B usiness
O bject
R eference
O ntology
Program

s i m p l i f y i n g   s e m a n t i c s

Working Paper

BG1

Business Ontology: Graphical Notation-1

Constructing Signs for Business Objects

Issue: Version - 4.01 - 01-July-2001

BORO

BG1

Business Ontology: Graphical Notation - 1

Constructing Signs for Business Objects

Contents

CONTENTS BG1

BG1

BUSINESS ONTOLOGY:
GRAPHICAL NOTATION - 1

CONSTRUCTING SIGNS FOR BUSINESS OBJECTS

1 Introduction

The BORO paper *OP4—Business Object Ontology Paradigm* helps us to develop an understanding of what business objects are. However, that is only a precursor, albeit an important one, to the real business of modelling. The accuracy and flexibility of the object ontology give us a powerful way of seeing the business. We harness this power by building models that describe what we see.

In the *BG—Business Ontology: Graphical Notation* we focus on object syntax; in other words, on how we 'write' the signs for objects and their patterns. We learn a notation for describing business objects in models. In this paper we look at the individual signs for the main types of business object. These are the signs with which we build the business object model. We focus on what they mean and how they work. Then, in the next paper (*BG2— Constructing Signs for Business Objects' Patterns*), we look at signs for business objects' patterns.

The *BG—Business Ontology: Graphical Notation* help us to develop an understanding of object syntax and the notation for business object models. They also deepen and broaden our understanding of object semantics. Using a notation for describing objects naturally leads to a better understanding of them. For example, because the notation explicitly signs the key structural patterns (super–sub-class, class–member and whole–part) these are clearly visible and so easier to understand. And because the notation gives each pattern a different sign, it helps us see that they are different.

Learning this notation is essential for understanding the *MW—The BORO Methodology: Worked Examples* Papers, which work their way though examples of re-engineering existing computer systems into business objects. As well as providing useful illustrations of both object semantics and object syntax, these examples will provide us with further experience of how the notation is used.

1.1 Main types of business object

In this paper we look at the object notation's basic signs for the following main types of object:

- Individual objects,
- Class Objects,
- Tuple objects (and, more specifically, whole–part tuple objects), and
- Dynamic objects.

We see how the signs are constructed, what they mean and how they are used.

1.2 Why use a two-dimensional notation for a multi-dimensional model?

Before we look at the notation, I should explain why it is two-dimensional. I have asserted a number of times in *The BORO Working Papers* that objects free us from the two-dimensional constraints imposed by paper and ink technology. I have suggested that this enables us to take advantage of computer technology's ability to handle multi-dimensional structures. However, the notation we are about to look at is on paper and so only two-dimensional. Why is this so and why doesn't it constrain the overall business model to two dimensions? To understand the answers to these questions, we need to look closely at the 'technology' that we use when business modelling.

1.2 Why use a two-dimensional notation for a multi-dimensional model?

Modelling the business is currently done by humans. It is human brains, and not computers, that construct and revise the business model. This means that the human mind needs to 'interface' with the business model. The object notation has to be easily read by humans.

Human biotechnology and computer technology both constrain how we can 'process' a business model. (Processing currently means 'see'—we do not touch or hear business models, let alone taste or smell them.) Computer technology constrains our visual 'interface' to two dimensions. The 'inputs' we receive from a computer system, whether on a screen or a print-out, are two-dimensional. The biotechnology of human eyes' retinas is also constrained to two-dimensional images. Furthermore, human brains are trained to process the kind of information in business models on two-dimensional surfaces.

From a practical point of view, this means that the sensible solution is to construct and review the multi-dimensional business model through two-dimensional views. Digestible two-dimensional chunks are an easy and effective way for the human brain to absorb the model. And its multi-dimensionality is not affected.

This solution can give computer technology an important role. Business models are static—in both traditional and object modelling; they map the time dimension onto the spatial dimensions. This means that the business model is not itself an information processing system; it is only stored information—data. However, producing a two-dimensional view of a multi-dimensional business model does take processing. So, at least in theory, we need the power of a computer to store the multi-dimensional model and produce the two-dimensional views.

In practice, using a computer with good CASE tool software can make the administration of storing the model and the processing of views easier, but it is not essential. I have found that constructing a multi-dimensional business model from two-dimensional paper views (in other words, using paper and ink technology) is a practical option—particularly when working with small models. The vital decisions about the construction and review of the business model happen in the brain of the business modeller, which is not excessively hampered by a paper model.

This is just as well because CASE tool software is not yet fully geared up for business object modelling. At a more mundane level, I have found a computer graphics package an invaluable aid to producing the paper views; the results are much more legible than hand-drawn ones. While computers are not essential at the business modelling stage, it is a different story when the model is turned into a working system. Then, computer technology becomes essential.

2 Constructing signs for individual objects

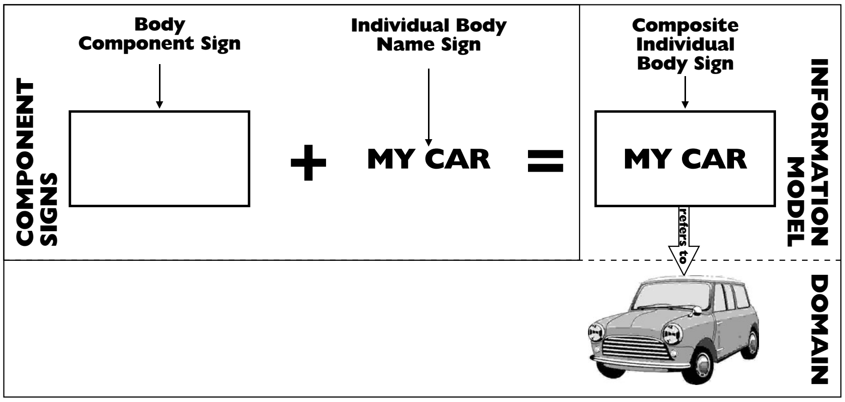
Let's now look at the two-dimensional notation. In object semantics, an individual object is a plain and simple extension. This is referred to (directly mapped onto) by a sign in the model. We use different signs for the different types of individual objects:

- Individual body, and
- Individual event.

2.1 Constructing a sign for an individual body

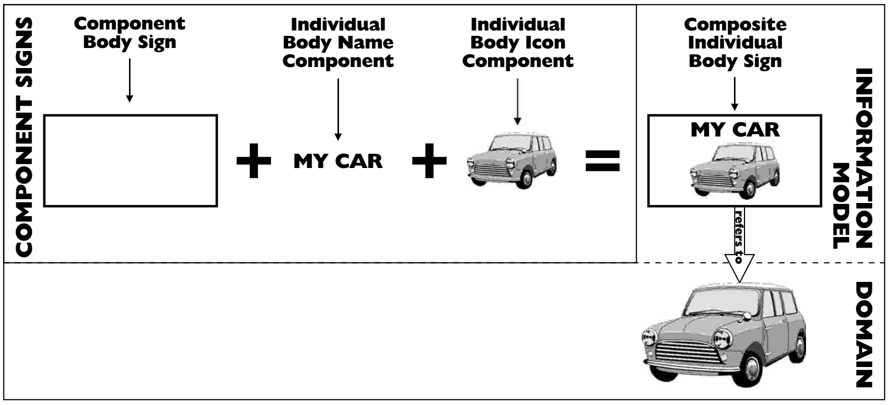
The sign for an individual body is constructed out of two components. A body sign, which is a rectangle, and a name sign that is the name of the body. We put the name sign inside the body sign (shown in *Figure BG1-1*). This figure also diagrams the extra-model reference link between the individual body sign in the model and the body object in the domain.

Figure BG1–1
Individual body sign



Sometimes, to aid recognition, we include an icon of the individual body inside the body sign (shown in *Figure BG1–2*).

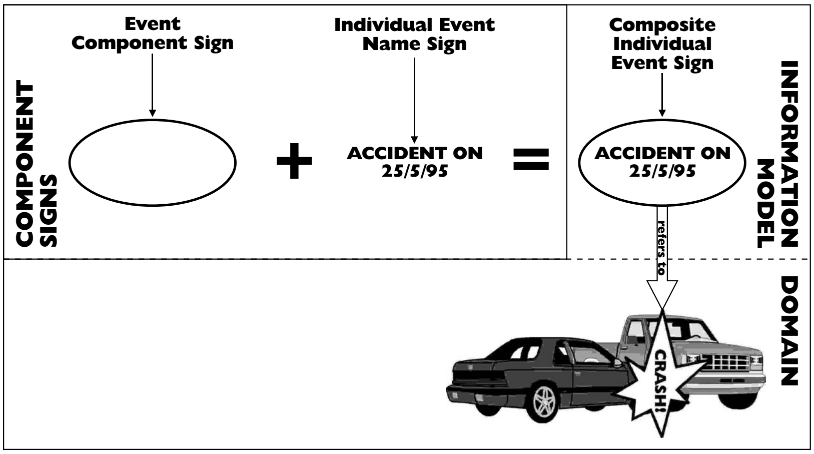
Figure BG1–2
Alternative individual body sign


COMPONENT SIGNS
Component Body Sign
Individual Body Name Component
MY CAR
Individual Body Icon Component
Composite Individual Body Sign
MY CAR
refers to
INFORMATION MODEL
DOMAIN

2.2 Constructing a sign for an individual event

We construct the sign for an individual event in a similar way out of two components. An event sign, which is an ellipse, and a name sign, which is the name of the event. We again put the name sign inside the event sign (shown in *Figure BG1–3*).

Figure BG1-3
Individual event sign



2.3 Constructing individual object name sign components

The shape of the component body and event signs show their type; so, all signs of the same shape are the same type. We use everyday language for the name components. These differentiate between signs for different objects. They help us recognise which object a particular composite sign refers to. To avoid confusion, a convention, within each model, indicates that the name signs are unique; no two individual objects have the same name sign.

3 Constructing signs for classes of objects

We now look at how to construct the signs that refer to classes. We also look at the signs for the class pattern's two important tuples connecting classes:

- Class–member, and
- Super–sub-class.

3.1 Constructing a sign for a class of individual objects

We first look at how to construct a sign for a class of individual objects. Just as there are different signs for an individual body and an individual event, there are different signs for a class of individual bodies and a class of individual events.

3.1.1 Constructing a sign for a class of individual events

Remember that we construct a class of individual objects by collecting together the extensions of those objects and treating the collection as a single object. This single object is what the class sign refers to.

A class of individual events only contains events, so we use the same elliptical event sign as a component. We put the name of the class in this ellipse. We then indicate that we have constructed the event class out of individual events by putting two smaller superimposed ellipses—signs for the member events—in the bottom right corner. *Figure BG1–4* gives an example. In this example, we have also put the name sign for a member of the class, 'accidents', in the smaller ellipse. Often, however, a member name sign takes up too much space and we have to leave it out.

