

Chapter 5

Complex Objects

We have seen some examples of ontologies, expressed in a simple metamodel derived from the Web Ontology Language OWL. We now look at representing complex objects, objects which are wholes consisting of configurations of parts.

A world of objects

Like humans, systems of interoperating agents on the Semantic Web live in a world of objects. One absolute requirement for successful living is to be able to identify objects. We need to know that the things we see when we wake up in the morning are the same things that we saw before we went to sleep the night before. Many objects are complexes of parts making up a whole. A second very strong requirement is to be able to tell which parts make up which wholes.

As we have seen in Chapter 2, in the human world there are physical objects and objects which are institutional facts. Institutional facts behave very differently from physical objects. Information systems deal almost exclusively with institutional facts. We want to use our intuitions about humans dealing with objects to help us understand how agent programs deal with objects. Our most accessible intuitions are of physical objects. But because physical objects are so different from institutional facts, our intuitions about physical objects are misleading. We need to think about how we as humans deal with institutional facts in order to transfer this experience to the design of information systems.

Imagine that I have lost a bag containing an Australian \$1 coin and my driver's licence. All three are physical objects, but the coin and driver's licence are also institutional facts. Suppose I replace them. Clearly, the new bag is different from the old bag. The new coin has the same value as the old, but is a different coin. The new licence is a different piece of paper, but is the institutional fact different? We need to look a little more closely.

An institutional fact is a record of a speech act. The relevant speech act is a declaration by an agent of the Queensland Department of Motor Transport that I am a competent driver and authorized to drive a motor vehicle on public roads in Queensland. The licence document is a record of this act. But when I ask the Department for a new licence document, they don't have to make another speech act. They don't require that I pass another series of driving tests. What they do is consult their records (these days in their information system), discover that they have a record of my being authorized to drive in Queensland, and issue another licence document. The institutional fact was not lost, only a copy of the record of the speech act. So my new bag contains a new coin but the same licence to drive.

Nearly all institutional facts are like my driver's licence. When a speech act is made, there are generally multiple copies made, and if one copy is lost it is generally possible to consult other copies and make new ones. Think of citizenship, as recorded in a birth certificate or a naturalization certificate. Think of the institutional fact of being a student enrolled at a particular university, or of being a graduate with a particular degree, or the record of having purchased something, or of having made a payment on your credit card account. The only way these institutional facts can be destroyed is by all copies of the record of the speech act being destroyed. This does indeed sometimes happen, but the practice of keeping multiple records insures against loss of institutional facts. Database management systems have elaborate procedures for backup and restore partly for this reason.

Notice that unlike the licence to drive, the \$1 coin, an institutional fact, was lost along with the bag. This tells us that there is more to institutional facts than that discussed in the previous paragraph. To tell a more complete story, however, would take us too far from the point. The objects in an ontology behave much like the licence to drive, so we will develop the example further.

Our two key problems are how we can identify an object and how we can tell which parts belong to which whole. So how do we identify my licence to drive?

The Queensland Department of Motor Transport, who made the speech act creating the institutional fact, identifies the licence to drive by its licence number. However, when I go to get a new copy of the licence I may not remember the licence number. The Department recognizes this possibility, so makes use of their practice of issuing a licence to only one person and only one licence to a person, by recording also my identity, using my name and date of birth. It gets these of course from documents I

present at the time of my original licence issue. The documents are further institutional facts, records of the speech act in which I was named. So if I can present documents verifying my identity, the Department can search its records and find its record of my licence to drive and from this issue me a new copy. Their database system will represent the licence number as a primary key and my name and date of birth as a secondary key.

A licence to drive is not a simple object, but composed of several parts. To see this, consider the process in which the institutional fact was created. First, a licence number is created by some process. Second, details of the type of licence are entered. Third, my name and date of birth are extracted from my identity documents. Fourth, my address is extracted from documents verifying my address. Fifth, some identifying physical characteristics are recorded. Finally, because there are many circumstances in which officials must confirm that I am the person to whom the licence was issued and no other identity documents may be available, the Department takes a photograph and a sample of my signature and adds these to the record. A driver's licence is a complex object.

There are millions of driver's licences issued by the Queensland Department of Motor Transport. Each licence has several parts, so there are tens of millions of parts. How do we know which parts make up which wholes? To see this, we need to look at the physical objects constituting records of the licence. One such object is the driver's licence document issued by the Department for me to carry. Here, all the parts are assembled into a single card sealed in plastic. So the parts making up a licence are held together by all being in the same container.

But the Department also maintains the record, in a very different form. They keep the record in a database system. One record type in the system will contain all the textual information in separate fields. So these parts are held together in a similar way to the paper document, by all being fields in the same database record. But the photograph and signature are images. These may be stored in different databases, as records one of whose parts is a field containing the licence number. These parts are connected to the textual record by all three records containing the same licence number. In database terminology the three records are held together by foreign key dependencies.

Ontologies versus models

Of course individual information systems as well as ontologies represent objects. The examples in Chapter 4 show that an ontology looks very much like a conceptual model as has been used for many years in designing information systems. In fact, the two are closely related, but not identical. A conceptual model relates to a particular information system, while as we have seen an ontology is a representation of a world external to any particular system, usually shared by a community of interoperating information systems.

Further, a conceptual model is a specification, with the detailed content and representation of individuals left as an implementation decision. Therefore two different implementations of the same conceptual model may be quite different. Think of a relational database implementation versus an object-oriented database implementation. In particular, think of difference in the way a driver's licence is represented as a licence document and in the Department of Motor Transport's database.

