Chapter 3 Behavior Models
Object Types and Operations

In component-based development you must separate the external behavior of a component from its internal implementation. You describe behavior by specifying the component's type: a list of actions it can take part in and the way it responds to them. The kinds of actions we focus on here are the *operations* that the object may be requested to perform. The type specification in turn has two parts:

- The static model of an object's internal state and of information exchanged in the operation requests, using attributes, associations, and invariants
- Specifications of the effects of the actions on the component, using the vocabulary provided by the static model

This chapter describes how to derive and write a type specification.

We dealt with the static model in Chapter 2; we next want to specify actions in a way that abstracts from the model's many possible implementations. An action is specified by its effect on the state of the object and any information exchanged in the course of that action. This state is described as a type model of the object and of its in/out parameters; the effect is specified as a precondition/postcondition pair. Effects can either be written textually or depicted as transitions on a state chart.

At this stage, the objective is only to specify the actions and not to implement them (although we will look at some program code as examples). The latter part of this chapter also briefly discusses programming language classes and explains how they relate to the specifications.

The key to designing an implementation is to choose how the objects inside the component collaborate to provide the specified effects. Such collaborations are the subject of Chapter 4.

3.1 *Object Behavior: Objects and Actions*

In component-based development, you must construct software from components whose insides you can't see; you must treat them as black boxes. When you construct your own components, you m ust build them so that they will work with a wide variety of others even as their internal implementations change or are upgraded. Components that aren't interoperable have little value. For that reason, we are interested in separating external specification of behavior from the internal works. (This has been the situation in hardware for years; that it's novel to our profession should perhaps be an issue of some embarrassment for us.)

© ***object behavior*** The effects of an object on the outcomes of the actions it takes part in along with the effects of the actions on the object.

3.1.1 Snapshot Pairs Illustrate Actions

Object state changes as a result of actions. Given the object snapshot in Figure 3.1(a), if a client requested a session of the javaCourse, we end up with Figure 3.1(b). The new session is assigned to paulo, because he is qualified to teach that course. A scheduleCourse action occurs between the two snapshots. These before-and-after snapshots sometimes provide a useful way to envisage what each action does. Looking at the diagram, can you see what cancel(session-32), reschedule(session-5, 2000/1/5), or qualify(paulo, catalysis-Course) would do?

This is the primary reason for making a static model: we choose objects and attributes, whether written inside the types or drawn as links, that will help us define the effects of the actions. It would be difficult to describe the effect of schedule course without the model attributes depicted on the snapshots.

© ***action occurrence*** A related set of changes of states in a group of objects that occurs between two specific points in time. An action occurrence may abstract an entire series of interactions and smaller changes.

3.1.2 Pre- and Postconditions Specify Actions

The limitation of snapshots is that they show particular sample situations; we want to describe the effect an action has in all possible situations. We can do that by writing post-conditions—informal statements or formal expressions that define the effect of an action, using—the same navigation style as invariants in Section 2.5, Static Invariants. For example:

action schedule_course (reqCourse: Course, reqStart: Date)
pre: Provided there is an instructor qualified for this course
who is free on this date, for the length of the course.
post: A new confirmed session has been created, with course = reqCourse,
startDate = reqStart, and endDate – startDate = reqCourse.length.

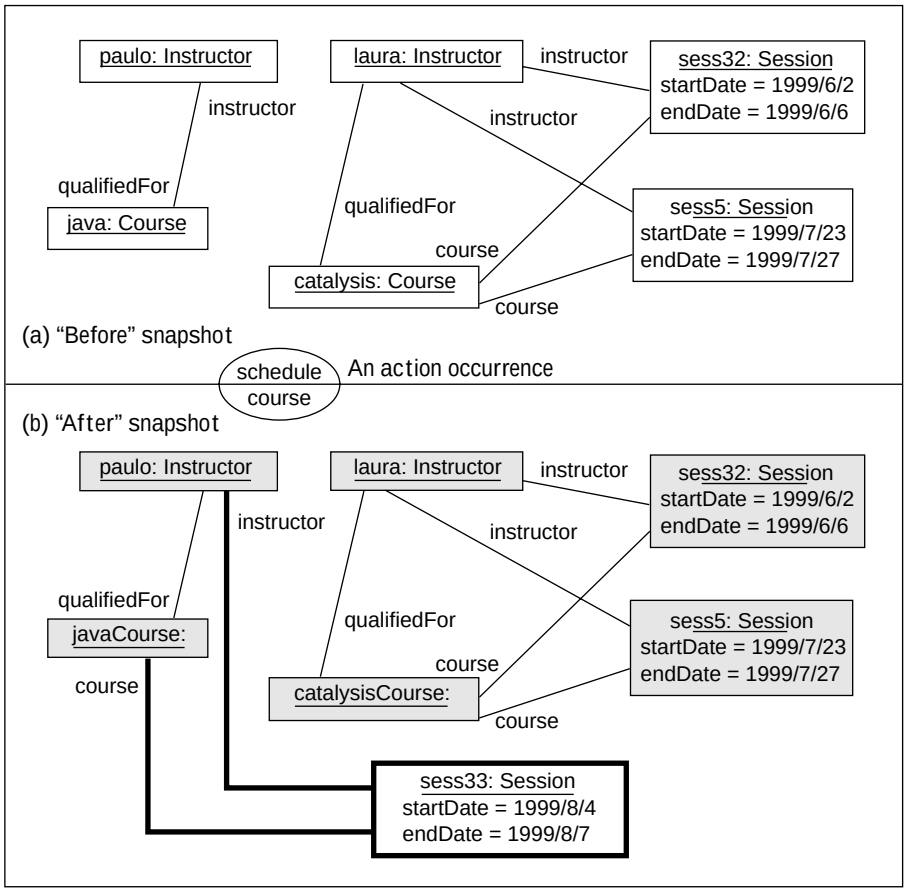


Figure 3.1 An action occurrence causes a change in state.

Notice that we have stated only some parts of what this action does. In fact, this is one of the nice things about specifying actions rather than designing them: you can stipulate only those characteristics that you need the outcome to have, and leave the rest unsaid, with no spurious constraints. This is exactly what's required for component-based development: we need to be able to say "A plug-in component must achieve this" but should not say how it achieves it or what additional things it might do, permitting many realizations. It is easy to combine requirements expressed in this way. Different needs can be anded together, something you can't do with chunks of program code. Moreover, different versions, expressed in different subtypes, can add their own constraints to the basic requirement (see Section 8.3.5, Joining Type Specifications Is Not Subtyping).

© ***action type*** The set of action occurrences that conform to a given action spec. A particular action occurrence may belong to many action types.

© ***action spec*** A specification of an action type. An action spec characterizes the effects of the occurrences on the states of the participating objects (for example, using a postcondition).

Actions can be joint (use cases): They abstract multiple interactions and specific protocols for information exchange, and describe the net effect on all participants and the summary of information exchanged.

Actions can also be localized, in which case they are also called operations. An operation is a one-sided specification of an action. It is focused entirely on a single object and how it responds to a request, without regard to the initiator of the request.

3.1.3 Types

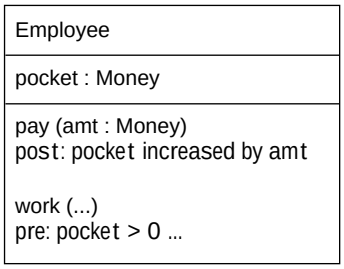
Different objects react in different ways to the same action. But rather than describe each object separately, we group objects into types: sets of objects that have some (but not necessarily all) behavior in common. A type is described by a type specification, which tells how some actions affect the internal state of the object and, conversely, how the state affects the outcome of actions.

Usually, types are partial descriptions. They say, "If you do X to one of my members, the resulting response will have this property and that property." But they don't always tell you everything there is to know about the outcome, and they don't tell you what will happen if you perform actions that aren't mentioned. This incompleteness is important, because it means that type specs can be easily combined or extended, essentially by anding them together. A type is quite different from a class in a programming language, which is a prescription telling the object *how* to do what it does.

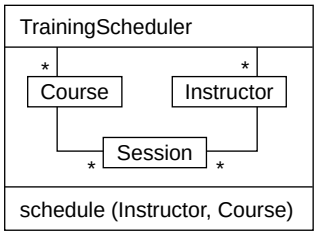
A client defines the type it expects of any other object it will use: the minimal set of actions it must exhibit. An implementor of an object defines the type(s) she provides: that set of actions she guarantees to meet now and through subsequent releases. This implementation can be used by the client if the provided type can be shown to conform to the expected one.[1]

Types often correspond to real-world descriptions. An Employee is something that does work when you give it money; a Parent is something that does work and gives you money; a Shopkeeper gives you things when you give it money. All these descriptions are partial, focusing on the behavior that interests certain other objects that interact with them. In object design, we build systems from interacting objects, so these partial perspectives are crucial. Each object could play several roles, so we need to easily talk about Employee * Parent: someone who does work when paid (by the appropriate other person) and also provides money (ditto).

1. Even if the implemented type was not declared a priori as implementing the expected one, specific programming languages may impose stronger restrictions. See Chapter 6, Abstraction, Refinement, and Testing.



Because these descriptions are "black box" ("I don't care how my parent gets the money as long as she or he provides it"), it would be nice if we could describe the actions entirely without reference to anything inside the object. That is possible for simple behaviors but not for complex ones. "Why will my employees not work when I ask them to?" "Because their pockets are empty." "How can their pockets be filled?" "You pay them." In this conversation, it is implicit that an Employee can have a pocket representing an amount of money. It doesn't really matter to the Employer how or where employees keep their money; it is just a model, a device to explain the relationship between the actions of payment and request to work.[2]



In a complex model, we find a few attributes, such as pocket, insufficient, and we tend to use pictorial attributes instead. But the idea is the same: the model is principally there to explain the effects of the actions. We can use the same principle to describe small, simple objects and large, complex systems. Of course, the large complex systems will need a few more tools for managing complexity and structuring a specification, but the underlying ideas will be the same.

© ***type*** A set of objects that conform to a given type spec throughout their lives.

© ***type spec*** A description of object behavior. It typically consists of a collection of action specs and a static model of attributes that help describe the effects of the actions. A type spec makes no statement about implementation.

3.1.4 Objects and Actions Model Business and Software

In Section 2.7, Models of Business; Models of Components, we remarked that a type model can deal with things in the real world, or it can model the internal state of a larger object such as a computer system or component. We showed this graphically by drawing the type of the component containing the types of the objects it "knew" about.

The techniques in this chapter can be used to specify either changes in the real world or changes inside a component (see Figure 3.2); but what both situations have in common is that we are specifying only the outcome, or *effects*, of the actions rather than what goes on inside. We close our eyes between the start and end of every change and describe only the comparison between the two snapshots of the business or system state.

Our terms *object* and *action* cover a broad range:

2. Legal contracts have the same structure (and are far more muddled): title, terms and definitions, actual contractual conditions using those terms

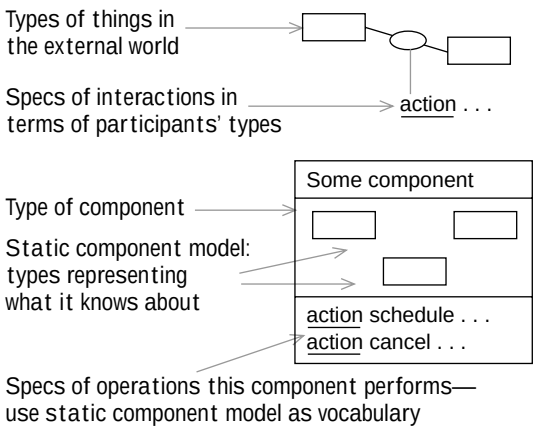


Figure 3.2 Real world through software.

- *Object* includes not only individual programming-language objects but also software components, programs, networks, relations, and records as well as hardware, people, and organizations—anything that presents a definable encapsulated behavior to the world around it or can be usefully thought of as such.
- *Action* includes not only individual programming-language messages or procedure calls but also complete dialogs between objects of all kinds. We can always talk about the effects of an action even without knowing exactly who initiates it or how it works in detail, as in this schedule_course example.

The diagram in Figure 3.1 can be seen in two ways. First, it can be a picture of the real world. The objects represent human instructors, scheduled sessions, and so on. The attributes represent who is really scheduled to do what, as written on the office wall planner and the instructors' diaries. An action is an event that has happened in the real world, and, invariably, it can be looked at in more detail whenever we wish. Scheduling a course involves several interactions between participants and resources.

Alternatively, the diagram may be about what a particular object knows about the world outside it (which may be different from some other object's view). In particular, it can be a model of the state of a software component.

The occurrence of the schedule_course action could represent a dialog between players in the real world. A representative from the client's company contacts the course scheduler in the seminar company, negotiates the dates and fees for a new session of the course, and updates the office wall planner.

Equally, the action could be an abstraction of a dialog with a software system. In that case, because of the Golden Rule of OO design (that we base the design on a domain model), we can use the same picture to denote objects (whether in a database or main memory) that the system uses to represent the real world. The interesting actions are then the interactions between the system and the rest of the world: They update the system's knowledge of what is going on in the world, as represented in the attributes.

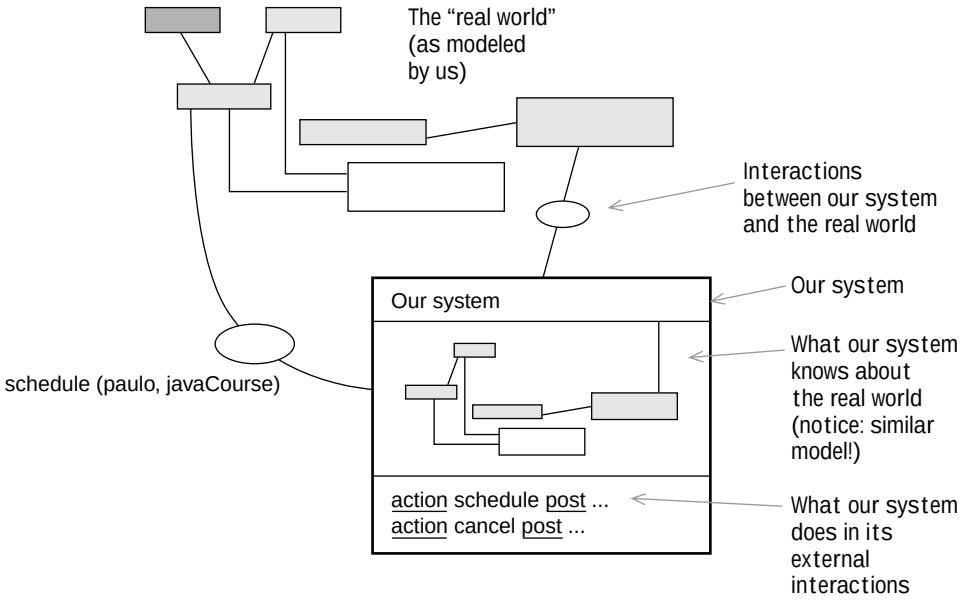


Figure 3.3 Models of domain and system.

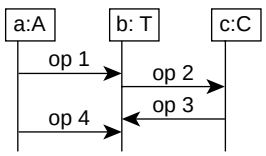
Now schedule (paulo, javaCourse) can refer to whatever dialog someone must have with our system to get it to arrange the session, and we can use snapshots of the system state to describe the effect the action has on the system (see Figure 3.3). In turn, the system’s state (as described by the snapshots) will have an effect on the outcome of future actions, including the outputs to the external objects (including people!) who interact with it.

3.1.5 Two Kinds of Action

There are two main kinds of actions we are concerned with in this book. They correspond to individual and collective behaviors of objects.

The first kind**,** a *localized* action (often called an *operation* in code), is an action which a single object is requested to perform; it is specified without consideration of the initiator of the action. You can recognize localized actions by their focus on a single distinguished object type:

action Type::actionName (...) ...



In program code, one object requests that another object perform an operation; the result is a state change, and some outputs. The interactions are illustrated with a sequence diagram: Objects are vertical lines, and each operation request is an arrow. This also applies outside of code; objects a,b, and c could be real-world objects such as client, company, and instructor; or

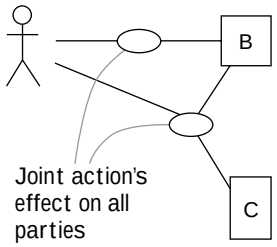
they could be instances of software classes, such as session, calendar, and event. This is the subject of the current chapter.

The second kind of action is a *joint* action To describe behavior and interactions of a group of objects, we focus on the net effect of interactions between multiple objects, and we specify that effect as a higher-level action with all objects involved. A joint action is written

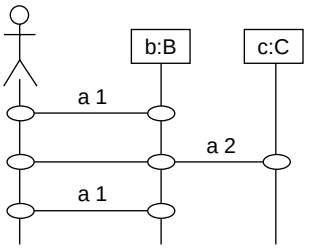
action (party1: Type1, party2: Type2, ...) :: actionName (...)

Notice that the joint action is not centered on a single distinguished object type. There are *directed* variations of joint actions in which a sender and a receiver are designated, but the action effect is still described in terms of all participants.

At the business level, it takes a sequence of interactions between client and seminar company, including enquire, schedule, deliver, follow-up, and pay, to together constitute an abstract purchaseCourse action. This sequence has a net effect on both client and seminar company: not only has the seminar company delivered a service and gained some revenue, but also the client has paid some fees and gained knowledge. In software, it may take a sequence of low-level operations via the user interfaces (UIs) of multiple applications and databases to complete a scheduleCourse operation. Such a joint action, also called a use case, is the subject of Chapter 4.



Each occurrence of such a joint action is shown as a horizontal bar with ellipses in a sequence diagram, whereas the finer-grained operations were depicted as simple arrows. Note that each action occurrence could be realized by many different finer-grained interactions, eventually reducing to a sequence of operations.



3.2 *More-Precise Action Specifications*

Well-written postconditions can be used as the basis for verification and testing. For this purpose, we should write the postconditions in a more precise style: as test (Boolean) functions. You can use the Boolean expression part of your favorite programming language; we will use a general syntax from UML called Object Constraint Language (OCL). It translates readily to most programming languages but is more convenient for specification.

The other benefit of writing the postconditions more formally is that doing so tends to make you think harder about the requirements. The effort is not wasted. You would have had to make these decisions anyway; you're just focusing on the most important ones and getting a better end result.

The rest of this section deals with key features of the more precise style. It is applicable to both business and component modeling. Later sections differentiate the two and discuss action specification in greater detail.

3.2.1 Using Snapshots to Guide Postconditions

A postcondition states what we want an end result to be. For example, let's suppose one instructor can be the mentor of one other instructor; perhaps some of them get too outrageous in class from time to time. The action of assigning a mentor is, informally, as follows:

action assign_mentor (subject: Instructor, watchdog: Instructor)
post: The watchdog is now the mentor of the subject.

This can be shown in a pair of snapshots (Figure 3.4).

There wasn't any mention of mentors in the model we drew earlier, so we needed to invent a way of describing them. Every instructor might or might not have a mentor, so this fragment of static model seems appropriate.

