Development of a Software Engineering Ontology for Multi-site Software Development

Pornpit Wongthongtham, Elizabeth Chang, Tharam Dillon and Ian Sommerville

Abstract - This paper aims to present an ontology model of software engineering to represent its knowledge. The fundamental knowledge relating to software engineering is well described in the textbook titled Software Engineering by Ian Sommerville that is now in its eighth edition [1] and the white paper, SoftWare Engineering Body Of Knowledge (SWEBOK), by the IEEE [2] upon which software engineering ontology is based. This paper gives an analysis of what software engineering ontology is, what it consists of and what it is used for in the form of usage example scenarios. The usage scenarios presented in the paper highlight characteristics of the software engineering ontology. The software engineering ontology assists in defining information for the exchange of semantic project information and is used as a communication framework. Its end users are software engineers sharing domain knowledge as well as instance knowledge of software engineering.

Index Terms—Software Engineering, Ontology Development, Multi-site Software Development

1 Introduction

Having realized the advantages of multi-site software development, major corporations have moved 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. 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.

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 located at the same site in a multi-site environment. It is quite often the case that these people have never met face-to-face, have different cultural and educational backgrounds, interpret methods in different ways, etc.

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 with 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 those of 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 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 [3] 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. Recent events such as the WOMSDE (Ontologies and Metamodeling in Software and Data Engineering) workshop, ONTOSE (Ontology, Conceptualization and Epistemology for Software and Systems Engineering) and SWESE (Semantic-Web Enabled Software Engineering) have focused on ontologies in software engineering, demonstrating that ontologies are becoming increasingly important in the area of software engineering as they provide a critical semantic foundation.

- *P.W. is with Digital Ecosystems and Business Intelligence Institute, Curtin University of Technology, Australia. E-mail: ponnyl@gmail.com*
- *E.C. is with Digital Ecosystems and Business Intelligence Institute, Curtin University of Technology, Australia. E-mail: e.chang@curtin.edu.au*
- *T.D. is with Digital Ecosystems and Business Intelligence Institute, Curtin University of Technology, Australia. E-mail: t.dillon@curtin.edu.au.*
- *I.S. is with St Andrews University, Scotland. E-mail: ifs@cs.st-and.ac.uk*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

2 Ontology in Software Engineering

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 [4]. Therefore, in this field, it is used to refer to what exists in a system model.

An ontology, in the area of computer science, represents 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 [5].

An ontology, in the artificial intelligence field, is an explicit specification of a conceptualisation [6, 7]. In such an ontology, definitions associate the names of concepts in the universe of discourse (e.g. classes, relations, functions) with a description of what the concepts mean, and formal axioms that constrain the interpretation and well-formed use of these terms [8].

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 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 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 necessarily be 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 information or actual data including project agreements and project understanding. The project information 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. The domain knowledge is quite definite, while the instance knowledge is particular to the 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 software engineers through the Internet. All team members, regardless of their location, can query the semantically linked project information and use it as the common communication and knowledge basis for raising discussion matters, questions, analysing problems, proposing revisions or designing solutions and the like.

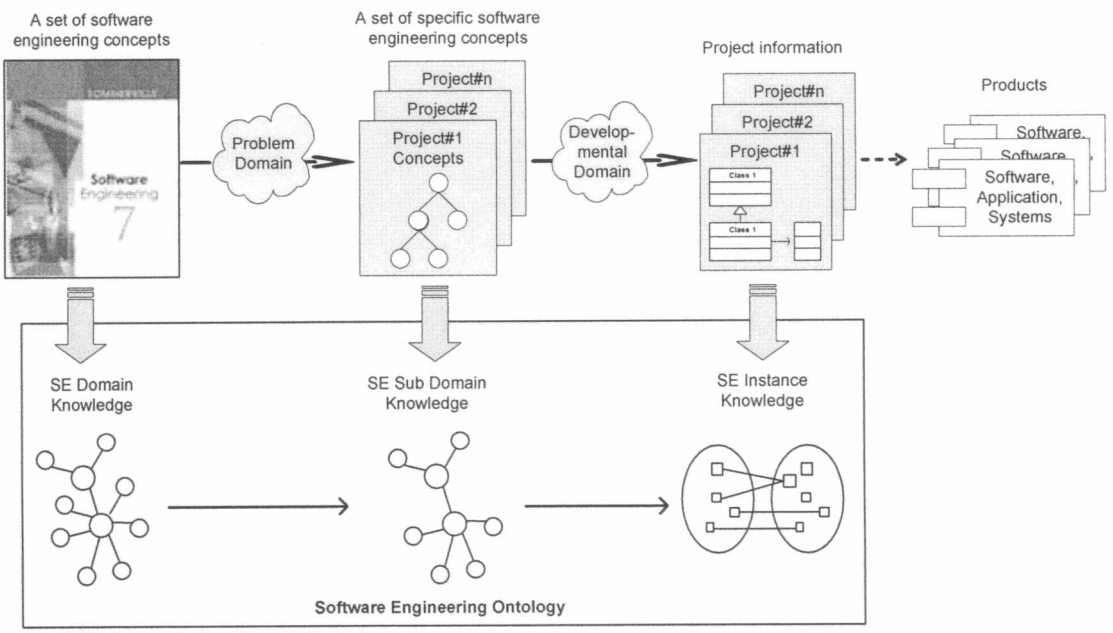


Fig. 1. Schematic overview of software engineering knowledge representation

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 [9] which accurately reflect the world. The notion of a concrete thing applies to what software engineers perceive 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 [7]. 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 may 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 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 to enable 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 The Software Engineering Ontology

There are several ontology languages available such as Resource Description Framework (RDF) [10], Web Ontology Language (OWL) [11], DARPA Agent Markup Language (DAML) [12], Ontology Interchange Language (OIL) [13], DAML+OIL [14], Simple HTML Ontology Extensions (SHOE) [15] etc. for capturing knowledge of interest. Different ontology languages have different facilities. The most recent development in standard ontology languages is OWL from the World Wide Web Consortium (W3C) (http://www.w3.org/). It has the most complete set of expressions for capturing the different concepts and relationships that occur within ontologies; therefore, the software engineering knowledge is captured in OWL.

Software engineering ontology is like other OWL 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 constructed of descriptions of software engineering concepts that specify the conditions that must be satisfied by project data in order for it to be a member of the classes.

The OWL ontology fundamentals for modelling the software engineering domain include ontology class and subclasses, ontology properties (data type property and object property), association between ontology class and ontology property, property characteristics, constraints or restrictions, and ontology instances. Details of these components of OWL ontologies can be found in various literatures in [11, 16, 17].

