Business Object Reference Ontology Program

simplifying semantics

# Working Paper

## BG2

# Business Ontology: Graphical Notation-2

# Constructing Signs for Business Objects' Patterns

Issue: Version - 4.01 - 01-July-2001

Contact

For queries regarding this document, or the BORO Program in general, please use the following email address:

*contact@BOROProgram.org*


# BG2

## Business Ontology: Graphical Notation - 2

# Constructing Signs for Business Objects' Patterns

## CONTENTS

# CONTENTS
## BG2

BORO

# BG2

## Business Ontology: Graphical Notation - 2

# Constructing Signs for Business Objects' Patterns

## 1 Introduction

The working paper *BG1— Constructing Signs for Business Objects* describes how to construct signs for the basic types of objects in object semantics. In this paper we move up a level. Instead of looking at individual signs, we look at the syntax of signs that describe patterns of business objects. We examine how this syntax works using the following examples of fundamental patterns found in our investigations of object semantics in *OP4—Business Object Ontology Paradigm*:

- Patterns for the connections between extensions,
- State hierarchy patterns,
- Time ordering patterns,
- Cardinality patterns for tuples classes, and
- Patterns for compacting classes.

## 2 Patterns for the connections between extensions

Extension is a central notion of logical and object semantics. Many of the patterns we have analysed so far turn out to have structures based on it. For

instance, the sub-part tuple (the generalised whole–part and super–sub-class tuple) is based on the extension of one of the related objects containing the other.

Closely related to sub-part tuples are two other patterns based on structural connections between extensions: the distinct and overlapping patterns. These two patterns occur at two levels:

- The individual object level, and
- The class level.

This is similar to the sub-part pattern, which is the whole–part pattern at the individual object level and the super–sub-class pattern at the class level. Let's now investigate these patterns, starting at the individual object level.

## 2.1 Individual object level patterns

At the individual object level, any number of individual objects can have the distinct or overlapping pattern, but the pattern is at its simplest when only two objects are involved. So we start by looking at pairs of distinct and overlapping objects then move onto larger groups of objects. Finally, we examine the following associated patterns:

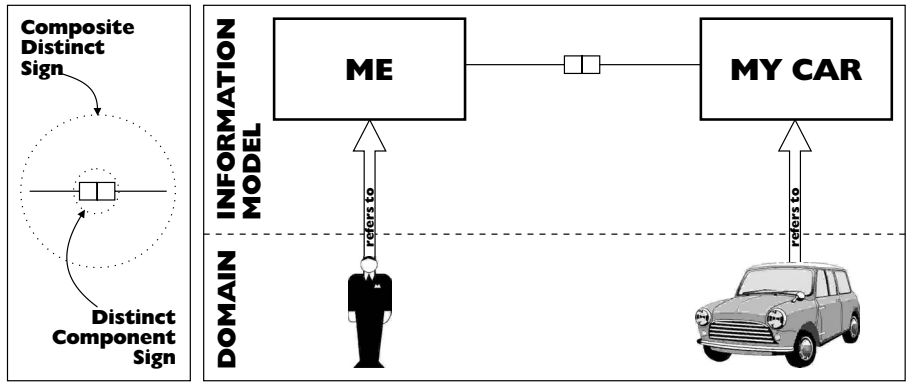
- Inheriting distinct and overlapping patterns,
- Known and unknown distinct and overlapping individual objects,
- Partition patterns for distinct individual objects,
- Intersection pattern for overlapping individual objects, and
- Fusion pattern for individual objects.

We also work out what objects the signs for distinct and overlapping individual patterns refer to.

### 2.1.1 Distinct pairs of individual objects

Two individual objects that do not have any spatio-temporal parts in common are distinct. For example, my car and me are distinct—no part of my car is also a part of me. We model this distinct pattern with the sign shown in *Figure BG2–1*.

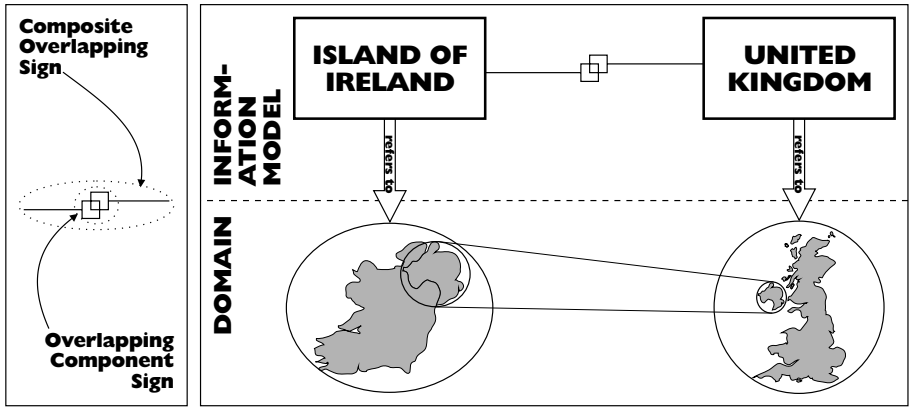
Figure BG2–1 Distinct individual objects sign




### 2.1.2 Overlapping pairs of individual objects

A pair of individual objects that have parts in common overlap. For example, the island of Ireland and the country United Kingdom overlap; the country of Northern Ireland is a part of both individual objects. We model this overlapping pattern with the sign shown in *Figure BG2–2*. (This and subsequent examples involving countries use our simple intuitive view of country objects. We re-engineer a more sophisticated view in *MW—The BORO Methodology: Worked Examples*.)

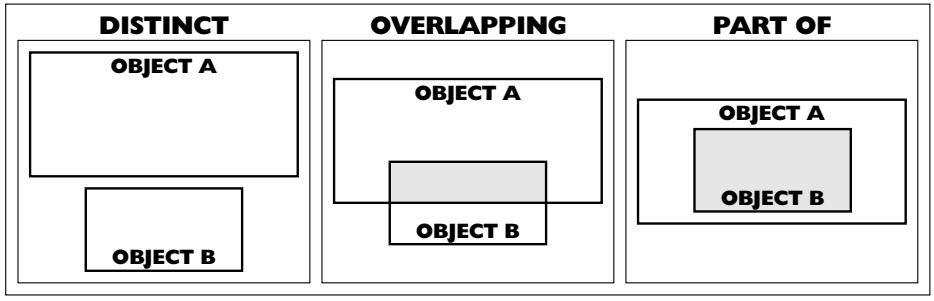
Figure BG2-2
Overlapping individual objects sign




### 2.1.3 Three main types of connection for pairs of individual objects

We have now looked at what are, from an extension point of view, the three main patterns of connection between pairs of individual objects; distinct, overlapping and whole-part. As illustrated by *Figure BG2-3*, a pair of individual objects must fall under one of these patterns. It could be argued that the whole-part pattern, where one individual object completely contains another, is an extreme case of overlapping. However, the convention is to consider these as separate patterns with their own signs.

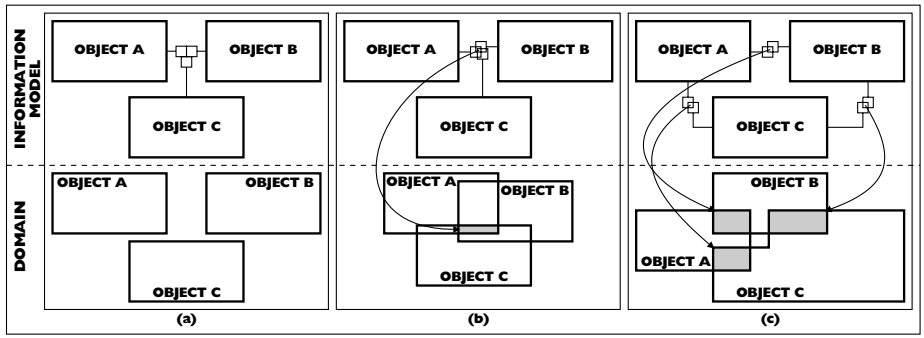
Figure BG2-3
Pattern for individual objects



### 2.1.4 Larger groups of individual objects

Groups of individual objects larger than two can have a variety of patterns of connection. All the individual objects can be distinct [as shown schematically in *Figure BG2-4* (a)]. Or they can all overlap [shown in *Figure BG2-4* (b)]. It is also possible that some will be distinct and others will overlap. Furthermore, it is possible that even if every pair in a group of individual objects overlap, the whole group will not overlap [shown schematically in *Figure BG2-4* (c)]. The same is not true for distinctness; if every pair is distinct, then the whole group is distinct.

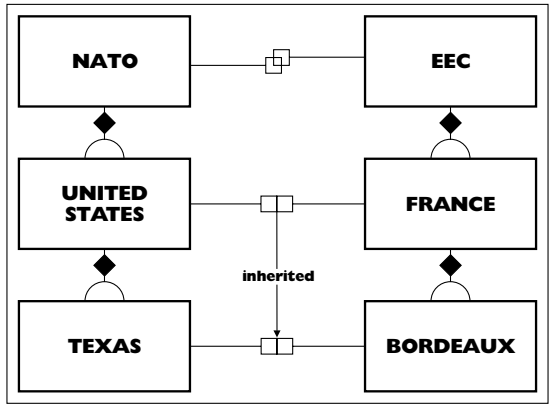
Figure BG2-4 Schemas for larger numbers of individual objects




### 2.1.5 Inheriting distinct and overlapping patterns

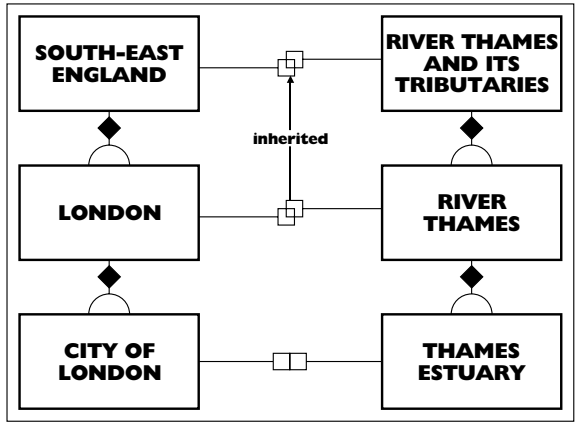
Distinct and overlapping patterns for individual objects are inherited in opposite directions along the whole-part hierarchy. Distinctness is inherited down the hierarchy. So, as the United States and France are distinct, their parts—for example, Texas and Bordeaux—are also distinct. This is modelled in *Figure BG2-5*. The model also shows NATO and the EEC (which have the United States and France as parts) overlapping, proving that distinctness is not inherited up the whole-part hierarchy.

Figure BG2-5
Inheriting distinctness




Overlapping is inherited up the whole–part hierarchy. So, as London and the River Thames overlap, any wholes of which they are parts also overlap. For instance, South-East England and the River Thames and its tributaries overlap. Overlapping, however, is not inherited down the hierarchy (illustrated by the distinct City of London and Thames Estuary in the model in *Figure BG2-6*).

Figure BG2-6
Inheriting overlapping




This inheritance has implications for how we model. I have found it useful to push the distinct connections as far up the whole–part hierarchy as they will go and the overlapping connections as far down the hierarchy as they will go. This increases the number of objects that can inherit the pattern and so automatically increases the functionality of the model. It also compacts the model as it


replaces a number of lower-level distinct connections (higher-level overlapping connections) with a single connection.

### 2.1.6 Known and unknown distinct and overlappingpatterns

As mentioned earlier, a pair of individual objects must either be distinct, overlap or one part of the other. However,I we do not always know which pattern holds and sometimes cannot find out without considerable analysis. In many cases, it is not worth the effort of finding out and we can leave the point unresolved. In this situation, we model our ignorance with a lack of signs.