Since an ontology must facilitate interoperation, it must include specification of individuals sufficiently concrete that two different systems will be able to represent them in the same way, at least externally. In particular, this includes naming conventions and identification schemes. This is why a driver's licence is identified by both licence number and by the identity of the person to whom it is issued. It is further why the person to whom the licence is issued is identified in three different ways. In our examples of Chapter 4, specification of individuals was central to the Z39.50 system, representations of individuals were central to the Periodic Table, and the SNOMED system with several non-exhaustive facets requires the representation of individuals independent of the classification systems.

Internal representation is of course still an internal design decision. Different systems will implement the identification in different ways, so long as externally they are compatible. So if the ontology uses a relational data model, an object-oriented implementation must provide views to represent the internal structure of objects as tuples in tables linked by foreign keys. If a player has its internal system structured differently from the ontology, it must store the tables giving the correspondences between internal and external identifiers and naming schemes. For example, in a bibliographic application, the internal representation of citations and bibliographic references may be in IEEE format, different from the ontology which might be in ACM format.

So an ontology is generally concerned only with the shared objects and not with the internal workings of any of the interoperating systems, so has different content. As we have repeatedly seen, ontologies are largely about social reality, representations of institutional facts. Different players are empowered to make the speech acts creating different classes of institutional facts. Much of what goes on in the participating systems is *epistemological*, asking for and receiving information about a changing social reality (recall the distinction between performative and informative speech acts in Chapter 2). Since the participating systems communicate generally solely by exchange of messages, the classes and *choreography* (which messages can and can not be sent in particular states of complex systems of exchanges) of the messages is also part of the ontology. Although an ontology and a model are managed by similar technologies and have similar representations, they deal with different issues.

Further, it is common for ontologies to be assembled at least partially from standard components. The Z39.50 ontology of Chapter 4 is a generic query system which can be applied in many applications, each of which has its own set of use attributes. The EDI system sketched in Chapter 1 is a component of very many electronic commerce exchanges. The SIC classification of Chapter 4 is very widely used. Assembling an ontology partly from standard components introduces a number of problems not normally encountered when building data models. We will see other standard components when we consider RDF and OWL in Chapters 10 and 11. Chapter 15 describes some of problems in serving an ontology with imported components.

Complex objects

Our information systems are often about things which have a complex structure. To take a more extensive example than a driver's licence, for many purposes an aircraft (referred to below as aircraft A) is a complex thing. To its manufacturer's parts management system, aircraft A appears as a bill of materials, represented as a whole/part structure. To the airline using it, aircraft A appears as a collection of seats, represented as a set/instance structure. To the manufacturer's production scheduling department, aircraft A appears as a process and a set of completions of process steps (the parts are the various stages of production). To a safety inspection system, aircraft A appears as an ordinary physical object together with a set of properties.

Aircraft A is the same thing in all cases. Further, these various views of aircraft A interact as the respective information systems interoperate. Person P may wish to book a seat on an aircraft which has been recently inspected, which was manufactured in a given time period and has a particular type of avionics system. Perhaps person P was the engineer who designed the system and supervised its manufacture during the given time period, and wants to feel how it works in routine use when recently tuned according to specifications.

Each of the information systems dealing with aircraft A represents it as a collection of fragments. In the same way that we need to know how the parts of a licence to drive are held together, our agents need to know how these fragments come together to represent the entire aircraft. This is called the problem of *unity*. Our agents also need to be able to identify aircraft A in its various representations. This is called the problem of *identity*.

Complex objects are of course often represented in conceptual models of information systems. For example, Figure 5.1 shows two classes of complex objects modeled in the ER notation, *Event* and *Team*. An instance of *Event* has parts which are instances of *Race*, and an instance of *Team* has parts which are instances of *Competitor*. Relationships among the complex objects are between instances of their parts.

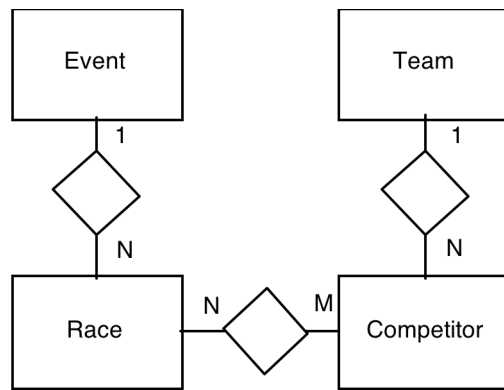


Figure 5.1 Model of two complex object classes

Notice that the ER modeling system does not distinguish between the parts of complex objects and the links between the parts of one and the parts of the other. Further, since it doesn't make sense to have a race without an event nor a competitor without a team, a representation of the model in a relational database schema would often be something like

$$R(\text{EventID}, \text{RaceID}, \text{TeamID}, \text{CompID}) \quad (5.1)$$

In which the complex objects are not explicitly represented as distinguished objects at all. It is left to the user to formulate queries which bring the parts together.

If the conceptual model of Figure 5.1 represents an ontology supporting many interoperating systems, the representation of the objects and their structure becomes more important. If the events are to be held at a particular sports stadium, the stadium management's information system is interested only in the events, their duration (the stadium is booked for an agreed period of time), and their type (the stadium's configuration may depend on what types of events are to be held). They don't need details of the races, nor any information at all about the teams or competitors. Similarly, the accommodation service needs to know about teams (because the team management makes the booking and pays the bills), but perhaps only the number of competitors as a property of the team, rather than the individual athletes. On the other hand, the event organizers need to know the details of the races and of the competitors. All these views need to be integrated for the systems interoperation to work.

We conclude that since an ontology models classes of objects external to any implementation, it is much more valuable to explicitly model the structure of complex objects than it typically is in a single system.

