Developing Software for and with Reuse: An Ontological Approach

Ricardo A. Falbo[1], Giancarlo Guizzardi[2], Katia C. Duarte[1], Ana Candida C. Natali[1]

[1]Computer Science Department, Federal University of Espírito Santo
Fernando Ferrari Avenue, CEP 29060-900, Vitória - ES - Brazil
falbo@inf.ufes.br

[2]Centre for Telematics and Information Technology, University of Twente
Enschede, The Netherlands
guizzard@cs.utwente.nl

Abstract

Software reuse has been pointed as one of the most promising technique to deal with quality and productivity problems. To support reuse, software processes have to consider two facets: developing for reuse and developing with reuse. In this paper we present an ontology-based approach for software reuse and discuss how ontologies can support several tasks of a reuse-based software process.

Keywords: Software Reuse, Software Engineering Tools and Techniques, Software Process, Ontologies.

1. Introduction

Software reuse is considered to be one of the most promising techniques to improve software quality and productivity. Effective software reuse requires collections of designed-for-reuse software components and mechanisms to retrieve reuse candidates, adapt them and even create new ones using the information provided by similar components [1]. Moreover, we need to bind those elements using a software process that really let to software reuse. This process must consider two different perspectives: developing reusable assets (developing *for* reuse) and developing using those reusable assets (developing *with* reuse).

In this context, ontologies can play an important role. An ontology can promote common understanding among developers, and can be used as a basis for software specification and development. Also, it can be used to improve access to information. However, one of the major drawbacks to a wider use of ontologies in Software Engineering is the lack of approaches to insert ontologies in a more conventional software development process.

In this paper, we propose an ontology-based approach for developing software *for* and *with* reuse. Section 2 discusses ontology applications and their relation with software reuse. In section 3, we discuss briefly a method for engineering ontologies and some aspects that you believe are essential to get the major benefits of the use of ontologies in software development. Since the current leading paradigm in Software Engineering is the object technology, we also present a systematic approach to derive object models from ontologies in order to derive reusable assets. A study case using our approach in the software quality domain is presented in sections 4 and 5. Section 6 discusses related works. Finally, in section 7, we report our conclusions.

2. Ontologies and software reuse

Ontologies are becoming an important mechanism for building software, since they can be used to overcome barriers created by disparate vocabularies, representations and tools.

According to Uschold [2], "an ontology may take a variety of forms, but necessarily it will include a vocabulary of terms, and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms". Thus, an ontology consists of concepts and relations, and their definitions, properties and constrains expressed as axioms. An ontology is not only an hierarchy of terms, but a fully axiomatized theory about the domain [3].

Jasper and Uschold [4] classified applications of ontologies in four main categories, emphasizing that an application may integrate more than one of these categories:

- Neutral Authoring: an ontology is developed in a single language and it is translated into different formats and used in multiple target applications.
- Ontology as Specification: an ontology of a given domain is created and it provides a vocabulary for specifying requirements for one or more target applications. In fact, the ontology is used as a basis for specification and development of some software, allowing knowledge reuse.

- Common Access to Information: an ontology is used to enable multiple target applications (or humans) to have access to heterogeneous sources of information that are expressed using diverse vocabulary or inaccessible format.
- Ontology-based Search: an ontology is used for searching an information repository for desired resources, improving precision and reducing the overall amount of time spent searching.

Although we are most interested in the use of ontologies as specification, we also agree that an ontology almost always has multiple purposes. This is specially highlighted in the case of software reuse. It is clear that the use of ontology as a specification is the basis for software reuse. But we have to regard other scenarios.

The neutral authoring scenario is also important, mainly when applications will be developed using different technology (e.g., objects and logics). This insight shows that we need to define different approaches to implement ontologies, each one suitable to the corresponding technology.

Common access to information scenario is essential to avoid misunderstanding among developers. It is vital for reuse tasks, such as adapting components and creating new assets based on existing ones, as well as for selecting black-box components and for providing access to shared data and services.

