Chapter 4 Interaction Models Use Cases, Actions, and Collaborations

Chapters 2 and 3 have described how to model the behaviors of an object by specifying operations in terms of attributes. However, the most interesting aspect of any design lies in the interactions among the objects: the way that the net behavior resulting from their collaborations realizes some higher-level function when they are configured together in a particular way.

Use cases, actions, and collaborations abstract the interactions among a group of objects above the level of an individual OOP message send. These interaction models let you separate abstract multiparty behaviors, joint or localized responsibilities, and actual interfaces and interaction protocols.

Section 4.1 provides an overview of the design of object collaborations. Section 4.2 begins with examples of object interactions to show that many variations in interaction protocols achieve the same net effect and so motivate the need for abstract actions. Section 4.3 introduces use cases and relates them to actions and refinement. Section 4.4 explains how actions and effects are related to abstract actions. Section 4.5 describes concurrency between actions and explains how to specify these constraints.

Collaborations—sets of related actions—are introduced in Section 4.6. Section 4.7 describes how to use collaborations to describe either the encapsulated internal design of a type specification or an "open" design pattern. The separation of actions internal to a collaboration from those that are external to a collaboration forms the basis for effective collaboration models and is the topic of Section 4.8.

4.1 *Designing Object Collaborations*

The big difference between object-oriented design (OOD) and the procedural style is that with OOD your program must not only work as a sequence of statements but must also be well decoupled so that it can easily be pulled apart, reconfigured, and maintained. You

must make an extra set of decisions about how to distribute the program's functionality among many small operational units with their own states. In return—if you do it well—you get the benefits of flexibility.

There are three big questions in object-oriented design.

- *What should the system do?* This is the focus of type specification in Chapter 3. You'll find guidance on putting it together in Chapter 15.
- *What objects should be chosen?* The first draft is the static model we used for the type specification, although it is modified by design patterns to improve decoupling.
- *Which object should do what, and how should the objects interact?* The most important criterion is to meet the specification. In addition, our goal is to separate different concerns into different objects while balancing the needs of decoupling with performance.

The art of designing the collaborations is so important that many experts advocate making collaborations the primary focus of object-oriented design—with, of course, a reasonable type specification first. In Catalysis, collaborations, like types, are therefore first-class units of design. A *collaboration* is a design for the way objects interact with one another to achieve a mutual goal. A type represents a specification of the behavior seen at an interface to an object. By contrast, a collaboration represents a design of the way a group of objects interact to meet a type specification.

There are several situations in which collaborations should be used.

- When you're designing what goes on inside a software component.
- When you're describing how users (or external machines or software) interact with a component you're interested in. It is useful to understand how a component is to be used before constructing it.
- When you're decribing how real world objects in a business organization (or a hardware design) interact with one another. This description is typically used to help explain the business in which a system is to be installed or updated.

4.2 *Actions (Use Cases) Abstract Complex Interactions*

We've already said that it's often useful to talk about what is to be achieved in some part of a complex design before you go into how it is to be achieved. Type specifications are about hiding the "how" inside individual objects and stating only the end result of invoking a particular operation.

Abstract actions do the same for interactions *between* groups of objects (as opposed to inside them). We know that we want to achieve the transfer of information and that it might involve the participants having a complex dialog, but we may wish to leave the details aside for a separate piece of work. The interacting participants might be objects inside a program, or people, or people and computers.

For example, how does the sale of a product from a wholesaler to a retailer work? Here are three different possibilities, as illustrated in Figure 4.1.

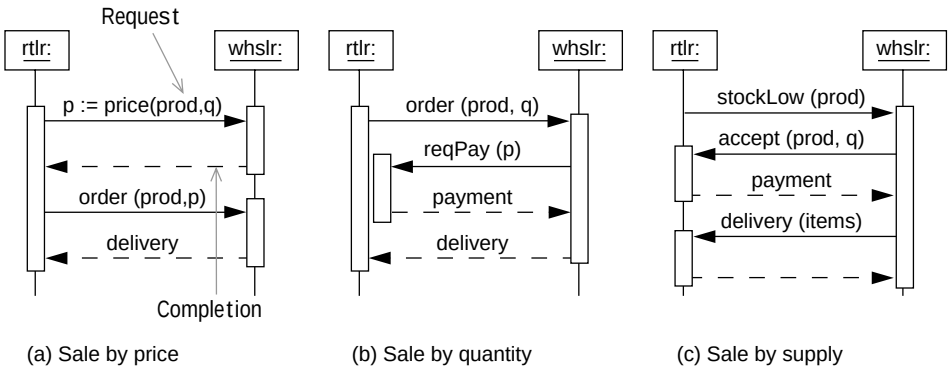


Figure 4.1 Three protocols for a sale of items.

a. The retailer first queries the price for a product and then requests a sale while providing payment; the wholesaler then delivers the items.
b. The retailer requests the sale, and the wholesaler calls back requesting payment of the appropriate amount. When the payment is provided, the wholesaler delivers the items.
c. The wholesaler triggers the retailer to buy the product, perhaps by monitoring the retailer's inventory systems. The retailer returns a payment, and the wholesaler then delivers the items.

But at some level of analysis or design, I don't want to bother with which of these protocols is used or whether all or any of them is available. What they have in common is the same net effect of transferring a quantity of items and money based on the wholesaler's price. They differ in who initiates which operation and the protocol of interactions.

In Catalysis, we can describe the whole dialog as a single interaction (Figure 4.2). On any sequence chart, a series of horizontal bars can be collapsed to a single bar, representing their combined effect. The horizontal bars connect objects that influence or are influenced by the outcome: there may be any number of them. If all the possible refinements have the same initiator, it can be marked with a shaded oval (Figure 4.2b).

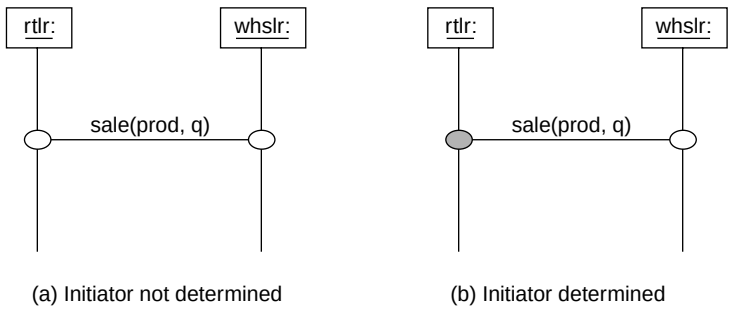


Figure 4.2 A sale as one action, without the details of the protocols.

4.2.1 Action Types and Refinements

Each horizontal arrow or bar represents an occurrence of an action. Just as object instances belong to types that group them according to behavior, so action occurrences belong to action types, to which descriptions of the action can be attached. An action type is shown as an ellipse associated with the types of object that can participate in it (see Figure 4.3). Participants are those objects whose states can be changed by, or whose states can affect

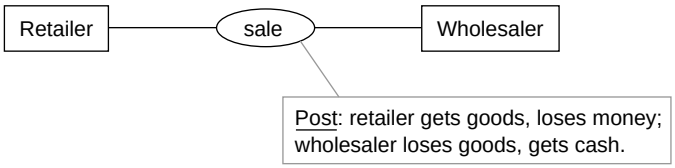


Figure 4.3 An action type with participants.

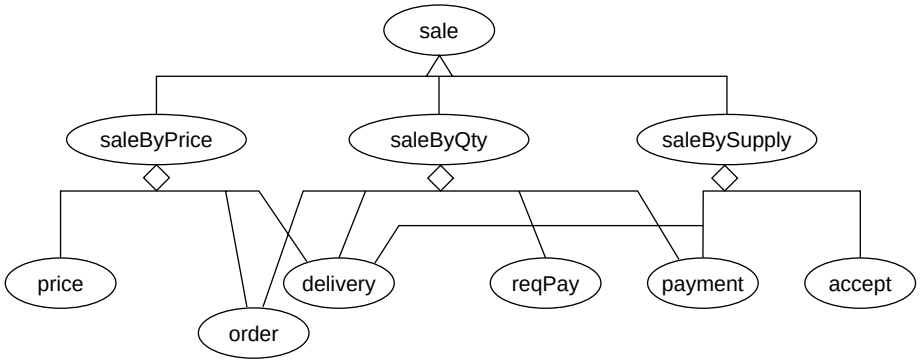


Figure 4.4 Different kinds of sale and and their constituent actions.

the outcome of, an action; there may be any number of them.

The value of an action such as sale is that it represents all protocols that achieve the same postcondition. We can show its relationship to different possible protocols, as in Figure 4.4. The subtyping symbol is used to show that there are different variants of sale, all of them sharing and extending its description. The diamond aggregation symbol decomposes an action into more-detailed constituents. An aggregation doesn't show exactly how the smaller actions make up the larger one. They might be a sequence or might occur repetitively or concurrently; that must be documented separately for each aggregation—for example, by using sequence diagrams such as that shown in Figure 4.1.

