Requirements Eng (2005) 10: 146–160
DOI 10.1007/s00766-004-0208-2

ORIGINAL ARTICLE

Joerg Evermann · Yair Wand

Ontology based object-oriented domain modelling: fundamental concepts

Received: 4 May 2004 / Accepted: 11 November 2004 / Published online: 13 January 2005

Abstract Understanding the business is an important step in information system (IS) development. Conceptual models are descriptions of the organizational context for which a system is developed, and are used to help understanding this context. However, conceptual modelling methods do not provide well-formalized ways to create domain descriptions. On the other hand, in the area of IS design and software modelling, languages exist (such as UML) that possess a high level of formality. Extending the use of these IS design languages to conceptual modelling, even though they have not been specifically intended for this, can lead to several advantages. In particular, it can enable the use of similar notation in several stages of system development. However, while object-oriented constructs such as "object" and "operation" have clear meaning in the context of software design, it is not clear what they might mean in terms of the application domain, and no rules or guidelines exist for using them to create useful descriptions of such domains. This paper suggests specific semantics for object-oriented constructs based on a mapping between ontologically derived concepts and object-oriented language constructs. The paper also proposes modelling rules to guide the construction of object-oriented conceptual models and to assure that such models describe only ontologically feasible application domain situations. While the results are applicable to object-oriented constructs in general, UML is used as an example. A case study to test the use of the proposed semantics and modelling rules is described.

Keywords Object-oriented modelling · System analysis · Business analysis · UML · Ontology

J. Evermann (✉)
School of Information Management,
Victoria University of Wellington, Box 600,
Wellington, New Zealand
E-mail: Joerg.evermann@vuw.ac.nz
Tel.: +64-4-4636857

Y. Wand
The Sauder School of Business,
The University of British Columbia,
2053 Main Mall, Vancouver, BC, Canada
E-mail: yair.wand@ubc.ca

Y. Wand
MIS Department, School of Social Sciences, Haifa University,
Haifa, 31905, Israel

1 Introduction

Information System (IS) projects begin by examining and understanding the business and organizational domain in which the information system is to be embedded. The system analysis phase of information systems development is concerned with representing this business domain (often called the "real world domain"). Such a description is termed a *conceptual model*: "Conceptual modeling is the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication" [1].

The description of the software system that is developed in the subsequent design phase is increasingly shaped by technical considerations. For describing the software system, the use of object-oriented techniques is well accepted, and a number of software modeling methods and languages have been proposed, beginning with the Booch method [2], Fusion [3], OOSE [4], and culminating with the Unified Modelling Language (UML) [5]. As these methods have been developed with the specific intention of designing and describing software, in this paper we term them object-oriented IS design languages. The meaning of constructs used in these design languages is well defined with respect to software concepts. In contrast, conceptual modelling languages are intended specifically for describing the application domain. The meaning of constructs used in these languages should be defined in terms of application domain concepts.

While the difference between analysis and design models, or between business and software models, is well recognized, the lack of rich, formal languages specific to conceptual modelling, combined with the availability of widely used IS design languages, has a number of detrimental effects: (1) IS development projects might begin without explicitly modelling the application domain and instead must rely on implicit assumptions of developers. (2) Even when the real world is explicated, the use of IS design languages for this task without specific guidance can lead developers to confuse aspects of the IS and the application domain. For example, an analyst may consider a job to be an object, with the already implicit understanding that jobs will be represented by a specific class in the object-oriented software system. However, in the business domain, a job is not necessarily viewed as an "object" (this is explained below). (3) The mapping from conceptual to design models is not explicated. This lack of explicit translation can again lead to hidden assumptions held by different stakeholders and developers.

While object-oriented modelling languages such as UML have been developed for modeling software systems, not application domains, there are several potential advantages for using them for modeling the latter. (1) Object-oriented (OO) IS design languages can provide a much-needed well-known, rich, formal or semi-formal notation for conceptual modelling. (2) A common language used in both systems analysis and system design can ease the transition between these two stages. Specifically, analysts can use models not only for communicating amongst themselves or with stakeholders, but also to communicate with software designers to convey information about software requirements. Using a single object-oriented modelling language for both analysis (conceptual modelling) and (software) design can serve to eliminate translation errors that might occur when two different languages are used. In other words, using one modelling language for representing both the application domain and the information systems can help alleviate the so-called impedance mismatch between two disparate languages [6–9].

The main problem to overcome in using object-oriented IS design languages for conceptual modelling is the lack of meaning of language constructs such as 'object', 'class', 'attribute', and 'operation' when used to model application domains. It is clear to the software designer or programmer what these mean in terms of the programming statements and code that ultimately results. Thus, these constructs possess implementation related semantics. However, it is much less clear to the business analyst what any particular object-oriented language construct might mean in terms of the application domain being analyzed, as these constructs possess no real-world semantics. In order to extend an object-oriented IS design language to describing business and organizational domains, we must attach application domain semantics to their constructs.

This paper proposes ontologically based semantics for object-oriented constructs to enable the use of object-oriented IS design languages for conceptual modelling. The proposed semantics are used to derive modelling rules and guidelines on *how* to use object-oriented languages for business domain modelling.

We use the UML [5] as an example for object-oriented modelling languages for the following reasons:

- It is the most prominent and widely accepted IS design language.
- It is an evolving language. Hence, research can have practical influence on the evolution of the language standard.
- It has been developed for modelling software and software components. Yet, it is not specifically limited to this use. Thus, it appears possible to extend the application area of UML to include business modelling.

The remainder of this paper proceeds as follows. Section 2 describes the methodology for assigning business meaning to object-oriented concepts and deriving modelling rules. The ontological foundations are introduced in Sect. 3. As object-oriented languages are comprised of large numbers of constructs, a complete analysis is beyond the scope of a single paper. This paper therefore focuses on the fundamental concepts; aspects of behaviour and interaction are covered in a different paper. To assign ontological semantics, ontology is first discussed in Sect. 4 from the perspective of UML and then UML is discussed from the perspective of ontology in Sect. 5. Section 6 describes empirical evidence to support the results. Finally, Sect. 7 points out possible extensions of this research.

2 Business meaning for object-oriented languages

In order to assign application domain semantics to a language, we must specify what exists, or is perceived to exist, in the domain. What exists in the world is the subject of *ontology* which is "that branch of philosophy which deals with the order and structure of reality in the broadest sense possible" [10]. A specific ontology makes assumptions about what exists and how things behave.

Assigning ontological semantics to a language amounts to answering two questions [11]:

1. How can an element of the real-world business and organizational domain (ontological concept) be represented in the chosen language? To answer this question, we propose a *representation mapping* from the set of ontological concepts into the set of language constructs that assigns each ontological concept a language construct with which to represent it.
2. How can a construct of the language be interpreted in terms of the real-world domain (ontologically)? To answer this question, we propose an *interpretation mapping* from the set of language constructs into the set of ontological concepts that assigns each language construct an ontological interpretation.

A number of requirements exist for this mapping. First, it should be complete, i.e. it should attempt to map all ontological concepts and provide meaning to as many object-oriented constructs as possible.[1] Second, it should be clear, i.e. it should be a one-to-one mapping between the ontology and the (subset of) object-oriented constructs. Third, it should be internally consistent, i.e. the mapping should preserve and retain the relationships among ontological concepts and those among object-oriented constructs. For example, mapping ontological things to objects should be done in order to preserve the relationship that ontological things have with ontological properties and objects have with attributes. Consequently, mapping ontological things to object states and ontological properties to object operations is not sensible. In other words, the mapping of a given language construct (e.g. that of objects) has implications for the possible mapping of other, related ones (e.g. that of attributes). Hence, the analysis must be a holistic one. We approach this problem in the following way: We begin with a proposed representation mapping of a subset of ontological concepts. With this initial mapping established, we continue with an interpretation mapping of as-yet unmapped object-oriented constructs. In this way, we can preserve existing (semantic) relationships among ontological elements, and the relationships that exist among object-oriented constructs.

