the real challenge that we face and that we need different ways of tackling this through better communication and conferencing systems and through systems that help resolve differences between the teams. Ontology is an important part of developing a shared understanding across a project. As Davenport and Prusak [1] mentioned, people cannot share knowledge if they do not speak a common language. Representing software engineering knowledge in the form of ontology is helping to clear up ambiguities in the terms used in the context of software engineering.

978-0-7695-3407-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ICCIT.2008.301

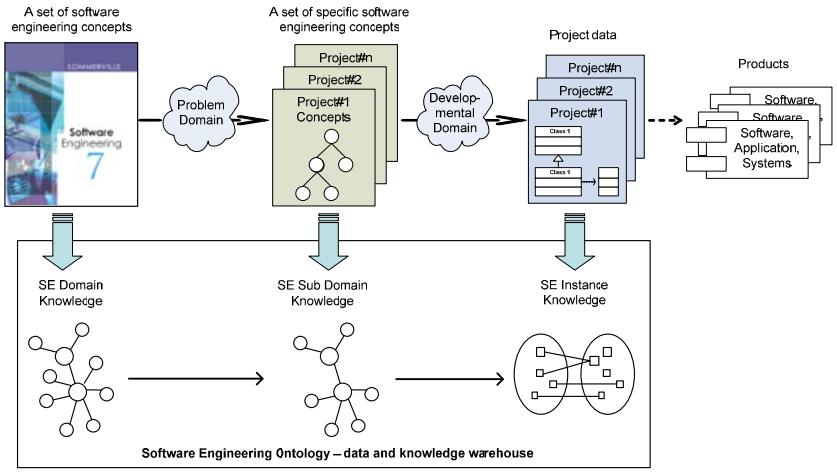

Fig. 1. Overview of software engineering ontology as data and knowledge warehousing.

2. Ontology vs Knowledge Representation

The term 'Ontology' is derived from its usage in philosophy where it means the study of being or existence as well as the basic categories [2]. Therefore, in this field, it is used to refer to what exists in a system model.

An ontology, in the area of computer science, is the effort to formulate an exhaustive and rigorous conceptual schema within a given domain, typically a hierarchical data structure containing all the relevant elements and their relationships and rules (regulations) within the domain [3].

An ontology, in the artificial intelligence field, is an explicit specification of a conceptualisation [4, 5]. In such an ontology, definitions associate the names of concepts in the universe of discourse (e.g. classes, relations, functions) with describing what the concepts mean, and formal axioms that constrain the interpretation and well-formed use of these terms [6].

For example, by default, all computer programmes have a fundamental ontology consisting of a standard library in a programming language, or files in accessible file systems or some other list of 'what exists'. However, the representations are sometimes poor for some certain problem domains, so more specialised schema must be created to make the information useful and for this we utilise ontology.

An abstract view of representing the software engineering knowledge is shown in Figure 1. The whole set of software engineering concepts representing software engineering domain knowledge is captured in ontology. Based on a particular problem domain, a project or a particular software development probably uses only part of the whole set of software engineering concepts. The specific software engineering concepts used for the particular software development project representing software engineering sub-domain knowledge are also captured in ontology. The generic software engineering knowledge represents all software engineering concepts, while specific software engineering knowledge represents some concepts of software engineering for the particular problem domain. For example, if a project uses purely object-oriented methodology, then the concept of a data flow diagram may not be necessarily included in specific concepts. Instead, it includes concepts like class diagram, activity diagram and so on. For each project in the developmental domain, there exists project data or actual data including project agreements and project understanding. The project data especially meets a particular project need and is needed with the software engineering knowledge to define instance knowledge in ontology. Note that the domain knowledge is separate from instance knowledge. The instance knowledge varies depending on its use for a particular project and is diverse according to project requirements, feasibility, etc. in each remote distributed team. The domain knowledge is quite definite, while the instance knowledge is particular to problem domain and developmental domain in a project. Once all domain knowledge, sub domain knowledge and instance knowledge are captured in ontology, it is available for sharing among remote software engineers through the internet. All team members, regardless of where they are, can query the semantically linked

the semantically linked project data and use it as the common communication and knowledge basis of raising discussion matters, questions, analysing problems, proposing revisions or designing solutions and the like.