The most-abstract actions represent only the effect of an interaction. The more-detailed ones are directional, stating who intiates the request and who processes it. The most-concrete actions are object-oriented messages.

4.2.2 Preview: Documenting a Refinement

Chapter 6, Abstraction, Refinement, and Testing, discusses refinement in detail, but we include a short discussion here to show how to relate abstraction and realization to each other in the context of actions.

One of the finer-grained actions in a sale is the delivery of the product to the retailer. Consider a simple retailer warehouse object, WHA. One of its operations is re_stock, by which one of the products in the warehouse is restocked by some quantity. One abstract description of this action uses a simple type model:

action WH_A::re_stock (p: Product, q: Quantity)
post: p.stock += q

One realization of this warehouse, WHB, provides a sliding door that can be moved to a particular product and then opened; then items are added to that product shelf one at a time before the door is closed. The actions at this level of realization need a slightly richer type model with a selected_product attribute:

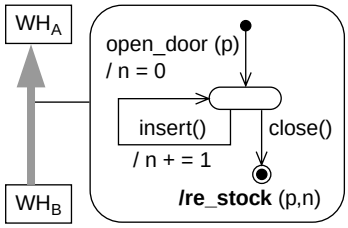
action WH_B::open (p: Product)
post: selected_product = p

action WH_B::insert ( )
post: selected_product.count += 1

action WH_B::close ( )
post: selected_product = null

It is clear that a certain sequence of these detailed actions constitutes a valid re_stock action. Thus the following sequence constitutes an abstract action: re_stock (p, n).

open (p); $insert_1$(); $insert_2$(); ... $insert_n$(); close()

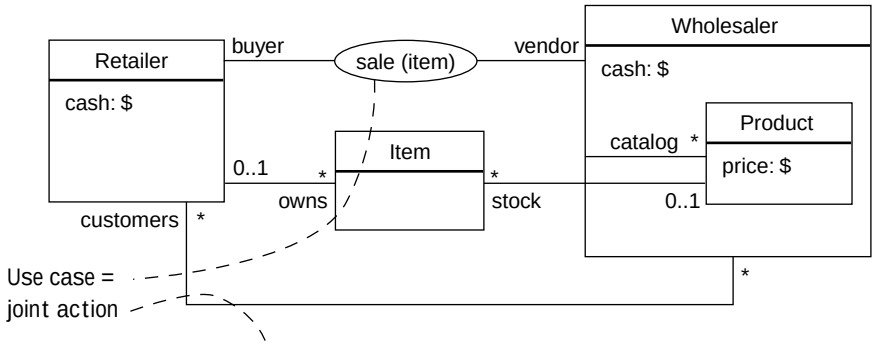


The refinement relation between the two levels of description documents this mapping. The state chart shows how the parameters and state changes in the detailed action sequence translate to the abstract action and its parameters. In this example we use a counter attribute to define this mapping; it is an attribute of a specification type representing a restocking in progress at the more detailed level (Pattern 14.13, Action Reification).

4.2.3 Joint Actions (Use Cases)

In Catalysis we believe in the value of separating the various decisions that must be made during a design. We therefore require two things: a way to describe the overall effect of an interaction without any attribution of who does what and a way to document who does what after we have made that decision.

A *joint action* represents a change in the state of some number of participant objects without stating how it happens and without yet attributing the responsibility for it to any



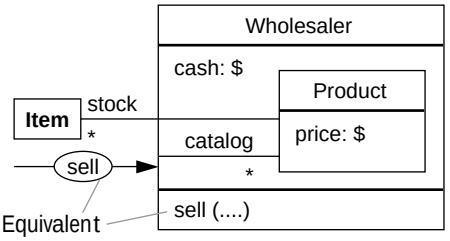

```
action sale (buyer:Retailer, vendor:Wholesaler, item: Item)
post: let price = item.product.price in
        -- items were in stock, retailer must be registered with w
            vendor.products.stock@pre–> includes (item)
    and     vendor.customers–> includes (buyer)
        -- and retailer must have cash to pay for that quantity of items
    and     buyer.cash@pre >= price
        -- cash and item have been transferred
    and     buyer.owns = buyer.owns@pre + item
    and     buyer.cash = buyer.cash@pre – price
    and     vendor.catalog.stock = vendor.catalog.stock@pre – item
    and     vendor.cash = vendor.cash@pre + price
-- can abbreviate to buyer.owns += item and buyer.cash –= price etc
```

Figure 4.5 A joint action, or use case.

one of the participants. We can write its postcondition more or less formally in terms of the models of the participants as shown in Figure 4.5.

4.2.4 Localized Actions

To assign responsibility for the execution of an action, use a localized action. Draw the action attached to the type or (more conventionally) list it in the bottom section of the type box.



In accompanying text, a localized action is prefixed with its type using the context operator:

```
action Wholesaler :: sell (retailer : Retailer, item : Item, out price : $)
pre         catalog.stock->includes(item)     -- this item was part of our stock
post        price = item.product.price        -- price returned to caller
    and     cash = cash@pre + price
    and     stock = stock@pre – item
```

In the specification of a localized action, self is defined and refers to the *receiving* object. We do not know who the invoker is. Return values (e.g., price) can be defined.

4.2.5 Joint versus Localized Actions

Let's look at some differences between joint actions and localized actions. First, localized actions are invoked. A localized action represents an interaction in which the receiver (the object taking responsibility) is requested to achieve the postcondition. It should do so if the precondition is met.

The message (or operation) in an object-oriented programming language is one realization of a localized action. Localized actions are a bit more general, encompassing any way in which an object can be stimulated to do something.[1] For example, in specifying an application program such as a drawing editor, we can think of the editor as a single object; the user operations, such as cut, paste, and move shape, are actions localized on the editor. They are abstract in that each one is invoked using a sequence of smaller actions (keystrokes, mouse moves, and clicks); there may be no single message to a particular object within the editor corresponding to, say, move shape.

Joint actions represent possibilities A joint action represents a specification of something that may occur. It can be referred to in other specifications (e.g., "To achieve restocking, one or more sales must occur") but can't be invoked directly from the program code. To do that, you must have an implementation (such as one of the sequences in Figure 4.1), which will tell you one of the ways to start.

Localized actions provide preconditions. When a localized action is invoked, if the precondition is true we are assured that the postcondition will be achieved. Joint actions represent descriptions of history: Looking at the history of the world, wherever this and that has happened, we call that a sale. Wherever something else happened that started the same way but ended up differently, we call it something else, such as a theft.

(We sometimes find it convenient to write "preconditions" on joint actions, but they are really only clauses that should be wrapped up in a big (...)@pre and conjoined with the rest of the postcondition.)

4.2.6 Action Parameters

The parameters of an action represent things that might be different from one occurrence to another. In any sale, the selling and buying parties may be different, and the item being sold is different; they are all parameters. The price also changes, so we could make it a parameter; however, we might assert that the price exchanged in our sales is always the price in the vendor's catalog. Therefore, although we could include it as a parameter, it would be redundant. Notice how different action parameters are from programming-language parameters: we are specifying the information content of the interaction and not its implementation.[2]

1. A directed joint action encompasses any way one object may stimulate another to do something.

In a joint action, some of the parameters may be distinguished as participants and drawn linked to the use case pictorially, whereas other parameters are written in text style. For example, in Figure 4.5, buyer and vendor are participants, whereas item is a parameter. In business analysis, the difference is a matter of convenience and is analogous to the equivalence of the associations and attributes of object types. In a software design, the participants can be used to represent objects that we know will definitely exist in the final code and that will, between them, take responsibility for executing the action. The list of parameters, on the other hand, represents information transferred between them whose implementation is yet to be determined.

In text, the list of participants can be written in front of the action name as partially localized context:

```
action (buyer:Retailer, vendor:Wholesaler) :: sale (item: Item)
post:  let price = item.product.price in
           vendor.customers → includes (buyer)
```

4.2.7 From Joint to Localized Actions

A joint action is interesting because its effect says something important about all participants. Although some joint actions, such as the one in Figure 4.2, do not designate any participant as an initiator, in general you can designate initiator and receiver. Here is the syntax for each variation of a three-way use case, sale, between a retailer, a wholesaler, and an agent. (We use the convention that the names of untyped parameters imply their types.)

- (retailer, wholesaler, agent) :: sale (x: Item)

  This represents a joint action with three participants, with no distinguished initiator or receiver. The effect refers to all participants, parameters, and their attributes.

- retailer -> (wholesaler, agent) :: sale (x: Item)

  A joint action, this time initiated by the retailer. It is now meaningful to mark some parameters as *inputs*—determined by the initiator by means unspecified in the effects clause—and others as *outputs*—determined by other participants, used by the initiator in ways not fully specified in the effects clause.

- retailer -> agent:: sale (x: Item, w: Wholesaler)

  A directed joint action that designates the retailer as initiator and the agent as the receiver; the sale is initiated by the retailer and carried out principally by the agent. Again, the effect refers to all participants and parameters. In this example, the wholesaler is identified to the agent by the retailer as a parameter. An alternative arrangement might leave the choice of wholesaler to the agent, appearing only in the effects clause based on some attributes of the agent.

