Using Objects and Patterns to Implement Domain Ontologies

Giancarlo Guizzardi[1,2]
Ricardo de Almeida Falbo[1]
José Gonçalves Pereira Filho[1]

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

Centre for Telematics and Information Technology, University of Twente [2]
P.O. Box 217, 7500 AE, Enschede, The Netherlands
e_mail: guizzard@cs.utwente.nl

Abstract

Ontologies are becoming an important mechanism to build information systems. Nevertheless, there is still no systematic approach to support the design of such systems using tools that are common to information systems developers. In this paper, we propose an approach for deriving object frameworks from domain ontologies and then we show the application of this approach in the software process domain.

1. Introduction

An Information System cannot be written without a commitment to a model of a relevant world, i.e., commitments to entities, properties, and relations in that world. Data structures and procedures implicitly or explicitly make commitments to a domain ontology [1].

Several projects in Artificial Intelligence have focused on using ontologies to promote knowledge sharing, and to substitute the usual database or object-oriented schema with an ontology, which offers a semantically richer model of the domain [2]. This trend has also acquired followers in the Software Engineering community. However, one of the major drawbacks to a wider use of ontologies in this area is the lack of approaches to insert ontologies in a more conventional software development process.

Since the current leading paradigm in Software Engineering is the object technology, we claim that we need a systematic approach to derive object models from ontologies in order to put ontologies in practice. In this paper we propose an approach to derive reusable object artifacts from domain ontologies. This approach comprises a spectrum of techniques, namely, a set of mapping directives, transformation rules and design patterns. In section 2, we briefly discuss some aspects of ontology development, including a method and a graphical language, and a past experience using them in the software process domain. In section 3 we present a formalism to represent ontologies and a framework that implements the theoretical foundation of this language. In section 4, we present our approach to derive object models and frameworks from domain ontologies, showing how it was applied in the software process domain. In section 5, related works are discussed. Finally, in section 6, we report our conclusions.

2. Ontologies

It is impossible to represent the real world, or even a part of it, with all its details. To represent a phenomenon or part of the world, which we call domain, it is necessary to focus on a limited number of concepts that are sufficient and relevant to create an abstraction of the phenomenon in hand. Thus, a central aspect of any modeling activity consists of developing a conceptualization: a set of informal rules that constrain the structure of a piece of reality, which an agent uses to isolate and organize relevant objects and relations [3].

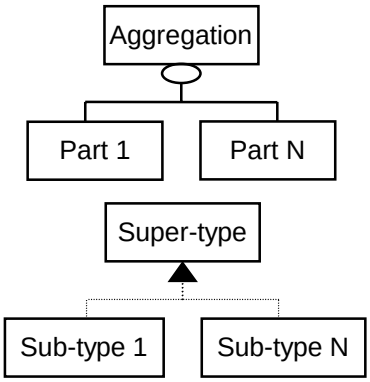
According to Guarino [4], "an ontology is a logical theory accounting for the intended meaning of a formal vocabulary, i.e. its ontological commitment to a particular conceptualization of the world". Based on such definition, an ontology consists of concepts and relations, and their definitions, properties and constrains expressed as axioms. An ontology should not be only an hierarchy of terms, but a fully axiomatized theory about the domain [5].

One of the main benefits of the use of ontologies in software development is the opportunity to adopt a reuse-based approach to the requirements engineering (RE). In traditional Software Engineering, for each new application to be built, a new conceptualization is developed. This reflects on how the RE is currently employed: for each new application, an elicitation phase is accomplished almost always from scratch, focusing on all particularities of the system in hand. This approach is extremely expensive since elicitation is the activity that requires most effort in the software development. Experts are scarce and costly resources but they are essential to this activity, so they should be better used. Therefore, it is important to share and reuse the captured knowledge.

In an ontology-based approach, requirement elicitation and modeling can be accomplished in two stages. First, the general domain knowledge should be elicited and specified as ontologies. These ontologies, in turn, are used to guide the second stage of the RE, when the particularities of a specific application are considered. This way, the same ontology can be used to guide the development of several applications, diluting the costs of the first stage and allowing knowledge sharing and reuse [5].

In [5], we proposed a Graphical Language for Expressing Ontologies (LINGO) and a systematic approach for engineering ontologies. In the RE, the use of a graphical representation is essential in order to facilitate the communication between requirement engineers and experts. In ontology building, such representation is basically a language representing a meta-ontology. Hence, this language has basic primitives to represent a domain conceptualization. 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 1 shows the main notations of LINGO and some of the axioms imposed by the whole-part relation. These axioms form the core of the mereological theory as presented in [7], namely the irreflexivity (A1), anti-symetry (A3) and transitivity (A4) axioms denote sufficient and necessary properties for all kinds of whole-part relations. Axiom (A7) denotes a special kind of part-or relation with non-sharable parts (composition). The remaining axioms complete the theory by defining suitable ontological distinctions.

Both language and method have been used in the development of complex information systems in areas such as Software Process [6], Port Management, Steel Metallurgy, and Media on Demand Management [10]. Although they have proven to be useful, we identify a great concern from the developers: how to put those ontologies in practice, that is, how ontologies can support actual software development? To show our approach to deal with this problem, we discuss in the next subsection the case of a software process ontology.



concept relation

$\forall x \quad \neg partOf(x,x)$ (A1)
$\forall x,y \quad partOf(y,x) \leftrightarrow wholeOf(x,y)$ (A2)
$\forall x,y \quad partOf(y,x) \rightarrow \neg\, partOf(x,y)$ (A3)
$\forall x,y,z \quad partOf(z,y) \wedge partOf(y,x) \rightarrow partOf(z,x)$ (A4)
$\forall x,y \quad disjoint(x,y) \rightarrow \neg\exists z\, partOf(z,x) \wedge partOf(z,y)$ (A5)
$\forall x \quad atomic(x) \rightarrow \neg\exists y\, partOf(y,x)$ (A6)
$\forall x,y \quad partOf(y,x) \rightarrow \neg\exists z\, partOf(y,z)$ (A7)

Figure 1 – Main notation of LINGO.

2.1. Software Process Ontology

In [6], we developed a software process ontology and used it to promote knowledge integration in a Software Engineering Environment (SEE). Part of this ontology is presented in figure 2. In this model, cardinality constraints are used to specify the number of concept instances that can be involved in a relation. The cardinality (0,n) does not impose any restriction and, for that reason, its not graphically represented. Other cardinality possibilities include (0,1), (1,1) and (1,n). Whenever used, these cardinalities incorporate new axioms to the model. In figure 2, cardinality (1,n) in the relation **output** implies that $\mathbf{(\forall a)\ (activity\ (a) \rightarrow (\exists s)\ (output(s,a))}$. Cardinality (1,1) stills adds that $\mathbf{(\forall s,a_1,a_2)\ (output(s,a_1) \wedge output(s,a_2) \rightarrow a_1 = a_2)}$. Although the examples presented above represent only binary relations, the formalism used is expressive enough to model relations of any arity. Likewise, reflective relations (relations between instances of the same concept) and conditional relations (AND and XOR tight relations) can also be represented.