Software engineering domain knowledge constructs should be sought in ontology, a well founded model of reality. Ontology is used to analyse the meaning of common conceptual modelling constructs [7] which accurately reflect the world. The notion of a concrete thing applies to what software engineers perceived based on software engineering domain knowledge. In this light, the notion of ontology is a solution for software engineering knowledge representation.

When the knowledge of the software engineering domain is represented in a declarative formalism, the set of software engineering concepts, their relations and their constraints are reflected in the representation which represents knowledge. Thus, the software engineering ontology can be defined by using a set of software engineering representational terms. Then a conclusion from the knowledge of what is can be determined.

In order for the software engineering domain knowledge to be shared amongst software engineers or applications, agreement must exist on the topics about which information is being communicated. The issue of ontological commitment is described as the agreement about concepts and relationships between those concepts within ontology [5]. When the software engineering ontology is committed, it means agreement exists with respect to the semantics of the concepts and relationships represented. Therefore, in order to know what the software engineers are talking about, agreement is arrived at. The software engineers agree to share knowledge in a coherent and consistent manner.

The software engineering ontology is organised by concepts, not words. This is in order to recognise and avoid potential logical ambiguities. The software engineering ontology has been developed for communication purposes, thus, it could differ greatly from other ontologies developed for different purposes. The main purpose of the software engineering ontology is to enable communication between computer systems or software engineers in order to understand common software engineering knowledge and to perform certain types of computations. The key ingredients that make up the software engineering ontology are a vocabulary of basic software engineering terms and a precise specification of what those terms mean. For software engineers or computer systems, different interpretations interpretations in different contexts can make the meaning of terms confusing and ambiguous but a coherent terminology adds clarity and facilitates a better understanding. Software engineering ontology has specific instances for the corresponding software engineering concepts. These instances contain the actual data being queried in the knowledge-based applications. The software engineering ontology includes the set of actual data or instances of the concepts and assertions that the instances are related to each other according to the specific relations in the concepts. The main purpose of the software engineering ontology is for enabling knowledge sharing and reuse. In this sense, the software engineering ontology is a specification used for making ontological commitments. In practice, an ontological commitment is an agreement that is consistent and coherent with respect to theory specified by the software engineering ontology.

3. Fundamentals for Modeling Software Engineering Domain

Software engineering ontology is like other ontologies in other domains which consist of instances, properties and classes. Software engineering ontology consists of instances representing specific project data, properties representing binary relations held among software engineering concepts/instances, and classes representing the software engineering concepts interpreted as sets that contain specific project data. The software engineering ontology classes are built up of software engineering concepts' descriptions that specify the conditions that must be satisfied by project data in order for it to be a member of the classes.

The relationships between classes or instances represented by data type property and object property come from two different sources in software engineering ontology. Data type property associates classes or instances to an XML schema data type value or an RDF literal. Object property associates a class to a class or an instance to an instance. Association between class and property does not always generate the representation of a class as a bundle of owned properties. In other words, software engineering ontology classes have no owned software engineering properties. They are independent of each other. Software engineering ontology properties may have sub-properties and it is possible to form hierarchies of properties like classes. Sub-classes specialise their super-classes in the same way that sub-properties specialise their super-properties. The relationships among classes are binary and have distinctive

beginnings and ends. Object properties link ontology class from the domain to ontology class from the range. Data type properties link ontology class from the domain to an XML schema data type value from the range. It is possible to specify multiple classes as the domain or the range for a property. If multiple classes are specified, the domain or the range of the property is understood to be the union of the classes. Software engineering ontology supports a fixed defined extent for an ontology class. It is used to define a class description of the enumeration kind. Software engineering ontology also supports fixed defined data values of data range for a data type property. It specifies the set of data values of the data range. In software engineering ontology a property is defined by default as having range and domain and both range and domain can apply to any class in the software engineering ontology. The scope of the property does not limit and attach to the classes on which it is defined.

