The Association Construct in Conceptual Modelling – An Analysis Using the Bunge Ontological Model

Joerg Evermann

School of Information Management, Victoria University,
Wellington, New Zealand
Joerg.Evermann@mcs.vuw.ac.nz

Abstract. Associations are a widely used construct of object-oriented languages. However, the meaning of associations for conceptual modelling of application domains remains unclear. This paper employs ontological analysis to first examine the software semantics of the association construct, and shows that they cannot be transferred to conceptual modelling. The paper then explores associations as 'semantic connections' between objects and shows that this meaning cannot be transferred to conceptual modelling either.

As an alternative to the use of associations, the paper proposes using shared properties, a construct that is rooted directly in ontology. An example from a case study demonstrates how this is applied. The paper then shows an efficient implementation in object-oriented programming languages to maintain seamless transitions between analysis, design, and implementation.

1 Introduction

Object-oriented modelling languages are increasingly being used for describing business and organizational application domains (conceptual modelling). In order to have well-defined meaning, their constructs must be defined in terms of the elements of the application domain [1]. The use of constructs without clearly defined meaning can lead to ambiguous or confusing models.

However, the meaning of the association construct remains unclear, as the following attempts at a definition show[1]:

> An association is "the simplest form of a relationship" [2, p. 195].
>
> "An association represents the relationships between objects and classes" [3, p. 26].
>
> "Relationships associate one object with another" [4, p. 18].

[1] Note that this concerns the semantics for conceptual modelling only. Software semantics for associations are discussed in Sect. 3.

O. Pastor and J. Falcão e Cunha (Eds.): CAiSE 2005, LNCS 3520, pp. 33–47, 2005.

> "If two classes have an association between them, then instances of these classes are, or might be, linked." [2].

> "An association sets up a connection. The connection is some type of fact that we want to note in our model" [5, p. 105].

The original definition by Rumbaugh sheds little light on the issue:

> "A relation associates objects from n classes. ... A relation is an abstraction stating that objects from certain classes are associated in some way." [6, p. 466].

This lack of clarity remains even in the latest UML standard:

> "An association defines a semantic relationship between classifiers." [7, p. 2-19].

To provide a clear definition of associations in terms of the elements of the application domain, we must first determine what exists, or is assumed to exist, in the application domain. Ontologies specify what concepts exist in a domain and how they are related. Hence, to define the meaning of an association, we must map them to an ontological concept [1, 8]. This mapping must support the syntactic features of associations as much as possible. For example, associations connect two or more classes of objects. Hence, they should be mapped to an ontological concept that connects two or more sets of things or objects.

Note that this paper is concerned with association semantics from the perspective of an application domain analyst, not a software designer or programmer. In fact, association semantics are problematic also for the latter case, as shown in [9, 10]: The relationship between the association construct and programming language implementations is often unclear.

The paper proceeds as follows. Section 2 introduces the ontology that is adopted for this analysis. Next, the paper identifies the usage and the semantics of associations in software modelling (Sect. 3). It shows that software semantics cannot be transferred to conceptual modelling. Section 4 examines associations as 'semantic connections' and shows that there is no ontological concept with similar use or meaning. Hence, associations have no semantics when used for conceptual modelling.

The use of mutual shared properties is proposed as an alternative to the use of associations in conceptual modelling. We advocate the notation of association class attributes to represent mutual properties (Sect. 5). An example from a case study demonstrates this technique using a real modelling situation (Sect. 6). Finally, Sect. 7 demonstrates that the proposed technique can be efficiently and transparently implemented in object-oriented programming languages in order to maintain seamless transitions between object-oriented system analysis, software design, and implementation.

2 Ontology

This research does not relate to a particular application domain but to a language construct that is not domain-specific. Hence, an ontology on a suitable level of abstraction is required. A number of proposed high-level or upper-level ontologies exist [11, 12, 13, 14, 15, 16, 17].

Among these, the ontology proposed by Bunge [12] stands out because it has been empirically validated in a variety of applications and application domains [18, 19, 20]. Furthermore, it has repeatedly been shown to provide a good benchmark for the analysis of modelling languages and language constructs [21, 22, 23, 8, 24].