The proposed ontological semantics (i.e. the mapping of ontology to language) are expressed in *semantic mapping rules* that relate elements of the modelled domain to elements of the language. These rules can be used to guide the modeller in representing aspects of the application domain using particular language constructs.

Once the mapping is formed, it can be used to transfer ontological assumptions to the language. We require that constraints that relate ontological concepts must also hold between the respective language constructs. This leads to *language rules* that govern the combined use of language elements. These rules constrain the models that can be constructed with the semantically enhanced object-oriented constructs, and serve to assure that the constructed models "make sense" ontologically.

The example language used in the paper, UML, includes several types of diagrams that can provide the modeller with different perspectives on the modelled domain. Accordingly, for UML, the language rules may lead to *intra or inter-diagram rules* depending on whether they relate language elements used within a single diagram or in different diagrams.

We note that it is not the purpose of this work to develop modelling rules for system design, but rather to extend the use of object-oriented design languages into conceptual modelling for information system analysis. Hence, the derived rules might not be applicable for system design, i.e. software modelling. As well, we note that the proposed rules do not necessarily guide us in *how* to perceive the world. Thus, we suggest rules on how to model elements of the application domain, but not on how to identify them.

Since the modelling rules proposed in this paper constrain the use of existing object-oriented languages, a model constructed according to the proposed semantics and proposed rules still conforms to object-oriented principles. Hence, such models can retain their meaning in terms of software and be translatable to programming elements. We note that subsequent transformations may be needed in order to create software, e.g. in order to increase the computational efficiency of the software, to adapt to certain database technologies or software frameworks. Such implementation considerations are beyond the scope of this paper.

3 Ontological foundations

This research uses the term 'ontology' in its original philosophical sense, understood as meta-physics or the philosophy of existence [10]. An ontology is a fundamental philosophical position akin to a set of beliefs about the existence of certain entities in external reality. As such, it cannot be justified or debated a priori. A specific ontology makes specific assumptions about what is perceived to exist in reality and therefore it is not a language, though it has to be expressed using a language.

This philosophical understanding of ontology contrasts with that applied in artificial intelligence (AI), knowledge engineering (KE) and computer science research, e.g. [12–14]. In these areas, ontologies are understood as dictionaries, taxonomies, categorization schemata or modelling languages, not as descriptions of philosophical positions about reality. Thus, they do not imply a commitment to belief about reality, but rather formalisms to describe such a commitment. Such ontologies can be referred to as "IS ontologies". We note there have been calls in the IS ontology literature for returning to the philosophical meaning of ontology.[2]

3.1 Bunge's ontology

The specific ontology chosen for our purposes is based on Bunge's work [15, 16]. As mentioned above, the

[1]We provide meaning to a subset of object-oriented concepts; some concepts have only implementation meaning and do not reflect any phenomenon in the modelled domain.

[2]A return to philosophical ontology has been argued for e.g. by [10]: "The computer science use of the term 'ontology'... is taken as nearly synonymous with knowledge engineering in AI, conceptual modeling in databases, and domain modeling in OO design. We believe it is important... to maintain that 'ontology' is not simply a new word for something computer scientists have been doing for 20–30 years; ontology is 100s, if not 1000s, of years old, and there are many lessons learned in those centuries that we may borrow from philosophy along with the terms".

choice of a specific ontology is not derived from theoretical considerations, but rather reflects a set of beliefs. However, a number of pragmatic reasons lead us to choose Bunge's ontology:

- It is rooted in ontological work done over a long period in the past: "Our work is in line with an old and noble if maligned tradition: that of pre-Socratic philosophers, Aristotle, Thomas Aquinas, Descartes,..., Peirce, Russell, and Whitehead" [15].
- It is well formalized as an axiomatic system, using a set theory representation.
- It has not been developed specifically for use in information systems analysis and design, but is instead based on "the ontological presuppositions of contemporary scientific research, topped with new hypotheses compatible with the science of the day" [15].
- It has been adapted to information systems modelling and shown to provide a good benchmark for the evaluation of modelling languages and methods [17–29].
- It has previously been used to suggest an ontological meaning to object concepts [26, 30].
- It has been empirically shown to lead to useful predictions [31–35].

The following introduces the ontological concepts of Bunge's work [15, 16] most relevant to our purpose, as adapted for information system modeling [11, 36]. A brief synopsis is presented in Table 1.

The world is made up of substantial *things* that exist physically in the world. Therefore, concepts such as "skills" and "jobs" are not things.

A thing possesses (substantial) *properties*. Properties in general are those possessed by a set of things, e.g. "colour", "speed", "salary", etc. An individual property is representable as the value of a property in general, such as "blue in colour", "speed of 100 mph" or "salary of $2000".

Table 1 Concepts of the BWW-ontology

Ontological concept	Explanation
Thing	Fundamental concept, the world consists of things and only of things
Property	Things have properties
Intrinsic property	Property of one thing
Mutual property	Property of two or more things
Law	Restriction on or relation of properties
Composition	Things can be composed to form composite things
Emergent property	Property of a composite thing not possessed by its parts
State function	Function describing a property of a thing
Functional schema (Model)	Set of state functions describing things
State	Value vector assigned to state functions of a schema
Natural kind	Set of things adhering to a set of laws (common behaviour)

Properties can be either intrinsic or mutual. *Intrinsic properties* are those that a thing possesses by itself, e.g. "colour", whereas *mutual properties* exist between two or more things, e.g. "employed by". A property can be described in terms of a set of values (possibly from different domains). For example, the salary of an employee of a company is a mutual property comprising base pay, overtime pay and Sunday pay. Hence, its co-domain is the set of three-element sets (in this case a sub-set of the power sets of monetary values).

Things can combine to form a *composite* thing. Composite things can be decomposed into parts that are in turn things and there exist simple things that cannot be decomposed further. Composite things must possess properties in addition to those of their components. Such properties are termed emergent properties.

Things are not created or destroyed; they merely change properties, are combined to form composites, or broken down to their components.

No two things have exactly the same set of individual properties. Thus, properties can be used to identify things.[3]

Properties in general are represented by *state functions*, the values of which express individual properties. A set of state functions comprises a *functional schema* or a *model*. A thing may be described by different models. A functional schema is usually used to model similar things.

A *law* is any restriction on the individual properties, and hence on the values of the state functions of a thing. The set of laws that a thing adheres to determines its behaviour.

A set of things having the same property is called a *class*. A set of things having several properties in common is called a *kind*. A set of things adhering to the same laws is called a *natural kind*. Since laws relate properties, a natural kind implies a set of properties as well.

It is important to note that the thing is the primary concept, not classes, kinds, or natural kinds. Consequently, there can be no class, kind or natural kind without members. As laws determine possible states, a natural kind is the set of things that exhibit like behaviour.

Change may be quantitative, in which case one or more individual properties are changed, or it may be qualitative, in which case properties in general are acquired or lost. The acquisition or loss of behaviour is generally concurrent with loss or acquisition of properties in general. Rather than assigning things a new name on every change of a property, Bunge advocates keeping the name of a thing until it changes its natural kind (principle of nominal invariance): "A thing, if named, shall keep its name throughout its history as long as the latter does not include changes in natural kind—changes which call for changes in name." [15, p. 221]

[3]In an information system, we may not know or care about all properties and instead use artificial identification attributes to represent the set of identifying properties.

