Chapter 5 Interaction Models — Use Cases, Actions, and Collaborations

Outline

Chapters 3 and 4 have described how to model the behaviors of an object by specifying operations in terms of attributes. However, the most interesting aspects of any design lies in the interactions among the objects i.e. how 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, and let you separate abstract multi-party behaviors, joint or localized responsibilities, and actual interfaces and interaction protocols.

Section 5.2 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 5.3 and Section 5.4 introduce the continuum from abstract localized to joint actions and use cases, and discuss how they defer specifics of protocol, parameter passing, and action initiation. A concrete example in code shows how these ideas apply even to detailed design. Section 5.5 discusses how to interpret abstract actions, and relates them to refinement, effects, and use cases.

Collaborations — sets of related actions — are introduced in Section 5.6. Section 5.7 describes how to use collaborations to either describe the encapsulated internal design for some type specification, or an ‘open’ design pattern. The separation of actions internal versus external to a collaboration forms the basis for effectve collaboration models, and is the topic of Section 5.8.

5.1 *Designing object collaborations*

The big difference between object oriented design and the procedural style is that you not only have to make your program work as a sequence of statements: it also has to be well-decoupled so that it can easily be pulled apart, reconfigured, and maintained. You have to make this extra set of decisions about how to distribute the program's functionality between all these little operational units with their own states; and in return — if you do it well — you get all the above benefits.

The big questions in object oriented design:

- What should the system do?
    - — which was the focus of type specification in the previous chapter. Guidance on putting it together in Chapter 16, p.581.
- What objects to choose?
    - — The first draft is the static model we used for the type specification; though modified by design patterns to improve decoupling.
- How should the objects interact?
    - — Most importantly, so as to meet the specification!
    - — Also such as to separate different concerns into different objects; but 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. We agree, though we believe it is useful to get some idea of the requirements first.

In Catalysis, collaborations are therefore first-class units of design, like types. A collaboration is a design for how 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.
- A collaboration represents a design of how a group of objects interact to meet a type specification.

There are several situations in which to use collaborations:

- Designing what goes on inside a software component.
- 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.
- Decribing how 'real world' objects in a business organisation (or a hardware design) interact with one another. This would typically be to help understand the business in which a system is to be installed or updated.

5.2 *Object Interactions — A Need for Abstract Actions*

For any action that involves multiple objects, there are many possible variations in interaction protocols that achieve the same net effect, so we need a way to abstract away from these protocols. Consider the interactions between a retailer and wholesaler, in which the retailer purchases some quantity, q, of a product, prod, and pays for it. Figure 89 depicts 3 different interaction protocols for this transaction; other variations are possible. The solid arrows represent requests, and the dashed arrows represent completion of the request.

We need to abstract away alternate protocols

a. The retailer first queries the price for some product, then requests a sale by providing the product and payment; which returned the items.

b. The retailer requests the sale, and the wholesaler calls back requesting payment of the appropriate amount. When the payment is returned, the wholesaler returns items.

c. This 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.

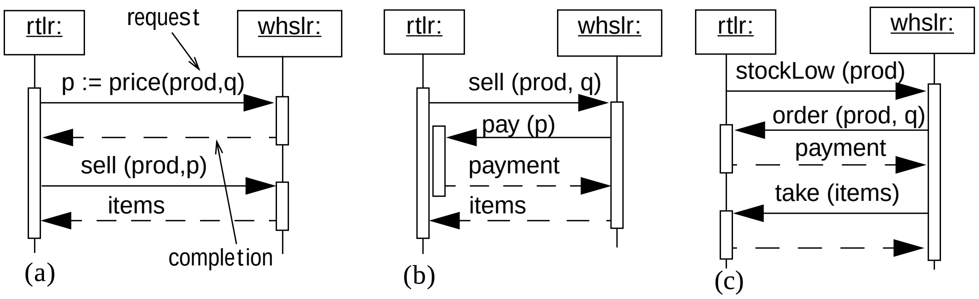


Figure 89: Three protocols for a sale of items

When modeling at the level of point-to-point interactions we are often forced to decide specific protocols e.g. should the retailer first enquire about the cost for a purchase and send the payment along with the request to sell, or should the wholesaler make a callback to the retailer for payment? These differences do not matter at the abstract level.

Point-to-point interactions force detailed decisions

Clearly, these 3 variations all achieve the same basic effect: a sale transaction between retailer and wholesaler, with the same essential information exchange. We could describe the specifics of each interactions, as shown in Figure 89. However, we would like an abstract model that describes all three variations. 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.

All three variations have the same net effect

We could abstract out the details of these interactions in two ways:

An abstract 1-sided ‘sell’ operation

- We could adopt a 1-sided view of the action, in which the wholesaler provides a single service sell, describing what the wholesaler does in a sale without relating it to the retailer, and regardless of the specific protocol. Section 5.3 discusses this.

Or an abstract joint action

- At a coarser and more abstract level, we could describe the overall effect of this interaction with a single joint action, sale, including its effect on both retailer and wholesaler. This is covered in Section 5.4.

This lets us focus on ‘activities’

As we will show, the idea of abstract actions — both joint, and localized — provide a sound basis for modeling that focuses on activities and tasks, rather than on specific interaction protocols for achieving these tasks; and give a principled way to define ‘use cases’. Abstract actions also enable us to describe more general ‘connectors’ between components, as explained in Chapter 16. Chapter 14, *Refinement and Abstraction*, will discuss in detail how different levels of description are related to each other.

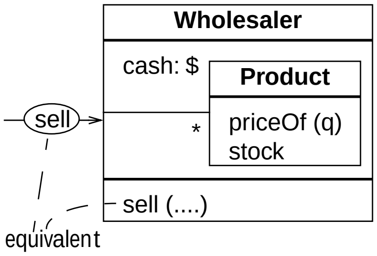
5.3 *Abstract Localized Actions*

An abstract, localized 'sell' action

Let us view sell as an action localized on the wholesaler i.e. an action described completely from the perspective of the wholesaler, ignoring the role played by the retailer. All three protocols variations reduce to a single more abstract request, sell, in which some requestor asks for a given quantity of a product, providing correct payment for it, and is returned the appropriate set of items.

```
-- a wholesaler sells a quantity of a product to some requestor for some payment
action Wholesaler::sell (p: Product, q: Quantity, pay: Money): Set(Item)
    pre: -- provided the payment is correct for that quantity of the product
        pay = p.priceOf (q) and
        -- and the product is in stock
        p.stock >= q
    post: -- an appropriate set of items (correct quantity and product) is returned,
        result.product = p and result->size = q
        -- and stock and money are updated
        p.stock = p.stock@pre - q and cash = cash@pre + pay
```

It ignores the role of the retailer



This only describes the effect this action has on the wholesaler — relating it to price, stock, and quantity; it does not say how this action relates to the retailer e.g. that the product requested is something needed by the retailer, or what the retailer does with the items returned. This is the nature of a localized action; the requesting object remains anonymous and type-less, since we say nothing about how the action relates to its state. The localized action can be shown listed on the bottom of the relevant type box, or as a 'action ellipse' targeted at the receiver, with an anonymous initiator.