- initiator: Object -> agent :: sale (....)

2. Perhaps we should have used a markedly different syntax.

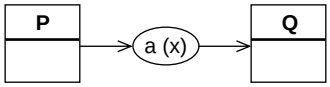
A directed joint action initiated by an object and received by the agent. Nothing is known about the type of the initiator or its role in this action; hence, initiator is declared to be of unknown type Object.

Our use case template permits these additional distinctions to be made:

use case	sale	
participants	retailer, wholesaler	
initiator	retailer	-- also listed as participant
receiver	wholesaler	
parameters	set of items	-- can separate inputs/outputs for directed actions

A localized operation is a degenerate case of a joint action with a distinct receiver, in which nothing is known or stated about the initiator's identity or attributes. All relevant aspects of the initiator are abstracted into the input and output parameters of the operation. The following is a fully localized operation that cannot refer to the initiator at all.

Agent:: sale (....)



In the diagram notation, we show an arrow from the initiator to the action and from the action to the receiver if either one is known. Each action can have any number of participants and parameters.

4.2.8 Inputs and Outputs

Within the effects clause, there is no strong difference between participants and parameters: both can be affected by the action. The distinction is intended to document a partial design decision. The participants exist or will be built as separate entities, with some direct or indirect interaction between them to realize the action. Together, the parameters represent information that is passed between the participants, encoded in a form not documented here, and possibly communicated differently.

The concept of inputs and outputs is meaningful only for directed requests: either fully localized operations or joint actions with initiators and receivers, in which the invoker of an operation somehow provides the inputs and uses the outputs. In the case of joint actions that have no distinguished initiators or receivers, the effect is expressed in terms of, and on, all participants, and there is no need to explicitly list inputs or outputs.

The input parameters in a directed action are simply attributes of the initiator in a corresponding undirected joint action; they represent state information known to the initiator when it provides the inputs to a directed request. Similarly, the outputs of a directed action are state changes in attributes of the sender in the joint action. When parameters are used in a joint action, the parameter list represents information exchanged that is not fully determined by attributes in the participants; that is, they provide a degree of nondeterminism.

4.2.9 Abstracting a Single Operation in Code

We have seen how an entire sequence of interactions between objects can be abstracted and described as a single joint action. We will next see how even in program code, an operation invocation itself has two sides: the sender and the receiver. By using input and output parameters, a localized operation specification decouples the effect on the receiver from any information about the initiator.

Consider the following interaction sequence between retailer and wholesaler. The retailer first requests the price of some quantity of the product. It then requests a sale, paying the required amount, and gets as a return a set of items. Let us examine the sell operation. Its spec, based on the type model shown in Figure 4.6, could be

```
action Wholesaler::sell ( prod: Product, q: integer, payment: Money ) : Set(Item)
    pre: -- provided the request product is in our catalog, and payment is enough
         catalog->includes (prod) and payment >= prod.price * q
    post: -- the correct number of items has been returned from stock
         result ->size = q and catalog.stock -= result and cash+= payment
```

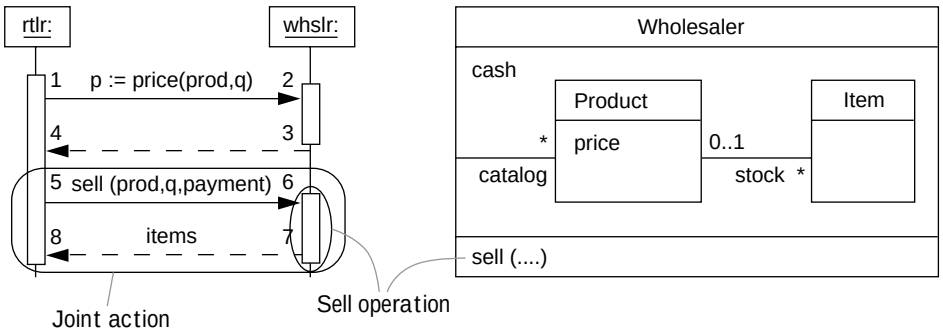


Figure 4.6 Scope of operation versus joint action. The numbers represent increasing points in time.

This spec says nothing about how the values of p, q, and payment are related to any attributes of the retailer. Nor does it say what effect the returned set of items has on the retailer. Although this lack of detail is useful when you're designing the wholesaler in isolation, in the bigger picture of the overall interaction between retailer and wholesaler, those details are important.

The numbers in Figure 4.6 will help you understand what is going on here. The numbers mark increasing points in time (although the separations may be a bit artificial for a procedure-calling model of interactions).

1. Retailer has just issued the first price request.
2. Wholesaler has received that request.
3. Wholesaler has just replied with the requested price.
4. Retailer has accepted the returned price.

5. Retailer has just issued the sell request.
6. Wholesaler has just received that request.
7. Wholesaler has completed processing the request and has just returned the items.
8. Retailer has accepted the items.

When we wrote our specification for Wholesaler::sell, the span we were considering was from 6 to 7, no more and no less. Thus, we ignored all aspects of the retailer in the specification.

If we wanted to describe the effect including points 5 and 8, we would include the fact that the product, quantity, and payment from the retailer are precisely those previously exchanged with the wholesaler (at time 5), and we would increase the inventory of the retailer (step 8). We can introduce attributes on the retailer to describe the product, quantity, and payment known to it at point 5 and the set of items that will be increased at step 8; we can then define a directed joint action.

```
action (r: Retailer -> w: Wholesaler) :: sale( out items: Set(Item))
let (  prod = r.prod,      -- the product the retailer wants
       q = r.qty,          -- the quantity the retailer wants
       pay = r.pay ) in (                -- the amount to be paid
      pre:
            -- provided retailer has enough cash
            r.cash >= pay
            -- and product is available and in stock
            and w.catalog->includes (prod) and prod.stock->count >= q
      post:
            -- payment and inventory of that product have been appropriately transferred
            r.cash -= pay and w.cash+= pay and
            items.product@pre = prod and items->count = q
            r.items += items and w.catalog.stock -= items
)
```

Note that the effect is defined almost completely in terms of attributes of the retailer rather than by parameters that would otherwise be unrelated to retailer attributes. Some of these attributes may even correspond to local variables within the retailer's implementation, in which the product, quantity, and payment due are stored after step 4. The particular set of items transferred has been modeled as an output because the specific items can be determined by the wholesaler in ways not specified here, provided they are of the right product and quantity.

4.3 *Use Cases Are Joint Actions*

Joint actions are useful for describing business interactions, interactions within a software design, and interactions between users and software. We like to separate the specification of an action from its refinement into smaller actions.

The analysis idea of a use case is a joint action specification at the business or user level. When describing a use case, you may prefer a diagram view without the type mod-

els and may want to use a form that looks a bit more like a narrative template for review by customers. Making it precise is still your job.

use case	sale
participants	retailer, wholesaler
parameters	set of items
pre	the items must be in stock, retailer must be registered, retailer must have cash to pay
post	retailer has received items and paid cash wholesaler has received cash and given items -- formal versions hidden

It is also useful to document informally the performance requirements of a use case. Such requirements include whether it is considered a primary or secondary use case (alternatively, priority levels), the frequency with which it is expected to take place, and its concurrency with other use cases.

use case sale
.....

priority	primary
concurrent	many concurrent sales with different wholesaler reps no sale and return by the same retailer at the same time
refinement criteria	-- what to consider when refining this use case into a sequence
frequency	300-500 per day
performance	less than 3 minutes per sale

In the rest of this book we will sometimes explicitly show the informal use case template equivalent of a specification. However, the joint action form can always be represented in the narrative use case template.

© ***use case*** A joint action with multiple participant objects that represent a meaningful business task, usually written in a structured narrative style. Like any joint action, a use case can be refined into a finer-grained sequence of actions. Traditionally, the refined sequence is described as a part of the use case itself; we recommend that it be treated as a refinement even if the presentation is as a single template.

Many use case practitioners recommend listing the steps of the use case as part of its definition; we do not do this because it mixes the description of a single abstract action with one specific (of many possible) refining action sequence. Instead, first describe the use case as a single abstract action without any sequence of smaller steps; document its postcondition. This approach forces you to think precisely about the intended outcome, a step back from the details of accomplishing it.

When you separately refine the action (Chapter 6, Abstraction, Refinement, and Testing), it is useful to document the refinement textually in a use case template.

use case	telephone sale by distributor
refines	use case **sale** (the pre/post of sale could be shown here)
refinement	1. retailer calls wholesaler and is connected to rep 2. rep gets distributor membership information from retailer 3. rep collects order information from retailer, totaling the cost

4. rep confirms items, total, and shipping date with wholesaler
5. both parties hang up
6. shipment arrives at retailer
7. wholesaler invoices retailer
8. retailer pays invoice

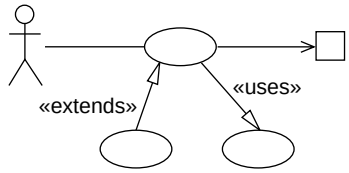
abstract result **sale** was effectively conducted
with amount of the order total and items as ordered

Some leading use case practitioners recommend adding an explicit "goal" statement to a use case. In Catalysis this is taken care of by refinement. Goals can usually be described as some combination of the following.