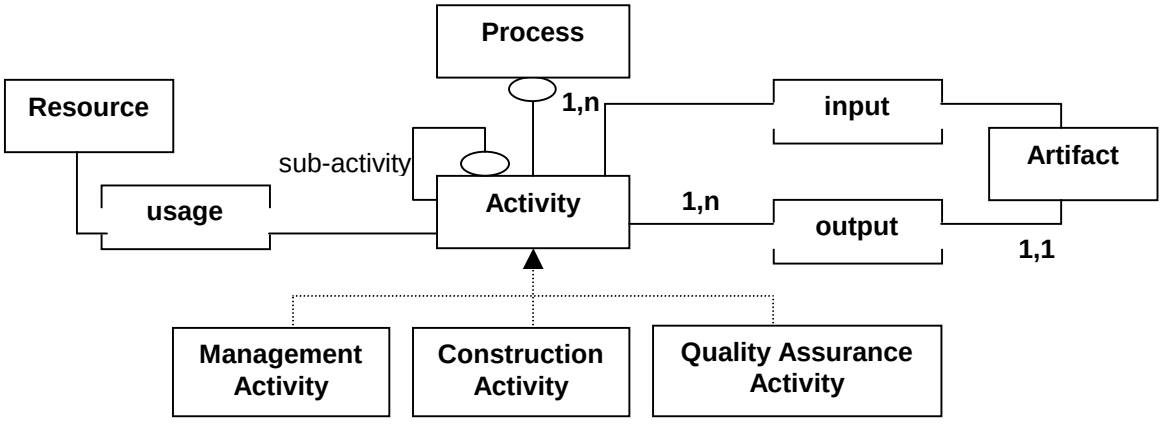


Figure 2 – Part of a Software Process Ontology (SPO).

Besides the epistemological constructs, ontologies explicitly represent knowledge at a signification level through the use of formal axioms. These axioms can be of two types:

consolidation axioms (CA) and *derivation axioms* [5]. 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. Derivation axioms can have root in the meaning of the concepts and relations or in the way these concepts and relations are structured. When axioms are defined to show constraints imposed by the way concepts are structured, they are called *epistemological axioms* (EA). When they describe domain signification constraints, they are called *ontological axioms* (OA) [5]. Cardinality constraints, as discussed above, are examples of epistemological axioms. Several axioms were defined in this ontology. Table 1 shows some of them, indicating their type. It is important to notice that the axioms **(EA4)** and **(EA5)** are directly derived by the usage of the whole-part relation between activities.

Id	Axiom	Type
EA1	$(\forall a)$ constructionActivity(a) $\rightarrow$ activity(a)	Epistemological
EA2	$(\forall a)$ managementActivity(a) $\rightarrow$ activity(a)	Epistemological
EA3	$(\forall a)$ qualityAssuranceActivity(a) $\rightarrow$ activity(a)	Epistemological
EA4	$(\forall a_1, a_2, a_3)$ subActivity(a_1,a_2) $\wedge$ subActivity(a_2,a_3)$\rightarrow$ subActivity(a_1,a_3)	Epistemological
EA5	$(\forall a_1, a_2)$ subActivity (a_1,a_2) $\rightarrow \neg$ subActivity (a_2,a_1)	Epistemological
CA1	$(\forall a, s)$ input(s, a) $\rightarrow$ artifact(s) $\wedge$ activity(a)	Consolidation
CA2	$(\forall a, r)$ usage(r,a) $\rightarrow$ resource(r) $\wedge$ activity(a)	Consolidation
OA1	$(\forall a)$ composedActivity(a) $\leftrightarrow \exists a_1$ subActivity(a_1,a)	Ontological
OA2	$(\forall a, a_1, r)$ (subActivity(a_1,a) $\wedge$ usage(r,a_1)) $\rightarrow$ usage(r,a)	Ontological

Table 1 – Axioms of the Software Process Ontology.

Since the SEE was implemented using objects, we had to derive an object model from the domain ontology. This represented a design problem that was informally solved. More recently, other developers have experienced the same problem. The methodology presented in this paper has been proposed to address this issue, i.e., the systematic object-oriented implementation of domain ontologies.

3. A Hybrid Formalism to Support Ontologies-to-Objects Mapping

As shown in Table 1, we used first-order logic as the language to specify the axioms of the formal theory. First-order logic is widely known for its expressive power and its ontological neutrality, therefore adding minimal ontological commitments. However, due to the goals of this work, it is convenient to adopt a formalism that lies at an intermediate abstraction level, between first-order logic and object-orientation. For this purpose, we used a hybrid approach based on pure first-order logic, relational theory, and, predominantly, set theory.

The choice to create a language mainly based on set theory was highly motivated by an important issue: set theory is a complementary *extensional* perspective to the *intentional* nature of first-order logic and, at the same time, a natural option as a conceptual model for reasoning about objects. To clarify this point, the following example is used: let the intention of the concept mortal be *"A mortal is an entity whose life ceases at a point in time"*. The logic predicate **mortal(x)** states that **x** is a **mortal** and, therefore, the characteristics defined by the intention of this concept applies to **x**. It also (implicitly) states that **x ∈ Mortal**, i.e., to the set of all the elements of the considered world to which the intention of the concept applies. In an object-oriented perspective, if **x** is an instance of **mortal**, it means that **x** belongs to the **mortal**

class, i.e., to the set of all instances of the considered world that share the same properties and the same definition.

Because of these characteristics of set theory, to build a model using the proposed set-based language is an important step in a systematic translation between the logic and the object worlds. Moreover, the language preserves the expression power of the first-order logic without adding significant ontological commitments, therefore, being suitable to play the same role in the axiomatization process. Finally, although formal, the language is kept as simple as possible, defining only what is absolutely necessary to accomplish its goals. The odd convergence of these specific requirements motivated our decision for defining a new set-based formalism instead of using an existent one, such as Z [8].

The theoretical foundation for our formalism is briefly presented below. We also discuss how the primitives of this formalism are related to the LINGO building blocks.