4 Software Engineering Ontology Modelling

Various formalisms have been developed for modelling ontologies, notably the Knowledge Interchange Format (KIF) [18] and knowledge representation languages descended from KL-ONE [19]. However, these representations have had little success outside Artificial Intelligence (AI) research laboratories [20, 21] and require a steep learning curve. KIF provides a Lisp-like syntax to express sentences of first order predicate logic and descendants of KL-ONE include description logics or terminological logics that

provide a formal characterisation of the representation [22]. Traditionally, AI knowledge representation has a linear syntax. There have been recent efforts, documented in the literature [23-25], to use UML for ontology modelling. In UML, ontology information is modelled in class diagrams and Object Constraint Language (OCL) constraints [24]. However, there is controversy regarding whether or not ontology goes beyond the standard UML modelling, so that standard UML cannot express advanced ontology features such as constraints or restrictions, and [26] cannot totally conclude whether the same property was attached to more than one class, and does not support the creation of a hierarchy of properties. Therefore, additional notations need to be defined in order to leverage expressiveness in the ontology. Note that the models underlying ontology should be distinguished from its use in software development to model the application domain model. This kind of agile modelling method for ontology design has some benefits derived from using the same paradigm for modelling ontology and knowledge.

In this study, graphical notations of modelling software engineering ontology as an alternative formalism are presented. The main aim is not only to create a graphical representation to make it easier to understand, but importantly, this model should be able to capture the semantic richness of the defined software engineering ontology. In the next sections, graphical notations are presented to facilitate the software engineering ontology modelling process. Some concepts or terms represented by the notations have multiple presentations.

4.1 Class Notations

The notation of the software engineering ontology class is represented as a rectangle with two compartments. The top compartment is for labelling the class and the second compartment is used for presenting properties, if there are any, related to the class or to an XML schema data type value. It is mandatory to specify the word '<<Concept>>' above the class label in the top compartment. The generalisation symbol appears as a line with one end empty and the other with a hollow triangle arrowhead. The empty end is always connected to the class being subsumed, whereas the hollow arrowhead connects to the class that subsumes. Figure 2 shows an example of a class *ClassRelationship* presentation and its sub-classes.

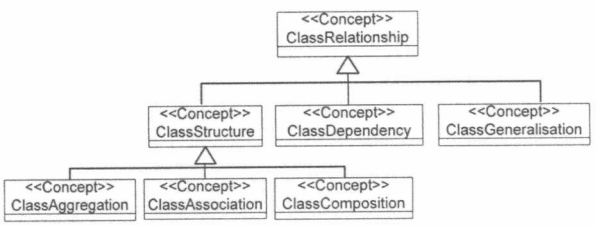


Fig. 2. Class ClassRelationship presentation and its sub-classes

4.2 Property Relation, Characteristic and Restriction Notations

Notations for software engineering ontology properties include data type property, object property, its characteristics, its restrictions, and association class attached property.

The data type property of the software engineering ontology class can be expressed in the bottom compartment of class notation. This is an alternative design for data type properties. That means the top compartment is called the 'domain'. In the bottom compartment, notation formats as in the order of data type property name, its characteristic, its type (e.g. String, Integer) and its restriction. The type of data type property is considered as a range. The characteristics of data type property can be either functional or non-functional represented by the words 'Single' or 'Multiple' respectively. An enumeration in software engineering ontology is represented in curly brackets ({}). Figure 3 expresses that class *ClassOperation* has data type property *Class_Operation_Name* which is functional and its type is String. It also defines functional data type property *Class_Operation_Visibility* to relate a set of data values of 'public', 'private' and 'protected'.

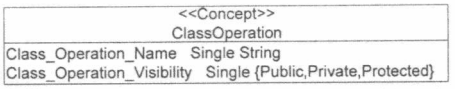


Fig. 3. Class ClassOperation presentation and its data type properties

The object property of the software engineering ontology can be expressed in the bottom compartment of class notation like data type properties. This is an alternative design for object property. In this manner, the top compartment is still called its domain. Notation format in the bottom compartment is the same as the order of the object property's name, its characteristics, class name (its range), and its restriction. Class name expression as a range can assert the complex class description such as union (represented by symbol 'U'), intersection (represented by symbol '∩'). In addition, an object property can be expressed as an arrow with an open arrowhead and with a text label of the object property's name. This is an alternative design for object properties. The arrow points from the domain of property to the range of property. Its restrictions can be expressed in the bracket after its name.

The characteristics of object property can be functional, inverse functional, non-functional, symmetric or transitive; represented by the words Single, Inverse, Multiple, Symmetric or Transitive respectively. Alternatively, functional property notation is represented as an open arrowhead with a text label of the property name and symbol number 1 near the arrowhead, while a non-functional property notation is represented without a number 1 near the arrowhead. However, at the arrowhead, the cardinality restriction can be presented near the arrowhead. Alternatively, inverse functional property notation is represented as an

open arrowhead with the arrowhead pointing in the opposite direction to its inverse property notation and a dot line placed in the middle which links the inverse functional property symbol and its inverse. Alternatively, symmetric property notation is represented simply as a solid line with a text label of the property name. Alternatively, a transitive property notation is represented as a dotted arrow with an open arrowhead and with a text label of its property name. Figure 4 shows class *Class* and its properties.

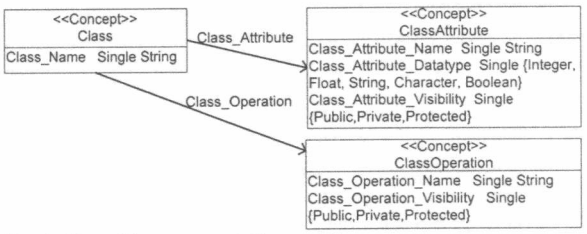


Fig. 4. Class Class presentation and its properties

Symbols∀, ∃, and ▯ represent restrictions allValueFrom, someValueFrom, and hasValue respectively. For the cardinality restriction, symbols equal =, greater than and equal to ≥ and less than and equal to ≤ respectively represent cardinality specifying the exact number, minimum cardinality specifying the minimum number and maximum cardinality specifying the maximum number. Cardinality constraints can be placed with the format of x..y where x is the value of minimum cardinality and y is the value of maximum cardinality. The asterisk * is used as part of the specification to indicate the unlimited upper bound. Figure 5 (a) shows that at least two relations *Relating_Activity* relate instance from class *JoinTransition* to class *Activity* and the property *Related_Activity* restricts instance from class *JoinTransition* to exactly one instance of class *Activity*. In other words, in join transition (from activity diagram) there are at least two related activities transited into one related activity as shown in Figure 5 (b).