- *Static invariant:* Whatever happens, this must be true afterward.
- *Action specification:* This action should get me to this postcondition.
- *Effect invariant:* Any action that makes this happen must also ensure that.

Refinement allows us to trace use cases back to the level of such goals.

4.3.1 A Clear Basis for Use Cases



Use case has become a popular term in object-oriented development, where it is defined as "the specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system." This concept is given a solid foundation in Catalysis based on actions and refinements.

In the common literature, a use case is typically a joint action involving at least one object (called an *actor*) outside the system of interest; the granularity of the action is such that it accomplishes an objective for the actor. The use case is refined into a more detailed sequence of actions and is explored with sequence diagrams illustrating that collaboration. Use case diagrams can be used to capture the action refinement. Most current accounts of use cases fail to separate the specific action refinement (one of many that may be possible) from the single abstract action because the use case definition itself includes the specific sequence of steps followed to accomplish the use case; this lack of separation can make it difficult to handle alternative decompositions.

The use case approach also defines two relationships between use cases—*extends* and *uses*—to help structure and manage the set of use cases. These relationships are defined in UML as follows.

© ***extends*** A relationship from one use case to another that specifies how the behavior defined for the first use case can be inserted into the behavior defined for the second use case.

© ***uses*** A relationship from a use case to another use case in which the behavior defined for the former use case employs the behavior defined for the latter.

The extends relation serves two purposes. First, it is used to define certain user-visible behaviors as increments relative to an existing definition—for example, to define different interaction paths based on configurations or incremental releases of functionality. Second,

it does so without directly editing the existing definition—a fancy editing construct. Catalysis meets these objectives within the framework of actions and packages, in which a second package may specify additional behaviors or paths for the same basic service from another package (Chapter 7, Using Packages).

The uses relationship between use cases is meant to let use cases share existing use cases for some parts that are common. There is, however, a conflict between the often-stated goal of having a use case correspond to a user task and the need to factor common parts across use cases. This leads to some confusion and variations in interpretation even among use case consultants. Catalysis provides *actions* and *effects* as the basis for this sharing; "using" another use case means that you use its effect or quote the action itself.

Based on refinement, Catalysis provides a more flexible mapping between abstract actions and their realizations. For example, here are two partially overlapping views of a sale. A customer views sale as some sequence of <order, deliver, pay>. A salesperson may view sale as a sequence of <make call; take order; wait for collection; file commission report; collect commission>. Both views are valid and constitute two different definitions of a sale.

Consider the following example from an Internet newsgroup discussion, which highlighted some of the confusion surrounding a precise definition of use case.

> *A system administers dental patients across several clinics. A clinic can refer a patient to another clinic. The other clinic can reply, accepting or otherwise updating the status of the referral. Eventually, the reply is seen back at the referring clinic, and the case file is updated. Later, the final treatment status of the patient is sent back to the referring clinic. Then there is a financial transaction between the two clinics for the referral.*

Several questions arise.

- Is this one large-grained use case, Refer Patient?
- Are there separate use cases for Send Referral, Accept Referral, Get Acceptance, Final Referral Status, and Transfer Money?
- Suppose that Accept Referral is done by a receptionist printing it from the system and then leaving it in a pile for the dentist to review. The dentist reviews the referral and annotates acceptance. The receptionist then gets back on the system and communicates that decision. How many use cases is that?

In Catalysis, all these actions are valid at different levels of refinement. There is a top-level action called Refer Patient. In our approach, the name you choose for a use case is helpful, but its meaning is defined by the pre- and postconditions you specify for that use case. As with any other action (see Section 3.1.5, Two Kinds of Actions), a use case can be refined into a sequence of finer-grained actions; alternative refinements may also be possible. The steps of a use case are also actions, just as the use case itself is an action.

Use cases give reasonable guidelines on how fine-grained a use case should become, if only at the bottom end of the spectrum: If the next level of refinement provides no meaningful unit of business value or information, do not bother with finer-grained use cases; just document them as steps of the previous level of use case.

4.4 *Actions and Effects*

An effect is simply a name for a transition between two states. We can define joint effects just as we define localized effects in Section 3.8.3.

```
effect (a: A, b: B, c: C) :: stateChangeName (params)
    pre:    ...
    post:   ...
```

Actions and operations describe interactions between objects; an effect describes state transitions. You use effects to factor a specification or to describe important transitions before the actual units of interaction are known. Just as attributes are introduced when convenient to simplify the specification of an operation, independent of data storage, so can effects be introduced to simplify or defer the specification of operations.

Effects can also be used when responsibilities of objects (or groups of objects, in the case of joint effects) are decided but their interfaces and interaction protocols are not yet known. You can then express actions in a factored form without choosing interfaces or protocols.

```
effect Wholesaler::sell (x: Item)
    pre:    -- item must be in stock
    post:   -- gained price of item, lost item

effect Retailer::buy (x: Item)
    pre:    -- must have enough to pay
    post:   -- has paid price of item, gained item
```

The joint action (or another effect) can then be written conveniently using a single post-condition:

```
action (r: Retailer, w: Wholesaler) :: sale (x: Item)
    post:   r.buy (x) and w.sell (x) and r: w.registrants@pre
```

Every action introduces an effect, which can be referred to by *quoting* it. This is a way to use the specification of that action, committing to achieving its effect without committing to specifically invoking it in an implementation.

```
.... [[ (r,w).sale (x) ]]
```

The joint action can also be *invoked*. This means that one of the protocol sequences that realizes the joint action will be executed; specific participants and parameters are bound, but how they are communicated is left unspecified.

```
.... ->(r,w).sale (x)
```

It is not meaningful to “invoke” an effect.

4.5 *Concurrent Actions*

All actions take up some period of time. Some actions are more interesting because of what they do while they are in operation than what they have achieved after they have finished.

For example, a wholesaler and a retailer have an ongoing supply relationship in which the retailer's stock is maintained by regular sales by the wholesaler. We can show that supply consists of a number of sales (see Figure 4.7). A guarantee clause is used to state what the supply relationship means:

```
action     (retailer, wholesaler) :: supply
guarantee  retailer.stock->size >10     -- the retailer's stock will never go below 10
rely       wholesaler.in_business       -- provided the wholesaler is always in busi-
ness
```

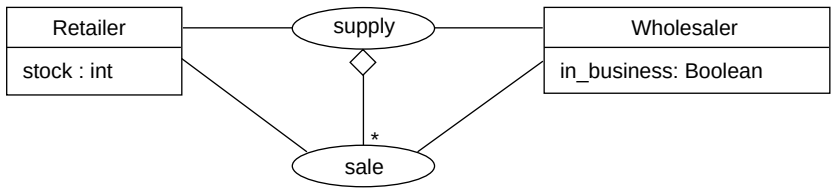


Figure 4.7 A supply activity consists of many sales.

The rely condition qualifies the description, saying that it is valid only if the stated condition remains true through the life of the occurrence. If a concurrent action causes it to be false for any of period of time, then we cannot point to any overlapping period and say, "There is an occurrence of supply." (Not according to this description at least; there may be other descriptions that define what supply means over such a period.)

If you want to define supply to mean that the retailer's stock is maintained whenever the wholesaler is in business even though this might vary during the period of a supply occurrence, you should put it all in the guarantee:

```
guarantee wholesaler.in_business => retailer.stock–> size > 0
```

In some cases, we are interested both in what an action achieves from beginning to end and also what it does while in operation. For example:

```
action     (retailer, wholsaler):: supplyForYear (amount:Money, from:Date, to:Date)
guarantee  retailer.stock–> size < 10 and (from:today<to)⇒
           retailer.orders[source=wholesaler]–>size > 0
post       wholesaler.income +=amount
   and     retailer.outgoings +=amount     -- net effect is transfer of money
rely       wholesaler.inBusiness       --makes no sense if not in business
pre        from < to       --makes no sense if dates aren't in order
```

Both the guarantee and the postcondition are qualified by both the precondition and the rely condition.

4.5.1 Rely and Guarantee

The rely and guarantee clauses contribute to an action speicification as follows:

- An action description has a signature and four clauses: precondition, rely, guarantee, and postcondition.
- A guarantee clause states a condition maintained while an action occurrence is in progress.
- A postcondition states a condition achieved between the beginning and the end of an action's occurrence.
- A precondition states what must be true at the start of an action occurrence for all of this description to be applicable.
- A rely condition states what must be true throughout the action occurrence for all of this description to be applicable.
- There may be more than one description of an action having different rely conditions and preconditions.

4.5.2 Granularity

The guarantee applies only between the beginnings and ends of actions in the next, more detailed level of aggregation. A sale might be implemented as making an order followed by a later delivery; in that case, the retailer's stock might temporarily go below 10, but only while a sale is in progress that will bring it up again.

The rely condition works on the same level of granularity.