Bunge [12] proposes that the world is made up of *things* which physically exist in the world. A thing possesses *individual properties*, each of which corresponds to a *property in general*. For example, being colored red is an individual property of a particular thing; color is a property in general.

Properties are either intrinsic or mutual. *Intrinsic properties* are ones that a thing possess by itself, e.g. color, whereas *mutual properties* are shared between two or more things, e.g. voltage of a processor and memory unit, temperature of a heater and surrounding air, etc.

Two or more things can interact with each other. *Interaction* is defined by the history of a thing: If the way in which the properties of a thing change depends on the existence of another thing, then the second is said to act on the first. Ontologically, for things to interact, e.g. for a thing A to act on a thing B, there must exist a mutual property P of A and B. A change of P in A is also a change of P in B. The change of P may then cause further changes in B. For example, for a heater to warm the air in a room, the heater and surrounding air share a mutual property, temperature. When the action of the heater changes the temperature, this change leads to changes in room temperature.

3 Software Semantics of Associations

Associations, introduced to graphical object-oriented modelling in [6], originate in object-oriented programming languages. Here, they have well defined meaning: "Associations are often implemented in programming languages as pointers from one object to another" [25, p. 27], "a relation expresses associations often represented in a programming language as pointers from one object to another." [6, p. 466]. For example, the association "attends" in Fig. 1 may be implemented in the following way, making use of pointers. These pointers are then used to enable method calls.

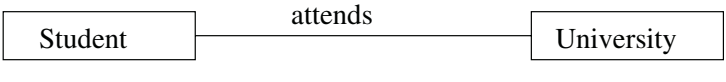

Fig. 1. Association example

```
    class University {
        void PayFees(int studentno, int semester, float amount);
             ... }
    class Student {
        AttendedUniversity *University;
             ... }
...
    AttendedUniversity -> PayFees(12345, 2, 880.00);
```

The above example shows the software semantics of associations: They provide the means for interaction by message-passing in software [2]. Booch [26] terms this a *usage relationship*, Coad & Yourdon [27] call them *message connections*. For the remainder of the paper we call them "use associations" [3].

Associations as enablers of message-passing are useful for describing software. However, the following arguments show that these software semantics are not transferable to the conceptual modelling of application domains.

First, the interaction mechanism in [12] is not based on message passing. Ontologically, for things to interact, e.g. for a thing[4] A to act on a thing B, there must exist a mutual property P of A and B. A change of P in A is also a change of P in B. The change of P may then cause further changes in B.

Second, the message passing mechanism for interaction has previously been examined with respect to Bunge's ontology and found to be an unsuitable construct or mechanism for describing real world business domains [8, 23, 29][5]. Consider the following examples:

- The machine sends a message to the part to move itself to a new location
- The general ledger sends a message to an office desk to depreciate its value
- A truck sends a message to the crate to load itself onto the loading dock

[2] Although, as pointed out in [9], UML is contradictory in allowing a dependency to express dynamic behaviour on the classifier level, while requiring a link, an association instance, to enable dynamic behaviour on the instance level. In this paper we assume that, as the instance level requires a link, the classifier level must require an association to enable dynamic behaviour.

[3] We note that there exists a $<< use >>$ stereotype of a dependency in UML. However, following the argument by [9] in the previous note, we subsume this notion under our "use association". [9] further differentiates between the static pointer aspect and the dynamic message-passing aspect. In her terms, we subsume both aspects under "use association".

[4] When the term "thing" is used, the discussion relates to the ontology in [12]. When the term "object" is used, the discussion relates to object-oriented languages. Things in the application domain are represented by objects in an object-oriented description [28, 22].

[5] This also agrees with the informal assessment in [30].

Such descriptions are common in software specifications but such messages have not been observed between these things in the real world. Clearly, a machine does not send messages to parts. Instead the parts are moved by an operator[6].

Third, messages should not be interpreted as ontological things. Interaction with such message things would necessarily also have to occur by messages. Hence, a thing A interacts with thing B by exchanging the message thing M_1. For this to occur, A must interact with thing M_1 by means of another message thing M_2, etc. This leads to an infinite regress [7].