Finally, an ontology-based search has great potential to improve structuring and searching in component libraries. As pointed by Jasper and Uschold [4], an ontology may play several roles to assist search: (i) it can be used for semantically structuring and organizing the information repository (in our case, a component library); (ii) it may be used as a conceptual framework to developers think about this repository and formulate queries; (iii) it can be used for refining queries; and (iv) it may be used to perform inference to improve the query.

Analyzing these scenarios, we can notice that software reuse can take several advantages from the use of ontologies. However, the ontology development process must be flexible enough to consider all these scenarios.

3. Using ontologies in domain engineering

Several process models have been proposed for software reuse, almost always establishing parallel tracks for domain engineering and software engineering. Domain engineering concerns the work required to establish a set of software artifacts that can be reused by the software engineer [5], as shown in figure 1.

The purpose of domain engineering is to identify, model, construct, catalog and disseminate a set of software artifacts that can be applied to existing and future software in a particular application domain [5]. In the domain engineering track, ontologies can act as both a domain model and a component in the repository. It can also be used for structuring the repository, as mentioned above.

In this paper we are particularly interested in the use of an ontology as a domain model and how to derive components from it. Then, in the following subsections we discuss an approach for building ontologies and for deriving object frameworks from them. It should be noticed that, to regard all potential scenarios discussed in section 2, we need an approach that guides the ontology developer to achieve these goals.

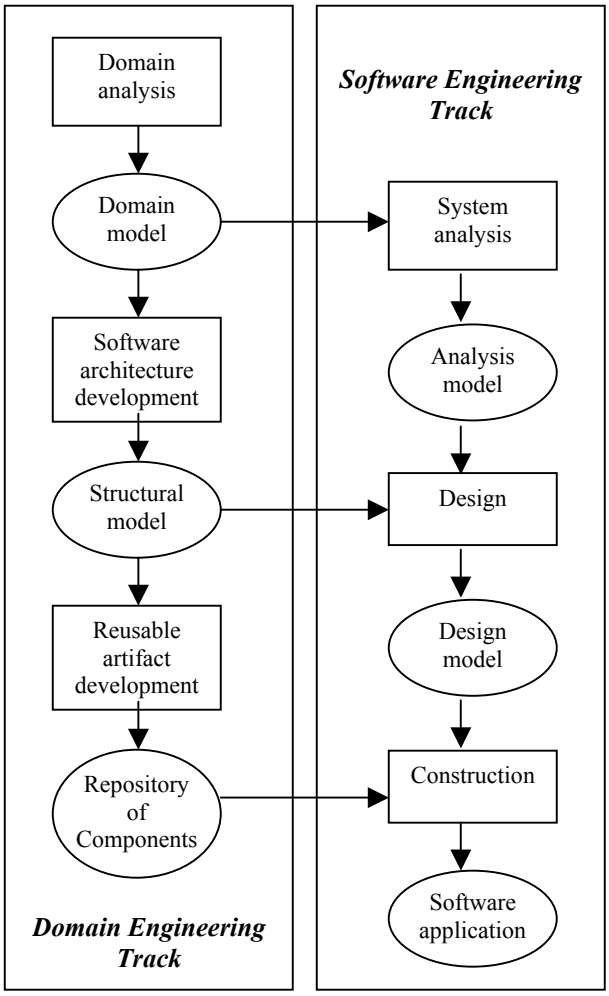


Figure 1 - A process model that emphasizes reuse [5].

3.1 A systematic approach for building ontologies

Basically, the proposed approach encompasses the following activities [3] as shown in figure 2: purpose identification and requirement specification, ontology capture, ontology formalization, integrating existing ontologies, and ontology evaluation and documentation. The dotted lines indicate that there is a constant

interaction, albeit weaker, between the associated steps. The filled lines show the main work flow in the ontology building process. The box involving the capture and formalization steps enhances the strong interaction, and consequently iteration, between these steps.

The first activity - Purpose identification and requirement specification - concerns to clearly identify the ontology purpose and its intended uses, that is, the competence of the ontology. To do that, we suggest the use of competency questions [6].