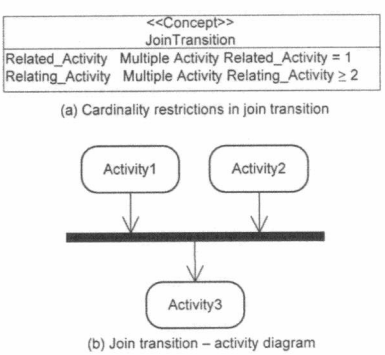


Fig. 5. Class *JoinTransition* presentation and its properties

Association class can be used to participate in further associations for property in software engineering ontology. To specify association classes, a dotted line is used to attach the property notation to the class notation. Figure 6 shows class *ClassAggregation* which is an association class of object property that links class *Class* together. Association class *ClassAggregation* can participate in further associations with bundles of properties.

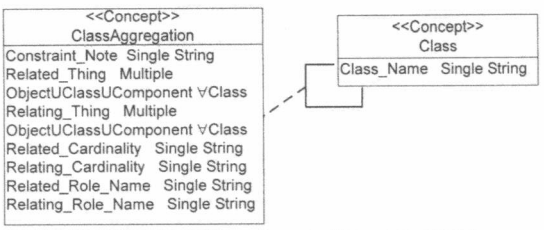


Fig. 6. Association class *ClassAggregation* presentation

4.3 Instance Notation

An instance notation is represented as an ellipse with a dotted line attached to its class or property. If it is an instance of property, then the ellipse contains the property name followed by a colon and then the instance name. Unlike an instance of class, in the ellipse there is only the instance name. To make it easy to read, a dotted line is attached to most of the class name or property name of its class instance or property instances. Figure 7 shows populating the UML class diagram shown in Figure 7 (a) into the ontology model shown in Figure 7 (b) as instances.

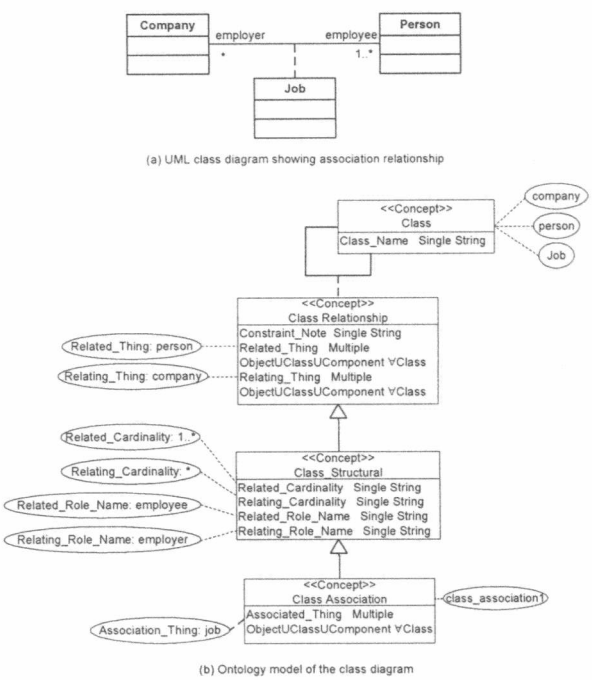


Fig. 7. Presentation of instances of classes and properties

5 Software Engineering Ontology Design and Implementation

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 [1] and the Software Engineering Body of Knowledge (SWEBOK) [2] upon which we base our design. The software engineering ontology contains 362 concepts and 303 relations.

Due to limited space, in this section we will illustrate the design by choosing some specific examples of common, widely-used concepts such as UML class diagram, UML activity diagram, UML use case diagram, entity diagram.

5.1 Ontology Model for a UML Class Diagram

The first example is a UML class diagram ontology. A class diagram is a diagram that represents a set of classes and their interrelationships [27]. Commonly, the structural details of classes expand their attributes and their operations. The relationships within these five main classes are: dependencies, generalisations, aggregations, compositions, and associations. The relationships of aggregations, composition and association are used to structure a class, so in the ontology model of class diagrams, they are grouped together. Figure 8 shows the UML class diagram ontology.

5.2 Ontology Model for an Entity-Relationship Diagram

An entity-relationship diagram represents conceptual models of data stored in information systems [27]. Figure 9 shows an ontology model of entity-relationship diagrams. There are three main basic components in the entity-relationship diagrams which are entities, attributes and relationships. Entity attributes can be classified as being simple, composite or derived. 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 *has_Subdivided_Attribute* 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 link to ontology class *EntitiyAttribute*. 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. 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.