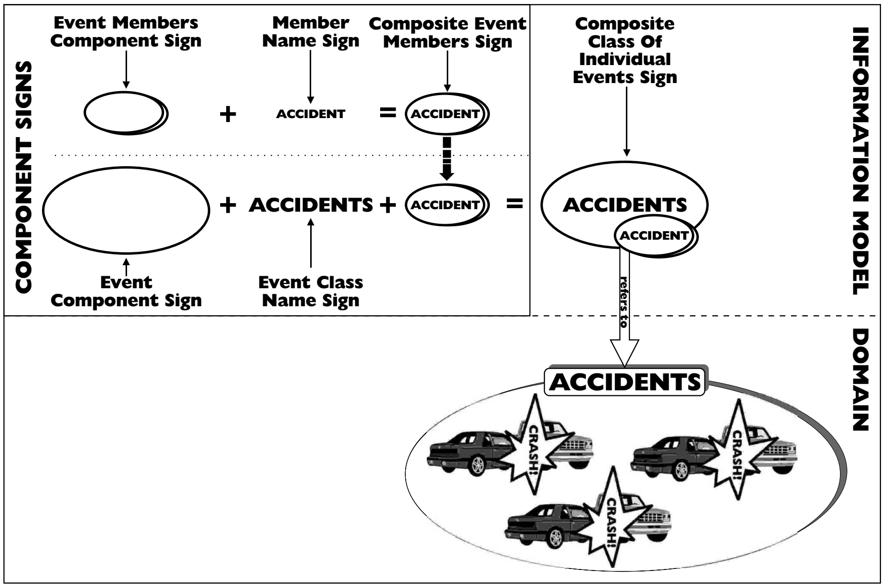
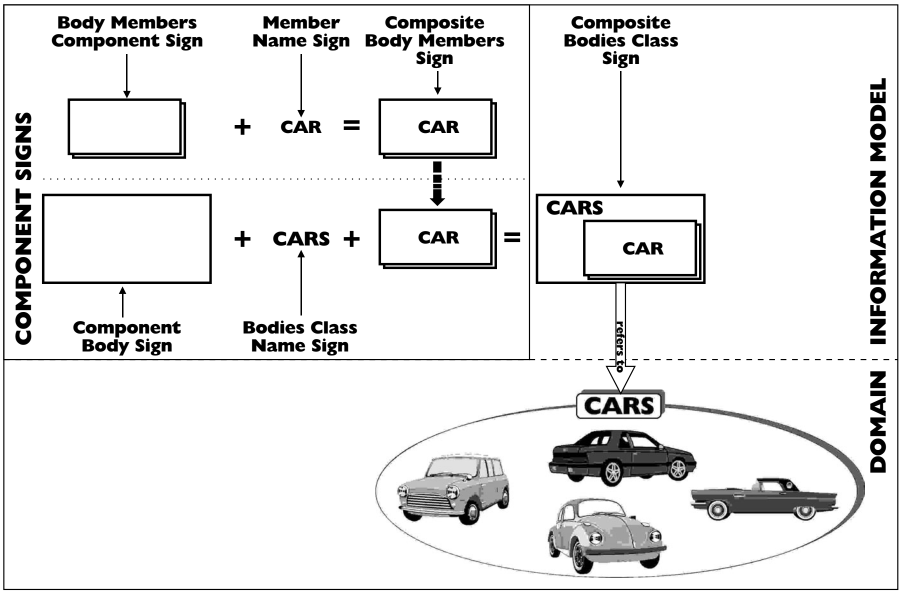


Figure BG1–4
A class of individual events sign

3.1.2 Constructing a sign for a class of individual bodies

The sign for a class of individual bodies follows the same pattern. We use the same rectangular box sign that we used for individual body signs and show the class has members using two smaller superimposed rectangular boxes in the bottom right corner. Again we name the class and, if there is enough space, the potential members. The name sign for the class is in the larger class rectangle and the name sign for a member of the class is in the smaller member rectangles (shown in *Figure BG1–5*).

Figure BG1-5
A class of individual bodies sign



3.1.3 Constructing class name and member name sign components

We use class names, as we used individual object names, to differentiate the signs (and so identify the classes). As before, we keep the names unique within each model. Unlike some notations, we use different names for a class and its members. These other notations, will, for example, call both the cars class and its individual members 'car'. I have found that this causes confusion. In object semantics, a clear distinction is made between the class and its members. In object syntax, this is reflected in different names for the class and its members—often, as here, the plural and singular forms of a noun. Using the car example, the class is called (and so the class name sign is) 'cars' and an individual member is called a 'car'.

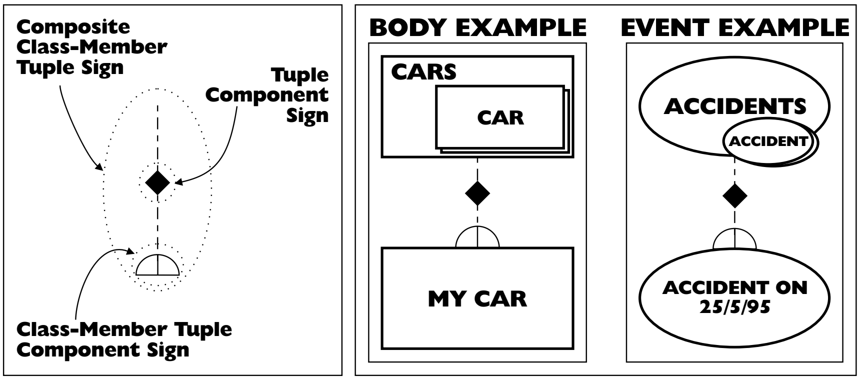
3.2 Constructing a sign for a class-member tuple

Individual objects (whether bodies or events) that are members of a class belong to that class; that is, there is a class-member tuple connecting each member object and the class. This tuple is central to the notion of a class, so we need to have a sign for it in our notation.

3.2.1 Classes and members

The class–member tuple is, strictly speaking, a couple <individual object, class> that belongs to the class–member tuples class. We model it by drawing a class–member tuple sign. This is a dashed line joining the relevant class and member signs. It has, at the member end, a semi-circle with a line through it (shown in Figure BG1–6). This is intended to look like the Greek character epsilon 'ε'—the mathematical sign for class membership. We show that the connection is a tuple by putting a black diamond, the sign for a tuple, on the line.

Figure BG1–6
Class–member tuple sign

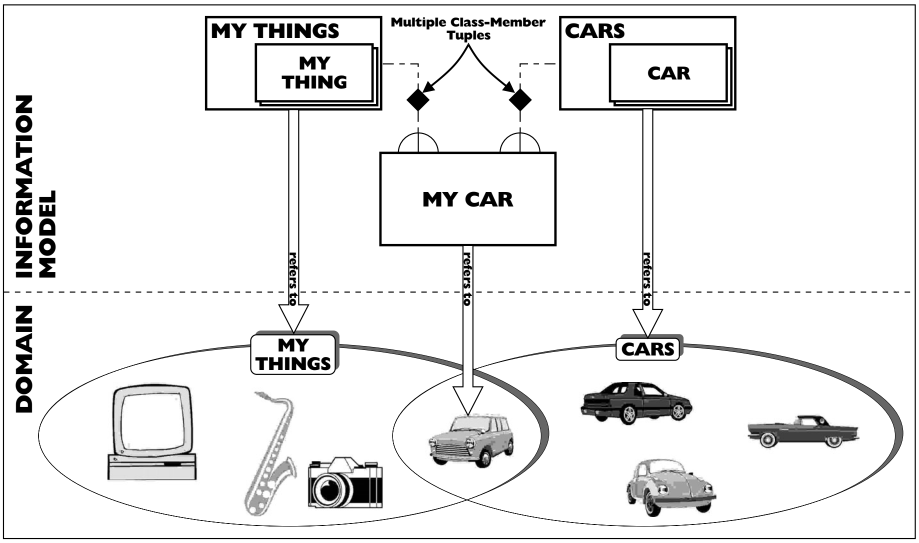


As you can see from the figure, we use the same class–member tuple sign for body and event classes. This is understandable because the underlying pattern is the same. There is also an informal convention (followed in Figure BG1–6) that we draw classes higher up the page than their members; though in some complicated diagrams, it is not possible to do this.

Members of more than one class

Unlike some notations, this can easily model an object that is a member of more than one class—what we called multiple classification. We just join the object's sign to each of the relevant class signs with class–member tuple signs (shown in Figure BG1–7).

Figure BG1-7 Multiple class-member tuple signs



An accurate class-member sign pattern

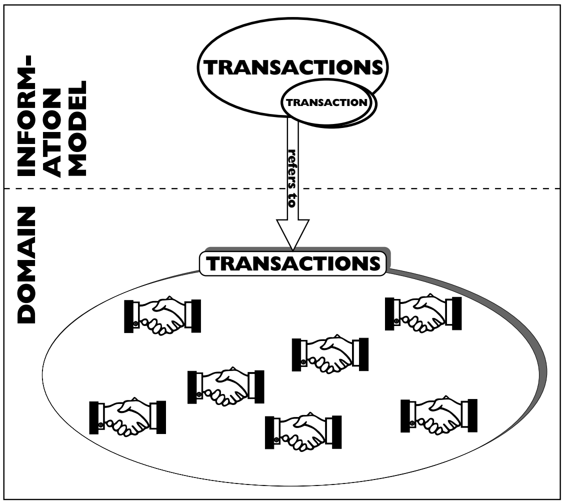
The class-member pattern is a very strong pattern; one that is central to object semantics. So is its reflection in the information model, the class-member sign pattern. Because one is a reflection of the other, they have similar patterns. However, a common mistake is to assume they have the same pattern. This is not so. The information model's 'ignorance' leads to differences. We look at one of these now.

It is natural and normal to assume a class has members. A class is a class because it captures some common patterns of its members; so, it is reasonable to assume it has members. Because a class sign's purpose is to model a class, it also appears reasonable to assume that it will reflect this characteristic—to think that a class sign is always linked to some member signs (sometimes called instances).

But this is wrong—it turns out that it is natural and normal in an information system for a class sign to have no member signs. In fact, it is quite common. For example, we talk about types of wild animals without having any notion of a particular animal. We talk about elephants (the class elephants) or gorillas (the class gorillas) without ever knowing a particular elephant or gorilla (members of the classes elephants and gorillas). Our minds, as information systems, have no member signs for the class signs we are using.

Member sign-less (instanceless) class signs are an almost universal rule in the shipped versions of business computer system packages. For example, accounting packages are usually shipped with a transactions file (a class sign) that has no individual transactions (in object terms, member signs)—the situation shown in *Figure BG1–8*.

Figure BG1–8
The instanceless transactions class sign



3.2.2 Modelling lack of membership information

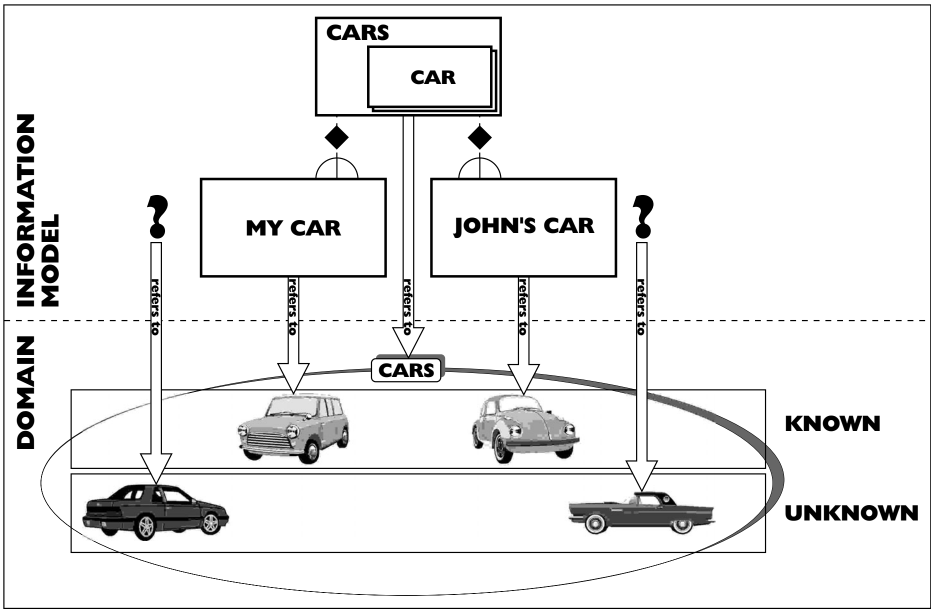
No information system is completely ‘informed’. This includes human minds, which are considered information systems. We now illustrate this with two types of ignorance that arise when modelling the class–member pattern:

- Unknown members, and
- Unknown membership.

