Building Ontologies in a Domain Oriented Software Engineering Environment

Paula Gomes Mian and Ricardo de Almeida Falbo
Federal University of Espírito Santo, Vitória – Brazil, 29060-900
{pgmian, falbo}@inf.ufes.br

Abstract

Ontologies can be used in Domain Oriented Software Engineering Environments (DOSEEs) to organize and describe knowledge and to support management, acquisition and sharing of knowledge regarding some domain. However, ontology construction is not a simple task. Thus, it is necessary to provide tools that support ontology development. This paper discusses the use of ontologies to support domain-oriented software development in ODE, an Ontology-based software Development Environment, and presents ODEd, an ontology editor developed to satisfy the requirements for an ontology editor in a DOSEE. These requirements include the definition of concepts and relations using graphic representations, automatic generation of some classes of axioms, derivation of object frameworks from ontologies, and ontology instantiation and browsing.

Keywords: Ontologies, Ontology Editors, Software Engineering Environments, Domain Oriented Software Engineering Environments.

Classification: Software Engineering. This article is submitted to the general congress.

1 Introduction

Software quality and productivity can be improved by the use of *Software Engineering Environments* (SEEs). They can automate several tasks of the software development process, making easier to control it. But, in many cases, developers are building systems in non-familiar domains. *Domain Oriented Software Engineering Environments* (DOSEEs) are a special class of SEEs that uses domain knowledge to guide software developers across the several phases of the software process. DOSEEs organize the application domain knowledge facilitating the understanding of the problem during system development [1].

According to Oliveira [1], *ontologies* are a good way to describe and organize domain knowledge in a DOSEE. However, building ontologies is not a simple task. It involves the specification of concepts and relations that exist in the domain, besides their definitions, properties and constrains, described as axioms [2].

In this paper, we present *ODEd*, an ontology editor developed in the context of ODE (*Ontology-based software Development Engineering*) [3]. In section 2 we briefly discuss some aspects of DOSEEs and the use of ontologies to promote knowledge management in them. Section 3 discusses some aspects of ontology building. Section 4 discusses the main requirements for an ontology editor in a DOSEE. Section 5 presents ODEd and how it supports these requirements. In section 6 we discuss related works. Finally, in section 7 we report our conclusion and future work.

2 Domain Oriented Software Engineering Environments and Ontologies

One great difficulty in software development is that, many times, developers do not know or are not familiarized with the domain in which the software is being developed. To deal with this problem, several research groups have proposed to improve and to evolve Software Engineering Environments (SEEs) to support software development considering peculiar characteristics of the domain [1]. The identification of the need to support domain-oriented software development and the

limitations of the conventional SEEs to support this kind of development drove to the definition of Domain Oriented Software Engineering Environment (DOSEE).

To build a DOSEE, it is necessary to define a model that turns explicit the basic conceptualization of the domain. Ontologies have been used for this propose and, therefore, they can be very useful to support domain orientation in a SEE.

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. An ontology consists of concepts and relations, and their definitions, properties and constrains expressed as axioms [2].

DOSEEs can take several advantages from the use of ontologies. In fact, a DOSEE should support domain-oriented software processes. Several process models have been proposed with this purpose, almost always establishing parallel tracks for domain engineering and software engineering. In the domain engineering track, ontologies can act as both a domain model and a component in a repository of reusable artifacts. It can also be used for structuring this repository.

In [2] it was proposed an ontological approach for domain engineering that considers three main activities: domain analysis, specification of a reuse infrastructure and implementation of that infrastructure. This ontological approach involves ontology development (domain analysis), mapping ontologies into object models (infrastructure specification) and the development of Java component (infrastructure implementation). The first phase - ontology development - involves the following activities [2,5]:

- *Purpose identification and requirements specification*: it concerns to clearly identify the ontology purpose and its intended uses, that is, the competence of the ontology. To do that, competency questions are used;
- *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 domain experts;
- *Ontology formalization*: aims to explicitly represent the conceptualization captured in a formal language;
- *Integration of existing ontologies*: during the capture and formalization steps, it could be necessary to integrate the current ontology with existing ones, in order to seize established conceptualizations;
- *Ontology evaluation*: 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 as those proposed by Gruber [6];
- *Ontology documentation*: 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.

To support this ontology-based domain engineering process in ODE (Ontology based Development Environment) [3], *ODEd* (ODE's Ontology Editor) was built. ODE is a Process-Centered SEE that is developed using ontologies. The main goal of ODEd is to support ontology development in ODE and to evolve it to a DOSEE.

3 Requirements of an Ontology Editor for a DOSEE

Analyzing several ontology editors available, each one developed for a different context, we were able to identify a set of requirements that should be satisfied by these editors to support ontology development. Since we are interested in tools for ontology development in a DOSEE, we will evaluate these requirements from this point of view.