Representation of identity and unity in a single information system

Single system example

The main purpose of this Chapter is to see what we need to do to represent unity and identity in an ontology supporting interoperating information systems. Before we look at these issues in ontology, we will look at how these are represented in the more familiar conceptual models underlying single information systems.

Figure 5.2 shows a fragment of an information system supporting an order-entry/ order fulfilment style application. The model is ER, with the notation showing *many-to-one* relationships as having an arrowhead at the *one* end (*One-to-one* relationships have arrowheads at both ends.) The lower part of the figure is an abstract model of the process from a more general ontology. It shows a distinction between the interaction with the customer (*Transaction*) and the processes that the organisation must undertake to fulfil the order (*Activity*). The notational convention is that the entity type at the arrowhead end of a relationship is mandatory, while the entity type at the undecorated end is optional. For example, in order for there to be a valid instance of *Purchase Transaction*, there must already be a valid instance of *Customer* for it to be related to. However, there can be instances of *Customer* associated with no instance of *Purchase Transaction*. We will say that *Purchase Transaction depends on Customer*.

From the Figure, we see that *Purchase Transaction* depends on *Customer* and *Product*, while *Activity* depends on *Purchase Transaction* and *Product*. In terms of Searle's framework of institutional facts

(see Chapter 2), the purchase transaction between an instance of *Customer* and the supplier in respect of a instance of *Product* counts as the customer buying the product in the context including: the identifier of the customer is an instance of *Customer* in the supplier's database, as is the identifier of the instance of *Product*; and that the customer instance is not barred from participating because of poor credit; and that the instance of *Product* refers to products which are in stock and to which the supplier holds title. The context for making a speech act recorded in *Purchase Transaction* includes the existence of valid instances of *Customer* and *Product*.

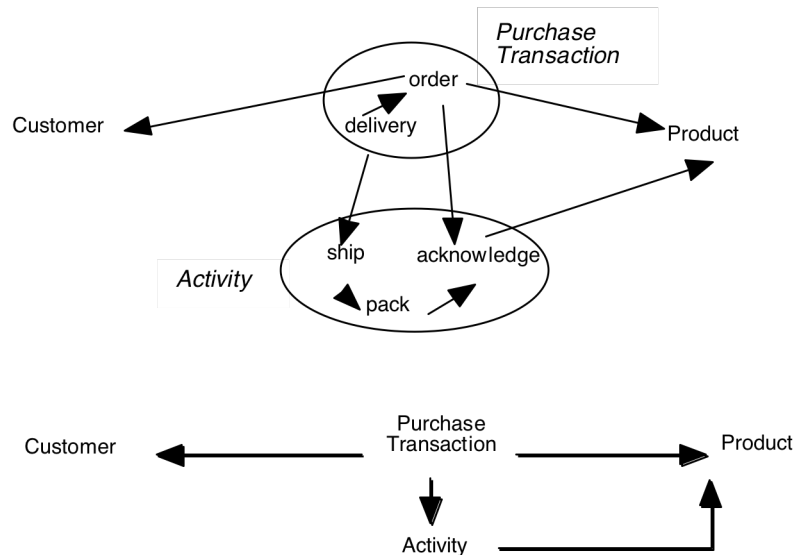


Figure 5.2: A fragment of a refinement of an order-processing system. Upper part shows decomposition into parts of *Purchase Transaction* and *Activity* from lower part

The upper part of the Figure shows a refinement of the lower model, showing both *Purchase Transaction* and *Activity* as processes with parts. This is a fragment of a more specific ontology. The more general *Purchase Transaction* is implemented in the refinement as an *Order* followed by a *Delivery*, while the more general *Activity* process is implemented by first acknowledging the order (*Acknowledgement*), then packing it (*Pack*), then shipping it to the customer (*Ship*). One of the constraints on the refinement of the more general ontology is that each object in the more specific ontology must behave according to the pattern of the more general object it refines. So the behavior of the parts of *Purchase Transaction* must refine the behavior of the whole, as must the behavior of the parts of *Activity*.

Relationships can also be refined into parts. The relationships between *Order* and *Acknowledge*, and between *Delivery* and *Ship*, are both refinements of the relationship between *Purchase Transaction* and *Activity*.

This refinement of classes and properties illustrates that institutional facts can be viewed at coarser or finer granularities, depending on the purposes of the viewer, underscoring the importance of the whole-part relationship.

Axioms for identity and unity

We begin our investigation with formal criteria for unity and identity developed in the OntoClean project, and by seeing how these formal criteria apply.

As we will see below, there are some kinds of object to which one or the other of identity and unity do not apply. However, we will begin with objects to which both apply. The OntoClean axioms for unity are developed using theory of wholes and parts, called mereology³. A whole is a collection of parts, and we say

$$P(x, y) \text{ is true if } x \text{ is a part of } y \quad (5.2)$$

³ See Mereology in Explanations, Appendix C

An object x is an integrated whole if it has a division into parts that is a closed system under a suitable equivalence relation⁴ *sameWhole* called the *unifying relation*. The axiom given is

$$\forall y(P(y, x) \rightarrow \forall z(P(z, x) \leftrightarrow \text{sameWhole}(z, y)) \quad (5.3)$$

That is to say, every part of x in a certain division is related by the unifying relation to every other part in that division, and to nothing else. In our driver's licence example, the unifying relation for the paper licence record is "being present on the same card", while in the database record, the unifying relation for the parts in different tables is "dependent on a particular licence number". All and only the parts for a given licence satisfy these relations. If the whole is an html form $f.html$ and the parts are the individual fields, then the unifying relation for field x is "in file $f.html$ ". All and only the fields in $f.html$ are parts of that form. If the whole is a student's academic record, then all of its parts have the student identifier as a key or foreign key. The unifying relation for the academic record for student y is "record having key or foreign key value y ".

