

# Frameworks and component-based development

Alan Wills

*Trireme International Ltd*

*24 Windsor Road Manchester M19 2EB*

Email: alan@trireme.com, <http://www.trireme.com>

Phone: +44 161 225 3240, Fax: +44 161 257 3292

---

**Abstract.** *Classes are not the best focus for object-oriented design. The most useful components in a design are frameworks – schemes of interaction between objects. These can be worked on by separate authors, kept in libraries, and recombined to make many different end-products. This paper explores the more formal aspects of this approach, within the context of popular OOA/D notations and programming languages.*

**Keywords.** *Generic classes, re-use, collaborations, component-based development, object-oriented frameworks.*

---

## 1 Objects are not the best focus for design

One of the big motivations for taking up OO design is that it promises some form of re-use. The naïve interpretation of “re-use” is that you can pull pieces out of an old program and install them in a new one; but that doesn’t often work very well. A more mature strategy is to look for similarities in different pieces of design work (perhaps in the same system, or in the same project, or in the same organisation) and to expend some resources on creating (and subsequently refining) a generic piece of design that covers all the similar cases; then it’s likely that subsequent designs will be able to instantiate the generic component. Much experience has shown that this results in faster more maintainable designs, if less efficient end-products.

These generic pieces of design – let’s call them *frameworks* – are rarely about one object. Instead, they are about the relationships and interactions between members of groups of objects. Most of the design patterns discussed in books [1] and bulletin boards are based around frameworks: for example, the “observer” pattern which keeps many views up to date with one subject; or “proxy”, which provides a local representative of a remote object; or any of the more specialised design-ideas that are fitted together to make any system.

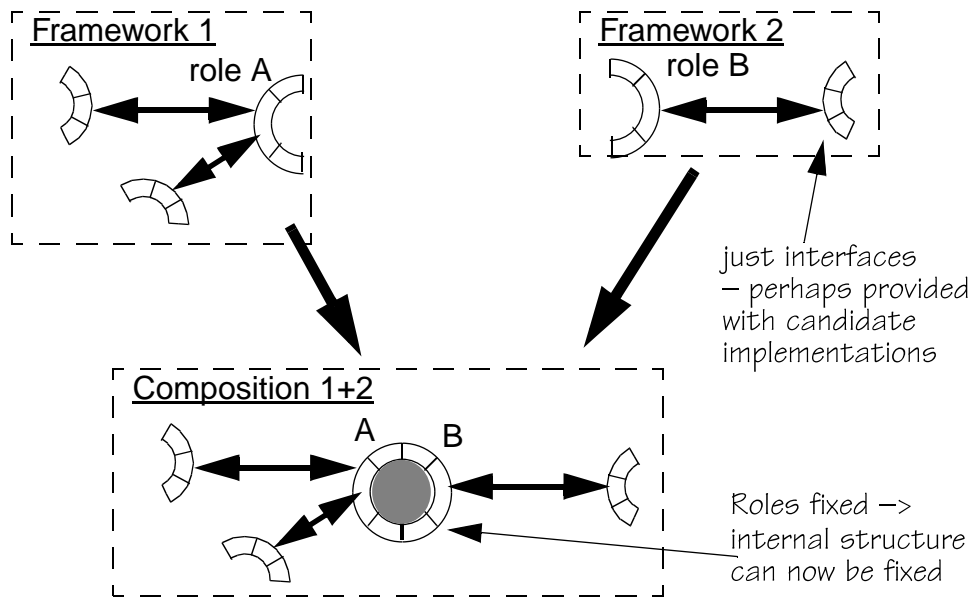
An object in a running system may be a participant in several frameworks. For example, it could be playing the role of the observed subject of one or several observer frameworks; and simultaneously being a proxy. Indeed, neither of these frameworks would be of much use unless they were designed in the assumption that their participants will play other roles.

For each role it plays in the different frameworks of which it is a participant, an object has an interface – the behaviour expected of it within that framework. So when we

**Presented at OO Information Systems, London, 18 December 1996**

Published in Proceedings of OOIS’96, ed D Patel, Y.Sun, D.Patel, [“Springer 1997]

Composing Frameworks



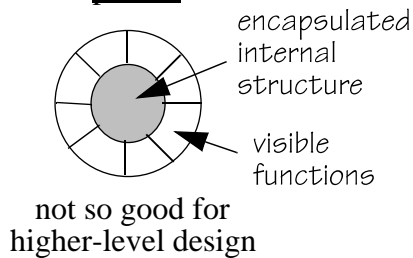
**Figure 2**

design a framework, we discuss an interface for each participant. We know that each will have other interfaces – which will have some effect on its state – but as designers of the framework, we cannot anticipate what they will be.

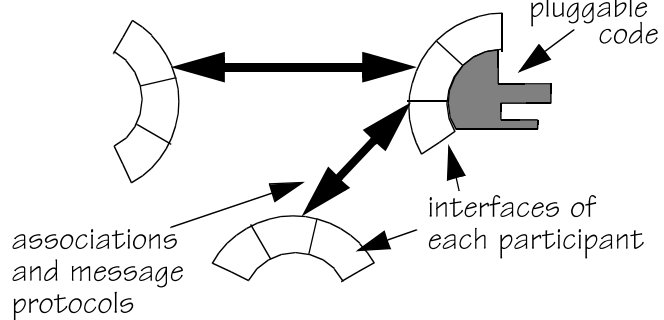
Looked at this way, we should think of OOD not as being so much about encapsulated objects – these are just the final result of composing frameworks. Instead (Figure 1), we should focus on the frameworks, which each contain interfaces of several ‘partial’ objects.

The objects emerge (Figure 2) when we compose frameworks (to make a system, or to make a bigger framework). Once it is clear what the complete requirements on an object are, then its internal structure can be fixed. The emphasis in a framework is to design the scheme of interactions between its participants, and may therefore just represent its participants as empty interfaces. However, in general, it is usual to provide some ‘pluggable’ code for each participant, which can easily be combined with the code for other roles.

Traditional object picture



Framework



**good for generic design-components**

**Figure 1**

## 1.1 *This paper*

The purpose of this paper is explore how exactly these frameworks are composed into systems in the context of a more formal approach to OO design – like that of Fusion[2], Syntropy[3], or Catalysis[4].