4.5.3 Exploring Interference

Sequence diagrams are useful for exploring how concurrent actions may potentially interfere with one another. Suppose that we know that sales consist of making an order, delivering, and paying and that the order is not accepted unless the wholesaler has the items in stock (see Figure 4.8). The * against the sale action signifies that several sales may be in progress at once, although each sale is between one wholesaler and one retailer.

Suppose that a retailer orders 10 widgets; then another retailer orders 15 more; the wholesaler delivers the 10, and then has insufficient widgets to fulfill the order for 15. We can see this kind of situation more easily by drawing sequence diagrams (see Figure 4.9). (Remember that each vertical bar on a sequence diagram is an object, and not a type.) Also it's often useful to draw snapshots of the states, although here it is necessary only to write the single stock attribute.

Perhaps Wholesaler needs an "earmarked for delivery" attribute as well as a stock count. Alternatively, we may prefer to ban concurrent sales or even concurrent transactions of various kinds (see Figure 4.10). Explorations of potential interference scenarios

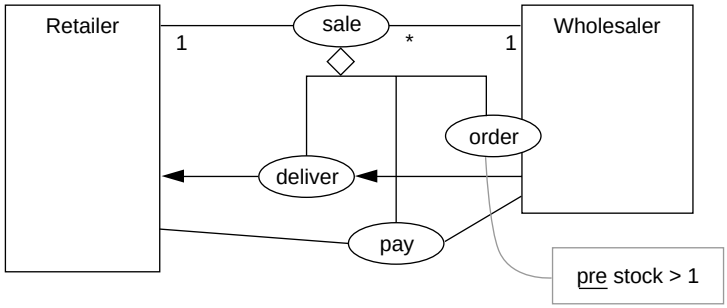


Figure 4.8 Constituent actions for a sale.

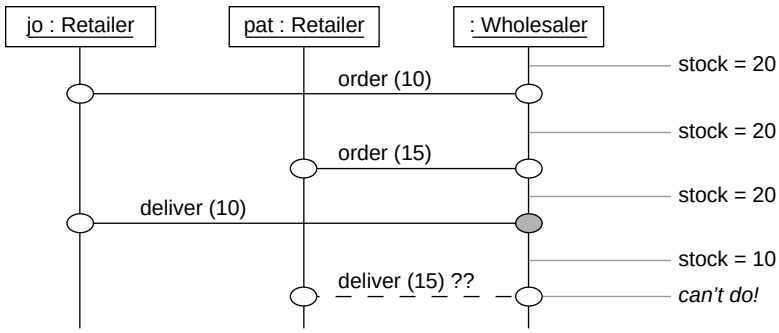


Figure 4.9 Attempted concurrent sales.

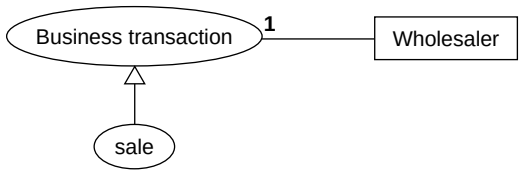


Figure 4.10 Action multiplicity = 1 bans concurrent actions.

are useful, although they are not a fully formal technique. If you are designing an operating system or a nuclear power station, please refer to a text on verifying concurrent designs.

We sometimes want to draw on a sequence diagram the extension of an action over time, indicating that its constituent actions (even if we don't know what they are) are interleaved with those of another action (see Figure 4.11).

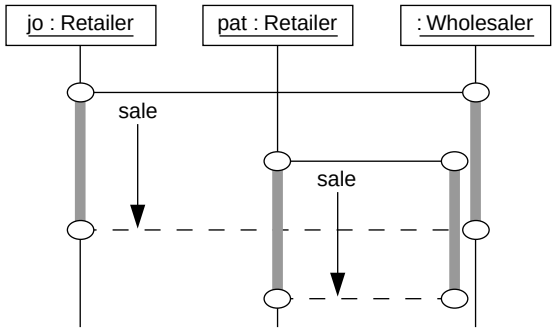


Figure 4.11 The extension of an action over time.

4.5.4 Meaning of Postconditions

Given that actions may overlap, we must take a broader view of the meaning of a postcondition. For example, sale means that, somewhere between the start and the end of the action, the wholesaler's till gets fatter by the price of the items. But does it? If you compare the actual contents of the till before and after the sale, the difference is probably not the same as is documented. It depends on what else has happened during that interval: other sales may have been completed; the manager may have taken the cash to the bank; there may have been a refund or a robbery.

A postcondition therefore means not that the difference between before and after will always be literally what is stated. Instead, it means that if you take into account concurrent actions, the cumulative effect will be what you'd calculate from all the postconditions of the actions. So if you compare the till contents at the beginning and the end of the trading day, you should get a balance equaling the sum of all the sales, bankings, robberies, and so on.

4.6 *Collaborations*

A collaboration is a set of related actions between typed objects playing certain roles with respect to others in the collaboration, within a common model of attributes. The actions are grouped into a collaboration so as to indicate that they serve a common purpose. Typically, the actions are used in different combinations to achieve different goals or to maintain an invariant between the participants. Each role is a place for an object and is named relative to the other roles in the overall collaboration.

For example, the subject-observer pattern uses a set of actions that enables the following: the subject to notify its observers of changes; the observer to query the subject about its state; and the observer to register and deregister interest in a particular subject. These actions, taken as a set, form a collaboration. When several actions have the same participants, it is convenient to draw them on top of one another. The collaboration in Figure 4.12 describes a set of three actions between retailers and wholesalers. This collab-

oration is a refinement of the joint sale use case we specified earlier because particular sequences of these refined actions will realize the abstract action.

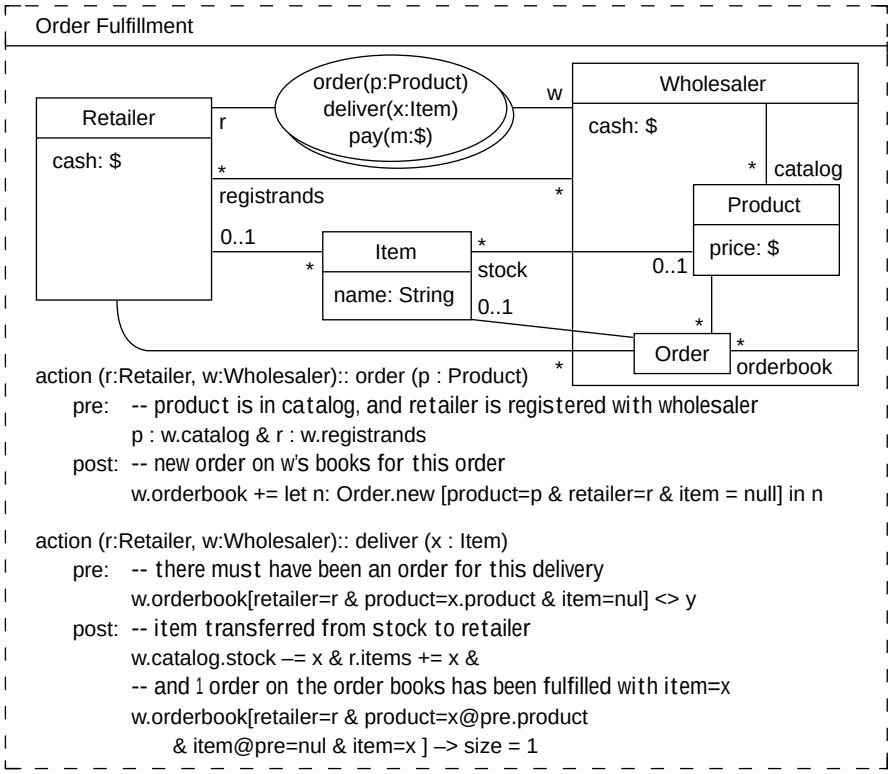


Figure 4.12 A collaboration that realizes the abstract sale.

A collaboration represents how responsibilities are distributed across objects and actions, showing which actions take place between which objects and optionally directing or localizing the actions. The actions are related by being defined against the same model and achieve a common goal or refine a single, more abstract action.

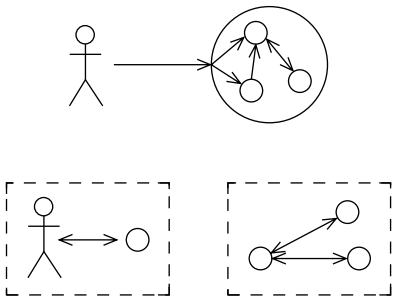
Associated with a collaboration is a set of types: those that take part in the actions. Typically, they are partial views, dealing only with the roles of those objects involved in this collaboration. For example, the retailer in this collaboration may well have another role in which it sells the items to end customers.

Note that a type specification is a degenerate special case of a collaboration spec, one in which all actions are directed and nothing is said about the initiator.

© ***collaboration*** A set of related actions between typed objects playing defined roles in the collaboration; these actions are defined in terms of a common type model of the objects

involved. A collaboration is frequently a refinement of a single abstract action or is a design to maintain an invariant between some objects.