3.1 – Theoretical Foundation for a Set-based language

Sets are collections of zero or more elements whose members are unique and their order is immaterial. Sets can be finite or infinite. Finite sets with a small number of elements are usually represented by the enumeration of its members. Otherwise, they are represented by formation rules or by the definition of the characteristics and properties that all its members must have in common (intention). In our approach, concepts are defined as sets. For example, as mentioned before, the statement $\mathbf{x \in Mortal}$ commits $\mathbf{x}$ to the concept **Mortal**, both intentionally and extensionally.

Another fundamental building block in the LINGO meta-ontology is the primitive *relation*. This primitive represents a semantic link that exists among a set of (one or more) concepts. In our approach, relations are mapped to the synonymous primitive in set theory. In set theory, a n-ary relation can be defined by the n-tuple $\mathbf{R = (C_1,C_2...C_n,\ p(x_1,x_2...x_n))}$, where each $\mathbf{C_i}$ represents a different set involved in the relation, and $\mathbf{p(x_{i,})}$ is a functional predicate open in *n* variables that maps each element from the cross-product $\mathbf{C_1 \times C_2 \times ... \ C_n}$ onto a boolean value. In this case, the set $\mathbf{R^*}$ (solution set) is the subset of $\mathbf{C_1 \times C_2 \times ... \ C_n}$ whose all members $\mathbf{e_i}$ satisfy the predicate $\mathbf{p(e_i)}$.

Using the output relation example, shown in Figure 2, we can illustrate the equivalent description in set theory: $\mathbf{output = ((Activity,\ Artifact,\ output(a,s))}$. For now on, the propositional function $\mathbf{p(x,y)}$ will be used as synonym of the the n-tuple that defines the relation, assuming that the function is defined in some cross-product $\mathbf{C_1 \times C_2 \times ... \ C_n}$.

In set theory, some essential operations are defined to express the relations between sets ($\subseteq$ - proper-subset or $\subset$ - subset; $\cup$ - Union; $\cap$ - Intersection; $\setminus$ - set difference; $\wp$ - power set), properties of sets (# - cardinality**)** and relations between sets and its members ($\in$ - Membership) [11]. In addition to this, the basic logical operators ($\wedge$ - conjunction; $\vee$ - disjuntion; $\oplus$ - exclusive disjunction; $\neg$ - negation; $\rightarrow$ - conditional; $\leftrightarrow$ - biconditional) and quantifiers ($\forall$ - universal; $\exists$ - existential; $\exists!$ - exists one and only one) form the core of the formalism employed in this work. To extend this core, two additional functions have been defined:

- **Imagem (Im):** Let $\mathbf{A}$ and $\mathbf{B}$ be two sets and Φ be the set of all binary relations $\mathbf{R}$ that exist in our considered universe. The function **Im** has two arguments the element $\mathbf{a \in A}$ and $\mathbf{R \in \Phi}$, and it returns an element $\mathbf{B' \in \wp(B)}$. The element $\mathbf{B'}$ is a member of a powerset and, therefore, it is a set. In this case, $\mathbf{B'}$ is the set that contains all members of $\mathbf{B}$ to which $\mathbf{a}$ is associated in the context of the relation $\mathbf{R}$, i.e., the *range* of $\mathbf{a}$ in respect to $\mathbf{R}$. The function **Im** can be formally defined as: $\mathbf{Im:A \times \Phi \rightarrow \wp(B)}$, such that

$\forall a: A, R: \Phi, B' : \wp(B)\ Im(a,R) = B' \leftrightarrow \forall\ b:B'\ (a,b) \in R^*$. Conversely, for each element **a ∈ A** associated to an element **b ∈ B'** in relation **R**, **a** is also a member of the range of the adjunct function **Im(b,R)**, i.e., $\forall a:A, R:\Phi, b:B'\ (b \in Im(a,R)) \leftrightarrow (a \in Im(b,R))$. Using the relation output as an example, a possible valid image set could be: **Im(Planning, output) = {ProjectPlan,Schedule}** and, consequently, **Im(Schedule, output) = {Planning}.** Extending this function definition to n-ary relations $R = \{(C_1,C_2,...,C_n, p(x_1,x_2,...,x_n)\}$, we then have $Im:C_1 \times \Phi \rightarrow \wp(C_2 \times C_3 \ldots \times C_n)$. It is important to notice that **Im** is a distributive function, i.e. $Im(\{ProjectPlan,Schedule\},output) = Im(ProjectPlan, output) \cup Im(Schedule, output)$. Consequently, we can define the general form for this function as $Im: \wp(A) \times \Phi \rightarrow \wp(B)$.

- **Selection (σ):** Our second extension to the core formalism is the Relational Algebra *selection* operator [12]. Relational algebra is a query procedural language composed by a set of operations that act on relations. This operator acts on a relation by selecting tuples that satisfy a given predicate. Since the associations between concepts and their properties constitute relations, this operator can be used to select elements within a set that share a common feature. Generally, let **A** be a set whose members have a given **w** property. Let **B** be the subset of **A** whose members have the property **w** related through the operator **op** to the expression **z**. In relational algebra terms, **B** is called a selection of **A** and this can be formalized as $B \leftarrow \sigma_{(w\ op\ z)}(A)$. The operator **op** can be any relational or logical operator, depending on the type of the operands.

3.2 – The `Set` framework

Figure 3 shows a support framework that plays a fundamental role in our ontology-to-Java objects mapping process. This framework implements the mathematical properties described by the theoretical foundation presented above. The methods of the `Set` class are summarized in table 2.

Operation	Operation prototype	Functionality
$\supseteq$	`A.contains (B)`	Verify if set **B** is contained in set **A**
=	`A.equals (B)`	Verify if set **B** is equals to set **A**
$\cup$	`A.union (B)`	Returns the set $A \cup B$
$\cap$	`A.intersection (B)`	Returns the set $A \cap B$
#	`A.cardinality ( )`	Number of elements of set **A**
$\{C \mid C \subseteq A\}$	`A.subset("C")`	Returns the set **C** if **C** is a subset of **A**
/	`A.difference(B)`	Returns the difference between two sets
∈	`A.in(x)`	Verify if the element **x** belongs to set **A**
+	`A.add(x)`	Adds the element **x** to the set **A**
-	`A.remove(x)`	Removes the element **x** from the set **A**
Im	`Set.Im(a,r1)`	Returns the set **Im(a,r1)**
Im	`Set.Im(A,r1)`	Returns the set **Im(A,r1)** where **A = {a1,..,an}**
σ	`Set.select("w",op,"z")`	Returns the selection $\sigma_{(w\ op\ z)}(A)$

Table 2 – Brief description of the methods implemented by the `Set` class

The `Set` class is a generic container that is able to hold extension sets for all kinds of concept instances. To be accessible, each member of a set must have a unique identifier. The `SetElement` interface deals with this requirement, providing an identification mechanism

through the `getKey` method. For an instance of any class to be held in a `Set`, it must implement the `SetElement` interface. Consequently, the `Set` class is actually a set of `SetElement` instances. The primary key for these elements is typed as `Object`, which is the top-most class in the Java hierarchy. This is done in order to give the application classes total freedom regarding implementation decisions.