Section 2 discusses some basic ideas, defining this paper’s notion of a framework more precisely, and presenting briefly an interpretation of the meaning of the key constructs in object modelling notations.

Section 3 illustrates a framework involving just static relationships (as could be used for modelling) and presents a notation for its instantiation and composition with others.

Section 4 extends the idea to frameworks involving interactions between the participants.

Section 5 presents some further ideas, and the paper ends with a summary.

## 2 **Frameworks and models**

### 2.1 *What is a framework?*

A framework is:

- a goal
- a set of typed roles
- a set of [abstract] operations
- a set of trigger rules

Briefly, the goal is an informal description of purpose which this framework is intended to achieve, and often contains an invariant which we’re seeking to preserve — it might be to maintain a constantly correct view of a subject. The typed roles represent the participating objects and their static relationships (how they are linked by pointers etc) — for example, Subject and Observer. The operations represent their interactions — the messages they send to each other, such as update. The triggers characterise state changes caused by external operations which must prompt the interactions. External operations are those in which an object is involved by virtue of its participation in other frameworks.

In general, an object in a running system will not just play one role in one framework: for example, an observer of one object may also be a proxy for another, perhaps a view displayed on a client PC. In a sense, the frameworks are the interesting pieces of design, and the objects (and the classes that describe them) are just where they are tied together in a particular system. This interesting perspective has been promoted by Reenskaug[5], Ossher[6], Nierstrasz[7] and others.

Designing and developing frameworks, rather than classes, gives a much more flexible set of components to keep in your library. Ideally, to design a system, you select or build a set of frameworks, and fit them together. In practice, this is a bit of an exaggeration: you tend to begin with the basic classes, and then work out how they fit together in terms of frameworks.

When we want to generalise ideas and use them in many different combinations, it is frameworks that are interesting. Classes come in when you combine the frameworks in a particular application.

When you incorporate a framework into a system, you are imposing the relationships and protocol of actions that it expresses onto some selected objects (belonging to selected classes) in the system. The idea is that this should then achieve the framework's stated goal for that group of objects.

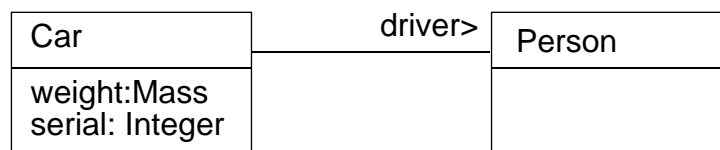
There are two ways of incorporating a framework into a design:

- Use an automatic tool where you select the framework and the objects you want to instantiate it for; the tool does all the installation. VisualAge for example, does this very well for Observer-like patterns; but it has only a restricted repertoire.
- More generally, work out by hand how the framework applies in your piece of design; check that you've done it properly; document the checks, so that when your design or the framework alters, the amount of rechecking is minimised to the changed parts.

## 2.2 Models

We will discuss frameworks in terms of a diagrammatic notation; so it is as well to understand what the lines and boxes of the notation mean. This section therefore briefly summarises the interpretation used in this paper. (More detail can be found in [8]).

A diagram in an object modelling notation is a set of assertions about objects, and can be translated into that form. (If not, there's something seriously wrong with it. Ask your OO Design Methodologist to provide such a translation: if they can't, suspect their foundations. An abstract syntax of their notation, written in their own notation, won't do!)



In our interpretation (which is more precise than, but consistent with, conventional accounts), this diagram says:

- |                           |    |   |
|---------------------------|----|---|
| x:Car,                    | -- | for any member x of the type "Car"                              |
| (x.weight):Mass           | -- | the expression "x.weight" refers to a member of the type "Mass" |
| & (x.serial):Integer      | -- | and "x.serial" refers to a member of the type "Integer"         |
| & (x.driver):Person       | -- | and "x.driver" refers to a member of the type "Person"          |
| & (x.driver.~driver) == x | -- | and "driver" has an inverse called "~driver"                    |
| & (p:Person,              | -- | and for any Person, p,  |
| (p.~driver):Car )         | -- | p.~driver is always a Car                                       |

(A pedagogical aside: we teach this as part of object oriented and design courses for practicing industrial programmers. We find that restricting the notation to the characters that can be typed on a US keyboard helps the formalisms gain readier acceptance. Hence the omission of `..` and the use of `'&'` rather than `'&'`, and `':'` instead of `'>'`.)

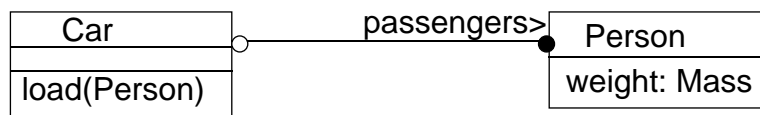
Types are just sets of objects.

In a specification model, attributes are considered to be abstract. If you look at actual members of type **Car** — whether in the real world or in a software representation — you might or might not find specific parts labelled **weight** or **serial**. The purpose of attributes is to help describe the type and its behaviour to its users.

In this interpretation, no distinction is made between Values and Objects. The integers, for example, are a set of objects which happen to be immutable; ‘operations’ like “+” don’t actually change anything: they are static relationships between the members of the type. No distinction is made between attributes and associations — if it is not convenient to draw **Person** as a separate box, we could have made **driver:Person** an attribute of **Car**.

An attribute like **driver** is part of a “model” in the sense that its main function is to help describe the types: nothing is said about how the attribute should be implemented. In a system built to deal with **Cars** and **Persons**, **driver** might be a pointer, or a relation in a database, or a function that derives the information from elsewhere. Or it might not be directly implemented at all: the internal structure can be very different from the specification model, whose main purpose is to help explain to the clients the externally visible operations.

If an association is decorated with a “•” (or “\*”), then the result of the query is a set, while the “o” (or “0,1”) means that the value of the query may be NIL:



means:

(x:Car, (x.passengers): Set(Person)) &  
 (p:Person, (p.~passengers):(Car+{NIL}) &

(where Set is a generic type whose parameter restricts its membership).