A more subtle ignorance occurs when two individual objects are signed in the model as overlapping, but appear distinct because no common part objects are signed. For example, the model in *Figure BG2-2* signs the island of Ireland and the country, the United Kingdom, as overlapping but does not contain an overlapped object that is a part of the two objects. However, this does not imply that the objects are distinct, just that the model does not 'know' any of the parts in the overlap.

### 2.1.7 The distinct and overlapping individual objects pattern objects

According to object semantics, we should be able to point to the objects referred to by a model's signs. None of the signs should refer to mysterious unknowable objects. This raises the question of what objects the distinct and overlapping signs refer to. Take, for example, the distinct sign in *Figure BG2-1*. What object does this refer to?

The distinct and overlapping individual object signs work in a similar way to the individual whole-part sign and most other pattern signs. They refer to an object and its class. It is tempting to suggest that as we talk about distinctness as a connection, that the distinct sign should, like the whole-part sign, refer to a tuple object. This will not work because the distinct and overlapping patterns are, unlike the whole-part pattern, symmetric.

This means that saying 'A is distinct from B' is no different from saying 'B is distinct from A'. (Saying 'my hand is part of my arm' is different from 'my arm is part of my hand'.)

We can see how this causes a problem with an example. Consider the distinct sign in *Figure BG2–1*. Assume that this refers to the tuple <me, my car>. The couple <me, my car> is belongs to the distinct tuples class. We have no guarantee that the couple <my car, me> also belongs to the distinct tuples class. This raises the decidedly contradictory possibility of me being distinct from my car, while at the same time my car is not distinct from me.

The distinct and overlapping objects are actually the classes of the distinct (or overlapping) objects. In the example, the distinct object is the class {me, my car}. Our earlier problem is resolved, because, unlike a tuple, members of a class are not ordered; {me, my car} is the same class as {my car, me}. The distinct and overlapping pattern objects, are then classes of classes—the class of distinct class objects and the class of overlapping class objects. Examples of the two pattern objects are diagrammed in *Figure BG2–7*.

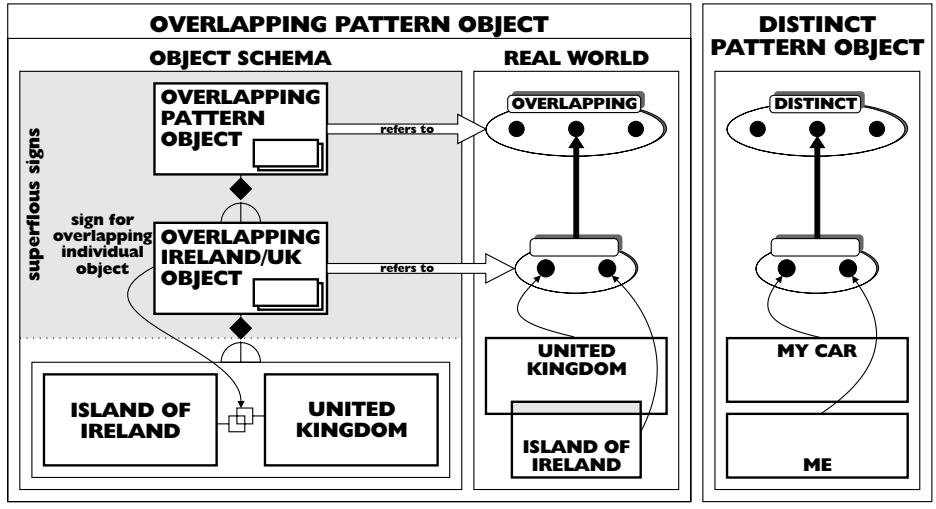
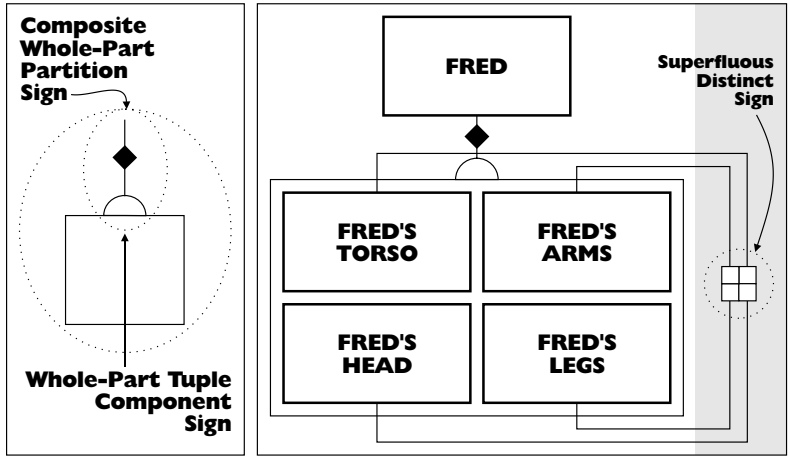



Figure BG2–7 Individual object examples of distinct and overlapping pattern objects

### 2.1.8 Partitioning patterns for distinct individual objects

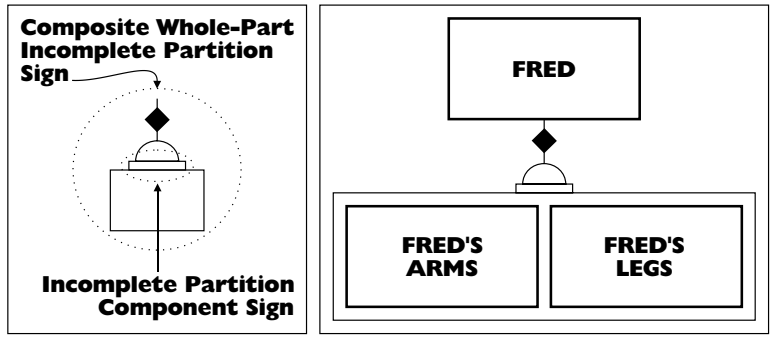
Distinct patterns, particularly useful distinct patterns, frequently arise from the partition of an object into distinct parts. We find this a natural way of seeing. For instance, when we see a person, we are almost instinctively already partitioning them—arms (hairy), legs (long), face (round), etc. The partitioning objects are distinct parts of the whole object and we can model this by combining the whole-part and distinct patterns into a partition pattern. The sign for the composite pattern is shown in *Figure BG2-8*. The component whole-part tuple sign describes the whole-part element and the partition box, the distinct element. Individual objects contained within the partition box are distinct.

Figure BG2-8
A partitioned individual object




When we model, we often do not want to partition an individual object completely; we only want to look at some of its parts. Then, we use a partial or incomplete partition. We sign the incompleteness with a partial sign (a small flat rectangle) between the whole-part sign and the partition box (shown in *Figure BG2-9*).

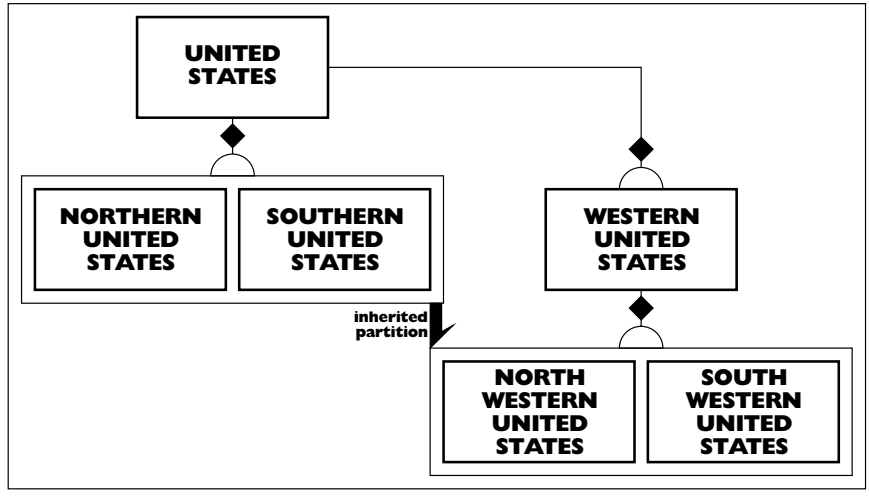
Figure BG2-9
An incompletely partitioned individual object




## 2.1.9 Inheriting partition patterns

Individual object partitions are inherited down the whole–part hierarchy as the example in *Figure BG2–10* shows. The partition of the United States into the Northern United States and the Southern United States is inherited by the Western United States—giving us the North-Western and South-Western United States.

Figure BG2–10
Individual object partition inheritance



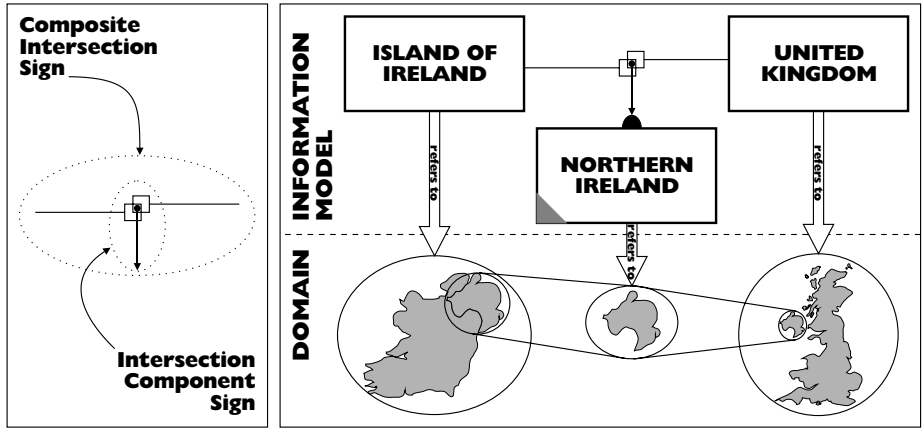

As with distinct and overlapping inheritance, this has implications for how we model partitions. I have found it useful to push the partitions as far up the whole–part hierarchy as they will go. This increases the number of objects that can

inherit the pattern, and so automatically increases the functionality. It also compacts the model, eliminating the need for a number of lower level partitions.

### 2.1.10 Intersection pattern for overlapping individual objects

Sometimes we take two overlapping individual objects and recognise their overlapping part as an object. This pattern is called an intersection and is signed in the model. In the example shown in *Figure BG2-11*, we sign the intersection of the island Ireland and the country, the United Kingdom, to give the country Northern Ireland.

Figure BG2-11 Intersected individual object

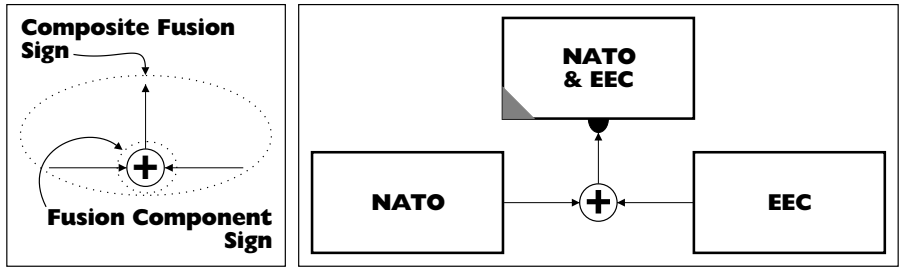



The intersecting object, the country Northern Ireland, is logically dependant on the intersected objects. This is signed in the model in two ways. First, this is shown by a logical dependency component sign. This is a black semi-circle at the intersecting object end of the composite intersection sign (shown in Figure 10.11). Second, Northern Ireland is signed as derived with a grey triangle in the bottom left corner of the Northern Ireland sign. This derived component sign is needed because, when the Northern Ireland sign appears on a schema that does not have both the Ireland and the United Kingdom signs, we cannot draw its intersection sign and so its logical dependency sign. Then, the derived sign reminds us that it is logically dependent.