Modelling unknown members

For every class in an information system, when we look at it objectively from outside the system, we can divide its members into known and unknown. Known if the information system has a sign for them; otherwise, unknown (or, more accurately, unknown by the information system but known by us—otherwise, we could know that they were unknown). This distinction has nothing to do with the class or its members. It is a feature of the information model (shown in *Figure BG1-9*.)

Figure BG1-9
Known and unknown class-members



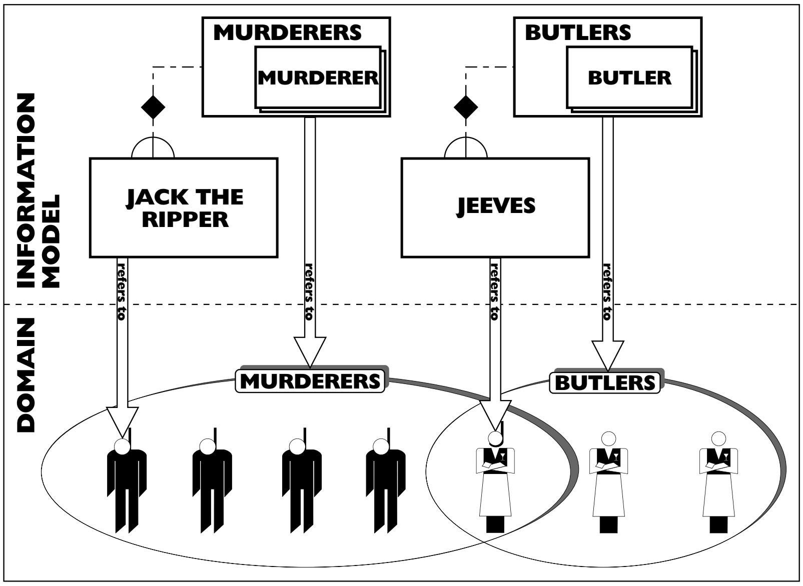
It is common for a member to be unknown because it has not yet come into existence. When it does, the information system can then construct a sign for it. This happens, for instance, when a new country is created. It happened recently for the Czech Republic and Slovakia; ten years before they were created, no-one would have known about these two countries. But when Czechoslovakia decided to separate into two countries, people began to become aware of them. The extension of the class countries in the real world did not change. All that changed was the construction of new signs for the Czech Republic and Slovakia in information systems.

Modelling unknown class membership

It is important to remember that, not only do signs have to be constructed in the system for the class's members, but also for the class-member tuples connecting members to the class. Our minds automatically and unconsciously supply this link; so, it is easy to forget that it needs to be explicitly constructed. We can illustrate this with an example where the system starts off knowing about the member of a class but not its membership of the class.

Consider an Agatha Christie type of detective novel, in which a murder has been committed in a country house. At the beginning of the novel, we are introduced to each of the characters; the butler, the lord of the manor, the chambermaid, and so on. We know that, by convention, one of these is the murderer. Assume that Jeeves the butler is the murderer—in other words, a member of the class of murderers. Now, when we start reading the book we know Jeeves and know the class murderers, but have not (yet) found out that Jeeves is a member of the class murderers. Figure BG1–10 shows the state of our knowledge.

Figure BG1–10 Jeeves the butler as an unknown member of the class murderers



At some stage, as the plot unfolds, we realise the butler is the murderer. As we already have signs for the butler and the class murderers, all we need to do is construct the class–member tuple sign between the two. The result is shown in Figure BG1–11. Notice that there is no change in the domain. The butler belonged to the class murderers all along, what happens when we solve the mystery is that we learn of his membership.

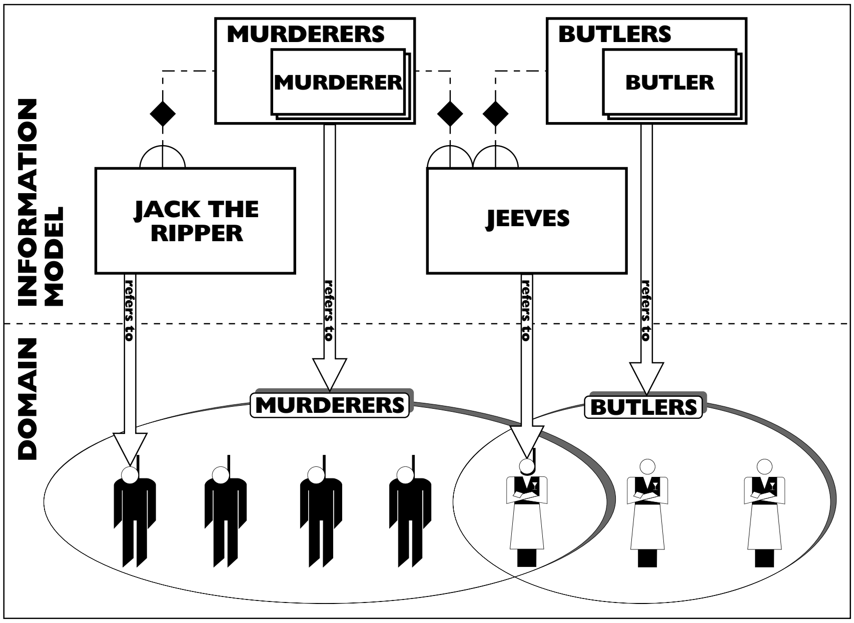


Figure BG1–11 Jeeves the butler as a known member of the class murderers

The constructive nature of modelling

This and the previous example of 'ignorance' have highlighted what might be called the constructive nature of signs and so information. Signs only exist if we construct them. This is obvious when we start to think about it. How could a sign exist that has not been constructed? We shall see, as we work through this paper, the fundamental impact this constructive nature has on information modelling.

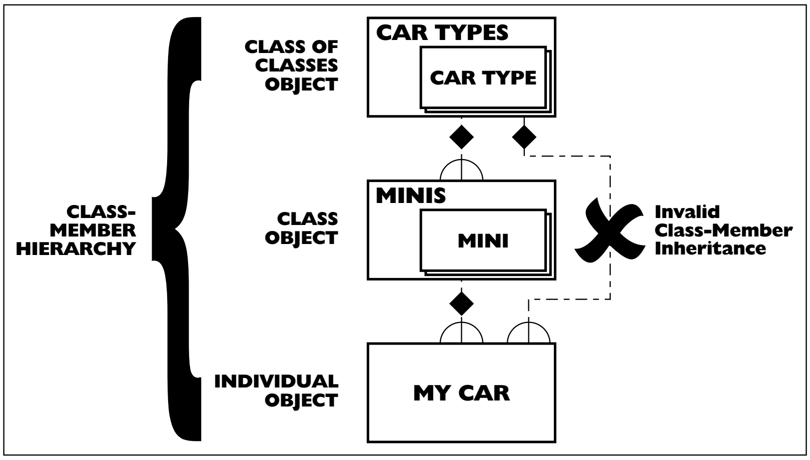
3.2.3 Classes as members of classes

So far we have only considered classes of individual objects. However, we recognised in the logical paradigm that classes were objects and so could, like individual objects, be collected together into classes—giving us classes of classes objects. This means that the class–member tuple, with its <class, member> format, can have any type of object (individual, class or tuple) in its member place. We now look at how we sign the class–member pattern for classes of classes, and, in the process, see how we capture class–member hierarchy patterns in the model.

Class-member hierarchy

The sign for describing a class as a member of a class is exactly the same as that for describing an individual object as a member of a class. The example shown in *Figure BG1-12* is taken from our original example of classes of classes in *OP3—Logical Ontology Paradigm* (illustrated by OP3's *Figure OP3-27* and *Figure OP3-28*). As we can see, the class-member sign is used in the same way for members that are classes as for members that are individual objects. *Figure BG1-12* is also an example of how we model a simple class-member hierarchy.

Figure BG1-12 Car types—an example of a class-member hierarchy



Class membership inheritance

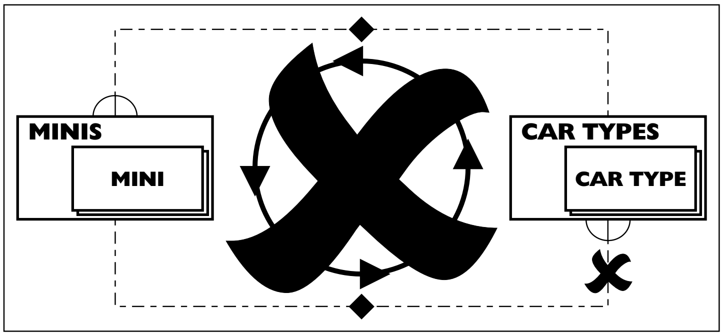
We shall see later on that various patterns are inherited down (and sometimes up) the different hierarchies. However class membership is not one of these. It is inherited neither up nor down the class-member hierarchy. Consider my car in *Figure BG1-12*. It is a member of the class minis, which is itself a member of the class car types. But this does not imply that my car is automatically a member of the class car types. In fact, as shown, it is not a member.

This should not be surprising. Classes capture patterns by collecting together similar objects. It is unlikely that a collection of similar classes, such as car types, would share their car type pattern with their members. For example, that my car (a member of minis) would behave like a car type.

Ban on circularity

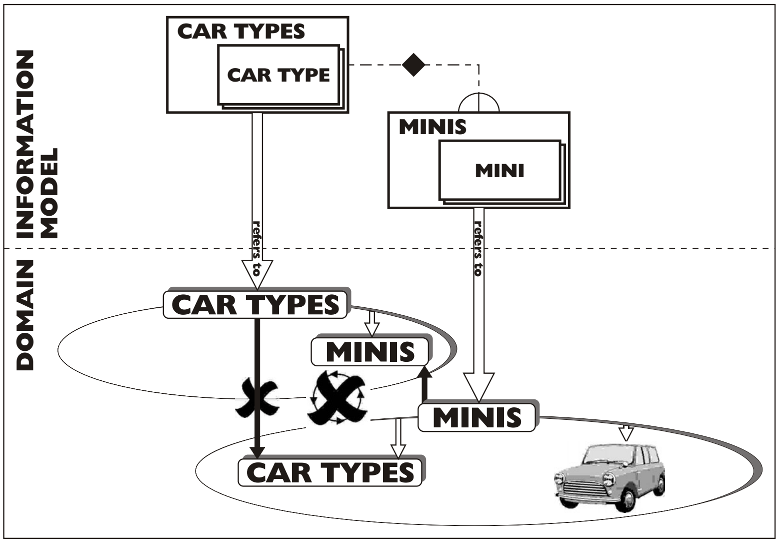
Because we construct classes from extensions, we cannot construct a class with itself as a member. Furthermore, we cannot construct a class that is a member of a class lower down the class-member hierarchy. The impossible situation is shown in *Figure BG1-13*. We recognise this impossibility in the information model. We do not allow class signs to be instances of class signs lower down the class-member sign hierarchy.

Figure BG1-13 Impossible circular class-member hierarchy



It is the nature of our understanding of space (and time and space-time) that makes this circularity impossible. This can be shown using the reference diagram in *Figure BG1-14*.

Figure BG1-14 Impossible circular class-member hierarchy reference diagram



3.3 Constructing a sign for a super–sub-class tuple

We have just looked at the signs for capturing the class–member patterns. However, this is only one of class's two main structural patterns. The super–sub-class connection is the other. We now look at the signs for this second main structural pattern. Together these two provide a framework that helps give classes their enormous power.