4 Representation mapping: assigning ontological meaning to object constructs

We begin the assignment of ontological semantics with a representation mapping of an initial set of fundamental ontological concepts. This forms the basis for the interpretation mapping of further object-oriented constructs in the next section. First we discuss the most basic ontological concept, that of a *thing*.

4.1 Things

A primary concern of every modelling endeavour is the identification of the basic structure of the domain. The modeller needs to decide what to model as an object and what not to model as an object. Questions such as "is an employee an object?" or "are skills or transactions objects?" are of fundamental importance. Because every Bunge-thing is a physical (substantial) thing in the world, we propose that a Bunge-thing is equivalent to an object. The inverse is not necessarily true: Not every object in software design is a physical (substantial) thing equivalent to a Bunge-thing. "Locations", "jobs", "orders" [4] are not substantial things in the ontological sense yet are still often modelled as objects. If we want to assign the ontological semantics of a thing to an object, we must follow *semantic mapping rule* 1:

Rule 1 Only physical (substantial) things in the world are modelled as objects.

Figure 1 is an example UML class diagram to show the implications of our rule. The model is taken from [37] and described as an analysis level model. It depicts a situation typically found in object-oriented models. According to rule 1, "Order" and "OrderLine" should not be modelled as classes, as their instances (i.e. objects) have no physical (substantial) counterpart thing in the real world. Hence, we must find an alternative description for concepts such as "Order", etc.

4.2 Properties

Having identified what to model as an object, and what not to, we find that many of the items that we rejected as objects are properties of things. For example, while an employee should be modelled as an object, skills of employees should not, nor should addresses of employees, as these are not substantial things. Instead, we suggest representing these properties by object-oriented attributes.

In ontology, every thing possesses properties, which may either be intrinsic, possessed by the thing alone, or mutual, joint properties of two or more different things.

[4]We are not interested in the physical manifestations, such as an order form. These are merely vehicles to carry the description of an order.

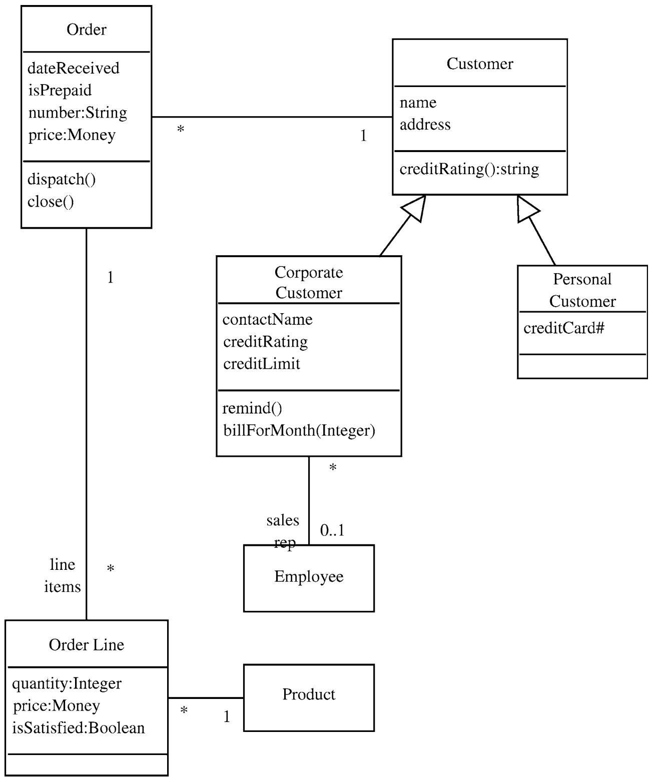

Fig. 1 Example UML class diagram without ontological semantics (Fowler and Kendall [37])

We propose that all properties be represented by attributes of objects. Thus, when using object-oriented languages for conceptual modelling, we suggest following *semantic mapping rule* 2:

Rule 2 Ontological properties of things must be modelled as object-oriented attributes.

One implication of this rule is a clear distinction between attributes and objects that reflects that they represent two different ontological concepts—properties and substantial things. For example, while a skill is a property of a person, it is not part of a person. Rules 1 and 2 together lead us to propose the following *semantic mapping rule*:

Corollary 1 Attributes in an object-oriented description of the real world cannot represent substantial entities.

Consider for example a person having a language skill. Because a skill is not a substantial thing, it must not be modeled as an object but must be an attribute of the person. Note that in software design, a skill is often modelled as an object. This is an example of the differences between a conceptual model and a software model.

4.2.1 Mutual properties

For reasons of clarity of mapping, we propose that all properties, intrinsic and mutual ones, are represented by attributes. In object-oriented languages, attributes of association class instances (UML terms these "link objects") are attributes that belong to a 'connection' of two or more objects. Thus, we propose that intrinsic properties be represented by attributes of 'ordinary' object classes, while mutual properties be represented by attributes of association classes. Specifically, we propose that bundles of mutual properties be represented by an association class, each property being represented by an attribute of that association class. For example, the attribute "Order volume" represents a mutual property between a supplier and a customer and the attribute "Salary" represents a mutual property between an employer and an employee. This proposed mapping is expressed by the following *semantic mapping rule*:

Rule 3 Sets of mutual properties are represented by attributes of association class instances (Rule 5 below further clarifies the notion of sets of properties in this rule).

For example, consider the "Job" modelled in Fig. 6 (below). A "Job" is not a substantial thing, but rather a 'connection' between two things, the employee, and the company. We therefore model it as an instance of an association class. The attributes of the association class "Job" represent mutual properties that exist when an employee is assigned a job.

With this representation of a bundle of mutual properties, association class instances cannot represent substantial things, because in that case the attributes would represent intrinsic properties of these things, not mutual ones.

Corollary 2 Association class instances do not represent substantial entities.

Because of this rule and corollary, substantial entities should not be represented by association class instances. For example (Fig. 2), the controller device with which a computer controls a robot, is a substantial thing. Hence, the representation of it as an association class instance is wrong. It should instead be modelled as an object participating in the association (Fig. 3). Note that some

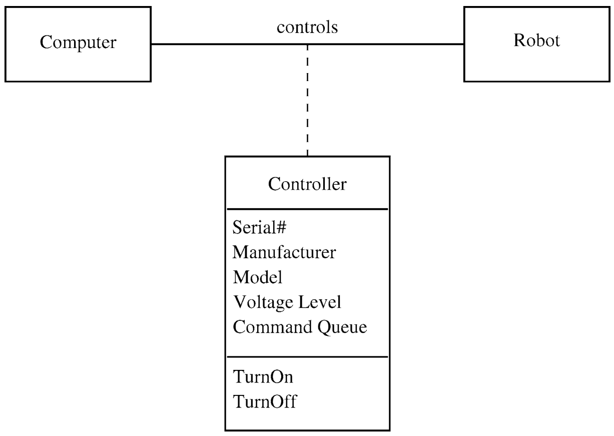

Fig. 2 A substantial association class

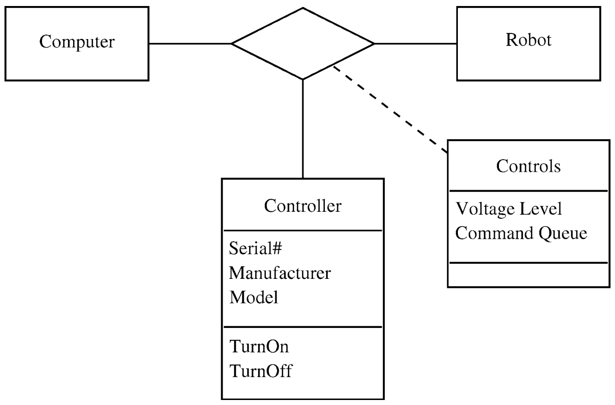

Fig. 3 A reinterpreted substantial association class

attributes represent mutual properties of all three things, e.g. the voltage level or the command queue. These remain association class attributes. In general, we propose the following rule as a corollary of Rule 3:

Corollary 3 If instances of an association class of an n-ary association are intended to represent substantial things, the association should be replaced by a regular class and an association class with arity $(n+1)$.