### 2.1.11 Fusion pattern for overlapping individual objects

Sometimes we construct a new object by fusing a number of overlapping individual objects. The extension of the new object is the fusion of the extensions of the individual objects. If the individual objects were distinct (as in *Figure BG2-10*) then we would have a partition pattern. Where they overlap, we have the potential for a fusion pattern. For example, NATO and EEC overlap and so we can fuse them to get NATO & EEC. This is the geographic area covered by countries that are members of both NATO and the EEC. This fusion is recorded in the model in *Figure BG2-12* with a fusion sign. As with the intersecting pattern, the fusion pattern creates a logical dependency. This is signed with the same logical dependency and derived signs.

Figure BG2-12
Fusion sign




## 2.2 Class object level patterns

The distinct and overlapping patterns between extensions, which we have just examined for individual objects, appear again at the class level. Although, at that level, the super-sub-class hierarchy plays the role of the whole-part hierarchy. We now analyse the class level patterns in the same way as we analysed the individual object level ones. Like before, we start with the simple patterns that hold between pairs of distinct and overlapping classes before moving onto larger groups of classes.

We then examine a similar set of associated patterns:

- Inheriting distinct and overlapping class patterns,

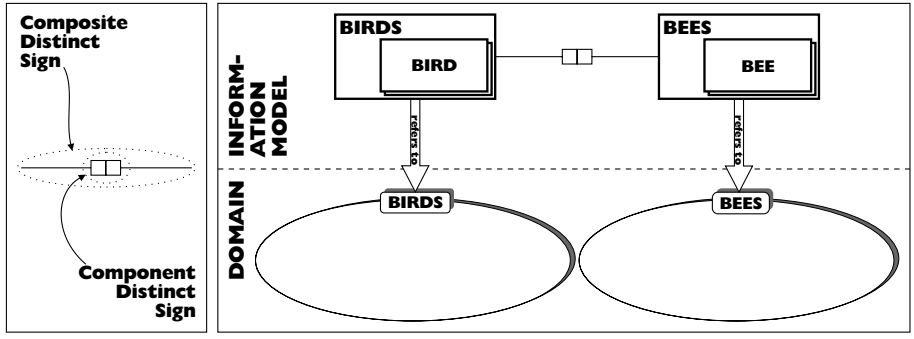
- Known and unknown distinct and overlapping classes,
- Partitioning patterns for distinct classes,
- Intersection pattern for overlapping classes, and
- Fusion pattern for classes.

We also work out what objects the distinct and overlapping class signs refer to.

## 2.2.1 Distinct pairs of classes

A pair of classes that does not have any members in common is distinct. For example, the classes birds and bees are distinct. A member of the class birds is never a member of the class bees. As shown in *Figure BG2–13*, we sign this pattern with the same distinct sign we use for individual objects. Distinctness is a connection between classes; so, the class signs, and not their member signs, are linked.

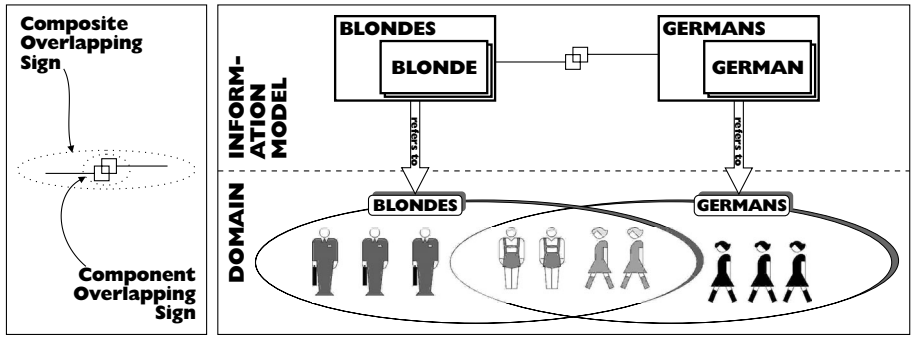
Figure BG2–13
Distinct sign




## 2.2.2 Overlapping pairs of classes

A pair of classes that has members in common overlap. For example, the classes blondes and Germans overlap—there are Germans with blonde hair. As shown in *Figure BG2–14*, we sign this pattern with the same overlapping sign that we use at the individual object level. Overlapping, like distinctness, is a connection between classes, so the overlapping sign links class signs.

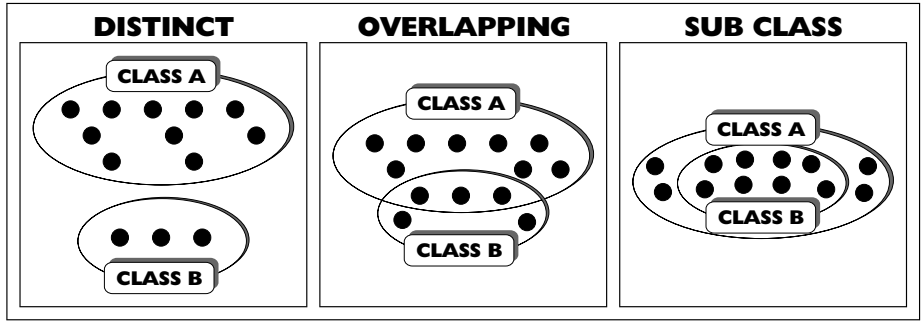
Figure BG2-14
Overlap sign




## 2.2.3 Three main types of connection for pairs of classes

From an extensional point of view, pairs of classes have a similar set of structural patterns to individual objects. These are the distinct, overlapping and sub-class (matching individual object's whole-part) patterns shown in *Figure BG2-15*. A pair of classes must fall under one of these patterns. We could regard the super-sub-class pattern, where one class completely contains another, as an extreme case of overlapping. However, in a similar fashion to individual objects, the convention is to consider this a sub-class and not an overlapping pattern.

Figure BG2-15
Pattern for classes

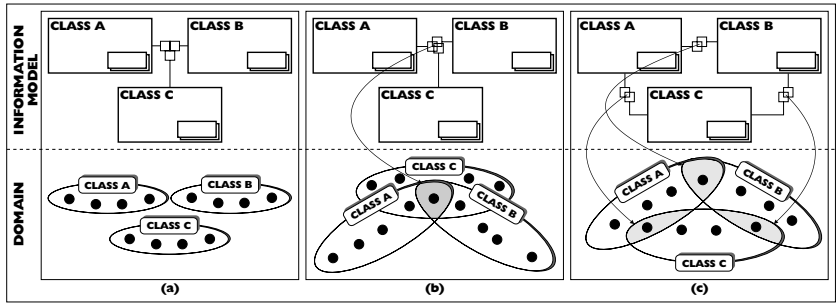



## 2.2.4 Larger groups of classes

As with individual objects, for groups of classes larger than two, there are a wider variety of possible patterns of connection. It is possible for them all to be distinct [shown in *Figure BG2-16* (a)]; or for them all to overlap [shown in *Figure BG2-*

16 (b)]. It is also possible that some will be distinct and some will overlap. Even if every pair in a group of classes overlaps, the whole group may not overlap [shown in Figure BG2-16(c)]. However, the same is not true for distinctness. If every pair of classes in a group is distinct, then the group is distinct.

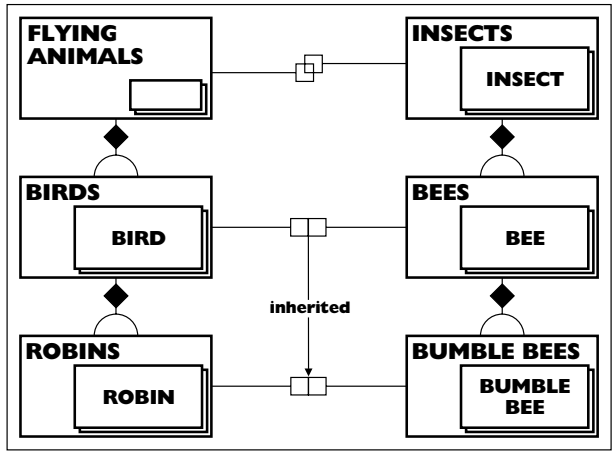
Figure BG2-16 Schemas for larger numbers of classes




## 2.2.5 Inheriting distinct and overlapping patterns

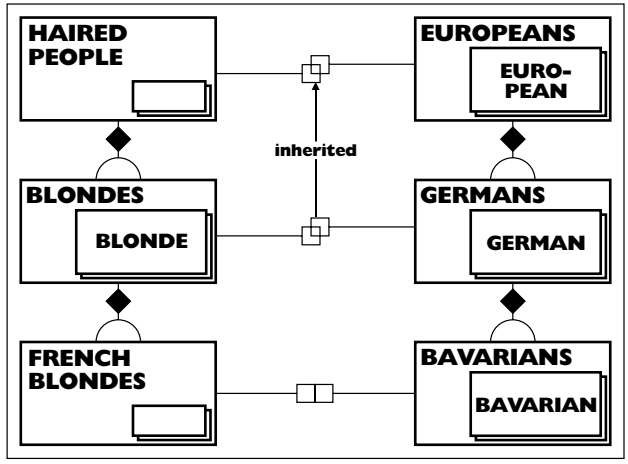
Both the distinct and overlapping class patterns are inherited along the super-sub-class hierarchy, but in opposite directions (matching the patterns for the individual object level's inheritance along the whole-part hierarchy). The distinct pattern is inherited down the hierarchy. For example, the classes, birds and bees, are distinct and so their sub-classes, robins and bumble bees, inherit that distinctness. But, as Figure BG2-17 illustrates, their super-classes flying animals and insects do not, thus proving distinctness is not inherited upwards.

Figure BG2–17
Inheriting distinctness




Overlapping is inherited up the hierarchy. For example, as illustrated in *Figure BG2–18*, the classes blondes and Germans overlap and so their super-classes, haired people and Europeans do as well. However their sub-classes, French blondes and Bavarians are distinct proving that overlapping is not inherited down the hierarchy.

Figure BG2–18
Inheriting overlapping




As with the individual level connections, this inheritance has implications for how we model. We push the distinct connections as far up the super–sub-class hierar-

chy as far as they will go and the overlapping connections as far down the hierarchy as they will go. This compacts and increases the functionality of the model.

### 2.2.6 Known and unknown distinct and overlapping patterns

We often do not know all the members of a class. So we cannot always say whether a group of classes is distinct or overlapping and sign this in the model. This lack of information is not necessarily a problem. We only need to know the relevant distinct or overlapping patterns. Working out every pattern, relevant or otherwise, would be a waste of time.

However, when we want to model a group of classes as overlapping, it helps to know at least one common (overlapped) member. There is, in principle, nothing wrong with signing them as overlapping when we do not know a common member. However, this is not a good policy. Finding a common member is a sure way of confirming that the classes do indeed overlap. Even if we are reasonably sure that they do, it makes sense—as a safety check—to follow a policy of confirming our intuitions. We can do this simply and effectively by finding a common member. *Figure BG2-19* illustrates this process of confirmation. If we cannot find a common member, this should make us suspect that the classes do not, in fact, overlap.