OntoClean also has an identity axiom which requires another equivalence relation called an *identifying relation*. A property R carries an identity criterion iff

$$R(x) \text{ and } R(y) \leftrightarrow x = y \quad (5.4)$$

That is to say if x and y have the same property R , they are the same object. For example, an object has the property *time interval* with three attributes, a starting time t_s , and ending time t_e and a duration $d = t_e - t_s$. One identity relation is *same-starting-time-and-same-ending-time*, which would be used for say timetabling classes. Another identity relation is *same-duration*, which would be used for say cost estimation.

Our intent in this example is to see how these ideas apply to the problem of interoperating information systems. The variables in the preceding formulas therefore are intended to be instantiated by various business objects handled by the information systems. The business objects are all either institutional facts or brute facts associated with institutional facts, therefore the classes we are thinking about are generally representations (brute facts which are records of institutional facts).

A key characteristic of the representations we are going to deal with is that they can be reproduced, so that a paper form is equivalent to a filled-in screen form is equivalent to a pattern of magnetic domains on a computer disk. This raises some subtle issues. In the non-computerised world, the representations of some institutional facts cannot be copied. For example, a copy of a cheque is not a cheque. A copy of a banknote is a counterfeit, so not money. A copy of a passport is not a passport, but can be evidence that a passport exists.

In information systems, the problem is often to restrict copying. Almost any representation in an information system can be copied from keyboard to screen to memory traces in a computer to magnetic domains on disk to printer and so on, within a restricted domain. Some representations, for example product catalogues, can be copied very widely. This is one of the reasons why explicit representations of identity are so important in information systems – it is essential to prevent the proliferation of copies from confusing the institutional facts being created and stored in the systems.

More deeply, the institutional facts exist only in their representations, so a complex fact will exist only as a complex of related representations. This is why we need to pay attention to unity and identity.

Application to the example – individual objects

We first look at a representation of a single class (entity type in ER terminology) which is the source of no many-to-one relationships, in the example of Figure 5.2 *Customer* and *Product*. (We will call these universal targets *independent classes*.) The semantics of the ER model are based on set theory, so the principal interpretation of the representation of the class is as a set of instances. An instance is represented as a tuple of atoms, called *attribute values*.

The tuple of values can be constructed in a number of ways. One way is to store the tuple directly, in which case the unifying relationship is said to be lexical rather than logical. That is to say, unifying relationship for the tuple of attribute values is "being present in the same tuple". We have already encountered this kind of unifying relation, in the Department of Motor Transport's tables from which the licence is reconstructed. The paper copy of the driver's licence also has a lexical unifying relation, namely "being present on the same piece of paper".

⁴ See Equivalence Relation in Explanations, Appendix C

Another way to represent a tuple of values is to construct the tuple as a view (a query, which is equivalent to a logical formula). The assembly of records from different tables to reconstruct our driver's licence is an example. This is a logical construction, but still based on the most primitive tuples which associate the identifier of the object with the institutional fact represented by a particular attribute value.

So the unifying relationship for a representation of an institutional fact is nearly always at least partly lexical.

In database theory, an instance of a class is identified by a subset of attributes, whose pattern of values is unique for each tuple (called a *candidate key*). It is very common for the system to be designed with an attribute intended as a candidate key, such as *driver's licence number* or *product number* or *customer identifier*. This attribute is generally called a *primary key*. So in the case of *Product*, an identifying relation can often be represented logically as

$$\text{same-object}(x, y) =_{\text{df}} \text{product number}(x) = \text{product number}(y) \quad (5.5)$$

Remember that we are referring to the business object, which is characterised by its representation.

However, the identifying relation (5.5) is not sufficient. It works well for the internal working of the organisation, but is not generally a satisfactory way of representing identity outside the particular organisational context, in the same way that the driver's licence number is not a sufficient identifier for replacing the licence document. Just as we were looking to replace the driver's licence issued to me and presented my personal identification, a customer is looking for citric acid 99.9% purity in 200 litre drums, not product CA999-200. A customer knows their own name, address, and other details, not generally their customer number. Even though the product identifier is in practice used as the primary key, there must always be a candidate key composed of attributes whose values are known by all parties in potential interactions with the information system.

There could be a number of such candidate keys. For example, in the United States an organisation has built an exchange for purchase of electronic parts. In this system, the electronic parts have besides their distributor-specific product identifier a global identifier called *Federal Supply Clauses* which can be used by agencies of the United States Government, but not by other purchasers. So the identifying relationship depends on context.

OntoClean uses the terminology that a class has a property that *carries* identity or unity if all instances of that class have values for the property which can be used as the basis for an identity or unity relation, respectively. As we have seen generally in previous chapters and will see more formally in the next Chapter, classes generally occur in subsumption hierarchies, where each more specific level can introduce new properties. The class at which a property is first introduced is said to *supply* the property. If the property is the basis of an identity or unity relation, then the class is said to *supply* identity or unity, respectively.

Objects that are classes

Our discussion of products raises a significant point. What the information systems designer thinks of as an instance can be from the perspective of the ontology a class, not an instance. The information system designer's concept of an instance of *Customer* is also an instance in the ontology, but the same is not true of *Product*. In particular, product number CA999-200 or "citric acid 99.9% purity in 200 litre drums" both identify product classes rather than instances. There can be an amount of the product in various places in the warehouse, in various stages of manufacture, and in various shipments to customers, not to mention waste or spoilage. Our identity criteria as so far presented do not go deep enough.