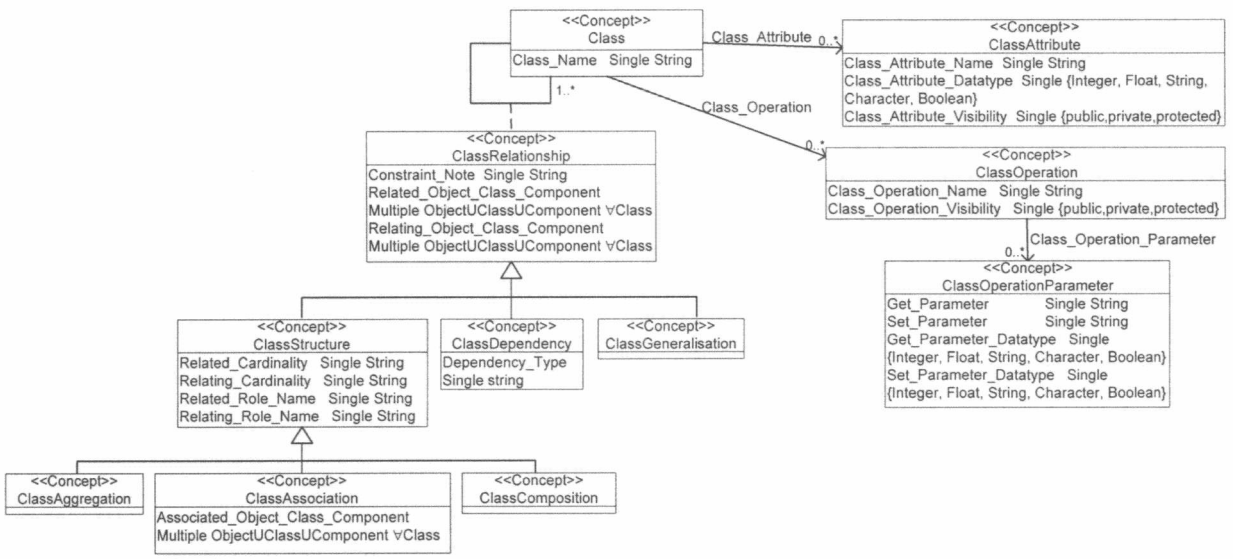


Fig. 8. UML class diagram ontology

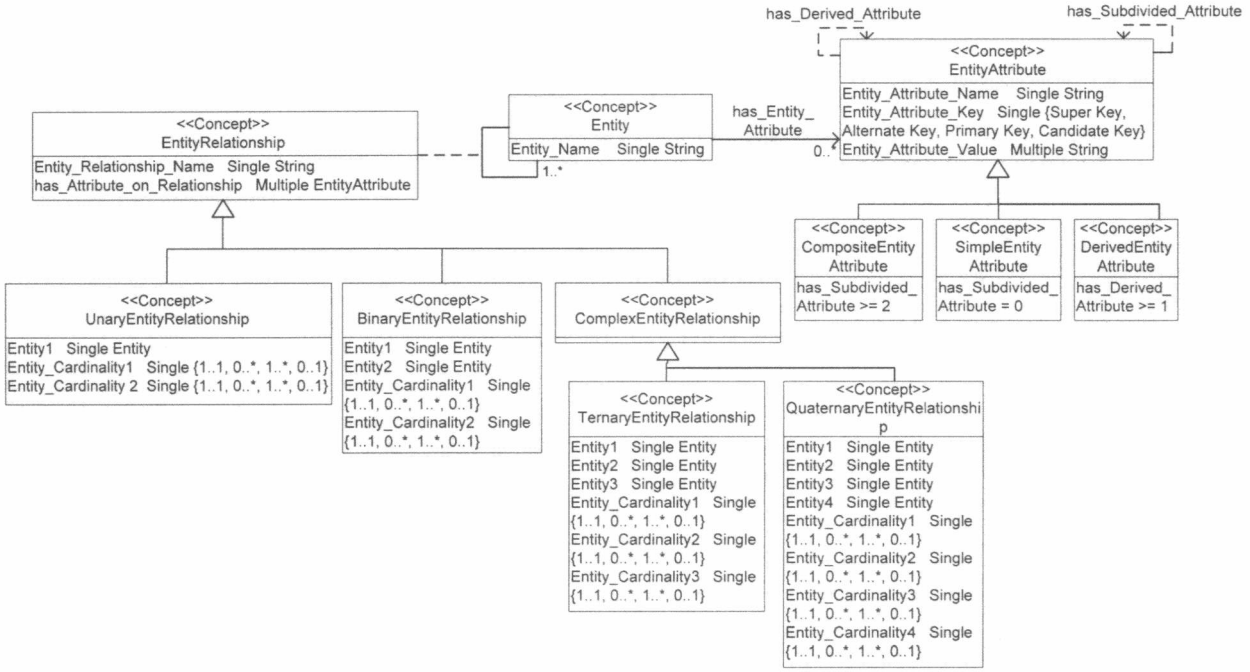


Fig. 9. Entity diagram ontology

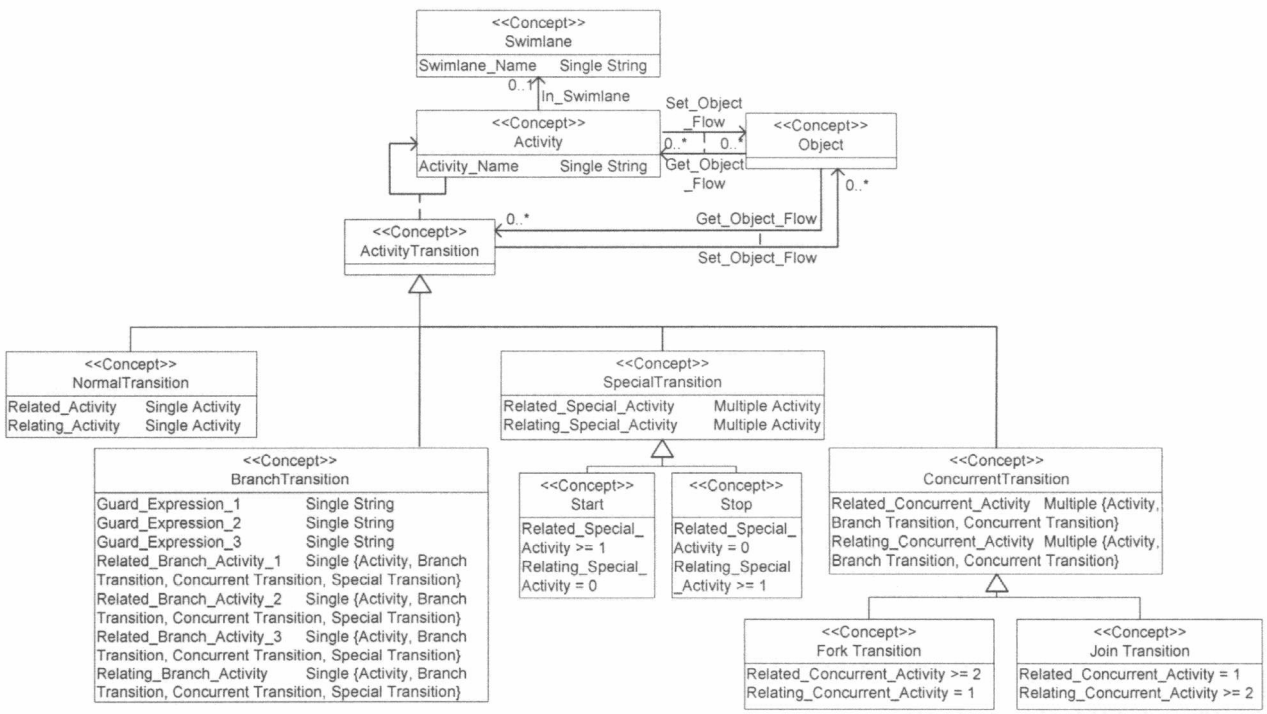


Fig. 10. UML activity diagram ontology

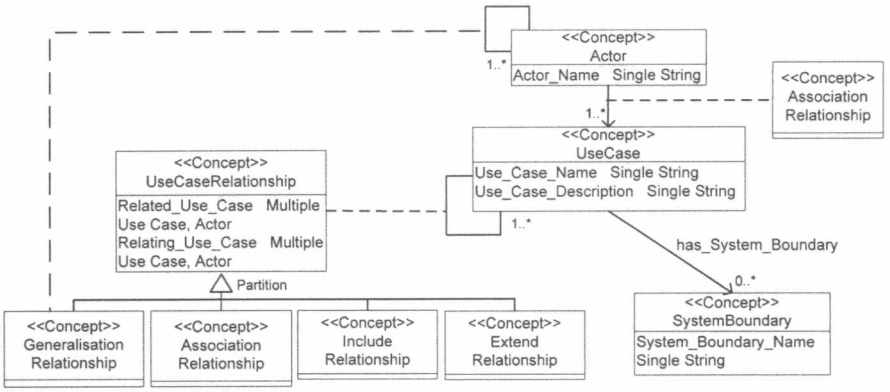


Fig. 11. UML usecase diagram ontology

There are three main degrees of entity relationships: unary, binary and complex. The complex entity relationship can be further divided into quaternary and ternary. 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 i.e. relation *Entity1*. For a binary relationship, there are two entities in the relationship i.e. in relations *Entity1* and *Entity2*; while in a ternary relationship, there are three entities in the relationship i.e. in relation *Entity1*, *Entity2*, and *Entity3*. For a quaternary entity relationship, there are four entities in the relationship i.e. in relations *Entity1*, *Entity2*, *Entity3*, and *Entity4*.