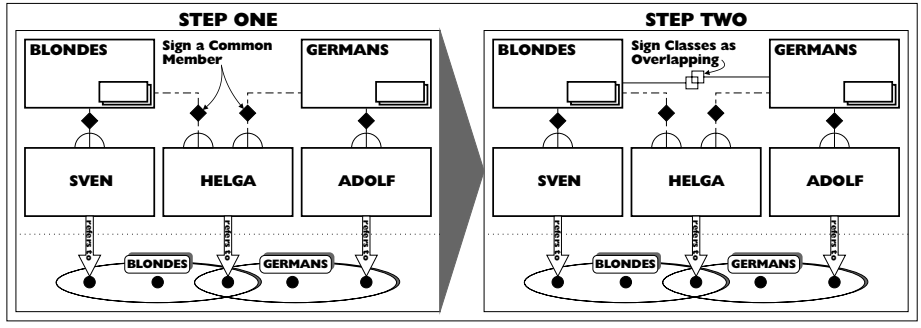



Figure BG2-19 Constructing confirmation of overlapping

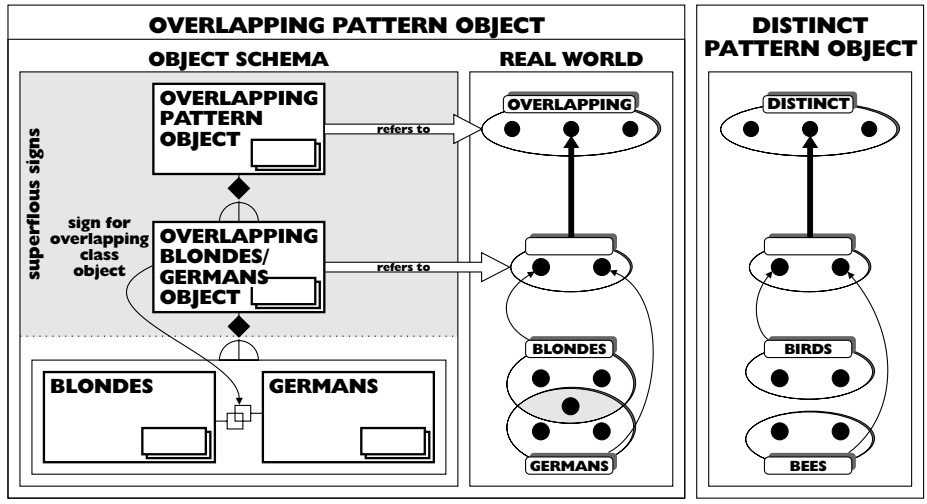
Things are not so easy for distinct patterns. No matter how many distinct instances two class signs may have, this does not prove that their classes are distinct. There may be an unknown object that is a member of both classes. So a group of classes cannot be logically proven to be distinct in the same way as they

can be proven to be overlapping. This means we need to exercise caution before signing classes as distinct in the model.

### 2.2.7 The distinct and overlapping class pattern objects

The strong reference principle requires that, as we have signed distinct and overlapping class patterns, the signs refer to objects. These are constructed in the same way as their individual object cousins. They are the classes of the distinct (or overlapping) classes. We can illustrate this with the distinct birds and bees classes from *Figure BG2-13*. Its distinct sign refers to the class {birds, bees}, which has the birds and bees classes as its only members. Furthermore, this class is a member of the distinct class. This is shown in *Figure BG2-20*, which also shows an example of the pattern for the construction of the overlapping class.

Figure BG2-20 Class examples of overlapping and distinct pattern objects

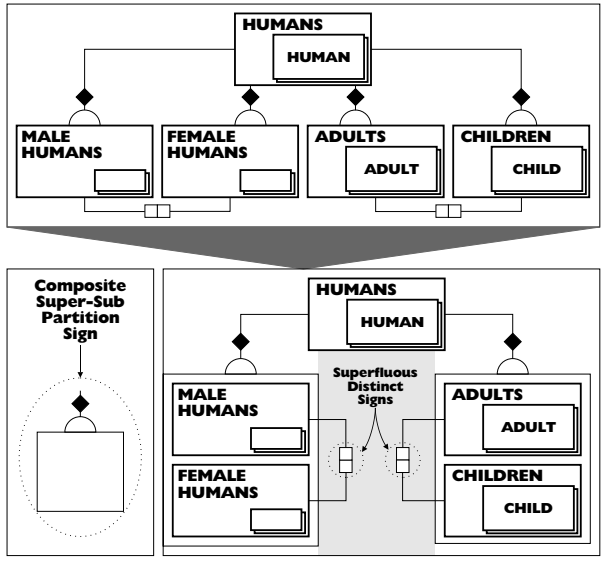



### 2.2.8 Partitioning patterns for distinct classes

Like individual objects, where a distinct pattern is often part of a larger individual partition pattern, distinct class patterns are often part of a larger class partition pattern. A type of partitioning class pattern has been a natural way of seeing since well before the emergence of the substance paradigm and its secondary

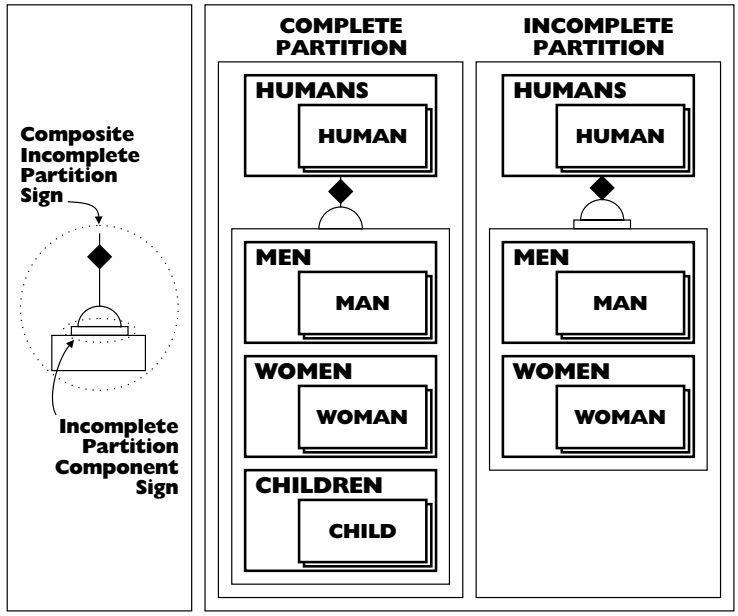
substance hierarchy. For example, when we think of the class humans, we almost instinctively start partitioning it, maybe by gender. Then, even though it contravenes the substance paradigm's single classification restriction, some of us also start thinking of alternative ways of partitioning, for example into adults and children. In the class partition pattern, the partitioning class is divided into distinct partitioned sub-classes. As shown in *Figure BG2-21*, the notation is similar to the individual object partition sign—with the component super-sub-class sign replacing the whole-part sign.

Figure BG2-21
Partitioned classes




Often, the partition pattern does not partition a class completely, partitioning only some of its members into distinct classes. This is a partial or incomplete partition and is signed by adding an incomplete partition component sign to the composite partition sign. As shown in *Figure BG2-22*, this is a small flat rectangle that is put between the super-sub-class sign and the partition box.

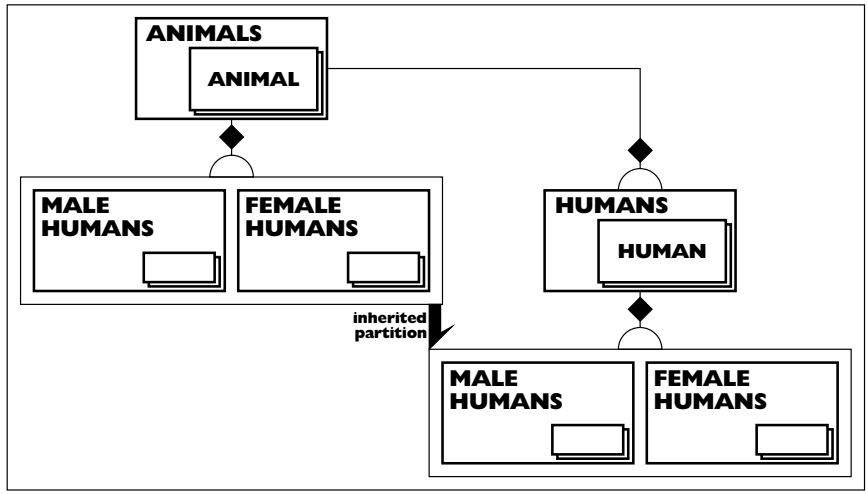
Figure BG2–22 Incompletely partitioned classes




**Partition pattern inheritance**

As the example in *Figure BG2–23* shows, partition patterns (like distinct patterns) are inherited down the super–sub-class hierarchy. The partition into distinct male and female animals classes is inherited down the super–sub-class hierarchy to the distinct male and female humans classes partition. (You will notice that the inherited partition is the more general male/female partition from *Figure BG2–21* rather than the human specific men/women partition from *Figure BG2–22*.)

Figure BG2-23 Partition inheritance




As with individual level partitions, this has implications for how we model class partitions. It is useful to push them as far up the super–sub-class hierarchy as they will go, increasing the number of classes that can inherit the pattern. This compacts and increases the functionality of the model.

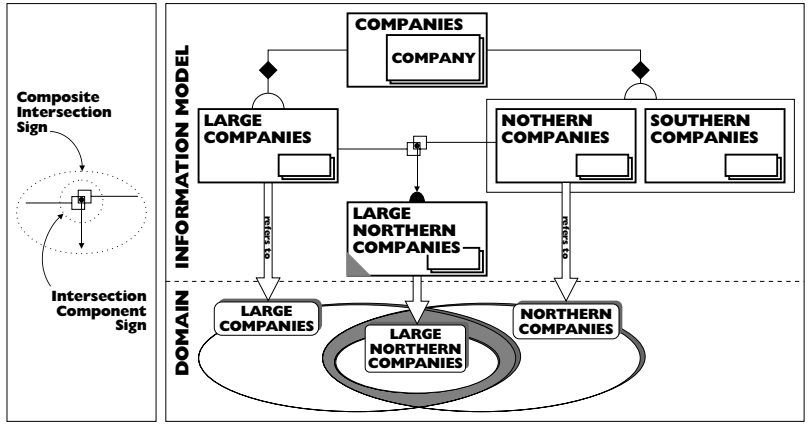
### 2.2.9 Intersection patterns for overlapping classes

Sometimes we want to work with a class constructed from objects that are members of the overlap of a group of classes. This is an intersection pattern, which goes one step further than the overlapping pattern and constructs the class of the overlapped members. The intersection pattern only applies to overlapping classes, it cannot apply to the other two types of class patterns: distinct and sub-class. Distinct classes have no members in common and so have no use for the intersection pattern. Sub-classes have all their members in common with their super-class, and so the intersection pattern would not produce a new class.

We can see how the intersection pattern works with an example. Assume we are targeting a group of companies for a sales campaign and we are going to select the group from a comprehensive list. The list identifies whether companies are large and whether their headquarters are in the north or south of the country. If

we target large companies in the north (in other words, the class of companies whose members belong to both the large companies class and the northern companies class) then we need the intersection pattern shown in *Figure BG2-24*. This illustrates the intersection sign, which is an enhanced version of the overlap sign.