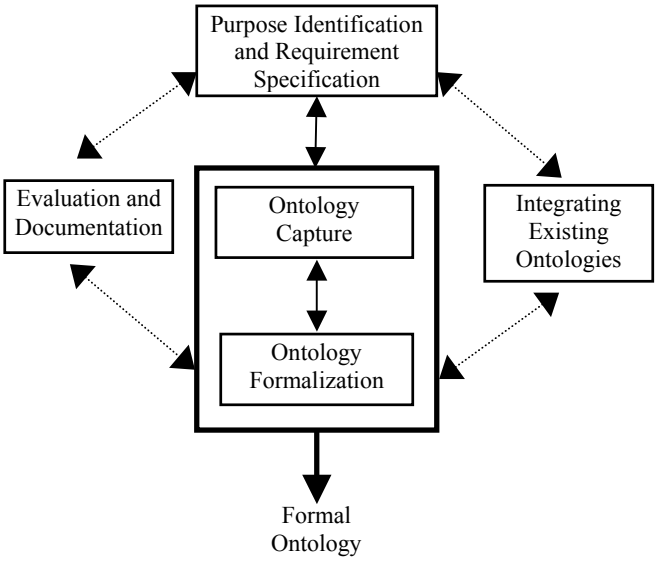
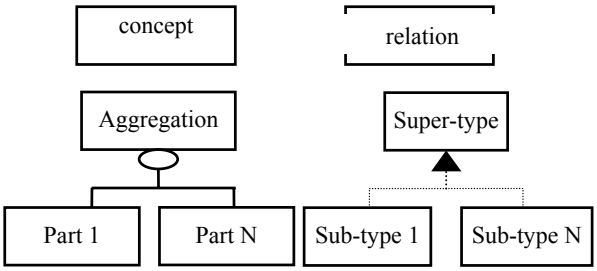


Figure 2 - Steps in the ontology development process.

In ontology capture, the goal is to capture the domain conceptualization based on the ontology competence. The relevant concepts and relations should be identified and organized. A model using a graphical language, with a dictionary of terms, should be used to facilitate the communication with the domain experts. As a graphical language for expressing ontologies, we proposed LINGO [3]. LINGO basically represents a meta-ontology, and thus, it defines the basic notations to represent a domain conceptualization. That is, in its simplest form, its notations represent only *concepts* and *relations*. Nevertheless, some types of relations have a strong semantics and, indeed, hide a generic ontology. In such cases, specialized notations have been proposed. This is the striking feature of LINGO and what makes it different from other graphical representations: any notation beyond the basic notations for concepts and relations aims to incorporate a theory. This way, axioms can be automatically generated. These axioms concern simply the structure of the concepts and are said *epistemological axioms*. Figure 3 shows part of LINGO notation and some of the axioms imposed by the whole-part relation. These axioms form the core of the mereological theory as presented in [7].

Besides the epistemological axioms, other axioms can be used to represent knowledge at a signification level. These axioms can be of two types: *consolidation axioms* and *ontological axioms* [3]. The former aims to impose constraints that must be satisfied for a relation to be consistently established. The latter intends to represent declarative knowledge that is able to derive knowledge from the factual knowledge represented in the ontology, describing domain signification constraints.



(A1) $\forall x \ \neg partOf(x,x)$
(A2) $\forall x,y \ partOf(y,x) \leftrightarrow wholeOf(x,y)$
(A3) $\forall x,y \ partOf(y,x) \rightarrow \neg \ partOf(x,y)$
(A4) $\forall x,y,z \ partOf(z,y) \wedge partOf(y,x) \rightarrow partOf(z,x)$
(A5) $\forall x,y \ disjoint(x,y) \rightarrow \neg\exists z \ partOf(z,x) \wedge partOf(z,y)$
(A6) $\forall x \ atomic(x) \rightarrow \neg\exists y \ partOf(y,x)$

Figure 3 - LINGO main notation and some axioms.