Individual types are described this way in code

At the programming level, all actions will be fully localized i.e. operations. When any single object is specified as a type to be implemented, you describe operations that must be implemented to meet this specification, regardless of who invokes them, or why. The invoker can expect the specified outcome — returned values and other changes of state — provided the specified pre-conditions were met— including input parameters and initial state.

Localized actions are refinable until code

Code-level operations will typically not be refined any further before being implemented. Thus, when an operation signature is declared in Java or C++, it is implemented to precisely that signature, assuming a single call and return. Localized actions, in general, can be refined. Thus, with appropriate mappings, any one of the three protocol variations in Figure 89 could achieve the sell action specified above.

Inputs, outputs represent information exchange

If a localized action is subject to refinement, what do the input and output parameters mean? Inputs represent information that is somehow determined or selected by the invoker, through a protocol that is yet to be defined. Outputs represent information that is determined or selected by the receiving object and returned to the invoker, again through some protocol.

Some protocols may exchange extra information

You could design an interaction protocol in which the retailer pays at least the required price, and the wholesaler returns the appropriate change. Should we introduce change as an output parameter in the abstract sell action? Similarly, you could have a protocol in which the retailer first inquires about the catalog of available products before selecting a product to order. Should the catalog then be one of the parameters of the abstract sell action?

Only retain what is relevant to your abstraction

This is a modeling decision you must make; every abstraction is created for a particular purpose, focusing on some aspects while deferring others. If it is important for this purpose to describe change returned then include it in the action spec; otherwise defer it as an artifact specific to a particular refinement. The catalog could also be treated as a detail of a particular refinement i.e. one particular mechanism for the retailer to identify the product of interest.

Decide what refined paths correspond to the abstraction

When such questions arise, first sketch out the abstract action with its specification. Walk through paths in the alternate refinements and decide which paths, with well defined start and end points, constitute the abstract action. Lastly, decide how to abstract the net effect and information exchanged so as to permit alternate refinements, to capture the essence of the transaction at the abstract level.

Some paths will not result in a ‘sell’

There will almost always be some paths through the refined protocol which do not constitute a valid abstract action. For example, in protocol (c), the retailer may decline to make the purchase the wholesaler suggested. Such a sequence would clearly *not* be considered as a sell; it will either be considered insignificant at the more abstract level; or will constitute a different abstract action at that level; or you may model it as an exception to the abstract action. Chapter 14, *Refinement and Abstraction*, describes how to model these mappings and exceptions.

5.4 *Abstract Joint Action = Use Case*

Localized actions provide a 1-sided view of an interaction. Of course, designs are mostly about how these interactions affect all participants involved, achieving some overall desired effect e.g. when a retailer’s inventory of a particular product drops below some threshold, that retailer re-stocks from a wholesaler, affecting attributes on both sides. Joint actions and collaborations serve this role.

Designs are rarely about single objects

The different interaction protocols in Figure 89 clearly depend, and have an effect, on both parties involved — retailer and wholesaler. We need a way to specify the overall effect of any of these interactions on both parties. For this we define a single action, sale, that spans the start to the end of all three interaction variations, and write a specification that abstracts the effect of all three sequences on the objects involved (Figure 90).

A joint action relates and affects all parties

Such an action is called a use case, following [UML]. It can involve multiple participants, has an overall effect, and can be refined into different interaction protocols. In particular, we will prefer the term use case for actions at the business level, where the granularity of the action accomplishes a meaningful objective for one of the participants; and prefer a slightly different form to document them as use cases. The terms are otherwise interchangeable.

A joint action is called a ‘use case’

A use case represents a set of interactions that occur, across history, involving some group of objects. It is characterized by a signature and postcondition, showing the overall effects of the interaction; but abstracts away from the details of exactly what dialogue of operations are performed between the objects.

It’s effect is described like any other action

All the techniques we introduced in Chapter 4 — including snapshots, invariants, attributes and specification types providing a ‘vocabulary’ for action specifications — apply to use cases and joint actions as well; you could try to sketch some before / after snapshots for the joint action sale.

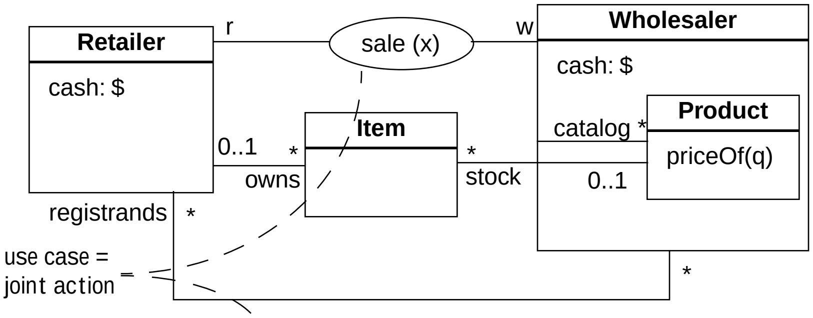
Snapshots and related techniques apply to it

The action is represented within a type model by an ellipse linked to type-boxes, each link representing a ‘participant’. Associated with the action (on the diagram, or in the Dictionary or other documentation) is one or more action-specs: a signature — consisting of named participants and parameter — plus pre and postcondition. Joint actions are often better named as *nouns* e.g. sale, rather than sell.

It has multiple participants

We make a distinction between participants and parameters. Participants are objects that play an active role in the use case, and are listed before the operation name. For joint actions, ‘self’ is a tuple consisting of the participants, except they must be explicitly named. Parameters represent information that must be exchanged or selected to determine the effect of the use case, and that would not be determined fully by the attributes of the declared participants.