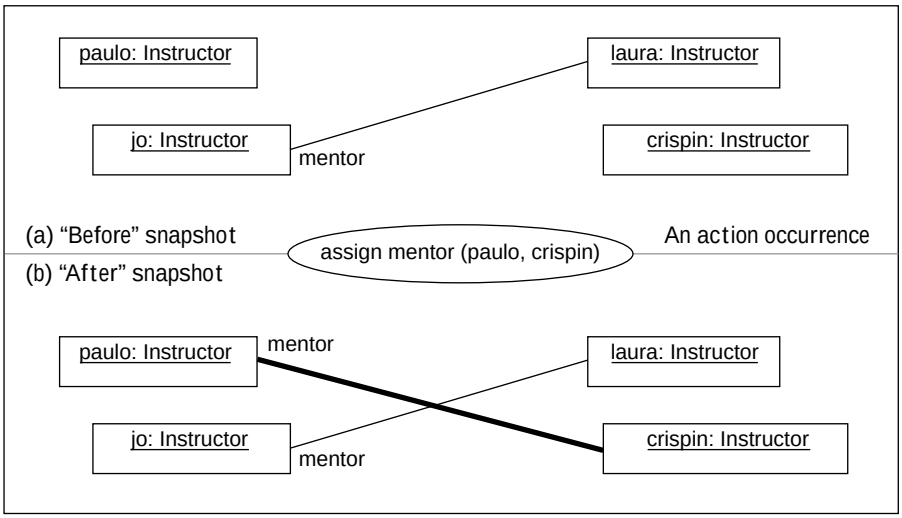


Figure 3.4 Assigning a mentor.

Now we can write the action in terms of this association:

action assign_mentor (subject: Instructor, watchdog : Instructor)
post -- the watchdog is now the mentor of the subject
subject.mentor = watchdog

Notice the following points.

- The postcondition states what we need; it doesn't say anything about aspects we don't care about (although we might want to be more explicit about what happens to any existing mentee of the watchdog). Looking at the snapshot, you can see how the example we illustrated corresponds to the change.
- Associations are by default bidirectional, so it isn't necessary also to write watchdog.mentee = subject. However, that would be an alternative to what we wrote.
- Navigation expressions in an action spec should generally start from the parameters. (So mentor=watchdog would be wrong—whose mentor?) Other starting points are self (in actions performed by a particular object) and variables you have declared locally, in, for example, forAll and let clauses (see Section 2.5.2).

Informal → snapshot → formal. This basic procedure is the general way to formalize a postcondition. However, you must be careful of alternative cases: a snapshot illustrates only one case, and so you may need to draw several to get a feel for the gamut of possibilities. It's the action postconditions you're really trying to determine; the snapshots are mainly thinking tools.

3.2.2 Comparing Before and After

A postcondition makes an assertion about the states immediately before and after the action has happened. For every object there are therefore two snapshots and two complete sets of attribute values to refer to. By default, every mention of an attribute in a postcondition refers to the newer version; but you can refer to its prior value by suffixing it with @pre.

- subject.mentor@pre refers to subject's old mentor.
- subject.mentor.mentee@pre refers to subject's new mentor's old mentee.
- subject.mentor@pre.mentee@pre refers to subject's old mentor's old mentee.
- subject.(mentor.mentee)@pre refers to same as previous.
- subject.mentor@pre.mentee refers to subject's previous mentor's *new* mentee.

Each navigation expression is a way of getting from one object to another. By default, the navigation is within a single snapshot; @pre can be applied to an expression to evaluate it in the preceding snapshot. But what you get from an expression is the (constant) identity of an object; and unless you keep applying @pre, further expressions will always evaluate in the newer time. Figure 3.5 shows a before-and-after snapshot and the object referred to by subject.mentor.mentee@pre. The before-and-after times are immediately before and after an action occurred. Here is an example:

```
action   assign_mentor (subject: Instructor, watchdog : Instructor)
post     subject.mentor = watchdog and     -- watchdog is now subject's mentor,
         -- and if watchdog had a previous mentee, they now have none
         let ex_mentee = watchdog.mentee@pre in
              ex_mentee <> null ==> ex_mentee.mentor = null
```

Here are some points to note.

- null is used to represent "no object" when an association permits it.

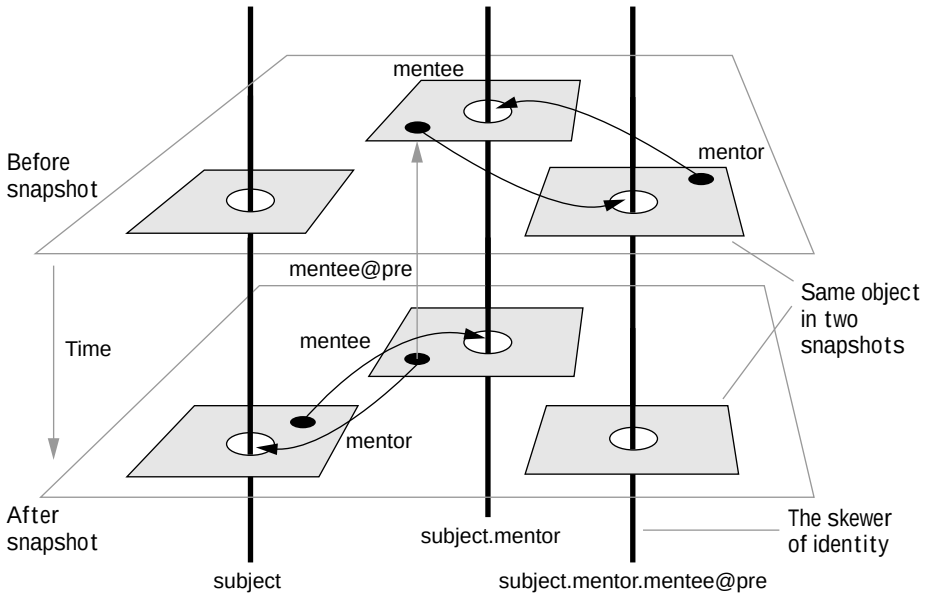


Figure 3.5 The kebab model of object history.

- ==>, also written implies, is used to mean: if ... then ... .
- @pre takes you back to the previous value of a changeable attribute; parameters refer to the same object, so there is no point in writing subject@pre.
- An action spec deals with just two states, so x@pre@pre is undefined.

Precise abstraction raises pertinent questions. The level of detail here is enough to draw out debate. When this example is discussed in groups, this is often a point when discussion arises about what should happen to the ex-mentee (dreadful expression! I hope never to be one). For example, should the static model be revised to allow more than one mentee per mentor?

Whatever the answer, this is a business question; but it might not have arisen at this early stage if we hadn't tried being more precise. And yet we have done so without waiting until we are wading in reams of program code.

3.2.3 Newly Created Objects

As a consequence of an action, new objects can be created. The set of these objects has the special name new in a postcondition,[3] and there are some special idioms for using it. After drawing the snapshot in Figure 3.1, we can write

```
action  schedule_course (reqCourse: Course, reqStart: Date)
```

```
post:   let ns: Session.new [course = reqCourse and startDate = reqStart
                                 and endDate – startDate = reqCourse.length ]
        in   ns.instructor.available@pre(startDate, endDate)
    -- there is a new Session—call it ns—with the attributes specified; and
    -- its instructor was initially available for the requested period
```

Notice the following points.

- The power of the postcondition is that it lets you avoid unnecessary detail. We have not said which instructor should be assigned nor how one should be chosen from the available ones. We have limited our statement to only the requirements we need: that the person chosen should have no prior commitment.
- We have deferred some complexity by assuming a parameterized attribute available defined with instructors. We can define its value later.

3.2.4 Variables an Action Spec Can Use

An action spec tells about the outcome of a named action happening to a set of parameter objects and (for localized actions) a receiver. The spec of an action can use the following:

- self, referring to the receiver object whose type is written
- The parameter names referring to those objects
- A result object
- The attributes of the type of self written or drawn as associations
- The attributes of the parameters and the result
- Local names bound in let, forAll, exists, and so on.

3. Postconditions have no sequencing, solving an age-old problem:
action which_came_first?() post: Chicken.new <> 0 & Egg.new <> 0

3.2.5 Collections

In the superposed pair of snapshots in Figure 3.6, the new state is shown in bold. Many of

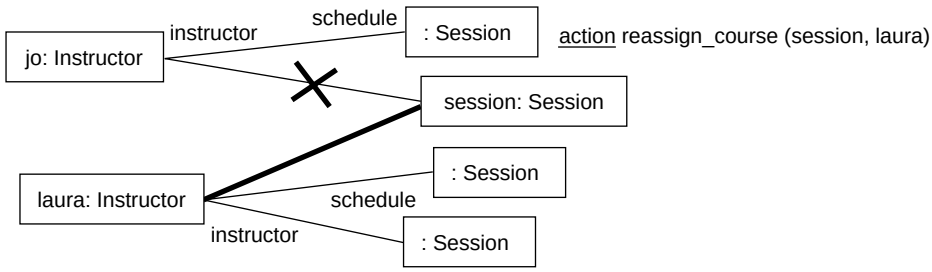


Figure 3.6 Snapshot for action reassign_course.

the associations in a model are of multiple cardinality and by default represent sets (without nested sets—we call these *flat* sets). We can use the collection operators (see Section 2.5.2):

```
action   reassign_course (session : Session, new_inst: Instructor)
post:    -- An existing Session is taken off one instructor's schedule and placed
            onto this new one
         let ex_instructor = session.instructor@pre
         in
              ex_instructor.schedule = ex_instructor.schedule@pre – session
   and        new_inst.schedule = new_inst.schedule + session
```

Notice the use of + and – with collections—only the set union and difference. (The construct collectionAttribute= collectionAttribute@pre + x is so common that some of us have taken to writing collectionAttribute += x. But if you do this, please remember that this is not an assignment but is merely a comparison between two states. Also, a few extra keystrokes are usually better than the overloading of +, +=, and so on.)

We could perhaps more simply have asserted

```
session.instructor = new_inst
```

Because the static model tells us a session has only one instructor, this might have been adequate. However, a designer might mistake the meaning of this and make this session the only one the new instructor is assigned to, deleting all the instructor's other commitments. So we choose to be more explicit.[4]

4. There is a deeper issue concerning "framing." In a fully formal spec such as for safety-critical systems, you would be more explicit about which objects are left untouched.

3.2.6 Preconditions

Many of the action postconditions we define make sense only under certain starting conditions, which can be characterized by a precondition. The precondition deals only with one state, so it doesn't have @pre or new. For example:

```
action   assign_mentor (subject : Instructor, watchdog : Instructor)
pre      -- happens only if the subject doesn't already have one
         subject.mentor = null
post     ...as before ...
```

The precondition is also a purely Boolean expression and has no side effects; it can refer only to inputs and initial values of attributes. The corresponding postcondition is not guaranteed by the implementor if the precondition did not hold.

Precise preconditions are essential for system safety properties—the things to guard against to avoid undefined behaviors. The postconditions are primarily used to document the state changes and outputs guaranteed by the implementation.

3.2.7 More-Precise Postconditions: Summary

This section has looked at the basics of writing action specifications precisely enough to form the basis for testing a component and to make the model explicit enough to uncover business issues.

The techniques we have seen can be used to describe the interactions that occur within a business; or they can describe the actions performed by a software system or component; or—the simplest case—they can describe the operations performed by an individual object within a software design. That is what we will look at next.

(The syntax of action specs and postconditions is shown later in Exhibits 3.1 and 3.2. Specifying requirements for a complete software system, with a user interface and so on is the topic of Chapter 15, How to Specify a Component. Specifying the interface to a substantial component is covered in Chapter 10, Components and Connectors.)

3.3 *Two Java Implementations of a Calendar*

This chapter is about specifying types: what a component does as seen from the outside and ignoring what goes on inside. But "brains work bottom up," so it will be easier to understand what the specification means if we can see the kinds of implementation that it can have. Let's start the time-honored way: we'll hack the code first and write up the spec afterward.[5]

5. If this offends your sense of decency, please skip to the next section. You may wish to avert your eyes from the naked code on display.

Our seminar scheduling application will have many classes in its implementation. One likely class is a calendar that tracks various scheduled events for various instructors. We start with two different Java implementations of the calendar. Then we show how the external behaviors can be specified independent of implementation choices and even of implementation language and technology. We will ignore any UI aspects.

Both implementations support just four external operations on a calendar; they may introduce other internal operations and objects as needed.

- addEvent adds a new event to the current calendar schedule.
- isFree determines whether an instructor is free on given dates.
- removeEvent deletes an existing event from the schedule.
- calendarFor returns the scheduled events for a particular instructor. It is returned as an Enumeration—that is, a small object that has operations to step through the collection until the end.

3.3.1 Calendar A Implementation

The implementation of calendar A keeps a separate unordered vector of events for each instructor in a hashtable, keyed by the instructor. The calendar's internal interactions are described in the sequence diagram in Figure 3.7, with each arrow indicating an operation request. Upon receiving an addEvent request, the calendar first creates a new event object. It then looks up the event vector for the current instructor in its hashtable, creating a new vector if none exists. The new event is added to this vector, and the hashtable is updated. The Java code for this design is shown on the next two pages.

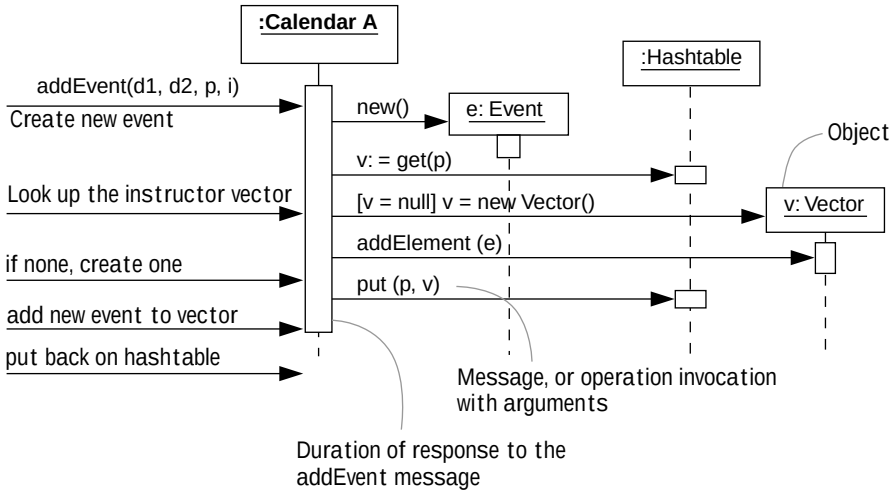


Figure 3.7 Internal design interactions of calendar A.

```
import java.util.*;

// This calendar organizes events by instructor in a hashtable keyed by instructor
class Calendar_A {
    private Hashtable instructorSchedule = new Hashtable();

    // provided no schedule conflict, this creates and records new event
    public Event addEvent (Date d1,Date d2, Instructor p, Object info){
        if (! isFree (p, d1, d2)) return null;

        Event e= new Event (d1, d2, p,info);
        Vector v = (Vector) instructorSchedule.get (p);
        if (v == null) v = new Vector ();
        v.addElement (e);
        instructorSchedule.put (p,v);
        return e;
    }

    // Answer if the instructor free between these dates
    // do any of the instructor's events overlap d1-d2?
    public boolean isFree (Instructor p, Date d1, Date d2) {
        Vector events =(Vector) instructorSchedule.get (p);
        for   (Enumeration e = events.elements(); e.hasMoreElements (); ) {
            Event ev = (Event) e.nextElement ();
            if (ev.overlaps (d1,d2)) return false;
        }
        return true;
    }

    // remove this event from the calendar
    public void removeEvent (Event e) {
        Vector v = (Vector) instructorSchedule.get (e.who);
        v.removeElement (e);
    }

    // return the events for the instructor (as an enumeration)
    public Enumeration calendarFor (Instructor i) {
        return ((Vector) instructorSchedule.get(i)).elements();
    }
}

// internal details irrelevant here
class Instructor { }

// represents one session
// Just two public operations: delete() and overlaps()
class Event {
    Date from;
    Date to;
```

```
    Instructor who;
    Object info; //additional info, e.g. Session
    Calendar_A container; // for correct deletion
    Event (Date d1,Date d2, Instructor w, Object i) {
        from = d1;
        to = d2;
        who = w;
        info = i;
    }

    // does this event overlap the given dates?
    boolean overlaps (Date d1, Date d2){
        return false;
    }

    public void delete() { /* details not shown*/ }
}
```

3.3.2 Calendar B Implementation

This version uses a more complex representation, not detailed here, to maintain the events so that they are indexed directly by their date ranges. This data structure is encapsulated behind an interface called EventContainer that does all the real work.

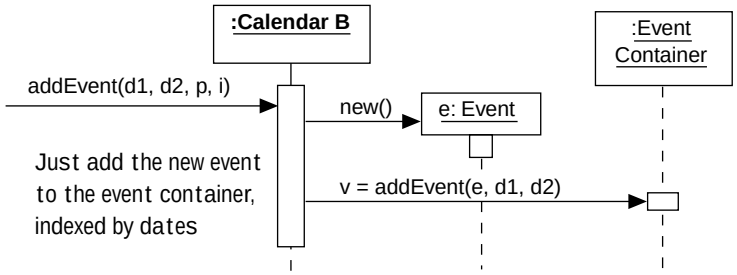


Figure 3.8 Internal design interactions of calendar B.

The internal interactions for this calendar implementation are shown in Figure 3.8, and the Java code starts below and continues on the next two pages.

```
import java.util.*;

// Organizes events by their dates using a fancy event container
class Calendar_B {
    privateEventContainer schedule;

    // create the event and add to schedule
    public Event addEvent (Date d1, Date d2, Instructor p, Object info) {
        Event e= new Event (p,info, schedule);
```

```
            schedule.addEvent (e, d1, d2);
            return e;
        }

        // is instructor free between those dates?
        // are any of the events between d1-d2 for this instructor
        public boolean isFree (Instructor p, Date d1, Date d2) {
            for   (Enumeration e = schedule.eventsBetween (d1, d2);
                        e.hasMoreElements (); )
                  if (((Event) e.nextElement()).who ==p) return false;
            return true;
        }

        // remove the event from the schedule
        public voidremoveEvent (Event e) {
            schedule.removeEvent(e);
        }

        // return the events for the instructor (as an enumeration)
        public Enumeration calendarFor (Instructor i) {
            // implementation not shown
            // presumably less efficient, since tuned for date-based lookup
            // e.g. get all eventsBetween (-INF, +INF)
            // select only those for instructor i
            return null;
        }

}

// internal details of instructor irrelevant here
class Instructor { }

// Just one public operation: delete() shown
// dates not explicitly recorded; container maintains date index
class Event{
        Instructor who;
        Object info;
        EventContainer container; // for correct deletion
        Event (Instructor w, Object i, EventContainer c) {
            who = w;
            info = i;
        }

        public void delete() { /* details not shown*/ }
}

// event container: a fancy range-indexed structure
```

```
interface EventContainer {
    // return the events that overlap with the d1-d2 range
    Enumeration eventsBetween (Date d1, Date d2);
    // add, remove an event
    void addEvent (Event e, Date d1, Date d2);
    void removeEvent (Event e);
}
```

3.4 *Type Specification of Calendar*

A client could use either implementation of the calendar; both of them implement the same type. We must describe this type so that a client can use either implementation based solely on the type specification (this example is small enough to illustrate the details).

Figure 3.7 and Figure 3.8 show that the internal representation and interactions differ widely between the implementations. Our behavior specification must abstract these irrelevant "internal" interactions and include only interactions with objects that *the client should be aware of*.[6] What we really want to specify is the calendar together with some abstraction of its (hidden) event container, hashtable, vector, and so on (see Figure 3.9).

Our type specification must be precise enough that the client understands what assumptions the implementations can make and what guarantees they provide in return. For example, are the events returned by calendarFor ordered by increasing dates? When you delete() an event, is a separate call to removeEvent on the calendar required? If it is, which one should be done first? What happens if the dates d1,d2 are not in the right order on a call to isFree?