Someone could argue that another graphical language is unnecessary. Cranefield and Purvis [8], for example, advocate the use of UML as an ontology modeling language. We partially agree with their arguments, but we decided not to use some existing graphical language due two main related reasons. First, an important criterion to evaluate ontology design quality is minimum ontological commitments [9]. Based on this principle, a graphical language in this context must embody only notations that are necessary to express ontologies. This is not the case of UML and majority graphical languages available. Second, since an ontology intends to be a formal model of a domain, it is important that the language used to describe it has formal semantics. Again, this is not the case of the majority graphical languages available, including UML. However, we cannot ignore that UML is a standard and its use is widely diffused. Moreover, there are efforts to define UML semantics, such as pUML [10]. Based on that, we are now studying to define a subset of UML that can play the same role of LINGO following the same thread of [8].

Backing to our ontology development process, the formalization activity aims to explicitly represent the conceptualization captured in a formal language. Again, based on the minimum ontological commitment criterion,

we argue that, when it is not necessary any special commitment with a specific meta-ontology that proves itself to be adequate to the ontology in development, first order logic should be the preferred formalism, since it is such a formalism that embeds less ontological commitments.

During the capture and/or formalization steps, it could be necessary to integrate the current ontology with existing ones, in order to seize previously established conceptualizations. Indeed, it is a good practice to develop general modular ontologies, more widely reusable, and to integrate them, when necessary, to obtain the desired result.

Finally, the ontology must be evaluated to check whether it satisfies the specification requirements. It should also be evaluated in relation to the ontology competence and some design quality criteria, such those proposed by Gruber [9]. It should be noticed that the competency questions play an essential role in the evaluation of the completeness of the ontology, specially when considering its axioms.

All the ontology development must be documented, including purposes, requirements and motivating scenarios, textual descriptions of the conceptualization, the formal ontology and the adopted design criteria. A potential approach to document an ontology is to use a hypertext, allowing browsing along term definitions, examples and its formalization, including the axioms. The use of XML can be worthwhile.

We advocate, based on our experience in ontology development, that the approach described easies the development of quality ontologies, specially in those aspects concerning minimum ontological commitments criterion. However, when considering ontology as a specification, this striking feature is also a problem, since the ontology is built generally in a high abstraction level to be directly reused in software development. We have experimented to reuse ontologies in the development of knowledge-based systems, information systems (using object technology) and hypermedia systems. Except for the first case, we identify a need to lower the abstraction level of our ontologies to actually put them in practice. To deal with this problem, we have been working in ways to create more reusable assets from the ontologies. Next, we present our approach to derive object-based artifacts from ontologies.

3.2 From domain ontologies to objects

To support ontologies to Java-objects mapping, we proposed a systematic approach that is composed of a set of directives, design patterns and transformation rules [11]. The directives are used to guide the mapping from the epistemological structures of the domain ontology (concepts, relations, properties and roles) to their counterparts in the object-oriented paradigm (classes, associations, attributes and roles). Contrariwise, design patterns and transformation rules are applied in the ontological and consolidation axioms mapping, respectively. The application of these guidelines is also supported by a Java-framework that implements the mathematical type Set.

To derive objects from domain ontologies, it is worthwhile to adopt a formalism that lies at an intermediate abstraction level between first-order logics and objects. For this purpose, we used a hybrid approach based on pure first-order logic, relational theory and, predominantly, set theory [11]. So, the first step in our approach is the complete axiomatization of the domain theory using the set-based formalism.

Once defined the Set-based axioms, we can initiate the object mapping. Concepts and relations are naturally mapped to classes and associations in an object model, respectively. Furthermore, methods are created in both classes involved in an association. Properties of a concept shall be mapped to attributes of the class that is mapping the concept. Although this approach works well in most cases, it is important to point some exceptions that we have found:

- some concepts can be better mapped to attributes of a class in an object model because they do not have a meaningful state in the sense of an object model;
- some concepts should not be mapped to an object model because they were defined only to clarify some aspect of the ontology, but they do not enact a relevant role in an object model;
- relations involving a concept that is mapped to an attribute (or that is not considered in the mapping) should not be mapped to the object model.

Subsumption relations do not require any additional implementation, i.e., subtype-of relations among concepts can be directly mapped to generalization/specialization relations among classes. However, it is not the case of Whole-Part relations. The UML notation for aggregation does not guarantee the fulfillment of the mereology theory constraints. To deal with this problem, we developed a design-pattern (whole-part design pattern) [11].