Figure BG2-24 Intersected classes



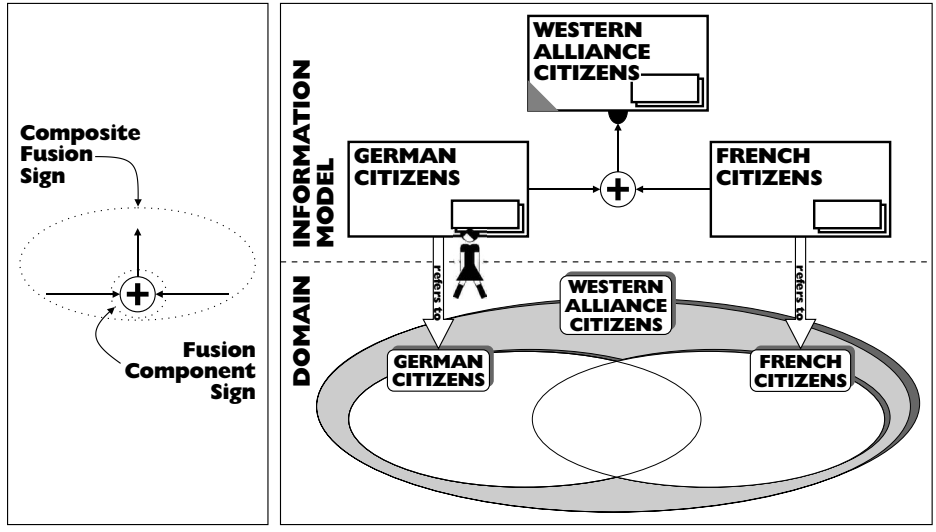

The example in *Figure BG2-24* also confirms that only overlapping classes can be intersected. It is pointless intersecting the classes northern companies and southern companies because they are distinct (as they are part of a partition). So we know in advance that the intersecting class would be empty. The intersected classes in the intersection pattern must be overlapping so that the intersecting class has members.

The intersecting class, large northern companies, is logically dependant on the intersected classes, large companies and northern companies. The logical dependency is shown by a black semi-circle sign at the end of the intersection sign (seen in *Figure BG2-24*. The class is derived by the logical dependency. This is shown by the derived sign, a small grey triangle in the bottom left corner of the class sign. Again this is visible in *Figure BG2-24*. This derived component sign becomes an integral part of the composite sign for the class. It needs to be because the class sign can appear in other schemas without the intersection sign and so the logical dependency sign. Then the derived sign reminds us of the logical dependency.

## 2.2.10 Fusion patterns for overlapping classes

Sometimes every member of a group of overlapping classes has an interesting characteristic and this is captured by a class that pools all the members of the group of classes. For example, at some future date it may be decided to make the citizens of France and Germany citizens of a new Western Alliance state. The class Western Alliance citizens is the pooling of the members of the classes French citizens and German citizens. This pattern is called a fusion and is modelled using a fusion sign (shown in *Figure BG2-25*). You will notice that the classes French citizens and German citizens overlap; it is possible to have dual citizenship. If they did not (the classes were distinct), this would be a partition pattern. The fused class, Western Alliance citizens, is logically dependant on the classes French citizens and German citizens. This is shown in the same way as for intersected classes, with a logical dependency and a derived sign.

Figure BG2-25
Fusion sign




## 2.2.11 A close-knit family of extension patterns

This examination of the patterns of connections between extensions has revealed a close-knit family of patterns. We have seen how patterns at the individual object level repeat themselves at the class level. How patterns are inher-

ited up and down the super–sub-class and whole–part hierarchies. How we can and should generalise the connections along their inheritance hierarchies to compact and increase the functionality of the model. The examples have given us a feel for how these patterns work with one another. As we get more experience of business object modelling, they will become second nature.

# 3 State hierarchy patterns

In the working paper *OP4—Business Object Ontology Paradigm* we examined how object semantics explained substance's states as physical bodies that are temporal parts of other physical bodies. Here, we look at the basic object syntax for states. We look at the sign for a state and how to model the following state patterns:

- State–sub-class hierarchy patterns,
- State–sub-state hierarchy patterns,
- Distinct state patterns,
- Partitioned state patterns, and
- Overlapping state patterns.

These are all spatio-temporal patterns. In the next section, we look at temporal (time ordered) patterns.

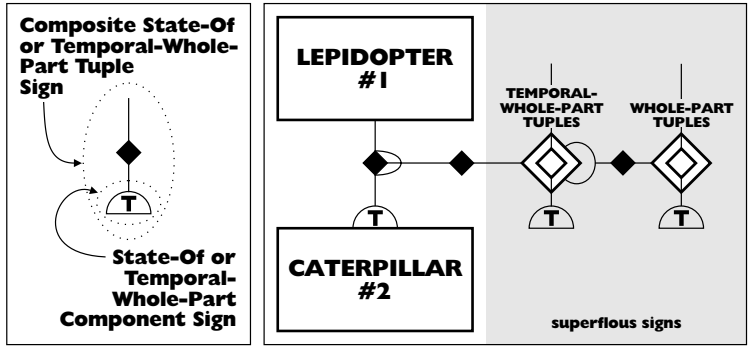
## 3.1 The state-of sign

A state is a physical body that is a temporal part of another physical body. This link between the state and the physical body is a particular type of whole–part tuple. Consider the lepidopter example from *OP4—Business Object Ontology Paradigm* (illustrated in OP4's *Figure OP4–14*), where caterpillar #2 is a state of lepidopter #1. As *Figure BG2–26* shows, the state–of tuple is a couple <lepidopter #1, caterpillar #2> belonging to the temporal–whole–part tuples class. (This is the states tuples class; all states are, by definition, temporal parts of physical bod-

ies.) The temporal-whole-part tuples class is, in turn, a sub-class of the whole-part tuples class.

As the couple belongs (distantly) to the whole-part class, we sign it with a whole-part sign. To reflect the fact that caterpillar #2 is a temporal part (state) of lepidopter #1, the composite state-of sign has a state-of or temporal component sign. In *Figure BG2-26*, the whole-part and temporal-whole-part tuples classes are drawn. However, these are normally left out of the schemas because they are superfluous, implied by the state-of or temporal-whole-part sign.

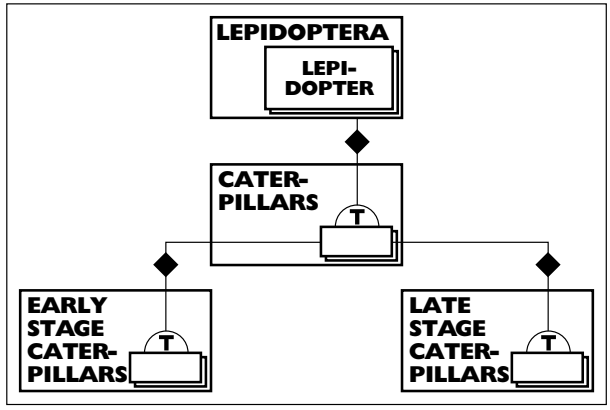
Figure BG2-26 Temporal-whole-part or 'state-of' sign




## 3.2 State-sub-state hierarchy patterns

We saw in *OP4—Business Object Ontology Paradigm* that states can have states and this leads to a state-sub-state hierarchy pattern. In the example illustrated in OP4's *Figure OP4-17* and *Figure OP4-18*, caterpillars had early and late stage sub-states, where a substate is defined as a temporal part of a temporal part. So, as shown in *Figure BG2-27*, the pattern is signed using the state-of sign. You should notice that this pattern is at the member level, with the state tuples signs connecting the classes' member signs.

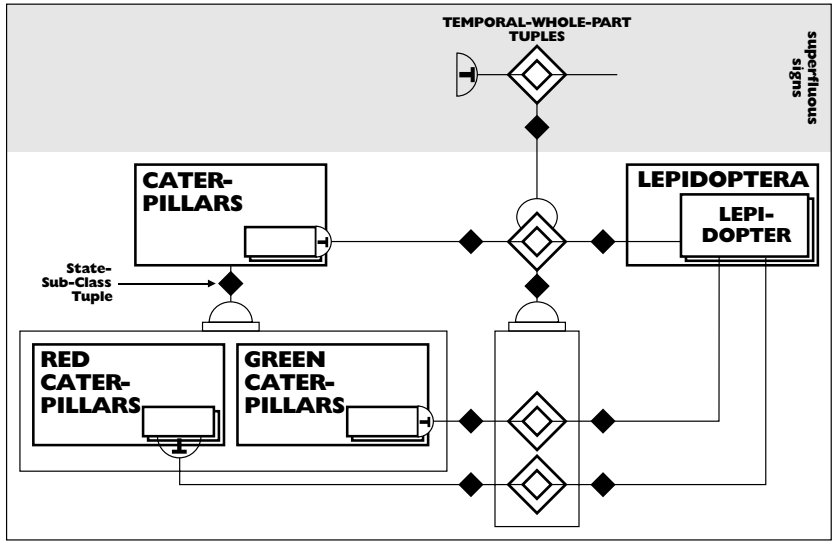
Figure BG2-27 State-sub-state hierarchy pattern




## 3.3 State-sub-class hierarchy patterns

*OP4—Business Object Ontology Paradigm* also shows us that states are collected into state classes that can have state-sub-classes. This state-sub-class pattern is just a super-sub-class pattern, where the classes are state classes. *Figure BG2-28* shows this using the example illustrated in OP4's *Figure OP4-19* and *Figure OP4-20*, where the caterpillars (state) class has red and green (state) sub-classes.

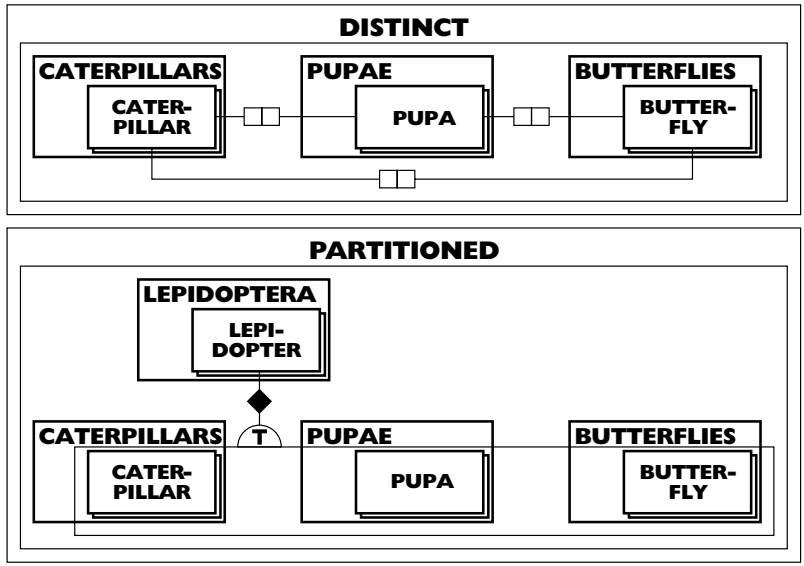
Figure BG2-28 State-sub-class hierarchy pattern




## 3.4 Other extension-based state patterns

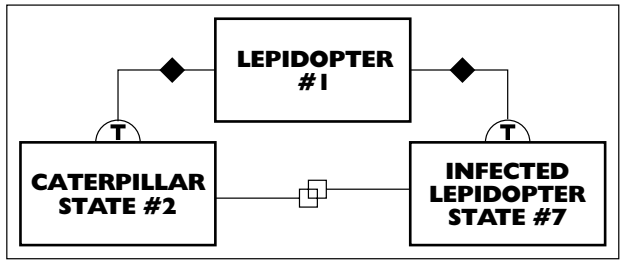
States, as physical bodies, fall into the same extension-based patterns as other physical bodies. For instance, they have the distinct, overlapping and partitioned patterns we examined in the beginning of this paper. We illustrate this using the lepidopter example again. Its states are distinct and also completely partition the lepidopter object. *Figure BG2-29* models these two patterns. You can see that the partition is modelled connecting the classes' members icons, this is because it operates at the member level.