In an entity relationship, cardinality can be specified as a string which can be a string of 1..1 (one and only one), 0..* (zero or more), 1..* (one or more), 0..1 (zero or one) as shown in the ontology model. Attributes can also be assigned to relationships referring to relation *has_Attribute_on_Relationships* in the ontology model.

5.3 Ontology Model for a UML Activity Diagram

An activity diagram shows the control flow from activity to activity [27]. Mainly, activity diagrams contain activities, transitions, swimlane, and objects. A locus of activities is specified by a swimlane. This refers to relation *in_Swimlane* in the ontology model of activity diagrams in Figure 10. Every activity belongs to exactly one swimlane; however, transition may force it to 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 shows the path from one activity to the next activity. This means that, ontology class *NormalTransition* that has a cardinality restriction, restricts only the one activity in the relations *Related_Activity* and *Relating_Activity*. Secondly, special transition is further divided into an initial and a stop transition. The initial transition is where the activity diagrams start. This means that, ontology class *Start* which has a cardinality restriction, 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 ontology class *Stop 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 class *BranchTransition*. Lastly, concurrent transition is further divided into a fork and a join transition. The fork transition represents the splitting of a single flow of control into two or more flows of control. This means that ontology class *ForkTransition*, 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 *JoinTransition*, which has cardinality restriction, restricts at least two activities in relation *Relating_Concurrent_Activity* and only one activity in relation *Related_Concurrent_Activity*.

5.4 Ontology Model for a UML Use Case Diagram

A use case diagram shows a set of use cases and actors and their relationships. Commonly, use case diagrams contain actors, use cases, and relationships. Figure 11 shows an ontology model of use case diagrams. Actors and use cases refer respectively to ontology classes *Actor* and *UseCase*. System boundary referring to ontology class SystemBoundary defines use cases limits.

Relationships between use cases, referring to ontology class *UseCaseRelationship*, are categorised into four types of, firstly, generalisation relationship – referring to ontology class *GeneralisationRelationship*; secondly, association relationship – referring to ontology class *AssociationRelationship*; thirdly, include relationship – referring to ontology class *IncludeRelationship*; and lastly, extend relationship – referring to ontology class *ExtendRelationship*. Only the association relationship defines the relationship between actors and use cases, and only the generalisation relationship defines the

relationship between actors. Based on the design of software engineering ontology as shown by the above examples, software engineering ontology was implemented. This is done by choosing the implementation language, implementation tools, and involves the translation of the ontology conceptual framework developed into the implementation language.

There are several ontology languages available such as Resource Description Framework (RDF) [10], Web Ontology Language (OWL) [11], DARPA Agent Markup Language (DAML) [12], Ontology Interchange Language (OIL) [13], DAML+OIL [14], Simple HTML Ontology Extensions (SHOE) [15] and the like for capturing knowledge of interest. Different ontology languages have different facilities. The most recent development in standard ontology languages is OWL from the World Wide Web Consortium (W3C) (http://www.w3.org/). It has the most complete set of expressions for capturing the different concepts and relationships that occur within ontologies. Therefore, it was our preferred choice for this work.

6 Software Engineering Ontology Deployment

In this section, we illustrate how software engineering ontology allows knowledge sharing and facilitates the communication framework.

6.1 Knowledge Sharing

Knowledge sharing through the software engineering ontology eliminates misunderstandings, miscommunications and misinterpretations. 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. For example, one would like to share design of a use case diagram shown in Figure 12. The use case diagram used as example here is derived from the book of Enterprise Java with UML [28].

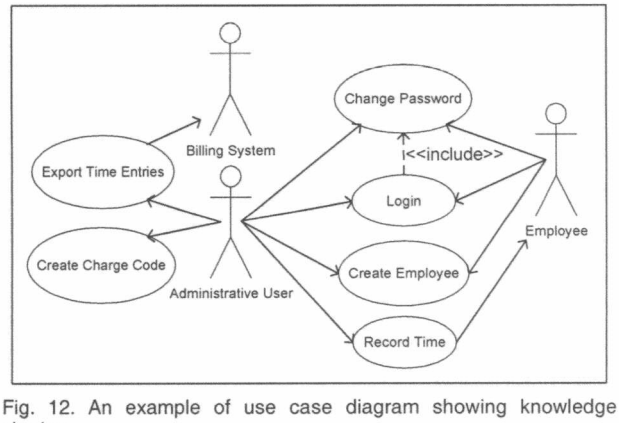


Fig. 12. An example of use case diagram showing knowledge sharing

A list of actions is as follows:

1. Adding new instances of class *UseCase* named 'Export Time Entries', 'Create Charge Code', 'Change Password', 'Login', 'Create Employee', and 'Record Time'.
2. Adding new instance of class *IncludeRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Change Password' and *Relating_Use_Case* with instance of class *UseCase* named 'Login'.
3. Adding new instance of class *AssociationRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Change Password' and *Relating_Use_Case* with instance of class *Actor* named 'Administrative User'.
4. Adding new instance of class *AssociationRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Login' and *Relating_Use_Case* with instance of class *Actor* named 'Administrative User'.
5. Adding new instance of class *AssociationRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Create Employee' and *Relating_Use_Case* with instance of class *Actor* named 'Administrative User'.
6. Adding new instance of class *AssociationRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Record Time' and *Relating_Use_Case* with instance of class *Actor* named 'Administrative User'.
7. Adding new instance of class *AssociationRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Change Password' and *Relating_Use_Case* with instance of class *Actor* named 'Employee'.
8. Adding new instance of class *AssociationRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Login' and *Relating_Use_Case* with instance of class *Actor* named 'Employee'.
9. Adding new instance of class *AssociationRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Create Employee' and *Relating_Use_Case* with instance of class *Actor* named 'Employee'.
10. Adding new instance of class *AssociationRelationship* relating relations *Relating_Use_Case* with instance of class *UseCase* named 'Record Time' and *Related_Use_Case* with instance of class *Actor* named 'Employee'.
11. Adding new instance of class *AssociationRelationship* relating relations *Related_Use_Case* with instance of class *UseCase* named 'Export Time Entries' and *Relating_Use_Case* with instance of class *Actor* named 'Administrative User'.
12. Adding new instance of class *AssociationRelationship* relating relations

Related_Use_Case with instance of class *UseCase* named 'Create Charge Code' and *Relating_Use_Case* with instance of class *Actor* named 'Administrative User'.

