A Concept-Oriented Approach to Support Software Maintenance and Reuse Activities

Dirk Deridder
Programming Technology Lab
Vrije Universiteit Brussel, Brussels, Belgium
Dirk.Deridder@vub.ac.be - http://prog.vub.ac.be/

Abstract. A major activity in software development is to obtain knowledge and insights about the application domain. Even though this is supported by a wide range of tools and techniques, a lot of knowledge remains implicit in the resulting artefacts (e.g. class diagrams). Examples of such implicit knowledge are amongst others the links between the different artefacts, the knowledge that is lost as a result of iterative refinements, and the knowledge that is regarded common sense by the involved parties. Most of this knowledge resides in the heads of the stakeholder, the domain experts, and the developers. As it is likely that they can no longer remember it or that they are no longer available when software reuse and maintenance activities are initiated, this poses a problem. In this paper we present ongoing research in which we focus on our use of an ontology as a medium to tackle this problem. Making the different kinds of knowledge explicit will be accomplished by representing them as concepts in the ontology. Subsequently we will link these concepts to the artefacts created during analysis, design and implementation. This enables a bi-directional navigation between the concepts and the artefacts which consequently will serve as a vehicle to start reuse and maintenance activities.

1 Introduction

During the software development life-cycle the system to be built is conceived through and documented by different kinds of artefacts. Besides modelling the final solution, these artefacts (such as class diagrams) reflect our knowledge about the application domain to a considerable extent. Unfortunately a lot of this knowledge remains implicit and most often resides in the heads of the different people concerned. The most important of this implicit knowledge are amongst others the links between the different artefacts, the knowledge that is lost as a result of iterative artefact refinements, and the knowledge that is regarded as common sense by the involved parties. This includes the link between the conceptual artefacts and their concrete realizations in the implementation.

Especially in the context of software maintenance and reuse activities it is imperative to have access to the afore-mentioned knowledge elements. This is not surprising, for these activities require a major effort in understanding the existing system. For achieving this understanding, it is helpful if you can rely on the people who were involved with the development of the original system. In practice however it is likely that they are no longer available or, if they are available, that they no longer have the specific knowledge you seek. As a result you will spend a major part of your time trying to figure out which concepts are represented by which parts of the code, and vice versa.

On top of this it is likely that the artefacts are not always internally consistent as well as externally consistent. Both types of consistency refer to the correct and consistent usage of terminology and concepts. For internal consistency this usage is confined to one artefact. In the case of external consistency it boils down to the consistent and correct usage between different artefacts. Most of the time these inconsistencies are a direct result of the implicitness of knowledge such as the links between the artefacts. Let's say that a person wants to consider a certain artefact for reuse, both violations of consistency will be quite confusing and will seriously hamper performing this task.

To overcome these problems we propose the use of an ontology as a store for this knowledge and as a driving medium to support the persons who are charged with executing reuse and maintenance activities. For this purpose the different concepts, that are represented in the artefacts, will be defined and stored in the ontology. To obtain an initial set of concepts we will complement the already existing application engineering cycle with a domain engineering cycle. This combination is quite similar to the *dual life-cycle* as presented by Reifer in [6]. Subsequently these concepts will be "glued" to the different (elements of) artefacts through extensional and intensional concept definitions. This will enable a bi-directional navigation between the concepts represented in the conceptual artefacts (e.g. `Car Fleet Management`) and their concrete realizations in the code (e.g. a group of classes).

This research was partially performed in the context of the SoFa project ("Component Development in a SOftware FActory : Organization, Process and Tools"). During this project we delimited our scope of investigation to *Business Support Systems*. Such systems are characterized by the fact that they support knowledge intensive, enterprise critical processes that are cooperatively performed by groups of knowledgeable business workers. In what follows we will assume that the applications of which we speak comply to this characterization.