Considering consolidation axioms, we identified two cases to address. First, consolidation axioms that concern to object types, do not need any mapping since we are working with a strongly typed language – Java. Second, we developed a design pattern (consolidation pattern) to deal with consolidation axioms whose purpose is to describe preconditions that must be satisfied or properties that must hold so that a relation could be established between two elements.

Finally, it is necessary to map ontological axioms to the object model. These axioms are formalized to answer to the competency questions of the ontology. Methods are

derived from ontological axioms, using a set of transformation rules [11].

3.3 Final regards about our approach

We have been using the approach described in several domains, such as software process modeling [11], software quality and video on demand. To show the application of our approach, in the next two sections we present part of the work done in the software quality domain. Different CASE tools can be thought in this domain, such as tools for quality planning and tracking and a knowledge management system. In fact, we have already developed two applications using the infrastructure derived: a tutorial to guide novice software in learning about software quality and ControlQ, a tool to support quality planning and tracking.

4. Developing for reuse: an experience in software quality domain

As pointed by Crosby, cited by Pressman [5], "the problem of quality management is not what people don't know about it. The problem is what they think they do know". Before we can devise a strategy for producing quality software, we must understand what software quality means. But this is not an easy task. There are several information sources (books, standards, papers, experts, and so on) using many different terms with no clear semantics established. There is not a consensus about the terminology used, what causes misunderstanding and several problems in the definition of a quality program. To deal with these problems, we developed an ontology of software quality. Several books, standards, and experts were consulted during the ontology development process and a consensus process was conducted. Due to limitations of space, we present only part of this ontology, concerning only the following competency questions:

1. Which is the nature of a quality characteristic?
2. In which sub-characteristics can a quality characteristic be decomposed?
3. Which characteristics are relevant to evaluate a given software artifact?
4. Which metrics can be used to quantify a given characteristic?

To address these competency questions, the concepts and relations shown in figure 4 were considered. As shown in this figure, a software quality characteristic can be classified according to two criteria. The first one says if a quality characteristic can be directly measured or not. A non mensurable characteristic must be decomposed into subcharacteristics to be computed by the aggregation of their subcharacteristic measures. A mensurable characteristic can be directly measured applying some metric. The second classification enforces that product characteristics should only be used to evaluate software artifacts. Artifact is highlighted since it is a concept from the software process ontology [3], which were integrated with the quality ontology been presented.

From LINGO notation, the following epistemological axioms can be derived:

$(\forall\ qc)\ (nmensqc(qc) \rightarrow qchar(qc))$ (E1)
$(\forall\ qc)\ (mensqc(qc) \rightarrow qchar(qc))$ (E2)
$(\forall\ qc)\ (prodqc(qc) \rightarrow qchar(qc)\ )$ (E3)
$(\forall\ qc_1, qc_2)\ (subqc(qc_1, qc_2) \rightarrow\ \neg\ subqc(qc_2\ , qc_1))$ (E4)
$(\forall\ qc_1, qc_2)\ (subqc(qc_1, qc_2) \leftrightarrow\ superqc(qc_2\ , qc_1))$ (E5)
$(\forall\ qc_1, qc_2, qc_3)\ (subqc(qc_1, qc_2) \wedge subqc(qc_2, qc_3) \rightarrow subqc(qc_1\ , qc_3\ ))$ (E6)
$(\forall\ qc)\ (mensqc\ (qc) \leftrightarrow \neg\ (\exists\ qc_1)\ (subqc(qc_1, qc)))$ (E7)
$(\forall\ qc)\ (nmensqc(qc) \rightarrow (\exists\ qc_1)\ (subqc(qc_1, qc)))$ (E8)
$(\forall qc,m)(mensqc(qc) \rightarrow (\exists\ m)\ (quant(m, qc))$ (E9)
$(\forall qc,a)(prodqc(qc) \rightarrow (\exists\ a)\ (relev(a, qc))$ (E10)

where the predicates *qchar*, *nmensqc*, *mensqc* and *prodqc* formalize the concepts of quality characteristic, non mensurable quality characteristic, mensurable quality characteristic and product quality characteristic, respectively, and the predicates *subqc/superqc*, *quant* and *relev* formalize the whole-part, quantification and relevance relations, respectively.