Used properly, these diagrams should be embedded within proper specification and design documents, with descriptive narrative. It’s often convenient to deal with different aspects of a design in different parts of the text – for example to talk about drivers in one place and passengers in another – and so we allow that one type can be mentioned in many diagrams. The meaning of the whole document is just all the assertions conjoined together – so for example, a **Car** has both **passengers** and a **driver**.

The box representing a type may have three parts, the third of which lists operations which may be applied to its members. Normally, this list is just a summary, and it is necessary to describe the operations in more detail separately. Here we will use postconditions to specify operations – for example:

**Car::load (boarder:Person)**  
**post** passengers == old(passengers) + boarder &  
 weight == old(weight) + boarder.weight

(“+” is overloaded here to mean “ ” when applied to sets.)

The postcondition is a relation between two states, before and after any occurrence of the operation; any subexpression may therefore refer to its value beforehand using **old(...)**. It can be thought of as a predicate a designer would have to satisfy. The benefit of using postconditions in specifications and in the earlier stages of design is that they allow the details of design to be deferred.

In the later examples, we will also use postconditions to describe operations not yet localised to any one object — where the effects are decided, but not yet who will take the responsibility for achieving them.

### 3 Defining and instantiating frameworks

We'll begin with a framework which is purely about modelling, with no actions involved. The subsection headings refer to the attributes that we earlier said a framework has — a goal, a set of typed roles, a set of [abstract] operations, and a set of trigger rules.

#### 3.1 *A Goal*

Suppose, as a professional object modeller, I am called to analyse a domestic trades agency, which sends people to your home to do plumbing or electrical work and the like. I find that the agency handles several types of Job Category (electrical, plumbing, central heating, ...) and each Job Occurrence (scheduled for a particular date at a given address) is for a particular Category. Each Job Category requires a set of Skills (e.g. heating requires electrical and plumbing) and each WorkPerson has a set of Skills. There are two essential invariants: that WorkPersons aren't double booked (i.e. Job Occurrences they're scheduled for don't have overlapping dates); and that for every Job Occurrence, the skills of the scheduled WorkPerson must include at least the skills required for the Job Category.<sup>1</sup>

#### 3.2 *Typed roles*

While drawing the model, it occurs to me that this kinds of scheduling problem must occur in many situations. Therefore I generalise the type names a bit, so that it comes out like Figure 3. This is a simple variety of framework: it is a generic template for part of a model. I add the framework to my library of re-usable design pieces under the name ResourceAllocation, and deliver a copy of it to my clients, together with the assertion:

```
ResourceAllocation
    [ ResourceFacility \ Skill,
      Resource         \ WorkPerson ]
```

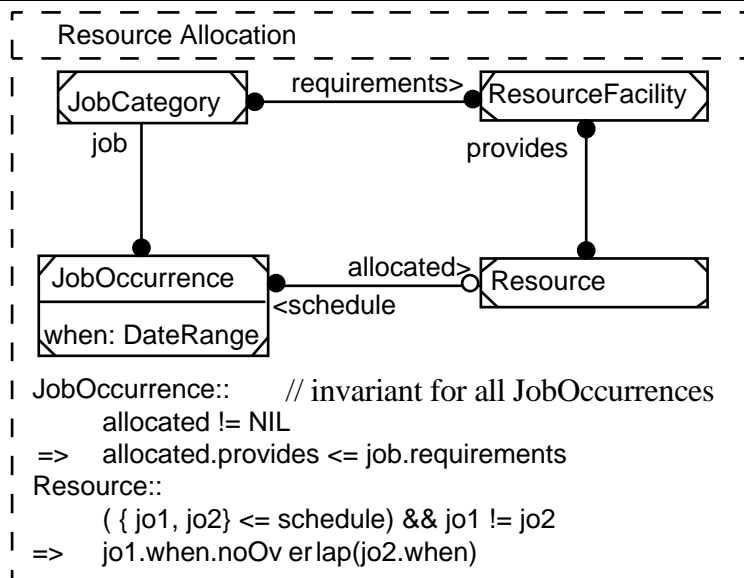
which means “make a copy of ResourceAllocation, substituting the names as in the bracketed list.” Which provides my clients with the model they need.

The marks on the corners of the types in the framework show that they and their associations and attributes are effectively parameters of the framework, and may be substituted where the framework is applied. If the framework referred to a type like “Date” or “integer”, defined and fixed elsewhere, that would appear as an ordinary, un-renamable type.

The following week, I am asked to model the scheduling for a teaching organisation. It has a portfolio of courses, offered on various dates throughout the year. Each instructor is qualified to teach some set of courses, and some courses require certain other skills such as knowledge of OS/2. Each course must be allocated a room, and there are various facilities that rooms must offer (equipped with PCs, has video, capacity>20, ...).