The framework also defines two other classes: `PersistentSet` and `MemberSet`, both sub-types of `Set`. The former is a set that is able to handle its permanent storage in total transparency from the perspective of the class users. When the `store()` method is invoked in a `PersistentSet`, the class performs the serialization of all its members. The original state of the objects (as well as their relations) can be afterwards restored by the invocation of the `retrieve`( ) method.

Finally, persistent sets can be used as an interesting alternative to implement databases [10]. Using this paradigm, a database can be seen as a **family** $\Im$ (set of sets), which contains all the sets existing in the application. Since, in this case, each set will be a member of another set, they must also be univocally identifiable. The `MemberSet` is, thus, provided to enable this situation.

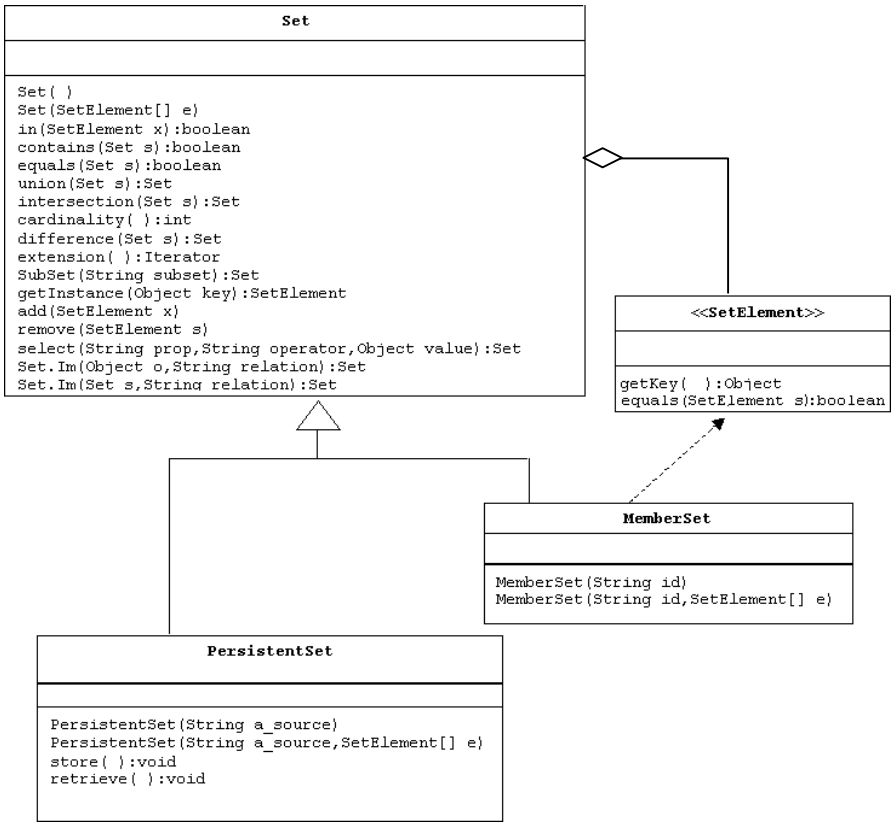


Figure 3 - Framework that implements the mathematical type Set.

4. Using Objects and Patterns to Implement Domain Ontologies

The problem of consistently generating computational infrastructures from conceptual models has been known for a long time by the software engineering community as the so-called *Impedance Mismatch Problem (IM)* [13]. In the scope of this work, the conceptual models are domain ontologies and the computational infrastructures are object-oriented

frameworks. The use of domain ontologies to realize the domain analysis activity in a software engineering process contributes with innumerous benefits [10]. However, the impedance mismatch problem is amplified: instead of performing just one step to translate between two levels of abstraction (conceptual models to computational infrastructures), two steps are necessary. The first step is to translate from an ontological level model (domain axiomatized theory) to an epistemological conceptual model (conceptual view of class diagrams) without loosing the explicit representation of knowledge. The second step is the translation between the domain model to its computational concretization - an activity that, in domain engineering terms, is called *domain design*.

Our systematic approach to address this two-level IM problem is composed of a set of directives, design patterns and transformation rules. 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. Contrariwise, design patterns and transformation rules are applied to the mapping process of consolidation and ontological axioms, respectively. The rational application of these conceptual tools supported by the Set framework is able generate consonant Java implementations for the ontology axiomatizated theory. In section 4.2 the mapping directives are discussed. The Design Patterns and the transformation rules are presented in sections 4.3 and 4.4. The following subsection shows the formalization of the ontology depicted in figure 2.

4.1 – Ontology formalization using the set-based language

The first step in our approach for mapping domain ontologies to objects is the complete axiomatization of the domain theory using the set-based formalism. Besides the derivation (epistemological and ontological) and consolidation axioms, we need definition axioms to express concepts and relations. Given the model of Figure 2, the following definition axioms can be derived. The notational convention used is: **(CD)** - concept definition axioms, and **(RD)** - Relation definition axioms.

(CD1) P = Process **(CD2) A = Activity** **(CD3) R = Resource**
(CD4) S = Artifact **(CD5) M = Management Activity**
(CD6) C = Construction Activity **(CD7) Q = Quality Assurance Activity**

(RD1) procAggregation = (Process, Activity, procAggregation(p,a))
(RD2) subActivity = (Activity, Activity, subActivity (a_1,a_2))
(RD3) usage = (Resource, Activity, usage(r,a))
(RD4) input = (Artifact, Activity, input(s,a))
(RD5) output = (Artifact, Activity, output(s,a))

The following axioms translate the epistemological (EA) and ontological (OA) axioms shown in Table 1 and those derived from cardinality constraints (**EA6 – EA8**) to the Set-based formalism:

(EA1) $C \subset A$ **(EA2) $M \subset A$** **(EA3) $Q \subset A$**
(EA4) $\forall a_1,a_2,a_3$ $(a_1 \in subActivity(a_2)) \wedge (a_2 \in subActivity(a_3)) \rightarrow (a_1 \in subActivity(a_3))$
(EA5) $\forall a_1,a_2$ $(a_1 \in subActivity(a_2)) \rightarrow (a_2 \notin subActivity(a_1))$
(EA6) $\forall$ s:Artifact #Im(s,output) = 1
(EA7) $\forall$ a:Activity #Im(a,output) $\geq$ 1
(EA8) $\forall$ p:Process #Im(p,procAggregation) $\geq$ 1

(OA2) $\forall$a:Activity ComposedActivity(a) $\leftrightarrow$ Im(a,subActivity) $\neq \varnothing$
(OA2) $\forall$a:ComposedActivity, r:Resource usage(a,r) $\leftrightarrow$ r $\in$ Im(Im(a,subActivity),usage)

4.2 – Mapping directives

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. 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 worthwhile 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.