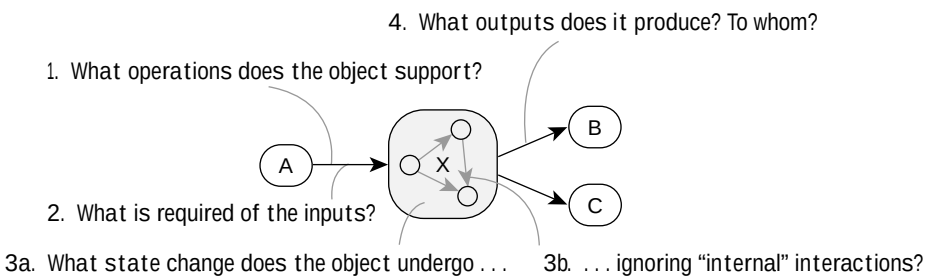


Figure 3.9 External object behavior abstractions internal details.

6. Try understanding a bureaucratic government office in terms of its internal interactions.

3.4.1 From Attributes to Operation Specification

Following is a sequence of steps to arrive at a precise type specification of the calendar.[7] We assume that the calendar can return output values to the client and that all other interactions are internal details that should not be known to the client. We omit discussion of error conditions and exceptions for now; they are covered in more detail in Section 3.6.3, Multiple Action Specs: Two Styles, and Section 8.4, Action Exceptions and Composing Specs.

1. List the operations: addEvent, isFree, removeEvent, and calendarFor.

2. Write informal operation descriptions of each one.

- addEvent creates a new event with the properties provided and adds it to the calendar schedule.
- isFree returns true if the instructor is free in the date range provided.
- removeEvent removes the event from the calendar.
- calendarFor returns the set of events scheduled for the instructor.

At this stage, it's usual to start sketching a static type diagram (see Figure 3.10), even though completing it is the focus of a later step. Draw a diagram that includes the nouns mentioned in the action specs and their associations and attributes.

3. Identify the inputs and outputs. At the level of individual operations in code, these are usually straightforward, perhaps already known.

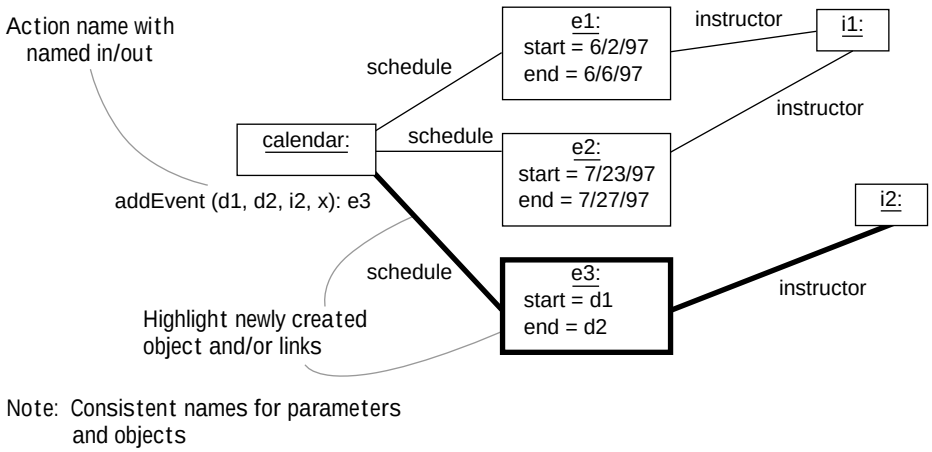


Figure 3.10 A snapshot pair for an action occurrence.

addEvent (date1, date2, instructor, info): Event

7. Thanks to Larry Wall for pulling apart the steps involved.

isFree	(instructor): Boolean
removeEvent	(event)
calendarFor	(instructor): Enumeration

4. Working from your initial type diagram, sketch a pair of snapshots before and after each operation. Draw them on one diagram, using highlights to show newly created objects and links and X and for objects or links that do not exist in the "after" snapshot. Name the input and output parameters to the action occurrence consistently with the snapshots.

After an addEvent, the highlighted objects and links are created; the output is e3. On the same snapshot, after a calendarFor (i1), the snapshot is not changed, and the output enumeration will list {e1, e2}. For read-only functions such as isFree, check whether there is some way the information could be extracted from every snapshot.

5. Draw a static type diagram of the object being specified, generalizing all snapshots (see Figure 3.11).[8] Here are the attributes mentioned by each operation.

- addEvent: Calendar schedule represents events currently in the calendar. Each event has attributes instructor and start and end dates. The overlaps attribute will be convenient.
- isFree: Instructor has an attribute free on a given date, constrained by the events scheduled for that instructor as described by the instructor's schedule.
- removeEvent: No new attributes are needed; schedule on calendar suffices.

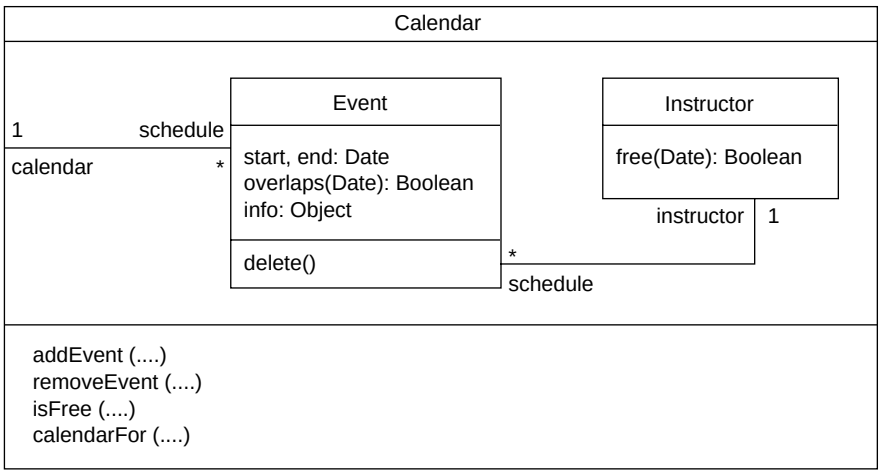


Figure 3.11 Type model attributes are used to specify operations.

8. Not all tools can draw one type inside another. An alternative is given in Section 3.11.

– calendarFor: Use a schedule attribute on instructor; note that the externally provided operation is not the same as the attribute that models the required state information.

6. Document the invariants that the model should satisfy.

```
-- the start of any event must be before (or at) its end
inv Event:: start <= end

-- instructor free on any date means "no event on his schedule overlaps that date"
inv Instructor:: free(d: Date) = ( self.schedule [overlaps(d)] ->isEmpty )

-- event overlaps (d) means same as "d between start and end, inclusive"
inv Event:: overlaps(d: Date) = (start <= d & end >= d)
```

7. Specify operations. Make the operation specs more precise.

```
-- the addEvent operation on a calendar
action Calendar::addEvent (d1: Date, d2: Date, i: Instructor, o: Object): Event
        pre:    -- provided dates are ordered, and instructor is free for the range
                   of dates
                d1 < d2 & {d1..d2}->forAll (d | i.free (d))
        post:   -- a new event is on the calendar schedule for those dates and
                   that instructor
                result: Event.new [ info = o &
                                    start = d1 & end = d2 & instructor = i & calendar = self]
```

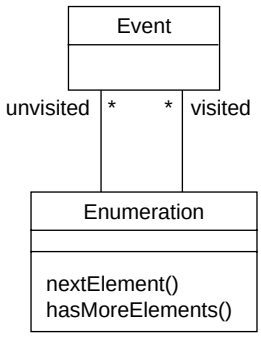
A function is an operation that may return a result and causes no other state change.

```
-- is a given instructor free for a certain range of dates?
function Calendar::isFree (i: Instructor, d1: Date, d2: Date) : Boolean
        pre:    -- provided the dates are ordered
                d1 < d2
        post:   -- the result is true if that instructor is free for all dates between
                   d1 and d2
                result = {d1..d2}->forAll (d | i.free (d))

-- remove the given event
action Calendar::removeEvent (e: Event)
        pre:    -- provided the event is on this calendar
                schedule->includes (e)
        post:   -- that event has been removed from the calendar and instructor
                   schedules
                not schedule->includes (e) and
                not e.instructor.schedule@pre->includes (e)

-- return the calendar for the instructor; also a function or side-effect-free
   operation
function Calendar::calendarFor (i: Instructor): Enumeration
        pre:    -- none; returns an empty enumeration if no scheduled events
                   true
        post:   -- returns a new enumeration on the events on that instructor's
                   schedule
                result: Enumeration.new [unvisited = i.schedule]
```

8. Create parameter models. Describe (by a type model) any input and output parameter types and their attributes and operations to the extent that the client and the implementor need to understand and agree on them.



The Enumeration returned by calendarFor could also be modeled explicitly. It provides two operations, informally specified next. These operations could be made more precise by using the two attributes on the enumeration.

action Enumeration::nextElement() : Event
pre: -- provided the enumeration is not empty
post: -- returns (and visits) an unvisited event, in no particular order

function Enumeration::hasMoreElements() : Boolean
post: -- true if all events have been visited

Event has a delete() operation that is visible to the client. Clearly, the client needs to know the effects of this operation—for example, does delete also remove it from the calendar? They can be specified directly using the same type model:

```
-- deletion of an event
action Event::delete ()
    pre:    true
    post:   -- the event is no longer on the calendar's or instructor's schedule
            not (calendar.schedule)@pre->includes (e) and
            not (instructor.schedule)@pre->includes (e)
```

Or, more concisely (as discussed in Section 3.8.5):

```
-- deletion of an event
action Event::delete ()
    pre:    true
    post:   -- the same effect as removing the event from the calendar
            -- (though not necessarily by calling the removeEvent method)
            [[ calendar.removeEvent (self) ]]
```

Note that an adequate specification of Calendar requires a specification of other object types that are client-accessible, such as Event and Enumeration.

9. Write a dictionary of terms and improve your informal specifications.

instructor	the person assigned to a scheduled event
schedule	the set of events instructor is currently scheduled for
free	if an instructor is free on a date, it means that no event on his schedule overlaps with that date
Calendar	the collection of scheduled events
schedule	the set of events currently "on" the calendar
Event	a scheduled commitment (meeting, session, etc.)
when	the range of dates for this event
instructor	the instructor assigned to this event
overlaps	if an event overlaps a date, it means that date lies within the range (inclusive) of dates of the event

Even if invariants and operation specifications will not be formalized, you can concisely define the terminology of types and attributes and consequently of operation requirements. Contrast the following updated informal operation specifications with the ones we started with. (Which is worse: reams of ambiguous narrative, or tomes of formal or pseudoformal syntax with no explanatory prose?)

```
-- add an event to a calendar
action Calendar::addEvent (d1: Date, d2: Date, i: Instructor, o: Object)
        pre:    -- provided dates are ordered, and instructor is free for range of dates
        post:   -- a new event is on calendar for those dates and that instructor

-- is a given instructor free for a certain range of dates?
function Calendar::isFree (i: Instructor, d1: Date, d2: Date) : Boolean
        pre:    -- provided the dates are ordered
        post:   -- return is true if instructor is free for all dates between d1 and d2

-- return the calendar for the instructor
function Calendar::calendarFor (i: Instructor): Enumeration
        pre:    -- no assumptions; could return an empty set enumeration if no
                   scheduled events
        post:   -- returns an enumeration on the events on that instructor's
                   schedule

-- deletion of an event
action Event::delete ()
        pre:    -- no assumptions
        post:   -- the same effect as removing the event from the calendar
                -- (though not necessarily by calling the removeEvent method)
```

10. Improve the model or design by some refactoring. For example, we can remove the repeated constraint d1<d2 by introducing a DateRange type, with attributes start,end dates and overlaps(date), and an invariant on these attributes.

3.4.2 The Resulting Object Type Specification

Calendar requirements have been specified in such a way that they can be fulfilled by either implementation—or indeed by any other that behaves suitably. The actions have been listed, and we have described the effect of each action on our model of the calendar state. Figure 3.12 shows the specification task's main products.

3.5 *Actions with Invariants*

Actions cause changes in attributes. Invariants are rules about relations that must hold between attributes. How are actions and invariants related?

3.5.1 Actions Need Not Duplicate Invariants

Operation specifications can be simplified by taking advantage of constraints in the type model. Consider the type model of Scheduler in Figure 3.13 with these invariants:

```
inv Instructor::    -- only assigned to sessions I am qualified to teach
```

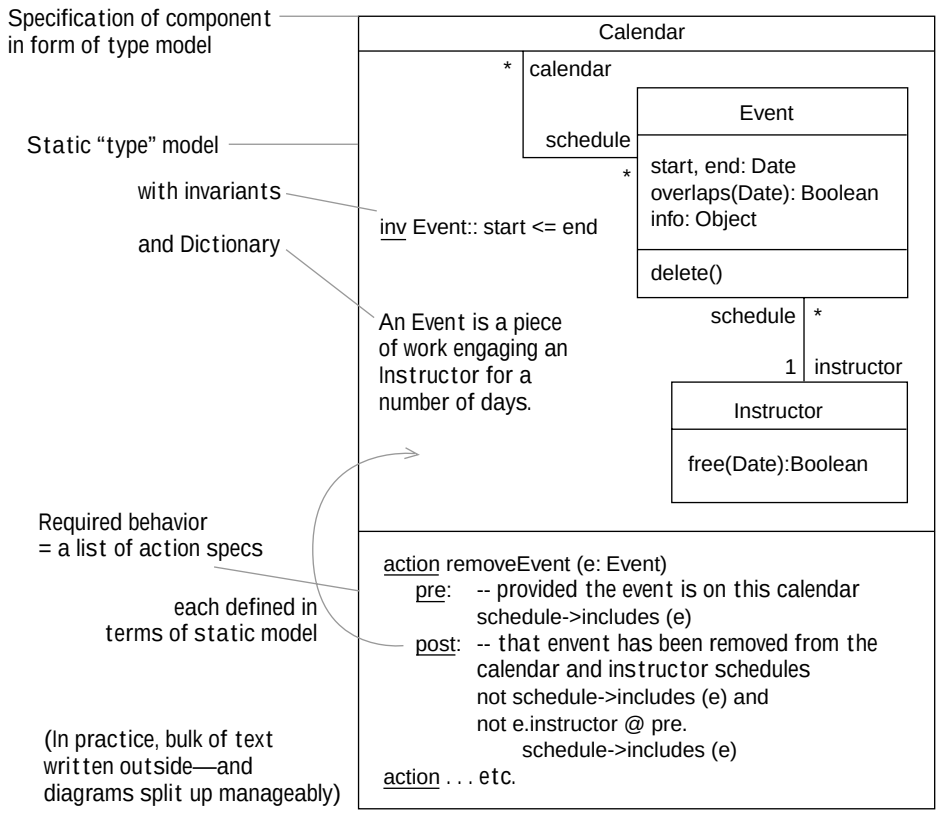


Figure 3.12 The product of behavior modeling is an object type spec.

```
    qualifiedFor -> includesAll (sessions.course)
        -- never double-booked; no 2 assigned sessions that overlap
    sessions ->forAll (s1, s2 | s <> s1 implies not s1.overlaps(s2))
inv Session::       -- only confirmed with assigned instructor
    confirmed ==> instructor <> null and
    -- session dates cover course duration
    end = start + course.duration -- assume suitable "Date+Duration: Date"
```

Let us try to define an operation against this model.

```
-- change the dates of a session
action Scheduler::change_dates (s: Session, d: Date)
    pre:    s.start > now and                -- (1) not from the past
            s.course <> nil and              -- (2) has a valid course
            s.course.duration : Days         -- (3) course has a valid duration
    post:   s.start = d and                  -- (4) start date updated
            s.end = d + course.duration      -- (5) end date updated
```

Which parts of the specification are necessary, and which are unnecessary?

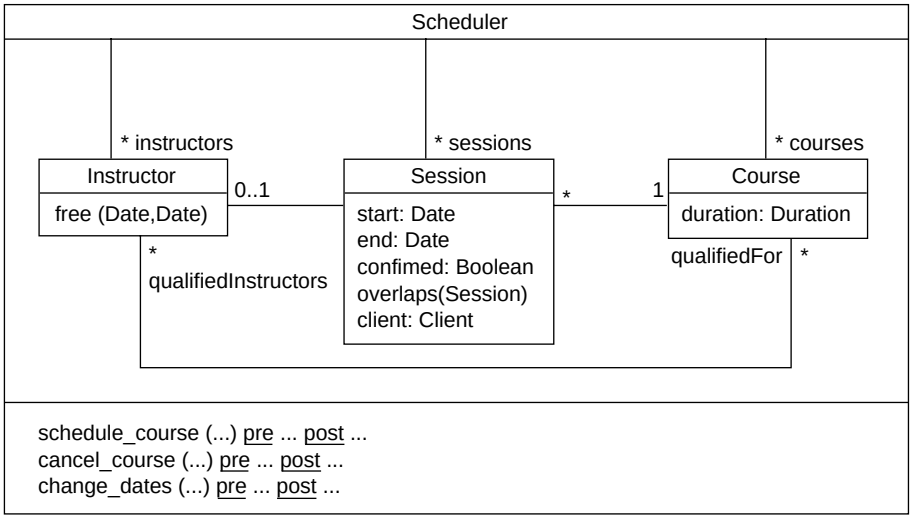


Figure 3.13 Scheduler type model with invariants.

1. *Necessary*: One cannot change the dates of a session from the past.

2. *Unnecessary:* It would not make sense to change the dates of a session that did not have a course. However, the type model already uses a multiplicity of 1 to state that any session object must have a corresponding course. The operation parameter already requires s to be of type session, so we do not need to repeat (2); using the type name Session implies all required properties of session objects.

3. *Unnecessary:* The postcondition refers to start + course.duration, which makes sense only if duration was a valid Duration. Once again, the type model already stipulates that every course has a duration attribute that is a valid Duration.

4. *Necessary:* This is the essential part of the postcondition.

5. *Unnecessary:* It seems reasonable that the end date of the course is also changed. However, the relationship between the start and end dates and the course duration is not unique to this operation, so it has been captured in the type model as an invariant. It is sufficient to state that the start date has changed; the invariant implies that the end date is also changed.

The unnecessary parts would not be incorrect, only redundant. Removing them leaves a much simpler operation specification:

```
action Scheduler::change_dates (s: Session, d: Date)
    pre:    s.start > now          -- not from the past
    post:   s.start = d            -- start date updated
```

A more interesting example is schedule_course.

```
-- this spec deals with scheduling a confirmed course
action Scheduler::schedule_course (who: Client, c: Course, d: Date)
```

```
pre:    -- Provided there is an instructor qualified and free for these dates
        c.qualifiedInstructors ->includes (i | i.free (d, d+c.duration))
post:   -- A new confirmed Session has been created for that course, client,
           dates
        s:Session.new [ confirmed & client = who & course = c & date = d ]
        -- assigned one of the course qualified instructors who was free
        instructor : c.qualifiedInstructors [free(d, d+c.duration)@pre]
```

It is already an invariant that any confirmed course must have a qualified instructor and that instructors cannot be double-booked. Hence, the italicized parts of the postcondition are redundant, and the last line in this specification can be omitted. Implementations may choose among the available qualified instructors in different ways.