Figure BG2-29 Distinct and partitioned states




In *Figure BG2-30*, we have used the overlapping caterpillar and infected lepidopter states from OP4's *Figure OP4-21* and *Figure OP4-22* to illustrate how we sign an overlapping state.

Figure BG2-30 Overlapping states




# 4 Time ordered temporal patterns

In *OP4—Business Object Ontology Paradigm*, we examined how object semantics explains changes using these two types of object:

- States, and

- Events.

We now look at the object syntax for their time ordered temporal patterns.

## 4.1 State changes

In object semantics, states are objects and often ordered in time. This ordering can take a number of patterns; we only look at this sample here:

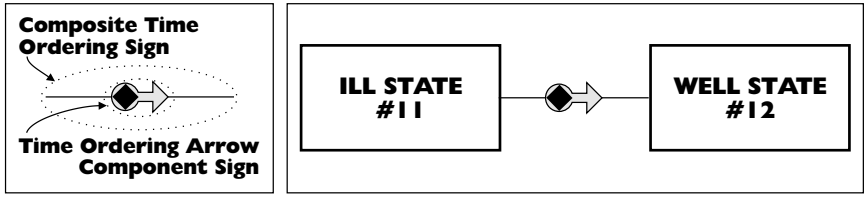
- Simple state 'change' patterns,
- Sequence of states pattern, and
- Alternating states pattern.

We then investigate how the state life history of an object is constructed from a states' time ordering patterns.

### 4.1.1 A simple state 'change' pattern

The simplest state change involves a 'change' from one state to another—for instance, a change from an ill state into a well state. The states are ordered in time – one after the other. To describe this pattern, we construct a tuple of the two states and sign its order with a component time ordering arrow sign (shown in *Figure BG2-31*).

Figure BG2-31
Sign for time ordering



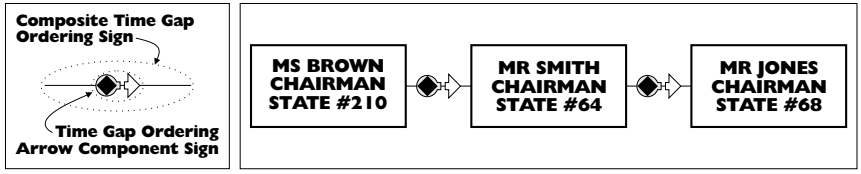

### 4.1.2 A time sequence of states pattern

Often the states of an object fall into a time sequence pattern. We can describe this pattern at an individual object level or generalise it to a class level—as in the chairman and lepidopter examples below.

**Individual object level sequence**

The chairman thought experiment from *OP4—Business Object Ontology Paradigm* (illustrated in its *Figure OP4–29*) provides a good example of a time sequence pattern of individual states. Each new resignation and appointment leads to a new chairman state. If we extend the pattern in the thought experiment we get a sequence, over time, of chairman states—all states of the chairman object. In this case, the sequence of states has a temporal gap. This is modelled with the time 'gap' ordering arrow component sign shown in *Figure BG2–32*.

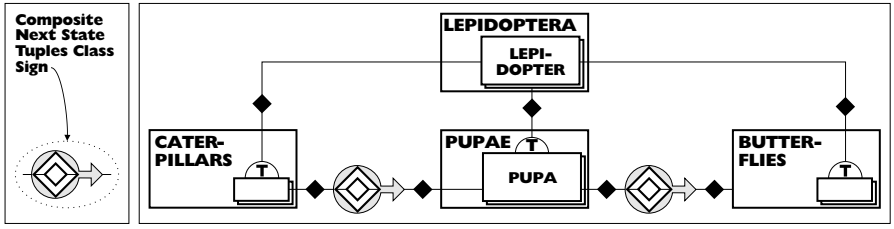
Figure BG2–32 Individual object level sequence of states




**Class level sequence**

The ubiquitous lepidopter provides us with an example of a class level sequence pattern. Caterpillars develop into pupae that develop into butterflies. It is the members of the classes that develop, not the classes themselves, so the ordering sign is linked to the class members' signs (shown in *Figure BG2–33*), not the class signs.

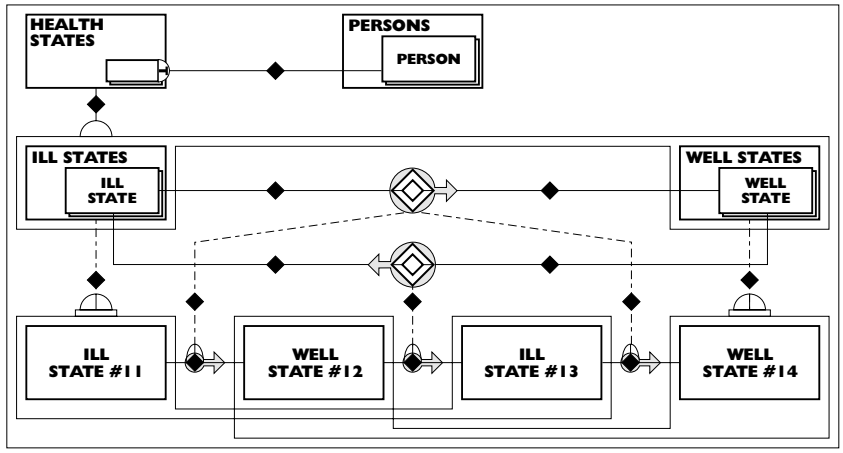
Figure BG2–33 Class level sequence of states




### 4.1.3 Alternating state patterns

States also fall into an alternating pattern, as shown in the well and ill states example in *OP4—Business Object Ontology Paradigm* (*Figure OP4–28*). We model this using the sign for time ordering (shown in *Figure BG2–34*). You should notice that, in this case, the model shows both the individual object and the class level ordering.

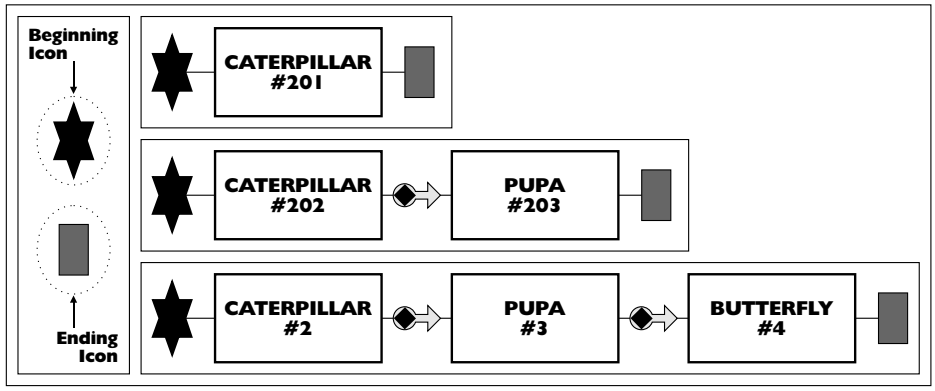
Figure BG2-34
Alternating state patterns




### 4.1.4 An object's state life history

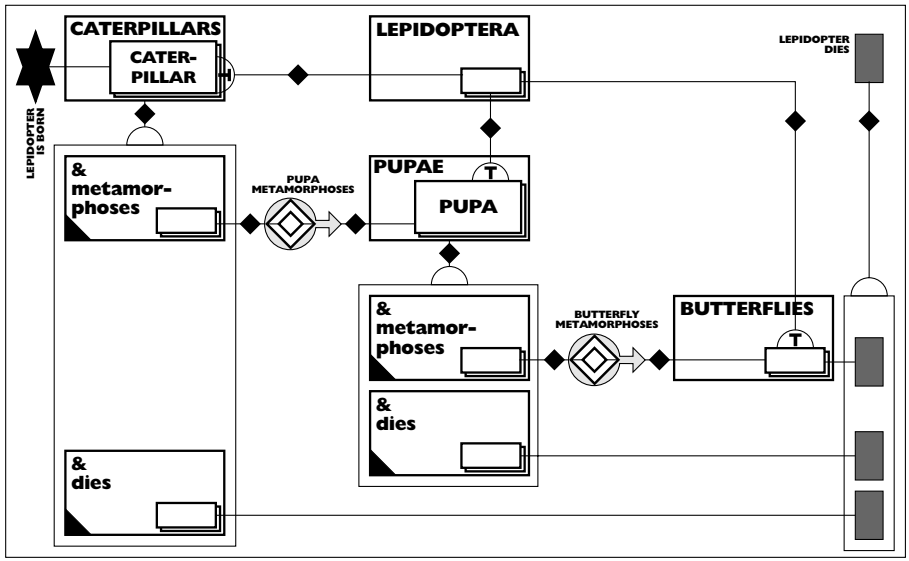
These signs for states' time orderings allow us to tell an individual object's state life history (or indeed, a class of objects' state life histories). Consider the lepidoptera example again. To determine its state life history we first need to find all the possible patterns for its individual states. *Figure BG2-35* provides a simplified version of these in the form of state life histories for three individual lepidoptera—each one dying at a different stage of development. Notice the beginning and ending signs. These are, as you can see, based on the space-time map icons.

Figure BG2-35
Three individual lepidopter state life histories



We generalise these individual level patterns into a class level history; the result is *Figure BG2–36*. Notice that as the state life histories are of the individual states of the physical object, the time ordering pattern is between the members and not the classes. This is a very simple example. Normally, an object would have a number of different state partitions, across which states would overlap (illustrated in*Figure OP4–21* and *Figure OP4–22*).

Figure BG2–36
A class level lepidoptera state life history




People familiar with traditional modelling may recognise this as object syntax's version of the entity paradigm's life history diagrams. Getting a picture of something's life history is an extremely useful part of business modelling. However, the entity life history has to work within the confines of the entity paradigm, which typically constrains it to a tree-like structure. Using the more powerful and sophisticated object semantics enables us to construct a much more accurate, and so useful, picture of a life history.

## 4.2 Event cause and effect time orderings

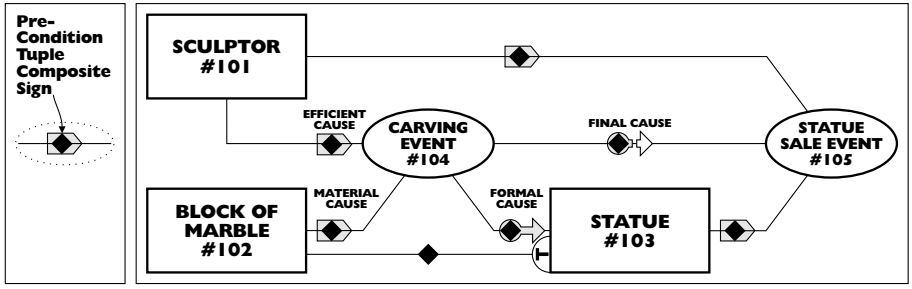
As well as a life history perspective on objects, object syntax offers a cause and effect perspective centred on events. In *OP4—Business Object Ontology Paradigm*

we discussed how Aristotle analysed understanding into the following four types of cause:

- Efficient cause,
- Material cause,
- Formal cause, and
- Final cause.

We now look at how these are modelled with time ordering signs. We do this by example. We model, using object syntax, the 'sculptor carving a statue' example illustrated in *Figure OP4–37*. The result is *Figure BG2–37*. We use a new sign (the pre-condition sign) for the efficient and material causes because the causes are not ordered before or after the event, but around it. The efficient and material causes are differentiated because the material cause has a temporal–whole–part connection with the formal cause.