A class defines a formation rule for its instance and, therefore, can be seen and manipulated as a set in a meta-level architecture. Consequently, the classification relations in the formalism do not require any specific implementations, i.e., relations such as $\mathbf{a} \in \mathbf{A}$, are totally resolved by the programming language typing mechanism through the creation of an object **a** of type **A**.

For the mapping of *relations*, there are some issues that still must be discussed. Figure 2 shows a relation **output** between the concepts **Activity** and **Artifact**. In our approach, this relation is translated to an association between the corresponding two classes in the object model and both classes have a method `output()`. In this case, with the invocation of method `output()` in an object a_1 of type `Activity`, it is possible to have access to all the artifacts produced by a_1. This resulting set is formally specified by the formula **Im(a1,output))**. Likewise, the method invocation in an artifact instance s_1 returns its producer activity, or, **Im(s1,output)**. The returned type of the relation methods depends directly on the cardinality axioms associated to the relation. For instance, since in the scope of the **output** relation an **Activity** may produce several artifacts, **output** is mapped to a `Set` variable in the `Activity` class and, hence, this is the type returned by the invocation of the synonymous method on this class. When a relation has a cardinality axiom imposing an inferior limit equals to 1, this constraint is reflected in the class constructors ensuring the establishment of the relation.

Like classification, subsumption does not require any additional implementation, i.e., subtype-of relations among concepts can be directly mapped to generalization/specialization relations among classes. An axiom like $\mathbf{M} \subset \mathbf{A}$ states that the concept **ManagementActivity** is a subtype of **Activity** (intentionally and extensionally). Since all elements contained in **M** also belong to the set **A**, every Management activity ($\boldsymbol{m} \in \mathbf{M}$) is an activity as well. The subsets of a concept are actually partitions of that concept inside that domain. For example, there is no element in the set **Activity** that does not belong either to **ManagementActivity**, **QualityAssuranceActivity** or **ConstructionActivity**. For this reason, the concept that represents a super-type is always mapped to an abstract class.

Finally, the directives consider non-trivial mappings, e.g., n-ary relations, relation properties and conditional relations. At last, they advise the choice between primitives to model a domain entity (Guarino discussion about sortals, temporal neutrality and ontological rigidity is a good example of this [9]).

4.3 – Consolidation Axioms

Considering consolidation axioms, we identified two cases to address. Consolidation axioms that concern to object types, do not need any mapping since we are working with a strongly typed language – Java. This is the case of axioms (CA1) and (CA2), shown in Table 1. Nevertheless, there is another type of 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. Examples of this type of axiom can be found in the Mereology theory presented in Figure 1. For a relation to be set between a composition and a candidate part two properties must hold: asymmetry and exclusiveness (A7). Asymmetry is a property that is formed by the conjunction of the axioms (A1) and (A3), i.e. the irreflexivity and anti-symmetry constraints respectively. According to the transitivity axiom (A4) this property must be reified recursively. In other words, let **x** be a composition, for **y** to be set as a part of **x** the following relation properties must hold: (i) **x** cannot be equal to **y**; (ii) **x** cannot be a part of **y** or be a part of any part of **y**; (iii) **y** cannot already have a relation established with another whole. The following axiom formalizes this property: (A8) $\mathbf{- \forall x,y\ composition(x) \wedge (y \in partOf(x)) \rightarrow asymmetric(x,y) \wedge \neg\exists z\ (y \in partOf(z)).}$

Generally speaking this type of consolidation axioms will have the form $\mathbf{\forall x{:}X,\ y{:}Y\ r_1(x,y) \rightarrow (preCondition_1) \wedge (preCondition_2) \wedge ... \wedge (preCondition_n)}$. This generic form can be transposed to a pattern that should guarantee the evaluation of each of precondition before a relation can be established. The figure 4 shows this *Consolidation Pattern* on the left and its application to the axiom (A8) above.

The Consolidation Pattern uses the pattern *Template Method* defined in [14]. In this case the *template method* is the method $\mathbf{setr_1}$ and *hook methods* are the methods responsible for evaluating the fulfillment of the preconditions.

```
Public class X
{
 public boolean setr1 (Y y)
 {
  boolean result = false;
  if (result = (checkCondition1(...)
  && checkCondition2(...) ... &&
  checkConditionn(...))
  {
     r1.add(y2);
     y.setr1(this);
  }
  return ok;
 }
private boolean checkCondition1(...)
private boolean checkCondition2(...)
private boolean checkConditionn(...)
}
```

```
Public class Composition
{
 public boolean setComposition(Part c)
 {
  boolean result= false;
  if asymmetry(c) &&  exclusiveness(c)
  {
    result=true;
    part.add(c);
   (c.part()).setComposition(whole);
  }
  return result;
}
public boolean asymetry(IPart c);
public boolean exclusiveness(IPart c);
}
```

Figure 4 - The Precondition Pattern

4.3.1 – The Whole-Part relation.

The figure 1 presents the theory (mereology) embodied by a generic whole-part relation. Notwithstanding, the underlying axioms implied by the proposed notation are not well mapped to aggregations in an object model, i.e., UML notation for aggregation does not guarantee the fulfillment of the imposed constraints. Since this theory is valid in any type of

whole-part relations, a generic strategy defining a solution pattern can be modeled. Figure 5 depicts our *Whole-Part* ontological pattern. This pattern is built using the *PreCondition* pattern described in the previous section and the *Delegation* pattern presented in [14]. By using these patterns the `Whole` class is able to guarantee to its associated concrete class (`A`) the verification of the suitable set of constraints before a relation between `A` and its candidate parts can be established. This service is offered to the concrete class through a delegated method (`setPart`).