4.7 *Uses of Collaborations*



Collaborations are used to describe designs in two primary forms. In an *enscapsulated* collaboration the behavior of an object can be specified as a type; it can then be implemented, with the object comprising others that collaborate to meet its behavior specifications. Individual classes fall into this category.

In an *Open* collaboration, a requirement expressed as an invariant or joint action can span a group of objects; a collaboration is a design for this requirement. Services (infrastructure services—transactions and directory services—and application-specific ones—spell-checking or inventory maintenance), use cases, and business processes usually fall into this category.

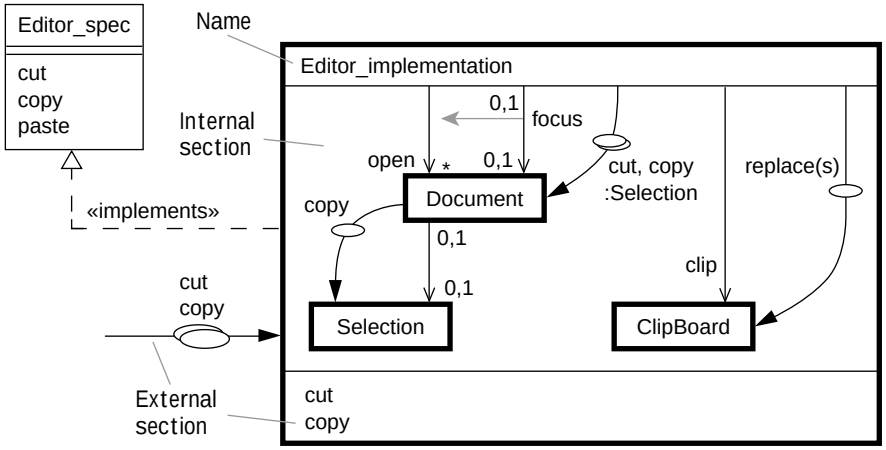


Figure 4.13 A collaboration implementing the Editor type.

4.7.1 Encapsulated Collaboration: Implementing a Type

A collaboration that has a distinguished "head" object can serve as the implementation of a type. Like a type, such a collaboration can appear within a three-part box. The difference is that the middle section now includes actions (directed or not) along with the collaborating types and links (directed or not) between them.

So in fact the "type" is now a class or some other implementation unit (such as an executable program whose instance variables are represented as global variables within the process). The collaboration describes how its internals work. The Editor_implementation type and those within its box are all *design* types (rather than hypothetical *specification* types, as discussed in Section 3.8.6). Any implementation of this collaboration must implement them in order to realize this collaboration.

The collaboration diagram in Figure 4.13 shows five actions indicated by the action ellipses on the lines between types: the external cut and copy operations that the editor must support according to the specification, and the internal cut, copy, and replace actions—between the editor, the focus document, its internal selection, and the clipboard—that will realize it. The lines without ellipses represent type model attributes—now containing directions and eventually denoting specific implementation constructs such as instance variables. As usual, we can choose to show only some of the actions in one appearance of this collaboration and show the remaining actions on other pages; all model elements can be split across multiple diagram appearances.[3] You would normally show all internal actions required for the external actions on that diagram.

4.7.2 Interaction Diagrams

A Catalysis collaboration diagram shows object and action types; it does not indicate what sequence of these internal actions realizes the specified effect of a cut. An *interaction diagram* (Figure 4.14) describes the sequence of actions between related objects that is triggered by a cut operation. It can be drawn in two forms.

- *A graph form:* Actions are numbered in a Dewey decimal manner: 1, 2, 2.1, 2.2, 2.2.1, and so on. For consistency, we prefer to show actions with an ellipse —⬭→; however, directed actions can be shown with simple UML arrows, optionally with a "message flow" arrow next to the action name. This diagram highlights interobject dependencies; sequencing is by numbering. The encapsulated objects could be shown contained within the editor, as in Figure 4.13.
- *A time-line/sequence form:* This diagram highlights the sequences of interactions, at the cost of interobject dependencies; otherwise, it captures the same information as the graph version. Again, multiparty joint actions, such as entire use case occurrences, require an alternative notation to the arrow.

Typically, an interaction diagram shows only two or three levels of expanded interactions, with a specification of the actions whose implementation has not been expanded; presenting too many levels on one drawing can get confusing. Interaction diagrams can also be used at the business level (Section 2.7) and at the level of code (Section 3.3.1).

3. Within the scope of a package, as discussed in Chapter 7, Using Packages.

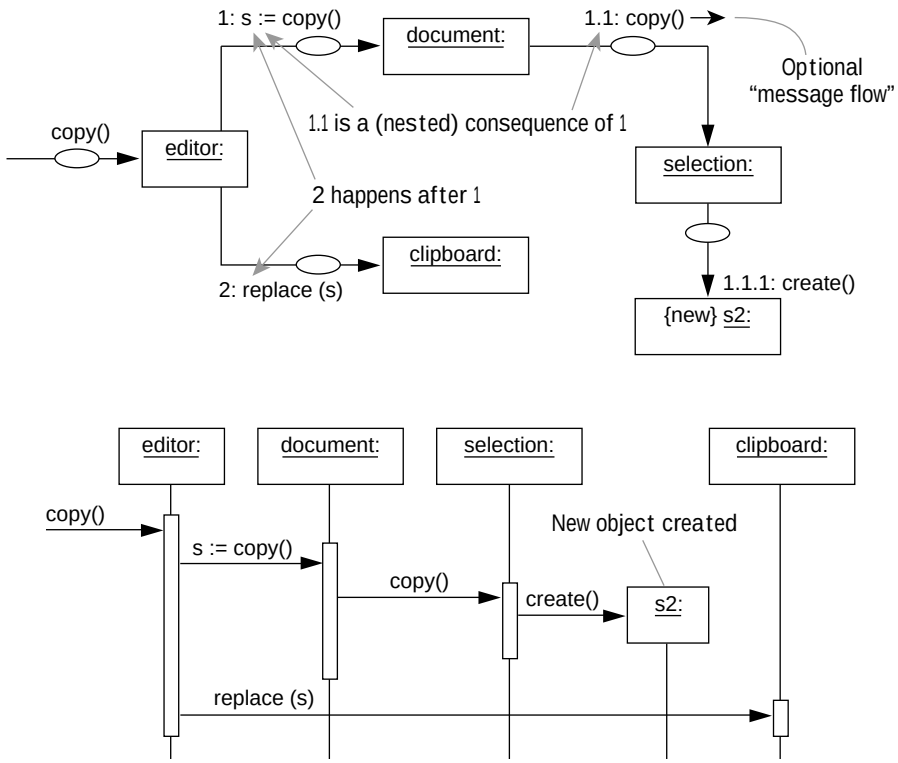


Figure 4.14 Two forms of interaction diagrams.

4.7.3 Sequence Diagrams with Actions

An action occurrence is an interaction between two particular points in time involving specific participant objects bringing about a change of state in some or all of them. An action occurrence is shown pictorially on a scenario or interaction diagram in one of two ways:

- As a horizontal bar (with arrows or ellipses) in a sequence diagram
- As a line with an ellipse —○→ in an interaction graph; sometimes abbreviated to an arrow from initiator to receiver

In the action sequence diagram in Figure 4.15, each main vertical bar is an object. (It is not a type: If there are several objects of the same type in a scenario, that means several bars.) Each horizontal bar is an action. Actions may possibly be refined to a more-detailed series of actions—perhaps differently for different subtypes or in different implementations. The elliptical bubbles mark the participants in each action; there may be several. If there is a definite initiator, it is marked with a shaded bubble.

We use horizontals with bubbles instead of the more common arrows because we want to depict occurrences of abstract actions, even of complete use cases. They often do not

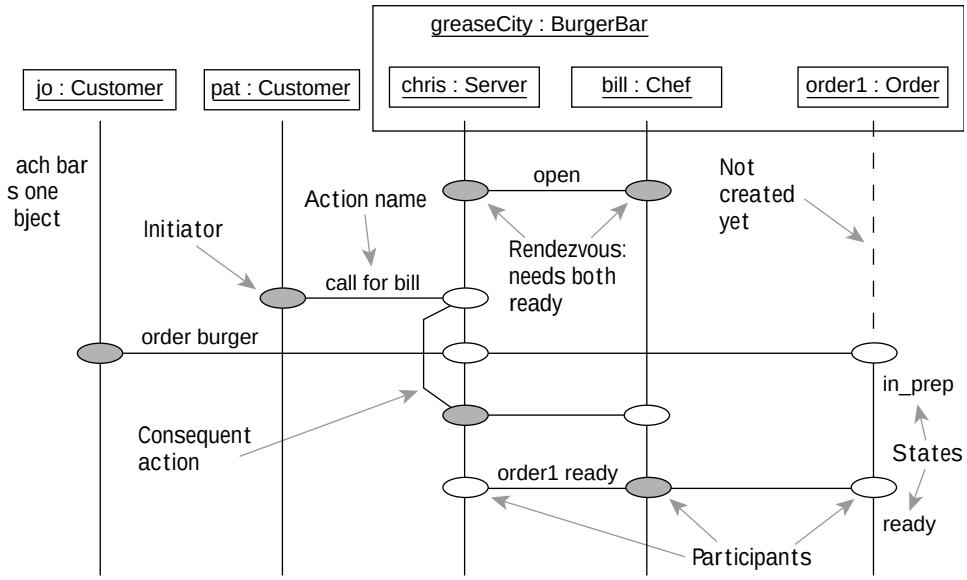


Figure 4.15 Scenario sequence diagram with joint actions.

have a distinguished "sender" and "receiver" and may often involve more than two participants. Arrows are acceptable for other cases, including to illustrate the calling sequence in program code.