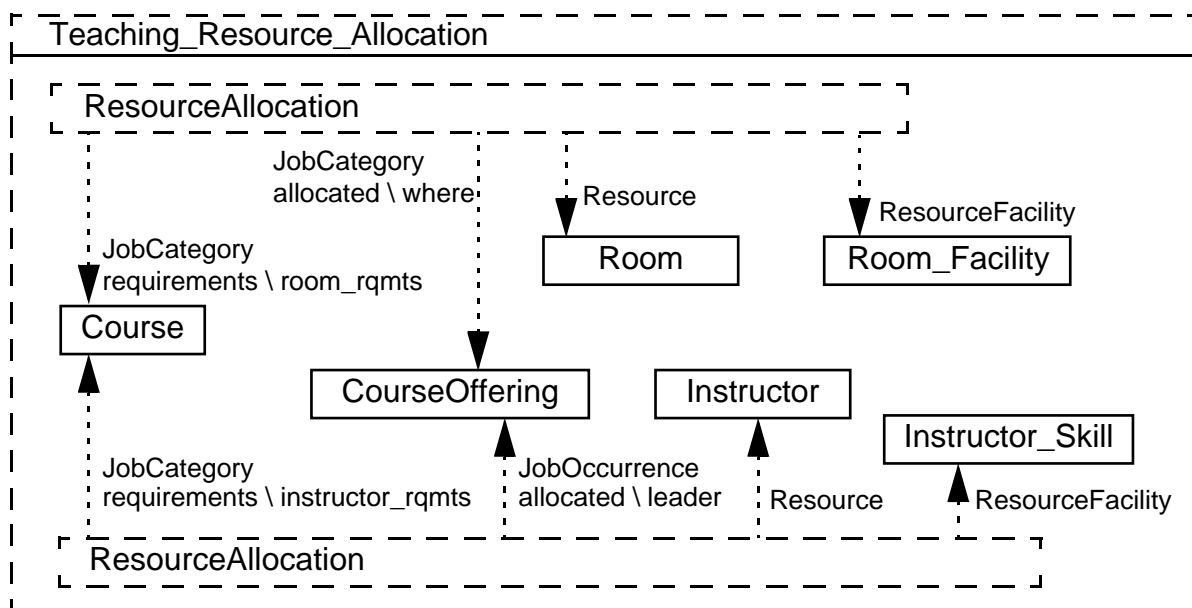
---

1. John Cameron outlined a problem, using this example, to which Frameworks are a solution.



**Figure 3.** Framework with model and invariants

I spot that this is a resource allocation problem — two such problems combined, actually. Accordingly, I write down an instantiation of Resource\_Allocation, as in Figure 4 (Teaching\_Resource\_Allocation).

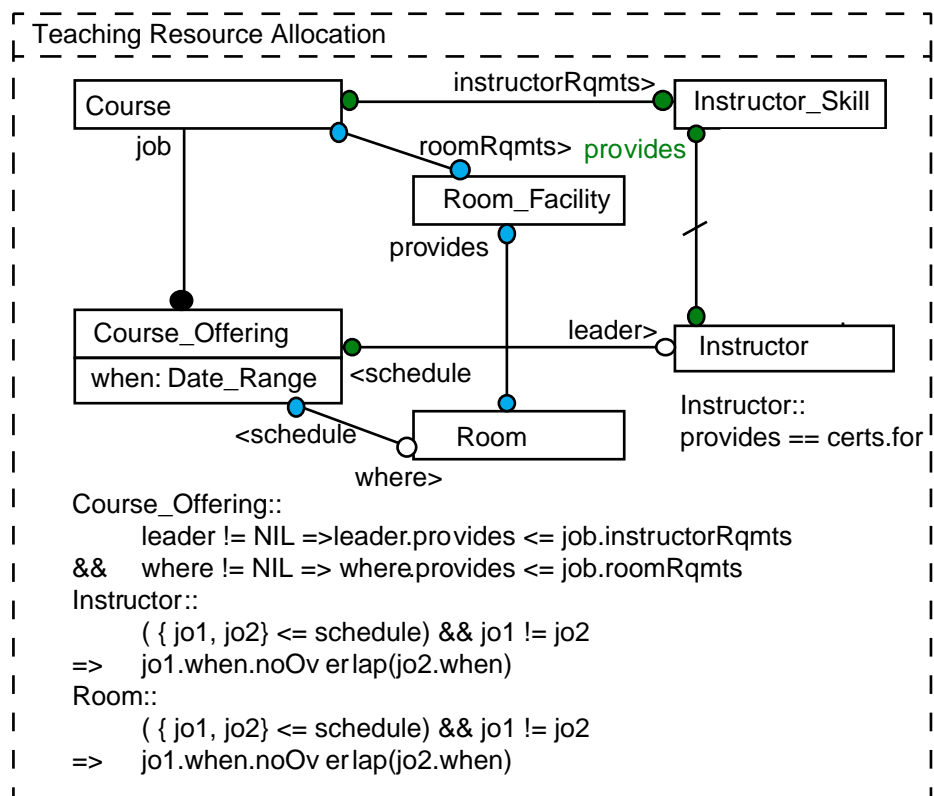


**Figure 4.** Framework instantiation

This figure shows an experimental more pictorial notation for the substitutions. Each dotted arrow shows how a parameter-type is specialised — thereby imposing on the argument type all the assertions implied by the framework. Attribute and link names can also be substituted.

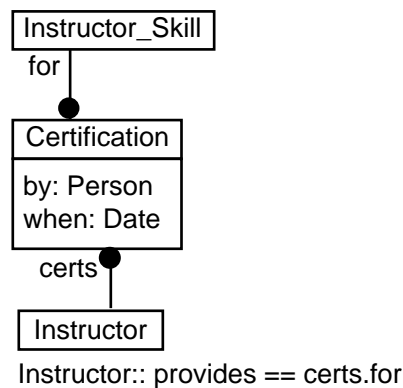
I send this to my clients together with a copy of the ResourceAllocation framework from my library, and a pleasantly large invoice.

What does this mean? Just that the model for the teaching outfit is made by taking together all the types and assertions implied by both instantiations of the framework. Imagine for a moment I have a support tool that understands all this; I can ask it to show me the resulting model, which would appear as in Figure 5.