As a further consequence of Rule 3 and Corollary 2, aggregates or composites should not be modelled as instances of association classes, as they do not represent bundles of mutual properties but substantial things. For example (Fig. 4), the processor and memory are associated to perform computations. The association has an attribute "computing power". This violates rule 3 and corollary 2. Instead, the computer that possesses the computing power property should be modelled explicitly as the aggregate of memory and processor (Fig. 5) having processing power. In general, we propose the following rule as a further corollary of rule 3:

Corollary 4 If association class instances represent composite things, the association class must be replaced by an aggregate class where the parts are the associated classes and the original attributes now represent emergent properties.

More on composites and aggregates below.

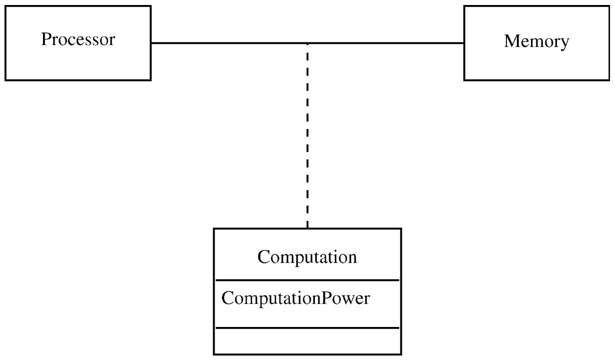

Fig. 4 An association represents a composite

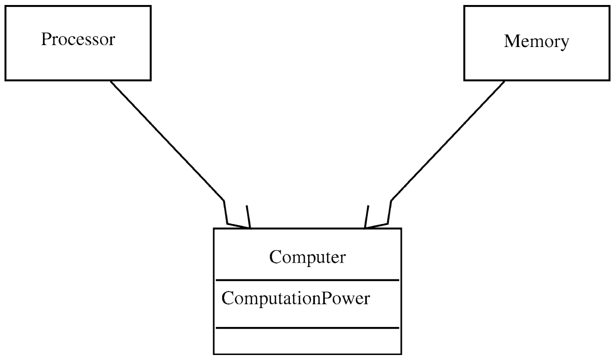

Fig. 5 Reinterpreting an association classs as a composite

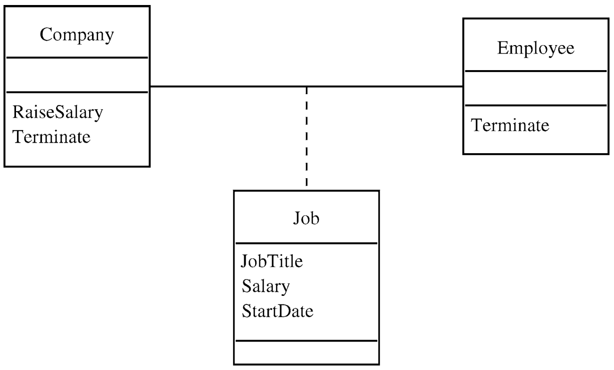

Fig. 7 Asssociation class and operations

4.3 Characteristics of association classes

Ontologically, all change is tied to substantial things. There can be no change without a thing that changes. Since association classes and their instances do not represent substantial things, they cannot possess methods or operations. This leads to the following *language rule*:

Corollary 5 An association class cannot possess methods or operations.

This corollary proscribes assigning operations or methods to association classes. How then do mutual properties change? Given that properties represented by association classes are mutual to the participants of the association, they are changed when the things represented by the participant classes undergo changes. Hence, we propose that methods and operations intended for association classes should be modelled with the participant classes. In the example of Fig. 6, the operation "RaiseSalary" should be modelled with the company, and the operation "Terminate" should be modelled with either the company or the employee or both, depending on the real-world situation. Figure 7 shows this interpretation, which is formalized in *language rule* 4:

Rule 4 Methods and operations that change the values of attributes of an association class must be modelled for one or more of the classes participating in the association, not for the association class.

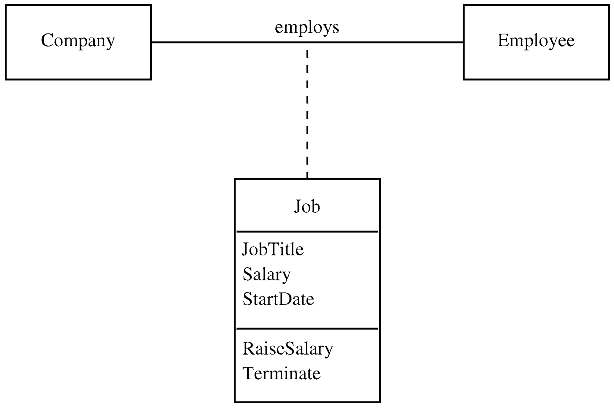

Fig. 6 A conceptual association class in UML

There can be no change without a thing that undergoes that change. Association classes do not represent changeable things and thus cannot be associated with state machines:

Corollary 6 An association class cannot be associated with a state machine.

An association class instance represents a set of mutual properties. An association class instance without attributes would represent an empty set of mutual properties, which is ontologically meaningless. Hence:

Corollary 7 An association class instance must possess at least one attribute.

In object-oriented models, associations can appear without attributes. Corollary 7 proscribes this. Furthermore, as properties in ontology cannot themselves possess mutual properties with other properties or things, an association class instance must not be associated with any other object or association class instance.

Corollary 8 An association class must not be associated with another class.

Since association classes do not possess instances that are interpreted as Bunge-things, they do not model ontological kinds, and cannot be generalized or specialized.[5]

Corollary 9 An association class must not participate in generalization relationships.

Mutual properties usually occur together. For example, some mutual properties arise out of interaction, e.g. after a customer and a car rental company interact, there exist a number of mutual properties such as 'DateOut', 'DateDue', 'BalanceDue', and 'DepositPaid'. We suggest that a set of mutual properties that arises out of one interaction be modelled in a single

[5]Note that the notion of property precedence in Bunge's ontology describes laws, not a generalization of properties.

association class. Each association class expresses related concurrent properties; different association classes should be used if properties are not necessarily concurrent:

Rule 5 An association class represents a set of mutual properties arising out of the same interaction.

Therefore, two classes may be linked by more than one association. For example, consider the interaction of a customer with a company—ordering an item—that gives rise to mutual properties such as 'Payment Balance', 'Expected Delivery Date', etc. between the customer and the company. Consider another interaction when the customer returns a defective product for warranty. Mutual properties such as 'Return Merchandise Number', 'Fault Description' etc. arise. These two sets of properties should be modelled as two separate association classes between the company and the customer.

4.4 Composition and aggregation

Often, the modeller is confronted with the fact that different things are combined in some way to yield other things. However, not every combination of things is ontologically meaningful.

Object-oriented languages such as UML distinguish between composition and aggregation, which differ along two semantic dimensions: In a composition relationship, the parts of the composite are existentially dependent on the composite and also cannot be part of any other composite. In an aggregation, the parts can exist independently of the aggregate and can also, at the same time, be parts of multiple aggregates. The object-oriented aggregation semantics are equivalent to the ontological notion of composition, whereas the more restrictive object-oriented composition has no ontological counterpart. The following *language rule* expresses this:

Rule 6 A composition relation must not be modelled.