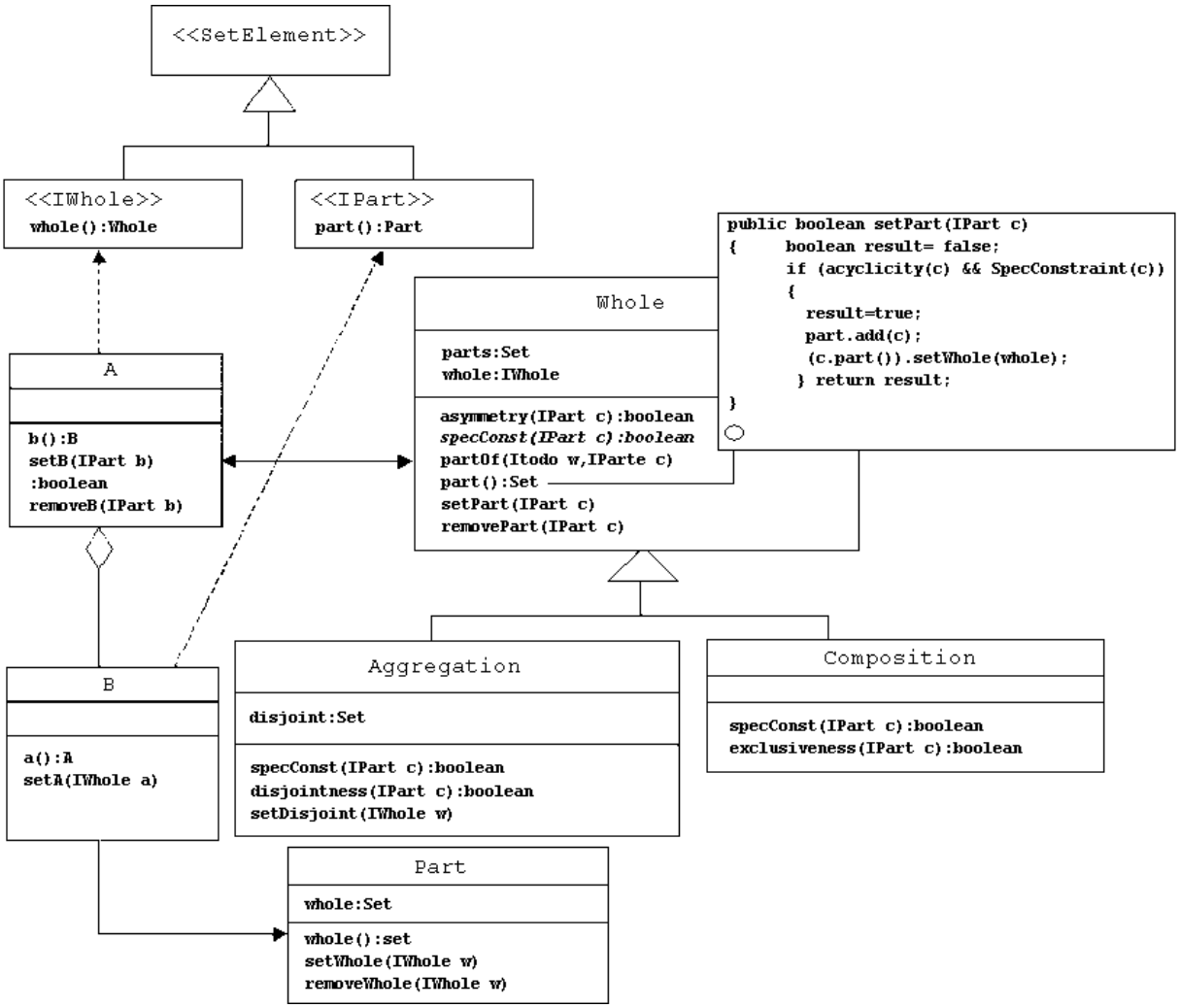


Figure 5 - The Whole-Part pattern (WP)

To be able to derive the `setPart` method through the usage of the *PreCondition* pattern another axiom had to be created. The following axiom extends axiom (**A8**) to generic whole-part relations: (**A9**) - $\forall x,y\ (y \in partOf(x)) \rightarrow asymmetric(x,y) \wedge specificConstraint(x,y)$. For the Composition relation the predicate **specificConstraint** represents the exclusiveness property (**A7**). Conversely, for an aggregation relation, it must assured that the part does not aggregate any whole disjoint to this one and therefore **specificConstraint** represents the axiom (**A5**).

The `Whole` class is a handler that maintains a reference to the parts associated to this whole. It also encapsulates the consolidation axioms of the generic whole-part theory. Additionally, it is hierarchically divided in two subclasses, namely `Aggregation` and

`Composition`, each of them encapsulating specific consolidation constraints represented by the predicate specificConstraint in the axiom (**A9**). One can observe in figure 5 that the method `setPart` in this class was generated by the application of the *PreCondition* pattern on the axiom (**A9**). The `specConstraint` method is declared abstract on class `Whole`. Its concrete implementations are provided by the subclasses `Aggregation` and `Composition`.

The interfaces `IWhole` and `IPart` must be implemented by the concrete classes (`A` and `B`). The methods `whole()` and `part()` on these interfaces provide to the concrete classes the access to its respective handlers (`Whole` and `Part`). The guarantee of implementation of these methods allows the handlers to perform precondition verification tasks in a generic way.

4.4 – Ontological Axioms

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. The axiom (**OA1**), for instance, answers to the following question: *for a given composed activity a_1, which resources are used by this activity?* The solution set for this question must be returned by the invocation of the method `usage()`in an object a_1 of the `Activity` class. However, for this type of methods to be derived from ontological axioms, a set of transformation rules were defined. These transformation rules are presented below.

T0: $\forall$ x:X, $\forall$ y:Y $r_1(x,y) \leftrightarrow y \in C \Rightarrow$

$Im(x, r_1)$:Type $\equiv$ C, such that if # $Im(x, r_1) = 1$ then Type = Y else Type = Set

This rule states: if for each instance **x** of type **X**, **x** is engaged with all instances **y** from set **C** (and only instances of this set) in a relation **r1**, the set returned by the function **Im(x, r1)** will be exactly **C**. The type returned by the method that implements the function in the derived class depends on the cardinality of the relation. Hence, if **x** is related to only one instance of **Y**, the returned value shall be of type **Y**, otherwise, it shall be of type `Set`, in the case a set of **Y**.

T1: $\forall$ x:X, $\forall$ y:Y $r_1(x,y) \leftrightarrow (y \in C) \wedge$ (property$_1$(y) operator expression), such that expression = property$_2$(x) $\oplus$ constant $\Rightarrow$ $Im(x, r_1)$:Type $\equiv \sigma$ property(y) operator expression**(C).**

Let **D** be a subset of **C** in which all its elements has one of its properties satisfying a specific relation with an given expression. This expression can denote a property of **x** (instance of **X** with which **C** is associated through the relation **r1**) or a constant value. An example of the former case is presented as follows: Let the concept HumanResource be a subtype of Resource. Suppose that an instance of human resource is used by an activity if: (i) the resource is allocated to the same process that the activity belongs; (ii) the "experience required" to perform the activity is lower then the "level of experience" property of the human resource. **$\forall$ h:HumanResource, a:Activity usage(h,a) $\leftrightarrow \sigma$** level of experience(h) > experience required(a) **(Im(Im(h, allocation), aggregation)).**

In this case the set returned by the function **Im(h,usage)** will be exactly the set **(Im(Im(h, allocation), aggregation))** after the application of the relational algebra selection operator. Like in the previous rule, the type returned by the method `usage()` implemented in the class `HumanResource` depends directly on the cardinality of the relation.