Axioms (E1) to (E3) are derived by the subsumption theory. Axioms (E4) to (E7) are some imposed by the whole-part relation. Finally, axioms (E8) to (E10) are given by cardinality constraints.

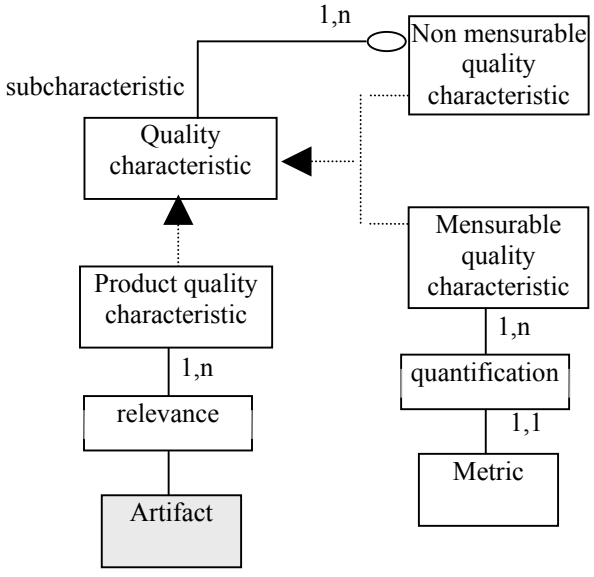


Figure 4 – Part of the software quality ontology.

Several consolidation axioms were defined, such as:
$(\forall qc,qc_1)(subqc(qc_1,qc) \wedge prodqc(qc) \rightarrow prodqc(qc_1))$ (C1)

This axiom says that if a product quality characteristic (qc) is decomposed in subcharacteristics (qc_1), then these subcharacteristics should also be product characteristics.

From the ontology presented, we derived a framework, shown in Figure 5, following the approach described in subsection 3.2. All classes derived directly from the ontology are prefixed by the character "K", indicating that their objects represent knowledge about the software quality domain. The remainder classes are from the Whole-Part design pattern used. The *Whole* class, for instance, is a handler that maintains a reference to the parts associated to this whole. The interfaces *IWhole* and *IPart* must be implemented by the concrete classes (*KNonMeasurableQC* and *KQualityCharacteristic*, respectively). The methods *whole()* and *part()* on these interfaces provide access to its respective handlers (*Whole* and *Part*).

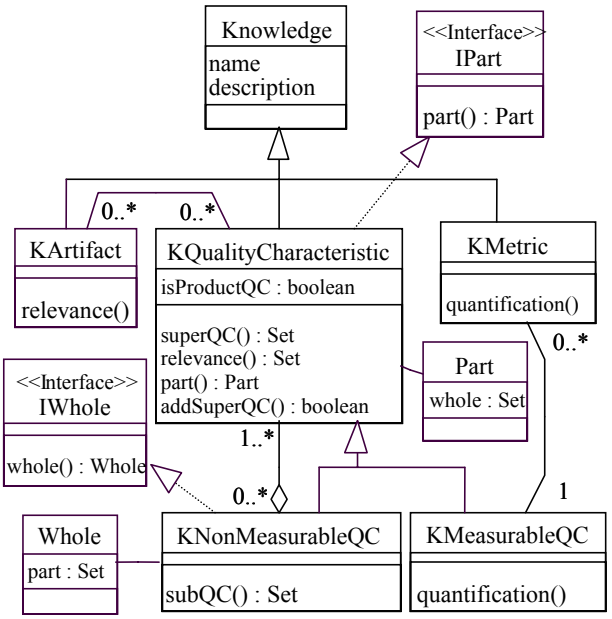


Figure 5 – Part of the Knowledge Package.

The consolidation axiom (C1) was implemented by the method *addSuperQC*, using the consolidation pattern, as shown in Figure 6. Due to space limitation, we do not discuss the ontological axioms mapping here.

```
addSuperQC (KNonMeasurableQC: qc): boolean
{
    boolean result = false;
    if (result = (qc.isProductQC && this.isProductQC))
    {
        superQC.add(qc);
        qc.addSubQC(this);
    }
    return result;
}
```

Figure 6 – Consolidation axiom mapping.