In summary, as message-passing does not have an ontological equivalent, "use associations" cannot be mapped to a suitable ontological concept. Thus, according to [1, 8] they possess no ontological semantics for the modelling of application domains.

4 Connection Semantics of Associations

Rumbaugh et al. [25] maintain the importance of an association as a conceptual, real-world construct: "We nevertheless emphasize that associations are a useful modelling construct for ... real-world systems ... " [25, p. 31], "It is important that relations be considered a semantic construct" [6, p. 467], "associations define the way objects of various types can be linked or connected - enabling the construction of conceptual networks" [32, p. 259], a "class relationship might indicate some kind of semantic connection" [26, p. 96][8].

We claim that associations used as "connections" between objects ("connection associations") have no ontological equivalent for the following reason. The term "semantic", as used in [6, 26], implies meaning and human interpretation. Hence, semantic connections are imposed on a domain as perceived by an observer, rather than directly observable in the domain. They represent properties that are *relevant or meaningful to a modeller or an observer.*

For example, the fact that a student attends a university (Fig. 1) is not observable in the domain; only the properties (e.g. student number, fee balance), and the behaviour of the student (e.g. attending class, sitting exams) and the university are observable. Some behaviour may be relevant and is interpreted as the student attending the university. To other observers, or for different model purposes, this behaviour may be irrelevant or may be interpreted as a different semantic connection.

---

[6] Note that it is quite possible in organizational settings for human actors to pass messages to each other: Letters can be exchanged, invoices sent and orders received. In contrast to the messages between parts and machines, ledgers and desks, or trucks and crates, the letters, invoices, and orders that are exchanged between human actors are substantial, physical things. .

[7] A similar argument is used in [31] to argue against association instances as objects.

[8] In Stevens' analysis [9], associations as connections correspond roughly to "static associations", although the latter are defined by their relationship to implementation, rather than their relationship to application domain elements.

Examining the way in which associations are used, e.g. the "attends" association in the above example, shows that there exist two distinct types of semantic associations.

First, some associations represent functions of past interaction of objects. Consider the association 'enrolled' between a student and a university. 'Being enrolled' is a result of past or ongoing interaction, namely that of the registration and enrollment process. The definition in terms of interaction also shows that the association is viewer or modeller dependent: A different definition of 'enrolled' may be based on class attendance rather than the act of registration. The association 'being on' between a shipping crate and the loading dock also depends on interaction, but not between the crate and the dock. These never interacted directly. Instead, this association is the result of past interaction between e.g. the crate and the forklift.

Second, consider the association 'distance' between two objects. Distance is defined by an observer or modeller based on properties of things, such as their location. Consequently, different distance measures are possible, for example in terms of road distance, traveling time, etc. This kind of semantic connection between objects does not depend on interaction, but represents functions of individual properties of things.

We conclude that, as semantic associations are observer dependent functions either of interaction or of intrinsic properties, they do not correspond to anything that exists in the application domain. Hence, they should be explicitly represented in functional form, rather than by an association construct that, because of its programming heritage, obscures their nature.

5 Conceptual Modelling with Mutual Properties

The previous sections showed a lack of ontological semantics for associations as 'use associations' (Sect. 3) and 'connection associations' (Sect. 4). Hence, they should not be used for describing application domains. Instead, we propose using ontological concepts directly. Since 'use associations' are intended to represent interaction, and 'connection associations' represent functions of the interaction history or individual properties, the relevant ontological concepts are those related to interaction and properties.

Interaction and Mutual Properties. The adopted ontology [12] specifies that two or more things may share mutual properties. Hence, any change of a mutual property in one thing is a change in all the things that share the property (Sect. 2). When a series of changes in an object A involves changes to a mutual property P, this may start a series of changes in the things that share this property, e.g. thing B. Thing A has acted on thing B through property P.

Hence, interaction can be described in terms of mutual properties. Instead of employing 'use associations' with poorly defined ontological semantics, we propose using mutual properties for conceptual modelling.

Using mutual properties in conceptual models requires a language construct (graphical symbol) to represent them. For this, we use attributes of association

classes: (1) Intuitively, the idea of an attribute corresponds well to the ontological concept of a property [22, 28]. (2) Attributes of association classes are graphically shown as connecting two or more objects (e.g. Fig. 2). Intuitively, this corresponds well with the idea of a single property being shared by two or more things.