3.5.2 Redundant Specifications Can Be Useful

We have seen how certain elements of an operation specification are implied by the invariants. Writing them would not be incorrect, only redundant. It can still be useful to write them down; note the change_dates example earlier. However, it is worth distinguishing those parts of the specification the designer should explicitly pay attention to—the invariants and necessary parts of operation specs—from those parts that would automatically be satisfied as a result.

Just as we can introduce *derived attributes*—those marked with a / that could be omitted because they are defined entirely in terms of other attributes—we can also introduce *derived specifications:* properties we claim would automatically be true of any correct implementation of the nonderived specifications. Using /pre: /post:, we can more explicitly define the change_dates operation:

```
action Scheduler::change_dates (s: Session, d: Date)
    pre:        s.start > now                   -- necessary
      /pre:     s.course <> nil and             -- derived: multiplicity 1
                s.course.duration : Days        -- derived: attribute definition
    post:       s.start = d                     -- necessary
      /post:    s.end = d + course.duration     -- derived: session invariant
```

The same holds for derived invariants. Of course, we would write only those claims we consider important to explicitly point out.

```
inv Session::     end = start + course.duration
/inv Session::    start = end – course.duration -- derived: definition of +, –
```

© ***redundant specs*** A specification (including invariants and pre- and postconditions) that is implied by other parts of the model but is included for emphasis or clarity. Such specs are prefixed with a /.

3.5.3 Static Invariants

A static invariant is implicitly anded to the precondition and the postcondition of every action within a defined range of actions. In the simplest case, the range of an invariant

means all operations on members of the type it is defined for (see Figure 3.14). Consider this code:

```
action Scheduler::change_dates (s: Session, d: Date)
    pre:    s.start > now          -- not from the past
    post:   s.start = d            -- start date updated
```

It combines with the invariant that every confirmed session has an assigned instructor; and an instructor is assigned only to a course she is qualified for; and she is never double-booked, to effectively yield

```
action Scheduler::change_dates (s: Session, d: Date)
    pre:    s.start > now          -- not from the past
        &   provided other invariants hold at start of action
    post:   s.start = d            -- start date updated
        &   s.confirmed ==>        -- if still confirmed
                (s.instructor <> null -- will have an assigned instructor
                 & s.instructor : s.title.qualifiedInstructors -- who is qualified
                 & s.instructor is not double-booked )
```

However, the private operations of any implementation may see situations in which the invariant is "untrue." For example, suppose the user assigns an instructor to a session whose dates overlap an existing assigned session. One acceptable implementation would be to deassign the existing session, but in the actual code, this might happen after assigning the new one. Thus, the invariant is temporarily broken between *internal* actions in the code, making it even more important to have the invariants properly scoped to a given set of externally available operations.

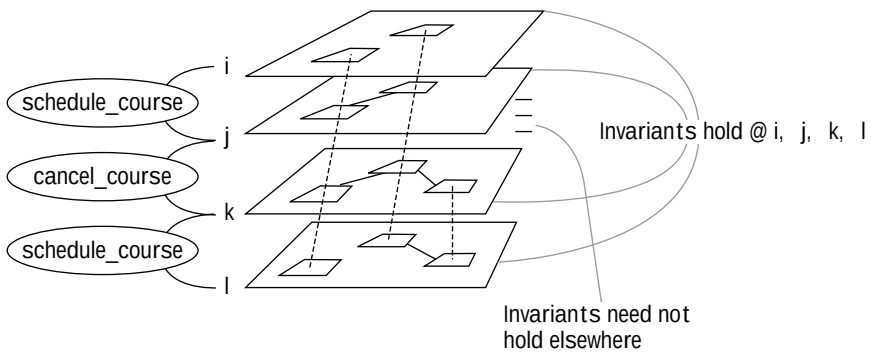


Figure 3.14 Invariants hold before and after a range of actions.

3.5.4 Effect Invariants

A static invariant is expected to hold before and after the actions in its range. Sometimes, an effect is required to be true of all the actions in its range. For example, suppose we want to count every operation invocation on our calendar. An *effect invariant* defines an effect that is invariant across all actions in its range and is implicitly anded to the postconditions

of those actions. Unlike a static invariant, it can refer to before and after states. An invariant effect does not need to be named and cannot use parameters.

```
inv effect Calendar::count_invocations post: count += 1
```

An effect invariant is anded to the postcondition of all actions in its range; it implicitly adds the last clause to this spec:

```
-- remove the given event
action Calendar::removeEvent (e: Event)
    pre:    -- provided the event is on this calendar
            schedule->includes (e)
    post:   -- that event has been removed from the calendar and instructor
               schedules
            not schedule->includes (e) and
            not e.instructor@pre.schedule->includes (e)
            -- and the effect invariant is implicitly applied
            and count += 1
```

By using effect invariant conditions in the postconditions, we can describe effects that apply selectively to any action that meets the condition. For example, here's how to keep a count of all actions that create or remove an event on the schedule:

```
inv effect count_event_creations_and_deletions
    post:   -- if the set of Events before and after differ, count this action
               occurrence
            schedule@pre <> schedule ==> count += 1
```

© ***invariant effect*** A transition rule that applies to the postcondition of every action in the range of the invariant; by writing a conditional (eff1 ==> eff2) you can impose the rule selectively on those actions that have effect eff1.—for example, "all operations that alter x must also notify y."

3.5.5 Context and Control of an Invariant

The type in which an invariant is written is called its context. It applies only to the operations of that type. (In Chapter 4 we will also see contexts of actions between groups of collaborating objects.)

Any object claimed to conform to the type should make it look to clients as if the invariant were always true. While the client is waiting for an operation to complete, the invariant can be broken behind the interface that the type describes; but it must be restored when the operation is complete. Behind that interface are components of the design that have their own nested contexts and invariants that govern them (see Figure 3.15).

As with any specification, it is possible to write invariants that cannot be satisfied by an implementor. In your spec of a Sludge vending machine, you can write an invariant that the weather is always sunny in northern England; but I cannot deliver such a device.[9] But

9. Although, come to think of it, with the right concoctions in the cans ...

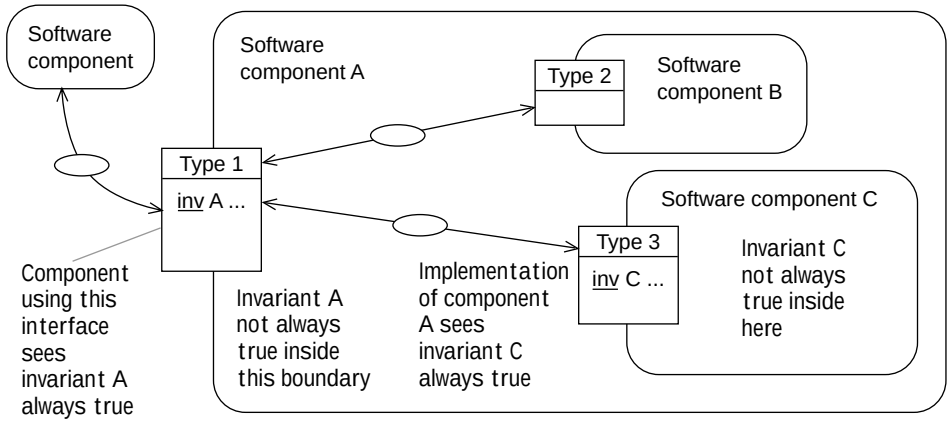


Figure 3.15 Invariants outside their contexts.

if you write an effect invariant that the machine's cashbox will always fill as the can stack decreases, I believe I can do that.

To achieve it, you must employ certain techniques in your design. For example, if you forget to include a stout metal case around the outside, your assets will monotonically decrease: people can get directly at the cans (and any cash that others may have been foolish enough to insert). Similarly, the developer of an alphabetically sorted list of customers cannot guarantee that the list will remain sorted if other designers' code can directly update customers' names. You can guarantee only what you have control over.

In designing to meet an invariant, then, you must think not only of your own immediate object but also of all the objects it uses; and you must be aware of any behavior they have that might affect the specs you are trying to meet. Fundamentally, objects must be designed in collaborating groups—the subject of Chapter 4.

3.6 *Interpreting an Action Specification*

An action specification generalizes all occurrences of the action; in other words, it should hold true for every snapshot pair (Figure 3.16), much as a type model generalizes all snapshots in Figure 3.11. Given a type T with operation M whose operation spec has a precondition P and postcondition Q, we interpret this operation spec as follows:

> *If you examine the history of any object that correctly implements* T *and find in that history any occurrence of the operation* M*, then if* P *was true of the invocation parameters and attribute values immediately before that invocation occurred,* Q *should have become true immediately after that invocation completed.*

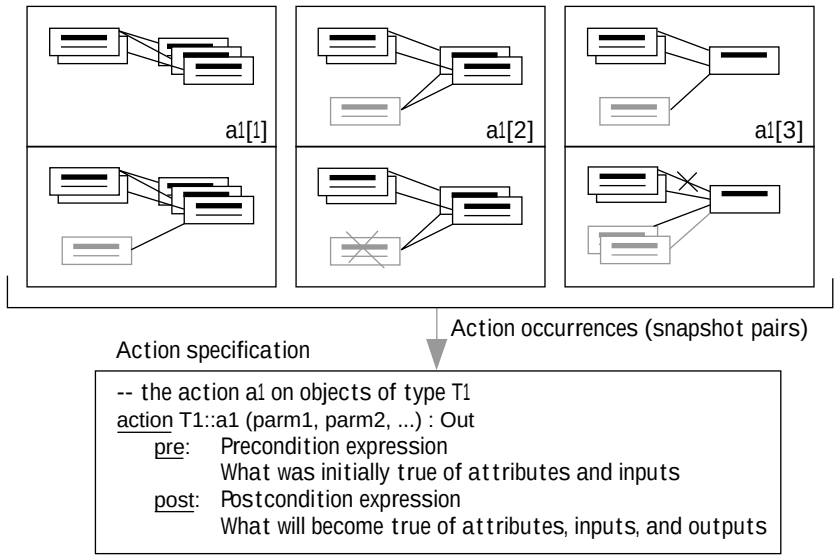


Figure 3.16 Action specification generalizes snapshot pairs.

3.6.1 An Action Spec Is Not an Implementation

Writing a specification for an operation is very different from writing an implementation. The spec is simply a Boolean expression: a relation between the inputs, initial state, final state, and outputs. An implementation would choose a particular algorithmic sequence of steps, select a data representation or specific internal access functions, and work through iterations, branches, and many intermediate states before achieving the "final" state. Consider the specifications of these operations in contrast to their possible implementations:

function Calculator::squareRoot (in x: Real, out y: Real)
pre: not (x < 0)
post: y > 0 and y * y = x -- more realistically, allow rounding errors

action Scheduler::schedule_course (reqCourse: Course, reqStart: Date)
pre: Provided there is an instructor qualified for this course
who is free on this date, for the length of the course.
post: A new Session has been created, with course = reqCourse,
startDate = reqStart, and endDate – startDate = reqCourse.length,
and with one of the qualified free instructors assigned

action FlightRouter::takeShortestPath (f: Flight)
pre: provided there is some path between the source and destination of f
post: f has been assigned a path from its source to its destination
there is no other path between f.source and f.destination shorter than f.path
(needs a supporting definition of path and its length)

These operations, and their corresponding type model attributes, have many possible implementations. Instructor qualification and schedules can be represented and calculated

in many ways, as can flight paths and square roots. No matter how we implement these operations, they must conform to this specification. If we were to run any test data, which met the preconditions of the specification, through an implementation, we would expect the postcondition to be satisfied; if it is not, we have found a bug in either the implementation or the specification.

Some specifications fully determine the outcome of an operation. Our specification of squareRoot allows many implementations and even more than one result: 2* 2 are 4, but so are –2 * –2. If we want to exclude the latter, we would add y>0. Similarly, schedule_course constrains the new session to be assigned a qualified, available instructor but does not specify which one be assigned. And takeShortestPath does not say which path should be selected in the event there were multiple paths with the same length; it says only that there is no path with a shorter length.

What should squareRoot do in the case of a negative input? or schedule_course in the case when a qualified instructor is not available? Our specs, as written, do not cover these other conditions but rather leave these behaviors unspecified. If we said nothing further about these operations, the implementations could ignore these other conditions. However, we can have multiple specs for an action (see Section 3.6.3, Multiple Action Specs: Two Styles).

A good operation specification is much like a test specification. With a little infrastructure support—such as query functions to map from concrete data representations to the abstract attributes used in the specifications and some means to capture initial values of attributes—these operation specifications can be mapped to test code that is executable at runtime, at least during testing or debugging.

The operation specs often map directly to tests. Thus, the spec for squareRoot easily translates into test code; so does schedule_course, after we write query functions to determine attributes related to instructor qualification and availability in terms of the concrete implementation. Some specifications may need to be refactored a bit to be tested effectively. A literal usage of takeShortestPath as a test specification would require generating all possible paths to show that the computed path is the shortest, but that's not a very practical test strategy.[10]

3.6.2 Parameter Types

Parameter types are an implicit part of pre- and postconditions. Our spec of squareRoot could be rewritten so as to make this explicit, although this is not the normal style:

```
action squareRoot (in x, out y)
    pre:    x: Real & not (x < 0)
    post:   y: Real & y*y = x
```

10. You can reformulate the spec to solve this; or you could design it in steps, specify each step, show that the specified steps yield the shortest path, and test only the steps.

We permit a shorthand for parameter types. A parameter that is not explicitly typed has a name that is a lowercase version of its type name. The following spec implicitly types all three parameters:

```
action Scheduler::schedule_course (course, client, date)
```

3.6.3 Multiple Action Specs: Two Styles

The effect of an operation can be specified with an explicit pre/post pair of conditions or with a single postcondition and no explicit precondition. The main difference is that within the explicit precondition (starting with pre:) all references are implicitly to the initial values of attributes; within a single postcondition clause we must explicitly indicate initial values using x@pre. These two are[11] equivalent:

```
action Scheduler::schedule_course (reqCourse: Course, reqStart: Date)
    pre:    a qualified instructor available for those dates
    post:   a new confirmed session with ....

action Scheduler::schedule_course (reqCourse: Course, reqStart: Date)
    post:   (qualified instructor available for those dates)@pre
             ==> (a new confirmed session with ....)
```

Notice the ==>, also written implies or if...then.... If the precondition is not met, we have said nothing about the outcome.

If you have just one specification of the action, both forms are equivalent. The main difference arises when you have multiple specifications of the same action,[12] such as for different views of an action. (For details, see Section 8.1, Sticking Pieces Together.) Following are the guidelines for choosing.

• To write a single complete spec for the action—for example, to define in one contract what the implementor must code, and the caller must be careful of before invocation—both forms are equivalent. The explicit pre/post form makes the client responsibility a bit more visible. The caller is responsible for invoking this operation only when the precondition is true; the implementor can assume the precondition is satisfied and must then guarantee the postcondition.

• To write a partial spec, which will be automatically composed with others for the same action,[13] things are a bit more complicated.

  – To define an outcome that must be guaranteed if your precondition holds, regardless of other partial specs, use a ==> b.

---

11. Different specification languages have used different approaches, including entirely implicit preconditions.

12. As we all know, your write_the_code action must (a) meet the specs, (b) run fast, (c) have wonderful documentation, (d) be completed tomorrow.

13. If the distinction between an explicit pre/post and a single post seems too subtle, use pre/post with explicit keyword restrictive.

Given another partial spec:—c==>d—the combined result is obtained by anding each one, so a client can fully rely on each partial spec:

(a==>b) and (c==>d)

– To define a precondition that can restrict (and be restricted by) the precondition of other partial specs, use pre a post b.

Given another partial spec—pre c post d—the combined result strengthens the precondition and postconditions of both specs:

pre a and b post c and d

Thus, here's how to write two specs for sqrt: one for a valid call and the other for the exception behavior required for negative numbers. (Specifying exceptions is described in more detail in Section 8.4, Action Exceptions and Composing Specs.)

```
action squareRoot (in x:Real, out y: Real)
    post:    not (x < 0) ==>  y*y = x

action squareRoot (in x:Real, out y: Real)
    post:    x < 0 ==>  (y = NAN)
```

An operation can be constrained by multiple specifications. Alternatively, the multiple specs can be combined into a single, more complex specification. Here, first, is an operation constrained by multiple partial specifications:

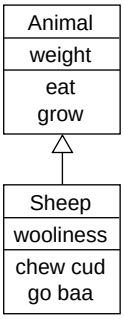
```
-- this spec deals with scheduling a confirmed course
action Scheduler::schedule_course (client, course, date)
    post:    (there is an instructor qualified for this course
              who is free on this date, for the length of the course.) @pre
             ==> A single new Session has been created for that course, client,
             dates and confirmed with one of the qualified free instructors
             assigned to it


-- this spec deals with a "loyalty program" for frequent course schedulers
action Scheduler::schedule_course (client, course, date)
    post:    (the client is above some volume threshold) @pre
             ==> The client has received a certificate for a free course
```

Any reasonable tool should relate multiple specifications for an operation and be able to present some combined form. Here is the same operation written with a single specification.

```
-- this spec deals with combined aspects of a request to schedule a course
action Scheduler::schedule_course (client, course, date)
    post:    (instructor available)@pre ==> single new session for that course, ...
             and (client above threshold)@pre ==> client has received free
             certificate
```

3.7 *Subtypes and Type Extension*



Because a type spec is just a description of behavior, an object can be a member of many types. In other words, it can play several roles. (In fact, an object is a member of every type whose specification it conforms to even if the type specification was written after the object was created.) And one type can be a subset, or *subtype*, of another even if they were defined separately. To say that all sheep are animals is the same as saying Sheep is a subtype of Animal. You expect of Sheep everything expected of Animals in general; but there is more to say about Sheep. Some objects that are Animals—that is, they conform to the behavior specification for that type—may exhibit the additional properties of Sheep.

Putting more into a specification, raising the expectations, reduces the set of objects that satisfy it. It's often useful to define one type specification by extending another, adding new actions, or extending the specifications of existing ones; subject to certain restrictions, this will result in the definition of a subtype.

© ***subtype*** A type whose members form a subset of its supertype; all the specifications of the supertype are true of the subtype, which may add further specifications. (Note that we use *subclass* to mean inheritance of implementation.)

3.7.1 Attributes and Invariants

A type defines a set of objects by specifying certain aspects of those objects; every object that conforms to that specification, regardless of its implementation, is a member of that type, and vice versa. For example, a ServiceEngagement type could define any object that constitutes a service engagement with a client. Any object with a suitable definition of the five attributes is a ServiceEngagement.

A subtype extends the specification of its supertype. It *inherits* all properties (attributes and invariants) of the supertype and adds its own specifics. Because all supertype properties still apply to it and because its members must conform to all properties, every member of a subtype is also a member of its supertype; a subtype's members are a subset of its supertype members.

```
Subtype->forAll (x | x.isTypeOf (Supertype))
```

The types CourseEngagement and ConsultingEngagement could both be subtypes of ServiceEngagement (see Figure 3.17). Objects of the CourseEngagement type have a total of six attributes defined on them; these objects may have different implementations of these attributes as long as they map correctly to the specified attributes and are related consistent with all invariants. Their fees are determined by the fees set for the course; the margin must factor in travel expenses and production costs for student notes. Engagement dates are fixed by the startDate and the standard course duration.