Starting from the initial state, each action occurrence in a scenario causes a state change. For joint and localized actions, we can draw snapshots of the state before and after each action occurrence. The snapshots can show the collaborators and their links to each other and to associated objects of specification types.

Other vertical connections show that participation in one action may be consequent on an earlier one. This situation might be implemented as directly, for example, as one statement following another in a program. Or it might be that a request has been lodged in a queue, or it may mean only that the first action puts the object in a suitable state to perform the second.

4.7.4 Scenarios

A scenario is a particular trace of action occurrences starting from a known initial state. It is written in a stylized narrative form, with explicit naming of the objects involved and the initial state, and it is accompanied by one of the forms of an interaction or sequence diagram.

scenario	order fulfillment out of stock
initial state	retailer has no more items of product p1; wholesaler has no inventory of p1 either
steps	

1 retailer places an order for quantity q of p1
2 wholesaler orders p1 from manufacturer m
3 wholesaler receives shipment of p1 from m
4 wholesaler ships q1 of p1 to retailer with invoice
5 retailer pays wholesaler's invoice

4.7.5 Open Collaboration: Designing a Joint Service

Unlike the previous editor example, some collaborations do not have a head object of which they are a part. There are no specific external actions on the objects that are being realized by the collaboration. These collaborations are shown in a dashed box to indicate the grouping. An open collaboration can have all the syntax of an encapsulated collaboration except that there is no "self." Like a type box, an open collaboration has a name, an internal section with participant types and internal actions, and an external section that applies to all other actions.

Figure 4.16 depicts a collaboration for lodging services, showing how responsibilities are distributed across three actions—check-in, occupy, and check-out—and across the two participant types. Lodges provide check-in, initiated by the guest; guests occupy rooms, initiated by the lodge; and check-out can be initiated by either.

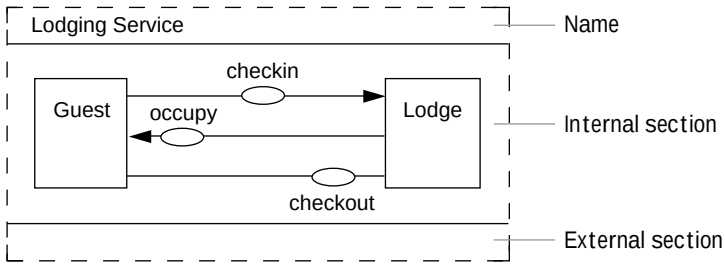


Figure 4.16 An open collaboration.

This style of collaboration is common when each participant is one role of many played by some other object and the collaboration is part of a framework. These generic pieces of design 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 and bulletin boards are based on such collaborations: 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 specialized design ideas that are fitted together to make any system.

As we discussed in Section 2.6, interface-centric design leads us toward treating these open collaborations as design units. Each interface of a component represents one of its roles, which is relevant only in the context of related roles and interactions with others. An open collaboration is a grouping of these roles into a unit that defines one design of a certain service (see Figure 4.17).

Collaborations, including encapsulated designs, are often built by composing open collaborations. The services in an open system are extended by adding new roles to existing objects and introducing new objects with roles that conform to a new service collaboration, subject to the constraints of existing collaborations.

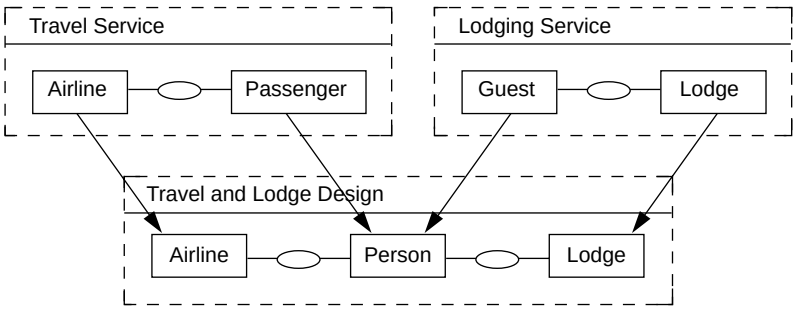


Figure 4.17 Collaborations can be decomposed and recomposed.

4.8 *Collaboration Specification*

Every interacting object is part of a collaboration and usually more than one. Every collaboration has some participants that interact with objects—hard, soft, or live—outside the collaboration. So every collaboration has external actions and internal actions, the latter being the ones that really form the collaboration. In the encapsulated collaboration in Figure 4.13, the external actions include the three specified operations on the editor implementation; in the open collaboration in Figure 4.16, all actions involving the guest or lodge—except the check-in, occupy, and pay actions—are external.

4.8.1 External Actions

If external actions are listed explicitly, they can be depicted as action ellipses outside the collaboration box or listed in the bottom section of the box. Encapsulated collaborations always have explicit external actions. Open collaborations typically have unknown external actions; you do not know what other roles, and hence actions, will affect the objects you are describing. External actions still have specifications in the form of postconditions.

For an encapsulated collaboration, if you wish to repeat an external action's spec (usually already given in the type specification for whatever this implements), it can be written inside the box in terms of self, representing any member of the implemented class. Equivalently, you can write it anywhere, context-prefixed with the class name Editor_implementation :: .

For an open collaboration, you can write an external action's spec outside the box; in that case, you must list the participants and explicitly give them names. External actions often take the form of placeholders in frameworks—actually replaced by other actions

when the frameworks are applied, as described in Chapter 9, Model Frameworks and Template Packages. Or they are constrained by effect invariants as described in Section 4.8.3.

4.8.2 Internal Actions

Internal actions are depicted as actions, directed or not directed, between the collaborators inside the middle section of the box. These actions also have specifications either in the body of the box, with explicit participants, or within the receiver types if they are directed actions. Alternatively, the specs can be written elsewhere, fully prefixed with the appropriate participant information.

4.8.3 Invariants

Because collaborations explicitly separate external from internal actions, you can now define invariants—static as well as effect invariants—that range over different sets of actions. There are two useful cases: ranging only over external actions (internal ones are excluded and need not maintain these invariants) and ranging over all actions, both internal and external.

You can write in the bottom section of the box an invariant that applies to all the external actions. A static invariant would be anded with all their pre- and postconditions; an effect invariant would be anded with all postconditions. This approach is useful for expressing some rule that is always observed when nothing is going on inside the collaboration but that is not observed by the collaborators between themselves. An open collaboration typically cannot list external actions explicitly, because they are usually unknown. Instead, you can use an effect invariant to constrain every external action to conform to specific rules. For example, the external effect invariant in Figure 4.18 states.

> *If any action on a guest causes that guest's intended location (*where*) on the following day to be different from the guest's current lodge location, that action must also cause a check-out to take place.*

This invariant applies to all external actions on a guest; hence, it excludes the checkin and occupy actions themselves. A guest who checks into a lodge when his intended location for the next day is not at that same location will *not* trigger a checkout. However, actions from other collaborations could trigger it: The home_burned_down action from the insurance collaboration, or the cops_are_onto_me action from the shadowy_pursuits collaboration definitely could trigger it.

Invariants can be written inside the middle section of the box and apply to both internal and external actions of this collaboration.

4.8.4 Sequence Constraints

You can draw a state chart showing in what order it makes sense for the actions to occur. This isn't as concrete as a program, because there may be factors abstracted away that permit different paths, and intermediate steps, to be chosen; a program usually spells out every step in a sequence. The state chart is a visual representation of sensible orderings that could equally, if less visibly, be described by the pre- and postconditions of the actions.

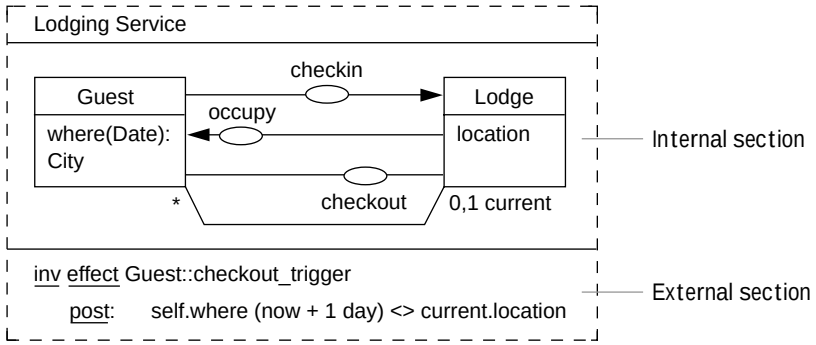


Figure 4.18 An external effect invariant.