Ideally, ontology editors should support an ontology development process. Therefore, an ontology editor should provide services to satisfy each one of the activities presented in section 3. To support purpose identification and requirement specification, an ontology editor should support *competency questions definition* (R1). During the ontology capture phase, the use of a graphical representation is essential in order to facilitate the communication between domain engineers and experts. Thus, an ontology editor should support the *definition of concepts, relations and its descriptions, preferentially, using a graphical language* (R2). To support ontology formalization, an ontology editor should give support to the *definition of axioms* (R3), to allow writing them informally and/or formally. Requirements R1 to R3 are important for editors in any context and they are considered basic requirements to an ontology editor.

An ontology editor should also provide services for *ontology integration* (R4), *ontology evaluation* (R5) and *documentation of the development process* (R6). Integration services (R4) can be important for DOSEEs, because a domain is, usually, wide and rich in details. A way to build large domain theories is to subdivide them in sub-theories. Each sub-theory is represented by an ontology [1], and to compose the domain theory, it is necessary to integrate the sub-theories, defining relationships among them, i.e., to integrate ontologies.

Ontology evaluation is important to guarantee that an ontology describes the domain it intends to model and to accompany its evolution in the DOSEE. During the usage of the domain theory in the DOSEE, it could be necessary to extend an ontology. Therefore, it is necessary to verify if the new definitions do not contradict other definitions formulated previously.

Finally, the documentation of the ontology development process can be important to share the knowledge about ontology development in a DOSEE. In spite of its importance, this requirement is not considered essential, because it is not vital for organization and representation of knowledge in a DOSEE.

An ontology editor should support not only construction but also ontologies usage. An important ontology use is knowledge acquisition. Thus, an ontology editor should support *ontology instantiation* (R7). Ontology instantiation in DOSEEs is important because instances of domains' concepts can be defined and stored in domain knowledge bases that can be used to support domain understanding [1].

An ontology editor should also allow the *neutral authoring of ontologies* (R8), i.e., to build the ontologies in a "neutral" language and, latter, in the project phase, define what technology will be used to represent them. This is important, mainly when several applications are developed using different technologies. In a DOSEE, an ontology can be developed, and then translated for implementing components using these technologies.

Since an ontology represents a knowledge model for a domain, all applications regarding that domain could share the vocabulary defined by the ontology. If the ontology editor is capable of generating software components from the ontology, they could be shared by these applications. The *specification of components for domain applications* (R9) starting from the ontologies built in the editor promotes, in a DOSEE, knowledge reuse, once the components are built using the common vocabulary defined in the ontologies and several applications can reuse then.

The implementation of the requirement of neutral authoring allows the ontology editor to *represent ontologies in multiple formats* (R10). This characteristic was not considered essential for a DOSEE since its main focus is to define domain models through ontologies.

Finally, a DOSEE should use the knowledge defined during the software development. Thus, *domain investigation* (R11) activities should be incorporated to the development processes accompanied by the SEE [1]. Therefore, an ontology editor should offer mechanisms to browse the ontologies defined.

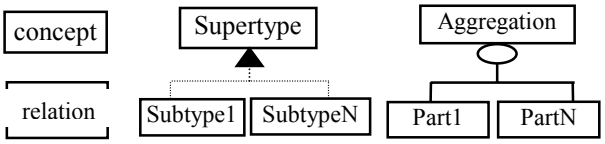
These requirements might be useful in several contexts. However, in our opinion, some of them are especially important for an ontology development tool in a DOSEE. An ontology editor in this context should support development of software by integrating knowledge to the SEE and allowing instantiating and browsing the knowledge repository. The tool should be able to generate a computational infrastructure in order to reuse it in conventional software engineering process. Therefore, we consider the requirements R4, R7, R9 and R11 as the most important for DOSEEs, especially R9 (components for domain applications) and R11 (domain investigation).

4 Ontology Development in ODEd

To show an example of the ontology development in ODEd we present a *Port Domain Ontology*. Due to limitations of space, we present only part of this ontology.

The first step of the ontology development is purpose identification and requirement specification. To support this phase, ODEd allows the user to define competency questions. Among the competency questions of the ontology described in this paper are: *What is the structure of an harbor?; What is the type of a ship?; Where is a ship from, i.e. what is its nationality?; Which ships are anchored on a harbour? In which deck?; Which freights does a ship transport?; What is the nature of a freight?; Which types of freight is a deck capable to operate with?; and Is some type of freight compatible with other, i.e. can they be operated jointly?.*

Once the competency questions are defined, it is possible to start the ontology capture. In [2], it was proposed LINGO, a Graphical Language for Expressing Ontologies. LINGO has the basic primitives to represent a domain conceptualization, i.e., 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 [2]. Therefore, axioms can be automatically generated. These axioms concern simply the structure of the concepts and are said epistemological axioms (EA). Figure 1 shows the main notations of LINGO and some of the axioms imposed by the whole-part relation.



(EA1) $\forall x \ \neg partOf(x,x)$
(EA2) $\forall x,y \ partOf(y,x) \leftrightarrow wholeOf(x,y)$
(EA3) $\forall x,y \ partOf(y,x) \rightarrow \neg \ partOf(x,y)$
(EA4) $\forall x,y,z \ partOf(z,y) \wedge partOf(y,x) \rightarrow partOf(z,x)$
(EA5) $\forall x,y \ disjoint(x,y \rightarrow \neg \exists z \ partOf(z,x) \wedge partOf(z,y)$
(EA6) $\forall x \ atomic(x) \rightarrow \neg \exists y \ partOf(y,x)$

Figure 1 - LINGO's main notations and some axioms.