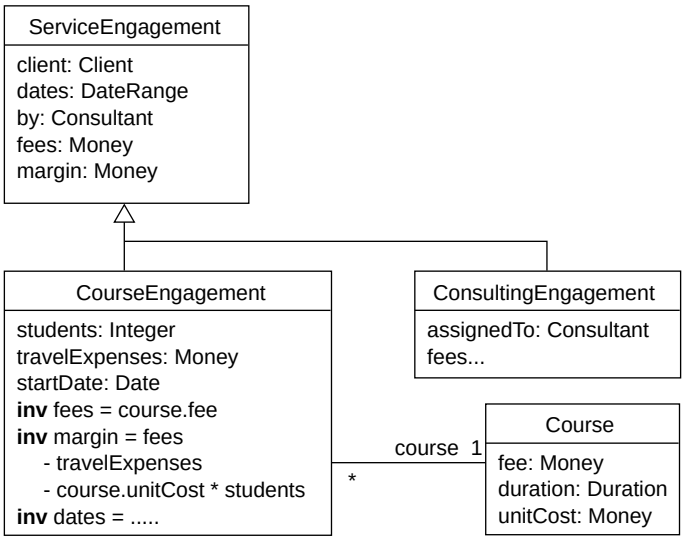


Figure 3.17 Subtype extends definition of supertype.

Clearly, attributes students and course do not apply to consultingEngagements. The rules that constrain dates, fees, and margins could be quite different. Still, all five common attributes can be defined for any consultingEngagement.

We could well discover further commonality between the subtypes: both of them follow the same basic rule for their margin.

margin = fees - travelExpenses - additionalCosts

Parts of this invariant are defined differently for each subtype. Consulting fees are determined by the expertise of the consultant and the length of the engagement. Additional costs for a course reflect the per-student production costs; additional costs for consulting may reflect the preparation time required for the engagement and the actual cost for the assigned consultant. Despite these differences, the broad structure of the invariant is the same and can be defined only once in the supertype.

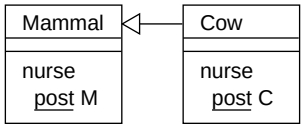
A type is not a class. A class is an object-oriented programming (OOP) construct for defining the common implementation—stored data and executed methods—of some objects, whereas a type is a specification of a set of objects independent of their implementation. Any number of classes can independently implement a type; and one class can implement many types. Some programming languages distinguish type from class. In some languages, writing a definition of a class also defines a corresponding type.[14]

14. Java distinguishes interface (type) from class (type and class). A C++ class is also a type. In Smalltalk, type corresponds to a message protocol; class is independent of type.

A subtype is not a subclass. Specifically, a subtype in a model does not imply that an OOP class that implements the subtype should subclass from another OOP class that implements the supertype. Subclassing is one particular mechanism for inheriting implementation with certain forms of overriding of *implementation*; however, with subtyping there is no overriding of *specifications*, only extension.

As with objects and attributes, there are many ways of partitioning subtypes. ServiceEngagements could be viewed based on their geographic location (domestic versus international), taxation status (taxable or not), nature of service provided (consulting versus training), and so on. Which of these are relevant is determined primarily by the actions that we need to characterize and the extent to which the subtyping helps describe these actions in a well-factored way.

3.7.2 No Overriding Behavior Specifications

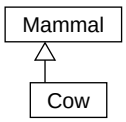


When one type is defined as an extension of another type, the subtype cannot *override* any behavioral guarantees of the supertype. The postcondition written for nurse on Cow does not override the corresponding specification for all Mammals; rather, it is an additional description about how cows nurse their young. In contrast, in an implementation class, a superclass can provide an implementation of some method that a subclass then overrides.

3.7.3 Common Pictorial Type Expressions

There are several commonly used combinations and variations of subtyping in models. This section outlines them and the corresponding notations.

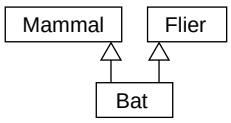
3.7.3.1 Subtype



Cow extends Mammal and it inherits all Mammal attributes and action specs. Cow may add more action specs for the same or different actions. Here it is viewed as sets of objects:

Cow $\subseteq$ Mammal

3.7.3.2 Multiple Supertypes

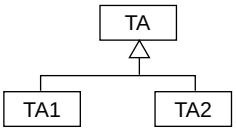


Bat has all the properties specified on the supertypes. Any action with specifications in more than one supertype must conform to them all. Viewed as sets of objects:

Bat $\subseteq$ Mammal * Flier

The subtype conforms to all the expectations that any client could have based solely on the guarantees of both supertypes. Further requirements particular to this subtype may be added. It's perfectly possible to combine two types that have conflicting requirements so that it would not be possible to implement the result.

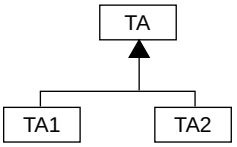
3.7.3.3 Type Exclusion



The two types are mutually exclusive—that is, no object is a member of both.

TA1 * TA2 = ∅

3.7.3.4 Type Partitioning



Every member of TA is a member either of TA1 or TA2 but not both. There may be more than one partitioning of a type, each drawn with a separate black triangle.

TA = TA1+TA2

Figure 3.18 shows an example.

3.7.3.5 State Types

TA1 and TA2 are sets (not true types) to which members of TA belong when in a given state defined by a predicate on TA (usually in terms of attributes of TA, but see Section 3.11). If the determining attributes are not const, then objects can migrate across the state types; otherwise, the classification of an object is fixed by the determining attributes at the time of its creation: *a* TA *is also a* TA1 *if it has property* x<a, *in which case it also has these other properties (*z *etc.)*.

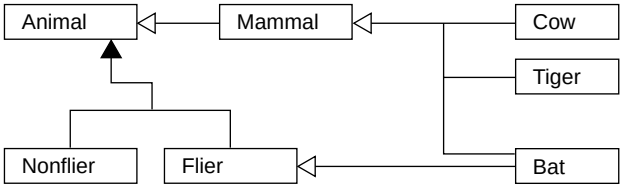


Figure 3.18 Type partitioning.

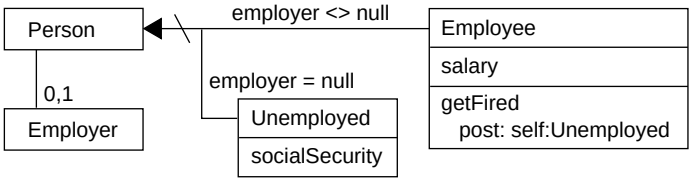


Figure 3.19 State types.

For example, suppose an employee is a person with an employer; employees have a salary (see Figure 3.19) and can get fired. This lets us *classify* objects based on a condition and define resultant properties they will have. For example, we could define two state

types of Person: Teenager (age in the range 13 to 19) and Baby (appropriate baby predicate), ascribing each with appropriate attributes to define:

```
RockConcert::admit (t: Teenager)
    post:    t.stipend.depleted

RockConcert::admit (b: Baby)
    pre:     b.accompanyingAdult <> null-- only if accompanied
    post:    b.isBawling
```

© ***state type*** A set of objects defined by a predicate: unlike a true type, objects may move into and out of it during their lives. The predicate is defined within a parent true type; for example, "caterpillar" is a state within "lepidopter."

3.7.4 General Type Expressions

Because types define sets of objects, we can use set operations to combine types and to define new types without drawing new boxes every time (see Table 3.1).

© ***type expression*** An expression denoting a type using set-like operators—for example, Women + Men.

3.8 *Factoring Action Specifications*

As behavior specs become complex, we need ways to factor them so that they are still understandable and maintainable.

3.8.1 Invariants

We saw in Section 3.5, Actions with Invariants, that invariants are implicitly conjoined with action specifications. Static invariants factor those constraints that apply to every state, and effect invariants capture rules about every state change. Both of them simplify action specs by making them less redundant.

Table 3.1 Type Expressions

Type	Explanation
jo : Student	Object jo is a member of the type Student and conforms to the behavioral requirements set by Student; this notation is the same as the (short form) notation for set membership.
Tutor * Student	The type whose members are in both these types.
Tutor + Student	The type whose members are in either or both types. We might define a type CollegeMember = Professor + Student.
Person – Pilot	The type whose members behave according to the first type's definition but not according to the seconds.
Object	The type to which all objects belong; all other types are its subtypes.
Impossible	The empty type to which no object belongs, characterized by any type definition that is inconsistent.
NULL	Has only one member, null (or ∅), the value of an unconnected link.

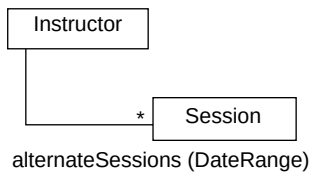
Seq(Phone)	Application of a generic type Seq(X) to a specific type Phone. Other standard generic types include Set and Bag.
[T]	The same as T + NULL; all "optional" attributes, including 0..1 associations, are of this form: their value could be either null or a valid member of T.
enum {on, off}	An enumerated type whose only members are the two listed. You can name this type Status = enum {on, off}.

3.8.2 Convenience Attributes Simplify Specs

Because invariants can simplify action specifications and because attributes themselves simply represent a precise terminology for use in action specs, we can often simplify specs by introducing suitable attributes and invariants.

```
-- cancellation of a session: might need to reassign the instructor to something else
action Scheduler::cancelCourse (s: Session)
pre:    -- if s was confirmed
        s.confirmed
        -- and if there was a tentative session within those dates
        and sessions->exist ( s1 | s1.tentative s1.datesWithin (s)
            -- for which the instructor who was assigned is qualified
            and s.instructor.qualifiedFor (s1.course) )
post:   -- then the instructor is assigned to one such session
```

To simplify this specification, we introduce a new term. The precondition refers to a set of sessions that need an instructor for a particular range of dates—in this case, the dates of the session being canceled. Why not introduce a parameterized attribute alternateSessions(dates) and simplify the operation spec?




```
inv Instructor:: alternateSessions (d: DateRange) =
    -- all tentative sessions within that date range (assuming necessary
       attributes!)
    sessions->select (s | s.tentative & s.dates.within (d)
        -- and that the instructor would be qualified to teach
        & self.qualifiedFor (s.course))

action Scheduler::cancelCourse (s: Session)
pre:    -- if s confirmed and any alternate sessions for that instructor on those
           dates
        s.confirmed and s.instructor.alternateSessions (s.dates)->notEmpty
post:   -- then the instructor is assigned to one such session
```

Judiciously chosen auxiliary attributes like this one can be quite effective in simplifying actions and invariants by introducing precisely defined terms that express the requirement in a natural way that's close to what a client might use (despite the formal syntax).

© ***convenience attribute*** A redundant attribute (possibly parameterized) that is introduced to simplify the specification of actions or invariants—for example, age defined as well as birthday.

3.8.2.1 Parameterized Attributes versus Read-Only Operations

A parameterized attribute, such as alternateSessions(dates), is an abstraction of state. When it is implemented, there will be a way to determine the value of that attribute for each applicable value of the parameter in its range. In this sense, it can be easily confused with a read-only operation: a query that can be invoked as a service returns a value and has no side effect. We choose to distinguish clearly between state abstractions (attributes) and invocable operations.

Specifically, parameterized attributes are constrained by static invariants. In contrast, any operation is defined by its pre- and postcondition (see Section 3.1.2, Pre- and Postconditions Specify Actions); a read-only operation would have a postcondition that defined the returned value in terms of the inputs and current state.

Parameterized attributes can sometimes be modeled using a Map collection type:

```
Instructor::          -- alternate sessions maps a daterange to a set of sessions
    alternateSessions: Map(DateRange, Set(Session))
```

You can conveniently refer to the domain and range as attributes of this map type:

```
alternateSession ->domain : Set(DateRange)
alternateSession ->range : Set (Set(Session))
```

3.8.3 Effects Factor Common Postconditions

Another kind of convenience construct is called an *effect*. This function can use @pre and so can be used to factor out the parts that are common among some of the action specs. For example, schedule_course is an action; we have decided there will be some interaction for a client to schedule a course. Two possible outcomes of this action are schedule_confirmed_course and schedule_unconfirmed_course. We define these as *named* effects with a single postcondition; referring to them by name is exactly equivalent to writing their specifications directly. Effects can be listed on the type box, marked with a stereotype <<effect>>, along with the actions.

```
-- saying that a schedule_confirmed_course has happened is exactly the same
   as saying...
effect Scheduler::schedule_confirmed_course (course, date)
                -- that there was some available instructor initially
post       instructorAvailable@pre (course, date)
                -- and a confirmed session is created
    and         Session.new [confirmed] → size = 1

-- saying that a schedule_unconfirmed_course has happened is exactly the same
   as saying...
effect Scheduler:: schedule_unconfirmed_course (course, date)
                -- that there was no available instructor initially
post       not (instructorAvailable@pre (course, date))
                -- and an unconfirmed session is created
    and         Session.new [unconfirmed] → size = 1
```

We can now simply use these two effects to specify the action schedule_course. The resulting spec means exactly the same as though we had written the full specifications of the two effects.

```
-- when a scheduler schedules a course
action Scheduler::schedule_course (course, date)
    pre:    true -- no precondition, because the postcondition covers all cases
    post:   -- either a confirmed course has been scheduled
            schedule_confirmed_course (course, date)
            -- or an unconfirmed course has been scheduled
    or      schedule_unconfirmed_course (course, date)
```

© ***effect*** A convenience postcondition introduced (and named) to factor parts of postconditions that are common across more than one action. Unlike ordinary predicates, an effect can contain the special postcondition operator @pre.

3.8.4 Pre ==> post versus pre & post

In the preceding example, the two alternative situations and the associated outcomes are represented in two different effects. Within the effect, the @pre part is anded with the post. This means that when we bring the two effects together in the eventual action spec, we can say, “Either this happens or that.”

An alternative style is to write the effects so each of them is a self-contained specification: “In this case ==> always do this” and “In that case ==> always do that.” This style means that the two specifications are anded, because they are both instructions that we want the implementor always to observe:

```
-- If an instructor is available, the course must be confirmed
effect Scheduler::when_instructor_available_confirm (course, date)
            -- if the instructor is available:
post    instructorAvailable@pre (course, date)
            -- then a confirmed session is created
            ==> Session.new [confirmed]

-- If an instructor is not available, the course must be unconfirmed:
effect Scheduler:: when_no_instructor_reject (course, date)
            -- if there was no available instructor initially ...
post    not (instructorAvailable@pre (course, date))
            -- then an unconfirmed session is created
            ==> Session.new [unconfirmed]
```

And we then explicitly compose these two effects differently:

```
-- You must always confirm or unconfirm a course depending on instructor
   availability:
action Scheduler::schedule_course (course, date)
    pre:    true -- no precondition, because the postcondition covers all cases
    post:   -- either a confirmed course has been scheduled
            when_instructor_available_confirm (course, date)
            -- or an unconfirmed course has been scheduled
    and     when_no_instructor_reject (course, date)
```

Which style should you choose? Nice examples can be found to support either style; they have different meaning, and the choice also influences the composition of specs, errors and exceptions, and so on.[15] Experience suggests the following guidelines.

- Write the effect postcondition in a (pre ==> post) style when you wish to ensure that there is no getting out of the contract and that if the precondition is true, then the post-condition will be met. Then combine them into actions using and. This approach is generally better when you're combining several separately defined requirements—for example, when you're building a component that conforms to the interfaces expected by several different clients.
- Write the effect postcondition in a (pre & post) style when you wish each to describe one of many possible outcomes. Then combine them into actions using or. This style is generally better when you're building a specification model from different parts within the same document. You must combine these effects with your eyes open: none of them makes any guarantees that the outcome it describes will be met, because they may restrict each other.

3.8.5 Referring to Other Actions in a Postcondition

Whenever any action is specified, it implicitly defines an effect. That effect can be referred to from another action. Sometimes you want to say, "This operation does the same as that, but also ..."—in other words, to reuse the specification of another action. Specifications can be quoted within others' postconditions (this is analogous to calling subroutines in code). However, you need not do this too eagerly. Specifications must be clearly understandable, and abstract attributes and effects can be used quite freely; it is the implementations that must use careful encapsulation and hiding.[16]

Suppose there were an operation by which an assigned instructor could be explicitly unassigned:

```
action Scheduler:: unassign_instructor (s: Session)
    -- s was previously confirmed, and its instructor is no longer assigned to s
post: s.confirmed@pre ==> s.instructor <> s.instructor@pre
        -- and various other "unassign" things take place
```

You may find that the action of canceling a session might also need to accomplish all the effects of unassigning. You can *quote* the spec of another action in [[...]]:

```
action Scheduler:: cancel_session (s: Session)
post:    s.confirmed ==> [[ unassign_instructor (s) ]]
```

The quoting syntax [[...]] is a predicate, possibly involving initial and final states, that says, "This action achieves whatever unassign_instructor would have achieved, with these parameters," but not necessarily by invoking that action. It doesn't say how this must be achieved; the designer might know another way to achieve the same effect. If you want to

15. This was a major difference between the specification languages Z and VDM.

16. Try telling a client, "I won't tell you what that operation does; it is encapsulated."

go into the semantics a little more, the quotation is the same as rewriting [[...]] with all the ((pre)@pre ⇒ post) of the quoted operation, with appropriate parameter and self substitutions.

If you decide that a part of what this action must do is to actually invoke a specific operation, you can record that decision by inserting an arrow in front of the operation:

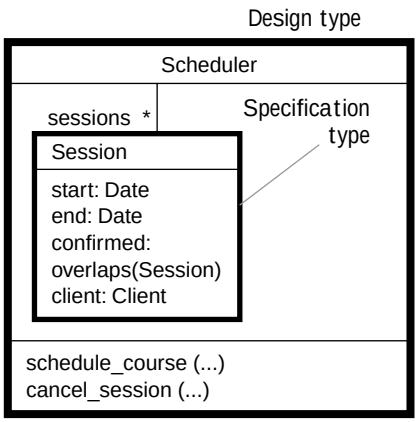
```
[[ –> self.cancel_session (s) ]]
```

This technique alters the postcondition to mean "The cancel_session operation has been invoked on self." The end result is no different, but we're now pinning down how to achieve it. You can quote operations on objects other than self with or without the ->.

© ***quoted action*** A postcondition can refer to another action by naming it within brackets: [[action(...)]]; this is called *quoting*, and it means that the effect specified for that action is a part of this postcondition. If written as [[->action(...)]], then the action must actually be invoked as part of the postcondition; if further prefixed with sent, it indicates that an asynchronous invocation must be made.

3.8.6 Specification Types versus Design Types

Specifications describe how a client can use a component. What we really want to say about a component is *what it does*: its behavior, or the actions it takes part in. We have seen how to specify the externally visible behavior of a type by specifying actions in terms of a type model of attributes. Because the component doesn't necessarily have to be implemented along the same lines as its model, the implementation need not explicitly represent distinct objects that belong to the types used within that model. Those types are used to structure the static model and relate it to the business.



It is possible to implement the Scheduler without a Session class and without unique and distinct objects for each session; such an approach would be quite common when you're assembling existing implementation components or legacy systems, such as a calendar (with events) and an employee database (with employees). When designing, some people like to distinguish those types they have decided to implement—*design* types—from those that are used simply to help write a specification—*spec* types. Design types can be drawn with a heavier border.

Clients are not interested in how it works inside: only the designer is interested in that, and the designer should be in a considerable minority. But to describe the actions clearly, we must write a model of the component's state. Here is a typical dialog or thinking process involved.

