

Improve your Sense of Ownership

Exploring a Design Principle

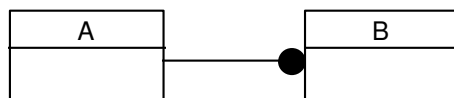
Charles Weir

© Object Designers Limited, August 1995

Finding the Limits

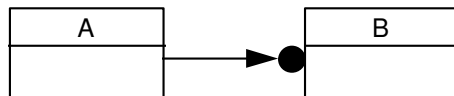
How far does a program object extend? It sounds like a trivial question. Surely an object covers only the memory allocated by the system for its space? A C++ object occupies the space allocated by the 'new' operator; a Smalltalk object the same. However, when we think about what we really mean by 'an object', in most cases we find that the answer is not at all trivial. For most objects are made up of other objects; they *own* other objects. This article will explore this *ownership* in detail.

The notation used here is based on Syntropy's implementation model (see [Cook&Daniels]). The following uses the C++ term "pointer" to mean an object identifier; Smalltalk programmers and others please make the appropriate translations:

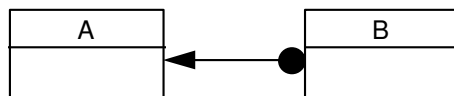


Indicates that for each B, there is exactly one associated A; for each A, there are zero or many Bs

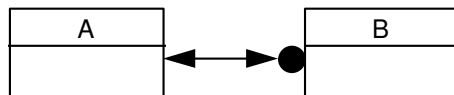
To this we add arrows to show how the associations are actually implemented in code:



An instance of A contains a collection of pointers to B (but a B has no references to any A's)



Each B has a pointer to its corresponding A (but As know nothing of Bs).



Each A has a collection of pointers to B; each B has a collection of pointers to A.

Note: Strictly speaking, these descriptions of the arrows give just one possible implementation of the required visibilities. For example, in the first case, A might contain (in C++) an embedded array of B's, or A might look up its B's in some central location.

Figure 1: A Note on Notation

Consider the Order object in Figure 2 (which uses the notation described in Figure 1). It has associations to each of its OrderItem objects. Each OrderItem then has an association to the ordered Product. It seems reasonable to suppose that the OrderItem objects belong to the Order: they will be stored, copied, and deleted, together. It is unreasonable to say the Product objects belong to the order: copying an Order doesn't increase the number of Products in the system.

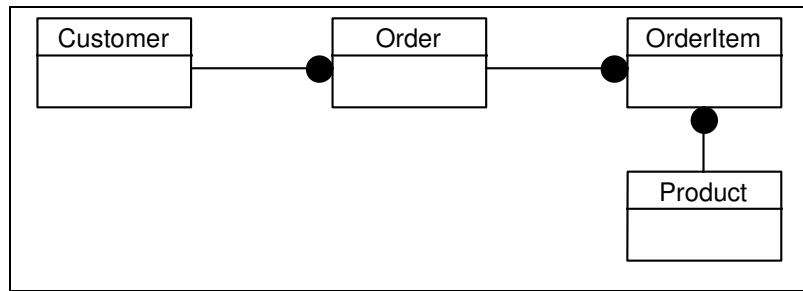


Figure 2: An Order Entry System

This distinction seems fine when we are talking about real-world situations, but does it have any meaning in software? Suppose that the Customer, Order, OrderItem, and Product objects are instances of classes within the memory space of an application. How might it affect the implementation to say that OrderItem is owned by Order?

Perhaps the fundamental point is that ownership defines what happens when we make a copy (clone) of an object. By 'copy', we mean an item with the same data, but a new identity - a conceptual copy. For example, we might want to copy an order to keep a historical record of its state at any given time, or to allow a customer to repeat an Order. A copy of an Order will reference copies of each of the OrderItems; however a copy of an OrderItem will reference the same Product object as the original. So we can use ownership to define what we mean by copying an object; it defines how far an object extends.

Where do we look for the owner of an object? Almost always an owner will correspond to an association, which when navigated, gives one object (or none). For example, in Figure 2, OrderItem could not be the owner of its associated Product, since for a given Product there may be many OrderItems.

If a Customer owns an Order, and an Order owns an OrderItem, does the Customer own the OrderItem? Or, to put it in mathematical terms, is ownership transitive? Returning to the copying idea, we see that the answer must be yes. If a copy of Customer makes a copy of Order, and the Order copy copies the corresponding OrderItem, then copying a Customer does indeed copy its associated OrderItems. Customer does indeed own the OrderItems.

