Chapter 4 Behavior Models — Object Types and Operations

Outline

In Catalysis, we separate the internal design of a component from its external behavior. Behavior is described by specifying the component's type — a list of actions it can take part in, and how it responds to them.

The type description in turn has two parts:

- the Static Model of an object's internal state, using attributes and associations and invariants;
- specifications of the effects of the actions on the component, using the vocabulary provided by the Static Model.

We dealt with the Static Model in the previous chapter; this present chapter deals with specifying actions. 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. This chapter describes how to derive and write precise action specifications, and how to interpret them.

At this stage, the objective is just to specify the actions, not implement them (though we will look at some program code as examples). The latter part of this chapter will also briefly discuss programming language classes, and how they relate to the specifications. The key to an implementation is the how the objects inside the component collaborate together to provide the effects specified here. Such collaborations will be the subject of the chapter after this.

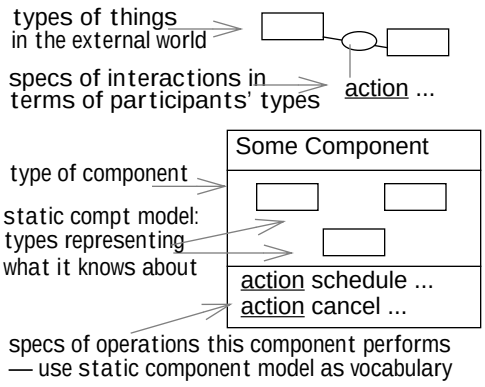
4.1 *Object Behavior — objects and actions*

In component-based development, you have to construct software from components you can't see the insides of: you have to treat them more or less as black boxes. And you have to construct your own components so that they will work with a wide variety of others: components that aren't interoperable have a low value. So it's not just that you are denying yourself a peep inside some specific black box: it's that there are so many, each with its own special features, that the only option is to isolate the features that you need. (As we said before, 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!)

For that reason, we are interested in separating external specification of behavior from the internal works. Ideally, we would like to describe the operations a component performs, without any reference to anything inside; but as we observed at the start of the previous chapter, that isn't possible: the instruction label on the black box has to include some sort of picture of what's inside — even if only a hypothetical picture. This is the Static Model. It can be very much simplified, so long as it provides a vocabulary for describing the operations; and provided the resulting model is accurate enough that users get the results they expect.

4.1.1 Business models, component models, object models

Near the end of the previous chapter, 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 — which we showed graphically by drawing the type of the component containing the types of the objects it 'knew' about.



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

4.1.2 Snapshot pairs illustrate actions

Actions change object state

Over their lifetime, objects undergo state changes as a result of actions. For example, if we had the object snapshot depicted in the top half of Figure 55, and a client requested a session of the javaCourse, then we might end up with the snapshot depicted in the lower half of Figure 55. The new session is assigned to paulo, as he is qualified to teach that course. An *occurrence* of the scheduleCourse action *separates the two snapshots*.