Anti-reflexivity (EA1), anti-symmetry (EA3) and transitivity (EA4) axioms denote sufficient and necessary properties for whole-part relations. The remaining axioms complete the theory by defining suitable ontological distinctions.

ODEd uses LINGO as a graphic language to describe ontologies, allowing the automatic generation of the LINGO's notations built-in axioms. Using these notations during ontology capture, an ontology engineer is also defining the group of axioms that they represent. ODEd uses this feature to automatically generate these types of axioms.

Besides epistemological axioms, other axioms can be used to represent knowledge. These axioms can be of two types: consolidation axioms (CA) and ontological axioms (OA) [2]. 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.

Figure 2 shows part of the Port Ontology in LINGO. In the port domain, a *harbor* is composed by *docks*, port installations where *ships* moor. Each dock is divided in *decks*, areas for ship

anchorage. A deck may be used to operate any type of products or it may have equipments to operate specific goods. The *capability* relation defines what kind of freight a deck can operate. Several types of ships circulate in the harbor, such as *passenger*, *marine* or *freight ships*. Ship's *nationality* indicates which country the ship comes from. The *freight transported* in a ship is *characterized* by its *freight nature*, which indicates the type of a freight (bagged, container, grain in nature, oil, etc). Freight natures are *compatible* if they can be operated in the same port infrastructure.

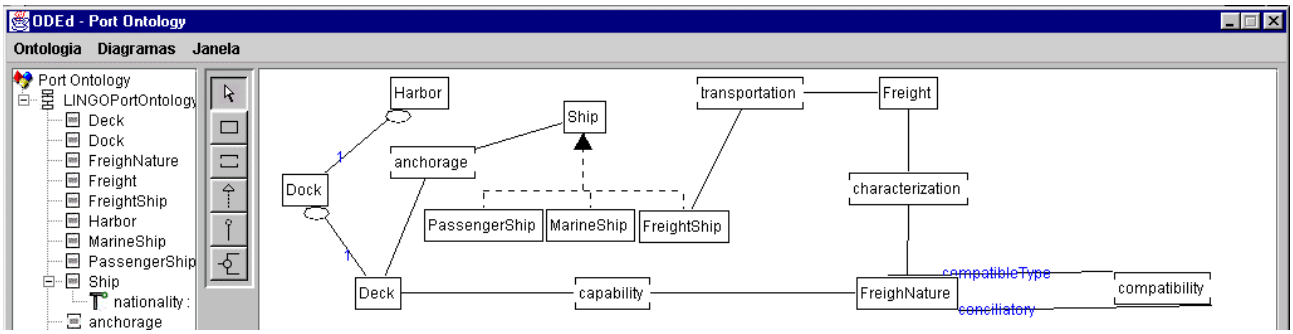


Figure 2 - The LINGO diagram of the Port Ontology.

Table 1 presents some axioms of the Port Ontology, indicating their type. Axioms (EA1) to (EA3) were derived from the ships hierarchy. (EA4) to (EA6) are directly derived by the usage of the whole-part relation among harbor, dock and deck. The axioms (OA1) and (OA2) are related to the *compatibility* relation.

Table 1 - Some axioms of the Port Ontology.

ID	Axiom	ID	Axiom
EA1	$(\forall s)$ (freightShip(s) $\rightarrow$ ship(s))	EA5	$(\forall h,do,de)$ (part(do,h) $\wedge$ part(de,do)$\rightarrow$ part(de,h))
EA2	$(\forall s)$ (marineShip(s) $\rightarrow$ ship(s))	EA6	$(\forall h) \neg$ part(h,h)
EA3	$(\forall s)$ (passengerShip(s) $\rightarrow$ ship(s))	OA1	$(\forall n_1,n_2,n_3)$ compatibility(n_1,n_2) $\wedge$ compatibility(n_2,n_3) $\rightarrow$ compatibility(n_1,n_3)
EA4	$(\forall$ h, do) (part(do, h) $\rightarrow \neg$ part(h , do))	OA2	$(\forall n_1,n_2)$compatibility(n_1,n_2) $\rightarrow$compatibility(n_2,n_1)

Beyond generating LINGO's pre-defined theories, ODEd also allows the user to compose his/hers own theories and apply then to relations in the ontology. This approach to represent theories is similar to that presented in [7]. The core idea is to use a categorization that organizes axioms. Axioms are classified according to association properties, such as *anti-reflexivity, anti-symmetry, atomicity, disjointed, exclusivity, reflexivity, symmetry* and *transitivity*. These properties are used to compose theories associated to relations of the ontology.

Each association property has a class associated. Methods of these classes are responsible for checking if the association properties represented by the class hold. For instance, the `antiSymmetry` method of the `AntiSymmetry` class is responsible for checking if a relation is anti-symmetric.