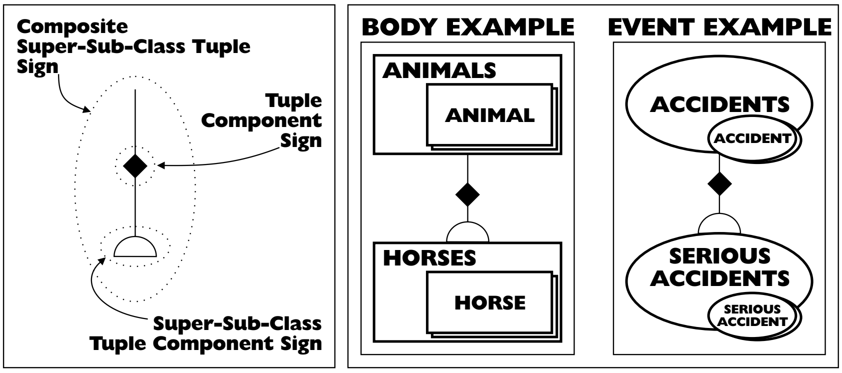
3.3.1 The super–sub-class pattern

The super–sub-class pattern resembles a whole–part pattern for classes. It is about classes containing other classes. For example, horses are animals—or, in class-speak, the class horses is a sub-class of (is contained in) the class animals. This containment or sub-class connection is between the super-class and the sub-class. Strictly speaking, it is the couple <super-class, sub-class> that belongs to the super–sub-class tuples class.

Super–sub-class sign

We model this super–sub-class pattern with a sign. It consists of a line joining the two relevant class signs with a semi-circle at the sub-class end. This is intended to look like the mathematical notation for sub-class—'É'. Because it reflects a tuple, it also has a black diamond tuple sign on the line. As we can see from *Figure BG1–15*, the same sign is used for body and event classes. There is no need for different signs because the connections have the same pattern.

Figure BG1–15 Super–sub-class tuple sign



3.3.2 Super–sub-class hierarchies

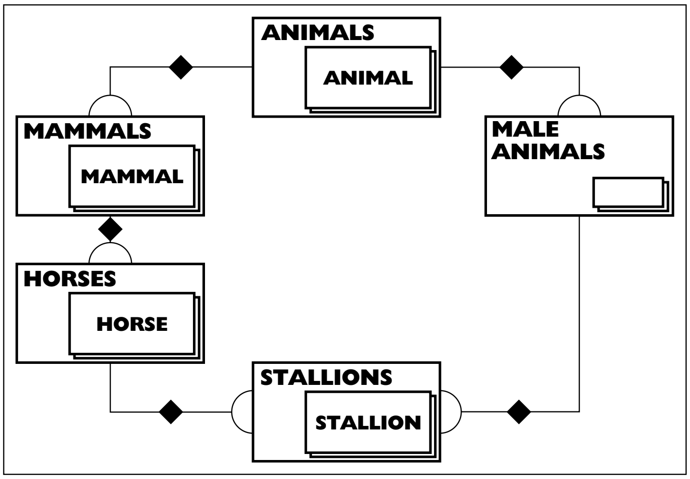
Typically, in a business model, classes are linked into a lattice hierarchy of super- and sub-classes. As we saw, when looking at the logical paradigm in *OP3—Logical Ontology Paradigm* (see OP3's *Figure OP3–23*), a tree hierarchy is too constraining to provide an undistorted reflection of reality.

Natural super–sub-class hierarchy structure

In this lattice hierarchy, a super-class may have multiple sub-classes and a sub-class may have multiple super-classes. For instance, the schema in *Figure BG1–16* models the super-class animals as having the sub-classes, mammals and male animals. It models the class stallions as having the classes horses and male animals as its super-classes.

When I construct a model of a super–sub-class hierarchy like this, I tend to automatically order the classes into a structure like the one in *Figure BG1–16*. As you can see, this follows an informal convention whereby super-classes are higher up the page than their sub-classes (though I find that in some complicated hierarchies it is not possible to do this).

Figure BG1–16 Natural super–sub-class hierarchy structure



Modelling descendant–sub-classes

The natural structure in Figure 9.16 subtly ignores the fact that the super–sub-class tuple can be inherited. The class stallions is a sub-class of the class horses and so contained in it. The class horses is a sub-class of the class mammals, which is a sub-class of the class animals. So, the class stallions is contained in the class animals. This means that we can, if we wish, recognise it as a sub-class and construct a sub-class sign in the model linking them.

Though we may need to do this for some classes, it is not a good idea to do it for all of them in a single schema. Why is this? Consider what the model for our simple example would look like if we included signs for all the possible sub-class tuples.

Figure BG1–17 illustrates the problem—the hierarchy becomes cluttered. If the super–sub-class hierarchy were larger, the problem would be worse because the number of potential sub-class tuples would increase dramatically. Modelling all these possible sub-class tuples would result in an impossibly cluttered schema.

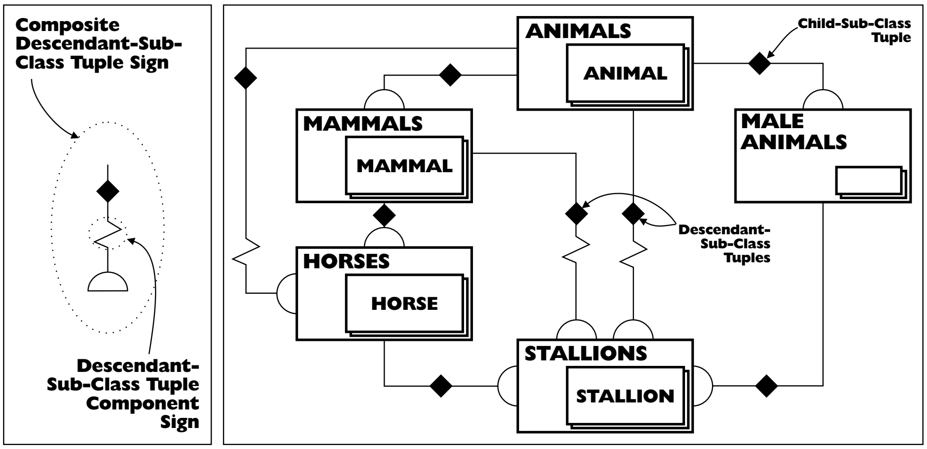

Composite Descendant-Sub-Class Tuple Sign
Descendant-Sub-Class Tuple Component Sign
ANIMALS
ANIMAL
Child-Sub-Class Tuple
MAMMALS
MAMMAL
MALE ANIMALS
Descendant-Sub-Class Tuples
HORSES
HORSE
STALLIONS
STALLION

Figure BG1–17
All possible sub-class tuples

Figure BG1–17 also, quite usefully, distinguishes between two types of sub-class tuples; child and descendant. The stallions-to-horses tuple is a child–sub-class tuple because there are no intermediate sub-classes explicitly modelled. On the other hand, the stallions-to-animals tuple is a descendant–sub-class tuple because the sub-classes, mammals and horses, are explicitly modelled as intermediate sub-classes. The sub-class sign we have been using until now is really the sign for the child–sub-class tuple. The descendant–sub-class tuple sign is a modified version of it, with an additional zigzag in its line (as shown in *Figure BG1–17*).

Deducing descendant–sub-class signs

Descendant–sub-class tuples logically depend on child–sub-class tuples, because we can 'logically' construct their signs from the signs for the child–sub-class tuples. More generally, we can logically deduce the sign for a descendant–sub-class tuple from a combination of sub-class tuples. This deduction has the following pattern:

A is a sub-class of B
B is a sub-class of C
C is a sub-class of D
<u>D is a sub-class of E</u>
Therefore:A is a descendant–sub-class of E

Where there can be any number of sub-class lines (except zero and one of course).

This is not a new logical deduction pattern. It is the same as one of Aristotle's syllogisms—one that looks like this:

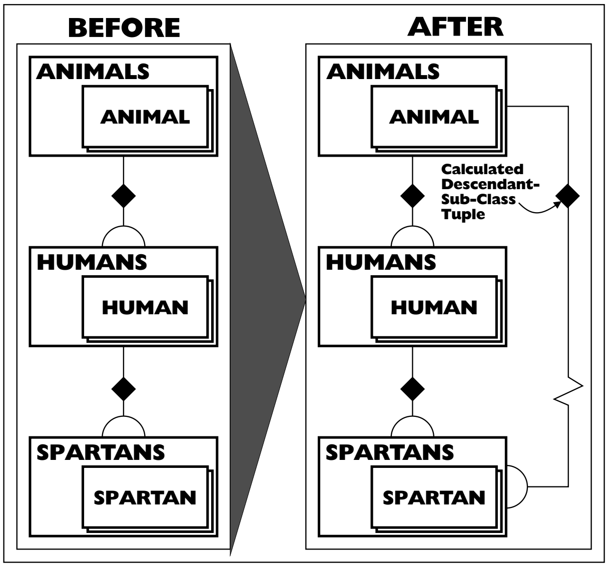
All Spartans are humans,
<u>All humans are animals,</u>
So all Spartans are animals.

It might be easier to see the resemblance when the syllogism is translated into class-speak—as below:

The class Spartans is a sub-class of the class humans,
<u>The class humans is a sub-class of the class animals,</u>
So the class Spartans is a descendant–sub-class of the class animals.

Figure BG1–18 shows this descendant calculation graphically.

Figure BG1–18 Descendant–sub-class calculations



Virtual descendant –sub-class signs

This descendant deduction pattern provides an opportunity to tidy up the sub-class clutter problem. The descendant signs can be virtual, calculated as required. As we discussed in *AS4—Focusing on the Things in the Business*, there is no reason why processes in the information system cannot represent business objects. This gives us the benefit of having signs for all the descendant–sub-class tuples without having to bear the cost of storing them—a significant compacting.

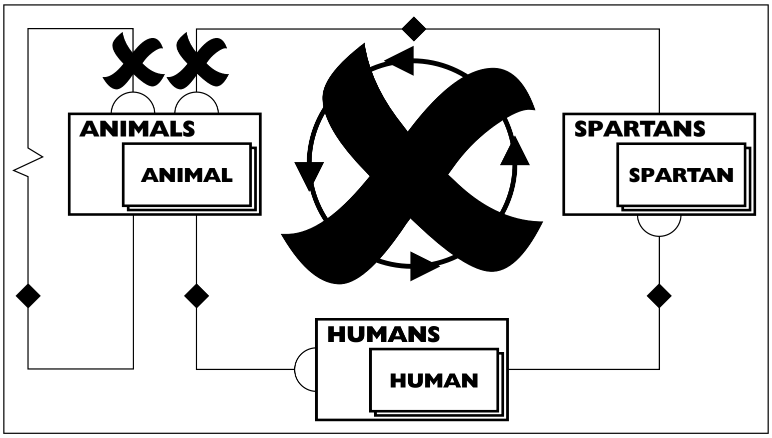
The power of computing makes this 'virtual' strategy more reliable. In a paper and ink environment, the information processor that deduces the descendant tuple signs is our minds. They are not particularly reliable processors, particularly of these sorts of logical calculations. However, in computer processing, we have a reliable logical processor. It can accurately and consistently calculate the signs.

I normally adopt a strategy of making most descendant–sub-class signs virtual. I construct views of the business model that only show the signs for child–sub-class tuples and those descendant–sub-class tuples that are essential. I make the signs for the other descendant–sub-class tuples virtual. This reduces the clutter in even the most complicated super–sub-class hierarchy to an easily manageable level.