Note that we merely borrow, for sake of convenience and familiarity, the graphical notation of association class attributes from UML. Associations and association classes themselves have no ontological interpretation and should not be used for conceptual modelling, as argued in Secs. 3 and 4. However, UML requires the use of a class symbol to represent attributes. This is a necessary evil that we accept in order to avoid introducing a new notation element. For this reason, association class symbols contain no name in the figures in Sec. 6. An alternative would be to introduce a new graphical or textual notation for shared attributes.

Functions of Interaction History. Ontologically, the history of interaction is the history of changes to mutual properties (Sect. 4). No graphical modelling exists in common object-oriented languages that could be used to express such functions. Instead, we propose to use simple textual notation, e.g. the following example in Prolog like notation:

```
property(downtown, location, 10).
property(campus, location, 20).
property(hospital, location, 15).
   ...
distance(O1, O2, D) :-
    property(O1, location, X),
    property(O2, location, Y),
    D is X - Y.
```

Now we can ask for the distance from downtown to the campus:

```
distance(downtown, campus, D).
```

Similarly, an example for a function of the event history of the world is the following employment association (again in Prolog):

```
history(acmecorp, johnsmith, hires, 20030701).
history(acmecorp, janedoe, hires, 20031001).
history(acmecorp, johnsmith, fires, 20040101).
    ...
connection(O1, O2, employs) :-
    history(O1, O2, hires, Time1),
    \+ history(O1, O2, fires, Time2).
connection(O1, O2, employs) :-
    history(O1, O2, hires, Time1),
    history(O1, O2, fires, Time2),
    Time1 > Time2.
```

We can now ask whether Acme Corp. employs Jane Doe:

```
connection(acmecorp, janedoe, employs).
```

As noted above (Sect. 4), semantic associations are relative to a modeller or observer. Hence, they should not be a part of the domain model, but explicitly noted as part of a perspective on or interpretation of the domain. This also means that different domain observers, modellers, or users of the final information system, can define a different set of functions that are relevant to them.

6 Case Study Example

This section presents an example that demonstrates the proposed conceptual modelling technique. The technique was used in an actual IS development project, carried out at a large North American university. The project goal was to develop an Internet based system to allow prospective students the opportunity to update their application information on an ongoing basis and enable them to see whether they meet application criteria.

This section focuses on the identification and representation of mutual properties and interaction. Only a brief excerpt of the complete analysis is provided, focusing on high school students, high schools, teachers, the university and the application process.

High school students and teachers are modelled as object classes. The relevant interaction (for purposes of the analysis) between a student and a high school is that of receiving course grades. Interaction occurs by means of changes to mutual properties (Sect. 5). Hence this is modelled using attributes of an association class that represent the mutual properties. In Fig. 2 the attributes are multivalued, i.e. there is a course name, a course year and a course grade for each course the student completed[9]. Teachers interact with students by changing the values of these attributes[10].

The properties in Fig. 2 arose themselves out of interaction. Therefore, prior mutual properties between teachers and students must have existed, such as homework submitted and read. Ultimately, the mutual properties are those of an interaction medium that is manipulated by the interacting objects. However, such media are rarely of interest to the conceptual modeller and are thus abstracted from. Note that this abstraction is a conscious decision of the modeller, and is always dependent on the purpose of the model.

Communication between the high school and the student can lead to meaningful semantic connections. For example, one can model the semantic connection

[9] Note again that we borrow only the graphical notation of association class attributes from UML. Associations and association classes themselves have no ontological interpretation. However, UML requires the use of a class symbol to represent attributes. Note that the association class itself is not named, as we have not assigned it ontological meaning (Sect. 5).

[10] Obviously, there occurs more interaction, e.g. in the classroom, which is not relevant to the university's admission system.