Uses of Ownership

Ownership is useful anywhere where we need to define a limit on the extent of an object. Let's look at some examples.

Memory Management

In languages like C++ with no garbage collection, we must explicitly free memory when we have no further use for it. Thus, for example, when we remove an Order object from memory, we must explicitly remove the OrderItems associated with it. In C++, this happens in the destructor for the Order instance, which deletes the pointers to the OrderItem instances. In terms of ownership, we say that the destructor must delete all the objects currently owned by its instance.

Resource and Association Management

Tidying up memory, though, is just one example of the wider issue of tidying up on deletion. And this applies equally both to systems written in C++, and to those in languages (Smalltalk, Eiffel), which do have automatic garbage collection. We need to tidy up resources and associations too. You didn't think Smalltalk and Eiffel ever needed destructors? Think again!

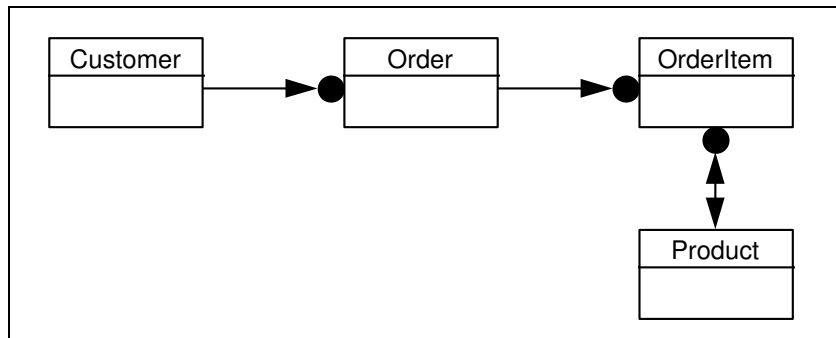


Figure 3: Product implements association to OrderItems

For example, suppose that the association between Product and OrderItem is implemented in both directions, so that each Product has a collection of references (or C++ pointers) to its related OrderItems (see Figure 3). Then, when we remove a Customer from the system, we must also remove all the associations from the corresponding Product objects. Otherwise, in C++ the Products would be left with an invalid pointer; in Smalltalk the corresponding items will not be garbage collected. Does ownership have an answer to this? Well, yes. We can phrase the ownership solution thus: an object has the responsibility to remove all non-owning references to itself when it is destroyed. Thus it is the responsibility of OrderItem to ensure that the collections in Product are tidied up correctly.

Similarly in languages with garbage collection there may well be system resources other than memory associated with an object, for example file handles, window handles, and semaphores. Such objects must ensure that these resources are freed up before they are deleted, and therefore must supply an operation that corresponds to a destructor (file>>close is a typical Smalltalk example). Once again, it is the responsibility of the owner to ensure that such operations are called before the object is discarded.

Making a Copy

We have already looked at copying, and how ownership helps define the meaning of a copy. However the actual algorithm for performing a copy is somewhat more complicated than the earlier discussion may have suggested.

For example, suppose that each Customer object keeps a list of all the OrderItems not yet delivered, as in Figure 4.

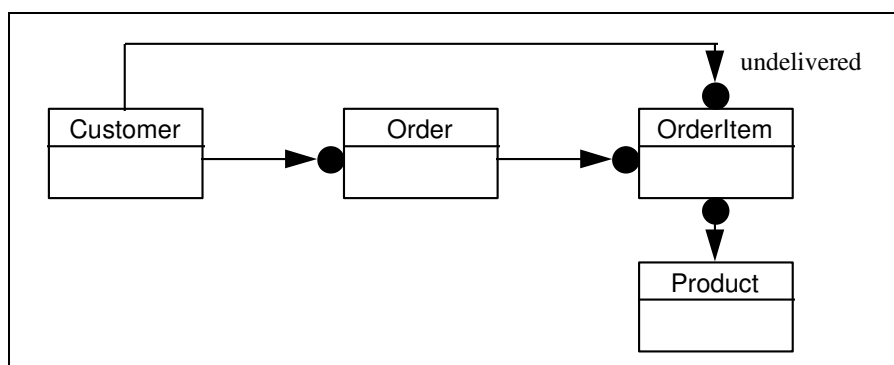


Figure 4: An Order Entry System implementing an 'undelivered' association

There are three different kinds of association on the diagram, as far as ownership is concerned: each has different behaviour on copying.

Type 1 is an *owning* association. The object owned is copied. The association from Customer to Order is an example, so a copy of a Customer makes a copy of its Orders.