Non-circular super–sub-class hierarchy structure

There is a logical constraint upon the super–sub-class hierarchy. Like the earlier class–member hierarchy, it cannot be circular. For example, animals from *Figure BG1–18* cannot be a sub-class of Spartans, as (falsely) indicated in *Figure BG1–19*. Because classes are built up out of extensions, it is impossible for any circularity to exist. A class, such as animals, cannot even potentially, be a sub-class of itself—in other words, contained in itself.

Figure BG1–19 Impossible circular super–sub-class hierarchy



Like the class–member hierarchy, it is the nature of our understanding of space and time (and space-time) that makes this circularity impossible. This is shown by the reference diagram in *Figure BG1–20*. Normally, we illustrate the super–sub-class structure by having one class contained in another. However, as the figure shows, this will not work for a circular structure; instead, we show the sub-class connection using an arrow.

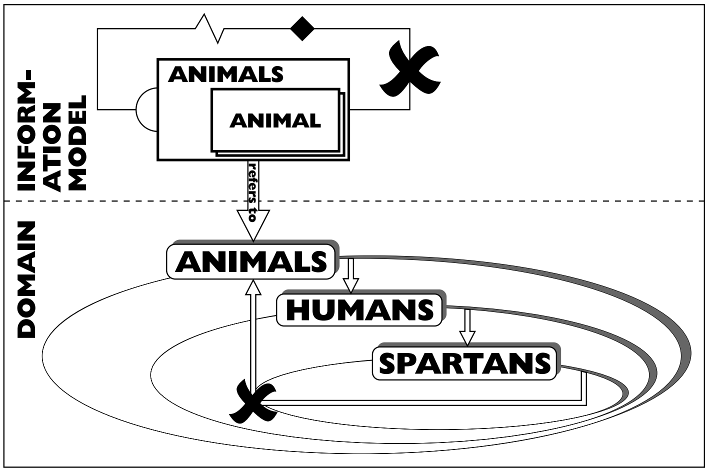


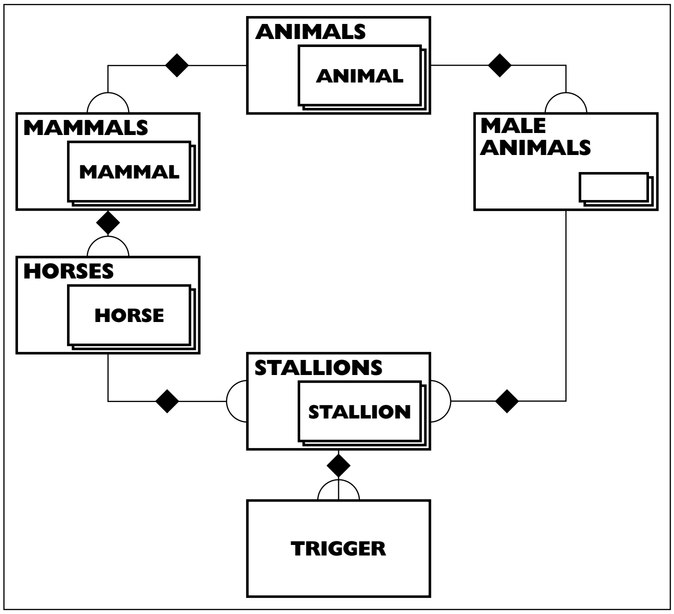
Figure BG1–20 Impossible circular super–sub-class reference diagram

Inheriting class membership

As we would expect, the patterns for classes are webby; in other words, the patterns for super–sub-class and class–member intertwine. One pattern is particularly important; it is the inheritance of class membership up the super–sub-class hierarchy.

To see how this works, we introduce Trigger the horse into the model in *Figure BG1–21*. We naturally tend to make him a member of the class stallions (shown in *Figure BG1–21*). Stallions is the hierarchy's lowest class. However, Trigger is potentially a member of all the hierarchy's higher classes, but our natural instinct is not to model these possibilities.

Figure BG1–21 Natural position for Trigger the horse



Why don't we model these potential higher class–member tuples? We had a similar situation to this earlier with child– and descendant–sub-classes. And the answer is the same here—they would clutter up the schema and the model. We can see this in *Figure BG1–22*, which shows the results of constructing all the class–member tuples for our example. The model is pretty cluttered and this is only a small hierarchy. A much larger hierarchy would be impossibly cluttered. We have a class–member, as well as a sub-class, clutter problem.

Figure BG1–22
All Trigger the horse's member possible class–member tuples

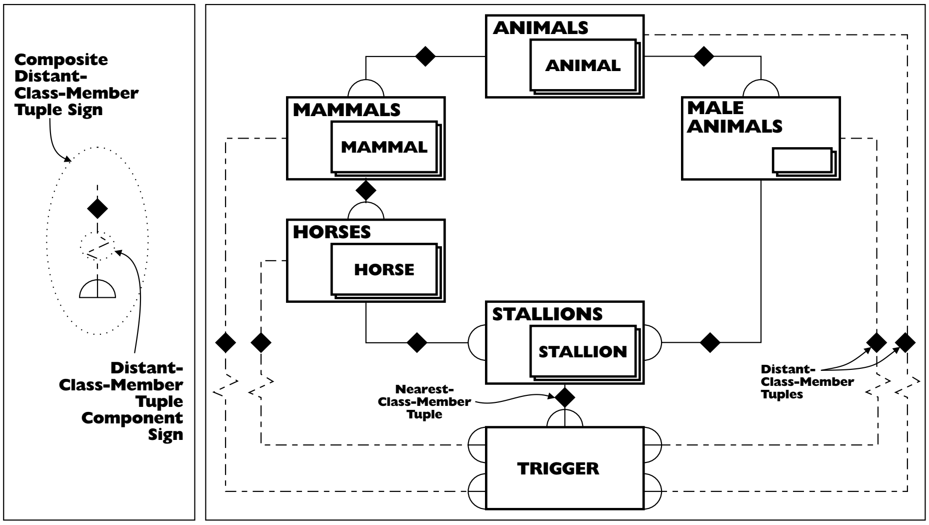

Composite Distant-Class-Member Tuple Sign
Distant-Class-Member Tuple Component Sign
ANIMALS
ANIMAL
MAMMALS
MAMMAL
MALE ANIMALS
HORSES
HORSE
STALLIONS
STALLION
Nearest-Class-Member Tuple
Distant-Class-Member Tuples
TRIGGER

Figure BG1–22 also illustrates a distinction between the sign for the lowest class–member tuple—now called the nearest-class–member sign—and the sign for other class–member tuples—now called distant-class–member signs. The distant-class–member signs use the same zigzag component sign as the earlier descendant–sub-class signs. Trigger's nearest class is stallions because there is no class below stallions in the super–sub-class hierarchy to which he belongs; so, the class–member tuple is a nearest-class–member tuple. Trigger is a distant-class–member of each of the classes horses, mammals, male animals and animals because there is a class below them in the class–member hierarchy, the class stallions, of which he is a member.

Deducing more distant-class-member signs

As with child–sub-class signs, we can deduce and construct distant-class–member signs from the nearest-class–member sign. Like before, this is done logically, without involving any analysis of what the signs refer to. The converse is, of course, not true. We cannot work out a nearer class–member sign from a more distant sign. This makes the nearest-class–member sign key; from it we can calculate all the distant-class–member signs.

More generally we can construct a distant-class–member sign from the class–member sign and a chain of super–sub-class signs up from its class sign. The deduction has the following pattern;

> A is a class–member of B
> B is a sub-class of C
> C is a sub-class of D
> <u>D is a sub-class of E</u>
> Therefore:A is a more distant-class–member of E

There can be any number of sub-class lines (except zero of course) in this calculation.

As with the earlier descendant–sub-class calculation pattern, this has the same pattern as one Aristotle's syllogisms (called barbara), which looks like this:

> Socrates is a man,
> <u>All men are mortal,</u>
> So Socrates is mortal.

It is easier to see the resemblance when it is translated into class-speak—as below:

> Socrates is a class–member of the class men,
> <u>The class men is a child–sub-class of the class mortals,</u>
> So Socrates is a more distant-class–member of the class mortals.

The distant calculation for Aristotle's syllogism is shown in *Figure BG1–23*.

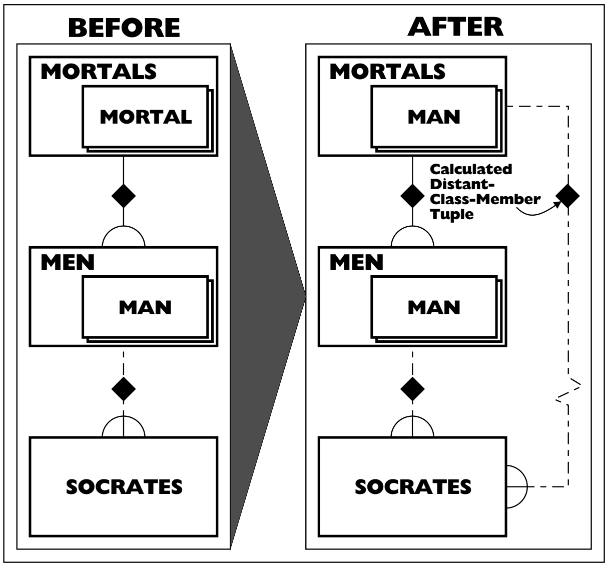


Figure BG1–23 Aristotle's barbara syllogism

People who have not yet developed a clear idea of the difference between the class–member and super–sub-class patterns often see this distant-class–member calculation process as the same as the earlier descendant–sub-class calculation process. When they develop a clear understanding of the differences between the two patterns, they then begin to see the differences between, as well as the similarities in, the two processes.

Compact class–member hierarchy models

The distant-class–member deduction pattern works in a similar way to the earlier descendant–sub-class pattern. This provides us with an opportunity to use virtual signs again and tidy up the class–member clutter problem. An opportunity to get the benefit of having signs for all the descendant–sub-class tuples, without having to bear the cost of visibly recording them. Once I model the nearest-class–member tuple, I can assume that all the distant-class–member tuples also 'virtually' exist.

I can then adopt the strategy of only modelling the nearest-class–member tuples and essential distant-class–member tuples. The signs for the many other distant-class–member tuples are virtual. This can reduce the clutter in even the most complicated class–member hierarchy to an easily manageable level.

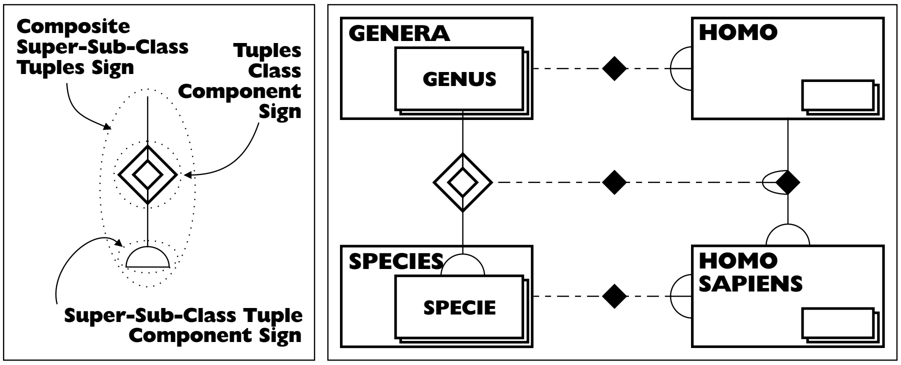
3.3.3 Super–sub-class tuples class

We naturally see the super–sub-class tuple as connecting classes. This is correct in one sense, the tuple can only connect classes. But, it does not mean the super–sub-class sign only connects class signs. If a class sign refers to a class of classes, then its member component refers to the member classes. Because these are classes they can be connected with their sub-class signs using the super–sub-class sign. Furthermore, because the member component sign refers to a class of members, the super–sub-class sign refers to a class of super–sub-class tuples.