Besides creating the classes to represent association properties, it is necessary to define how they can be composed into theories. To support theories composition in ODEd, the *PreCondition pattern* [5] is used. This pattern uses the *Template Method pattern* [8]. In ODEd, the hook methods are those of the association property classes responsible for evaluating the fulfillment of the preconditions of the corresponding relation theory. The generic format of the PreCondition Pattern used in ODEd is*: $\forall$x:X, y:Y relation(x,y) $\rightarrow$ (associationProperty1) $\wedge$ (associationProperty2) $\wedge$ ... $\wedge$ (associationPropertyN)*. Object frameworks generated by ODEd incorporate this to compose and verify relation theories.

The *compatibility* relation, for example, has the following properties: transitivity - if a freight nature *n1* is compatible with *n2* and *n2* is compatible with *n3*, then *n1* is compatible with *n3* (OA1); and symmetry - if a freight nature *n1* is compatible with *n2* then *n2* is compatible with *n1* (OA2).

Therefore, transitivity and symmetry properties should be incorporated to the *compatibility* relation theory, as shown in Figure 3. This form allows the user to associate association properties to relations.

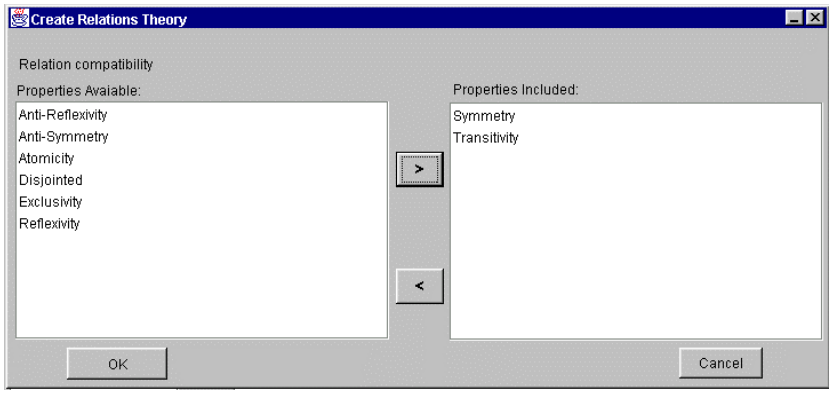


Figure 3 - Properties of the Compatibility relation.

4.1 Importing Concepts and Relations

Since the Port Domain is very complex, it is necessary to sub-divide it in order to facilitate its understanding. Thus, during the Port Ontology development, it was built a Port Structure sub-theory, considering aspects regarding the composition of an harbor, and this was integrated to the Port Ontology.

ODEd supports ontology integration in a very simple way. It is possible to import concepts from existing ontologies to the current one. If more than one concept is imported and there are relations between them, these relations are also incorporated to the ontology. Then, these concepts can be connected to the concepts of the current ontology. For example, in Figure 2, the *Deck* concept was imported from the Port Structure sub-theory and a new relation between *Deck* and *Ship* was created (*anchorage*).

If an imported concept or relation is removed from the original ontology, it is automatically removed from the ontology to where it was imported. It means that if the concept *Deck* is removed form the Port Structure sub-theory, it will be removed from the Port Ontology, as well as the *anchorage* relation.

4.2 From Ontologies to Object Frameworks

ODEd supports codifying the resulting ontologies in Java. To do that, it works based on the approach defined in [5] that defines a set of directives, design patterns and transformation rules for deriving object frameworks from ontologies. 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). The design patterns and transformation rules are applied in axioms mapping. The application of these guidelines is supported by a Java Set framework that implements the abstract data type Set [5]. In its current stage, ODEd considers the mapping directives and some design patterns. But, since ODEd does not yet support axiom definition, except those described through theories, the transformation rules are not being treated. In the next sections, we briefly discuss how ontology implementation is support by ODEd to derive the Port Ontology framework.

4.2.1 Mapping Directives

In the case of the Port Ontology, the classes `Ship` and `FreightNature` were derived from the corresponding concepts, as well as the associations `anchorage`, `transportation`, and `compatibility`, as shown in Figure 4. Properties of the concepts were mapped as attributes of the corresponding classes, as is the case of the property *nationality* of the concept *Ship*, which was

mapped as the attribute `nationality` in the class `Ship`. Also, for each derived attribute, methods to `get` and `set` values were created.

Still considering the mapping of relations, there are other issues that must be discussed. First, since in an ontology relations are bi-directional, the corresponding associations must be navigable in both directions. Thus, the associations are implemented as attributes, and there are methods in both classes to return them. The returned type of the relation methods depends directly on the cardinality associated to the relation [5]. For instance, since in the scope of the *anchorage* relation, several ships may anchor in a deck, the attribute `anchorage` is mapped to a `Set` variable in the class `Deck` and, hence, this is the type returned by the invocation of the `getAnchorage` method on this class. If the maximum cardinality were 1, the return type of the `getAnchorage` method would be a `Ship` object.