In some cases individual instances of products are routinely identified. A car or a computer or a piece of consumer electronics normally has a serial number plate on it, which identifies it as an instance of that class. This identifying plate is often the only difference between otherwise indiscernible objects of the same product number.

The citric acid example isn't like that, though. One need for identifying an instance of this product class is to identify a lot of the product subject to a particular transaction or activity. In this business context, the particular lot of the product which the transaction is about is determined quite late in the process, at the time of shipment. Often it is not determined until a particular collection of drums is loaded onto a truck. The drums may not be distinguishable one from another - their only labeling may be with the product class. So the particular instance of the product in this case would seem to be "the

drums loaded onto truck VVV1234 for delivery on 20 March, 2002 under delivery manifest number 77883322 to Acme Ltd".

However, another need is to identify the product in inventory at a particular time. Some products are produced in identified batches (eg pharmaceuticals) or have use-by dates, which can be identified as wholes of an amount of matter. However, many products are simply flows into and out of inventory. The product may be packaged in one form or another like our drums of citric acid, or may be stored in bulk (oil, say).

Note that unity can appear at some levels of granularity and disappear at others. Following an example of apples in the spirit of the purchasing application, the product class may be "Apple – Granny Smith in cases of 80". The cases carry no identification. The cases carry unity, because we can tell what is in which case, but not identity. So at the level of maintenance of inventory, shipping and receiving, the product is also considered to be bulk. However, a particular customer (a restaurant, say), may have ordered a small number of cases in a particular transaction. The cases may be able to be identified now (by location in the truck, say) during the delivery, but then go into the restaurant's cool store and merge with other cases in bulk again. When a case is opened, the apples in it are not identified, but an apple served to a particular customer of the restaurant may be identified, again by what amounts to packaging. To this customer, the apple carries identity not only of class, but also individuality, since it is the (only) apple served them at that time. It also has a *topological unity* (its parts are all contiguous), so is an individual in this context.

With the apple example, we have identity that depends very much on context. The last version had the apple served to a particular customer at a particular time. When we are engaged in an activity we often identify objects with reference to the engagement: my apple, the apple I just served, that apple. This sort of identification is called *indexical*. When we are interacting with an information system, the objects are often identified only by class. For example, in an e-commerce application, we might have a command "move product to shopping cart". This identifies indexically the product we have on the screen and the shopping cart associated with our account. In these situations, the system will have an alternate identification in a broader context. The product is identified by product number, say, and the shopping cart by our account identifier, so that internally the command is "move product P to shopping cart S", but we see only the command with indexically identified objects. Drag and drop commands are indexical.

This discussion of objects missing one of identity or unity gives us a taxonomy. In terms of OntoClean, the class *Customer* is a collection of individuals. (An *individual* has both unity and identity). The class *Product* is a collection of classes, which may or may not consist of individuals. The individuals may be distinguished only by name. But we have seen that the classes in *Product* often carry identity (as an instance of the identified class), but not unity; or unity but not identity (objects are anonymous) so that the lots of product are not able to be identified other than by the container they may be in. We can call these kinds of object (which carry class identity but not unity or unity but not identity) *bulk* objects as distinguished from *countable* objects (which carry both identity and unity so yield individuals).

We return to bulk objects in Chapter 12.

Dependent classes

Dependent classes like *Purchase Transaction* pose a somewhat more complex identity and unity problem. An instance of a dependent class requires the prior existence of instances of the classes depended on (in this case *Customer* and *Product*). The representation of objects belonging to dependent classes will often include identification of the objects depended on. Semantically, the independent classes define the most stable aspects of the operation since they must have instances in order for the other classes to have instances. The dependent classes represent records of the more dynamic aspects of the operation. The institutional facts referred to by the populations of the dependent classes have as some of their properties the identifiers of the records of other institutional facts. These other institutional facts must exist for the speech act creating the present institutional fact to be validly framed, so these records are important properties. A driver's licence requires identity documents. An invoice may require a purchase order, a delivery advice and a delivery acknowledgement.

So, although it is common for invoices to be identified internally by say an invoice number, a convenient way to obtain an identifying relation for wider contexts is to build on the identifying relations for the objects represented by the relevant independent classes. (This is essentially the formal mechanism of *weak entities* in ER modelling.)

Note, by the way, that in our example, *Purchase Transaction* is dependent on both *Product* and *Customer*. There is a third dependency, on *Activity*, but this dependency is subject to an additional integrity constraint whereby *Purchase Transaction* is transitively also dependent on *Product*, but the product instance reached in either path must be the same. To complete a purchase transaction an activity must have already been completed. The purchase transaction and activity both depend on instances of *Product*, but the instances must be the same. The products shipped must be the same as the products packed. This third dependency does not add anything to the first two.

If each of the relationships between *Purchase Transaction* and *Product* and respectively *Customer* were one-to-one, then the concatenation of the identifiers of *Product* and *Customer* appropriate to the context would be sufficient to identify an instance of *Transaction*. In general, however, the relationships are many-to-one, so that for each pair of *Customer* and *Purchase Transaction* instances there can be many instances of *Transaction*. An additional local candidate key is also needed. Internally one might use a sequence number and externally for example *Date*.

Identification of parts

Figure 5.2 shows in its upper half a refinement of the model in the lower half. The class *Purchase Transaction* has been refined into two parts, *order* and *delivery*; while the class *Activity* has been refined into three, *acknowledge*, *pack* and *ship*. How do we identify instances of these new classes?

We have looked at identification of instances of *Purchase Transaction* and *Activity*. One general way to identify parts of a whole is to use the identifier of the whole with the addition of a local part identifier, as in weak entities in the ER system. That method won't work in this case, since the whole is not represented. If we think of the lower part as being a representation of an earlier stage of the design and the upper part as being a representation of a later stage, then in developing the conceptual model of the information system, the lower half is replaced by the upper half when the system design reaches that state of refinement. When we are thinking about *order* and *delivery* there is no longer any entity *Purchase Transaction*.