In this paper we will mainly focus on the concept-part of the approach we envision. Since this depends heavily upon the use of an ontology we will provide a detailed description in Section 2. In it we will briefly discuss our motivation for selecting ontologies as a medium for our approach. Next we will elaborate on our view on ontologies where we will present a notion of concept networks and concept roles. To conclude the section we will present the SoFaCB (SoFa ConceptBrowser) ontology tool. In Section 3 we will sketch how we put this ontology tool into effect to support reuse and maintenance activities. In this section we will present intensional and extensional concept definitions as a way to couple concepts to artefacts/code. To end the section we will propose task ontologies as a way to overcome the gap between broad concepts and their narrow concrete realizations.

2 An Ontology in a Concept-oriented Approach

In computer science there are many different interpretations of what an ontology is or should be. Most of the time this interpretation is depending heavily upon the application in which the ontology is used. We prefer to use the popular but rather abstract definition of Gruber [4], being that an ontology is an explicit specification of a conceptualization. If we instantiate this definition for our work we get that an ontology represents a certain view on an application domain, in which we define the concepts that live in this domain in an unambiguous and explicit way. An important aspect of an ontology, which we believe is not covered by the definition of Gruber, is the referential role that this explicit specification (the concepts) plays. This means that it is used as a work of reference and that there exists a strict commitment from

the users towards the meaning of these concepts i.e. an *ontological commitment* is enforced. To avoid confusion we would like to stress that it is not our goal to create rigorous formal ontologies for the application domains under consideration. Instead we prefer a lightweight approach where it is desirable but not necessary to end up with a set of concepts that is usable as a standard for a larger community.

In the following subsection we will motivate the choice of an ontology as a medium for our approach. Next we will zoom in on our view on an ontology in terms of a concept network and the way the concepts in such a network can play different roles depending on the application context. We will conclude this section with a brief discussion of our lightweight ontology tool SoFaCB.

2.1 Ontology as a Medium : Motivation

If we keep our definition in mind and we retake the issues of implicit knowledge and internal/external consistency, it should become clear why we chose an ontology as a medium for our approach.

First of all, since an ontology is supposed to be an *explicit* specification of a conceptualization, it is natural to use it to capture and store this knowledge. That way we obtain a means to share this knowledge, which would otherwise remain inside the heads of the people who built the original system. Secondly, since an ontology is intended to define the kinds of things that exist in a domain, it provides a number of standard concepts to capture and organize these things. Examples of such concepts are identity, essence, and the subsumption relationship. Thirdly, since the knowledge is now made available in an explicit form it becomes possible to refer to it from within the different artefacts. This will greatly reduce the above-mentioned consistency problems which is mainly a result of the enforced ontological commitment from the artefact-creators towards the used terminology and concepts. Moreover this explicit specification will also prove beneficial for the comprehensiveness of the artefacts they produce. This could be attributed to the fact that a novice to the software system will have a cross-referenced dictionary-alike documentation of the artefacts to his or her disposal.

2.2 Concept Networks and Concept Roles

In Figure 1 we show an example of a partial concept network which we will use throughout this text. To enhance readability we will refer to these concepts by their preferred-label instead of their unique concept numbers. The `Toy Example` domain contains three concepts (clouds): `Car Fleet Management`, `Car`, and `Company Car`. In the `core` domain we have only shown two principal concepts: `Concept`, and `Domain`. All of these concepts are defined by their surrounding definitions. These definitions are connected to the concepts by relations (arrows). For instance for the `Car` concept, the definition slots used are: `has-preferred-label`, `has-definition`, and `has-image`. To introduce a new concept it suffices to "instantiate" the concept `Concept` and fill in the corresponding definition slots. Doing so you obtain a network structure that is very similar to conceptual graphs [8].