13. Adding new instance of class *AssociationRelationship* relating relations *Relating_Use_Case* with instance of class *UseCase* named 'Export Time Entries' and *Related_Use_Case* with instance of class *Actor* named 'Billing System'.

Sharing project information 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 information 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 information.

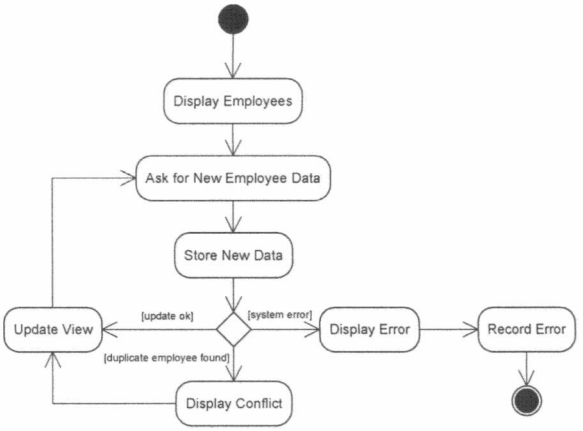
6.2 Communications

Software engineering knowledge formed into software engineering ontology facilitates communication framework among users and provides consistent understanding of the domain knowledge. For example, one would like to communicate changes of project design. Figure 13 (a) shows the original design of an activity diagram while Figure 13 (b) shows an updated activity diagram. The activity diagram used as an example here is derived from the book 'Enterprise Java with UML' [28].

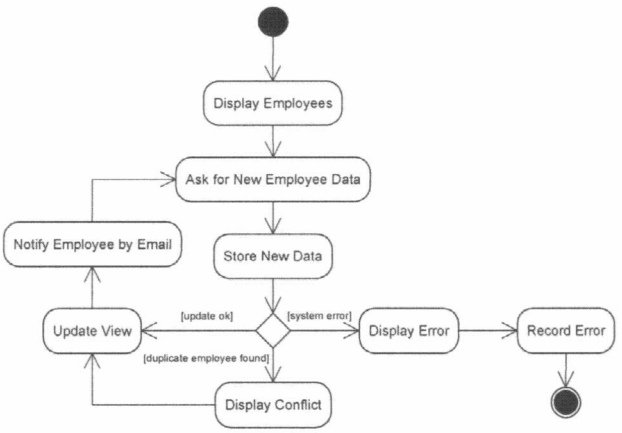
As can be noted when comparing Figure 13 (a) and Figure 13 (b), the software engineer has revised the transition of activity 'Update View'. Originally, activity 'Update View' transited to activity 'Ask for New Employee Data'. Revision has been made by activity 'Update View' transited to activity 'Notify 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 *NormalTransition* 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 *NormalTransition* 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 *NormalTransition* 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 any project information 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 the concept of project information which is being captured (statechart diagrams or activity diagrams) or where that project information resides (statechart diagrams or activity diagrams), it is assumed that this is determined by the member who specifies what the project information really means in the context.



(a) An original design of activity diagram



(a) A revised design of activity diagram

Fig. 13. An example of communication about design changes

Once they 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.

7 Software Engineering Ontology Evaluation

The software engineering ontology has been implemented in the OWL and deployed on a platform. It can be accessed at www.seontology.org. This section is devoted to evaluating the software engineering ontology through the deployed ontology on the platform.

Software engineering knowledge, formed into software engineering ontology, helps communications among team members and provides consistent understanding of the domain knowledge. 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 the platform, software engineering instance knowledge, in accordance with domain knowledge that is described in software engineering ontology, is extracted. By consulting the software engineering ontology, the platform enables references of software engineering domain knowledge and enables extraction of instance knowledge. For example, class diagrams referred to in the software engineering ontology assert how a set of classes is formed in the diagram. The specification imposing a structure on the domain of class diagrams i.e. elicitation of each class consists of class name, class attributes, class operations and relationships hold with other classes. Using software engineering domain knowledge, together with instance knowledge, the platform dynamically and automatically acts for a certain class instance that the member navigates to retrieve accordingly attribute instances, operation instances, and relationship instances together with the related class instance details.

Figure 14 shows examples of instances of class diagrams ontology that are navigated in the platform. In Figure 14(a), *Class* instance *CR_Customer* is navigated to consequently retrieve *ClassAttribute* instances and *ClassOperation* instances. *ClassRelationship* instances can also be navigated to subsequently retrieve *Class* instances that hold in the relationship and applicable properties of the relationship. For example, in accessing *ClassAssociation* instance, *Class* instances held in the relationship and properties like role name and cardinalities are automatically retrieved as shown in Figure 14(b). Similarly, if those *Class* instances are accessed, then a list of *ClassAttribute* instances and a list of *ClassOperation* instances are retrieved to show its attributes and its operations respectively. In accessing each *ClassAttribute* instance, details of attribute's name, attribute's data type, and attribute's visibility are shown as referred to *ClassAttribute* ontology in the software engineering ontology. In Figure 14(c), navigating *ClassAttribute* instance *CR_CustomerID*, its name of 'Customer ID', its data type of 'integer', and its visibility of 'public' can be revealed. The same as *ClassOperation* ontology referred in the software engineering ontology, in accessing each *ClassOperation* instance, details of operation's name, operation's visibility, and operation's parameters and parameters' data type can be retrieved. Moreover, indicating how concepts are inter-related constrains the possible interpretations of terms. For example, the structure of class, object, and component impose differences. Therefore, there needs to be some kind of consensus among the software engineering community or project teams. This then eliminates ambiguous concepts or terms. With team members having the same understanding of concepts, remote communications are more likely to proceed smoothly and effectively in the multi-site setting.