> "This command creates a session." *What's a session?* "It's a thing with dates, courses, and instructors." *What can I do with it?* "Oh, you can't get hold of one directly, but you can list all the sessions there are and cancel any of your own." *So there is one big list of sessions in there?*

> "Yup." *Isn't that a bit inefficient, considering we would need to search the whole list to determine schedules? Wouldn't it be better to organize by date and instructor?* "Uh, well yes, that's how it's really done, of course; and actually, there's really no such thing as a session. We just append the course and instructor names to events in the calendar. And actually, there's this hashtable..." *But if I think of it as a list of sessions, I'll understand how to use the system?* "Yes." *Thanks, that's what I need.*

In this scenario, not only the attributes of the scheduler system—the list of sessions—but also the type of objects they contain (Session) amount to a convenient fiction hypothesized to describe its behavior aside from all the implementation complexities. Session is a specification, or model, type. It is there only for the purpose of modeling. Types that are "really" there (in the sense that they are separable and take part in actions and we intend to implement them) are design types.

Many types are used for both purposes. For example, Date is often used in specifications and also has many implementations, and a good design would often have direct implementation of the specification types. Also, in some situations the specification requires an implementation not only of a primary type but also of related types required for input and output parameters. An example is shown in Step 8 in Section 3.4.1.

Typically, a design type will be specified with a model drawn inside it using specification types. Only a design type can participate in an action, and every type that is specified as participating in an action is a design type. Specification types do not really have actions of their own, but partial specifications (effects) can be attached to them for convenience, as shown in Section 3.8.7.

However, there is nothing to stop you from using a type in a model even though the type happens to have an implementation somewhere. In fact, good implementations of domain objects will often have their specs reused in this way. The more important design decision hinges on how the types in an implementation will be used, and those decisions involve joint actions recorded in collaboration diagrams.

© ***specification type versus design type*** A specification type is one that is introduced as a part of the type model of another type to help structure its attributes and effects in terms closer to the problem domain. The behaviors of the spec type are not themselves of interest, and the type may never be implemented directly.

A design type, in contrast, is one that participates directly in actions; its behaviors are of primary importance, and it is not just a means to factor the specification of some other type.

3.8.7 Factoring to Specification Types

In many cases the outcome of an action depends on the type of object, or objects, to which it is applied; indeed, this is one of the mainstays of the object-oriented approach. If our seminar system dealt with training and consulting engagements, the rules for canceling an engagement might be different. The appropriate parts of the effect of cancellation should be localized on the engagements (see Figure 3.20).

The outcome is different for each type of engagement: training sessions have the course material production canceled and instructor reassigned; consulting jobs may be charged for, confidential materials must be returned, and the consultant must be reassigned.

Although the spec is simplified by factoring it across the different types of engagements, the action is simply on the scheduler system; committing to more would be a matter of internal design.

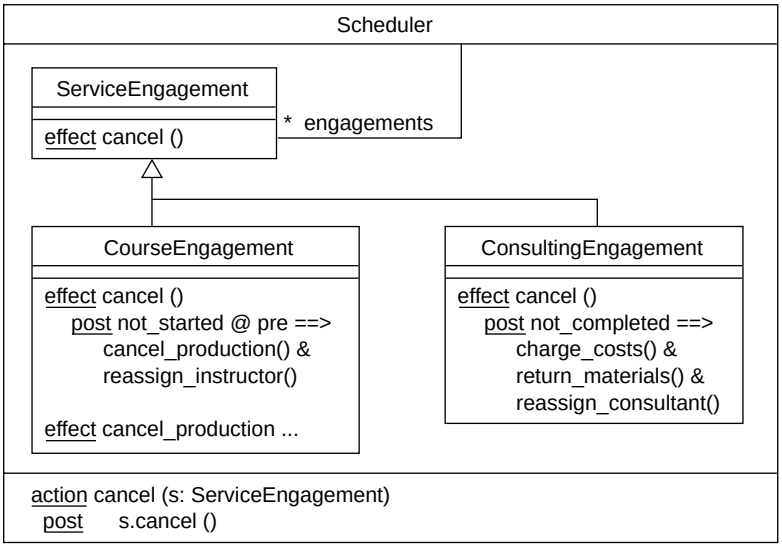


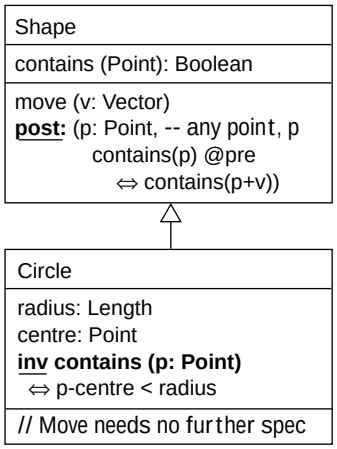
Figure 3.20 Localizing effects on specification types.

Does this mean that we are doing some design—assigning responsibilities and deciding internal interactions to the modeling types? Not really. We're only distributing the action spec among the concepts on which it has an effect. If there happens to be an implementation of, say, ConsultingEngagement, we are not referring to that implementation and the particular properties of its code: we're referring only to the specification. That said, the most straightforward design approach would parallel this spec localization.

3.8.8 Factoring: Specify Effects Abstractly

The earlier example placed very different outcomes in each type. But where possible, it pays to look for something common among the supertypes. The cancel operation could be defined abstractly:

```
effect ServiceEngagement:: cancel ()
post:      can_be_cancelled@pre ==>
      reassign_resources() &
      cancel_preparations()
      ..etc.
```



The subtypes would detail their own specs for can_be_cancelled and the other effects. This kind of factoring is the specification analog of the "template method" design pattern when you're implementing methods in superclasses. A straightforward design approach would parallel this factoring in the implementation, but alternative design choices would not invalidate the factored specification.

In some cases, we can even provide a complete spec in the supertype, with the subtypes simply defining the abstract attributes in more detail. For example, moving a Shape in a graphical editor is very different in terms of the attributes of each subtype; but we can express in common terms the required effect of what happens to the points contained within each Shape.

Although the most natural attribute models for circle, rectangle, and triangle would be quite different, we can abstract them into a single parameterized query, contains (Point); any shape is defined by the points it contains. We define move in terms of this abstract attribute and then simply relate the different shapes to this attribute.

3.9 *State Charts*

State charts, and their simpler cousins, "flat" state diagrams, can be useful modeling tools. In Catalysis, states and transitions that appear in a state chart are directly related to the attributes and actions in a type specification. The state chart merely provides an alternative view of the spec.

3.9.1 States as Attributes and Invariants

Sometimes it is easy to see distinct states that an object progresses through over its lifetime. A Session may go through tentative, confirmed, or delivered; if it is either confirmed or delivered it is considered sold. From another perspective, the session may be pendingInvoice, invoiced, or paid.

States are often drawn in a *state chart* showing the states and the relationships between them, as in Figure 3.21. Each state is a Boolean attribute[17]: an object either is or is not in that state at any time. The structure of states in the state chart defines invariants across these attributes.

- States in a simple state chart are mutually exclusive, with exactly one state true at a time, such as within sold; this is what the xor invariants mean.

17. We qualify the state name with superstate names to deal with nested states.

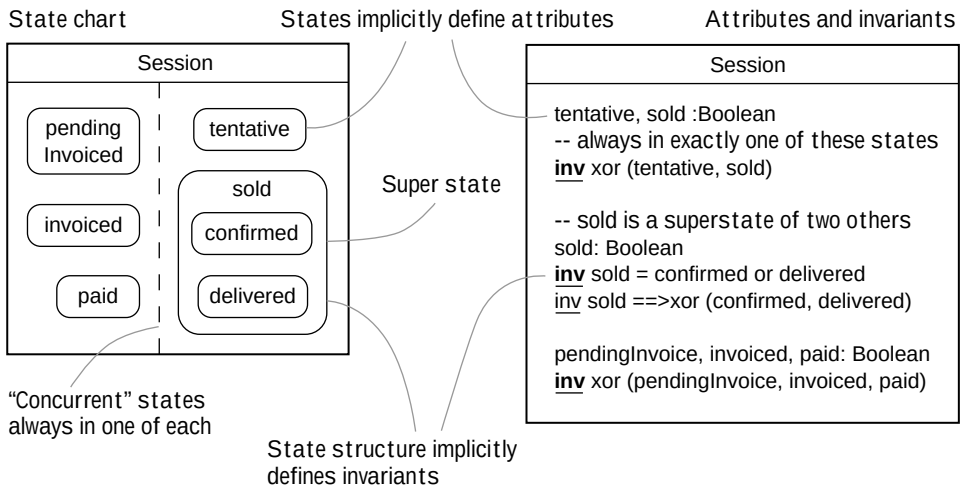


Figure 3.21 A state chart defines state attributes and invariants.

- A state chart can be nested inside a state. While the containing state is false, none of the nested states is true; while it is true, the nested state chart is live, meaning that one of its states (or one from each of its concurrent sections) must be true. This is the or invariant defining sold.
- A state chart can be divided into concurrent sections by a dashed line. Each of these sections is a separate simple state chart. The object is simultaneously in one state from each of the sections. No explicit invariants are needed because the two sets of states are independent. There is no paradox in this, nor necessarily any concurrent processing in the usual sense: it's just that a state simply represents a Boolean expression, and there is no reason that two such statements should not be true at the same time.

To represent business rules, we can separately introduce invariants that eliminate certain combinations, For example:

```
inv Session::      invoiced ==> sold
    -- A session can be invoiced only if it is sold
```

Because states simply define Boolean attributes, it is easy for states to be tied to the values of other attributes and associations via invariants. So for example, we can write

```
inv Session ::      -- an invoiced or paid Session always has an attached invoice
    (invoiced or paid) = (invoice <> null)
```

thereby tying the state to the existence of a link to another object.

© ***state*** A Boolean attribute drawn on a state chart. The structure of the states defines invariants on those attributes (such as mutually exclusive states, inclusive states, or orthogonal states); additionally, you should write explicit invariants relating the state attributes to other attributes in the type model.

3.9.2 State Transitions as Actions

In addition to defining state attributes and their invariants, state charts depict transitions between states. An example state chart for Session is shown in Figure 3.22: A session can be confirmed, tentative, or delivered; changing the dates might switch between tentative and confirmed.

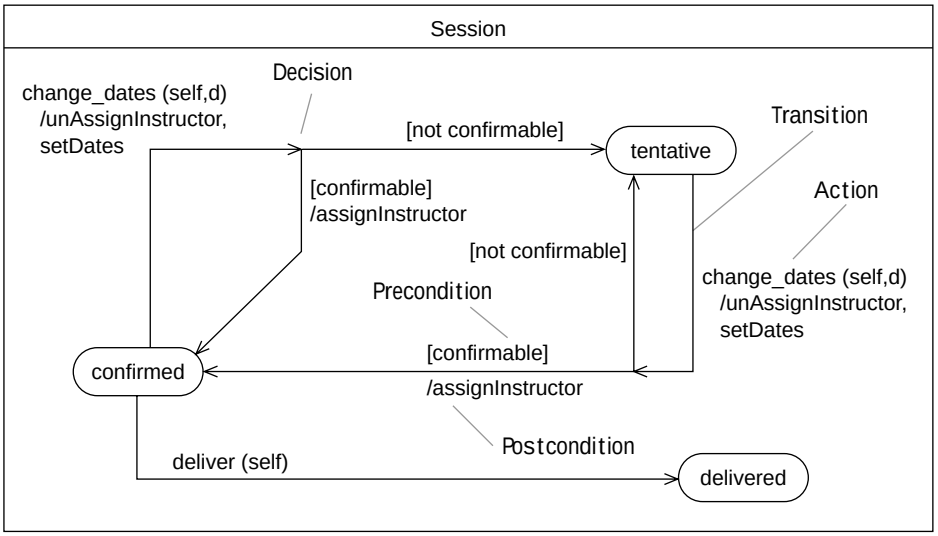


Figure 3.22 Session state chart with transitions.

The change_dates action has multiple transitions, which translate into multiple partial action specifications. The transition from the confirmed state is translated next. For brev-

ity, the state chart omitted the parameters used in the pre- and postconditions, but we fill these in the textual action specs.

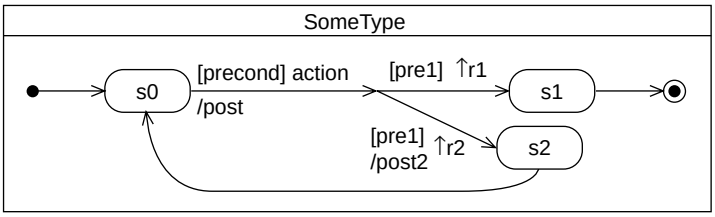
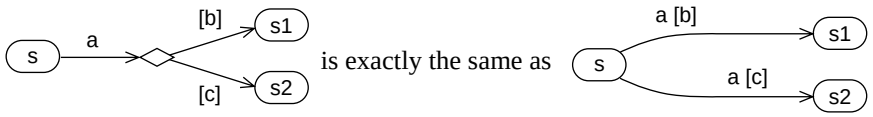


Figure 3.23 State chart notation.

```
action change_dates (s: Session, d: date)
-- if s was confirmed, i.e., transition coming out of the confirmed state
pre:    s.confirmed
post:   unAssignInstructor(s)  &   -- assuming an effect with that name
        setDates (s, d) &          -- assuming effect is defined
        (confirmable (d) @pre ==> s.confirmed & assignInstructor (s, d)) &
        (not confirmable (d) ==> s.tentative)
```

A state chart is part of the model of a type. The elements of state chart notation are shown in Figure 3.23.

The arrows indicate the sequences of transitions that are possible. • indicates the state that is first entered when the state predicates first become well defined. This can mean when the object is first created or when this nested state chart is entered. ◉ indicates where state predicates become undefined: when the nested state chart is exited or the object is no longer of any interest. Until reaching the "black hole," all the predicates for the states in the diagram should be well defined, either true or false.



The decision point is not a branch point in the programming sense but instead says that there are two possible outcomes of this action: its postcondition includes (s1 or s2). Without the preconditions, either outcome would satisfy this spec. Decision points can sometimes simplify transitions: Figure 3.22 could be simplified to a single decision point, with creation and date change actions transitioning via that shared point.

State transitions share the machinery of action specs clauses, which are described in Section 3.6. State transitions can be used to specify actions as well as named effects; a transition labeled with the keyword effect represents a named effect (Section 3.8.3) rather than an action.

[pre] is a precondition: the transition is guaranteed to occur only if pre was true before the action commenced. Notice that this does not say that this transition definitely does not occur if the precondition is false; to say that, make sure you show transitions going elsewhere when it's false.

With /post, some effect is achieved as part of executing this transition.

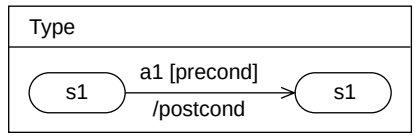
↑action means that a (more abstract) action is completed as part of executing this transition. We'll have more to say about this in Chapter 6, Abstraction, Refinement, and Testing.

[[receiver.action]] means that part of the effect of this transition is the same as the documented effect of action on receiver (which is self, the state chart object, by default).

[[–>receiver.action]] means that part of the effect of this transition is that action is actually performed by an invocation on receiver.

3.9.3 Translating State Transitions to Actions

A transition illustrates part of the spec of an action.




```
action        Type::a1
    pre:          s1 & precond
    post:         s2 & postcond
```

If there are several transitions involving one event, the effects are conjoined. State charts give a different way of factoring the description of an action, and a good tool would move readily between two views: state chart and textual action specification. Each state must be defined in terms of other attributes.

When you're using superstates, being in any substate implies being in the superstate. So any arrow leaving the superstate means that it is effective for any of the substates.

Transitions indicate the *completion* of the actions with which they are labeled. Although accomplishing the actions may take time, the transitions themselves are instantaneous; this will become more significant in Chapter 6, Abstraction, Refinement, and Testing.

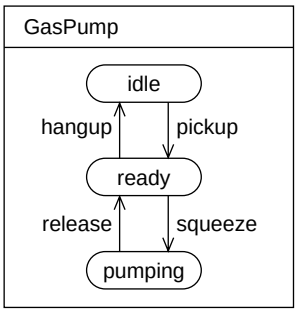
© ***state transition*** A partial specification of an action, drawn as a directed edge on a state chart. The initial and final states are part of the pre/post condition in the spec, and additional pre/post specs are written textually on the transition.

© ***state chart*** A graphical description of a set of states and transitions.

3.9.4 State Charts of Specification Types

When you're drawing state charts, be aware of the primary type that is being modeled. In simple cases the states and transitions are directly of the primary type being modeled. If we are trying to specify the behavior of a gas pump, the states and actions labeling the transitions are those of the pump itself. It translates directly into action specs such as these:

action Pump::hangup	pre: ready	post: idle
action Pump::pickup	pre: idle	post: ready



But when the primary type being modeled is complex, its states cannot necessarily be enumerated in the simple form required for representing it as a single state chart. The behavior of a Scheduler component such as the one in Figure 3.13 cannot be described on a single state chart (except with the most trivial states, exists, and all the interesting effects described in text on the transitions). This is because the state of the scheduler is defined by the states of its multiple sessions, instructors, and courses.

Instead, you should draw separate state charts for the specification types that constitute the type model of Scheduler. In reality, we are defining the states of the scheduler in terms of the states of its specification types. The transitions in the individual state charts show what happens to those objects for each of the scheduler's operations (see Figure 3.24).

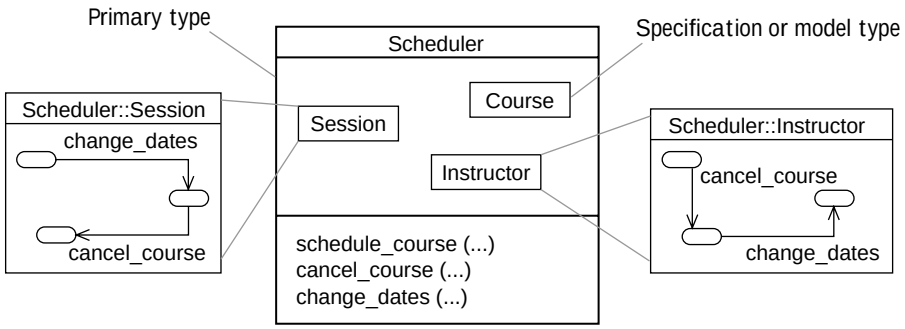
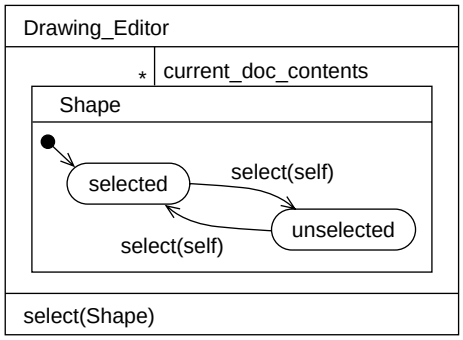


Figure 3.24 State charts of specification types.

Each individual state chart effectively specifies how every action on the primary type affects that one specification type. In contrast, a complete action specification defines how one action on the primary type affects any specification type member. The composition of all change_dates transitions, on any and all specification types, constitutes change_dates operation specification for the scheduler. Do not confuse this state chart view with internal design, when we will actually decide internal interactions between objects within the scheduler, the primary types whose behavior we describe will be these internal objects.



A useful technique in specifying a large component is to draw a state chart that focuses on all the elements of a particular type within a larger model—for example, showing what happens to the shapes in a drawing editor for each of the *editor's* operations. select(self) is a shorthand for select(s) [s=self]. It's important to realize that this is really a state chart for the editor, in which the states are defined in terms of the states of its shapes.

We can translate this to text form as follows.