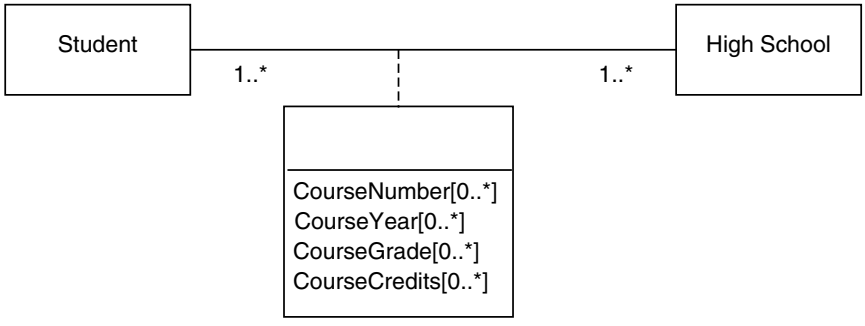

Fig. 2. Example: Student - High School interaction

'has graduated from' as a function of the interaction history: When the sum of course credits is above a certain threshold, we consider the student to have graduated[11]. Having graduated in this sense is an observer- or modeller-dependent function, rather than an object or event in the application domain. It may be of interest only in certain contexts, or it may be defined differently by different observers or modellers, e.g.

```
history(janeDoe, centralHigh, 100, 20040701).
history(johnSmith, southernHigh, 120, 20040701).
history(jimMiller, northernHigh, 150, 20041201).
    ...
property(Student, HighSchool, graduated) :-
    history(Student, HighSchool, creditEarned, Time1),
    creditEarned > 100.
```

We can now ask whether Jane Doe has graduated from Central High:

```
property("janeDoe, centralHigh, graduated).
```

Changing the mutual properties in a certain way will lead to a change in the student where she considers applying to a university. Applying to the university is interaction. Interaction implies mutual properties between the student and the university that can be manipulated by the student. We can either abstract from this information and simply call it "application information", or we can model the specific properties, e.g. "program applied for", "school grades submitted", etc. The student can change these shared properties. As a result of these changes, the admission process in the university is initiated (Fig. 3)[12].

This type of modelling forces the modeller to make a distinction between the grades awarded by the high school (they are shared between the school and the student), and the grades reported by the student for the admission request (they

[11] Abstracting from the formalities which are often attached for graduating.
[12] See previous note.

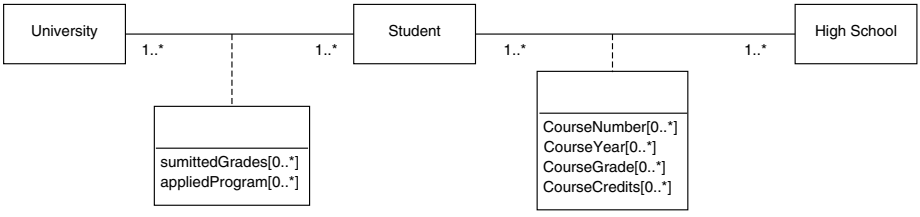

Fig. 3. Example: Student - University interaction

are shared between the student and the university). It may well be that these are different, e.g. due to the fact that the school may revise preliminary grades, or the student reports only a subset of grades to the university.

Changes to the shared properties are interpreted as interactions. Consequently, they can be modelled in UML interaction diagrams. Fig. 4 shows an example corresponding to the above excerpt of the case study.

A subsequent discussion with the project leader showed that modelling of mutual properties can help explicate the meaning of association class attributes. It forces the developer to identify precisely what is represented: "It's normally difficult to model a course ..., because it is a relationship. ... What do you mean by a course? The curriculum, the interaction, the grade?". In contrast, the above model (Fig. 2), with the clear ontological semantics proposed in Sect. 5, shows that students and universities share a set of properties that can be modified by either to initiate interaction. The lead system designer also noted the beneficial effect of the clear semantics for association class attributes: "Visually, this ... gives you a better sense of the relationships."

The case study results show that the proposed modelling method is feasible and can be applied in real projects. The subsequent discussion shows that the technique leads to clearer models and to models with clearer meaning than the use of associations for conceptual modelling.