5. Developing with reuse: a tool to support quality planning

In this section, we discuss briefly how the quality framework was used in the development of ControlQ, a tool that supports quality control. The goal is to allow quality planning and tracking. ControlQ's functionality includes:

- quality characteristic and metric knowledge management;
- quality planning, allowing to define quality evaluation activities that will be carried along the project. The quality manager defines for each one of these activities: *when* and *what* will be evaluated, which *quality characteristics* will be evaluated and from which *metrics* these characteristics will be computed;
- quality control, allowing to register the measurement results.

As pointed above, ControlQ was developed from the quality framework. Based on this framework, the tool architecture was composed of two packages: *Knowledge* package, shown in Figure 5, and *Quality Evaluation* package, shown in Figure 7.

The *Knowledge* package directly reflects the concepts of the ontology, representing the common knowledge of this domain. However, to support quality planning and control, other classes are necessary beyond those shaped. To address the specific ControlQ's requirements, we developed the *Quality Evaluation* package. The classes of this package represent specific concepts of the application, necessary to accomplish its goals.

As shown in Figure 7, a *quality control plan* defines all *quality evaluation activities* of a project. Theses activities define not only *what* will be evaluated (an artifact), but also *how* this evaluation will occur, i.e. which *quality characteristics* will be used to evaluate the artifact.

A *non measurable characteristic* must be decomposed into *subcharacteristics* to be computed by the aggregation of their *subcharacteristic* measures. For each one of these *subcharacteristics,* it is necessary to define its *weight* in the measurement. A *measurable characteristic* can be directly measured choosing a *metric* to quantify it. For each *choice*, indicating which *metric* will be used to quantify each *measurable characteristic*, the corresponding measure value is stored.

We can notice that the Quality Evaluation Package requests services from the Knowledge Package, as shown in Figure 8. It is not only a hazard. In fact, we claim that this two-layered architectural style is at the core of a developing with reuse approach. The application level concerns application classes which address the application

requirements. The knowledge level defines domain knowledge, which can be used by several applications.

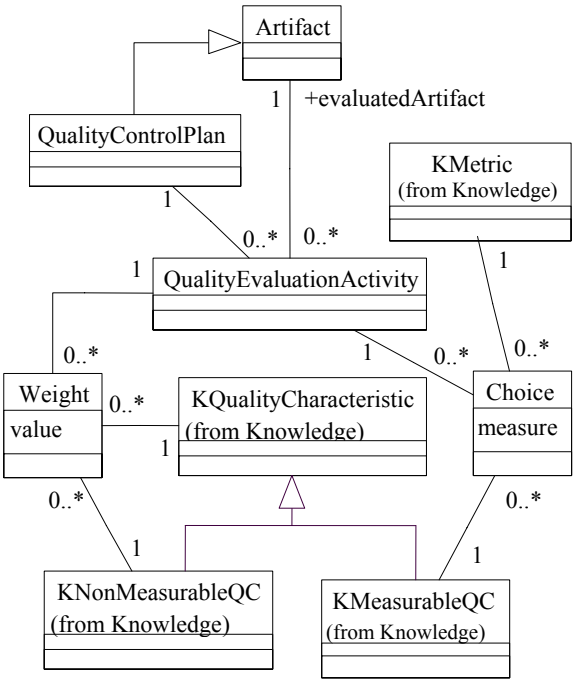


Figure 7 – Part of the Quality Evaluation Package.

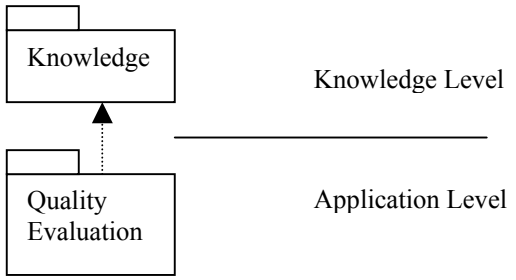


Figure 8 – ControlQ's two-layered architecture.