Here is an example. Consider the Linnaean biological scheme used to classify individuals into species and species into genera (singular—genus). This two-level structure is reflected in the Linnaean names for species. For instance, our species is *homo sapiens* where *homo* is the genus and *sapiens* the species within genus. This gives us an example of a super–sub-class tuple class between classes' members.

At the classes of classes level we have two classes; genera and species. The class genera has individual genus classes, such as *homo* as members. The class species has individual specie classes, such as *homo sapiens* as members. At the classes of individual objects level, we also have two classes; *homo* and *homo sapiens*. The class *homo sapiens* is a sub-class of the class *homo* as shown in *Figure BG1–24*. This particular super–sub-class pattern is just a particular example of a more general pattern. The members of the class species are sub-classes of the members of the class genera. This is a super–sub-class tuples class between the classes' members. Because it refers to all the different individual tuples between the various members, it is a class of tuples not an individual tuple. This is reflected in its sign, which uses a tuples icon instead of a tuple icon (shown in *Figure BG1–24*).

Figure BG1–24 The super–sub-class tuples class sign


Composite Super-Sub-Class Tuples Sign
Tuples Class Component Sign
Super-Sub-Class Tuple Component Sign
GENERA
GENUS
HOMO
SPECIES
SPECIE
HOMO SAPIENS

4 Constructing signs for tuples

We have finished looking at the notation for classes, an object with internal structure resulting from its construction from other objects. We now look at another constructed object with internal structure, the tuple object.

4.1 Constructing a tuple of individual objects and a tuples class

For our purposes, tuples exist with an associated tuples class. So we model the sentence 'Prince Charles is the father of Prince William' with a tuple and a tuples class (shown in *Figure BG1–25*).

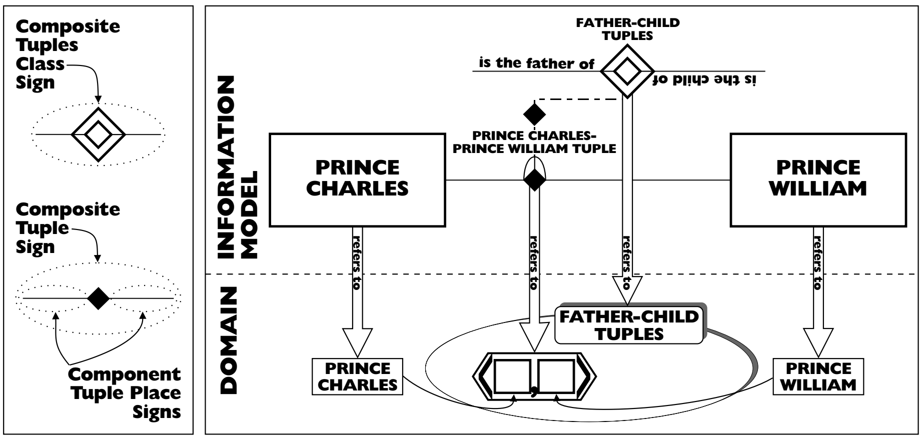


Figure BG1–25 Tuple and tuple class signs

Note these points:

- The solid black diamond is the component sign for the tuple.
- The connecting lines from the tuple component sign to other signs are called tuple place component signs.
- The tuple place component signs connect the sign for the tuple with the signs for the objects out of which it is constructed. These are called the tuple place objects. In the Prince Charles—Prince William tuple, the places are occupied by individual objects, but they could be occupied by any type of object.
- The component sign for a tuples class is two hollow diamonds, one inside the other.
- The lines from the component tuples class sign are called the (tuples) class place component signs.
- The father–child tuples class is a class object and so uses the standard class–member sign to link to its member tuple sign.

4.1.1 Occupied class place signs

A (tuples) class place is said to be occupied when its tuples class sign is connected to another object. For instance, the fathers class is connected to the father–child tuples class in Figure BG1–26. This occupation is signed by adding the component tuple sign, a solid black diamond, to the (tuples) class place component sign. The object to which the tuples class is connected is called a (tuples) class place object, an example is the fathers class in *Figure BG1–26*. Notice that the 'is a child of' class place sign in *Figure BG1–26* does not have a black diamond component because it is not occupied.

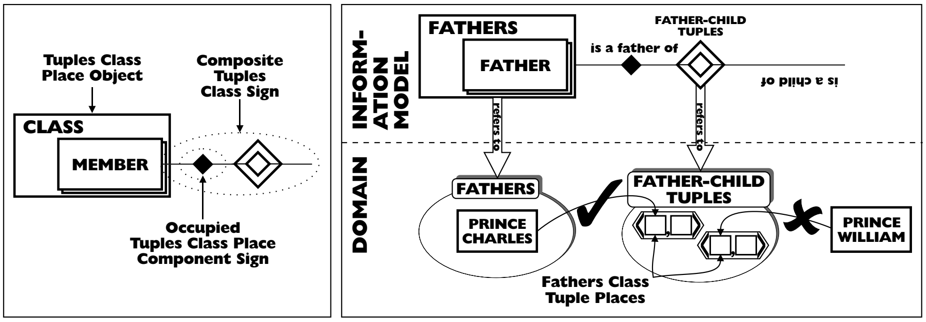


Figure BG1–26 Class place constraints on tuple places

4.1.2 Occupied class place constraints on tuple places

A class place object constrains which tuples can be members of its tuples class. In the example in *Figure BG1–26*, the fathers class is a place object, which implies the existence of a fathers tuple place in the tuple members of the tuples class. In simpler language, this means that one of the places of each tuple member of the tuples class is designated a father tuple place and must be occupied by a member of the father's class.

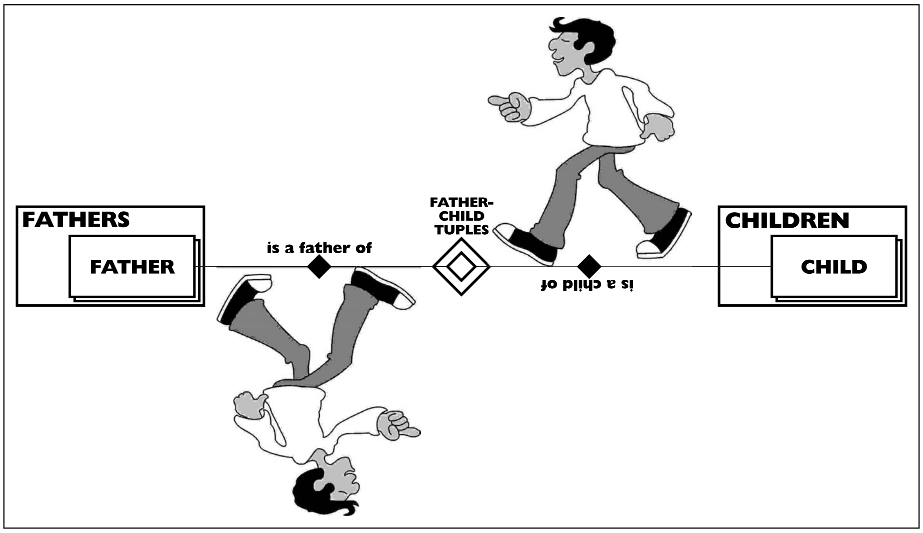
In the example illustrated in *Figure BG1–26*, the first place in the couple is designated the father couple place. This means that a couple with Prince Charles in its first place (<Prince Charles, ?>) can be a member of the father-child tuples class because Prince Charles is a member of the class fathers. But any couple with the format <Prince William, ?> cannot be a member, because Prince William is not a member of the fathers class.

4.1.3 Tuple and tuples class names

Tuples classes have names in the same way as other classes. However, in addition, both tuples and tuples classes have a name constructed from the names on their class place signs. The convention for constructing these names is that one of the class place signs is picked and then a mental walk is taken along the class place (or place) sign to the tuples class (or tuple) sign reading the text on the left and then along to the next class place (or place) sign.

In the example in *Figure BG1–27*, there are two 'walks'. We can start at the father member sign, and mentally walk past the father–child tuples to the child member sign, reading the 'is the father of' text on the left. This gives us the name 'father—is the father of—child'. Mentally walking the other way, from child to father, would give us the name 'child—is child of—father'.

Figure BG1–27 Convention for reading tuple names



4.2 Tuples classes inheriting patterns from classes

We can now begin to take advantage of the power that re-using patterns brings. We can re-use the class pattern on tuples classes. As we have just seen, they are classes—in class-speak; they are a sub-class of the class classes. So they inherit all the characteristics of a class and share all its patterns. For instance:

- They have tuple super–sub-class and tuple class–member hierarchies.
- They have child- and descendant–sub-tuples-classes.

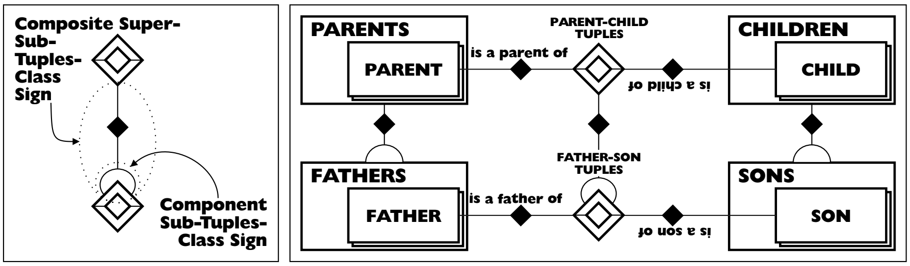
- They have nearest– and distant– class–member tuples.

They will also automatically inherit any new class patterns we construct (for example, the distinct and overlapping patterns we examine in the next paper, *BG2— Constructing Signs for Business Objects' Patterns*). Tuples classes inherit all this as the result of being a class object. We now look at an example of a class pattern being re-used for tuples classes, the tuple super–sub-class hierarchies.

4.2.1 Tuple super–sub-class hierarchies

As tuples classes are classes they can also have super– and sub-classes. For instance, parent–child tuples is a super-class of father–son tuples (shown in *Figure BG1–28*). Notice that the super–sub-class tuple uses the standard super–sub-class tuple sign.

Figure BG1–28 Super–sub-tuple-class sign



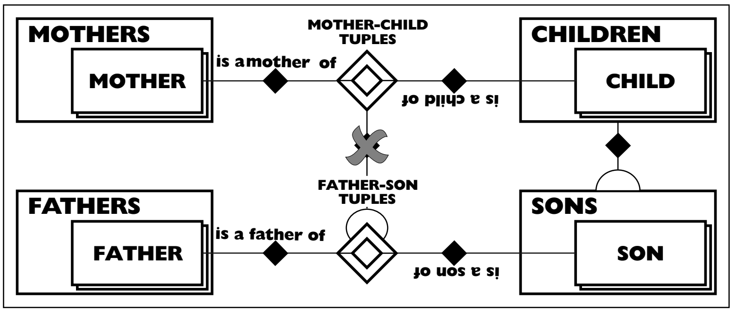
Modelling super-sub place classes

Care needs to be exercised when working out the super–sub-class tuples between the place classes of tuple super- and sub-classes. This tends to come with practice. In particular, as one moves from a tuple subclass to a tuple super-cools, the place classes should either remain the same or move up to a super-class. This is easiest to explain with an example.

Look at *Figure BG1–28*, the father–son tuples class has as one of its class place objects, the fathers class. Its super-class, parent–child tuples, has as its corresponding class place object, the parents class. As parents is a super-class of fathers, we can go in a full circle. Starting from father–son tuples we go along to fathers, up to parents, along to parent–child tuples and back down to where we started, father–son tuples. This works because place classes of a tuples super-class need to be either the same as or super-classes of the corresponding place classes of their tuples sub-class.