In the Bunge-ontology, a composite possesses at least one emergent property; otherwise, there exists not a composite but only a set of things. In other words, a composite thing must be more than simply the 'sum' of its parts, as expressed in the following *language rule*:

Rule 7 Every aggregate object must possess at least one attribute that is not an attribute of its parts, or participate in an association class in which none of its parts participates.

Consider again the example depicted in Fig. 4 of the processor and the memory. Only the fact that together the processor and memory give rise to an emergent property makes the composition meaningful. Otherwise, the processor and the memory would simply be two things that happened to be located close together.

Participation of the aggregate in an association class indicates that it possesses an emergent mutual property. For example, a soccer-team, the aggregate of 11 or more soccer players, can possess mutual properties such as game results or league standing with other soccer teams. These are emergent mutual properties that cannot be possessed by individual soccer players, the parts of the team.

Our rules ensure that only meaningful compositions of things are modelled as object-oriented aggregates.

5 Interpretation mapping: analyzing object constructs

The previous section examined the basic ontological concepts of things, properties and the composition of things. This section will examine object-oriented concepts and constructs to which we have not yet assigned ontological meaning. We explore consequences of the previously made representation mapping and assign ontological interpretation to additional object constructs.

5.1 Class

Object-oriented languages define a class as a *description* of a set of objects that share the same attributes and operations. Objects cannot exist without classes; they are created from class templates. In contrast, in Bunge's ontology things exist independent of any classes. A Bunge-class is the set of things that possess a common property, a Bunge-kind is the set of things that possess more than one common property and a Bunge-natural kind is a set of things that adhere to the same set of laws and consequently exhibit the same type of behaviour.

All of these are *sets of things*, not descriptions or templates. Thus, object-classes and Bunge-natural kinds are fundamentally different, and hence are not mapped onto each other.

We propose that an object-class is interpreted as a functional schema. A functional schema is a model of things that have similar properties and laws. In effect, things modelled by the same functional schema can be viewed as being instances of Bunge's natural kind. Things that are instances of the same natural kind adhere to the same laws; hence, they possess a set of common properties and have common behaviour. Thus, a class is defined by a set of common attributes and common methods representing the characteristics of a natural kind.

Finally, we note that in object-oriented programming and software design, classes can possess methods such as "Create" and "Destroy". We do not consider these methods when modelling the world, as things are not created, but rather assembled from other things. More on this below. Furthermore, since classes are not interpreted as Bunge-things, they cannot take action, change the properties of things, compose, or decompose them. Thus, a class cannot "create" its own objects.

5.2 Object identity

Object identity, while not explicitly modelled in UML, is an important aspect in object-oriented approaches. In ontology, things are identified through their unique set of individual properties and there exists no special identifier. An object ID as a special attribute has no ontological equivalent in the real world and is thus excessive, reflected by the following *language rules*:

Rule 8 Object ID's must not be modelled as attributes.

Rule 9 The set of an object's attribute values must uniquely identify the object.

Hence, the modeller must determine a combination of attributes that uniquely identifies an object. In the context of software design, it may not be desirable to include all identifying attributes. In this case, they may be summarized or aggregated in the form of an artificial object identifier. This identifier thus represents all those properties of a thing that ensure its uniqueness. However, in the conceptual model of the business or the organization, the identifying attributes should be modelled. At first glance, this might not always seem possible. For example, one would not usually wish to include in a model of the organization all the properties needed to distinguish between two employees. However, in everyday life people are distinguishing among people, and usually reflect the distinction by naming people, Hence, in the real world, identifiers such as a person's name do in fact reflect properties in the world, albeit ones that depend on e.g. the social context.[6] They are not an artefact of the modelling process.

5.3 Multiplicities

Object-oriented languages allow the modeller to assign multiplicities to attributes and association classes. Because multiplicities express different semantics in each case, they are dealt with separately, beginning with multiplicities of attributes.

5.3.1 Attributes

In Bunge's ontology, every thing that possesses a property in general possesses a particular individual property: Every thing that possesses the property "colour" is either red, blue, green, etc. Hence:

Rule 10 Every attribute has a value.

Special values such as 'NULL', 'NIL' or 'void' are sometimes used in object-oriented models to indicate the lack of a value. This contradicts Rule 10. Moreover, these special values are not elements of the co-domain of any real world property. Hence, we proscribe their use as an attribute value in any model.[7]

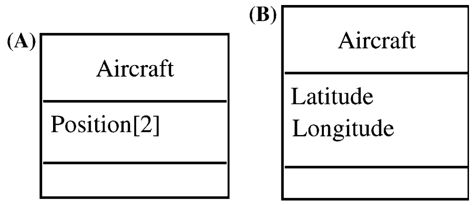

Fig. 8 Mutliplicity of attributes

State functions representing properties may be multi-valued. Hence, object attributes can be multi-valued. When multiplicity is attached to an attribute, it implies that all values are taken from the same domain (i.e., the attribute attains a value out of a power set of a certain domain). However, the semantics of a set as an unordered collection means that the modeller must take care when modelling multi-valued attributes and identify their meaning. Consider the example depicted in Fig. 8. While the "Position" in (A) is modelled with a multiplicity of two, this does not represent the true real-world situation, as the two values are not freely interchangeable. One represents longitude, the other latitude. A set representation cannot accommodate these semantics. Instead, they should be modelled as shown in (B). In general, we propose that values taken from the same domain are combinable into a multi-valued attribute only when their order is semantically irrelevant:

Corollary 10 Attribute multiplicities greater than one imply that the order of the different individual attribute value components is semantically irrelevant.

5.3.2 Association classes

In contrast to attribute multiplicities, multiplicities assigned to association classes express the number of other objects with which an object is associated. Thus, they represent the number of other things with which a particular thing shares a particular set of mutual properties. This is distinctly different from the meaning of attribute multiplicities and the above arguments are not applicable. Instead, we propose that multiplicities for association classes serve to support the semantics of acquisition or loss of properties.

Consider the example in Fig. 9. Wand et al. [28] argue that a distinction should be made whether a thing possesses mutual properties or not. Using an example of a book and a student which may borrow it, they argue that instead of associating a student with zero or more books that she borrowed, the stu-

[6]The name of the person can be considered a mutual property of that person and the remainder of the society. However, as this remainder is usually implicit, the name can be considered an intrinsic property of the person. The changing of a property from mutual to intrinsic is termed unarization [11].

[7]We also note that in Bunge's ontology not having a certain a property is not considered a property; hence, a 'null' property has no ontological meaning.