We distinguish participants from ‘parameters’



```
action (r:Retailer, w:Wholesaler):: sale ( x: Set(Item) )
    -- retailer, wholesaler are participants; items x is a parameter
    let ( price = prod.priceOf(x->size) ) in (
    pre: -- items were in stock, retailer must be registered with w
        x : w.products.stock and r: w.registrands and
        -- and retailer must have cash to pay for that quantity of items
        r.cash >= price
    post:-- a += b is short for a = a@pre + b; +, - apply to collections
        r.owns += x and r.cash -= price and -- retailer items and cash
        w.catalog.stock -= x and w.cash += price ) -- wholesaler items, cash
```

Figure 90: A joint action, or use case

Joint actions define participant types

The action is part of the definition of the participants’ types, but not of the parameter types. It shows that there is some way, not specified here, in which the participants may conduct a dialogue leading to the achievement of the postcondition, provided the precondition is true at the outset.

Its effect relates attributes of all parties

Note that the effect is specified in terms of both participants. The set of items sold has become a part of the owns set of the retailer, and is no longer in the wholesaler’s stock. In contrast, the localized action described in Section 5.3 treats this set of items as an output, and does not say anything about what the (anonymous) retailer does with it.

It is still a model

Being a model, there will always be alternate ways to describe the same action; and alternate abstractions that may capture different aspects for different reasons. For example, we could just as well describe the sale action with two parameters: the product being sold, and its quantity. The two descriptions would be equivalent; any refined protocol that realized one would also realize the other.

Use case template

When describing a use case (a joint action at the business level), you may prefer a diagram view without the type models; and use a form that looks a bit more like a narrative template for review by customers; making it precise is still your job:

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

Auxiliary use case information

It is also useful to document informally the performance requirements on a use case, whether it is considered a primary or secondary use case (alternately, priority levels), the frequency with which it is expected to take place, and 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
    performanceless than 3 minutes per sale
```

Through 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.

5.4.1 From Joint to Localized Actions

Joint actions can have initiators and receivers

A joint action is interesting because its effect says something important about all participants. Although some joint actions, such as the one in Figure 90, do not designate any participant as an initiator, in general you can designate initiator and receiver. Here are some variations of a 3-way use case, sale, between a retailer, wholesaler, and agent. We use the convention that variables names implicitly define their types.

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

> This represents a joint action with 3 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 effect clause; and others as ‘outputs’ — determined by other participants, utilized by the initiator in ways not fully specified in the effects clause.

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

> A ‘directed’ joint action which 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. Once 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 alternate arrangement might leave the choice of wholesaler to the agent, appearing only in the effect clause based on some attributes of the agent.

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

> A directed joint action initiated by some object, 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
```

<u>initiator</u>	retailer	-- also listed as participant
<u>receiver</u>	wholesaler	
<u>parameters</u>	set of items	-- can separate inputs/outputs for directed actions

A localized operation is a special '1-sided' case

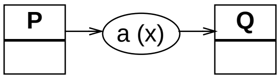
A localized 'operation' is a degenerate case of a joint action with a distinct receiver, in which nothing is known or stated about the initiator's identity or attributes. All relevant aspects of the initiator are abstracted into into the input and output parameters of the operation.

Agent:: sale (....)

A fully localized operation which cannot refer to the initiator at all.

Diagram arrows

In the diagram notation, we show an arrow from the initiator to the use case, and from the action to the receiver, if either one is known.



5.4.2 Inputs and Outputs

'Participants' document a partial design choice

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 / will be built as separate entities, with some direct or indirect interaction between them to realize the action; and the parameters together represent information that is passed between the participants, encoded in a form not documented here, and possibly communicated differently.

Inputs and outputs only apply to directed actions (joint or localized)

The concept of inputs and outputs are only meaningful for directed requests — either fully localized operations or joint actions with initiators / receivers, where the invoker of an operation somehow provides the inputs, and utilizes the outputs. In the case of joint actions without distinguished initiators or receivers, the effect is expressed in terms of, and on, all participants, and there is no need to explicit list inputs or outputs.

Parameters in joint actions provide non-determinism

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

Advanced Topic

5.4.3 Abstracting a single operation in code

Let us examine some code level interactions

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 to it: the sender, and the receiver. A localized operation specification de-couples the effect on the receiver from any information about the initiator, by using input and output parameters.

Consider this 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 91, could be:

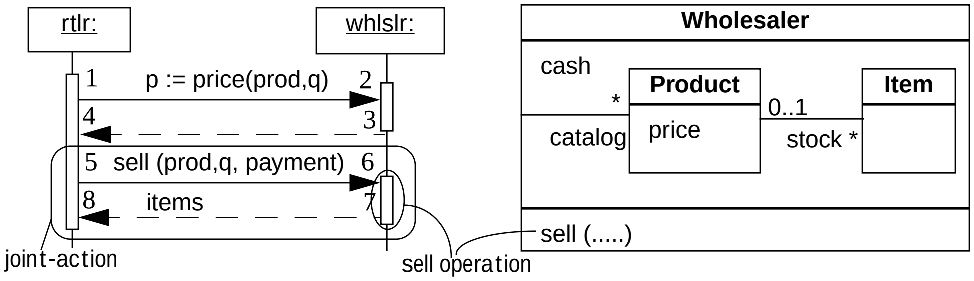


Figure 91: Scope of operation vs. joint action

```
action Wholesaler::sell ( prod: Product, q: integer, payment: Money ) : Set(Item)
    pre: -- provided the request product is on 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
```

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. While this is quite useful when designing the wholesaler in isolation, of course, in the bigger picture of the overall interaction between retailer and wholesaler, those details are quite important.

We identify distinct points in the interactions

The numbers in Figure 91 will help 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 and 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 just returned items