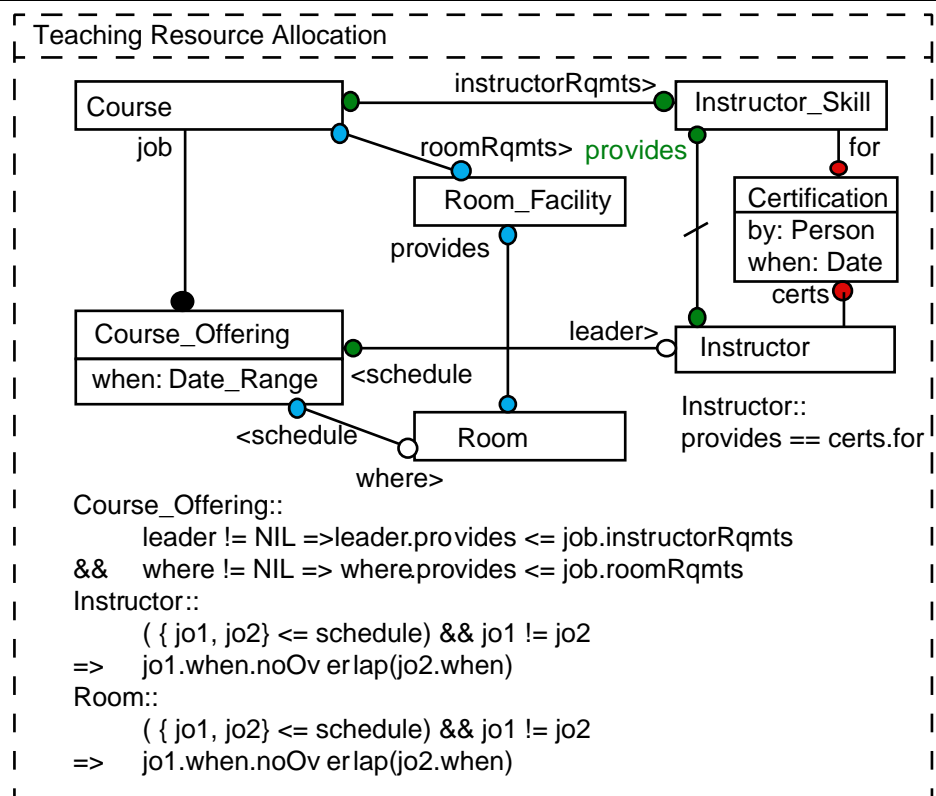


**Figure 5.** Unfolded instantiations of framework

If there are extra things I want to say about this particular model, then I only have to add them in. For example, we may need to record when an Instructor acquired a given skill, and who certified her. So I add the extra pieces of model to the analysis document, with a constraint that describes the `provides` attribute for Instructors in terms of the new links:



(Result in Figure 6.) In practice, I would probably also add some operations to the types; although it is useful just as a model, providing a common pattern of relationships and invariants.



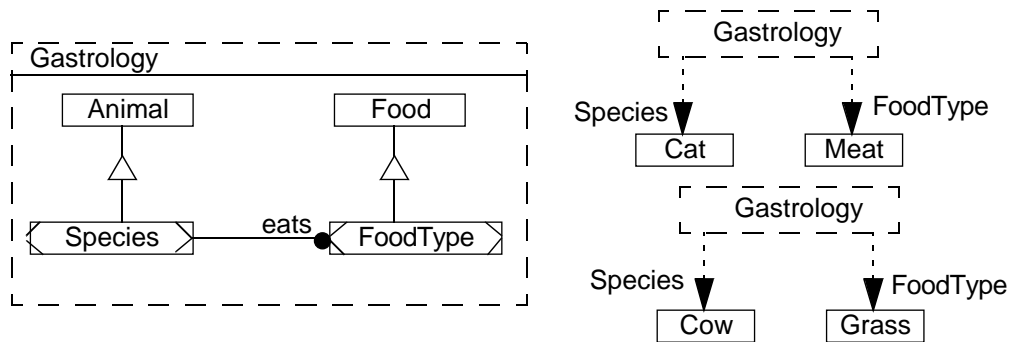
**Figure 6** Tool displays sum of all knowledge about Teaching Resource Allocation

### 3.3 Frameworks are more powerful than subtypes and generic types

Notice that the specialisation of the types in the generic framework is considerably more expressive than subtyping. If we attempt to interpret Figure 3 as a relation between four supertypes, then we appear to be saying that every JobOccurrence has a relationship called ‘allocated’ with any subtypes of Resources. In the generic interpretation, JobOccurrence is not a type, but a placeholder for one; and so are its attributes and links.

Another good example is the faulty old syllogism “Animals eat Food; Cows are Animals; Beefburgers are Food; hence Cows eat Beefburgers.” But the first statement should not be taken to mean that every object conforming to the type Animal can eat every instance of the type Food. A more explicit statement would be “For every subtype

A of Animal, there is a subtype F of Food such that all members of A can eat any member of F". Using the proposed notation, we would write:



**eats** might represent the assignment of food-items to specific animals in an automatic feeding system. We thus ensure that instances of **Grass** will be the only members of **Food** proffered as fodder to any **Cow**-instance.

Some programming languages provide for specialisation of individual generic types. That is not as powerful as framework schemes, in which groups of relationships between several types are specialised together. For example, if we try to model Gastrology using C++ templates, it is difficult to include all the appropriate restrictions without running afoul of fixpoint ambiguities:

<pre>template &lt;FoodType&gt;   class Species : public Animal {   FoodType eats;   ... } typedef Species&lt;Grass&gt; Cow;</pre>	<pre>template &lt;Species&gt;   class FoodType : public Food {   Species eatenBy;   ... } typedef FoodType&lt;Cow&gt; Grass;</pre>
---	--

The problem would be alleviated if templates were to encompass a package of classes, rather than just one. (Let us hope that the future `generic` feature of Java will be sensibly attached to packages!)

#### 4 Frameworks with interactions

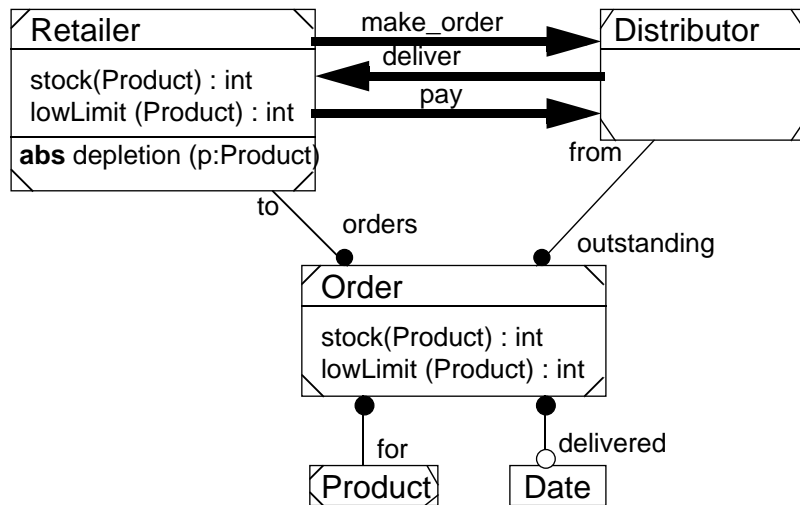
Figure 7 (Retail Distribution) shows a framework with three abstract operations. A Retailer is anyone who sells Products to the public; Distributors sell only to Retailers. An abstract operation *make\_order* is provided whereby the Retailer can create Orders; the Distributor can *deliver* Products, fulfilling an Order; and the Retailer can *pay* for them.