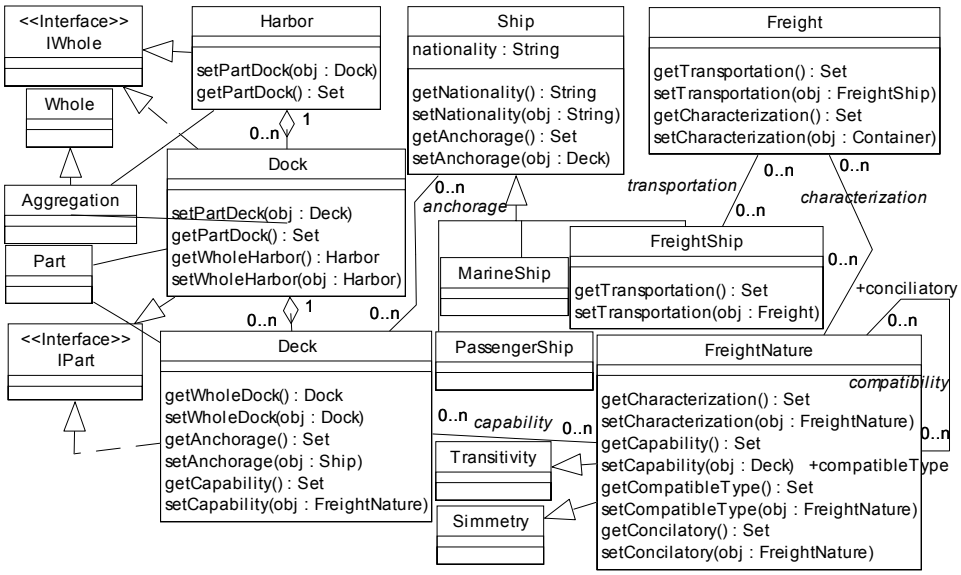


Figure 4 - The Port Framework generated by ODEd.

Reflexive relations, such as *compatibility*, are also mapped as associations, and generate methods for each association end. The name of these methods is, instead of the relation's name, the name of the roles played by the concept (`getCompatibleType` and `getConciliatory`). Whole-Part relations also are treated by specific methods. In Figure 4, the aggregation relation between *Harbor* and *Dock* originates the methods `getPartDock` and `getWholeHarbor` in `Harbor` and `Dock` classes, respectively. Subtype-of relations among concepts can be directly mapped to inheritance among classes. So, axioms (EA1) to (EA3) do not require any special treatment. In our example, the subtypes of ship give rise to the following sub-classes: `PassengerShip`, `MarineShip` and `FreightShip`.

4.2.2 Mapping Axioms

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 of whole-part relations. To deal with this problem, Guizzardi et al. [5] proposed the *Whole-Part Pattern*. In this pattern, the `Whole` class is able to guarantee to its associated concrete classes the verification of the suitable set of constraints before a relation between them can be established. The interfaces `IWhole` and `IPart` must be implemented by the concrete classes.

In the framework derived from the Port Ontology (Figure 4), the class `Dock` implements the interfaces `IWhole` and `IPart` respectively. Likewise, it is related to the handlers `Aggregation` and

`Part`. The class `Dock` has attributes of `Part` type and of `Aggregation` type. As shown in the code fragment below, the access to the decks of a dock is made through `Aggregation`. The inclusion of a new deck is made by including a new part in the aggregation variable. The axioms (EA3) to (EA5) are checked when the method `setPart` is invoked.

```
public class Dock implements IWhole, IPart
{  Aggregation a = new Aggregation();
   Part p = new Part();
   public void setPartDeck(Deck d)
   {  a.setPart(d);  }
   public Set getPartDeck()
   { return a.part();  }
}
```

The theory incorporated to the *compatibility* relation in Figure 3 is presented in the code fragment below. The class `FreightNature` is related to the classes `Symmetry` and `Transitivity` through the attributes `s` and `t`, respectively. Before setting a freight nature as compatible with the current freight nature, the compatibility theory is checked. As show in the code below, the method `setCompatibleType` of the class `FreightNature` is responsible for checking the *compatibility* theory before setting the freight nature **`n`** as compatible to the current freight nature (**`this`**). To verify the axiom (OA2), for example, the method `t.transitivity(this,n,"getCompatibleType")` of the `Transitivity` class is executed. This method evokes the method `getCompatibleType` of the freight nature **`n`**. Suppose that the current freight nature (**`this`**) is already compatible to the freight nature **`n`**. So, the current freight nature (**`this`**) would be returned when the `getCompatibleType` method is executed for the object **`n`**. Since the *compatibility* relation is also symmetric, it would be redundant to set **`n`** as a freight nature compatible to **`this`** (because of the symmetry property, **`n`** is already compatible to **`this`**). In this case, the transitivity property would be false and setting **`n`** as compatible type of `this` should not be allowed. The `transitivity` method would return false and the instance would not be created. In the other hand, if the current freight nature (**`this`**) is not compatible with any freight nature that is compatible with **`n`**, **`n`** can be added to the `compatibleType` list of the current freight nature (**`this`**).

```
public class FreightNature
{ Set compatibleType = new Set();
  Symmetry s = new Symmetry();
  Transitivity t = new Transitivity();
  public void setCompatibleType(FreightNature n)
  {  if ((s.symmetry(...)) && (t.transitivity(...)))
     { compatibleType.add(n); }
  } }
```

4.3 Browsing Ontologies

ODEd provides automatic generation of hypertexts based on the ontologies designed. Using these hypertexts, developers are able to browse and search the domain's concepts, relations and constrains.