8: retailer has accepted the items

The operation spec has very narrow scope

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

A broader scope needs a joint action

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 previous exchanged with the wholesaler (at time 5), and 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, and 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 qantity 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 has 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
)
```

Specify in terms of attributes of the retailer

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, since the specific items can be determined by the wholesaler in ways not specified here, provided they are of the right product and quantity.

5.5 *What do Use Cases and Abstract Actions mean?*

Abstract actions represent multi-party interactions, deferring many details of the interaction. They have two important interpretations: a retrospective one, by which you can examine a detailed history of objects and protocol-specific interactions and determine what abstract actions actually took place; and a teleological one, which acts more as a prescription on design yet to be done.

There are 2 meanings to an abstract action

5.5.1 Retrospective View — on Object History

Imagine looking at the entire detailed histories of your objects and trying to determine where, in this history, you had occurrences of different abstract actions. To discover whether you have an occurrence of a sale use case anywhere, search for object states using the following criteria:

You can retrospectively identify abstract actions in object histories

- You need to match an object, across some part of its history, to each link emerging from the action ellipse. They need not be separate objects, though often the logic of the pre or postcondition implies that they must be. For example, sale insists that the cash attributes of its participants change in different directions, so in this case matching one object to both participants wouldn't satisfy the postcondition.
- The type-boxes at the ends of each link constrain the types of the participants. Reject all permutations that are of the wrong types.
- For the parameters, you need to find a set of objects constituting a valid model-refinement of the parameter list. That is, a set of objects that contain the information implied by the parameter list. In the steps below about matching states to pre and postconditions, you will apply the abstraction function to retrieve the parameters from the refinement. Model refinement and the abstraction function are discussed in detail in Chapter 14.
- Choose a start-time and an end-time, and look at the states of your candidate participants and parameters at those times.
- Both the before-state and the after-state must satisfy the invariants of the model in which the action is defined: including both explicitly-written invariants and things like cardinalities of links, types of attributes.
- The before states must satisfy the action precondition; and the before and after states must satisfy the action postcondition.

5.5.2 Designer View — what should be built

The action mandates the designer(s) of Retailer and Wholesaler the joint responsibility to work out a way that they can perform buy together. It might consist of one operation or a long dialogue, and may involve other objects, we don't care. It must be capable of being put into operation when the precondition (and invariants in the model of which this action-spec is a part) is true. We haven't specified in this case which of them will be required to initiate it, although we certainly could.

The action prescribes some constraints on upcoming design

A joint action does not indicate how responsibilities are distributed between the participants. It is specifically intended for representing the important decision about there being some interaction involving these participants, not necessarily directly between them, and abstracting from the detail of who does what.

5.5.3 Refining an Action or Use Case

A preview of action refinement

Abstract actions and use cases will be refined. Chapter 14 will discuss refinement in detail, but we include a short discussion here to show how abstraction and realization levels are related to each other in the context of actions.

An abstract re_stock on WH_A

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

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

Realized by 3 finer-grain actions on WH_B

One realization of this warehouse, WH_B, provides a sliding door which can be moved to a particular product before opening; then items are added to that product shelf one at a time before closing the door. 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

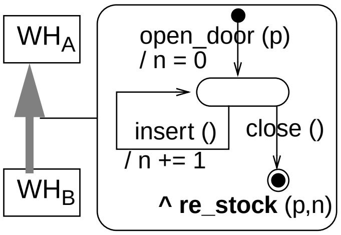
action WH_B::close ( )
post: selected_product = null

Is is clear that a certain sequence of these detailed actions constitute a valid re_stock action. Thus the sequence:

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

The refinement relation maps the action sequence

constitutes an abstract action: re_stock (p, n). 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 re_stocking in progress at the more detailed level.



5.5.4 Refinement and use case documentation

Some innovative use case practitioners recommend adding an explicit statement of ‘goal’ to a use case. In Catalysis this is taken care of by refinement. Goals can usually be described as some combination of:

A use case ‘goal’ means a refinement of invariant, action spec, ...

- invariant: what ever happens, this must continue to be true after
- action specification: an action that should get me to this postcondition
- effect invariant: any action that makes this happen, must also ensure that.

Refinement allows us to trace action refinements back to the level of such ‘goals’.

Likewise, many use case practitioners recommend listing the steps of the use case as part of its definition; we do not do this, as it mixes the description of a single abstract action, with one specific (out of many possible) refining action sequence. Instead, first describe the use case as a single abstract action without any sequence of smaller steps (Section 5.4, “Abstract Joint Action = Use Case,” on page 211); document its postcondition. This forces you to think about the intended outcome precisely, a step back from the busy details of accomplishing it.

Use case steps should only be documented as a refinement

When you do separately refine the action (Chapter 7, *Refinement*), it is useful to document the refinement textually in a use case template.

use case	telephone sale by distributor
refines	use case **sale**
refinement	1. retailer calls wholesaler and is connected to rep
	2. rep gets distributor memberhip information from retailer
	3. rep collects order information from retailer, totalling the cost
	4. rep confirms items, total, and shipping date with wholesales
	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

5.5.5 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 4.7.5.

Effects can name joint transitions

```
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 as 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.

An effect is not an interface operation

Localized effects can describe “responsibilities”

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. Actions can then be expressed 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 postcondition:

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

“Quoting” refers to the effect of an action

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) ]]
```

“Invocation” commits to an interaction

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

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

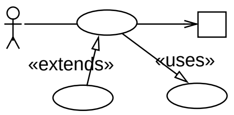
It is not meaningful to “invoke” an effect.

5.5.6 A Clear Basis for Use Cases

Use cases have a precise definition in Catalysis

The use case has become a very popular term in object-oriented development. A use case 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 upon actions and refinements.

A use case combines an action with a particular set of 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, where the granularity of the action is such that it accomplishes some objective for that actor. The use case will be refined into a more detailed sequence of actions, and 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, since the use case definition itself includes the specific sequence of steps followed to accomplish that use case; this can make it quite difficult to handle alternate 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 the UML as:

‘uses’ and ‘extends’ relationships

extends: A relationship from one use case to another, specifying 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. Firstly, to defne 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; secondly, to do so without directly editing the existing definition — a fancy editing construct. Catalysis meets these objectives within the framework of actions and packages, where a second package may specify additional behaviors or paths for the same basic service from another package (Chapter 8, *Packages*).

‘extends’ corresponds to refinement and packages

The uses relationships 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 oft-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 you use its effect, or quote the action itself.

‘uses’ is achieved with actions and effects

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 a sale as some sequence of <order, deliver, pay>. A salesperson may view a 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.

Refinement provdides further flexibility

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

How many use cases is this?

> *A system administers dental patients across several clinics. A clinic can refer a patient to another clinic. The other clinic can reply back, accepting or otherwise updating the status of the referral. Eventually, the reply is seen back at the referring clinic, and the case file updated. Later the final treatment status of the patient is sent back to the referring clinic. Lastly 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, Transfer Money?
- What if Accept Referral was actually done by a receptionist printing it from the system, then leaving it in a pile for the dentist to review. The dentist reviews, and annotates acceptance. The receptionist then gets back on the system and communicates that decision. How many use cases is that?

Actions and refinements solve this problem

In Catalysis, all these are valid actions 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 almost secondary; its meaning is defined by the pre / postconditions you specify for that usecase. Just like any other action (Section 4.3.5, “Actions and Operations Defined,” on page 141), a use case can be refined into some sequence of finer-grained actions; and alternate refinements may be possible as well. The steps of a use case are also actions, just as the use case itself is an action.

Use cases should provide business value

Use cases gives reasonable guidelines on how fine-grained a use-case should get, 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.

5.6 *Collaborations*

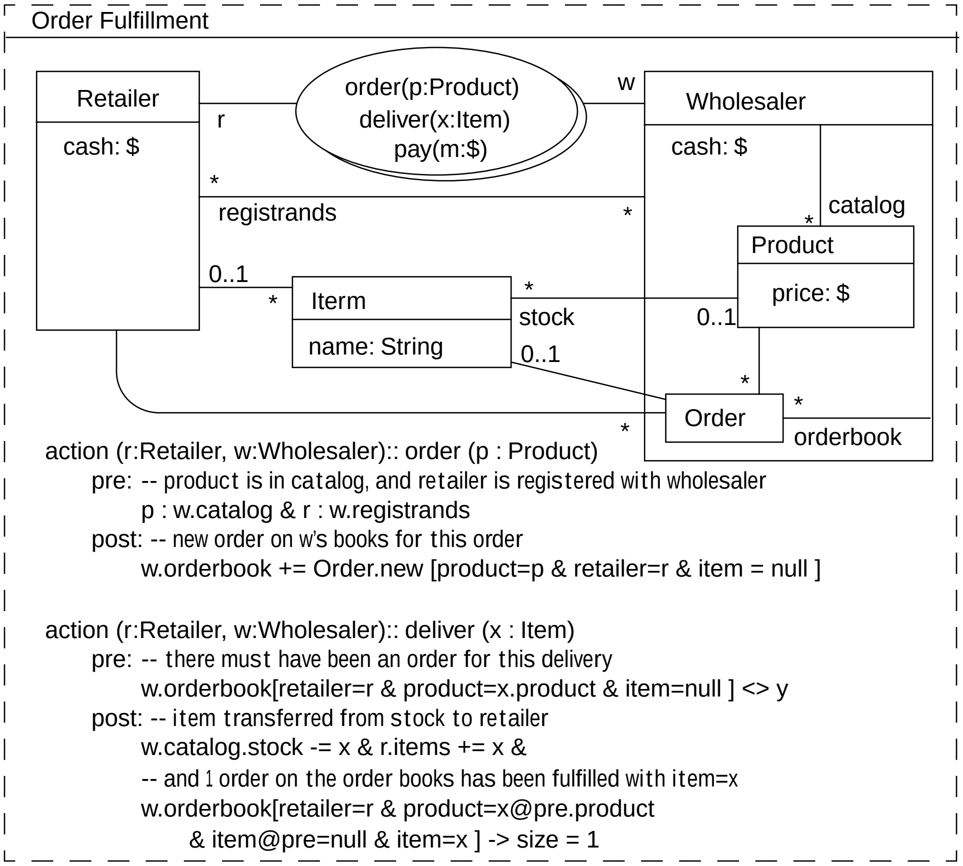
A collaboration is a set of related actions with typed participant roles

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 together into a collaboration so as to indicate that they serve a some common purpose. Typically, the actions will be used in different combinations to achieve different goals or maintain some 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 enabling the Subject to notify its Observers of changes; and the Observer to query the Subject about its state; and enabling the Observer to register and deregister interest in a particular Subject. These actions taken as a set form a ‘collaboration’.

‘order, deliver, pay’ is a refinement of ‘sale’

Where there are several actions with the same participants, it is convenient to draw them on top of one another. The collaboration in Figure 92 describes a set of 3 actions between retailers and wholesalers. This collaboration is a refinement of the joint sale use case we specified earlier, since particular sequences of these refined actions will realize the abstract action.



```
action (r:Retailer, w:Wholesaler):: order (p : Product)
      pre: -- product is in catalog, and retailer is registered with wholesaler
            p : w.catalog & r : w.registrands
      post: -- new order on w's books for this order
            w.orderbook += Order.new [product=p & retailer=r & item = null ]