Figure BG2–37
Object syntax's event perspective



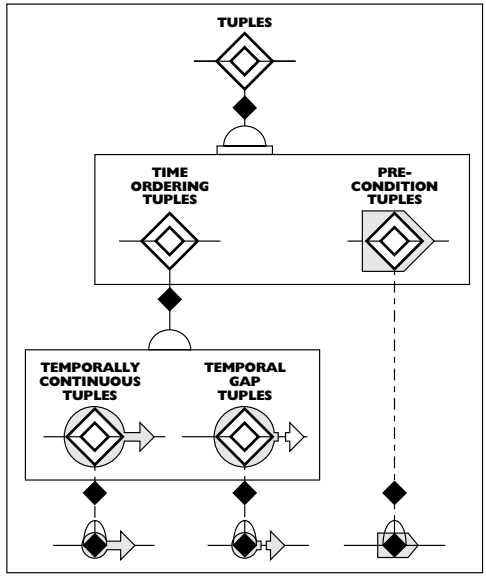

The life history and event perspectives complement one another. The life history fits the states into a pattern. The event perspective then explains that pattern by mapping what 'causes' the events that change the states.

## 4.3 Time ordering tuple objects

We have looked at various time ordering (and pre-condition) signs. We now examine, in deference to the strong reference principle, the objects that these signs refer to. They are tuples that belong to the appropriate pattern's tuples class. At the individual level, they are couple objects as indicated by the two place links

to the diamond tuple component sign. These are members of one or another of the time ordering or pre-condition pattern's tuples classes (illustrated in *Figure BG2-38*).

Figure BG2-38 Time ordering and pre-condition tuples classes




# 5 Cardinality patterns for tuples classes

We now move from time ordering tuples to a particular aspect of tuples classes. We look at a group of useful modelling patterns—cardinalities. Traditional information modelling uses cardinality patterns for its relational attributes and we re-engineer a version of the patterns here. A few differences arise because the tuples class and the occupied class places are objects in their own right in object semantics. This is a change from traditional modelling, where cardinalities are implicit parts of relational attributes.

In many cases, it is useful to describe the cardinality patterns of a tuples class, but this notation does not insist on it. A number of notations are used for describing cardinality in traditional information modelling; most of which can be

adapted to object semantics. I prefer to use the simple one described below, but it does not really matter which one is used. I suggest that you use the notation you feel most comfortable with, though remember it will probably need some amendments to cope with object semantics.

## 5.1 Types of cardinality pattern

*BG1— Constructing Signs for Business Objects* looked at the signs for tuples classes and their occupied class places. These occupied class places are the basis for the cardinality patterns. Cardinality is a pattern that, in object syntax, applies to occupied class place objects. When a cardinality pattern is signed, both an upper and a lower bound are specified. There are two levels for the lower bound—optional or one. There are also two levels for the upper bound—one or multiple. These upper and lower bound levels can be combined in four ways to produce four different cardinality patterns for the occupied class place:

- Optional-to-one pattern,
- One-to-one pattern,
- Optional-to-multiple pattern, and
- One-to-multiple pattern.

We now look at each of these in more detail.

### 5.1.1 Optional-to-one cardinality pattern

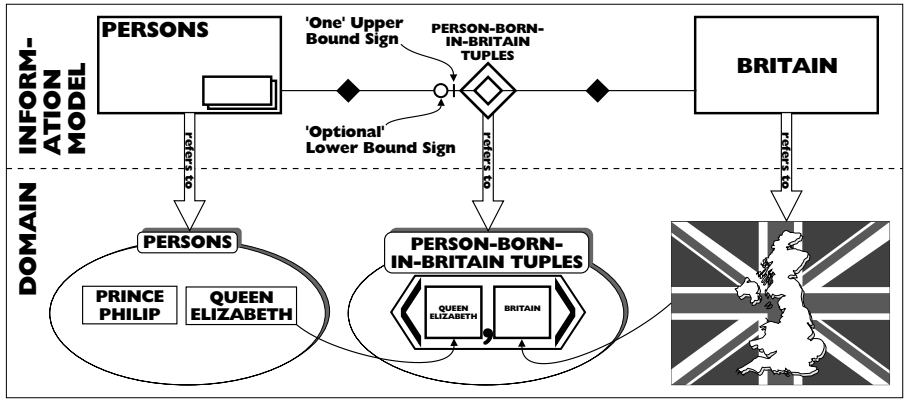
Consider *Figure BG2-39*, which models the person-born-in-Britain tuples class. What is the cardinality pattern of the class place occupied by the class persons? I have found that it is important when determining cardinality to confirm one's intuitions with specific instances. I go through this confirmation process step by step in this example.

Prince Philip and Queen Elizabeth are both members of the class persons. Prince Philip is a person and was not born in Britain. So it must be optional for a person to be born in Britain. Or, in object-speak—it must be optional for members of the

class persons to occupy the person place in a couple that is a member of the person-born-in-Britain tuples class. So the lower bound for the occupied class place is zero. This is signed in a similar way to traditional modelling with an 'O' on the line between the occupied class place and the tuples class sign.

Queen Elizabeth is a person and was born in Britain. So a person can be born in Britain (a member of the class persons can occupy the person place in a couple that is a member of the person-born-in-Britain tuples class). It is safe to assume that a person cannot be born more than once, in Britain or anywhere else. So the maximum number of times a person can appear in the person place of a person-born-in-Britain couple is once. This means the upper bound for the occupied class place is one, which is noted by a '1' sign on the class place link. By convention we draw the upper bound sign closer to the tuples class sign than the lower bound sign. Both these upper and lower bound signs are shown in *Figure BG2-39*.

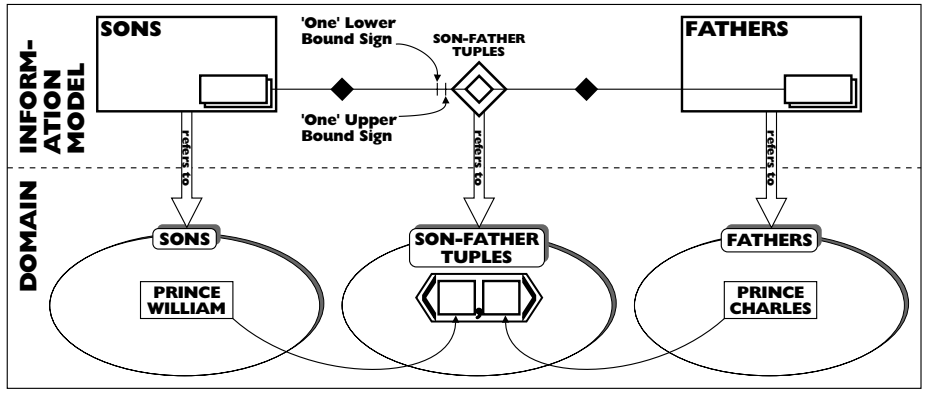
FigureBG2-39
Optional-to-one cardinality pattern




## 5.1.2 One-to-one cardinality pattern

If we now model the son–father tuples class with its place classes son and father then we get the schema shown in *Figure BG2-40*. A son always has one and only one biological father; so, every son appears once and only once in the son place of a father–son couple. This means the upper and lower bounds are both one. So two '1' signs are put by the occupied class place sign, next to the tuples class sign.

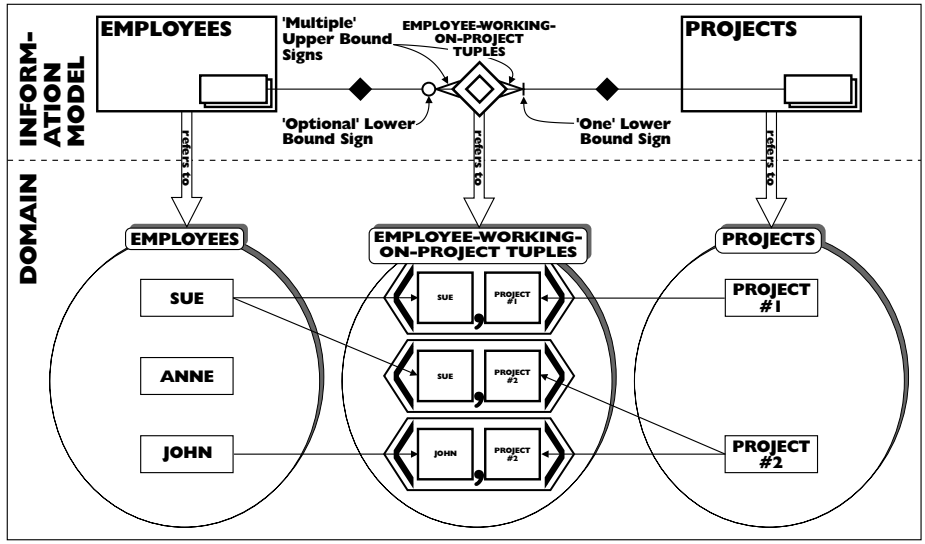
Figure BG2-40 One-to-one cardinality pattern




### 5.1.3 Optional-to-multiple and one-to-multiple cardinality patterns

Now consider the model in *Figure BG2-41*, this shows the employee-project tuples class originally illustrated in *Figure OP1-24* and *Figure OP1-25*. An employee will sometimes work on a number of projects. This means the upper bound for the occupied class place must be greater than one. For this, we use the multiple sign. As you can see, it looks like a crow's foot. Some employees, such as secretarial staff, will never work on a project. So the occupied class place has a lower bound of zero. We use the same 'O' sign that we used in Figure 10.40 for this. All projects have one or more employees working on them. So the lower bound of the occupied class place link is one and the upper bound is multiple. These optional-to-multiple and one-to-multiple cardinalities are is signed in *Figure BG2-41*.

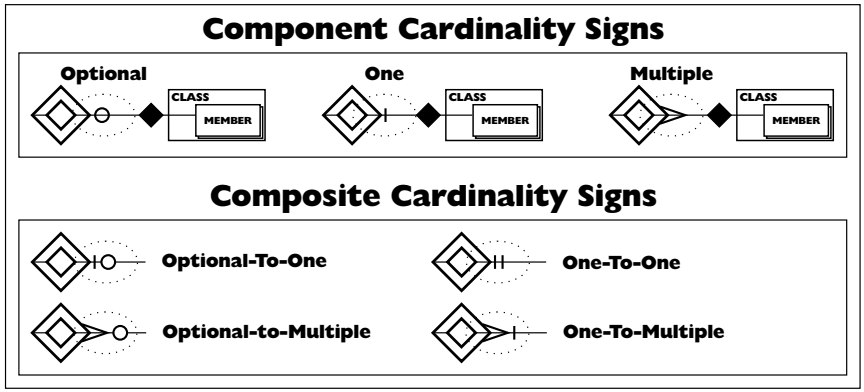
Figure BG2-41
Figure 10.41
Optional- and one-to-multiple cardinality patterns




## 5.1.4 Cardinality pattern signs

These examples cover the only four possible signs for the cardinality of an occupied class place. A full list is given in *Figure BG2-42*. If we are going to sign the cardinality of an occupied class place then we will use one of them. Remember, however, that unlike some traditional modelling notations, each occupied place of a tuples class can be given a cardinality pattern. So a tuples class can have as many cardinality patterns as it has occupied class places.