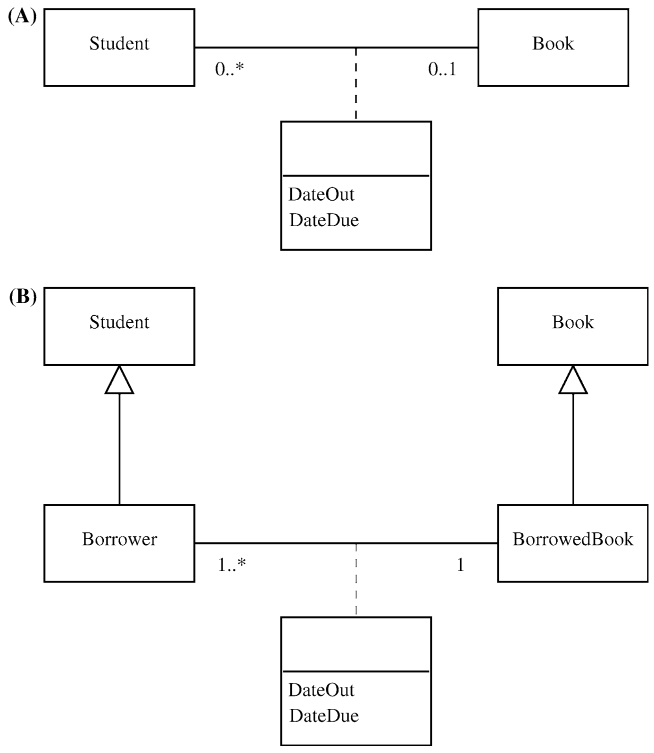

Fig. 9 Optional properties and re-classification

dent should be specialized into a 'Borrower' which borrows *one* or more books, while the more general 'Student' has not borrowed any books (Fig. 9). Their argument for this is that the model in Fig. 9a shows mutual properties ("DateOut", "DateDue") for the student, even if none are existent when the student has not borrowed any books. Hence, the model in Fig. 9a "folds" together two different functional schemata. We argue that a similar distinction must then be made between a borrower that has borrowed one book and a borrower that has borrowed two books, as the borrower acquires additional mutual properties.

However, the acquisition of the first set of mutual properties (with the first book in the example) also defines new behaviour for the thing acquiring this behaviour (returning the book, extending the loan, etc.). Moreover, this additional behaviour is identical for each subsequently acquired set of mutual properties (with the second, third, etc. book). Recall that natural kinds in ontology are defined by things that possess the same laws and consequently exhibit the same behaviour. Thus, the student and the borrower form two different natural kinds. As they form different natural kinds, they are described by different functional schemata. Hence, they must be represented by different classes.

We propose that the acquisition of additional behaviour is a criterion for distinguishing classes of objects. Often, such additional behaviour is manifested by additional attributes or associations. This implies that in certain domains it may be necessary to make further distinctions than the "zero" versus "one or more" distinction usually made. Thus, we propose the following *language rule*:

Rule 11 Classes of objects that exhibit additional behaviour, additional attributes or additional association classes with respect to existing classes, must be modelled as specialized sub-classes.

5.4 Object creation and destruction

Object creation and destruction have no direct equivalent in the Bunge-ontology as things cannot be created or destroyed. Instead, we relate these notions to Bunge's principle of nominal invariance (see above).

With this principle, object creation and destruction are related to acquisition or loss of properties. For example, a set of bricks, when combined into a house, 'creates' the new thing 'house' with the emergent property of 'NumberOfBedrooms'. This acquisition of properties is a result of composition. Assume that altering the way the bricks are combined makes the house into an office building. It undergoes a qualitative change, losing the property 'NumberOfBedrooms' and acquiring the property 'NumberOfOffices'. If we now assume that the house but not the bricks are part of our description of the world, then it would appear to us that a house had been created. Similarly, if we assume that 'house' but not 'office building' is part of our description, it would appear as if a 'house' had been destroyed. These examples motivate the following *semantic mapping rules*:

Rule 12 Object creation occurs when an object acquires a property (by composition with other objects or by other means) so that it becomes a member of a different object-class.

Corollary 11 Object destruction occurs when an object loses a property that is necessary for membership in a particular class.

We use this interpretation of object construction and destruction to complement Rule 11 and suggest a mechanism for re-classification that employs object creation and destruction:

Corollary 12 An object acquiring or losing additional behaviour, attributes or associations, will be destroyed as an instance of one class and created as an instance of another class.

In particular, if an object only acquires new characteristics then it is destroyed as a member of a super-class and created as a member of a subclass, and vice versa.[8]

[8]In ontology, an instance of a sub-class is an instance of all its super-classes. However, this is not the case in object-oriented languages; although, by virtue of inheritance and polymorphism it can behave as and substitute for an instance of this super-class.

This interpretation of object creation and destruction is for purposes of conceptual modelling, rather than software modelling. For example, in the real world, a customer is not created; rather, a person becomes a customer (but also remains a person) once she acquires the property of having bought an item. In the software system, we can however create a new customer record.

5.5 Class attributes

Object-oriented languages allow the modeller to attach attributes to classes or, technically, declare the scope of an attribute to be the class, not an instance. However, the functional schema that we have mapped to object classes is not a substantial thing and hence does not possess substantial properties that can be represented by attributes.

Instead, we propose to interpret a class attribute as the representation of a property of the composite formed of the things in the extension of the natural kind (modelled by the functional schema) that the class represents. Thus, the property is an emergent one and *language rule* 13 suggests that the composition be modelled explicitly:

Rule 13 Attributes with class scope (class attributes) should instead be modelled as attributes of an aggregate made of objects of the class.

A typical example is the property "Number of Instances", that is often ascribed to an object class. This is a property of the collection or composite of all the things that form the extension of the corresponding Bunge-natural kind. A specific example is shown in Fig. 10. Whereas UML allows us to model a number of aircraft as in situation (a), ontologically we require that the extension of the class of aircraft be made explicit by using composition as shown in situation (b).

5.6 Associations

There exist three kinds of associations in object-oriented languages. Composition and aggregation associations have been discussed above. Here we focus on 'ordinary' associations. In object-oriented languages, an association class is a subtype or specialization of both an association and a class, i.e. an association class *is* an association. Associations in the object-oriented approach are employed to enable message passing. However, since message passing is a software-related concept and considered without equivalent in the Bunge-ontology [25, 26, 38], we propose that associations which are not association classes are ontologically excessive. Hence, they should not be employed for conceptual modelling of the business and the organization and their use should be deferred until the software design phase. We express this using the following *language rule*:

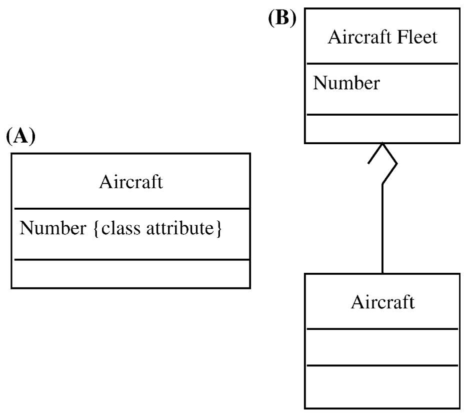

Fig. 10 Class attributes

Rule 14 Every ordinary association must be an association class.

One could argue that binary mutual properties, ones that either exist or do not, should be modelled by associations rather than association class attributes. However, such a suggestion would lead to ambiguities since the same ontological concept, mutual properties, would be mapped to two different object-oriented constructs, attributes (of association classes) and associations. In the interest of ontological clarity, this should be avoided. The attributes of association classes, not the associations themselves, should be interpreted as mutual properties (see mapping Rule 3).

For example, the mutual property of 'being employed' should not be modelled as an association between a Person and a Company. Instead, an association class should be used with attributes such as 'Job Title' and 'Salary'.

The example of a student attending a university from [28] demonstrates our interpretation. Ontologically, this is an example of interaction giving rise to mutual properties. Hence, the properties must be represented by association class attributes, such as 'CreditsOnFile', 'AcademicStanding' and 'TuitionBalanceOwing'. It may be that the interaction is outside the scope of the modelled domain or in the past, so that the re-classification and specialization of the person as a student is not modelled. If the interaction is part of the model, there must exist a class 'Person' of which 'Student' is a subclass. Only the students share mutual properties with the university and upon the enrolment interaction taking place, an instance of class 'Person' must be destroyed and re-created as instance of class 'Student' with the requisite mutual properties being represented by attributes of an instance of the association class. Furthermore, the interaction itself must be modelled, e.g. as an event or within a collaboration.