Type 2 is a *non-owning* association. The copied object references the same as the original. An example is the association from OrderItem to Product; a copy of an OrderItem references the same Product as the original.

Type 3 is *indirect ownership*. This requires special handling. For example, when copying a Customer we may neither create a new OrderItem for each ‘undelivered’ association, nor reference the original OrderItem; instead we must create all the directly owned copies (of Order objects), and locate the correct items amongst the OrderItems created in this way.

Figure 5 shows an object diagram with a Customer who has made a single Order of one item, not yet delivered; and the result of making a copy of this Customer. In the object diagram, each rounded rectangle represents an instance (not a class), and the lines represent associations. Observe how the three different kinds of association have been duplicated in different ways.

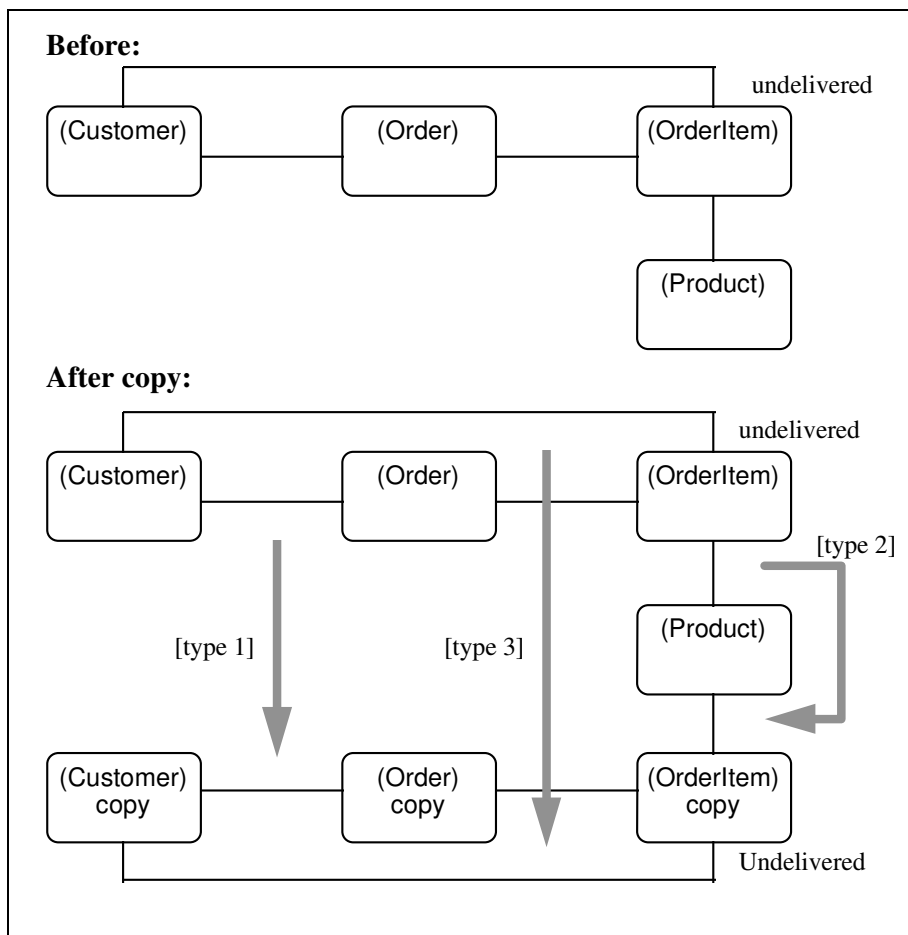


Figure 5: Object Diagram showing the result of copying a Customer.

Persistence

Just as ownership defines how we make a memory copy of an object, so it can also help copy an object to and from persistent store. One can implement a strategy where each object is always instantiated and saved together with all the other objects it owns. For example, to implement the invoice system, we might hold all Products in memory, and simply instantiate a Customer with all its owned objects whenever we need it (and delete it after processing it). However, it must be admitted that this is a relatively simple persistence strategy; for example, no other object may keep a reference to any part of the Customer. A more complete approach would defer loading Order objects (and indeed Product objects) until these were actually needed, and maybe also implement reference counting to decide when to remove each of these objects from memory.

Transactional Locking

When there are multiple threads of control in a system, we need to be able to 'lock' objects within one thread so that other threads must wait before accessing them. This will be increasingly important in future systems, as CORBA, OLE, and their variants allow processes to share objects. So what does it mean, to 'lock' an object? A lock implies some kind of flag (often a semaphore) associated with the object; the thread that updates the object first sets this flag until it has finished its transaction; other threads wait for the flag to reset before they may continue. However when we lock one object, do we need automatically to lock others as well? Here again, the concept of ownership provides a strategy. We can define locking an object to mean preventing access to it and to all the objects it owns. Locking an object locks all of its owned objects too.