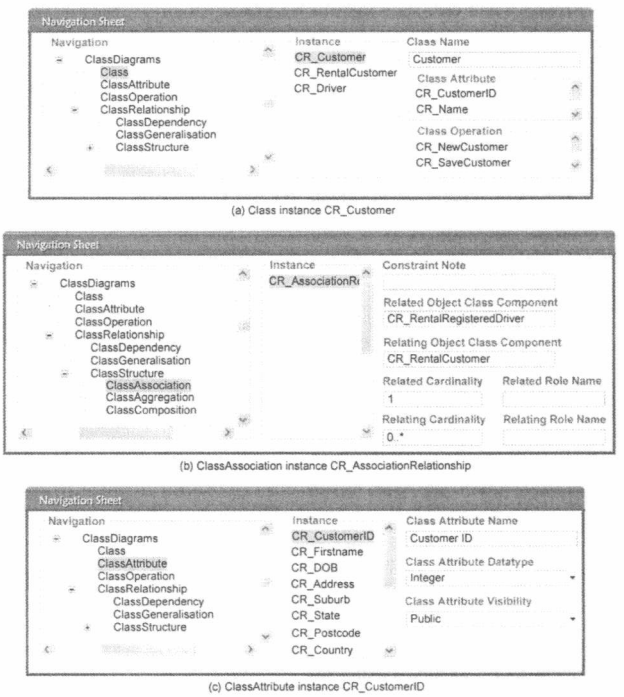


Fig. 14. Examples of instances of class diagrams ontology being navigated in the platform

8. Software Engineering Ontology in Practice

Practical uses of the software engineering ontology are given in this section through examples. We start with examples of problems encountered in the multi-site environment. Then, the use of the software engineering ontology in resolving these particular problems is explained.

The text transcription is difficult because work is carried out in an environment where development teams are widely-distributed geographically and team members are involved in many projects simultaneously. It is a typical means of global communication which is, however, neither efficient nor sufficient for a multi-site environment. Figure 15 shows such an example of the text transcription that, in a multi-site environment, makes less

of an impression.

I am struggling to understand why we need it. I think the system will be simpler for people to understand if we deleted the insurance registered driver.
My reasons for this are that the insurance registered driver is a sub type of the customer. This means that for every insurance registered driver object there must be a corresponding customer object. However, in the customer object we store values like customer type, insurance history value and rental history value. It does not make sense to have these values for the insurance registered driver. I also think people will be confused because we have the rental registered driver as an association with the rental customer (which is a sub type of the customer) but the insurance registered driver is a sub type of the customer.

Fig. 15 An example of text transcription which is neither efficient nor sufficient for multi-site communication

With ontology-based software engineering, the software engineering terms can be annotated or parsed with software engineering ontology concepts and can recall the necessary details and relevant information. We see from Figure 15 that the insurance terminology *registered driver*, *customer*, and *rental customer* can be annotated as class; the term of sub type can be annotated as subclass; the terms of customer type, insurance history value, and rental history value can be annotated as property; the terms of insurance registered driver object and customer object can be annotated as object.

Class, subclass, property, and object apply respectively to the concepts of class, generalization relationship, class property, and class object in the software engineering ontology. By specifying ontology class instances, relevant information of those instances can be discovered dynamically and automatically. For example, in Figure 16 by annotating 'Customer' as a class, details class Customer such as its attributes, its operations, and its relationships with other classes, is automatically presented in Figure 17. To make it realities or have real value, automatically drawing class diagram can be done.

This is carried out by referring to the software engineering ontology which asserts that concept class has its semantic of containing attributes, operations, and relationships holding among other classes. Automatically drawing out details facilitates others' greater understanding of the content, thereby reducing misunderstanding, and eliminating ambiguity.

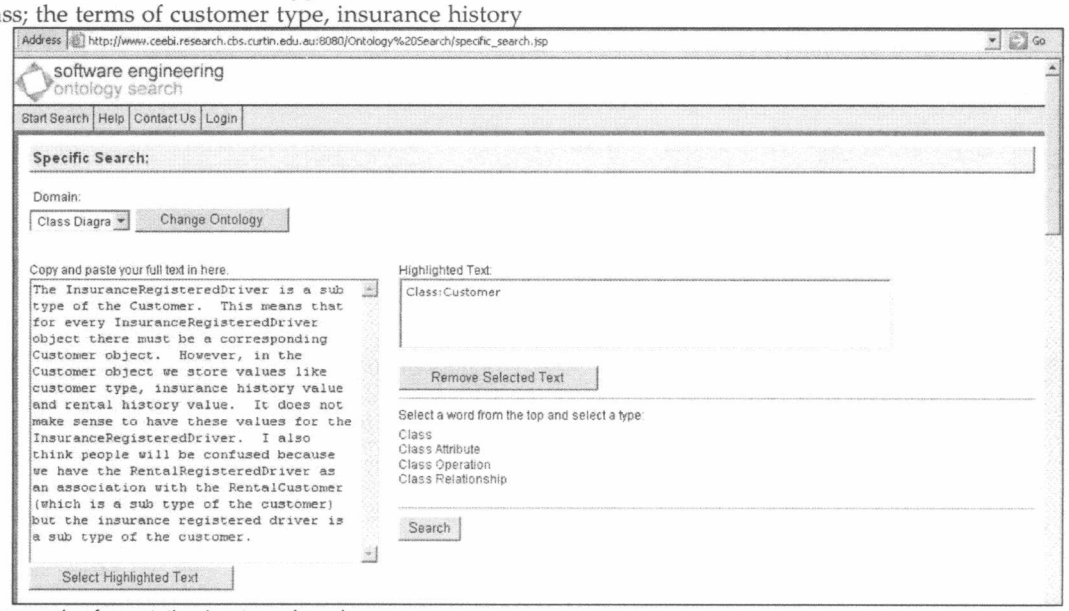


Fig. 16 An example of annotating 'customer' as class

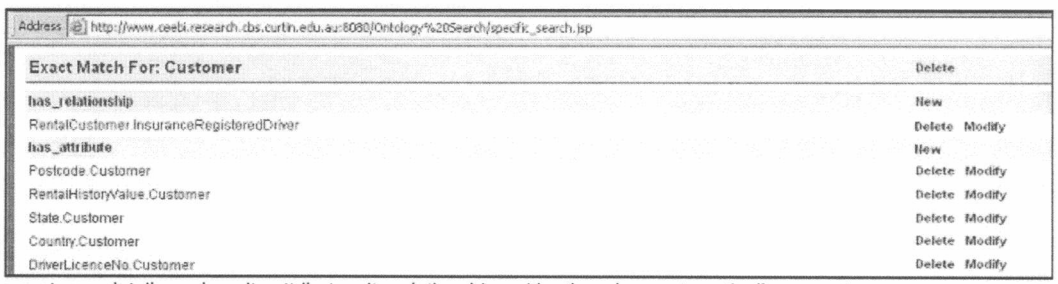


Fig. 17 Class customer details such as its attributes, its relationships with other class automatically present

9. CONCLUSION

In this paper, we have analysed the characteristics of software engineering ontology. We have then defined graphical notations of modelling software engineering ontology as an alternative formalism. The modelling notations are used to design software engineering ontology. We have only covered some distinguished part of modelling domain knowledge of software engineering as example. The practical software engineering ontology has been implemented and deployed. Deployment has been discussed in aspects of knowledge sharing and communication framework. The evaluation of the ontology is presented of its useful in practice. Finally, the deployed software engineering ontology applied to the realities of distributed development is given to demonstrate its real value to the software engineering ontology.