In a concept network we adhere the view that every element in it is a concept. This means that the relations we use are also represented by concepts. Thus the system is said to be self-contained. To group a set of related concepts we have introduced a `Domain` concept. The

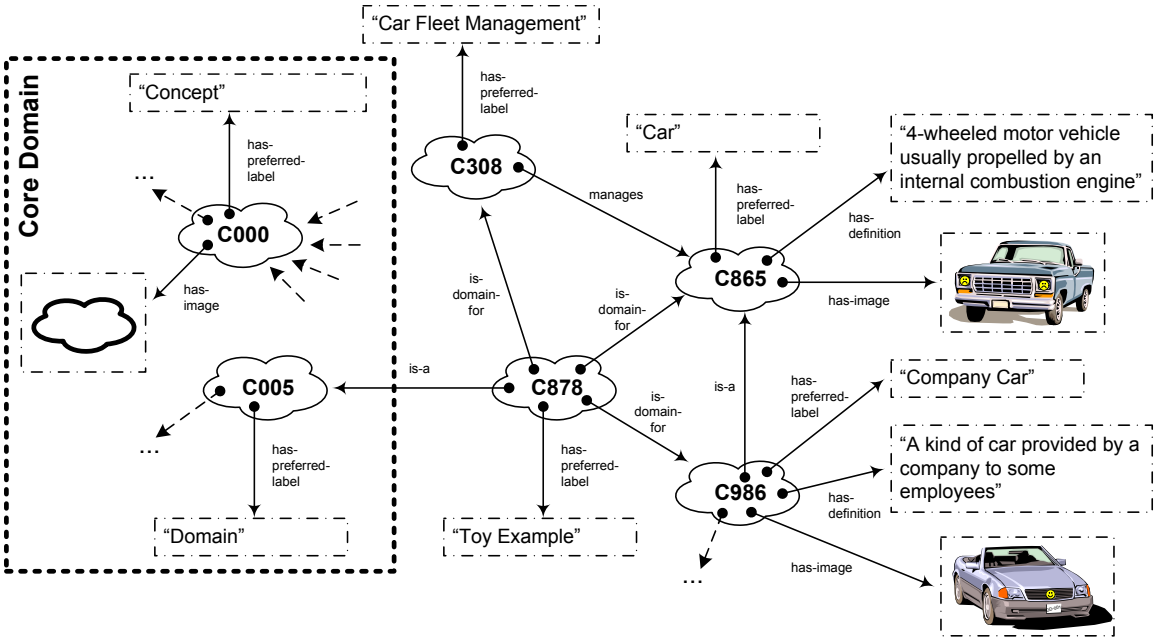

Figure 1: An example of a partial concept network,

semantics of this is that concepts can belong to many domains. As shown in the example, the `Domain` concept belongs to the `core` domain. This domain can be seen as an ontology for the concepts in the right part of the figure since these are described in terms of it.

If we generalize this observation we get three 'concept roles' in an ontology : a core role, a domain structural role, and an application role. These roles correspond to *virtual layers* in the ontology, where the concepts in the (bottom) application layer are described in terms of the concepts in the (middle) domain structural layer which are finally described in terms of the concepts in the (top) core layer. In our example you find concepts such as the `Manages` relationship in a domain structural role. The core role is taken up by concepts such as `Concept`, `is-a`, `...`, enabling us to create the `Company Car` concept for instance. Finally we have concepts with an application role such as `Car`, and `Car Fleet Management`. Note that concepts can shift their roles according to the context in which they are used. For example if we describe the concepts for a specific car fleet management application, then the `Car` and `Company Car` concepts will be used in a domain structural role.

2.3 The SoFaCB Ontology Tool

Many excellent tools for building and managing ontologies are already available [1, 2]. Since we pursued a lightweight approach and since we wanted to gain experience in the do's and don'ts in this field, we nevertheless decided to create our own lightweight tool from scratch. In this light we found the report [3] from the creators of Protégé very useful. This report presents the evolution of their work and several issues regarding ontology tool building.