This strategy may be too restrictive. For example it might be reasonable to modify a past Order in one thread even while another has the Customer object locked in order to change the Customer address. However ownership is nonetheless a reasonable pessimistic strategy for locking.

Multiple Ownership

Not all objects have a single owner. To take a (somewhat contrived) example, see Figure 6. Here we've replaced the Order object with a number of Invoice objects, representing an invoice sent to the customer.

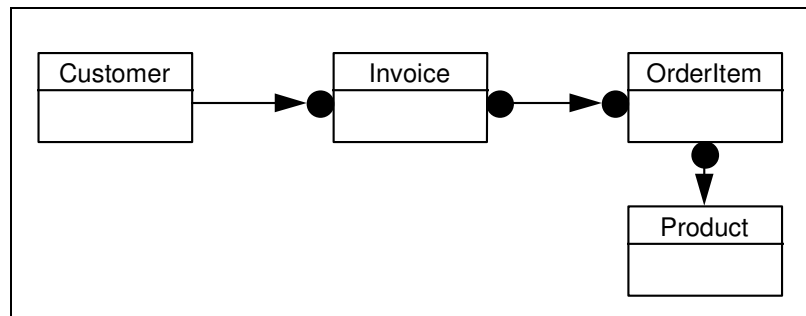


Figure 6: An Order Entry System with copiable Invoices

Each invoice shares the same OrderItems with a number of other Invoices, and we generate a new Invoice to send out by copying a previous one. OrderItems must therefore remain in the system for as long as any related Invoices are present. So in the implementation, each order item must keep a count of the number of Invoices referencing it. When we copy an Invoice, we do not duplicate its OrderItems, but merely increment their reference counts. When an OrderItem's count becomes zero (no Invoices left), the OrderItem goes, removing itself from the corresponding collection in its Product as it does so.

Note that only the association between Invoice and OrderItem contributes to OrderItem's reference count. We might say that reference counting allows an object to have a number of *shared* owners. Referring back to the types of relationship we defined earlier, then, we can add a fourth:

Type 4 is *shared ownership*. When copying the owner objects, we do not copy the items shared, but instead increment their reference counts. The association from Invoice to OrderItem is an example.

Figure 7 shows an object diagram with the result of copying an invoice. The reference count in OrderItem gives the number of Invoices sharing its ownership.

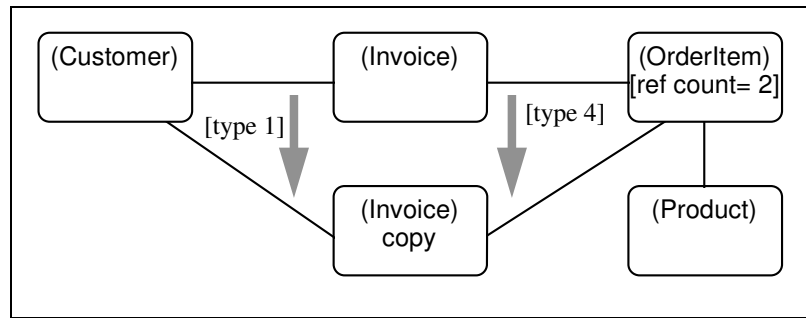


Figure 7: Object Diagram showing the result of copying an Invoice.

Conclusion

Ownership is a design principle. It's a helpful way to deal with particular programming situations, although many parts of an object-oriented system may have no need of the ownership paradigm at all.

However we can identify the following guidelines for where it does apply.

1. Every object has a single owner (or an internal count of the number of shared owners)
2. We can distinguish several kinds of association, with different ownership semantics:
 - Single Ownership
 - Shared Ownership
 - Non-owner associations
 - Indirect Ownership
3. Ownership may be transferred to and between objects; it is important to document this transfer in the code.
4. Ownership provides strategies for:
 - Copying
 - Memory management
 - Resource and association management
 - Database persistence
 - Transactional Locking

Ownership, then, provides a way to analyse aggregations of objects. It specifies behaviour by distinguishing types of association. It gives us a grasp on resources and persistence handling. Ownership is a helpful design principle.

References

- [Cook&Daniels] *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Steve Cook & John Daniels, Prentice Hall 1994, ISBN 0-13-203860-9