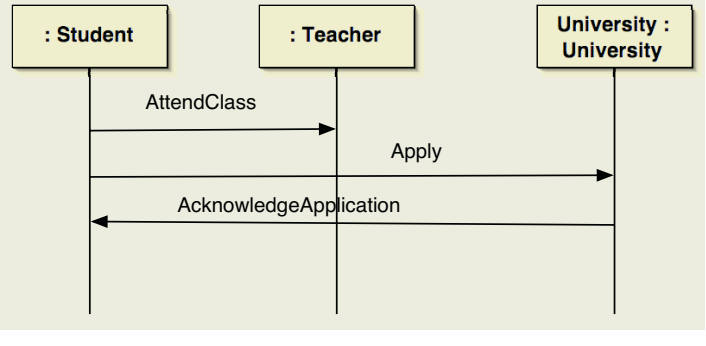

Fig. 4. Example: Student - University sequence diagram

7 Software Design and Implementation Without Associations

The main advantage of object-oriented methodologies is the seamless transition from conceptual modelling to system design, and to system implementation using the same paradigm and set of language constructs. This section shows how the proposed conceptual modelling technique can be seamlessly transferred to system implementation.

We describe a technique for object-oriented implementation that does not use object references or method calling. It implements shared mutual properties (conceptual level) by 'binding' object attributes (programming level) of two or more objects to ensure that they maintain identical values.

Aspect-Oriented Implementation. Attribute binding can be achieved in an efficient and completely transparent way by using aspect-oriented programming (AOP) techniques[13]. For demonstration purposes, an aspect for attribute binding has been developed in Aspect/J, a widely used AOP extension to Java [33]. The aspect allows binding together any two or more attributes of different objects. The following code shows two fictitious objects $a1$ and $a2$, both instances of class A, that share a property single mutual property by binding their respective properties $varA$ of $a1$ and $varB$ of $a2$. The example demonstrates that this binding is transparent to the application developer: No special bound properties or accessor methods need to be declared.

```
public class A {
  public Integer  varA, varB;
}

public static void main() {
  A  a1 = new A();
  A  a2 = new A();

  PropBinding.addBindings(a1, "varA", a2, "varB", "InteractionName");
}
```

As can be seen from the example, an interaction name can be associated with each binding. The aspect monitors the write accesses to bound attributes and updates them accordingly, ensuring that they maintain identical values. The aspect also maintains a history of every update of bound attributes. This history can be accessed and searched by object identifier and interaction name so that functions of the interaction history (as described in Sect. 5) can be defined.

[13] AOP allows the separate development and implementation of different aspects of an implementation (e.g. logging, security, persistence) in a transparent way. Individually programmed aspects are woven together when compiling the final software product. Mature AOP tools are available for most languages (e.g. AspectWerkz, Aspect/J, AspectC++, Aspect#).

Since two or more bound attributes represent a single shared mutual property, changes to one attribute are propagated to all bound attributes in a single atomic step. The aspect recursively propagates changes along bound attributes, until all bound attributes possess equal values. Only then is the program allowed to resume execution.

As there exist no object references, objects cannot interact by method calls along such references. Consequently, in a single-threaded model, explicit notification is necessary to pass control from one object to another. In a multi-threaded model, notification is not necessary, as control is not passed between objects, but may desirable in certain situations or for certain applications.

Notification in a Single-Threaded Model. In a single-threaded model, the implementation of the aspect allows explicit registration of a callback method. This presents two potential problems. First, the order of callback execution must be determined. In the current implementation, callbacks are executed in the order in which the attribute bindings are declared; other execution orderings are possible.

Second, a set of objects may possess shared mutual properties in such a configuration that actions by a notified object change bound attribute values before all remaining objects have received notification of the original changes. For example, an object t changes the value of a bound attribute k from value a to b. After propagating this change to objects u and v, the notification callback of object u is called and changes the value of the attribute k from b to c before object v is notified of the change from a to b by calling its callback method. From the perspective of the object v, the first change to b never happened as the object never gained control while k possessed value b. Note that objects cannot be notified before a change has been propagated to *all* bound attributes. This is because the attributes represent a *single* shared mutual property, and thus must be updated in a single atomic action.

Therefore, in the single-threaded model, the semantics of the implementation depend on the ordering of execution of callback. This requires great care by the programmer.