T2: $Im(x, r_1) \Rightarrow x.r_1()$
T3: $r_1(x,y) \Rightarrow x.r_1()$
T4: $r_1(x) \Rightarrow x.r_1()$

A relation **r1** between two concepts **X** and **Y** is mapped in the classes that represent these concepts to methods named after the relation. For instance, given an instance **x**, the invocation **x.r1()** returns the set of objects from **Y** associated to **x** in the relation **r1**.

T5: A SetTheoryOperation *a* ⇒ A.SetTheoryOperationImplementation(*a*)

This rule deals with the translation between the essential set theory operations (section 3.1) and the corresponding method implemented in the Set class. For instance, the set theory expression **A ∩ C** is translated to `A.intersection(C)`, where A and C are instances of the class `Set`.

T6: Im(A, r_1) ⇒ Set.Im(A," r_1")

T7: $\sigma_{\text{property(x) operator property(y)}}$(C) ⇒ Set.select(property(x),operator, property(y), C)

The rules **T6** and **T7** promote the replacement of the mathematical function *Image* and the *Selection* operator by the correspondent syntaxes through which they are implemented in the `Set` class. The method `select` (that implements the selection operator) receives as the operator parameter a `String` whose value follows the convention described below.

(i) The operands are two objects: = (`equals`), ≠(`not_equals`)

(ii) The operands are two basic types: =, ≠, ≥(`GTET`), ≤ (`LTET`), <(`LT`), >(`GT`)

(iii) The operands are an object and a set: ∈ (`in`),∉ (`not_in`)

T7: $x.r_1():Y \equiv C \Rightarrow$

```
public class X
 {
    public Y r1( )
   {
      return C;
}
 }
```

Finally, this last rule directly translates the axiom written in its left side to the implementation correspondent syntax in the chosen programming language. All the references to the instance **x** existent in the scope of set **C** (to which **x** belongs) are replaced by the Java reserved word `this`, so that references to methods of the same class will be made.

The code fragment below shows the derivation process for the axiom (**OA2**), and also its implementation in the `Activity` class.

(OA2) ∀ a:ComposedActivity, r:Resource usage(a,r) ↔ r ∈ Im(Im(a,aggregation),usage)

```
1. Im(a,usage):Set ≡ Im(Im(a, aggregation),usage)                    OA2, T0
2. a.usage():Set ≡ Im(a.aggregation(),usage)                           1, T2
3. a.usage():Set ≡ Set.Im(a.aggregation(),"usage")                     2, T6
4. public class Activity                                          3, T7
  {
     public Set usage()
     {
            return Set.Im(this.aggregation(),"usage");
     }
  }
```

Figure 6 depicts the class diagram derived from the process ontology presented in Figure 2. It is important to notice that the cardinality convention used by UML has exactly the opposite direction to the one used by LINGO. The reasons for that are explained in [5].

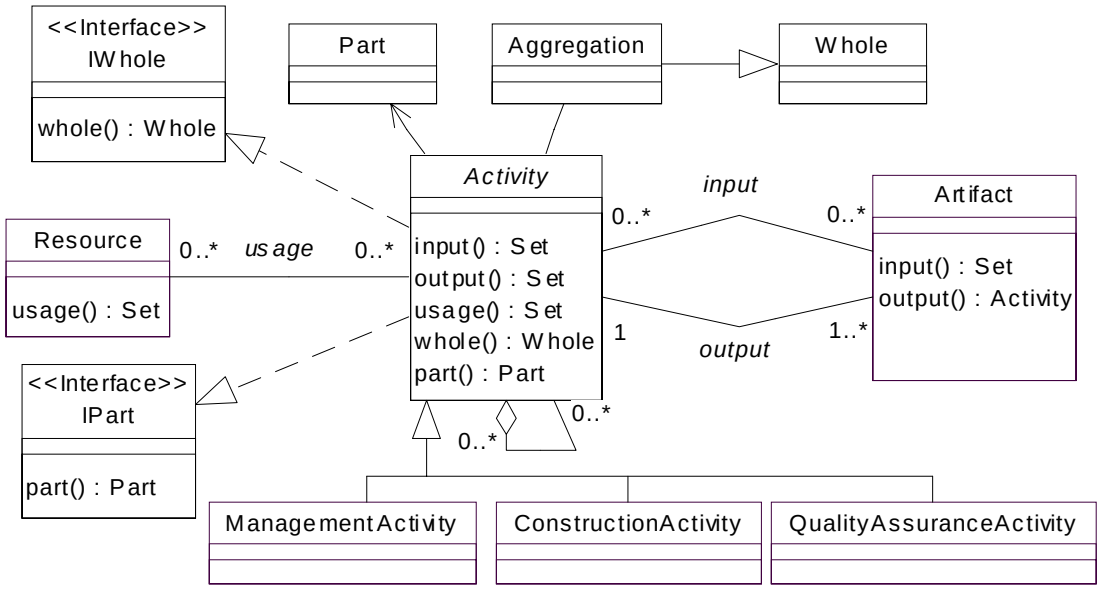


Figure 6 – Software Process Framework

5. Related Work

The Peirce project is an international collaborative effort to build a conceptual graphs workbench [15]. To accomplish interoperation among the different tools produced in the context of the project, a mathematical ontology was proposed and a software library was derived. The ontology contains taxonomic hierarchies for mathematical objects such as sets, groups, categories, relations, functions, preorders, partial orders and lattices. In [15] a specification for a Set class is formalized in several languages (Z, KIF, Conceptual Graphs - CG) and a set of C++ contracts is derived, showing pre/pos-conditions for the operations of the type. However, due to the focus of this project, the emphasis is on the object-oriented implementation of a CG processor and not on how to create object-oriented artifacts from a conceptual model.

Another interesting approach to address the impedance mismatch between the ontology and object-oriented abstraction levels is the use of design patterns. In [16] a set of design patterns for constraint representation in JavaBeans components is presented and computation reflection mechanisms are used to evaluate these constraints at run-rime. Likewise, in [17], three design patterns are used to promote Java implementation for ontologies represented in the OKBC knowledge model [18]. In this case, ontology concepts are either represented by reflection-backed JavaBeans classes, by an Active Object-Model (AOM), or by a mixed approach based on extending the classes from the AOM.