In Figure 2 we present a snapshot of the SoFaCB tool in which we visualize the concept `car` from the example in Figure 1. The value types for concept definitions that are currently allowed are limited to text, images, and concepts. Note that what is shown in the right pane of the browser is the on-the-fly HTML-translation of the underlying concept network for the

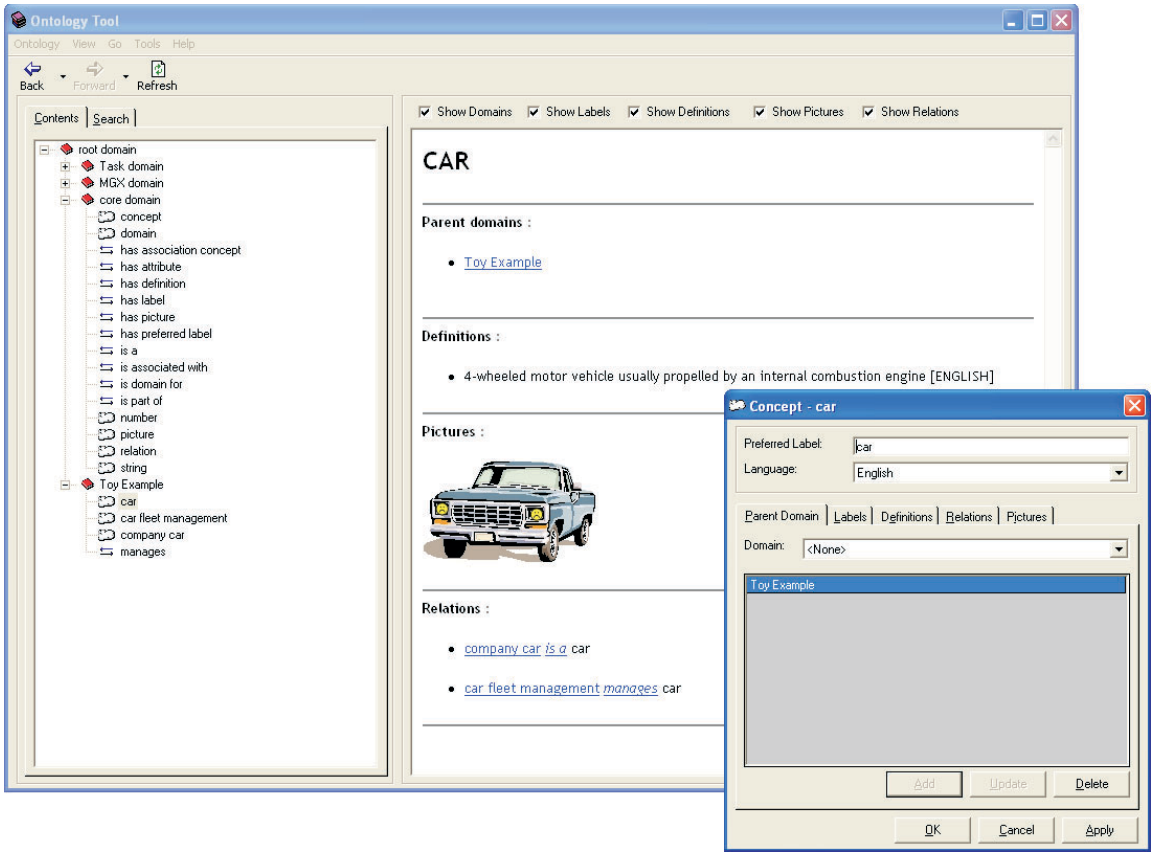

Figure 2: A snapshot of the SoFaCB tool. The left pane of the browser shows the available concepts, and the right pane shows the details of the selected concept. The smaller window on the foreground is used to enter values for the definition slots of a new concept.

selected concept.

Since a lot of domain models are represented as UML diagrams (class diagrams and activity diagrams) we have included a Rational Rose importer that translates these models into a concept network representation.