6. Related work

There are several works that are related to some part of our approach, mainly when considering ontologies development. Uschold and King [12] proposed what they called "a skeletal methodology for building ontologies", defining a small number of stages that they believed would be required for any future comprehensive methodology. In this sense, the method here proposed followed some of their guidelines and stretched it towards a more systematic approach for building ontologies.

In the TOVE (*TOronto Virtual Enterprise)* Project, Grüninger and Fox [13] proposed a method for building ontologies that presents some features that are very proper to its context, the enterprise modeling. In fact, we considered it an applied approach and not a general one. Nevertheless, many guidelines suggested by this method are interesting, such as the use of competency questions to guide the development, and were incorporated in the proposal presented here.

In [14] a set of design patterns for constraint representation in JavaBeans components is presented. Constraints are equivalent to what we call consolidation axioms and our approach to implement these axioms is also based on design patterns. However, these axioms represent only a subset of the knowledge that must be made explicit at the ontological level. Thus we need other mechanisms to capture, for example, ontological axioms, such as the transformation rules we have proposed [11].

7. Conclusions

Ontologies have great potential to deal with software reuse problems. In this paper we presented an approach to systematically develop ontologies and to derive object frameworks from them. This approach is, in fact, an ontology-based approach for developing for reuse. We show its application in the software quality domain. We also discussed how to develop with reuse when using a framework derived from this approach.

References

[1] P.A. González and C. Fernández, "A Knowledge-based Approach to Support Software Reuse in Object-oriented Libraries", in Proceedings of the SEKE'97, 1997.

[2] M. Uschold, "Knowledge level modelling: concepts and terminology", Knowledge Engineering Review, vol. 13, no. 1, 1998.

[3] R.A. Falbo, C.S. Menezes, and A.R.C. Rocha, "A Systematic Approach for Building Ontologies", in Proceedings of the IBERAMIA'98, Lisbon, Portugal, 1998.

[4] R. Jasper, and M. Uschold, "A Framework for Understanding and Classifying Ontology Applications", in Proc. of the 12th Workshop on Knowledge Acquisition, Modeling and Management (KAW'99), Alberta, Canada, 1999.

[5] R.S. Pressman, *"Software Engineering: A Practitioner's Approach"*, 5th Edition, New York: McGraw-Hill, 2000.

[6] M. Grüninger and M.S. Fox, "Methodology for the Design and Evaluation of Ontologies", Technical Report, University of Toronto, April 1995.

[7] W.N. Borst, "Construction of Engineering Ontologies for Knowledge Sharing and Reuse", PhD Thesis, University of Twente, Enschede, The Netherlands, 1997.

[8] S. Cranefield and M. Purvis, "UML as an Ontology Modelling Language", in Proc. of the IJCAI'99 Workshop on Intelligent Information Integration, Stockholm, Sweden, 1999.

[9] T.R. Gruber; “Towards principles for the design of ontologies used for knowledge sharing”, Int. Journal of Human-Computer Studies, vol. 43, no. 5/6, 1995.
[10] A.S.Evans and S.Kent, “Meta-modelling semantics of UML: the pUML approach”, in Proc. of the 2nd International Conference on the Unified Modeling Language. Editors: B.Rumpe and R.B.France, Colorado, LNCS 1723, 1999.
[11] G. Guizzardi, R. A. Falbo and J.G. Pereira Filho, “Using Objects and Patterns to Implement Domain Ontologies”, in Proc. of the 15th Brazilian Symposium on Software Engineering, Rio de Janeiro, Brazil, 2001.
[12] M. Uschold and M. King, “Towards a Methodology for Building Ontologies”, in Proc. Workshop on Basic Ontological Issues in Knowledge Sharing, IJCAI’95, 1995.
[13] M. Grüninger and M.S. Fox, “Methodology for the Design and Evaluation of Ontologies”, Technical Report, University of Toronto, April 1995.
[14] H. Knublauch, M. Sedlmayr and T. Rose, “Design Patterns for the Implementation of Constraints on JavaBeans”, in Proc. of the Net Object Days 2000, Erfurt, Germany, 2000.