The language chosen to build these documents was XML. To generate the XML documents, a set of tags was defined to represent the ODEd's meta-ontology. The ontologies' data (concept, relation, properties, etc.) were introduced in the XML files, marked with theses tags.The tutorials are presented to the user as HTML documents. In order to do so, the ODEd uses XSL (eXtensible Style sheet Language).

Figure 5 shows the hypertext derived from the Port Ontology. It is possible to visualize all ontology's concepts and relations and their definitions and properties. From the *Ship* concept, for example, the user can browse its sub-types and visualize their definition.

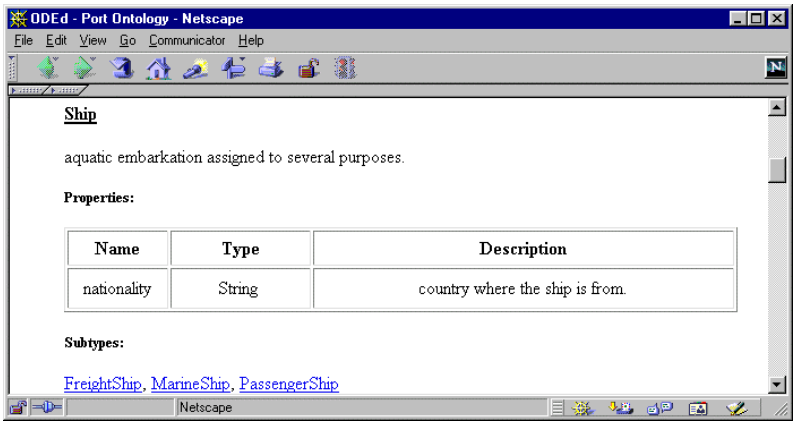


Figure 5 - Browsing the Port Ontology.

4.4 Ontology Instantiation in ODEd

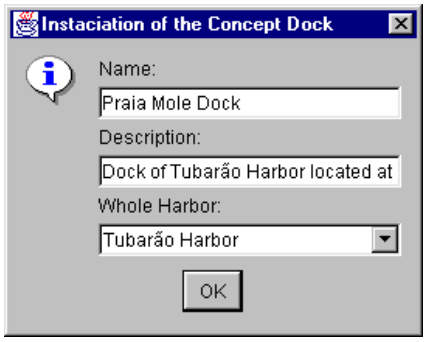
ODEd also intends to support an ontology-based knowledge acquisition approach. But, before beginning the knowledge acquisition activity, it is necessary to create a database to store the information about the instances of concepts and relations that compose the ontology. So, databases are automatically generated.

For each concept a table is created. Since every concept should have name and description, a class `Knowledge` is created in the frameworks generated by ODEd and there is a respective table *`Knowledge`* that contains those attributes. Every table is related to the *`Knowledge`* table to map the inheritance between the concepts and *Knowledge*, except those that are derived of concepts that are subtypes in some hierarchy, such as *`FreightShip`*. These tables have keys that indicate which are the tables that represent their super-types.

Relations (1:1) and (1:N) are mapped as foreign keys and relations (N:N), such as *compatibility*, are mapped in associative tables, whose primary keys are the identifiers of the concepts involved in the relation. To map whole-part relations, there is a unique table *`WholePart`* in which the values of identifiers of instances that belong to whole-part relations are stored.

Once the database was created, it is possible to begin the instantiation. ODEd uses an approach similar to that implemented in [9]. Customized windows are generated, based on the ontology contents, to allow the instance data input.

(a)



(b)

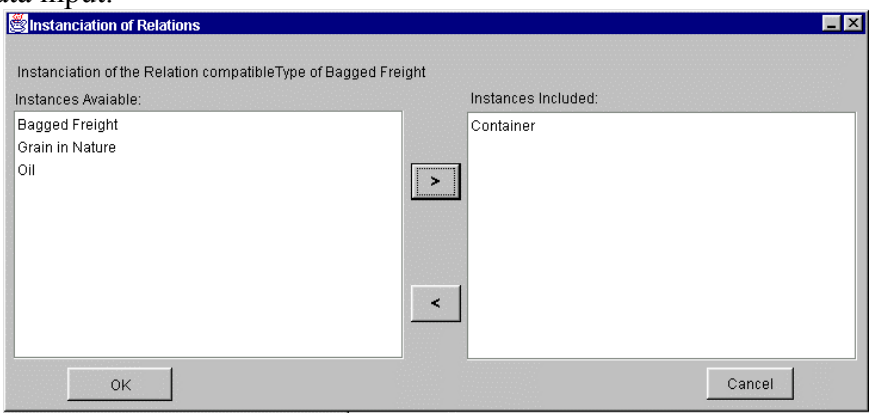


Figure 6 - Creating an instance of the concept Dock (a) and the relation compatibility (b).