There is no direct linkage between association and software engineering ontology class. The linkage is mediated by a software engineering ontology property. Most of the software engineering ontology classes normally have software engineering ontology properties, although this is not always true. Software engineering ontology property may or may not be owned by one or more software engineering ontology classes. The property can even remain by itself without the classes.

Software engineering ontology enriches the meaning of properties through the use of property characteristics as do other ontologies in other domains. The first characteristic is functional properties which have a maximum cardinality of one on its range. Another property characteristic is inverse functional properties which have a maximum cardinality of one on its domain. Software engineering ontology allows properties to be declared symmetric or transitive. Software engineering ontology properties are used to create restrictions which restrict the instances that belong to a class. An ontology property can have its range restricted when the property is applied to the domain class, either that the range is limited to a class only (allValueFrom) or that the range is one part of a class (someValueFrom). Notice that in allValueFrom restrictions, the range would not have been related with other classes apart from a specified class. In software engineering ontology, an ontology property can be constrained by cardinality restrictions on the domain giving the minimum (minCardinality), maximum (maxCardinality), or exact (cardinality) specified number of instances which can participate in the relation. A hasValue restriction describes the set of instances that have at least one relation along a specified property to a specific instance.

4. The Software Engineering Ontology

A process of design in the software engineering ontology refers to the process of design concepts, concepts hierarchy, relations, and constraints in the software engineering domain. Sources of software engineering knowledge are from the software engineering textbook of Ian Sommerville [8] and the Software Engineering Body of Knowledge (SWEBOK) [9] upon which we base our design. The software engineering ontology contains 362 concepts and 303 relations. Figure 2 shows overview of a part of software engineering ontology illustrating software engineering concepts construction.

Due to limited space, we will illustrate the design by choosing some specific examples of common widely used concepts i.e. entity diagram and activity diagram in this section. First example is an entity-relationship diagram which represents conceptual models of data stored in information systems [10]. In an ontology model of entity-relationship diagrams, there are three main basic components in the entity-relationship diagrams i.e. entity, attributes, and relationships which form three ontology classes i.e. *Entities* class, *Entity_Attributes* class and *Entitiy_Relationships* class respectively. *Entity_Attributes* class can be classified as being simple (i.e. *Simple_Entity_Attribute* class), composite (i.e. *Composite_Entity_Attribute* class) or derived (i.e. *Derived_Entity_Attribute* class). A simple attribute is composed of a single component and a composite attribute is composed of multiple components. In the ontology model, cardinality restriction in relation between *Entity_Attribute* classes defines attributes as being either simple or composite. A derived attribute is based on another attribute(s) and refers to relation *has_Derived_Attribute* restricting at least one relation. Key can be defined as attributes of super key, alternate key, primary key, or candidate key. This refers to relation *Entity_Attribute_Key* in the ontology model and restricts to one of super key, alternate key, primary key, or candidate key. An attribute can have a single or greater-than-one value. In the ontology model, cardinality restriction from relation *Entity_Attribute_Value* defines having a single or greater-than-one value. There are three main degrees of relationships which are unary (i.e. *Unary_Entity_Relationship* class), binary (i.e. *Binary_Entity_Relationship* class), and complex (i.e. *Complex_ Entity_Relationship* class). The complex

relationship can be further divided into quaternary (i.e. *Quaternary_Entity_Relationship* class) and ternary (*Ternary_Entity_Relationship* class. In the ontology model, cardinality restriction constrains the number of entities that participate in a relationship. For example, a unary relationship represents a relationship of one entity or, more precisely, that entity is self-linked. This means that in the ontology view there is only one *Entity* in the relation *Relating_Entity* and no *Entity* in the relation *Related_Entity*. In an entity relationship, cardinality can be specified as string which can be a string of 1 (one and only one), * (zero or more), 1..* (one or more), 0..1 (zero or one) and so forth as shown in the ontology model. Attributes can also be assigned to relationships referring to relation *has_Attribute_on_Relationships* in the ontology model.