In operation, the system will generate linked instances of each of the part classes. If we happen to want to look at combinations of instances of *order* and *delivery* as instances of the more general *Purchase Transaction*, we will create them by a query. So the whole only comes into existence when all of its parts do. Further, we might be interested in partially completed wholes. In these cases, only some of the parts exist, so the whole does not exist at all. So we can't identify the parts with reference to the whole.

What we can do is take advantage of the fact that the instances of the part classes are linked by foreign keys. Every instance of *delivery* is linked to an instance of *order*, every instance of *pack* is linked to an instance of *acknowledge*, and every instance of *ship* is linked to an instance of *pack*, which is in turn linked to an instance of *acknowledge*. If we identify the instances of the parts in some way, we can derive an identification of the whole. Every instance of *Purchase Transaction* includes an instance of *order*, as does every instance of partially completed *Purchase Transaction*. Similarly, every instance of *Activity* includes an instance of *acknowledgement*, as does every instance of partially completed *Activity*. So one way to identify the whole is by one of its parts which always exists: *Purchase Transaction* by *order* and *Activity* by *acknowledgement*. Identifying a whole by one of its parts is called *metonymy*.

Metonymy is quite common. News reports of activities of national governments often refer to the country by its capitol, as "Washington said ...", "The response from Tokyo was ...". Governments are often identified by the leader, as "the Bush administration", "the Blair government".

Application to the example – classes

Individual objects have now been identified as instances of their respective classes. We have seen that some individuals can also be seen as classes. This leads us to ask how the classes are identified in general. Classes represent sets of business objects, and a set of business objects is often the subject of discourse.

Institutional facts are completely characterized by representations of their records, so have a limited set of properties. In particular, there is no way to distinguish an invoice from a credit note without the record of the class of institutional fact. So it is hard to see that the class of invoices is anything other than the subclass of institutional fact whose representations contain an attribute called "type" which has the value "invoice".

So a class of institutional fact is represented by a class, which formally is a set whose members are instances, but the normal practice is to define the class *lexically* (by form, in this case as members of a named set) rather than *logically* (as subsets of a superset satisfying a predicate expressed in logic or SQL). The same class can have many different populations – indeed much of the code in an information system is devoted to updating the populations of classes. That is to say, much of the code is devoted to updating the representations of members of sets of business objects. So we can think of a class as a sort of container.

In practice, systems designers identify classes in several different ways. If the primary design vehicle is the ER model, the class (entity) is represented simply as a name on a graphical element, which has graphical connections to representations of names of attribute value sets and graphical representations of relationships with other classes. Often, the representation is a translation of the conceptual representation to a relational database table scheme, where the class name becomes the table name, and the relationships and attributes column names, the former labeled foreign keys.

Other ways also are used. Sometimes several classes are combined, possibly redundantly, in a single table (universal relation) – often as a view. In this case, to identify an instance of the representation of a class, the programmer needs to know which columns contain the preferred candidate key and which columns contain the attributes and foreign keys associated with that class. It is also possible to represent a conceptual model in a single table with four columns: *class*, *tuple*, *attribute*, *value*. Each row of this table contains a single value of a single attribute of a single tuple instance of a single class. (The tuple in this case is often identified by an arbitrary number which is not itself the value of an attribute.) Here, the programmer needs to know the name of the class in order to retrieve its instances.

So in practice the unifying relation for a class is something like an SQL statement (the statement, not the result), and the identifying relation is either the name of the class or the names of columns which are known by the programmer to constitute the representation of the class. Classes are therefore typically identified lexically.

Application to the example – the system as a whole

Finally, we take another step upwards in scale and look at the system as a whole. This system is a single system, which could be one of many operated by the same organization, so would have a name, say *Acme Corp Ordering System* (henceforth ACOS).

Either system in Figure 5.2 has a unifying relation, namely “is a part of ACOS”, but that unifying relation is not represented in the structure. The unity of the system is maintained by its operational environment: it is located on a particular cluster of servers operated by a particular company, compiled from modules held under a certain version in a certain repository, and so on. As an isolated system, it has no practical need for its unity to be represented internally. Both its unity and identity are represented indexically.

However, it is not difficult to represent a unifying relation. One way is to supply an additional class *ID* called the identifying class, with exactly one instance and to require a (unique) many-to-one relationship *id* from each class in the system to the identifying class. We can parameterise the identifying class by the name of the system, represented as ID_x . A class used as an identifying class for an entire system in this way is called a *terminal object*. The unifying relation of the refined model is a refinement of the unifying relation of the abstract model. In this case, we have

$$\text{isAComponentOfSystem}_x(z, y) = \exists z, y (\text{id}(z, ID_x) \text{ and } \text{id}(y, ID_x)) \quad (5.6)$$

where the unifying relation is also parameterized by the name of the system.

Given the unifying relation (5.6), we can represent an identity criterion for “is system ACOS” as

$$\exists x ID_x \text{ is a terminal object for system ACOS} \quad (5.7)$$

Because they are both parameterized by the name of the system, the unifying and identifying relations are both lexical rather than logical.

This is all a bit academic, of course, since we have already established that in the isolated system we have taken ACOS to be, there is no practical need to represent either unity or identity. However, as we see below, unity and identity become very important when we start seeing ACOS as a member of a community of interoperating systems.