For an example of incorrect modelling, look at *Figure BG1-29*. We cannot trace a full circle here because mothers is (rightly) not signed as a super-class of fathers. The problem here is that mother-child tuples has been signed incorrectly as a super-class of father-son tuples.

Figure BG1-29 Incorrect super-sub-class tuple



5 Constructing signs for whole-part tuples

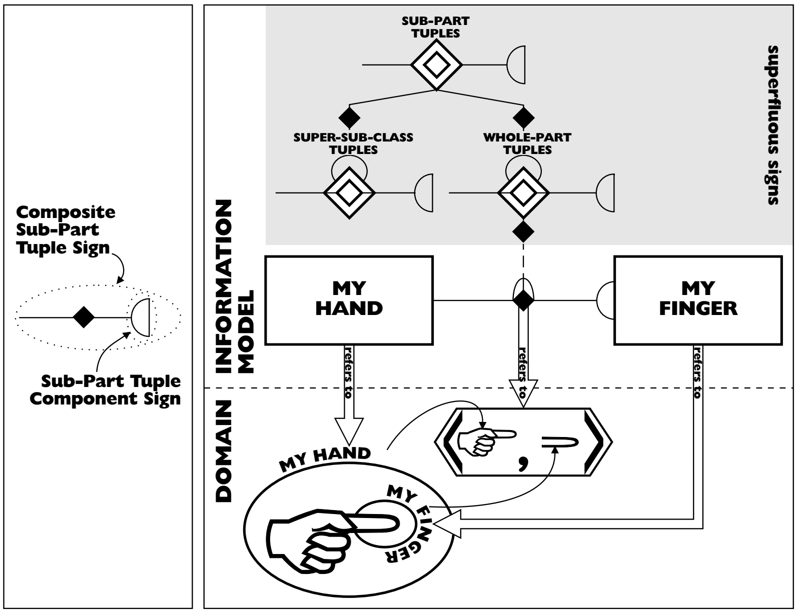
In Part Four, we noted how important whole-part tuples were to object semantics. This is recognised in the notation by giving whole-part tuples their own sign. We look at it now, along with the patterns of the underlying whole-part tuples that it is used to sign. We noted, in *OP4—Business Object Ontology Paradigm*, that the whole-part pattern is similar to the super-sub-class pattern. Here we see more evidence of this.

5.1 What are whole-part tuples?

But first let us remind ourselves how whole-part tuples fit into the class framework. Consider this example. My fingers are part of my hand. This means that there is a connection between my fingers and my hand. Using the same analysis as we used for general tuples above, this is a couple object <my fingers, my hand>, which is a member of the whole-part tuples class. This analysis is shown in *Figure*

BG1-30. Because the whole-part couple has its own sign, the whole-part tuples class sign and its class-member sign are redundant. They are only included in this model to make absolutely clear what the whole-part tuple is.

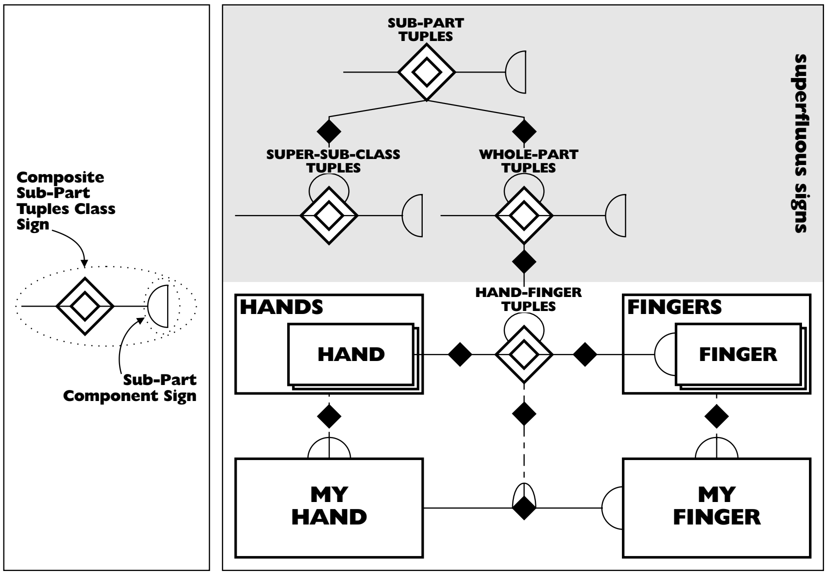
Figure BG1-30
My fingers are part of my hand



The composite whole-part sign is constructed from familiar components. Because the couple is a tuple, we use a tuple sign for it. As we mentioned above, the whole-part and the super-sub-class tuples are similar kinds of tuples, operating at different levels. So they have the same composite sign. Until now, we have called this the sub-class sign, but as we are generalising it across sub-class and whole-part, we rename it the sub-part sign.

The example above is of particular individuals. There are also classes whose members have whole-part patterns. We can extend the example to illustrate this. Fingers are parts of hands—in class-speak; the individual members of the fingers class are parts of the individual members of the hands class. How do we sign this whole-part tuple between members of a class? We use the individual whole-part sign, but suitably amended to show that it is the sign of a tuples class instead of a tuple – as shown in *Figure BG1-31*). Like the tuple level sign, the whole-part tuples class sign and its class-member sign are superfluous because the hand-finger tuples is signed as a whole-part tuples class.

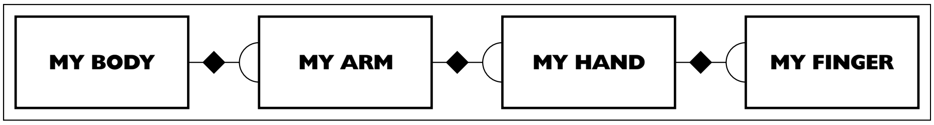
Figure BG1-31
Fingers are part of hands



5.2 Individuals whole-part tuple hierarchy

The individual whole-part tuples create an individual whole-part hierarchy. For instance, my fingers are part of my hand, my hand is part of my arm, and my arm is part of my body (shown in *Figure BG1-32*). As we can see, this is, in many ways, a super-sub-class hierarchy for individual objects.

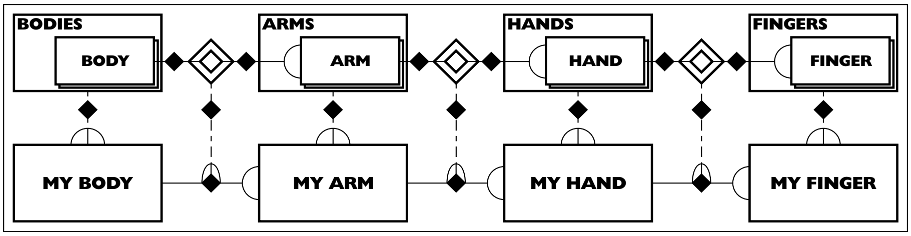
Figure BG1-32
Individual whole-part tuple hierarchy



5.3 Classes whole-part tuple hierarchy

Individual whole-part hierarchies can be generalised into whole-part tuples class hierarchies. For instance, the individual whole-part hierarchy shown in *Figure BG1-32* can be generalised to the class level (shown in *Figure BG1-33*.)

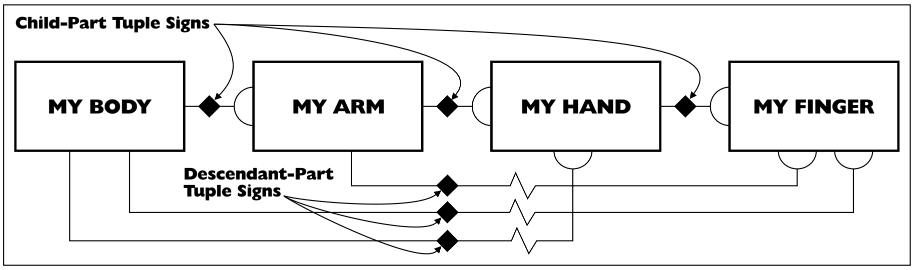
Figure BG1-33 Whole-part tuple class hierarchy



5.4 Child- and descendant-parts

Just as we drew a distinction between child- and descendant-sub-classes in the super-sub-class hierarchy, we draw a corresponding distinction here between child- and descendant-parts. To see this, we add all the potential whole-part tuples to the model in *Figure BG1-32* (see the result shown in *Figure BG1-34*).

Figure BG1-34 Child- and descendant-part tuples



A child-part is one that has no intervening parts (in the particular model being considered). Descendant-part tuples are ones with intervening parts. For example, in *Figure BG1-34*, 'my fingers are part of my hand' is a child-part tuple. Whereas, as 'my finger is part of my arm' is a descendant-part tuple because it has my hand as an intervening part.

5.5 Deducing descendant-part signs

Like descendant-sub-class tuples, a descendant-part tuples sign can be deduced from the child-part tuple signs and more generally from whole-part tuple signs. This deduction has the same pattern as the descendant-sub-class deduction:

A is a whole-part of B
B is a whole-part of C
C is a whole-part of D
D is a whole-part of E
Therefore:A is a descendant-part of E

Where there can be any number of whole-part lines (except zero and one of course). An actual example is:

My finger is a part of my hand, and
My hand is a part of my arm
ThenMy finger is a (descendant-) part of my arm.

Figure BG1-35 shows this deduction graphically.

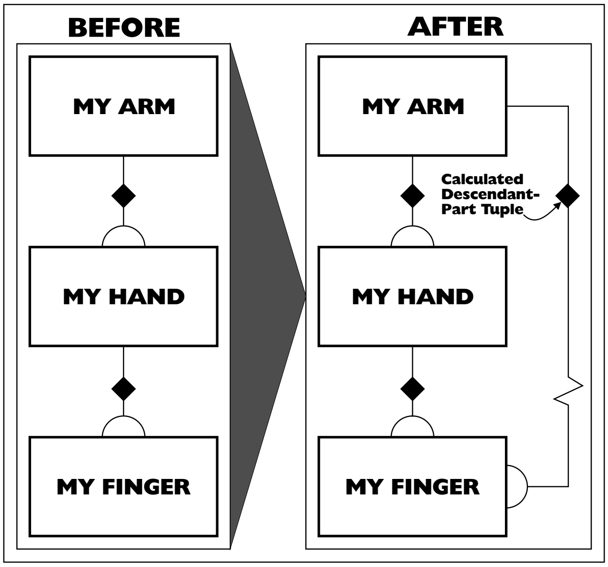


Figure BG1-35 Descendant-part tuple deduction

6 Constructing signs for dynamic objects

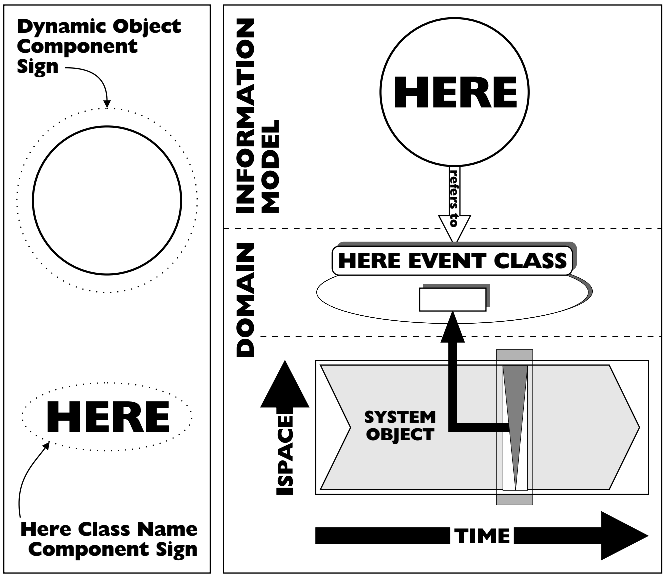
So far we have been constructing signs for timeless objects. We now look at signs for the time-bound dynamic objects described in *OP4—Business Object Ontology Paradigm*:

- The here event class,
- The now event class, and
- Current couples.