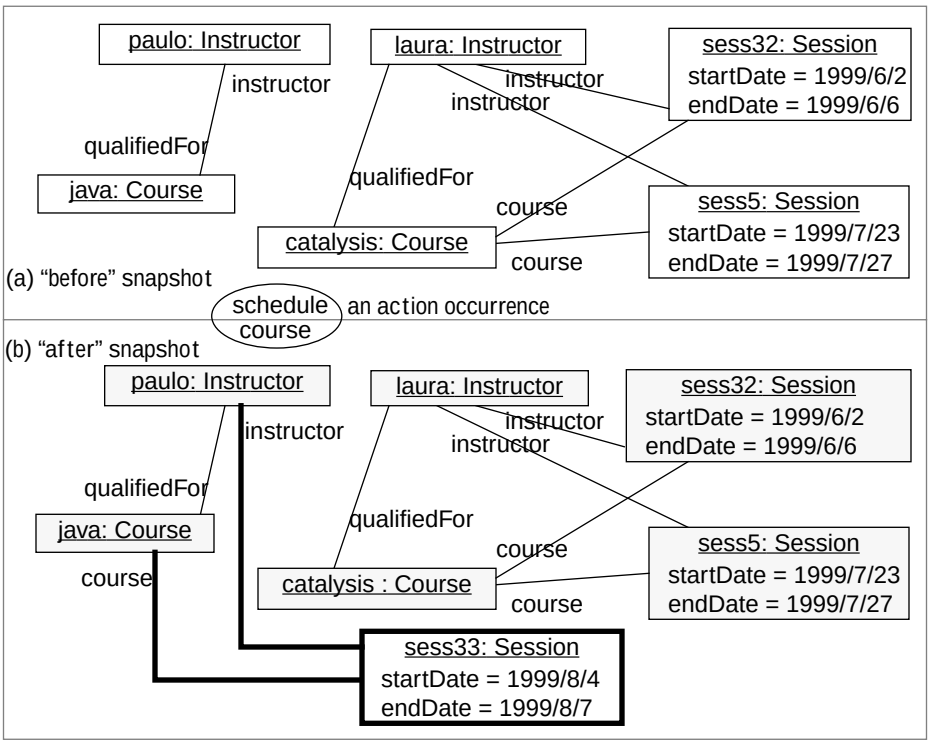


Figure 55: An action causes a change in state

Snapshots illustrate actions

These before / after snapshots provide a useful way to envisage what each action does. Looking at the diagram, can you see what "cancel(session-32)" would do? And what about "reschedule(session-5, 2000 / 1 / 5)", or "qualify(paulo, catalysisCourse)"?

Type model attributes help describe actions

This is the primary reasons for making a model of object state: 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 very difficult to describe the effect of schedule course without the model attributes depicted on the snapshots.

4.1.3 Pre and Post-conditions specify effect of actions

The effect of an action can be specified

The limitation of snapshots is that they show particular example situations. Of course, we want to describe what effect an action has in all possible situations. We can do that by writing 'postconditions' — informal statements or formal expressions that define the effect of an action, using the same 'navigation' style as invariants in Section 3.5, "Static Invariants," on page 115. 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.

Postconditions are (partial) specifications

Notice that we have only stated 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 just those characteristics as 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, permitting many realizations.

and can be combined easily.

It is also very easy to combine requirements expressed in this way: different needs can be **and**'ed together — something you can't do with chunks of program code. And different versions, expressed in different subtypes, can add their own extra constraints to the basic requirement (see Section 9.4.5, "Joining type specifications is not subtyping," on page 375).

4.2 *More precise postconditions*

Postconditions can be used as the basis for writing a test harness: valuable when developing any complex system; and even more valuable when you are couipling together a variety of components from who knows where! (More in Chapter 7.)

To be useful in this way, we should therefore write the postconditions in a more precise style. They should be boolean functions, and they should be read-only — a postcondition that changes what it is testing is no use! For this purpose, you can use the boolean-expression part of your favorite programming language. Here, we use a general form called Object Constraint Language [OCL]. It translates readily to most programming languages, but being purpose-built for specification, has one or two features that make it less clumsy in this context than, say, C++.

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 focussing on the most important ones, and getting a better end result.

The rest of this section deal with key features of the more precise style. It is applicable to both business and component modelling. Later sections differentiate the two, and go into more details of action specification.

4.2.1 Using snapshots to guide postconditions

A postcondition states what we want the end result to be.

For example, let's suppose one instructor may be the mentor of one other: perhaps some of them get too outrageous in class from time to time. The action of assigning a mentor is, informally:

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

This can be shown on a pair of snapshots (Figure 56).

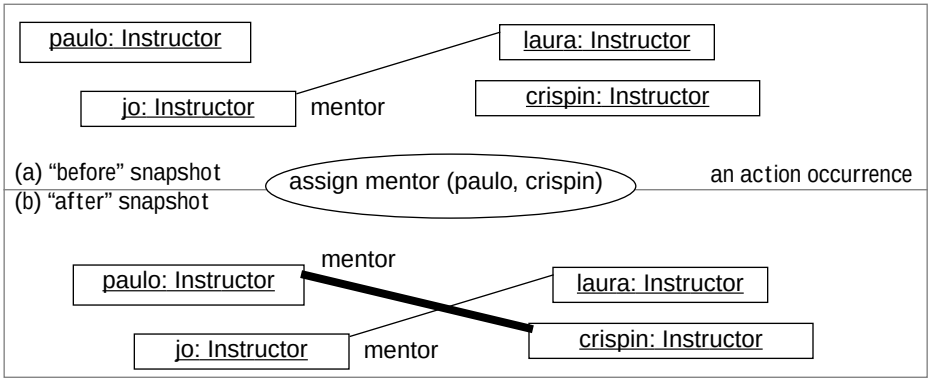
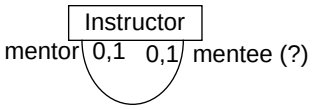


Figure 56: Assigning a Mentor

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.



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 postcondition states what we need; it doesn't say anything about aspects we don't care about (though 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 such as forAll and let clauses (Figure 49 on page 117).