Figure 6a shows an instantiation of the *Dock* concept. To store the value of the properties *name* and *description* presented in the form, when a register is inserted in the table *`Dock`*, a register is also automatically inserted in the table *`Knowledge`*. If the super-types of the concept have properties, its values should be associated to the subtype instantiated. Since *Dock* does not have super-types, no other property was inserted in the form. Associations with minimum cardinality 1 are instantiated

when the concept is instantiated. Thus, since every dock is part of an harbor, the user should choose, from a list of harbors in the database, the one the dock belongs to. In Figure 6a, the *Praia Mole Dock* belongs to the *Tubarão Harbor*.

For instantiating relations (N:N), the user should choose one among the instances of one of the concepts involved in the relation, and then choose, among the instances of the other concept, those that are linked to the first. Figure 6b presents the instantiation of the *compatibleType* role of the *compatibility* relation for the *Bagged Freight* instance. In the example, *Bagged Freight* is compatible to *Container*. A list of all freight natures already instantiated is exhibited and the user may select those that the *Bagged Freight* instance is compatible to.

It is worth to point, however, that before instantiating a relation, its theories must be verified. For example, suppose that *Container* (C) is compatible to another freight nature: *Special Container* (SC). Since the *compatibility* relation is symmetric, *Special Container* is compatible to *Container*. After creating the instance of the previous example (Figure 6b), the symmetry property assures that *Container* is compatible to *Bagged Freight* (BF). Due to the transitive property, the following relation among instances is created: BF↔C↔SC. It means that *Special Container* already is compatible to *Bagged Freight*. Therefore, it would be redundant to set *Special Container* as a compatible type of *Bagged Freight*. If the user tries to insert this relation instance in the database, an error message will be sent and the instance will not be created.

The classes of the framework generated by ODEd are responsible for checking the theories. It is necessary, therefore, that those classes have access to the database generated to store the instances of the ontologies, analyzing the soundness of the theories. To do so, besides generating the database and the domain classes in the framework, a persistence layer is also automatically generated by ODEd.

For each concept or relation that has a domain class in the framework, a shadow class is created in the persistence layer. Each one of those classes presents the necessary functionality to implement the persistence of the objects, such as to save, to remove or to update an object and to retrieve a group of objects.

Relations that generate associative tables and do not have their own shadow classes are handled by the shadow classes of the concepts involved in the relation. The *anchorage* relation, for example, is manipulated by the classes `ShipPers` and `DeckPers`. Each one of these shadow classes has a method, as shown below, to insert a register in the associative table *`anchorage`*.

```
public void insertAnchorage(String obj, String obj1)
{ String sLocSQL;
  Statement oLocSt;
  sLocSQL="INSERT INTO anchorage(idoShip,idoDeck)VALUES('"+obj+"','"+obj1+"')";
  oLocSt.execute(sLocSQL);
  ...   }
```

Similarly, in the class `FreightNaturePers`, there is a method `insertCompatibility` to create instances of the *compatibility* relation. Before inserting a register in the table *`compatibility`*, it is necessary to check the theory of the relation *compatibility*. Thus, the `FreightNature` class is associated to the `FreightNaturePers` class and the insertion method of the shadow class is called by the method `setCompatibleType`, responsible for checking the *compatibility* theory, as shown below.

```
public class FreightNature
{ FreightNaturePers pers = new FreightNaturePers();
  public void setCompatibleType(FreightNature n)
  {  if ((s.symmetry(...)) && (t.transitivity(...)))
     { compatibleType.add(n);
       pers.insertCompatibility (this.getIDO,n.getIDO); }
  } }
```

5 Related Work

There are many ontology editors described in the literature. *OntoEdit* [7] pursues the modeling of ontologies such that graphical means exploited for modeling of concepts and relations scale up to axiom specifications (using RDFS). The core idea is to use an axiom categorization. This categorization is centered on axiom semantic meaning rather than syntactic representation.

OILEd [10] supports the construction of ontologies in OIL. The editor allows the definition of concepts and relations and also supports the definition of some pre-defined axioms. OILEd has reasoning services that supports ontologies construction, integration and verification.

The *Java Ontology Editor* (JOE) [11] was developed to help users build and browse ontologies. It enables query formulation at several levels of abstraction. JOE provides a graphical user interface for editing ontologies, using Entity Relationship diagrams to represent them.

Protége-2000 [9] aims to support knowledge acquisition, and to reach interoperability with other knowledge representation systems. It has classes, instances of these classes, slots representing attributes of classes and instances, and facets expressing additional information about slots. Protégé-2000 generates knowledge-acquisition forms automatically based on the types of the slots and restrictions on their values allowing ontology instantiation.