Interoperating systems

Now that we have come to an understanding of the issues of identity and unity in a familiar single-system environment, we now turn to the less familiar environment of interoperating systems where there are additional things to consider. Remember we are thinking about the agent programs that execute the interoperations. This is why in the previous section we looked at how to identify an entire system. Each agent must identify itself to the other, so must know its own identity as well as the identity of the other.

An order processing system would naturally interoperate in an electronic commerce environment with a purchasing system operated by another organization. Further, neither system would generally expose itself in its entirety to the other. Each system in the interoperation would see something like in Figure 5.3. As with Figure 5.2, the upper system is a refinement of the lower. The lower system shows a purchasing system (left) and a supplier's order processing system (right). Both systems interoperate via *Transaction*. If the underlying system has an identifying terminal object, then the view can have one, too, so that the view is unified and identified in the same way as the underlying system. (If necessary, the two can be differentiated by differentiating the terminal objects.)

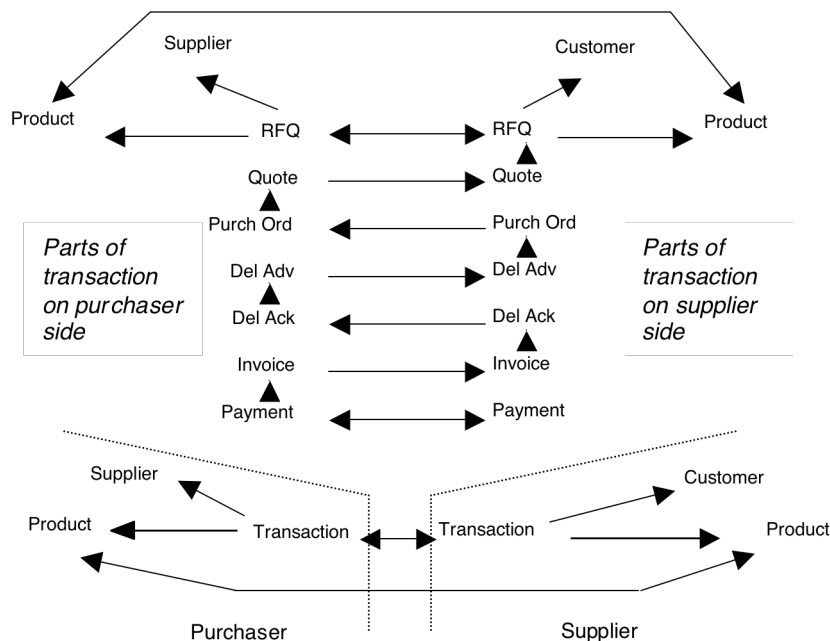


Figure 5.3: Purchasing and Supplier system interoperating
Lower by *Transaction*. Upper by *Transaction* refined into parts

Concentrating on the lower (abstract) system, note that an instance of *Purchase Transaction* is dependent on both *Supplier* and *Product* on the Purchaser side, and also on *Customer* and *Product* on the Supplier side. (Note that *Purchase Transaction* here is not the same as in Figure 5.2, but is closely related.) We focus first on the relationships involving *Product* on each side.

Each of the Purchaser and Supplier system has a class called *Product*. They are not the same, as the Purchaser can buy from many suppliers, and the Supplier can sell to many purchasers. But they will have a common subclass, consisting of the products that purchaser buys from that supplier.

Product in each system is an independent class. In our (simplified) system, a transaction involves an instance of *Product* on the Supplier side and an instance of *Product* on the Purchaser side. In fact, it is physically the same product on each side, moving say from one warehouse to another. There is therefore an issue of identity – the identifying relations for *Product* on each side need to be correlated, generally by a one-to-one mapping. The mapping need be neither injective nor surjective – one company may purchase only some its products from a given supplier, and purchase only some of the products offered by any given supplier. This mapping implicitly provides an extensional unifying relation for the subclass of products shared between a given customer and supplier. The correspondence between product identifiers is an identifying relation for that subclass.

This method of federating product catalogs by pair-wise correspondences works well for one-on-one interoperations. It can in principle be extended to an exchange. Each player makes a correspondence between their own catalog and the catalogs of each of the other players with whom they interoperate, and the exchange's product catalog can be constructed from the transitive closure of each of the pairwise correspondences. When two players interoperate for the first time, a particular correspondence may be able to be derived if each of the players interoperates transitively with others, so there is some economy gained from the fact that there are many players in the exchange. The transitive closure of the correspondences functions as an identifying relation for the product on the exchange.

But the accumulation of pairwise correspondences is not very satisfactory at the exchange level. There may be thousands of players. An individual player may have to establish pairwise correspondences many times before gaining benefits from transitive closure. Worse, there is no reliable way to be sure that all products are in fact uniquely identified. There may not have been enough pairwise correspondences established that the transitive closure includes every player who deals in that product. A new product which has never been traded is just about invisible.

For this reason, exchanges generally will establish a priori an exchange-level system of identifying objects traded. In this way a player needs only establish a correspondence between their own catalog identifier and the exchange's identifier. This is what ISBN numbers and bar codes on products are all about. An exchange-wide or industry-wide system of identifiers takes work to establish and continual coordination to maintain. The system is a system of institutional facts and needs bodies authorized to make the corresponding speech acts.

The relationships involving *Supplier* and *Customer* raise additional issues. It is normal for the Purchaser system to have a class *Supplier* (as in Figure 5.2), and for the Supplier system to have an analogous class *Customer*. However, what is an instance of *Supplier* for the Purchaser is the entire system of the Supplier, and what is an instance of *Customer* for the Supplier is the entire system of the Purchaser. As we have seen in the previous section, in isolated systems neither the unifying nor identifying relations for the whole system are generally represented. For interoperation, however, they must be. Each system must have a data structure which allows them to tie together their identities with various partners. Anthropomorphically, we might think of this as a primitive notion of "self". In the previous section, we used a terminal object for this purpose. So we have that the terminal object for each system must be mapped into an instance of a class of the other, in order for the two systems to interoperate. My system must know how the other system identifies me, and the other system must know how my system identifies it.