(This could represent either a 'business model' about objects in the real world; or it could be a design for a software framework, in which the objects are interacting software components. The same principles apply.)

The middle section of the framework diagram is the internal structure: this defines what goes on between the participants. The bottom 'external' section is what can be seen from outside.

Three internal abstract operations are shown: *make\_order*, *deliver*, and *pay*. These are interactions which take place only between members of the types (or their subtypes)

## Retail\_Distribution



```

op rtr:Retailer -> dist:Distributor :: make_order (prod:Product, q:int)
-- when a Retailer sends make_order to a Distributor ...

post -- the effect should be to add a new Order to the lists
let new= Order[quantity==q, to==rtr, from==dist, for==prod,
    paid==0, delivered==NIL, cost==dist.price(prod, q, rtr) ],
    rtr.orders += new & dist.outstanding += new

op dist:Distributor -> rtr:Retailer :: deliver(o:Order)
-- when a distributor sends 'deliver' to a Retailer, ...

post o.delivered != NIL -- the effect should be to mark the Order done
& (rtr.stock(o.for)+= o.quantity) -- and increase the stock
  
```

```

r:Retailer, p:Product, -- for any retailer and product
r.stock(p) < old(r.stock(p)) => abstract op depletion (p)
-- any operation that depletes the stock is what we call a depletion
  
```

```

abstract op Retailer:: depletion (p:Product)
-- any operation that is a depletion must also conform to this:

post stock(p) < lowLimit(p) => orders[for==p] != 0
-- if the resulting stock is lower than the appropriate limit, there will be an
order for it ("the subset of orders for which the 'for' attribute is p will not be
the empty set")
  
```

**Figure 7.** Framework with abstract operations

shown here. Their postconditions (one has been omitted for relative brevity) show the effects on both participants. They are all abstract operations.

An abstract operation is used in analysis or high-level design to summarise the effects of what will finally turn out to be a sequence of smaller messages, possibly in both directions. Both participants will be affected by the more detailed design, since each will have to understand the more detailed protocol (just as, for example, someone may know that money can be got from a bank ATM, but not know the detailed operating instructions). The idea is similar to refinable transactions and to use-cases[9]. Both participants are named in the headers and referred to in the postcondition.

The external operations of a framework are always abstract. This is because, as framework designers, *we do not know what other frameworks each participant may take part in*. What we can do is constrain the other actions in which the object is involved.

Retailer has the essential constraint that any action that depletes the stock must also conform to the postcondition under “depletion”. As we will see, this means that anyone who implements an object that is both a Retailer and something else (a Shop or a Restaurant, or ...) may have any other operations she likes; but any of those operations that reduces the stock must be considered to be a depletion, and must — whatever else it does — conform to the depletion postcondition as well.

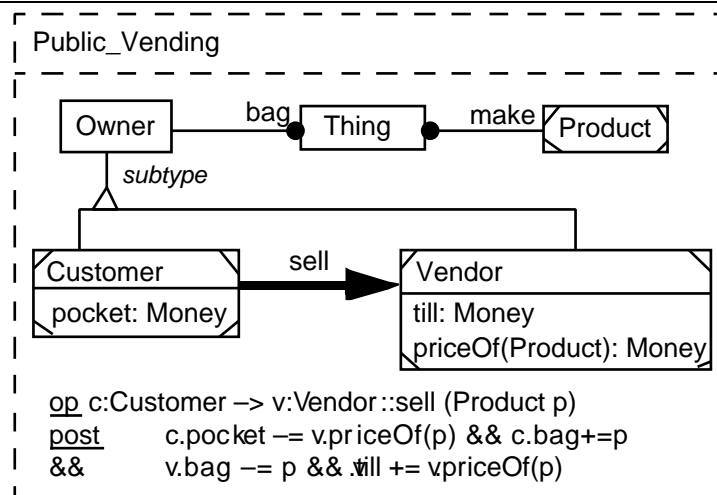
This is the point of a framework. It focusses on the relationships between a given set of roles, but always with the assumption that each object will also play other roles in other frameworks; and that these relationships will include operations affecting its state. So if an object which is a Retailer is also a Shop, then sales to customers will deplete the stock; if it is a Restaurant, cooking food (whether we successfully sell it or not) depletes the stock. We find out the missing information when we tie this framework to others.

(Other external abstract operations have been omitted: for example, relating to where the Distributor gets its own stock from.)

Back in Figure 1, we said that a Framework could include pluggable code of partial implementations of the objects. In this case, we have chosen not to do that, merely specifying the operations. However, certain design decisions have been laid down here. Firstly, we have decided exactly what operations will go between the participants, and what their effects will be; secondly, it is clear that anyone who designs a stock-depleting operation will be able to use `make_order` to help fulfill the re-ordering requirement.

#### 4.1 Composition of frameworks

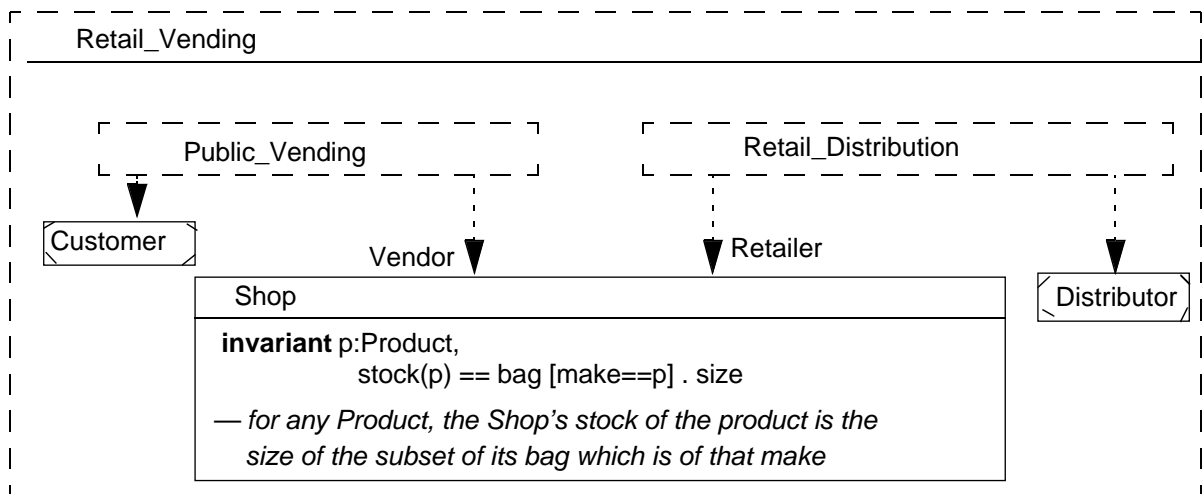
Figure 8 shows the relationship of Customers to Vendors. In the sell operation, cash and Products are transferred in opposite directions.