However, there are many improvements that can be made through future work which could consider software engineering ontology evolution. It is a 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.

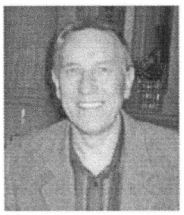
REFERENCES

1. Sommerville, I., *Software Engineering*. 8th ed. 2004: Pearson Education Limited.
2. Bourque, P. *SWEBOK Guide Call for Reviewers*. 2003 [cited 29 May 2003]; Available from: http://serl.cs.colorado.edu/~serl/seworld/database/3552.html.
3. Davenport, T.H. and L. Prusak, *Working Knowledge: How Organisations Manage What They Know*. 1998, Boston, MA: Harvard Business School Press.
4. Witmer, G. *Dictionary of Philosophy of Mind - Ontology*. 2004 [cited May 11, 2004]; Available from: http://www.artsci.wustl.edu/~philos/MindDict/ontology.html.
5. Wikipedia. *Ontology (computer science) From Wikipedia, the free encyclopaedia*. 2006 [cited 8 June 2006]; Available from: http://en.wikipedia.org/wiki/Ontology_%28computer_science%29.
6. Gruber, T.R. *A translation approach to portable ontology specification*. in *Knowledge Acquisition*. 1993.
7. 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.
8. 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.
9. 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.
10. Klyne, G. and J.J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. 2004 [cited; Available from: http://www.w3.org/TR/rdf-concepts/.
11. McGuinness, D.L. and F.V. Harmelen. *OWL Web Ontology Language Overview*. 2004 [cited; Available from: http://www.w3.org/TR/owl-features/.
12. Finin, T., et al. *Automatically Generated DAML Markup for Semistructured Documents*. in *2003 AAAI Spring Symposium on Agent-Mediated Knowledge Management (AMKM)*. 2003.
13. Fensel, D., et al., *OIL: An Ontology Infrastructure for the Semantic Web*. IEEE INTELLIGENT SYSTEMS, 2001. **MARCH/APRIL 2001**.
14. Horrocks, I. and F.v. Harmelen, *Reference Description of the DAML+OIL Ontology Markup Language*. 2001.
15. Luke, S. and J. Heflin, *SHOE 1.01 Proposed specification*. 2000, SHOE Project.
16. *Introduction to OWL*. 2006 [cited; Available from: http://www.w3schools.com/rdf/rdf_owl.asp.
17. Horridge, M., *A Practical Guide To Building OWL Ontologies With The Protege-OWL Plugin*, 1.0, Editor. 2004, University of Manchester.
18. Genesereth, M.R. *Knowledge Interchange Format - draft proposed American National Standard*. 1998 [cited; Available from: http://logic.stanford.edu/kif/dpans.html.
19. Brachman, R.J. and J.G. Schmolze, *An overview of the KL-ONE knowledge representation system*, in *Cognitive Science*. 1985. p. 171-216.
20. Farquhar, A., R. Fikes, and J. Rice. *The Ontolingua Server: A Tool for Collaborative Ontology Construction*. in *10th Knowledge Acquisition for Knowledge-Based Systems Workshop*. 1996. Banff, Canada.
21. MacGregor, R., *Inside the LOOM classifier*. SIGART bulletin, 1991. **2**(3): p. 70-76.
22. Genesereth, M.R. and R.E. Fikes, *Knowledge Interchange Format Version 3 Reference Manual*. 1992, Stanford University Logic Group.
23. Duric, D., *MDA-based Ontology Infrastructure*. Computer Science and Information Systems, 2004. **1**(1).
24. Kogut, P., et al., *UML for ontology development*. The Knowledge Engineering Review 2002. **17**(1): p. 61 - 64.
25. Evermann, J., *A UML and OWL description of Bunge's upper-level ontology model* Software and Systems Modeling, 2008.
26. Gašević, D., D. Djurić, and V. Devedžić, *Model Driven Architecture and Ontology Development* 2006: Springer.
27. Bourque, P., et al. *Guide to the Software Engineering Body of Knowledge*. 2004 Feb. 16, 2005 [cited.
28. Arrington, C., *Enterprise Java with UML*. 2001, New York, USA: John Wiley & Sons, Inc.

Pornpit Wongthongtham received a B.Sc. degree in Mathematics, M.Sc. degree in Computer Science, and her PhD degree in Information Systems. She is currently a research fellow within the Digital Ecosystems and Business Intelligence Institute, Curtin University of Technology, Australia. Her research interests include ontology and semantic web technology, software agents, web services and the like.

Elizabeth Chang is a Research Professor and Director of Digital Ecosystems and Business Intelligence Institute, a Tier 1 Centre of Research Excellence in Curtin University, Australia. She is also a Director for Centre for the Frontier Technology for Extended Enterprise. She has been awarded the Vice Chancellor's Outstanding Performance Award for 2005, and the Dean's Best Researcher of Year Award for 2005 and 2004. She has coauthored 3 books and has published over 300 scientific papers as book chapters, refereed international journals and refereed conferences as well as given numerous invited Keynote papers and obtained over 4 million dollar research and development funds from ARC, industry, local governments and other internal funds. Professor Chang has an extensive background, knowledge and skills in Academia, commerce and industry. Over the last 17 years, she has worked in 4 different universities in Australia namely Curtin, Newcastle, La Trobe and Swinburne Universities. She has been a CIO for Seapower Resources International, incorporating 40 warehouses in Asia, a multi-national logistics operator in Hong Kong and coordinate e-Logistics and e-Warehouse projects for 2 years and few years as a Senior Software Engineer for Philips Public Communication Limited on the Cable TV projects, Animal Research Institute on Farming project and Australian Airlines. Professor Chang has a PhD and research Master in IT and Software Engineering from La Trobe University, Australia.

Tharam Dillon is a Distinguished Research Professor at Digital Ecosystems and Business Intelligence Institute, Curtin University of Technology, Australia. His current research interests include Web semantics, ontologies, Internet computing, e-commerce, hy-brid neurosymbolic systems, neural nets, software engineering, database systems, and data mining. He has more than 650 papers published in international conferences and journals and is the author of 5 books and has another 5 edited books.



Ian Sommerville is a Research Professor of Computer Science at St Andrews University in Scotland. He has worked for many years with social scientists and was amongst the first computer scientists to explore how information from ethnographic studies of work could be used to inform system specification and design. This has led to his current research in the dependability of socio-technical systems where he is working on modelling responsibilities across organisations, organisational memory and coping with systems failure. Ian is the author of a widely used textbook on software engineering which was first published in 1982 and which is now in its 8th edition.