It is slightly more convenient to use a single postcondition clause than to separate the pre/post style.

```
action   Drawing_Editor::select (shape:Shape)
   post:    -- every shape in the current document is affected as follows
            current_doc_contents ->forAll ( s |
                 -- if it's the target and was selected, unselect it
                 ((s.selected & s=shape)@pre ==> s.unselected)
                 -- if it's the target and was not selected, select it
          &      ((s.unselected & s=shape)@pre ==> s.selected)
            )
```

3.9.5 Underdetermined Transitions

Sometimes a state chart is deliberately vague about the outcome of an action. The reason is usually to allow subtypes to make different choices, within broad constaints given by the supertype, or simply to define a minimal partial constraint.

At any moment, a transition is said to be *feasible* if, before the current action began, the system was in the state at the arrow's source end and if any precondition it is labeled with was true. You are allowed to write a state chart for which there are several feasible transitions at any moment; this is called an *underdetermined* set of transitions. When this is the case, what state will you end up in?

The answer is that you will end up in one of them, but as a client you can't make any assumptions about which one it might be. This doesn't mean it's random, only that there are forces at work that you, based solely on the current spec, are unaware of. As a designer you might be able to choose whichever you like; but you will probably be constrained by the requirements from another view or a particular subtype.

For example, dialing a phone number—dial_number—has several possible outcomes. As users we are unaware of the factors that will influence the outcome (see Figure 3.25).

There is an engineer's view in which you can describe what the outcome will be in terms of the capacity of the lines and whether the other end is engaged on a call; but from the point of view of the phone user at one phone, these factors are unknown.

Isn't it a bit pedantic to insist on drawing the picture that doesn't show the preconditions on dial_number? After all, moderately educated phone users know what really causes these outcomes, and even when they don't, there is always a cause that we, the designers of the phone system, know about.

Perhaps that's true in this case. But indeterminate state charts will be important when we discuss components. This component may be combined with a wide variety of others, including ones not yet known of; so we actually don't know what the causes are, only what the possible outcomes can be. This situation might happen if we allowed our phone instrument to be connected to a new kind of switching system. As long as we have a way of reusing this underspecified model and adding to it in another context, it is worthwhile to make this separation.

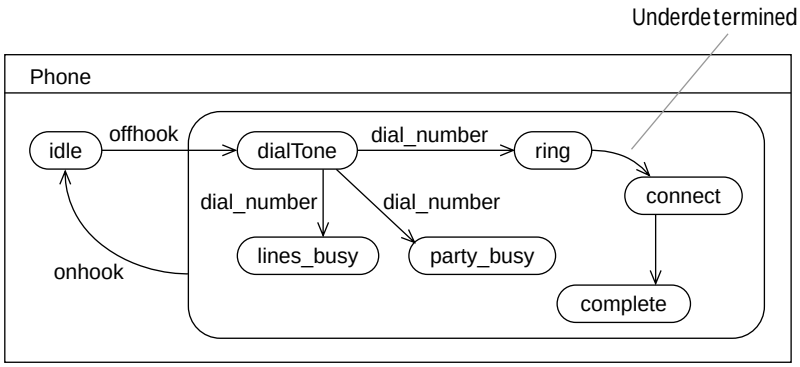


Figure 3.25 Underdetermined transitions.

3.9.6 Silent Transitions

The phone example shows one other way in which state charts can show nondeterminate behavior. It is possible for a system to change state without your knowing why or when and without your doing anything to it. Again, *silent* transition does not imply randomness. Instead, it means either that we don't know what might cause the change (as when we're waiting for public transport, which no mortal understands) or that, if we do know, we don't know whether or when that cause will happen (as when we're hanging on to see whether the phone will be answered).

Silent transitions also let us describe systems that are not purely reactive, because the partial descriptions permit transitions with unknown causes.

3.9.7 Ancillary Tables

State charts themselves provide a useful view of behavior that is different from the view provided by action specifications. State charts focus on how all actions affect one specification type, highlighting sequences of transitions, as opposed to focusing on the complete

effect of one action on all affected types. Two related tables can be helpful in conjunction with state charts to check for completeness and consistency.

First, a state chart can be represented by a state transition matrix, as shown in Figure 3.26. In the matrix, X means shouldn't happen; | means nothing happens; and [ ] means determined in subtypes. Writing the matrix is a valuable cross-check to ensure that each action has been considered in each state. This matrix can be generated automatically from the state charts themselves; it better highlights combinations that may have been overlooked.

The second type of table is a state definition matrix. Each state should be defined in terms of attributes and associations in the model. Frequently, these boil down to simple conjunctions of assertions about them and so are easy to show in a table (see Figure 3.27). Even if this table is never written, it is always useful to define each state as a function of the existing attributes and associations.

State	e1	e2	e3
S1	[g1] S2	X	I
S2	X	S3 / pp	X
S3	S2	[ ]	S1

Figure 3.26 State transition matrix.

State	attr1	assn1	assn2	Full Definition of State
S1	>30	null	<> null	attr1>30 & assn1=null & assn2 <> null
S2	>2, < 3	null	<> null	2 < attr1 < 3 & assn1=null & assn2 <> null
S3	> 0	<> null		0 < attr1 & assn1 <> null

Figure 3.27 State definition matrix.

3.10 *Outputs of Actions*

Much of the focus of a typical postcondition is on the effect of an action on an object's internal state. But we also need to describe the information that results from an action and is returned to the invoker as well as any output signals or requests that are generated to other objects. There are several approaches to this.

3.10.1 Return Values

An action can have a return type whose return value is the identity of a new or preexisting object used by the sender. Within the postcondition, **result** is the conventional name given to this value. Any actions can have a return value; those that have no other side effects are called *functions*.

```
function square_root (x:float)
    post: abs (result * result – x) < x/1e6
```

3.10.2 Out Parameters

Input parameters represent object references that are provided by the caller; return values represent object references returned to the caller to deal with as needed. Although out parameters can be broadly considered similar to return values, the details are somewhat different. The postcondition of the operation will determine the value of the out parameter and its attributes; however, the client will call this operation with these out parameters bound to an attribute selected by the client.

```
action Scheduler::schedule_course (course, dates, out contract)
    post: .... & contract: Contract.new [...]
action Client::order_course
    post: scheduler.schedule_course ( c1, 11/9, self.purchase_order.contract)
```

An ordinary parameter refers to an object; an out parameter refers to an attribute, which might be as simple as a local variable of the caller. An out parameter can therefore be used to specify that a different object is now referred to by the bound attribute; an ordinary parameter can only change the state of the object it refers to.

out is like a C++ reference parameter. Other programming languages, such as Java and Eiffel, do not have these features, showing that it is possible to do without them in an implementation language. However, the idea of having multiple return values is itself convenient in both specification and implementation.

3.10.3 Raised Actions

It is also possible to state as part of a postcondition that another action has been invoked either synchronously or asynchronously.

• *Synchronously:* The sent action will be completed as part of the sender. Its postcondition can be considered part of this one. It is written

```
[[ r := –> receiver.anAction(x,y) ]]
```

r is a value returned from the message.

• *Asynchronously:* The request has been sent; the action will be scheduled for execution later, and its completion may be awaited separately.

– Request sent to a specific receiver; action has been scheduled.

```
[[ sent m –> receiver.anAction(x,y) ]]
```

m is an event identifier that can be used elsewhere.

– Request sent to an unspecified receiver; action has been scheduled.
  [[sent m –> anAction(x,y)]]
– A previously sent action has been completed and returned r.
  [[m (...) = r]]

3.10.4 Specifying Sequences of Raised Actions

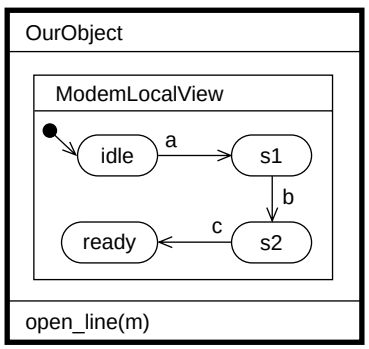
When you specify raised actions, it is sometimes necessary to specify that they happen in a particular sequence—to describe the protocol of a dialog. You might want an open_comms action to send certain messages to a modem object in a particular order. You might wish to specify this while retaining your basic premise of using only initial and final states in postconditions.

In effect, you are telling your designer something about the type of the intended receiver. Even if you tell me absolutely nothing else about it, I know that it can accept messages a, b, and c in a particular order and that at certain times you require me to have sent all three in that order.

This is equivalent to saying that I have been asked to get it into the state of "having received message c." I haven't been told what that state might signify as far as the receiver is concerned, only that I must get it there. But also, you tell me that I must first send message b: In other words, the modem has a state—as far as I am concerned—of "having received message b," which is a precondition of c.

So the simplest way to specify that a sequence of messages must be sent as part of the outcome of an action is to make a minimal local model of the state transitions for the receiver and specify that the final target state is reached:

```
action OurObject:: open_line(m:LocalViewOfModem)
pre:    m.idle
post:   m.ready
   &    various effects on our own state
```



By using the full apparatus of state charts, we can specify linear, branching, and concurrent sequences.

Used in this way, states shown as separate in this local view may turn out not to be separate in an implementation of the modem; they are, however, separate in our object's implementation, because it must generate them in sequence. OurObject would work if we provided a modem that ignored operations a and b but went straight from idle to ready on c. Provided that the intermediate states are purely specification attributes and that there are no operations for actually finding out its state, that's OK.

Sequential outputs are sometimes better dealt with as refinements, as described in Chapter 6, Abstraction, Refinement, and Testing. State charts are also used to describe a collaboration refinement (in which zooming in on an action shows it to be a dialog of smaller ones).

Occasionally it is useful to write a spec that refers to more than just the before-and-after moments in time—say, i, j and j, k. Putting the time indexes in [...] sets the scope within which the states are referred to.

```
action [i, j, k] T::m (...)
    post:    (x@j = x@i +3 ) &          -- x@j means value of x at time "j"
             (x@k = x@j +2 )
        ==>      (x@k = x@i + 5)
```

The main use for this would be in complex specs where you want to say, "The overall effect is the same as doing this followed by that" and in program proofs. Without explicit time indexes, you simply assume before and after and use @pre to distinguish them.

3.10.5 Sequence Expressions

Occasionally it is useful to write a sequencing constraint in text form, although it could usually be described using the preferred state chart technique from the preceding section. For example:

```
action Scheduler:: cancel_session (s: Session)
    post:
        [[   free_instructor (s.instructor@pre) ;
             reassign_instructor (s.instructor@pre) ;
        ]]
```

© ***sequence expression*** A textual representation of a temporal composition of actions; some sequence expressions can be translated into an equivalent state chart.

Sequence-expression [[..; ..; ..]] shows the permitted sequence of more detailed actions—not a prescribed program. The elements of the syntax are as follows, where S1 and so on are usually expressions about actions:

- S1 ; S2 S1 always precedes S2
- S1 | S2 S1 or S2
- S1 * Any number of repetitions of S1
- S1 || S2 S1 concurrent with S2

All such sequence expressions [[...]] are an abbreviation for a state model with the implication

```
(start ==> done)
```

The two states are defined by a state model as shown in Figure 3.28.

3.11 *Subjective Model: The Meaning of Containment*

When we specify a type, we often depict its attributes and specification types with a distinguished *root* type by using a form of visual containment, as shown in Figure 3.29(a). (An equivalent, alternative form distinguishes type nodes marked with «root» and selected

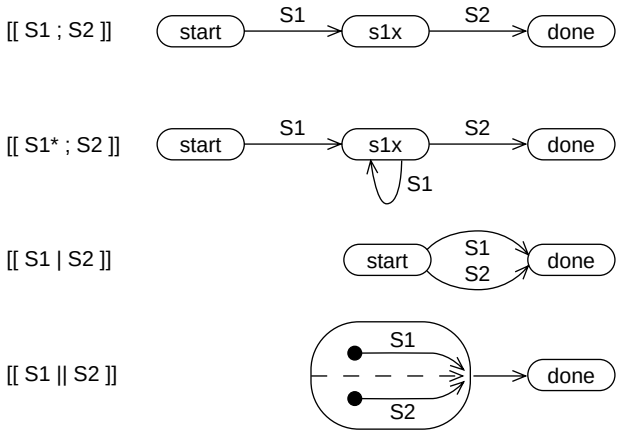


Figure 3.28 State chart equivalents for sequence expressions.

associations marked with «cross<Root>» in (b).) This is more than a cosmetic choice; it has a specific semantic meaning. Both diagrams in Figure 3.29 are equivalent to the

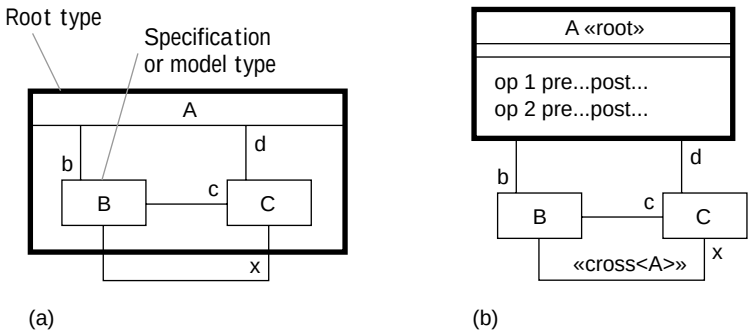


Figure 3.29 Type model: (a) with containment, (b) with root.

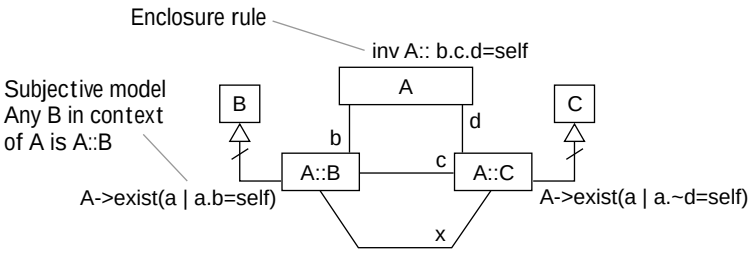


Figure 3.30 Interpretation of containment.

expanded model in Figure 3.30.

In the *enclosure rule*, all paths from A back to A that use only associations defined within the box (or that do not use any links marked «cross<A>») are guaranteed to get back to the same instance of A; that is, all links lie within the tree of objects that is rooted at self:A. By contrast, although a.b.**x**.d is certainly a member of type A, it need not be = a.

Thus, an engine is connected to a transmission, provided that they are within the same vehicle. This rule works as expected across multiple levels of nesting.

In the *subjective* model, the types locally named B and C, and their associations b, c, and d, all form part of a A's model; they are really *state types*. In other words, those members of B (or C) that happen to be contained within an A also satisfy any additional properties stated about B or C on this diagram. (Specifically, those additional properties do not

universally apply to all B or C objects.) Brought outside the boundary of A, they have prefixes to their names.

In a different context, not every B will necessarily be linked to a C. For example, to the Invoicing department, customers are linked to products, whereas Warehousing knows only that products have parts; customers, universally, may not be required to have either association. Containment represents a localized view, which is a state type of the common usage.

In addition, local usages can also directly refer to attributes or operations on their container.[18] This makes it simple to write localized specifications of actions and effects, as discussed in Section 3.9.4.

As a matter of style, we use containment to depict specification types that have been introduced simply to describe operations on the primary type of interest. Sometimes we also need to describe operations on the input or output parameters of these operations, as shown in Figure 3.11.

Without such a mechanism, it becomes difficult to know whether the properties defined on B are intrinsic (that is, apply universally) to all Bs, or whether those properties are defined only on those Bs that happen to be within an A. When an engine runs, does it always turn the wheels of a car? How about when it is propelling a boat? Or when it is mounted on a test jig at the mechanic's?

3.12 *Type Specifications: Summary*

This chapter deals in detail with the business of specifying actions—what happens in some world or in some system—while deferring the implementation. Indeed, we have seen an example (Section 3.4) of how two different implementations can have the same behavioral specification. The action specifications use the terms defined in a static model (as in Chapter 2). The static and action models together make up the specification of a complete type (see Figure 3.31). In the chapters that follow, we will use these ideas to build specifications of complete software systems and interfaces to components.

- An object's behavior (or part of it) can be described with a type specification.
- A type specification is a set of action specifications; they share a static model that provides a vocabulary about the state of any member of the type.
- An action spec has a postcondition that defines a relationship between the states before and after any of its occurrences takes place.
- A precondition defines when the associated postcondition is applicable.

In Chapter 4 we deal with the interactions between objects—both inside the object we have specified (as part of its implementation) and between our object and others—to understand how it is used by our (software or human) clients.

18. In the manner of *inner classes* in JavaBeans and *closures* or *blocks* elsewhere.

Exhibits 3.1 and 3.2 show the syntax of action specs and postconditions.

3.13 *Programming Language: Classes and Types*

Our focus in this chapter has been on specifying the behavior of objects using types and not on how to implement them with classes. This section briefly describes the link between modeling with types and implementing with classes.

© ***class*** (a) A language-specific construct defining the implementation template for a set of objects, the types it implements, and the other classes or types it uses in its implementation (including by class inheritance).

(b) An implementation concept that defines the stored data and associated procedures for manipulating instances of that class; the implementation construct can be mapped to OO languages and to procedural and even assembly language.

A class is an implementation unit that prescribes the internal structure of any object that is created as an instance of it. Class is an OO programming concept but not necessarily an OO programming *language* concept. There are patterns for the systematic translation of OO designs to other data and execution models. You can employ these patterns if, for example, you need to write in a traditional language such as Fortran or assembler, perhaps for special control of performance. In that way, you still get the benefits of OO design (modularity, reuse, and so on). Of course, OO programming languages best support object design. OO-to-non-OO patterns must also be applied outside the scope of your programming language.

For example, C++ works with an OO model in main memory but leaves persistent data up to you; you can't send a message to an object in filestore. If you can secure a good OO database you're in luck; but otherwise, typically you're stuck with plain old files or a relational database and must to think how to encode the objects. Your class-layer design should initially defer the question of how objects are distributed between hosts and media.

So there is a *class layer* of design described entirely in terms of classes, with related types, which can be implemented directly in a language such as Java, Eiffel, or C++ or otherwise by judicious application of class-to-nonclass patterns (see Figure 3.32).

Not all OOP language (OOPLs) have classes. In Self, an object is created by cloning an existing one. Objects delegate dynamically to others rather than statically based on their class inheritance; and methods can be added dynamically. Nevertheless, Self designs certainly use the idea of Type. Contrary to many early popular writings, polymorphism is about types—multiple implementations of the same interface—and not about classes and implementation inheritance.

Exhibit 3.1 Action Specifications

There are five constituents which may be part of an effect spec:

ReceiverType :: (parameter1 : Type1, parameter2 :Type2) : ResultType

Signature: a list of parameters—named values (references to objects) that may be different between different occurrences of the action(s) the spec governs.

- Some parameters may be marked out, denoting names bound only at the end of the operation.
- May also define a result type and a receiver type.

pre: *condition* — **Precondition** is a read-only Boolean function that defines the situations in which this effects spec is applicable. If the precondition is not true when an action starts, this spec doesn't apply to it, so we can't tell from here what the outcome will be. There might be another applicable spec defined somewhere else.

The parameter type constraints are effectively terms in the precondition: d1>3 and d2:Date and d1<d2 ...

May refer to the parameters, to a receiver self, and to their attributes, but not out parameters or any result.

post: *postcond* — **Postcondition** is a read-only Boolean function that specifies the outcome of the action (provided the precondition was true to begin with). The postcondition relates two states, before and after, so, the prior state of any attribute or subexpression can be referred to using @pre.