**Figure 8.** Another framework

A Shop is a type of object that plays the roles of both Retailer and Vendor. So now we compose the two frameworks into a single picture, with Customer, Shop, and Distributor as the key players. In `Public_Vending`, the Vendor’s stock was represented as a set of Things, each of which is an example of a Product; so this model must be tied up,

using an invariant, with the Retailer's stock which had been modelled as an integer for any Product.



(Result in Figure 9.)

The final step is to implement the types with classes. Supposing that `Shop:sell` is not refined further, but implemented as a single message, the designer will have to observe `Retail_Distribution::depletion` whenever stocks get depleted. In this case, the only known way of creating an `Order` when necessary is using `make_Order` – so a call to this will sometimes have to be part of executing `sell`.

#### 4.1.1 Building systems from collaborations

Shop is a synthesis that plays two roles. Each role is about the interaction with another type of object — or rather, a role of another object. The Shop functions by having enough roles to make a coherent unit: in this case, ensuring the throughput of stock.

Given a variety of different collaborations, it is possible to construct many different role-playing objects. Collaborations are plugged together by making objects that play roles in each (and sometimes more than one role in one collaboration, just as a person may wear more than one hat in an organisation). For each object, it is necessary to state how participation in one role affects the other by tying together their vocabulary of state changes — as is done with the Shop's 'bag' and 'stock' above.

But the main work of the design resides in the collaborations themselves, and plugging them together is relatively straightforward. Collaborations are the best focus for design, and objects are secondary. Following this principle results in more flexible designs.

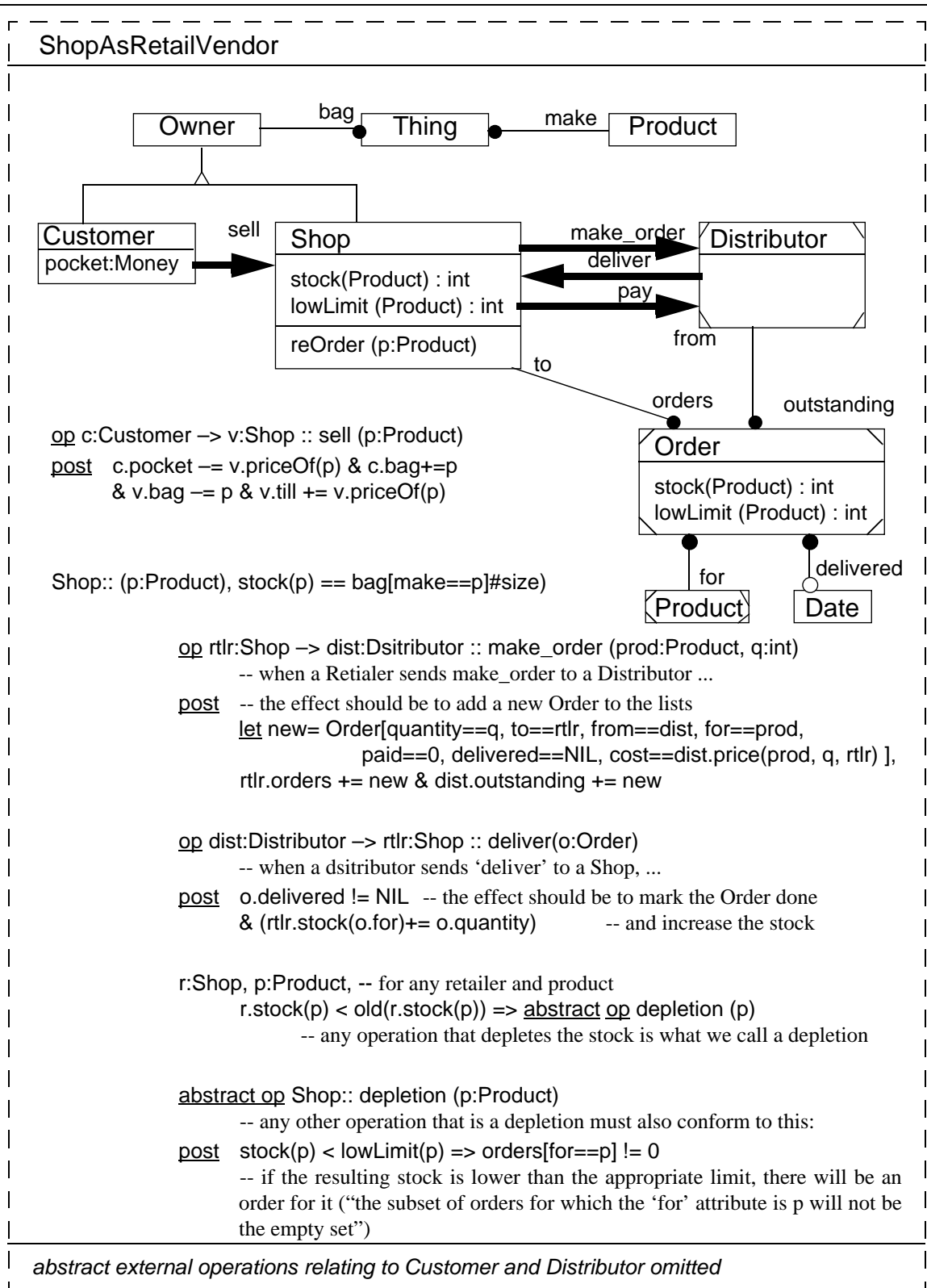
### 4.2 Trigger Rules

It is also possible to document “trigger rules” that state that, when an external action (from an object's other roles) causes a change of state matching certain conditions, then one of the internal actions must be invoked. For example, in `Retail_Distribution`, we could have insisted that

```

external Retailer::any
post  p:Product, (old(stock(p)) < stock(p) ) =>
      exists dist:Distributor, { dist.make_order(p,10) }
  
```

“For every external operation on Retailers, it must be part of the postcondition that if the stock of any product is depleted, then `make_order` must be invoked for some `Distributor`.”