An activity diagram shows the control flow from activity to activity [10]. Mainly, activity diagrams contain activities, transitions, swimlane, and objects forming ontology classes of *Activity*, *Transition*, *Swimlane*, and *Object* respectively. A locus of activities is specified by a swimlane. This refers to relation *in_Swimlane* in the ontology model. Every activity belongs to exactly one swimlane; however, transition may make it cross lanes. This means maximum cardinality restriction in relation *in_Swimlane*. Objects may be involved in the flow of control associated with an activity diagram. This refers to relations *set_Object_Flow* and its inverse, *get_Object_Flow*. Transitions of activities are classified into four main transitions. Firstly, normal transition (i.e. *Normal_Transition* class) shows the path from one activity to the next activity. This means that, ontology class *Normal_Transition* that has a cardinality restriction, restricts only the one activity in the relations *Related_Activity* and *Relating_Activity*. Secondly, special transition (i.e. *Special_Transition* class) is further divided into an initial transityion (i.e. *Start_Transition* class) and a stop transition (*Stop_Transition* class). The initial transition is where the activity diagrams start. This means that, class *Start_Transition* has a cardinality restriction and restricts at least one activity in relation *Related_Special_Activity* but no activity in relation *Relating_Special_Activity*. The stop transition is where the activity diagrams stop. This means that class *Stop_Transition* which has a cardinality restriction, restricts at least one activity in relation *Relating_Special_Activity* but no activity in relation *Related_Special_Activity*. Thirdly, branch transition which specifies alternate paths taken based on some guard expression refers to ontology *Branch_Transition* class. Lastly, concurrent transition (i.e. *Concurrent_Transition* class) is further divided into a fork transition (i.e. *Fork_Transition* class) and a join transition (*Join_Transition* class). The fork transition represents the splitting of a single flow of control into two or more flows of control. This means that ontology class *Fork_Transition,* that has a cardinality restriction, restricts at least two activities in relation *Related_Concurrent_Activity* and only one activity in relation *Relating_Concurrent_Activity*. The join transition represents the joining of two or more incoming transitions and one outgoing transition. This means that ontology class *Join_Transition,* which has cardinality restriction, restricts at least two activities in relation *Relating_Concurrent_Activity* and only one activity in relation *Related_Concurrent_Activity*.

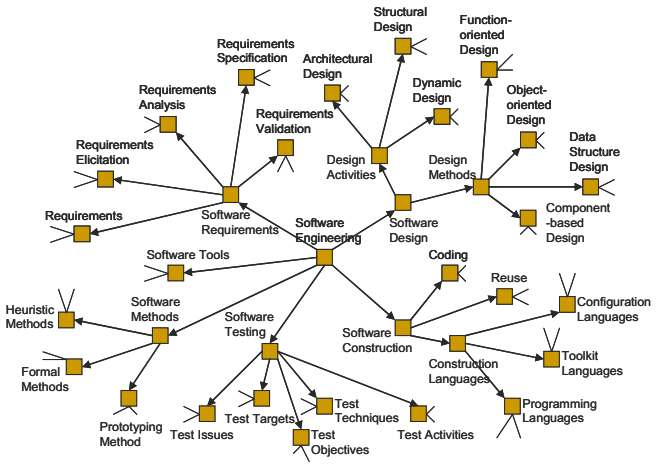

Fig. 2. Overview of a part of software engineering ontology.

5. Software Engineering Ontology as Knowledge Representation

In this section, we illustrate how software engineering ontology represents software engineering knowledge to facilitate the communication framework in multi-site software development environment. Software engineering ontology presents explicit assumptions concerning the objects referring to the domain knowledge of software development. A set of objects and interrelations and their constraints renders their agreed meanings and properties. Knowledge / Data warehousing through the software engineering ontology eliminates misunderstandings, miscommunications and misinterpretations. For example to represent knowledge of an activity diagram shown in Figure 3, following is a list of actions. Note that the activity diagram used as example here is derived from the book of Enterprise Java with UML [11].