Besides being lightweight, our prototype tool was also intended to be as generic as possible. With this we mean that the number of hard-coded concepts had to be limited (or ultimately non-existent). This objective becomes difficult to obtain when confronted with the higher level concepts. Examples of such (core) concepts are `concept`, `association`, `is-a`, ... By keeping these concepts soft-coded it becomes possible to adapt the tool from within itself. This is in part facilitated by our data model, which we kept as simple and as generic as possible, and the fact that the tool directly uses certain concept definitions to provide its functionality. A very simple example of the latter is the “fluffy cloud” image attached to the concept `concept`, which is used by the tool as an icon in the tree-view. More advanced uses of this reflective behavior would be to change the template for the definition slots (and corresponding user interface) for creating a new concept.

As a result of a number of non-functional requirements in the project that supported this research, we were not able to carry this genericity to its limit. For instance, in the current version it is not possible to alter some of the definitions of the core concepts because many of them are hard-wired into the tool. Presently we have started work on a new version of the tool

in Smalltalk in which we will eliminate this issue. In it we will also include intensional and extensional concept definition types in SOUL (Smalltalk Open Unification Language) [9]. SOUL is an interpreter for Prolog that runs on top of a Smalltalk implementation. Besides allowing Prolog programmers to write 'ordinary' Prolog, SOUL enables the construction of Prolog programs to reason about Smalltalk code. Amongst others this enables declarative reasoning about the structure of object-oriented programs and declarative code generation. The use of these intensional and extensional concept definition types will be explained in section 3.

3 Concept-oriented Support for Reuse and Maintenance

As we stated in the introduction, we will complement the application engineering cycle with a domain engineering cycle. Central to both cycles will be the ontology, which is used as the point of reference for the concepts. Initially the domain engineering cycle will provide the core set of concepts and relations that are used in the artefacts constructed in the application engineering cycle.

Whenever a reuse activity is initiated we will use the ontology to locate the asset to be reused. This is done by identifying the concepts needed to accomplish the reuse action, and by using the attached extensional and intensional concept definitions to locate the artefacts that 'implement' them. A similar approach is followed to support maintenance activities. In the following subsection we will briefly introduce the idea of both concept definition types.

3.1 Linking Concepts to Artefacts

Intensional and extensional concept definition types were based on the idea of software views as described in Mens *et al.* [5] and will make it possible to connect the concepts in the ontology to actual Smalltalk entities. An extensional definition will summarize all the entities to which we want to link a certain concept. Conversely an intensional definition will be represented by a SOUL-'formula' which makes it possible to calculate the corresponding Smalltalk entities. The former definition type is easy to formulate (just enumerate the entities), but is rather static and of limited use in highly evolving implementations. The latter can sometimes be very difficult to formulate (a logic rule must be formulated that describes the Smalltalk entities you want) but provides a highly dynamic and very powerful mechanism for code reasoning/querying purposes.

Even now that we have identified a mechanism that allows us to connect concepts to artefacts/code, one main question remains unanswered : How do you link very broad concepts (such as `Car Fleet Management`) to scattered code entities (such as a group of classes)? The answer to this question lies in the use of task ontologies which we will describe in the next subsection.

3.2 Describing High-level Concepts with Task Ontologies

We have based our idea of task ontologies on task models as described by Schreiber *et al.* in the *CommonKADS* methodology [7]. Task models allow us to abstract and to position the different tasks within a business process. A task is a subpart of a business process that represents a goal-oriented activity. Popularly stated, a task model provides a decomposition

of high level tasks into subtasks together with their inputs/outputs and I/O flow that connects them. The actual implementation of a task is described by one or more task methods. It is the decomposition of the high-level tasks into subtasks that we use to decompose broad (task-oriented) concepts into narrower concepts. This makes it possible to bridge the gap between broad concepts and Smalltalk entities.