Figure BG2-42 The four composite cardinality pattern signs




## 5.2 Cardinality patterns as objects

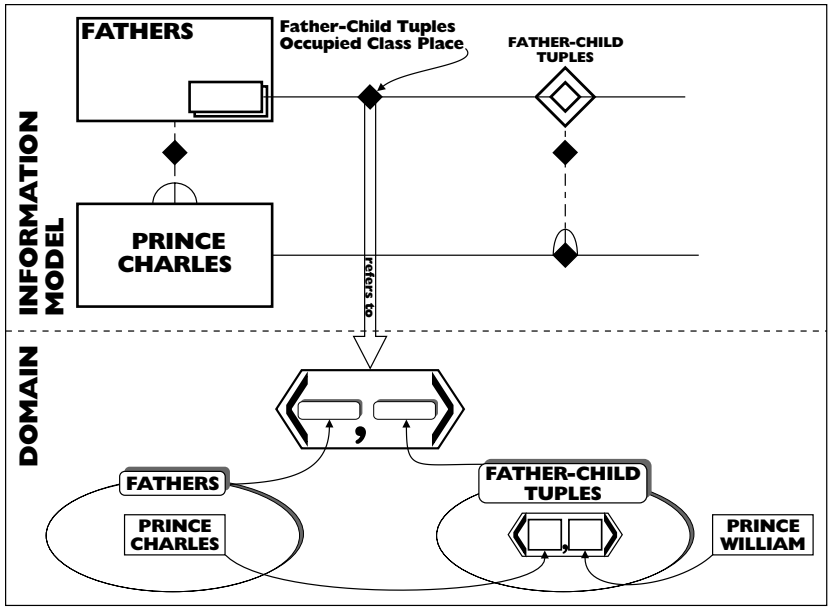
Cardinality signs, like the distinct and overlapping signs, refer to class objects. But, which class objects? If we analyse the model carefully we can see the members of the cardinality classes—occupied class place objects.

### 5.2.1 Occupied class places as objects

The working paper *BG1— Constructing Signs for Business Objects* looked at the signs for occupied class places (see *Figure BG1-25* and *Figure BG2-26*). We now work out what these signs refer to. We start with the signs for individual tuples and work up to the occupied class place signs.

The sign for individual tuples, such as <Prince Charles, Prince William>, is a black diamond. This component has a number of lines, called (tuple) place component signs joining the diamond to the signs for the objects that make up the tuple. For example, in *Figure BG2-43*, a component place sign joins the black diamond tuple sign to the Prince Charles sign.

Figure BG2-43
What occupied class places signs refer to



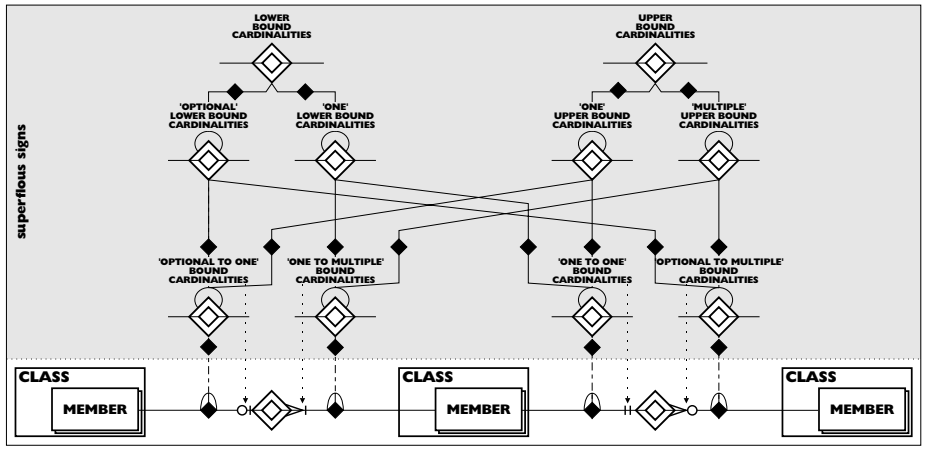

The tuples class signs have a component that looks similar. This is the (tuples) class place sign. Like the place component sign, it is a line. Unlike it, the line does not have to join the tuples class sign to another sign. For example, the class place sign on the right of the father-child tuples class in *Figure BG2-43* is not joined to anything. The class place sign can join the tuples class sign to another sign—as shown by the class place sign joining the fathers class sign to the father-child tuples class in Figure 10.43. When this happens, the class place is said to be occupied and a black diamond (the tuple sign) is added to the line.

What object does this occupied class place sign refer to? Despite the similarity of the signs, it cannot reflect a simple construction relationship as the tuple's place sign does. The connection between the father-child tuples and fathers classes is not one of a tuple constructed from an object. Instead, it is a tuple, <father-child tuples, fathers> (illustrated in *Figure BG2-43*). That is why the occupied class place component sign is a black diamond, the sign for a tuple. In general, occupied class place signs refer to a couple with the format <tuples class, place class>.

### 5.2.2 Cardinality classes with occupied class places as members

These occupied class places are the members of cardinality classes (shown in *Figure BG2-44*). For example, the one-to-one cardinality sign refers to the one-to-one bound cardinalities tuples class. This has as members all occupied class places with a one-to-one cardinality, including the one shown in the figure.

Figure BG2-44 Underlying cardinality model




The figure also illustrates how, once the composite cardinalities are seen as classes, they can be generalised into their elements. The one-to-one cardinality class can be generalised as a sub-class of the 'one lower bound cardinalities' and the 'one upper bound cardinalities' tuples classes. We can also see quite clearly how much easier it is to use the cardinality signs than the more long-winded couple and tuples class–member signs.

## 5.3 Inheriting cardinality patterns

There are constraint patterns for the inheritance of cardinality patterns up and down the super–sub-class hierarchy. They are easy to work out; so try doing it for yourself.

# 6 A pattern for compacting classes

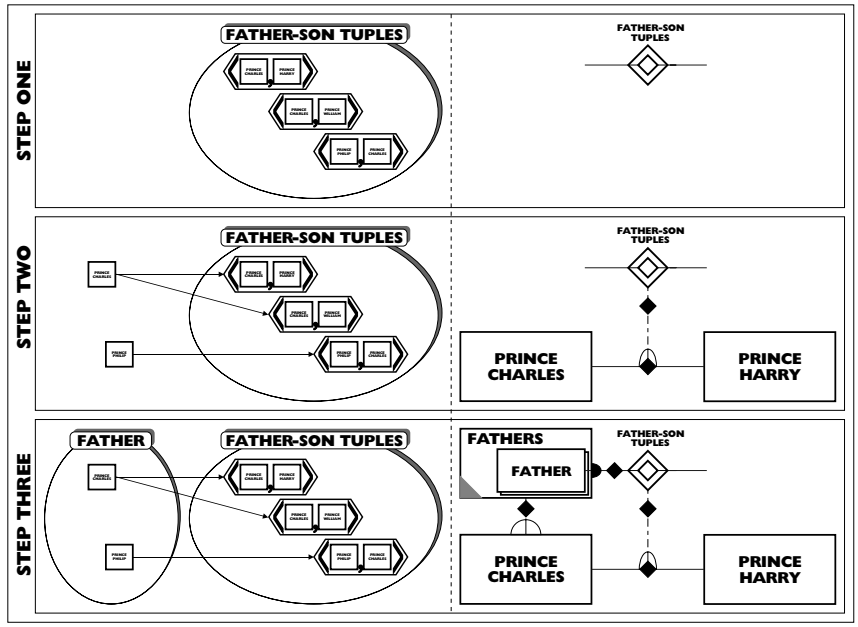
So far, in this paper, we have looked at how object syntax helps us model objects. Now, we turn our attention to a pattern that helps us generalise classes and so compact the model. This is the pattern of tuples classes defining their place classes.

Once we identify the pattern, we generalise the place classes up the super–sub-class hierarchy. We can then eliminate the original, less general, place classes. This compacts the model without compromising its information content. This is a good illustration of one way in which compacting works and how we handle it within object syntax. We shall re-use this compacting pattern in *MW—The BORO Methodology: Worked Examples*.

## 6.1 Constructing an example of the pattern

To illustrate the compacting, we need an example of the pattern. We get one by constructing a derived place class from a tuples class. Step one, shown in *Figure BG2–45*, is taking the father–child tuples class. At this stage, it has no occupied class places. Step two is identifying in each of the member tuples, the object that occupies the father place. Step three is collecting all these objects into a class. This gives us a fathers class that occupies one of the father–child tuples class places.

Figure BG2-45 Constructing the logically dependent place class fathers




The class fathers is defined as those persons who have a father–child tuple linking them to a child—so it is logically dependent on the father-child tuples class. This makes it derived. This is modelled in the usual way; with logical dependency and derived signs (shown in *Figure BG2-45*).

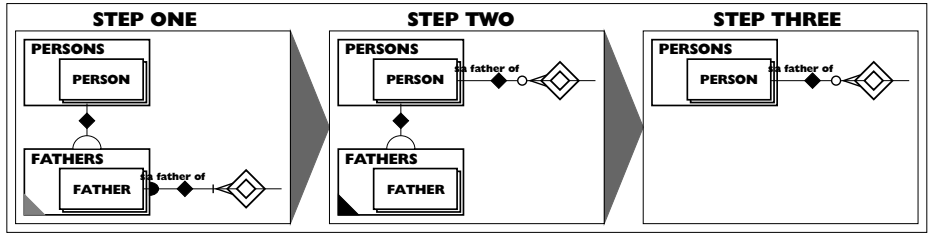
We will spot this pattern frequently if we keep asking whether there is a logical dependency between a tuples class and its place classes. Until now, we have tended to assume that they are logically independent. In this father–child tuples case, and many other cases, if we had asked ourselves the question, we would have realised that there is a logical dependency.

## 6.2 Using the pattern to compact the model

We now have an example of the pattern of tuples classes defining place classes. So we can illustrate the compacting. We do this in the three steps shown in *Figure BG2-46*. In the first step, we generalise the fathers class (the occupied class place) up the super–sub-class hierarchy to the persons class. In the second step,

we generalise the 'is a father of' occupied class place from the father class to the persons class. At this stage, the fathers class no longer has a role to play; so, we classify it as redundant. The grey derived component sign in step one becomes a black redundant component sign. In the third and final step, we eliminate the now redundant fathers class from the model.

Figure BG2-46 Making a derived place class redundant




Often, when we are growing a business model, we construct classes that are logically dependent on tuples classes. These normally serve a purpose during the early stages. But, in most cases, they are redundant and so do not need to be implemented. As the model matures, we compact it by eliminating the redundant classes.

In this example of the compacting process, we eliminated the fathers class. However, it is sometimes useful to keep a record of redundant classes. Then, we do not eliminate the class but leave it in the model flagged as redundant. It then occupies a kind of limbo, kept in the model for reference purposes only.

# 7 Summary

Compacting the model is an important part of business modelling, and generalising a class place's link up the super–sub-class hierarchy is a useful pattern for compacting. The other patterns we looked at are also useful when business modelling. We will find ourselves (re-)using most of them. As well as constructing object models of useful patterns, thispaper has helped us develop a clear idea of how the object notation captures patterns of business objects, an essential part of good business object modelling.

*MW—The BORO Methodology: Worked Examples* provides us with examples of all these patterns as it demonstratee how the BORO approach re-engineers the entity formats of existing systems into a reference business object ontology.

## 7 Summary


# BORO Working Papers - Bibliography

## The BORO Working Papers

### Volume A
### A—The BORO Approach

#### 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*

### Volume - O
### O—ONTOLOGY Papers

#### Book - OP
#### OP—Ontology: Paradigms

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

### Volume - B
### B—Business Ontology

#### 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*

**Volume - M**
**M—The BORO Re-Engineering Methodology**

**Book - MO**
**MO—The BORO Re-Engineering Methodology: Overview**

MO1—*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?*

# CONSTRUCTING SIGNS FOR BUSINESS OBJECTS' PATTERNS

A-I

## A



## C



## D



## E



## F



## I

## L



## M



## O



## P



## R



## S



## T



## W