This is not the same thing as a state chart showing how different orderings actually result in achieving different abstract actions. The actions of a collaboration may have many possible sequences in which they can sensibly be used, but each abstract action consists of only certain combinations of them. For example, there are many combinations in which it makes sense to press the keys of a UNIX terminal. The keystrokes and the responses you get on the screen are a collaboration. But there is a more abstract collaboration in which the actions are the UNIX commands. To form any one of these commands, you press the keys in a certain sequence given by the syntax of the shell language. The overall sensible-sequences state chart is more permissive than the state chart that realizes any one abstract command.

© ***collaboration spec*** A collaboration is specified by the list of actions between the collaborators, an optional list of actions considered "outside" the collaboration, action specs, static and effect invariants that may apply to either set of actions, and an optional sequence constraint on the set of actions.

4.8.5 Abstracting with Collaborations and Actions

The most interesting aspects of design and architecture involve partial descriptions of groups of objects and their interactions relative to one another. Actions and collaborations provide us with important abstraction tools.

A collaboration abstracts a detailed dialog or protocol. In real life, every action we talk about—for example, "I got some money from the cash machine"—actually represents some sequence of finer-grained actions, such as "I put my card in the machine; I selected 'cash'; I took my money and my card." Any action can be made finer. But at any level, there is a definite postcondition. A collaboration spec expresses the postcondition at the appropriate level of detail—"There's more cash in my pocket, but my account shows less." Thus, we defer details of interaction protocols.

A collaboration abstracts multiple participants. Pinning an operation on a single object is convenient in programming terms, particularly for distributed systems. But in real life—and at higher levels of design—it is important to consider all the participants in an opera-

tion, because its outcome may affect and depend on all of them. So we abstract operations to "actions." An action may have several participants, one of which may possibly be distinguished as the initiator 1. For example, a card sale is an action involving a buyer, a seller, and a card issuer. Similarly, we generalize action occurrences, as depicted in scenario diagrams, to permit multiparty actions, as opposed to the strictly sender-receiver style depicted by using arrows in sequence or message-trace diagrams. A standard OOP operation (that is, a message) is a particular kind of action. The pre and/or post spec of an action may reflect the change of state of all its participants. We can thus defer the partitioning of responsibility when needed.

A collaboration abstracts object compositions. An object that is treated as a single entity at one level of abstraction may actually be composed of many entities. In doing the refinement, all participants need to know which constituent of their interlocutor they must deal with. For example, in the abstract you might say, "I got some cash from the bank," but actually you got it from one of the bank's cash machines. Or in more detail, you inserted your card into the card reader of the cash machine.

Hence, actions and collaborations are useful in describing abstractly the details of joint behavior of objects, an important aspect of any design.

4.9 *Collaborations: Summary*

Collaborations (see Figure 4.19) are units of design work that can be isolated, generalized, and composed with others to make up a design.

To help design a collaboration, we can use different scenario diagrams: object interaction graphs and message sequence diagrams for software; and action sequence diagrams for abstract actions (see Figure 4.20).

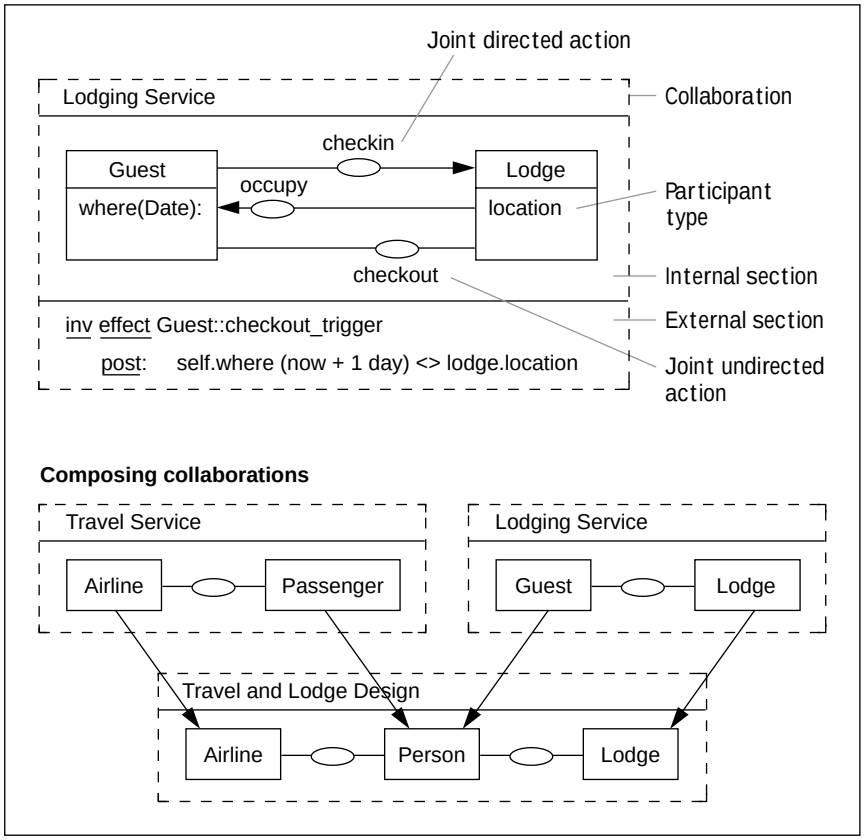


Figure 4.19 Collaborations.

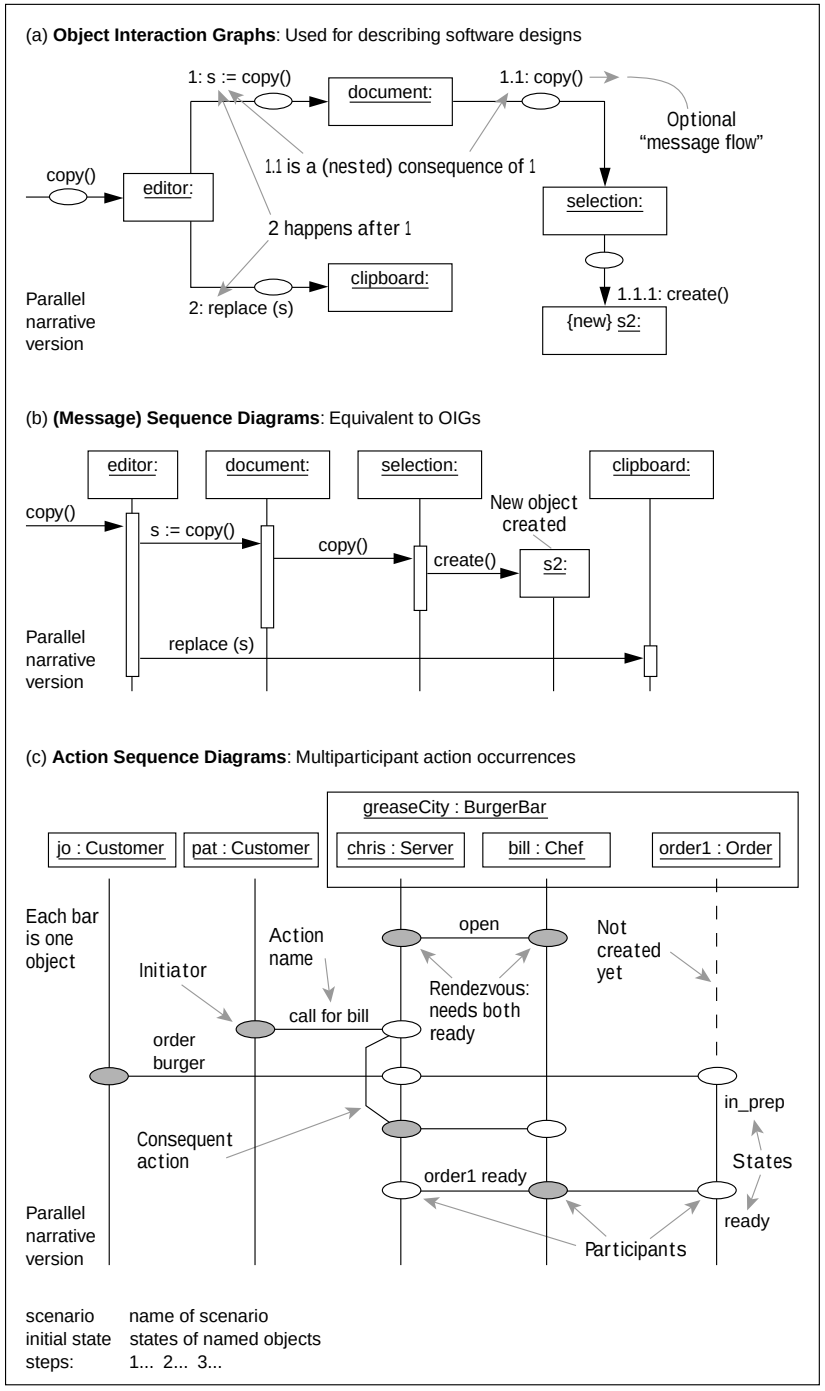


Figure 4.20 Interactions and Scenario Diagrams.