A postcondition may refer to self, all the parameters, any result, *and* their attributes.

rely: *condition* — For concurrent or interleaved actions, if the rely clause ever becomes false during the execution of the action, the implemention is no longer guaranteed to meet the specification (Section 4.5).

guarantee: *cond* — The implementor will maintain this true while executing the action concurrently with others, provided the pre and rely conditions hold (Section 4.5).

Pre, post, rely, and guarantee conditions are called *assertions*. Two further assertions appear in a type specification, outside any one action spec:

inv *condition* — True before and after every action in the model. Effectively anded to each pre- and postcondition.

inv effect *effect* — A "global" effects spec that applies to all actions conforming to its signature, pre, and rely conditions.

Exhibit 3.2 Special Terms in a Postconditions

A postcondition relates together two moments in time. By default, every expression denotes its value once the action is complete.

x@pre	The prior value of attribute x (there is no need to use them in a precondition). Here's an example of moving rooms: jo.room@pre -- jo's old room jo.room@pre.isDirty@pre -- prior state of jo's old room jo.room@pre.isEmpty -- current state of jo's old room jo.room.isEmpty@pre -- prior state of jo's new room
new	The set of all objects that exist in the after state that did not exist in the before, so T*new=(T – T@pre). Common usages with new in a postcondition: Egg*new -- all new Eggs from this action Egg*new [size>5] -- all new Eggs satisfying the filter Egg.new -- more familiar syntax for Egg*new Egg.new [size>5] -- new Eggs, with size>5 e : Egg.new[size>5] -- e is a new Egg with size>5 This is most commonly used within a let; it implies that there is at least one new Egg.
[[an action]]	Action quoting, equivalent to copying its specification into the present postcondition. It does not imply a necessary actual invocation of the action — just that the same effect is achieved. If there are several effects-specs applying to the quoted action, they are all implied.
[[->an action]]	The quoted action will definitely be invoked in an implementation.
result	Reserved names for the value denoted by an operation that a programmer can invoke as an expression. Here's an example: action square_root (x : int) post: x = result * result ... y = square_root (64); // y == 8

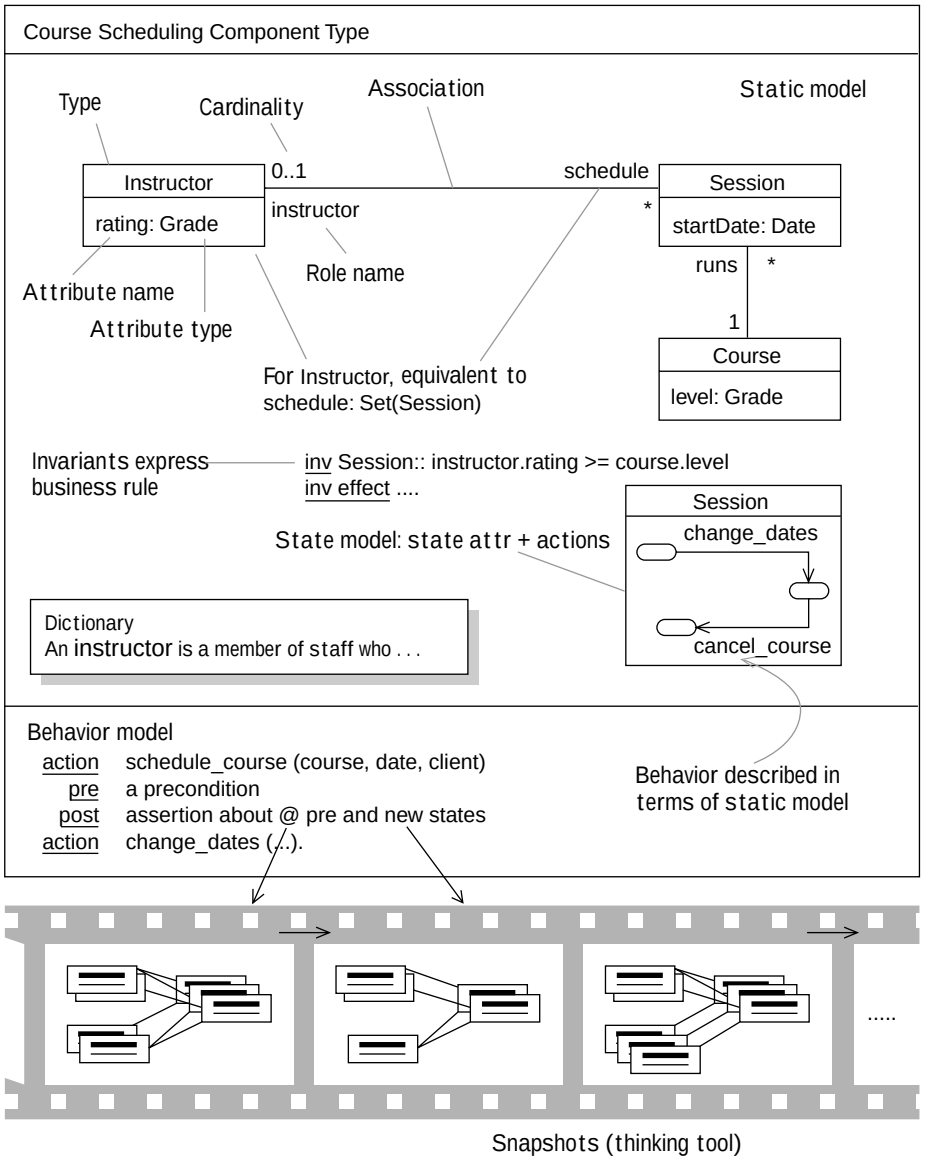


Figure 3.31 Type models and behavior specification.

3.13.1 Messages and Operations

A class contains code for the operations the object "understands"—that is, the operations for which there are specifications, and hence clients could expect to send it.

• A *message* is an invocation of an operation. It consists of the name of the operation, the identity of the recipient, and a set of arguments. A message is like a procedure call except

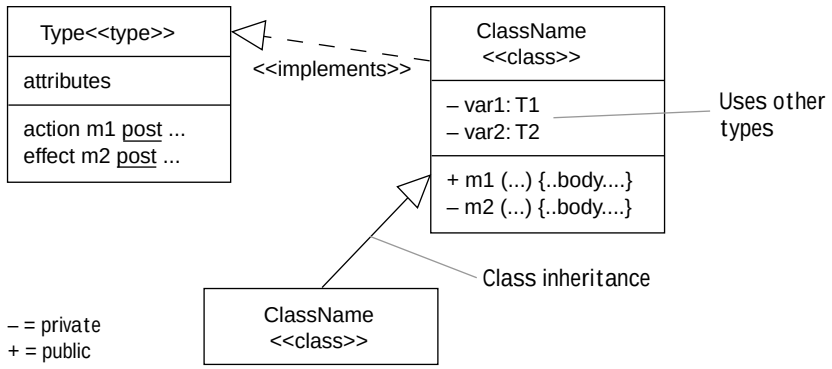


Figure 3.32 Class models.

that it is a request *to* an object; the same message sent to different classes of object can have different outcomes. "Who performs this operation?" has an interesting answer in an OO program; in a conventional program, it's just the computer!

• An *operation* is a procedure, function, or subroutine. The traditional OO term is *method*, but we prefer to avoid confusion with the method you follow (hopefully) to develop a program. In analysis, and at a more abstract level of design, we talk about actions: an action occurrence may be one or more operation invocations.

• A *receiver* is the object distinguished as determining which operation will be invoked by a given message. It is usually thought of as executing the operation, which has access to the receiver's variables.

Not all OOPLs have a receiver. In CLOS, it is the combination of classes of all the parameters that determines what method will be called; methods are not specifically attached to classes.

3.13.2 Internal Variables and Messages

There are four primary kinds of variables in an object-oriented program.

- *Instance variables*: Data stored within fields in each object
- *Parameters*: Information passed into, and out of, methods
- *Local variables*: Variables used to refer to temporary values within a method
- *Class variables*: Data stored once per class and shared by all its instances.

Operations read and write variables local to each object. A variable refers to an object. It is important to distinguish variables from objects, inasmuch as one variable may be capable of containing at different times several types of object (for example, different kinds of Shape) that will respond to messages (such as draw) differently. In some cases, several variables may refer to one object.

Every variable has several key features.

• *Type*: The designer should know what types of object can be held in a variable—that is, the expected behavior of the object to which it refers. In Self and bare Smalltalk, this is left to the design documentation; in C++, Java, and Eiffel, it is explicit and some aspects are checked by the compiler. Explicit typing is allowed in some research variants of Smalltalk because it makes it possible to compile more-efficient code; other compilers try to deduce types by analyzing the code.

• *Access:* This refers to which methods can get at a variable. In Smalltalk, all variables are encapsulated per object: Methods cannot get at the variables of another object of any class. In Java, variable access can be controlled within a package (a group of classes designed together) or at a finer grain. This arrangement makes sense, because each package is the responsibility of one designer or team; any changes within the package can readily be accommodated elsewhere within the same package. Encapsulation is important only between different pieces of design effort. C++ has access control per class: an intermediate position, making an intermediate amount of sense.

• *Containment*: In Smalltalk, Eiffel, and Java, all variables contain references to other objects—implicit pointers that enable objects to be shared and allow the uses of an object to be decoupled from its size and the details of its internal declaration. In C++, some variables are explicit pointers, and others contain complete objects. The latter arrangement yields faster code but no polymorphism;þthat is, one class is tied to using one specific other. This is not a generic design. In general, we consider containment to be a special and less usual case.

Within this book, we assume that variables are typed, that access can be controlled at the package level, and that variables contain implicit references—in other words, the scheme followed by Java, Eiffel, and others.

3.13.3 Class Extension

Inheritance, derivation, or extension mean that the definition of one class is based on that of one or more others. The extended class has by default the variables and operations of the class(es) it extends augmented by some of its own. The extension can also override an inherited operation definition by having one of its own of the same name.

Various complications arise in inheritance from multiple classes. For example, both superclasses may define an operation for the same message, or both superclasses themselves inherit from a common parent, and each language will provide some consistent resolution convention. Java and standard Smalltalk prohibit multiple inheritance of implementations for this reason.

Inheritance was at one time widely hailed as the magic OO mechanism that led to rapid application development, reduced costs, and so on. “Programming by adaptation” was the buzzphrase: You program by adding to and overriding existing work, and you benefit from any improvements made to the base classes. In fact, this approach turned out to be useful to some extent but only under adherence to certain patterns connected with polymorphism.

In general, if you want to base your code on someone else’s implementation, which happens to suit your need, it is best to use your favorite editor to copy and paste.[19] Unless you

want the spec of the other person's code, it's quite unlikely you'd want to inherit any modifications he or she makes. In fact, the big benefit of OO design comes more from polymorphism—conformance of many classes to one spec. If, in some cases, this is achieved by sharing some code, then it's nice but not necessary. Arbitrary code-sharing only couples designs that should be independent.

3.13.4 Abstract Classes

So ideally, a class should be extended only if the extension's instances will be substitutable wherever the superclass is expected. For example, if a drawing builder is designed to accept a Shape in one of its operation's parameters, then a Triangle should be acceptable because, presumably, the latter does everything a Shape is expected to do.

That raises the question of what a Shape is expected to do, and that takes us into the next section on types (object specifications). In programming languages, it is common for a class to represent a type. The class may perhaps define no internal variables or operations itself but instead only list the messages it expects. The rest is defined by each subclass in its own way.

A class that stands for a type, and that may include partial implementation of some operations, is called an *abstract* class. It should be documented with the full spec of the type.

It is now widely accepted good practice that nearly every class should either be an abstract class (prohibited from having instances but possibly with a partial implementation) or a *final* class (prohibited from having extensions).

Types help *decouple* a design—that is, make it less dependent on others. Ideally, each class should depend only on other types and not specific classes. In that way, it can be used in conjunction with any implementors of the types it uses.

But there is one case in which this does not work well: When you want to create an object, you must say which class you want it to belong to. However, there are a number of patterns, such as Factory, that help localize the dependencies, so that adding a new Shape to the drawing editor (for example) causes only one or two alterations to be necessary to the existing code.

3.13.5 Types

In a program, a type should be an implementation-free specification of behavior. Different languages provide different support for types:

A Java *interface* is a pure type—that is, client-visible behavior. You define interfaces for the major categories of clients you expect to have. You factor your services into different interfaces, and you can define some interfaces as extensions of other interfaces (sub-

19. If the code is available.

types) to offer suitable client views. This approach provides those clients with a pluggable type requirement, in which any object that provides that interface can be used.

```
interface GuestAtFrontDesk {
    void checkin();
    void checkout();
}
interface HotelGuest extends GuestAtFrontDesk, RoomServiceClient {
    ...
}
```

A class implements any number of interfaces and also implicitly defines a new type. Behavioral guarantees should be defined on interfaces but are not directly supported by the language itself. You cannot instantiate an interface, only a class.

```
class Traveler implements GuestAtFrontDesk, AirlinePassenger {
....
}
```

Now let's look at C++ A Java interface is very similar to a C++ *pure abstract* class, with only pure virtual functions and no data or function bodies.

```
class GuestAtFrontDesk {
    public:virtual checkin() = 0;
           virtual checkout() = 0;
};
```

Similarly, a Java extended interface is like an abstract subclass, still with all pure virtual functions.

```
class HotelGuest :public GuestAtFrontDesk, public RoomServiceClient {
    ....
};
```

You can use macros to make the distinction more visible in C++:

```
#define interface class
#define extends public
#define implements public
```

In Smalltalk, when a client Hotel receives a parameter x, that client's view of x can be defined by a set of messages HotelGuest={checkIn, checkOut, useRoomService} that the client intends to send to x. Hence, the type of x, as seen by that client, is the type HotelGuest. The language does not directly support, or check, types; but you can systematically use facilities such as *message categories* or *message protocols*.

- A client expects an object to support a certain protocol, HotelGuest.
- Any object with a (compatible) implementation of that protocol will work.
- It is often convenient to get that compatible implementation by subclassing from another class, but it never matters *to the client* whether or not we subclass; we could just as well cut and paste the methods, delegate, or code it all ourselves. In Smalltalk the only check is a runtime verification that each message sent is supported.

- The only time the client needs to know the class is to *instantiate* it.

The class of an object is not really important to that client as long as it supports the protocol. In Smalltalk this can be represented by systematically following programming conventions that use a message protocol or message category as a type.

3.13.6 Generic Types

A generic definition provides a family of specific definitions. For example, in C++, a template class SortedList<Item> could be defined in which everything common to the code for all linked lists is programmed in terms of the placeholder class Item. When the designer requires a SortedList<Phone>, the compiler creates and compiles a copy of the template, with Item replaced by Phone.

There are variants on the basic generic idea.

• *What is generic*: In C++ and Eiffel, classes and operations are the units of genericity. In Ada, in many well-thought-out experimental languages, and—with any luck—in a future version of Java, packages are generic. This means that you can define a generic set of relationships and collaborations between classes in the same style as the frameworks (Chapter 9, Model Frameworks and Template Packages). We argue that this is a very important feature of component-based design.

• *When validated*: C++ template classes are (and can be simulated by) macros, mere manipulations of the program text before it gets compiled. The template definition itself undergoes few compiler checks. This means that if the design of SortedList<Item> performs, say, a < comparison on some of its Items, the compiler remarks on this only if you try to get it to compile a SortedList<some class that doesn't have that comparison>. A big disadvantage of C++ template classes is that they cannot be precompiled: you must pass the source code around. By contrast, the generic parameters of FOOPS [Goguen] come with "parameter assumptions" about such properties. When first compiling the generic, the compiler will check that you have made all such assumptions explicit and can guarantee that it will work for all conforming argument classes.

Catalysis frameworks have parameter assumptions in the form of all the constraints placed on placeholder types and actions; these frameworks span single types and classes, to families of mutually related types and their relationships.

3.13.7 Class Objects

In Smalltalk, a class is an object just like everything else. A class object has operations for adding new attributes and operations that its instances will possess. Most commonly, only the compiler makes use of these facilities, but careful use of them can make a system that can be extended by its users or that can be upgraded while in operation. For example, an insurance firm might add a new kind of policy while the system is running. In this "reflexive" kind of system, there is no need to stop everything and reload data after compiling a new addition. Java offers comparable facilities.

Java also supports such a reflexive layer: Classes, interfaces, methods, and instance variables can all be manipulated as runtime entities, although in a more restricted form than in Smalltalk. The language still needs work in this area.

In open systems design, it is important that an object be able to engage in a dialog about its capabilities just as, for example, fax machines begin by agreeing on a commonly understood transmission protocol. This comes naturally to a reflexive language; others must have the facility stuck on. C++ has recently acquired a limited form of such a feature with runtime type identification (RTTI).

In C++, the static variables and operations of a class can be thought of as forming a class object but with limited features. There is no metaclass to which class-object classes belong and no dynamic definition of new classes.

3.13.8 Specifications in Classes

Classes are units of implementation, and they need clear links to specifications.

3.13.8.1 implements Assertions

To say that a class *implements* a type means that any client designed to work with a specific type in a particular variable or parameter should be guaranteed to work properly with an instance of the class.

In Java, types are represented by interfaces and abstract classes. Even though the complete specification of the type (pre- and postconditions and so on) is not understood by the compiler, the following clause documents the designer's intention to satisfy the expectations of anyone who has read the spec associated with the interface Food.

```
class Potato implements Food ...
```

Java allows many classes to implement one interface, directly or through class extension. The following clause should mean that the class implements the type represented by its superclass as well as extending the definition of its code.

```
class HotPotato extends Potato ...
```

In each case, the interface and abstract class referred to should be documented appropriately with a type specification.

In C++, public inheritance is used to document extension and implementation. private inheritance is used for extensions that are not implementations (apart from the simple restrictions mentioned earlier); but the usual recommendation is to use instead an internal variable of the proposed base type.

3.13.8.2 Constructors

A *constructor* has the property that it creates an instance of the class and thereby a member of any type the class implements:

```
class Circle implements Shape {
    ...
```

```
public    Circle (Point centre, float radius);
          // post return:Circle—the result belongs to this Class
          // — from which you can infer that return:Shape
```

Constructors should ensure that the newly created objects are in a valid state—that is, that they satisfy the expected invariants.

3.13.8.3 Retrieval

A fully documented implementation claim is backed up by a justification; the minimal version is a set of *retrieve* functions (see Section 6.7, Spreadsheet: Operation Refinement). Writing these functions often exposes bugs.

For every attribute in the type specification, a function (read-only operation) is written that yields its value in any state of the implementation. This retrieval can be written in executable code for debugging or test purposes, but its execution performance is not important. The functions are private. They are useful for testing but not available to clients.

```
interface Shape {
    attribute[20]  bool     contains (Point); // type model attribute
    public         void     move (Vector v);
                   // post (Point p,
                   //    old(self).contains(p) = contains(p.movedBy(v)))
}

class Circle implements Shape {
    private    Length     radius;
    private    Point      center;
    private    bool     contains (Point p)      // retrieval
               { return (p.distanceFrom(center) < radius); }
    public     void       move   (Vector v) { ... }
}
```

3.13.8.4 Operation Specs

An operation can be specified in the style detailed earlier in this chapter. You can refer to the old and new values of the internal variables (and to attributes of their types, and of the attributes' types, and so on).

Eiffel is among the few programming languages to provide directly for operation specs, but they can, of course, be documented with an operation in any language. In C++, suitable macros can be used; Java could use methods introduced on the superclass Object. For debugging, pre- and postconditions can be executed.

20. This takes liberties with Java syntax. A suitable preprocessor could convert attributes to comments after typechecking them or leave it as code for testing purposes.