To validate the idea of task ontologies we have set up an experiment in which we used the prototype tool to create a simple ontology which enabled us to express *task ontologies*. Consequently we used these concepts in the task ontology to describe the task models of a certain domain. Within this research project we have successfully used SoFaCB to describe a set of task models in the broadcasting domain. In this case you find concepts such as `Task`, `Input`, `Task Method`, ... in a domain structural role. The core role is taken up by concepts such as `Concept`, `is-a`, ..., enabling us to create the `Task` concept for instance. Within the broadcasting domain we have concepts with an application role such as `Transmission Schedule Management` (as a `Task`), `Pre-transmission Schedule` (as `Input`), `Transmission Schedule Deviations` (as `Input`), and `Post-transmission Schedule` (as `Output`). Remember that concepts can shift their roles according to the context in which they are used. For example if we describe a task method (a concrete implementation of a task), then the `Transmission Schedule Management` concept will be used in a domain structural role. To bridge the gap between the broad `Transmission Management` concept and the code, we will decompose it into subtasks such as `Verify Schedule`, `Generate Schedule Difference`, ...

Providing a business analyst with a task model makes it possible to direct him/her to ask a client questions about the needed task method with respect to the abstract task model. When basic components are created that correspond to the general tasks in the task model, it consequently becomes possible to propose a solution within the boundaries of the existing technological infrastructure. With respect to reuse these task models can thus be used to guide business analysts in performing a business analysis in which the task models are instantiated and adapted to satisfy specific customer needs.

4 Conclusion

In this paper we have presented an ontology as a medium for a concept-oriented approach to support software maintenance and reuse activities. This approach uses the ontology to capture (implicit) knowledge in software development artefacts as concepts. In this explicit form it becomes possible to share these concepts that would otherwise remain hidden with the people originally involved in the development of the system. Moreover it becomes possible to use the ontology as a point of reference which will improve the consistency of the software artefacts produced. As a means to link concepts to artefacts/code we propose the use of intensional and extensional concept definition types in SOUL. These will serve as a vehicle that enables a bi-directional navigation between both sides. To overcome the ‘conceptual gap’ between broad concepts and fine-grained (scattered) implementation artefacts we have presented task ontologies. Even though the research we presented here is still in its infancy we were already able to successfully validate some of these ideas in the IWT research project SoFa. During this project we successfully used our SoFaCB ontology tool to represent a task ontology within the broadcasting domain with which we supported business analysts in advocating reuse.

Acknowledgements

This work has been supported in part by the SoFa project (“Component Development and Product Assembly in a Software Factory : Organization, Process and Tools”). This project was subsidized by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and took place in cooperation with MediaGeniX and EDS Belgium.

References

[1] O. Corcho, M. Fernández-López, and A. Gómez Pérez. IST project IST-2000-29243 : OntoWeb - Ontology-based information exchange for knowledge management and electronic commerce : D1.1. technical roadmap v1.0. http://babage.dia.fi.upm.es/ontoweb/wp1/OntoRoadMap/index.html, 2001.

[2] A.J. Duineveld, R. Stoter, M.R. Weiden, B. Kenepa, and V.R. Benjamins. Wondertools? A comparative study of ontological engineering tools. In *Proceedings of the 12 th International Workshop on Knowledge Acquisition, Modeling and Mangement (KAW'99)*, Banff, Canada, 1999. Kluwer Academic Publishers.

[3] W. Grosso, H. Eriksson, R. Fergerson, J. Gennari, S. Tu, and M. Musen. Knowledge modeling at the millennium – the design and evolution of protégé-2000. In *Proceedings of the 12 th International Workshop on Knowledge Acquisition, Modeling and Mangement (KAW'99)*, Banff, Canada, 1999.

[4] T.R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[5] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. Software Engineering and Knowledge Engineering (SEKE2002), Ischia, Italy, 2002.

[6] D. J. Reifer. *Practical Software Reuse*. Wiley Computer Publishing, 1997.

[7] G. Schreiber. *Knowledge Engineering and Management - The CommonKADS Methodology - A software engineering approach for knowledge intensive systems*. MIT Press, 2000.

[8] J.F. Sowa. *Conceptual Structures - Information Processing in Mind and Machine*. The Systems Programming Series, Addison-Wesley, 1984.

[9] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Lab, Brussels, Belgium, 2001.