Constraints are equivalent to what we call consolidation axioms. These axioms represent only a subset of the knowledge that the must be made explicit at the ontological level. Constraints basically define pre-conditions that must be satisfied for a relation to be consistently established. Our approach to implement these axioms is also based on design patterns.

Finally, in [19], one finds an approach to create object models such as CORBA IDLs and Java classes and interfaces from Geographic Information Systems (GIS) Ontologies. The papers suggest the automatic generation of interfaces and IDLs from Ontolingua models. These interfaces constitute ontology skeletons that are, afterwards, complemented by

implementation code written in Java. Ontology editors, such as Ontolingua, have the ability to create CORBA IDL headers automatically, however, in this case, the behavior implementation for the interface methods would still rely on an ad-hoc translation process. Moreover, interfaces alone are not expressive enough to incorporate the knowledge related to all kinds of consolidation axioms, let alone, ontological derivation axioms.

6. Conclusions

Since Aristotle's theory of substance (objects, things and persons) and accidents (qualities, events and process) ontologies have been used in philosophy as a foundation for representing theories and models of reality. Their main purpose is to formally make explicit the semantic distinctions existent in portion of the world, accounted as a domain. Hayes [20] introduced the use of ontologies in Computer Science (more specifically in Artificial Intelligence). Since then, ontologies have been employed in areas such as computational linguistics, knowledge engineering, information integration and multi-agent systems. In addition to that, ontologies have been used in application areas such as enterprise modeling [21] and GIS [19], among several other examples.

In the software engineering realm, domain ontologies have been used to model the foundation over which meta-enviroments can be constructed [6]. Moreover, they contribute to the domain engineering phase, promoting a reuse-based practice in the requirements engineering level [10].

Nevertheless, few of the ontology construction methodologies lead to executable code and, there was still no systematic approach to fully promote their integration to the object-oriented software development practice. For this reason, most of the object-oriented implementations of domain ontologies rely on informal derivation processes.

In this paper a contribution to address this problem is presented: a methodology through which object-oriented frameworks can be systematically derived from domain ontologies. To accomplish this goal, we also proposed a formal representation language. The mathematical foundation of this language (set-theory) highly contributed to the feasibility of our approach. This is mainly due to its suitability to bridge the conceptual and implementation abstraction levels, respectively represented by first-order logic axioms and object models.

The derivation methodology proposed comprises a spectrum of techniques, namely, directives, design patterns and transformation rules. This paper shows how these conceptual tools together with the supporting `Set` framework can establish a sound path between our formally axiomatized theories and a related consonant implementation in Java classes.

We use the *Software Process Ontology* as an example to illustrate the methodology. The ontology presented was over-simplified due to the lack of space. In despite of that, the methodology has been tested in several case studies, ranging from software process [5,6] to video on demand management theories [10]. In all these experiments, we found the methodology effective, mainly because of: (i) its ability to capture the domain knowledge without imposing additional ontological commitments; (ii) its ability to successfully derive object frameworks capable of answering the relevant competency questions.

It is important to notice that our methodology is highly focused on the structural part of domain ontologies. Consequently, a natural extension of this work is to develop an approach to address the dynamic aspects of domains, i.e. behavioral ontologies.

References

[1] Chandrasekaran, B., et al., *"What are Ontologies, and Why Do We Need Them?"*, IEEE Intelligent Systems, pp. 20-26, January/February 1999.

[2] Valente, A., et al., *"Building and (Re)Using an Ontology of Air Campaign Planning"*, IEEE Intelligent Systems, pp. 27-36, January/February 1999.

[3] Guarino, N., *"Understanding, building and using ontologies"*, Int. Journal Human-Computer Studies, 46(2/3), February / March 1997.

[4] Guarino, N., *"Formal Ontology and Information Systems"*, In: Formal Ontologies in Information Systems, N. Guarino (Ed.), IOS Press, 1998.

[5] Falbo, R.A., et al.; *"A Systematic Approach for Building Ontologies"*. Proceedings of the IBERAMIA'98, Lisbon, Portugal, 1998.

[6] Falbo, R.A., et al.; *"Using Ontologies to Improve Knowledge Integration in Software Engineering Environments"*, Proceedings of SCI'98/ISAS'98, USA, July, 1998.

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

[8] Spivey, J. M. *"Understanding Z: A specification language and its formal semantics"*, Cambridge University Press, 1988.

[9] Guarino N. *"The Ontological Level"*. In R. Casati, B. Smith and G. White (eds.), Philosophy and the Cognitive Sciences, Vienna, Hölder-Pichler-Tempsky 1994.

[10] Guizzardi, G. *"A methodological approach for reuse-oriented software development based on formal domain ontologies"* (in portuguese), Federal University of Espírito Santo, Master Thesis, 2000.

[11] Roitman J. *"Introduction to modern set theory"*, Wiley-Interscience, New York, 1990.

[12] Silberchatz, A. et al. *"Database System Concepts"*, 3. ed. McGraw-Hill, 1997.

[13] Woodfield S.N. *"The impedance Mismatch between Conceptual Models and Implementation Environments"*, International Conference on Conceptual Modeling (ER'97), Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling, Los Angeles, California, Nov, 1997.

[14] Gamma E. et al. *"Design patterns : elements of reusable object-oriented software"*, Addison-Wesley, 1995.

[15] Ellis G; Callaghan S; *"A specification of a Set Class in Peirce"*, online: http://citeseer.nj.nec.com/29926.html, Oct, 1995.

[16] Knublauch H.; Sedlmayr M.; Rose T., *"Design Patterns for the Implementation of Constraints on JavaBeans"*, NetObjectDays2000,Erfurt, Germany, 2000.

[17] Knublauch H., *"Three Patterns for the Implementation of Ontologies in Java "*, OOPSLA'99 Metadata and Active Object-Model Pattern Mining Workshop, Denver, CO, USA, 1999.

[18] Grosso W. et al. *"Knowledge Modeling at the Milennium (The Design and Evolution of Protégé-2000)"*, Knowledge Aquisition Workshop, Banff, Canada, 1999.

[19] Fonseca, F. Egenhofer M. *"Knowledge Sharing in Geographic Information System"*, In: P. Scheuerman, (Ed.) The Third IEEE International Knowledge and Data Engineering Exchange Workshop, Chicago, 1999.

[20] Hayes P. *"The Naive Physics Manifesto"*, Expert Systems in Microeletronics age", D. Ritchie Ed., Edinburgh University Press, 1978, pp 242-270.

[21] Gruninger, M., and Fox, M.S., *"The Role of Competency Questions in Enterprise Engineering", Proceedings of the IFIP WG5.7 Workshop on Benchmarking - Theory and Practice*, Trondheim, Norway, 1994.