Notification in a Multi-Threaded Model. In multi-threaded applications, no callback methods are possible. Instead, all notification must be done by means of event signaling. The implemented aspect provides an event queue for every thread/object into which notifications can be added. Consequently, in the multi-threaded model the semantics of the attribute binding are well-defined and independent of the ordering of callback execution.

In this model, too, changes to attributes can occur before prior changes have been processed by the object. In the multi-threaded model, each object possesses its own notification queue, containing notifications about attribute changes. While a change notification for a bound attribute is still queued, i.e. it has not been processed by the object yet, this attribute may be changed again. However, for the same reasons as in the single-threaded model, only net change notifications are provided. Hence, whenever a new change notification is added

to the queue, the net-effect of this and all previously queued notifications will be computed. This net-effect notification replaces all other elements in the queue that notify of changes in the same attribute.

In summary, the implementation of mutual properties by attribute binding is efficient (linear in the number of bindings between properties), and possesses well-defined execution semantics.

LibPropC++. As an alternative to the aspect-oriented implementation of mutual properties presented above, an existing programming library has been considered. Property binding has been proposed and implemented for user-interface objects in a C++ library (LibPropC++) [34]. However, compared to the previous approach, this library has a number of weaknesses. (1) Its use is not transparent as it requires that object attributes must be explicitly declared as properties and appropriate accessor methods must be provided. (2) Binding of attributes in LibPropC++ is not designed to replace method calls. Objects still possess object reference pointers and need to call methods of other objects. (3) The library does not maintain an interaction history, so it cannot provide a foundation on which functions of past interaction can be defined.

8 Conclusions and Further Research

This research was motivated by problems with the meaning of associations in conceptual models. We identified the semantics of associations with respect to software and attempted to transfer this to conceptual modelling. However, as the interaction mechanisms in application domains (specified by an ontology) do not rely on message passing and method calling, associations as enablers of message passing have no application domain meaning.

We then discussed the intended use of associations to indicate semantic "connections" between objects. Our analysis showed that these "connections" are used to describe observer dependent properties, rather than substantial elements in the application domain. Hence, associations as "connections" have no ontological semantics in conceptual modelling.

This paper proposes an alternative technique for conceptual modelling that is rooted directly in ontology. To this effect, it defines ontological semantics for attributes of association classes, by mapping them to mutual properties. These mutual properties are the linkage between things and the means by which interaction occurs.

A case study was presented that demonstrates the use of this modelling technique. The paper further demonstrated that appropriate software technologies exist to seamlessly transfer this ontologically based modelling technique to software design and implementation.

To summarize, since the meaning of associations is undefined, we suggest not to use them. Instead, we propose using ontological concepts directly, by representing them as association class attributes. The contributions of this paper are threefold. (1) We have identified and pointed out ambiguities in the meaning of

associations. (2) We have shown that associations, as intended by their originators, have no ontological equivalent, i.e. no semantics. (3) We then proposed an ontologically based modelling technique and shown that it can be implemented efficiently in object-oriented programming languages.

While the initial case study, described in parts in Sect. 6, shows that the modelling and implementation approach are feasible, further research in three areas is needed. (1) The implementation on the programming level must be further analyzed to firmly define the implementation semantics of the approach. (2) The approach needs to be evaluated in a wider set of domains. To this effect, further case study applications will be undertaken. (3) Finally, the benefits of the proposal, in terms of model interpretation and model understanding, need to be determined in a controlled setting.