**Figure 9.** Composite displayed by tool of two frameworks

This technique mandates the use of internal actions more explicitly, and is particularly useful for implementing “business rules”.





The < operation itself can be renamed:

```

[ WindowList_framework
| SortedList_Template
| [ SortedList \ Window_List,
|   Item       \ Window
|   Item :: <  \ Window::in_front_of]
]

```

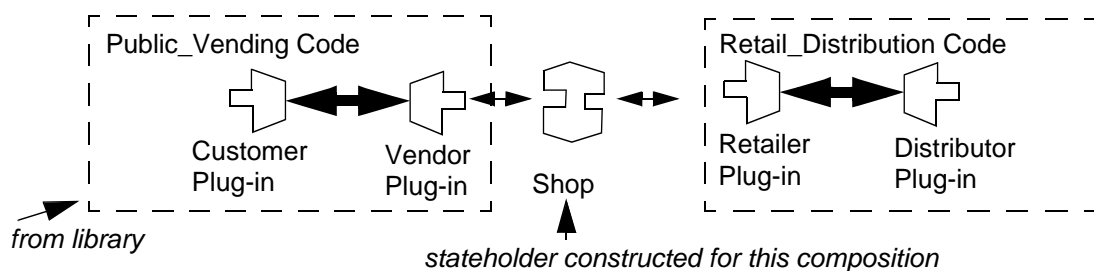
The use of the TotalOrdering trait ensures the right properties are specified for the client's ordering operation. This is something that you don't get with C++ templates, for example. Liskov's[10] "where" clause and the "theories" of FOOPS are the closest analogies, but both have the disadvantage of introducing a separate piece of syntax. Here, we separate the idea of a template from the idea of a class, which C++ insists are all one. Admittedly, this makes things a bit unwieldy for single-class frameworks; but is more powerful in general.

## 5 Code components

The above examples have all been about the construction of designs from frameworks. They are very useful library items, representing significant amounts of design effort. It is possible to envisage them being passed around and enhanced within an organisation.

On the commercial market, software developers usually prefer to publish executable code without the sources. They often prefer to sell the results of design, rather than design itself. A commercial component therefore tends to have a well-defined interface and an encapsulated internal structure — just what you'd expect in object oriented terms. This leads naturally to expectation that marketable business objects will represent things like 'a shop' or 'a consumer', 'an order', 'a product'; and that the relationships between them come about when such components are connected together.

But the thrust of the argument above is that more flexibility is obtained by inverting that approach, focussing on the collaborations first; objects appear where they are linked. How would such a scheme work?



**Figure 10.** Implementation composed by fitting together plug-in components

Role-delegation is the key pattern. Each conceptual object (Shop, Wholesaler, manufacturer) is actually represented by a stateholder plus a number of role-players. The function of the stateholder is to hold those parts of the state (such as stock) that are common to all the roles; all state and behaviour that is intrinsic to an interaction with another object is held within the appropriate role. Roles update the common state, and are observers of it, receiving notification when it changes.

One benefit of role-delegation is that it enables an object's roles to change dynamically; and the other is that the roles can be provided by separate packages of software. The stateholder objects can also be marketed as packages, though generally of less complex-

ity than the collaborations. The essential requirement is a well-defined specification at each role/stateholder interface. With a suitable kit of parts, different combinations result in different behaviour.

These examples use shops etc to illustrate a general principle which is more widely applicable. But looking specifically at the particular case of the consumer supply chain, would not each object — shop, distributor, manufacturer — reside only inside the host machine of its owner, the real-world shop etc? Not really: a Shop's machine requires objects acting as proxies for its Distributors and Customers, so that in building the system, collaborations with them are still the focus.

Example code can be found in [11].

## 6 Systems are built from frameworks

This has illustrated the specialisation of frameworks to make models and design, and their composition to build larger pieces. A complete system can be thought of as a combination of many frameworks, each of which may be specialised from a more general library item. (It's an interesting exercise, for example, to generalise the `Public_Vending` and `Retail_Distribution` frameworks to a single more general one that talks about transfers of ownership.)

Frameworks are more useful than classes for making re-usable pieces of design work. They are more flexible because they do not bind classes to be the units of work; and they allow the many roles of a single object to be designed separately.

## 7 References

- 1 E Gamma et al, *Design Patterns* [A-W 1994]
- 2 Coleman, D et al: *OO Development: the Fusion Method*[PH 1994]
- 3 S Cook & J Daniels: *Designing Object Systems* [PH 1994]
- 4 D D'Souza & A. Wills: "Extending Fusion: practical rigor and refinement" in R Malan et al, *OO Development at Work* [PH 1996]
- 5 T Reenskaug et al, *Working with Objects* [Manning/PH 95] 0-13-452930-8
- 6 W Harrison, H Osher, H Mili "Subjectivity in OO Systems" workshop reports, OOPSLA 94 & 95.
- 7 O Nierstrasz & D Tschritzis, eds, *OO Software Composition* [PHI 94]
- 8 D D'Souza, A Wills, and others. Catalysis – papers.  
At <http://www.iconcomp.com>
- 9 Ivar Jacobson et al, *OO Software Engineering* [AW 1992 0-201-54435-0]
- 10 M.Day, R.Gruber, B.Liskov, A.Myers "Subtypes vs Where Clauses: Constraining Parametric Polymorphism" [<http://www.pmg.lcs.mit.edu>]
- 11 A. Wills, "Frameworks" extended article. At <http://www.trireme.com/papers>