action (r:Retailer, w:Wholesaler):: deliver (x : Item)
      pre: -- there must have been an order for this delivery
            w.orderbook[retailer=r & product=x.product & item=null ] <> y
      post: -- item transferred from stock to retailer
            w.catalog.stock -= x & r.items += x &
            -- and 1 order on the order books has been fulfilled with item=x
            w.orderbook[retailer=r & product=x@pre.product
                  & item@pre=null & item=x ] -> size = 1
```

Figure 92: A collaboration that realizes the abstract “sale”

Collaborations distribute responsibility

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

Participant type is just one role of some object

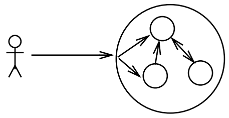
Associated with a collaboration is a set of types — those that take part in the actions. Typically, they will be partial views, just dealing with the roles of those objects involved in this collaboration. For example, the retailer in this collaboration may well have another role in which it sells the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

As a side note, 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.

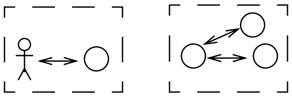
5.7 *Uses of Collaborations*

Collaborations are used to describe designs, in two primary forms:

Collaborations describe encapsulated or ‘open’ designs



Enscapsulated: the behavior of an object may be specified as a type; it can then implemented with that object being comprised of some others, collaborating to meet its behavior specifications. Individual classes fall into this category.



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

5.7.1 Encapsulated Collaboration — Implements a Type

A type is implemented by a collaboration

A collaboration with a distinguished ‘head’ object can serve as the implementation of a type. It can appear within a three-part box, like a type: 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.

The roles now represent ‘design’ types

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 4.10) — any implementation of this collaboration will need to implement them in order to realize this collaboration.

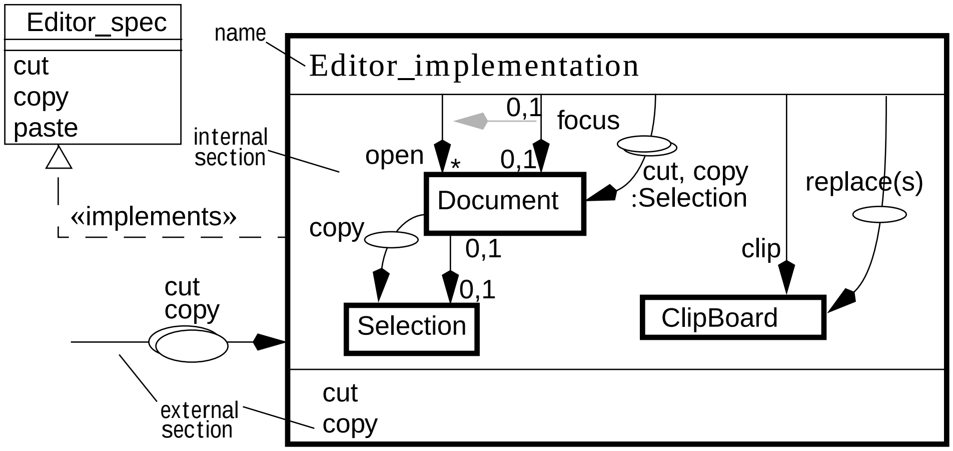


Figure 93: A collaboration implementing the Editor type

This collaboration diagram shows 5 actions, recognized by the action ‘ellipse’ on the lines between types: the external cut and copy operations which 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 with directions on them — and eventually denoting specific implementation constructs such as instance variables. As usual, we can choose to show just some of the actions in one appearance of this collaboration, and show the rest on other pages; all model elements can be split across multiple diagram appearances[1]. You would normally show all internal actions required for the external actions on that diagram.

7.1.1 Interaction Diagrams

Interaction diagrams show action sequences

This collaboration diagram shows object and action types; it does not indicate what sequence of these internal actions realize the specified effect of cut. An interaction diagram (Figure 94) describes the sequence of actions between related objects that are 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, etc. 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 itself. This diagram highlights inter-object dependencies; sequencing is by numbering. The encapsulated objects could be shown contained within the editor, as in Figure 93.
- A time-line / sequence form: this diagram highlights the sequences of interactions, at the cost of inter-object dependencies; otherwise, it captures the same information as the graph version. Again, multi-party joint actions, such as entire use case occurrences, require an alternate notation to the ‘arrow’ (Section 5.8.5).

Typical use of an interaction diagram will show just two or three levels of expanded interactions, with a specification of the actions whose implementation has not been expanded; more 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 4.4.1).

[1] Within the scope of a package, as discussed in Chapter 12

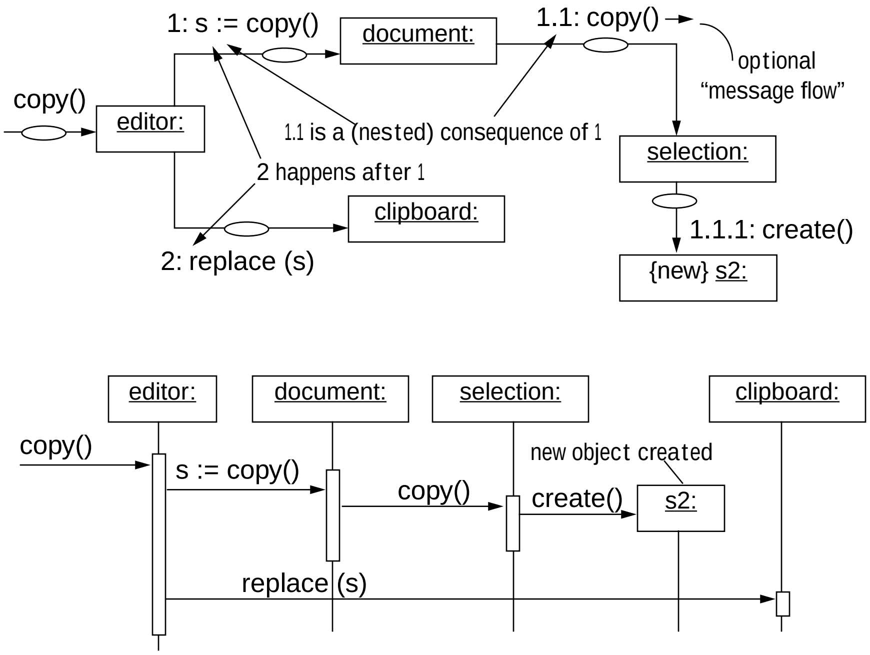