Note that Corollary 7 proscribes modelling association classes without attributes, as these are ontologically meaningless.

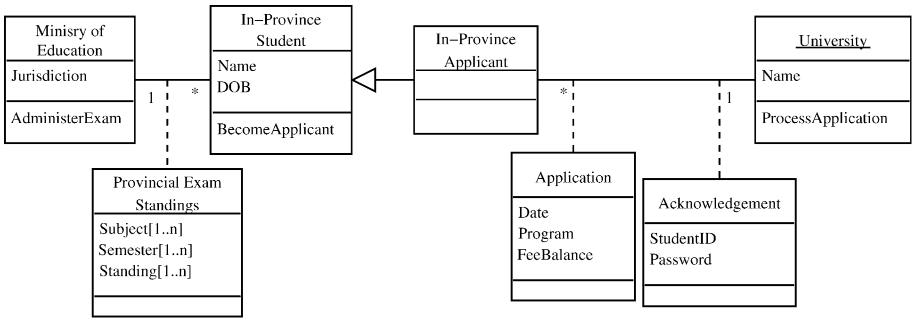

Fig. 11 Example class diagram from the case study

6 Case study

A case study was conducted to investigate the effect of the proposed semantics and modelling rules. The aim of the case study was to test the applicability of the rules in practical situations. This section highlights the main results. The organization under study is a large North American university carrying out a project for providing an online-system for prospective students to assess their likelihood of admittance. This project was chosen as the basis for the case study for two reasons. (1) The main stakeholders and informants of the analysis were available for interviews. Their involvement with the project and the analysis was recent so that rich data could still be gathered. (2) The project team had used UML in an attempt to describe the business. It was hoped that additional insight could be gained by comparing the project team's UML models with those developed independently as part of the case study.

As part of this case study, an independent analysis was undertaken in order to model the domain of student admissions and assessment. Interviews were conducted with relevant stakeholders and informants, and UML models (class and sequence diagrams) were independently constructed. All interviews were conducted on-site and employed open-ended questions. The proposed rules were used as guidelines for the analysis and the interviews, but were only disclosed to the interviewees after the independent domain model was completed and discussed with the project participants.

The process of analyzing the domain and modelling it according to the suggested ontological semantics for UML showed that all produced diagrams were valid UML models, and adhered to all regular syntactic rules. The models were not trivial and did not appear overly complex either. Hence, we conclude that use of the rules is at least possible in a practical setting. Figure 11 shows an example of an independently developed diagram. It shows an excerpt of the domain, representing high school students, who may choose to become university applicants. University and applicants are physical things. Consequently, they are modelled as object classes (Rule 1). Applications on the other hand are interactions; they do not exist as physical things and can therefore not be modelled as object classes.[9] Applicants possess mutual properties with the university. These arise out of the application interaction. Similarly, high school students possess mutual properties with the ministry of education, arising out of the interaction of taking a standardized test. Since application and test-taking are actions, not physical things, they are not modelled as object-classes. The mutual properties that arise out of them are represented by the association classes "Application" and "Provincial Exam Standings" (Rule 5).In accordance with Corollary 5, these association classes do not possess operations. Operations are modelled with the association participants (Rule 4).

The process of analysis and modelling has shown that at various stages in the process, the rules and proposed semantics led the modeller to include information in the model that was not necessarily relevant to building an information system, but that was important to properly understand the domain. The inclusion of this additional detail was a by-product, not the intention of the modelling rules.

Discussions with two project team members, the project lead (LF) and the lead developer (CH) of the project, confirmed that the independently developed model contained more information about the domain under study. Both team members commented on the fact that the original model contained many hidden and implicit assumptions:

> "We relied a lot on assumptions that were never written down in the model... yours is more comprehensive." (CH—Project Lead)
>
> "Ours have all sorts of stuff around that is assumed but not modelled." (LF—Lead Developer)

Both interviewees felt that the independently developed models would provide better understanding to a new team member or somebody unfamiliar with the domain.

> "Yours that you have are arguably better as an initial introduction." (LF)

[9]However, records of the interaction such as an application form are substantial things, but they are distinct from the interaction itself.

"It shows a better big picture view of how it fits together." (CH)

This might have been in part due to the additional information in the model, which was contextual information, i.e. information about the aspects of the domain which was not to be represented by the information system. As such, this information would be useful in enabling new members of the team to find out the rationale of specific requirements for the information system.

The independently developed model was generally felt to be complete and comprehensive, not only with respect to additional information, but also with respect to real-world aspects that were addressed in the initial model constructed by the project team:

"Certainly more comprehensive there [in the areas outside the IS scope], but even in the smaller, there's a somewhat simpler, more elegant view in a few cases." (CH)

The second important purpose for a conceptual model, besides the representation of a real world domain, is the starting point for IS design and software development. The independently developed model was not seen as defective in this way, and thus it could serve this purpose:

"I don't see any reason why you couldn't just take these [the models] and run with them." (CH)

During the discussion with the project lead, it also became clear that the use of UML without any guidelines was a big challenge in the project. Some analysts were aware of the subtle differences between possible interpretations of entities in the business domain and the difficulties associated with these differences:

"It's normally difficult to model a course object, because it is a relationship... What do you mean by a course? The curriculum, the interaction, the grade?" (LF)

"... presumably needs more thought to distinguish different kinds of objects." (LF)

"Modelling of non-substantial objects happens quite easily and naturally. You don't have to worry about empirical reality." (LF)

The last comment indicates that LF realized they might be modelling entities not corresponding to anything in the domain.

After the independently developed business model was completed and assessed by project participants, the modelling rules were disclosed to them. The project lead and lead developer agreed that modelling rules were necessary and helpful, both for guiding the modelling process and for ensuring model quality and consistency:

"Such rules would have helped in our group. The rules would tell whether a model is good and can help answer some questions. They seemed like a lot of valid questions to ask." (CH)

"Rules can force the modellers to think deeper about what they're modelling."(LF)

7 Conclusions and future research

This paper was motivated by the wish to extend the use of object-oriented languages from software design to conceptual (application domain) modelling. As object-oriented languages are not intended for this, it is necessary to attach "real world" (business) meaning to their constructs by assigning to each language construct an element of the real-world domain. The application domain is viewed as comprised of concepts specified by an ontology. This paper takes the position, supported by a series of prior applications, that Bunge's ontology is appropriate to guide object-oriented conceptual modelling. Hence, to assign business meaning to object-oriented languages, a mapping was created between the ontological concepts and object-oriented constructs. Furthermore, based on these mappings, the paper explored the consequences of transferring ontological assumptions to the language, resulting in semantic mapping rules and language use rules. These rules are applicable to object-oriented languages when they are used for conceptual modelling. The rules may not be applicable to, nor are they intended for, using object-oriented languages to describe software systems.

The paper began by mapping the central ontological concepts such as things, properties and composition to object-oriented constructs. With this representation mapping in mind, object oriented concepts such as classes, association classes, object identity, attribute and association multiplicities, object creation, class attributes and generalization/specialization were examined. Table 2 shows a summary of the proposed mapping. We have aimed at maintaining maximum ontological clarity, i.e. a one-to-one mapping between object-oriented constructs and ontological concepts. Consequently, Table 2 shows both the interpretation as well as the representation mapping.

The table shows that most of the basic ontological concepts can be mapped onto an object-oriented equivalent, indicating that object-oriented languages are expressive enough to model real-world application domains.