We move now to the refinement of the *Purchase Transaction* classes in the upper part of Figure 5.3. A business transaction is an institutional fact normally created in a series of speech acts organised into a process, using a semantic protocol like one of the EDI standards. This means that to carry out the interoperation the abstract system needs to be refined, so that *Purchase Transaction* has several parts. In the example of Figure 5.3, we have the interaction as proceeding from a request for quotation (*RFQ*) issued by the purchaser, through *Quote* by the supplier, *Purchase Order* by the purchaser, *Delivery Advice* by the supplier, *Delivery Acknowledgement* by the purchaser, *Invoice* by the supplier to *Payment* by the purchaser. In practice, of course, the interaction can be much more complex, involving many more exchanges of different classes, and the sequence need not be linear.

In the example, each side keeps copies of all messages, very likely linked to other aspects of their respective systems outside the view. Each message participates in a many-to-one relationship with an earlier message. The first message (*RFQ*) is shown with a one-to-one relationship between the parties – the relationship from Purchaser to Supplier represents acknowledgement by the supplier that a communication sequence has been established.

The refinement is conceived of as an articulation of the whole of *Purchase Transaction* into parts. We therefore need to consider the unifying and identifying relations required.

An instantiation of *Purchase Transaction* is as a process, so its parts come into existence one by one over possibly considerable time. (We do not need to take clock time into account, simply sequence.) Further, in the example the whole is not represented in the refinement, only the parts. A whole transaction instance is represented by the assembly of all of its parts. So besides unity and identity, we need to consider existence. We focus first on identity and unity.

Atomic parts (an *atomic part* has no parts itself) of an instance of *Purchase Transaction* are instances of *RFQ*, *Quote*, and so on. The first part to come into existence is *RFQ*, and it can be identified in the same way as we have discussed for the unrefined *Transaction*, as an instance logically by the

relationships with either *Customer* and *Product* or *Supplier* and *Product* for Supplier and Purchaser respectively, together with an agreed disambiguating attribute like *Date*. Since there needs to be agreement between the parties on the identification of *RFQ*, there needs to be an intersystem identity relation, which can be supplied by including all three of *Product*, *Supplier* and *Customer* relationships, taking advantage of the identification of the Purchasing system as an instance of *Customer* and the Supplier system as an instance of *Supplier*. Of course the identity relation for the *RFQ* instance also includes the lexical identification of it as an instance of the *RFQ* class. So we would say that an instance of *RFQ* is identified by the combination of purchaser, supplier, product and date.

The other parts as they come into existence can be identified by their possibly indirect relationship with *RFQ*, if necessary including a further local identifier based on *Date* or *Message-ID* or some other attribute whose scope includes the contexts of both parties.

Dependence on *RFQ* provides a convenient unifying relation for the whole transaction.

We now consider how to identify the whole, assuming we have a complete collection of parts. One obvious way is to employ *metonymy*, using the identifier of *RFQ* as the identifying relation of the whole. However, the various parts of the transaction come into existence over time, and in practice may never come into existence. At any given time, the populations of the classes refining *Purchase Transaction* will contain all sorts of incomplete transactions. Some of these incomplete transactions may be stopped. For example it often occurs that a purchase order is not ever issued in respect of a quote, and it sometimes happens that a customer does not pay. It therefore may suit the organisations to refrain from identifying a whole transaction until a part comes into existence which usually leads to completion, say *Purchase Order*. But of course the identification of a not-completed transaction does not guarantee that it will ever be completed. We will return to this issue in the discussion of subclasses and subsumption in the next Chapter.

Comment on the Examples

Some of the ontologies in Chapter 4 have whole-part relationships:

- Z39.50: in Figure 4.2 the property *in* whose domain is *ResponseRecord* and whose range is *ResultSet* can be interpreted as instances of *ResponseRecord* being wholes with parts instances of *ResultSet*. The semantics are the set-instance relationship. The identifying relation for *ResultSet* is *hasName*. *ResponseRecord* is a dependent class, with identifying relation a combination of *LocalIdentifier* and *hasName* composed with *in*. The unifying relation is that the *ResponseRecords* X and Y are parts of the same whole if $hasName(in(X)) = hasName(in(Y))$.
- Tic-Tac-Toe: in Figure 4.6 the class *Game* can be interpreted as a whole with parts instances of *Cell* via the *has* property. The identifying relation for *Cell* is the conjunction of *atPl*, *atH* and *atV* (instances X and Y of *Cell* are the same if $atPl(X) = atPl(Y)$ and $atH(X) = atH(Y)$ and $atV(X) = atV(Y)$). We noted in Chapter 4 that there is no facility in the ontology to identify instances of *Game*, so there is only one instance Exists G *Game*(G). The unifying relation is the property *has*. If the ontology were modified to include identified instances of *Game*, then *Cell* would become a dependent class in the same way as *ResponseRecord* in Z39.50.

We will return to the part-whole relationship in the examples at the end of the next Chapter, because the other examples are bound up with the class/subclass relationship.

Summary of identity and unity

We now know how to represent complex things in the environment of interoperating information systems. We keep the various aspects of the representation together with unifying relations and keep track of different things with identifying relations. We have seen that it is not always possible to find both a unifying relation and identifying relation, so that some things in the environment are what we might think of as bulk rather than distinguishable objects. We have seen that our information systems deal logically both with definite objects and with classes.