Figure 94: Two forms of interaction diagrams

5.7.2 Open Collaboration — Design a Joint Service

Collaborations can also represent pure design patterns

Some collaborations do not have a 'head' object of which they are a part, like the preceding editor example; 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, it has a name, an internal section with participant types and internal actions, and an external section that applies to all other actions.

Figure 95 depicts a collaboration for lodging services, showing how responsibilities are distributed across three actions — checkin, occupy, checkout; and across the two participant types — lodges provide checkin, initiated by the guest; guests occupy rooms, initiated by the lodge; and checkout may be initiated by either.

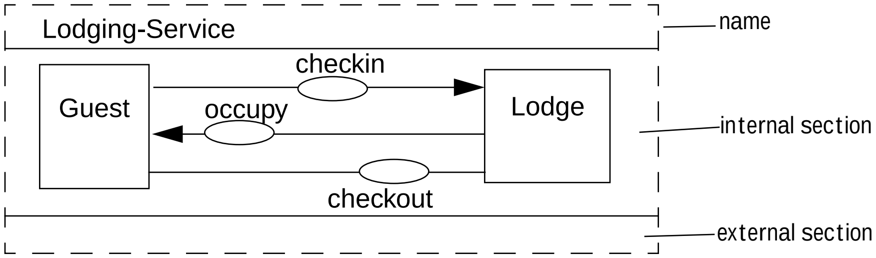


Figure 95: An "open" collaboration

The participant types represent related roles

This style of collaboration is common where 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 around 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 speciallised design-ideas that are fitted together to make any system.

We we discussed in Section 2.6, interface centric design leads us towards 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.

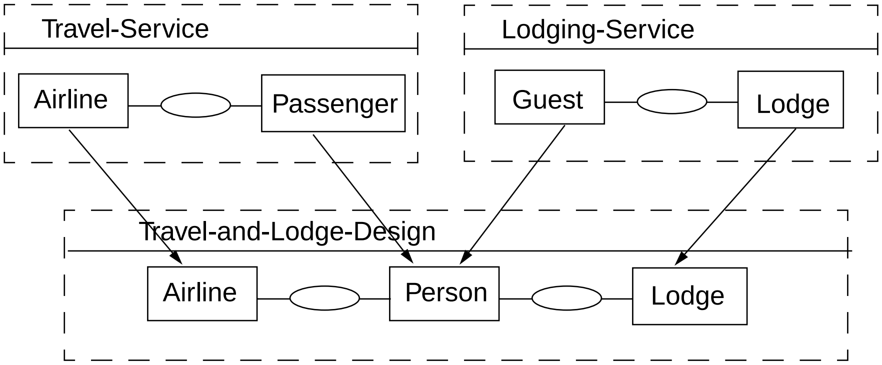


Figure 96: Collaborations can be de-composed and re-composed

Collaborations can be composed

Collaborations, including encapsulated designs, will often be 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.

5.8 *Collaboration Specification*

Every interacting object is part of some collaboration, 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 93, the external actions include the three specified operations on the editor implementation; in the 'open' collaboration in Figure 95, all actions involving the guest or lodge, excepting the checkin, occupy, and pay actions, are external.

A collaboration separates internal actions from external

5.8.1 External actions

If these are listed explicitly, they can be depicted as action ellipses outside the collaboration box; or may be listed in the bottom section of the box. Encapsulated collaborations will always have explicit external actions. Open collaborations will 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 post-conditions.

External actions are not explicitly listed for open collaborations

For an encapsulated collaboration, if you wish to repeat an external action's spec (it will usually have already been given in the type specification for whatever this is an implementation of), it can be written inside the box, written 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 :: ".

External actions are specified in a type spec...

For an 'open' collaboration, you can write an external action's spec outside the box; in that case, you have to list the participants and give them names explicitly. External actions often take the form of 'placeholders' in frameworks — actually replaced by other actions when the frameworks are applied, as described in Chapter 15; or they are constrained by effect invariants, as described below in Section 5.8.3.

Or by constraints on placeholder or unknown actions

5.8.2 Internal actions

These are depicted as actions — directed or not — 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. Alternately, the specs can be written elsewhere, fully prefixed with the appropriate participant information.

The main collaboration actions are internal

5.8.3 Invariants

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

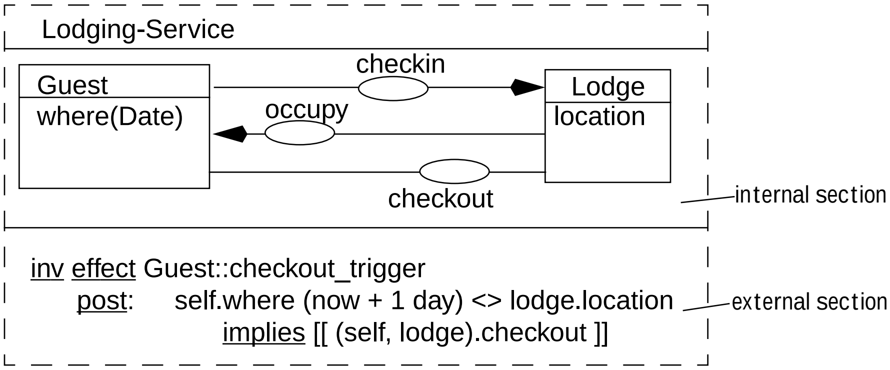
Invariants range over all actions, or only external ones

You can write an invariant that applies to all the external actions in the bottom section of the box. A static invariant would be **and**'ed with all their pre and postconditions; an effect invariant would be **and**'ed with all postconditions. This is very 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.

Effect invariants are useful in open collaborations

An open collaboration typically cannot list external actions explicitly, since these 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 97 states:

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