6.1 Constructing a sign for the 'here' event class

The here event class has as its single member an instantaneous time-slice of the system object—a physical body. This member moves, like all the dynamic events, along the time dimension with the system's 'consciousness'. The composite sign for the here event class is a circle, the component sign for a dynamic object, with the name 'HERE' in it (shown in *Figure BG1–36*).

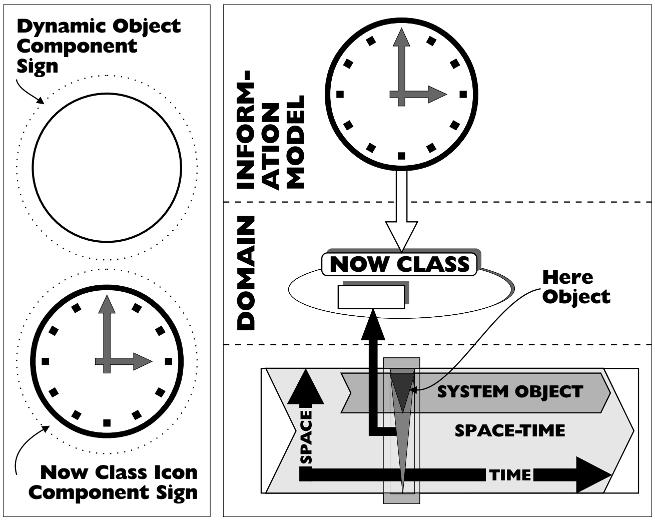
Figure BG1–36 Sign for the 'here' event class



6.2 Constructing a sign for the 'now' event class

The 'now' event class has as its single member the instant that contains the 'here' event class's member. It is signed using a circle containing a clock face – as shown in *Figure BG1–37*.

Figure BG1–37 Sign for the 'now' event class

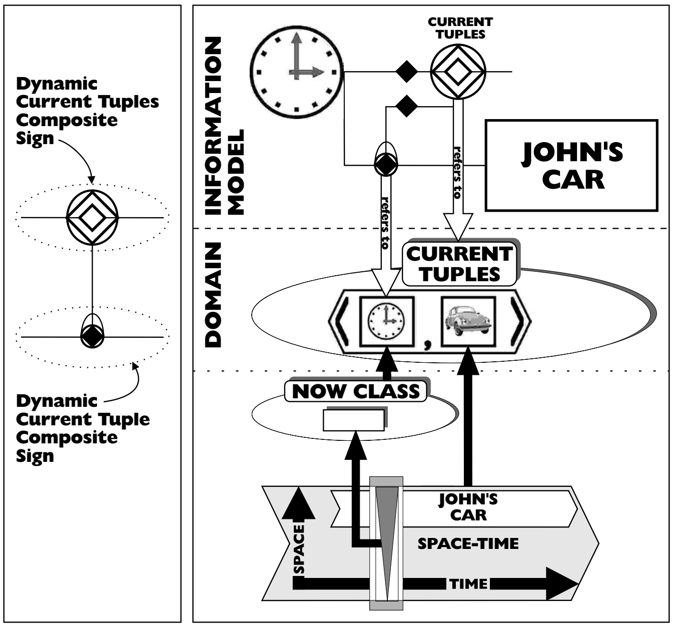


6.3 Constructing a sign for a current tuple

We now construct the signs for the current tuples class and its members, current tuples. However, the current tuples class sign is, to an extent, superfluous because any tuple signed as current automatically belongs to the current tuples class. This is done using a component dynamic circle sign (illustrated in *Figure BG1–38*). As you can see, one of the sign's current tuple places is linked to the now event class, the other(s) to the object(s) currently classified as current.

You can also see the current tuples class signed in *Figure BG1–38* as a tuples class with the dynamic circle component sign around it.

Figure BG1–38
Sign for a current tuple



7 Signs as objects—modelling the model

Object semantics applies to signs in the model as well as the objects that are modelled. According to object semantics, everything is an object with four-dimensional extension. Even the signs in the model are objects—they are model objects. We can see this clearly when we start modelling the model. This is not meta-modelling, this is more like modelling x modelling or (modelling)^2.

This (modelling)^2 clarifies one aspect of modelling that people sometimes find confusing. This is that the type of an object (for example, body or event), and the type of the model object that models it, are often quite different. This confusion about types sometimes manifests itself as a belief that the distinction between data and process in information systems (the signs in the model) reflects the distinction between objects and events in the real world. This resolves itself into a belief that data reflects objects and process reflects events. We discussed how mistaken this belief is in *AS4—Focusing on the Things in the Business*.

7.1 A (modelling)^2 model

Look at the (modelling)^2 model in *Figure BG1–39*. It models examples of the four major types of signs in our object notation; the individual body and event objects and the bodies and events classes. As the model shows, all these signs (model objects), are all individual physical bodies, whatever they refer to—whether event, class or body.

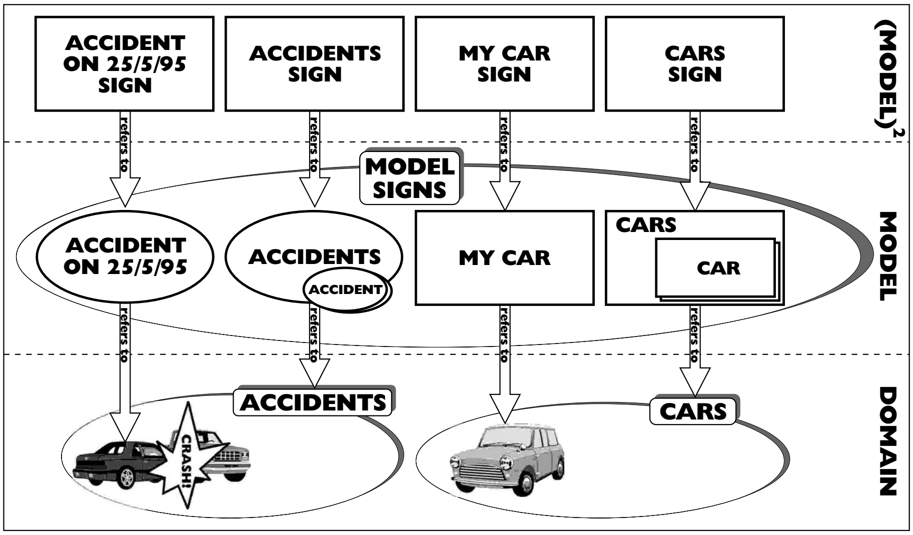


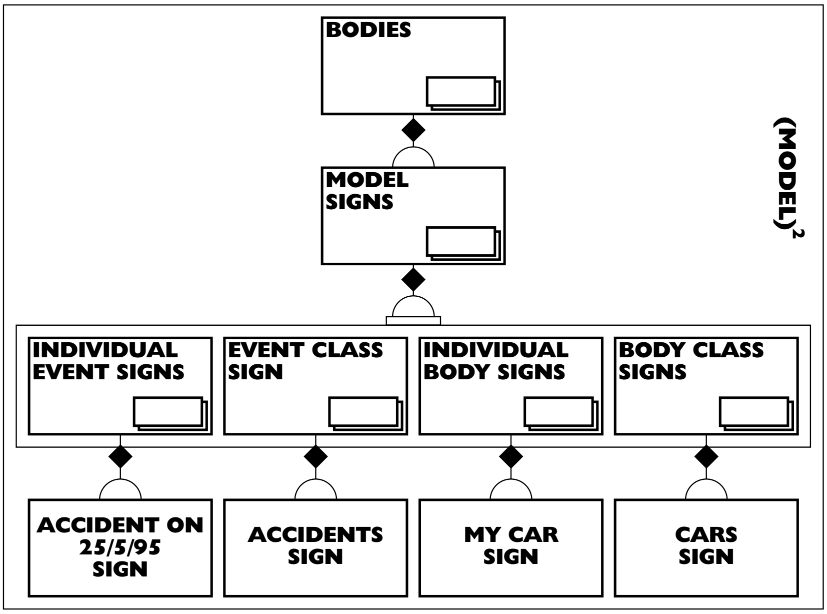
Figure BG1–39
Modelling body and event model objects

The model object for my car is an individual body sign. This sign is, like the body object it refers to, an individual body object in its own right. It has extension, it persists through time—though maybe not for as long as the body object it refers to. Individual body signs are the only type of model object where the model object and the object it refers to are of the same type.

The individual event sign is, like the individual body sign, an individual body. It has extension and it persists through time—it has spatial and temporal dimensions. However, the event model object, unlike the individual body object, is not of the same type as the object it refers to. The ‘accident 25/5/95’ sign is a body object with temporal extension; the accident it refers to is an event that does not persist through time.

Model class objects, like the model individual objects, are individual bodies and so different in type from the objects they refer to. The event and body class examples in *Figure BG1-39* illustrate how constructed objects with an internal structure, such as the two class objects, are flattened out in the object model into individual physical body objects. This (model)^2 structure is illustrated in *Figure BG1-40*.

Figure BG1-40 (Model)^2 objects



8 What's next

We have now looked at signs for all the major types of individual objects that we need to business object model. We have got a feel for what they mean and how they work. We are well on our way to being ready to start business modelling. In the following paper (*BG2— Constructing Signs for Business Objects' Patterns*), we look at the syntax of business object patterns. We see how we can use the object notation to model patterns of business objects.

BORO Working Papers - Bibliography

The BORO Working Papers

<u>Volume A</u>
<u>A—The BORO Approach</u>

Book AS
AS—The BORO Approach: Strategy

AS1—*An Overview of the Strategy*
AS2—*Using Objects to Reflect the Business Accurately*
AS3—*What and How we Re-engineer*
AS4—*Focusing on the Things in the Business*

<u>Volume - O</u>
<u>O—ONTOLOGY Papers</u>

Book - OP
OP—Ontology: Paradigms

OP1—*Entity Ontology Paradigm*
OP2—*Substance Ontology Paradigm*
OP3—*Logical Ontology Paradigm*
OP4—*Business Object Ontology Paradigm*

<u>Volume - B</u>
<u>B—Business Ontology</u>

Book - BO
BO—Business Ontology: Overview

BO1—*Business Ontology - Some Core Concepts*

Book - BG
BG—Business Ontology: Graphical Notation
Constructing Signs for Business Objects

Graphical Notation I
BG1— *Constructing Signs for Business Objects*
Graphical Notation II
BG2— *Constructing Signs for Business Objects' Patterns*

<u>Volume - M</u>
<u>M—The BORO Re-Engineering Methodology</u>

Book - M0
M0—The BORO Re-Engineering Methodology: Overview

M01—*The BORO Approach to Re-Engineering Ontologies*

Book - MW
MW—The BORO Methodology: Worked Examples

Worked Example 1
MW1—*Re-Engineering Country*
Worked Example 2
MW2—*Re-Engineering Region*
Worked Example 3
MW3— *Re-Engineering Bank Address*
Worked Example 4
MW4—*Re-Engineering Time*

Book - MA
MA—The BORO Re-Engineering Methodology: Applications

MA1—*Starting a Re-Engineering Project*
MA2—*Using Business Objects to Re-engineer the Business*

Book - MC
MC—The BORO Re-Engineering Methodology: Case Histories

Case History 1
MC1—*What is Pump Facility PF101?*

BUSINESS ONTOLOGY: GRAPHICAL NOTATION - 1

CONSTRUCTING SIGNS FOR BUSINESS OBJECTS

Symbols–H