Most of these tools previously cited emphasize the definition of concepts and relations, but they have none or little support to constrains definition. The most interesting initiative is the creation of axioms templates in OntoEdit [7]. This approach aids the construction of axiom classes that has similar structure, but it can not be applied to axioms that do not fit in its classification. This approach was incorporated to ODEd in order to facilitate axioms definition, but it is still necessary to define how to represent other types of axioms. Finally, in ontology instantiation, ODEd uses a similar approach to Protégé-2000 [9].

Reasoning services are an important feature [10] because they can be used in ontology evaluation. Other desirable services provided by some of these tools are the support to the cooperative work and the automatic generation of ontology documentation in HTML [10, 9]. This last feature is addressed by ODEd but no reasoning service is available. ODEd does not provide functionalities for collaborative ontology development such as versioning, integration and merging of ontologies.

Despite of being an important requirement for ontology design, only JOE [11] uses some kind of graphic representation. But it uses Entity Relationship models that are not adequate to ontology development [5]. ODEd adopts LINGO, a graphic language specially designed for ontology representation.

It is worthwhile to point that most of the editors previously mentioned were developed to support ontology design in the context of Semantic Web. None of them was developed to integrate domain knowledge in a SEE and then they do not emphasize a domain engineering approach, such as ODEd. ODEd automatically generates object frameworks from the ontologies. Those frameworks can be used to support software development in the domain. Also, since ODEd adopts LINGO as a graphical language, its built-in axioms and those axioms of theories defined by the user are automatically incorporated to the framework.

6 Conclusions

In this paper, we presented ODEd, an ontology editor that supports ontology development using graphic representations, besides promoting automatic generation of some classes of axioms and derivation of frameworks from ontologies. ODEd was built to support an ontology based approach for domain engineering in ODE.

As shown in Table 2, besides supporting the basic features of a tool for ontology development (requirements R1 to R3), ODEd supports all requirements considered essential for an ontology editor in a DOSEE (R4, R7, R9 and R11) and most of the requirements considered important in this context. However, important activities such as ontology integration (R4) and evaluation (R5) were not completely addressed. To support these activities, it will be necessary to integrate reasoning services in ODEd.

Table 2 - Requirements supported by ODEd.

Requirement	ODEd	Requirement	ODEd
Competency Questions	Yes	Ontology Instantiation	Yes
Concepts and relations using a graphic language	Yes	Neutral Authoring	No
Definition of Axioms	Partial	Components for Domain Applications	Yes
Ontology Integration	Partial	Multiply Representations	Partial
Ontology Evaluation	No	Domain Investigation	Yes
Ontology Development Process Documentation	No		

In the approach presented, ODEd is capable to derivate epistemological, consolidation and ontological axioms coming from relation theories. However, other ontological axioms, which do not fit in any category of axioms properties, could also be necessary to describe a domain. Therefore, it is necessary to define how these axioms should be treated in ODEd. It is being developed an axiom editor to ODEd that uses DAML+OIL and KIF to describe other axioms, and that allows ontology evaluation using a JTP inference engine.

Acknowledge

The authors thanks CAPES and CNPq for the financial support to this work.

References

1. Oliveira, K.M., Rocha, A.R.C., Travassos, G.H. and Menezes, C.S. Using Domain Knowledge in Software Engineering Environments. SEKE'99, Germany, 1999.
2. Falbo, R.A., Menezes C.S. and Rocha, A.R.C. A Systematic Approach for Building Ontologies. *Proc. of the IBERAMIA'98*. Lisboa, Portugal, 1998.
3. Falbo, R.A., Guizzardi, G., Natali, A.C.C., Bertollo, G., Ruy F.B. and Mian, P.G. Towards Semantic Software Engineering Environments. *Proc. of 14th Int. Conference on Software Engineering and Knowledge Engineering, SEKE'02*. Ischia, Italy, 2002.
4. Guarino, N. Formal Ontology and Information Systems. *Formal Ontologies in Information Systems.* IOS Press, 1998.
5. Guizzardi, G., Falbo R.A. and Pereira Filho J.G. Using Objects and Patterns to Implement Domain Ontologies. *Journal of the Brazilian Computer Society*. Vol. 8, no. 1, July 2002.
6. Gruber, T.R. Towards principles for the design of ontologies used for knowledge sharing. *Int. Journal of Human-Computer Studies.* Vol. 43, no. 5/6, 1995.
7. Staab, S. and Maedche, A. Ontology Engineering beyond the Modeling of Concepts and Relations. *Proc. of 14th European Conference on Artificial Intelligence, Workshop on Applications of Ontologies and Problem-Solving Methods*, 2000.
8. Gamma, E., Helm, R., Johnson R. and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
9. Noy, N.F., Sintek, M., Decker, S., Crubézy, M., Fergerson, R.W. and Musen, M.A. Creating Semantic Web Contents with Protégé-2000. *IEEE Intelligent Systems.* March/April 2001
10. Bechhofer, S., Horrocks, I., Goble C. and Stevens, R. OilEd: a Reason-able Ontology Editor for the Semantic Web. *Working Notes of the 14th Int. Workshop on Description Logics,* USA, 2001.
11. Mahalingam K. and Huhns, M.N. A Tool for Organizing Web Information. *IEEE Computer.* June 1997, Pp. 80-83.