```
inv effect Guest::checkout_trigger
    post:    self.where (now + 1 day) <> lodge.location
                  implies [[ (self, lodge).checkout ]]
```

Figure 97: An external effect invariant

They constrain unknown actions

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, most definitely could.

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

5.8.4 Sequence constraints

Internal actions may have general sequence constraints

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

This is not the same as a statechart 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 just certain combinations of them. For example, there are many combinations in which it makes sense to hit 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, you hit the keys in a certain sequence, given by the syntax of the shell language. The overall sensible-sequences statechart is more permissive than the statechart that realizes any one abstract command.

Specific sequences will be realizations of abstract actions

5.8.5 Role-Activity Diagrams

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. A scenario is a particular trace of action occurrences, starting from a known initial state. An action-occurrence is shown pictorially:

Scenarios show actions traces and state changes

- as a horizontal bar (with arrows or ellipses) in a role-activity diagram ;
- as a line with an ellipse —○—► in an interaction graph; sometimes abbreviated to just an arrow from initiator to receiver.

In the role-activity diagram below, each main vertical bar is an object (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 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.

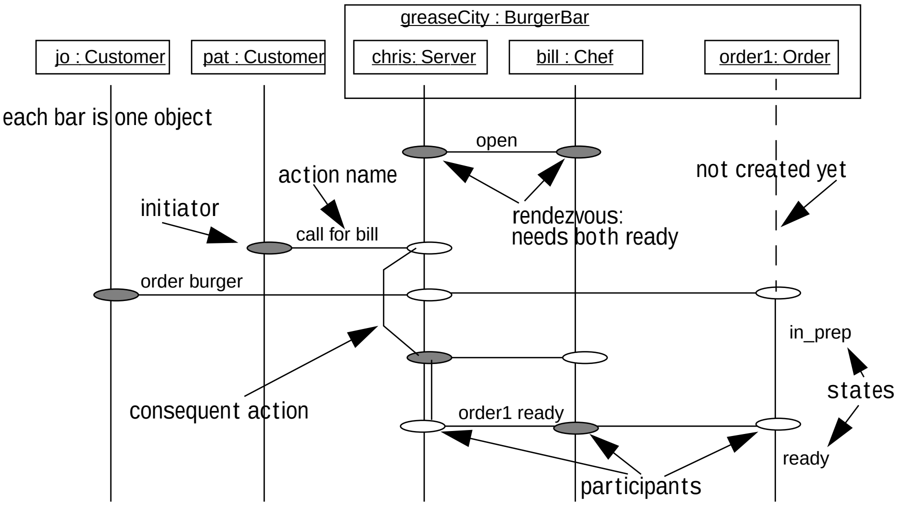


Figure 98: Scenario sequence diagram with joint actions

Actions bubbles are sometimes abbreviated as arrows

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 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 some state change. We can draw snapshots of the state before and after each action occurrence, for joint and localized actions. 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 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 just mean that the first action puts the object in a suitable state to perform the second.

5.8.6 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 each other. Actions and collaborations provide us with important abstraction tools:

- A collaboration abstracts detailed dialogue 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 e.g. "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 operation, since its outcome may affect and depend on them all. 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, seller, and card-issuer. Likewise, we generalize action-occurrences, as depicted in sce-nario diagrams, to permit multi-party actions, as opposed to the strictly sender-receiver style depicted by using arrows in sequence or message-trace diagrams. A standard OOP operation (= message) is a particular kind of action. The pre / post spec of an action may reflect the change of state of all of 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. In doing the refinement, all particpants need to know which constituent of their interlocutor they must deal with. For example, in abstract "I got some cash from the bank" — actually, you got it from one of the bank's cash machines. Or in more detail, you inserted your card in 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.

5.9 *Collaborations — summary*

Collaborations are units of design work that can be isolated, generalised, 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.

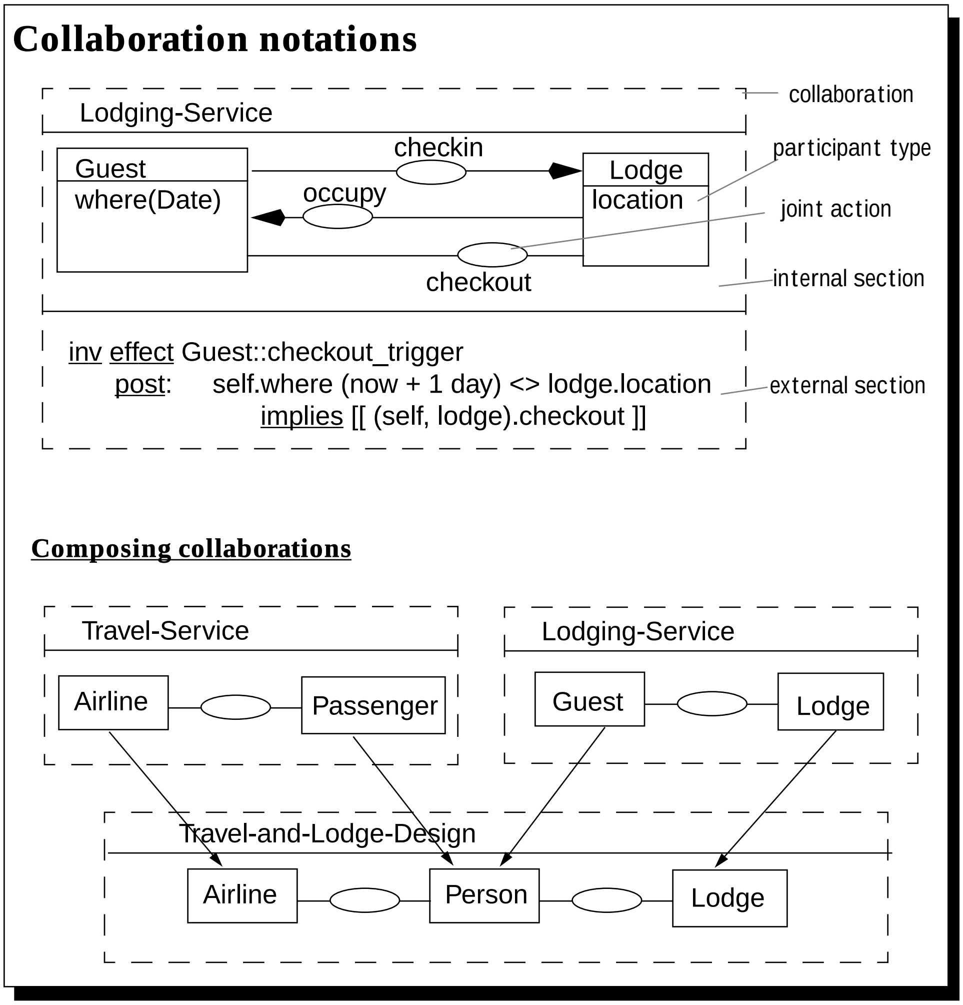
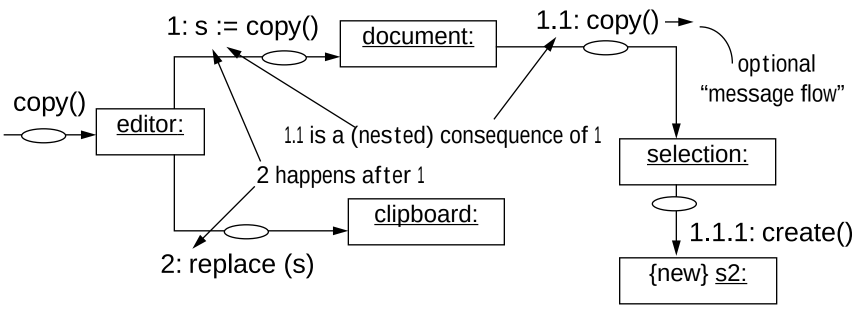


Figure 99: Collaborations

Scenario notations

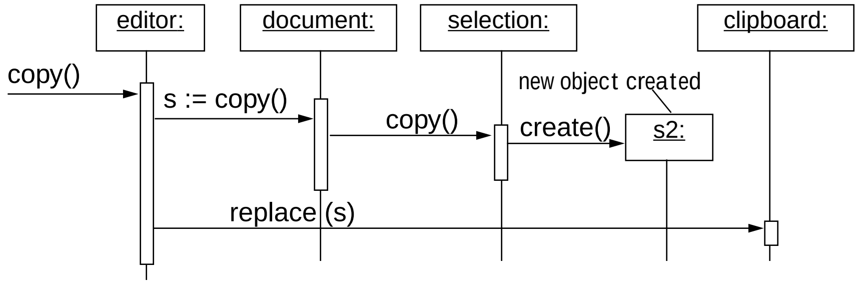
Object Interaction Graphs

Used for describing software designs



Message Sequence Diagrams

Equivalent to OIGs



Role-Activity Diagrams

Multi-participant action-occurrences

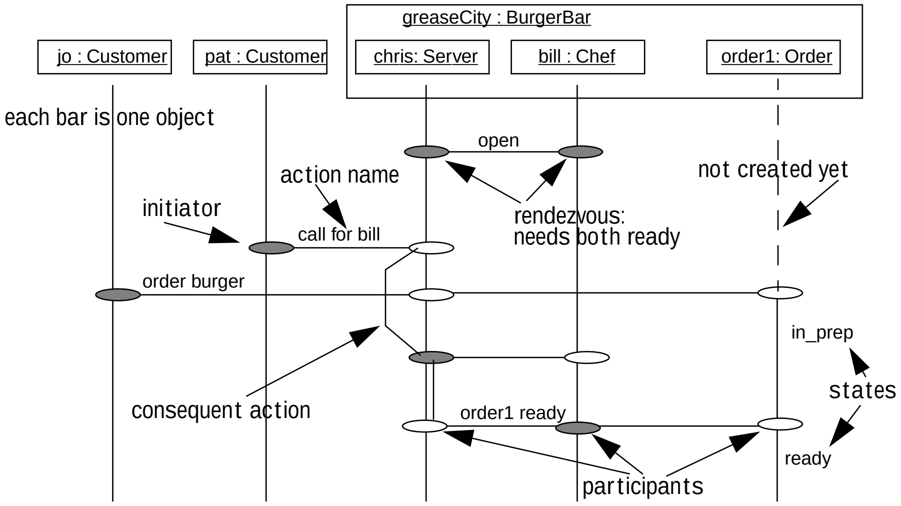


Figure 100: Collaborations