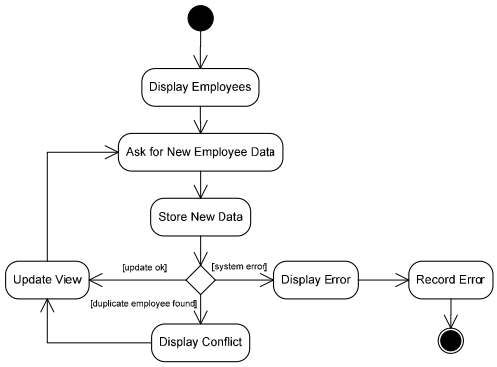

Fig. 3. An Activity diagram.

1. Adding new instances of class *Activity* named 'Display Employees', 'Ask for New Employee Data', 'Store New Data', 'Display Error', 'Record Error', 'Update View', and 'Display Conflict'.
2. Adding new instance of class *Start_Transition* relating relation *Related_Activity* with instance of class *Activity* named 'Display Employees'.
3. Adding new instance of class *Activity_Transition* relating relation *Relating_Activity* with instance of class *Activity* named 'Display Employees' and relating relation *Related_Activity* with instance of class *Activity* named 'Ask for New Employee Data'.
4. Adding new instance of class *Activity_Transition* relating relation *Relating_Activity* with instance of class *Activity* named 'Ask for New Employee Data' and relating relation *Related_Activity* with instance of class *Activity* named 'Store New Data'.
5. Adding new instance of class *Branch_Transition* relating relation *Relating_Activity* with instance of class *Activity* named 'Store New Data' and relating relation *Related_Activity* with instances of class *Activity* named 'Update View', 'Display Conflict', and 'Display Error'.
6. Adding new instance of class *Activity_Transition* relating relation *Relating_Activity* with instance of class *Activity* named 'Display Error' and relating relation *Related_Activity* with instance of class *Activity* named 'Record Error'.
7. Adding new instance of class *Stop_Transition* relating relation *Relating_Activity* with instance of class *Activity* named 'Record Error'.

Warehousing project data drawn based on a consensus of domain knowledge of software engineering formed in the software engineering ontology, makes information explicit. Having attached domain knowledge, it makes project data more understandable, linear, predictable and controllable. Users learn about some missing pieces that make sense of the attentive interaction among users. Alarms can be activated when there are some missing pieces while sharing project data.

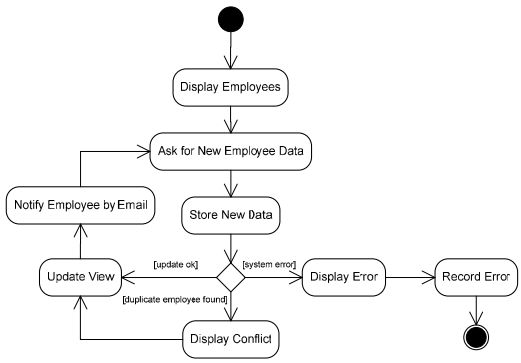

Fig. 4. An updated Activity diagram.

Warehousing software engineering knowledge and project data formed into software engineering ontology facilitates communication framework among software engineers and provides consistent understanding of the domain knowledge. For example, one would like to communicate changes of project design of the activity

diagram shown in Figure 3. Figure 4 shows an updated activity diagram. As can be noted when comparing Figure 3 and Figure 4, the software engineer has revised the transition of activity 'Update View'. Originally, activity 'Update View' transited to activity 'Ask for New Employee Data'. Revision has been made by activity 'Update View' transited to activity 'Notify Employee by Email' and activity 'Notify Employee by Email' transited to activity 'Ask for New Employee Data'. Functioning is as follows:

1. Delete instance of class *Normal_Transition* that has relation *Related_Activity* with instance of class *Activity* named 'Ask for New Employee Data' and has relation *Relating_Activity* with instance of class *Activity* named 'Update View'.
2. Add new instance of class *Activity* named 'Notify Employee by Email'.
3. Add instance of class *Normal_Transition* that links relation *Related_Activity* with instance of class *Activity* named 'Notify Employee by Email' and links relation *Relating_Activity* with instance of class *Activity* named 'Update View'.
4. Add instance of class *Normal_Transition* that links relation *Related_Activity* with instance of class *Activity* named 'Ask for New Employee Data' and links relation *Relating_Activity* with instance of class *Activity* named 'Notify Employee by Email'.