In summary, this work shows that object-oriented languages can be used for conceptual modeling and *how* this could be done, by providing the modeller with specific, operationalized modelling rules. Hence, this research shows how to extend the use of object-oriented languages from software design into conceptual modelling. This can provide for bridging the gap between system analysis and system design. Business analysts as well as software developers can use the

Table 2 Summary ontological mappings (ontological semantics)

Ontological concept	Object construct	Remarks
Thing	Object	
Property	Attribute	
Intrinsic property	Attribute of 'ordinary' class	
Mutual property	Attribute of association class	
Emergent property	Class attribute	
Functional schema	Class	
Natural kind	Set of objects (extension of object-class)	Described by object-class
Composition	Aggregation Composition	
Re-classification		Explained by object creation and destruction.
(Explained by loss or acquisition of properties)	Object creation	Employed to express re-classification
(Explained by loss or acquisition of properties)	Object destruction	Employed to express re-classification
	Object identifier	May indicate non-explicit identifying properties
	Association	

same familiar language to communicate not only amongst themselves, but also between the two groups. We believe this has the potential to reduce detrimental effects resulting from translations between different languages and can therefore increase the effectiveness of the development process.

The validity of the results presented in this work hinges on two choices: the ontological model used, and the specific proposed mapping from ontological concepts to object constructs. We note again that the choice of ontology is akin to specifying a set of beliefs about the nature of application domain. Such choice cannot be justified on theoretical grounds alone. Furthermore, since object-oriented constructs emerged in software design, they do not have a "natural" real-world meaning in the application domain. Hence, the proposed mapping also cannot be justified based on reasoning alone. Furthermore, it is not clear at the outset that the proposed rules are useful and usable.

It follows that the only way to validate the outcomes of such work is by empirical methods. To that end, the results of the research were tested in a case study of a medium-size project. The results provided support for the feasibility and applicability of the proposed semantics and modelling rules for medium size projects. The resulting models were found to be helpful and the modelling rules useful in the construction of the model. However, case studies by their nature cannot provide sufficient validity. We are presently conducting an experimental study to test in a more controlled way the benefits that can be derived from using the suggested ontological semantics and modelling rules.

References

1. Mylopoulos J (1992) Conceptual modeling and telos. In: Locoupoulos P, Zicari R (eds) Conceptual Modeling, Databases, and Cases. Wiley, New York, NY
2. Booch G (1994) Object Oriented Analysis and Design with Applications. Benjamin/Cummings, Redwood City, CA
3. Coleman D, Arnold P, Bodoff S, Dollin C, Gilchrist H (1994) Object-Oriented Development: The Fusion Method. Prentice-Hall, Englewood Cliffs, NJ
4. Jacobson I (1992) Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Wokingham, MA
5. OMG (2001) The Unified Modelling Language Specification. Version 1.4, The Object Management Group
6. Kolp M, Giorgini P, Mylopoulos J (2002) Information systems development through social strcutures. International conference on software engineering and knowledge engineering SEKE, pp183–190
7. Rozen S, Shasha D (1989) Using a relational system on Wall Street: the good, the bad, the ugly, and the ideal. Commun ACM 32(8):988–933
8. Roe P (2003) Distributed XML Objects. Joint modular languages conference JMLC, pp 63–68
9. Cilia M, Haupt M, Mezini M, Buchmann A (2003) The convergence of AOP and active databases: Towards reactive middleware. International conference on generative programming and component engineering GPCE, pp 169–188
10. Angeles P (1981) Dictionary of Philosophy. Harper Perennial, New York
11. Wand Y, Weber R (1993) On the ontological expressiveness of information systems analysis and design grammars. J Inf Syst 3:217–237
12. Noy NF, Hafner CD (1997) The state of the art in ontology design: a survey and comparative review. AI Mag 18(3):53–74
13. Uschold M, Gruninger M (1996) Ontologies: principles, methods, and applications. Knowl Eng Rev 11:2
14. Guarino N, Welty C (2002) Evaluating ontological decisions with OntoClean. Commun ACM 45(2):61–65
15. Bunge MA (1977) Ontology I: the furniture of the world, Vol. 3. D. Reidel Publishing Company, Dordrecht, Holland
16. Bunge MA (1979) Ontology II: a World of Systems, Vol. 4. D. Reidel Publishing Company, Dordrecht, Holland
17. Dussart A, Aubert BA, Patry M (2002) An evaluation of inter-organizational workflow modeling formalisms, Working Paper, Ecole des Haute Etude Commerciales Montreal, QC
18. Fettke P, Loos P (2003) Ontological evaluation of reference models using the Bunge-Wand-Weber model. Americas conference on information systems AMCIS, Tampa, FL
19. Green P, Rosemann M (2000) Ontological analysis of integrated process modelling. Inf Syst 25:2
20. Opdahl A, Sindre G (1993) Concepts for real-world modelling, Conference on Advanced Information Systems Engineering CAiSE, pp 309–327

21. Opdahl A, Henderson-Sellers B (1999) Evaluating and improving OO modelling languages using the BWW-model. Information Systems Foundation Workshop
22. Opdahl A, Henderson-Sellers B, and Barbier F (1999) An Ontological evaluation of the OML metamodel. In: Falkenberg E, Lyytinen K (eds) Information system concepts: an integrated discipline emerging. IFIP/Kluwer
23. Opdahl A, Henderson-Sellers B (2001) Grounding the OML meta-model in ontology. J Syst Softw 57(2):119–143
24. Opdahl A, Henderson-Sellers B (2002) Ontological evaluation of the UML using the Bunge-Wand-Weber model. Softw Syst Model 1(1):43–67
25. Parsons J, Wand Y (1997) Using objects for systems analysis. Commun ACM 40(12):104–110
26. Wand Y (1989) A proposal for a formal model of objects. In: Kim W, Lochovsky F (eds) Object-oriented concepts, languages, applications and databases. ACM Press/Addison-Wesley, pp 537–559
27. Wand Y, Weber R (1989) An ontological evaluation of systems analysis and design methods. In: Falkenberg E, Lindgreen P (eds) Information system concepts: an in-depth analysis. Elsevier science publishers, BV
28. Wand Y, Storey V, Weber R (1999) An ontological analysis of the relationship construct in conceptual modeling. ACM Trans Database Syst 24(4):494–528
29. Soffer P, Golany B, Dori D, Wand Y (2001) Modelling off-the-shelf information systems requirements: an ontological approach. Requirements Eng 6(3):183–199
30. Takagaki K, Wand Y (1991) An object-oriented information systems model. In: Proceedings of the IFIP working group 8.1 on the object oriented approach to information systems, Quebec City, pp 275–296
31. Bodart F, Weber R (1996) Optional properties versus subtyping in conceptual modelling: A theory and empirical test. International conference on information systems ICIS, p 450
32. Bodart F, Sim M, Patel A, Weber R (2001) Should optional properties be used in conceptual modelling? A theory and three empirical tests. Inf Syst Res 12:4
33. Cockroft S, Rowles S (2003) Ontological evaluation of health models: some early findings. In: 7th pacific asia conference on information systems PACIS, Adelaide, Australia
34. Gemino A (1999) Empirical comparisons of systems analysis modeling techniques, PhD Thesis, The University of British Columbia, Vancouver, BC
35. Weber R, Zhang Y (1996) An analytical evaluation of NIAM's grammar for conceptual schema diagrams. Inf Syst J 6(2):147–170
36. Wand Y, Weber R (1995) On the deep structure of Information Systems. Inf Syst J 5:203–223
37. Fowler M, Kendall S (2000) UML distilled: a brief guide to the standard object-oriented modelling language. Addison-Wesley, Reading, MA
38. Parsons J, Wand Y (1991) The object paradigm - two for the price of one? Workshop on Information Technology and Systems WITS, pp 308–319