References

1. Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? IEEE Computer (2004) 64–72
2. IBM: Developing object-oriented software: an experience-based approach. Prentice Hall, Inc., Upper Saddle River, NJ (1997)
3. Bahrami, A.: Object oriented systems development. Irwin/McGraw-Hill, Boston, MA (1999)
4. Embley, D.W.: Object-oriented systems analysis: a model-driven approach. Prentice Hall, Inc., Englewood Cliffs, NJ (1992)
5. Siegfried, S.: Understanding object-oriented software engineering. IEEE Press, New York, NY (1995)
6. Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: Proceedings of the 1987 Conference on Object Oriented Programming Systems and Languages and Applications, Orlando, FL., ACM Press (1987) 466–481
7. OMG: The Unified Modelling Language Specification. Version 1.5. (2003)
8. Wand, Y., Weber, R.: On the ontological expressiveness of information systems analysis and design grammars. Journal of Information Systems (1993) 217–237
9. Stevens, P.: On the interpretation of binary associations with the unified modelling language. Software and Systems Modelling **1** (2002) 68–79
10. Genova, G., Llorens, J., Martinez, P.: The meaning of multiplicity of n-ary associations in UML. Software and Systems Modelling **1** (2002) 86–97
11. Bennett, B.: Space, time, matter and things. In: Proceedings of the 2001 International Conference on Formal Ontologies in Information Systems FOIS, Ogunquit, Maine. (2001) 105–116
12. Bunge, M.A.: Ontology I: The Furniture of the World. Volume 3 of Treatise On Basic Philosophy. D. Reidel Publishing Company, Dordrecht, Holland (1977)
13. Chisholm, R.: A Realistic Theory of Categories - An Essay on Ontology. Cambridge University Press, Cambridge (1996)
14. Degen, W., Heller, B., Herre, H., Smith, B.: GOL: A general ontological language. In: Proceedings of the 2001 Conference on Formal Ontologies in Information Systems FOIS, Ogunquit, MA. (2001) 34–46
15. Fensel, D., van Harmelen, F., Horrocks, I., McGuiness, D.L., Patel-Schneider, P.F.: OIL: An ontology infrastructure for the semantic web. IEEE Intelligent Systems (2001) 38–45

16. Niles, I., Pease, A.: Towards a standard upper ontology. In: Proceedings of the 2nd International Conference on Formal Ontologies in Information Systems FOIS, Ogunquit, Maine 2001. (2001) 2–9
17. Sowa, J.F.: Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks Cole, Pacific Grove, CA (2000)
18. Bodart, F., Patel, A., Sim, M., Weber, R.: Should optional properties be used in conceptual modelling? A theory and three empirical tests. Information Systems Research **12** (2001) 384–405
19. Evermann, J.: Using Design Languages for Conceptual Modelling: The UML Case. PhD thesis, University of British Columbia, Canada (2003)
20. Gemino, A.: Empirical Comparisons of Systems Analysis Modeling Techniques. PhD thesis, University of British Columbia, Canada (1999)
21. Green, P., Rosemann, M.: Integrated process modelling: An ontological analysis. Information Systems **25** (2000) 73–87
22. Opdahl, A., Henderson-Sellers, B.: Ontological evaluation of the UML using the Bunge-Wand-Weber model. Software and Systems Modeling **1** (2002) 43–67
23. Parsons, J., Wand, Y.: Using objects for systems analysis. Communications of the ACM **40** (1997) 104–110
24. Wand, Y., Storey, V.C., Weber, R.: An ontological analysis of the relationship construct in conceptual modeling. ACM Transactions on Database Systems **24** (1999) 494–528
25. Rumbaugh, J., et al.: Object Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ (1991)
26. Booch, G.: Object oriented design with applications. Benjamin/Cummings, Redwood City, CA (1991)
27. Coad, P., Yourdon, E.: Object-Oriented Analysis. Yourdon Press, Englewood Cliffs, NJ (1990)
28. Evermann, J., Wand, Y.: An ontological examination of object interaction in conceptual modeling. In: Proceedings of the Workshop on Information Technologies and Systems WITS'01, New Orleans, December 15-16, 2001. (2001) 91–96
29. Parsons, J., Wand, Y.: The object paradigm – two for the price of one? In: Proceedings of the Workshop on Information Technology and Systems WITS 1991, New York, NY. (1991) 308–319
30. Cook, S., Daniels, J.: Designing object systems: object-oriented modelling with Syntropy. Prentice Hall, Hertfordshire, UK (1994)
31. Graham, I., Bischoff, J., Henderson-Sellers, B.: Associations considered a bad thing. Journal of Object-Oriented Programming **9** (1997) 41–48
32. Martin, J., Odell, J.J.: Object-oriented analysis and design. Prentice Hall, Englewood Cliffs, NJ (1992)
33. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications, Greenwich, UK (2003)
34. Porton, V.: Binding together properties of objects. http://excode.com/articles/binding-properties.html (2004) Last accessed Sept 23, 2004.