This example shows that a user can communicate about any project data that is captured as ontology instances. The design of an activity diagram is captured, and adheres to the concept of the UML activity diagram in the software engineering domain knowledge captured as software engineering ontology. This enables a meaningful communication about the design of activity diagram. Activity diagrams, statechart diagrams and state transition diagrams are related, thereby sometimes causing confusion. While a statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an activity diagram focuses on the flow of activities involved in the process. The activity diagram shows how these activities depend on one another. Conclusively, in determining what concept of project information is captured (statechart diagrams or activity diagrams) or where that project data resides (statechart diagrams or activity diagrams), it is assumed that this is determined by the member who specifies what the project data really means in the context. Once users are committed to the domain knowledge of activity diagrams and recognise that it is mainly constituted of activity and activity transitions and constraint attached, the commitment enables people to discuss the same topic (the topic of design of activity diagram). Consequently, people can coordinate their activities.

6. Conclusion

Software engineering knowledge and project data, formed into software engineering ontology, helps communications among remote team members and provides consistent understanding of the domain knowledge and project data. Software engineering ontology, together with its instance knowledge, is used as a communication framework within a project, thereby providing rational and shared understanding of project matters.

In this paper, we have analysed software engineering ontology as knowledge and data warehousing. We have presented the software engineering ontology. We have only covered some distinguished part of modelling domain knowledge of software engineering as example. Deployment has been discussed in aspects of knowledge and data warehousing and communication framework. However, there are many improvements that can be made through future work. Future work could consider software engineering ontology evolution. It is the case of software engineering domain knowledge changing with the introduction of new concepts, and change in the conceptualisation as the semantics of existing terms have been modified with time. This is totally outside the scope of this study because we assume that software engineering domain knowledge is mature and has undergone no further changes. Instead, instantiations in the software engineering ontology change with corresponding changes to the ontology.

7. References

1. Davenport, T.H. and L. Prusak, *Working Knowledge: How Organisations Manage What They Know*. 1998, Boston, MA: Harvard Business School Press.
2. Witmer, G. *Dictionary of Philosophy of Mind - Ontology*. 2004 [cited May 11, 2004]; Available from: http://www.artsci.wustl.edu/~philos/MindDict/ontology.html.
3. Wikipedia. *Ontology (computer science) From Wikipedia, the free encyclopedia*. 2006 [cited 8 June 2006]; Available from:

http://en.wikipedia.org/wiki/Ontology_%28computer_science%29.
4. Gruber, T.R. *A translation approach to portable ontology specification*. in *Knowledge Acquisition*. 1993.
5. Gruber, T.R. *Toward principles for the design of ontologies used for knowledge sharing*. in *International Workshop on Formal Ontology in Conceptual Analysis and Knowledge Representation*. 1993. Padova, Italy: Kluwer Academic Publishers, Deventer, The Netherlands.
6. Beuster, G. *Ontologies Talk given at Czech Academy of Sciences*. 2002 [cited; Available from: http://www.uni-koblenz.de/~gb/papers/2002_intro_talk_ontology_bang/agent_ontologies.pdf.
7. Wand, Y., V.C. Storey, and R. Weber, *An Ontological Analysis of the Relationship Construct in Conceptual Modeling*. ACM Transactions on Database Systems, 1999. **24**(4): p. 495-528.
8. Sommerville, I., *Software Engineering*. 8th ed. 2004: Pearson Education Limited.
9. Bourque, P. *SWEBOK Guide Call for Reviewers*. 2003 [cited 29 May 2003]; Available from: http://serl.cs.colorado.edu/~serl/seworld/database/3552.html.
10. Bourque, P., et al. *Guide to the Software Engineering Body of Knowledge*. 2004 Feb. 16, 2005 [cited.
11. Arrington, C., *Enterprise Java with UML*. 2001, New York, USA: John Wiley & Sons, Inc.