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

4.2.2 Comparing before and after

A postcondition makes an assertion about the states both immediately before and immediately after the action has happened. In every object, it therefore has two complete sets of attribute-values it can refer to. By default, every mention of an attribute refers to the newer version; but you can slide up to the prior state by suffixing it with @pre. So:

subject.mentor@pre	— our subject's old mentor
subject.mentor.mentee@pre	— subject's new mentor's old mentee
subject.mentor@pre.mentee@pre	— subject's old mentor's old mentee
subject.(mentor.mentee)@pre	— same as previous (= subject)
subject.mentor@pre.mentee	— subject's previous mentor's *new* mentee

Each navigation expression is a way of getting from one object to antoher, that normally works all within the same plane of time. @pre can be applied to an expression to make it evaluate in the previous time. But what you get from an expression is the identity of an object, which continues through from one time to another; and unless you keep applying @pre, further expressions will always evaluate in the newer time (Figure 57).

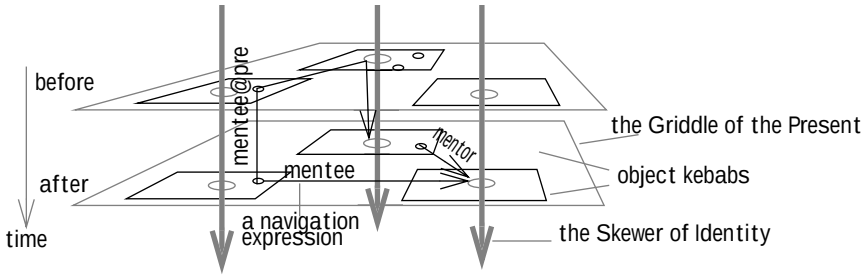


Figure 57: The kebab model of object history

An example:

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

Notice:

- The use of null to represent 'no object' where an association permits none;
- The use of '==>', also written 'implies' — meaning: if ... then ...
- @pre takes you back to the previous value of a changeable attribute , not to the previous state of the object it refers to. Parameters have the same values all the way through, so there is no point in the expression subject@pre.
- In an action spec, we're only dealing with two states, so x@pre@pre is undefined.

Abstract yet precise –> raises pertinent questions. The level of detail here is enough to draw out debate. Doing this example in groups, this is often a point where 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 around in the detail of the program code.

4.2.3 Newly-created objects

A thing that can happen as a consequence of an action is that new objects will be created. The set of these objects has the special name new in a postcondition, and there are some special idioms for using it. After drawing the snapshot in Figure 55 on page 131, 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 properties requested; and
    -- its instructor (as she was previously) was available for the requested period
```

Notice:

- The power of the postcondition, to avoid unnecessary detail. We have not said definitely which instructor should be assigned; nor how one should be chosen from the available ones. We have limited our statement to just the requirements we need, that whoever is chosen, they should have no prior commitment.
- How we have devolved some complexity, by assuming a parameterised attribute available defined with Instructors. We'll have to go back and write that sometime....

4.2.4 Collections

In this superposed pair of snapshots, the new state is shown in bold:

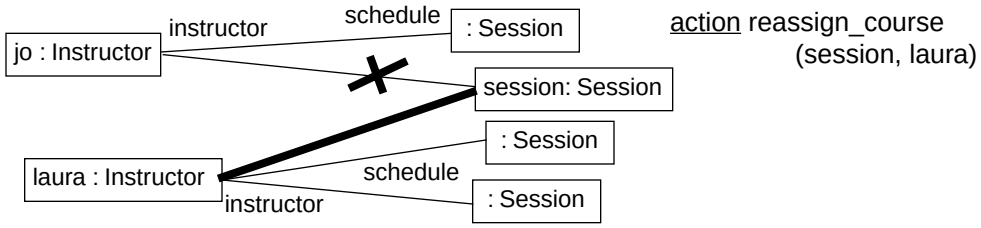


Figure 58: Snapshot for action reassign_course

Many of the associations in a model are of multiple cardinality, and by default represent 'flat' sets. We can use the collection operators (Figure 49 on page 117):

```
action   reassign_course (session : Session, new_inst: Instructor)
post:    --An existing Session is taken off one instructor's schedule, and 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:

- Use of + and – with collections — just 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, merely a comparison between two states.)
- We could perhaps more simply have asserted:

  session.instructor == new_inst

  Since the static model tells us a Session only has one instructor, this might have been adequate. However, an designer might mistake the meaning of this, and make this session the only one the new instructor is assigned to, deleting all his other commitments. So we choose to be more explicit.[1]

1. There is a big issue here concerning 'framing'. In a fully formal spec such as for safety-critical systems, you would have to be more explicit about what objects are left untouched.

4.2.5 Preconditions

Many of the actions we could define only make sense under certain starting conditions, which can be characterised by a precondition. The precondition only deals with one state, so it doesn't have @pre or new.

For example:

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

4.2.6 More precise postconditions — summary

This section has looked at the basics of writing action specifications precisely enough

- to form the basis of a test harness for 'plug-in' components or for a system being built;
- and to make the model explicit enough for discussion of the major business concerns.

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 tabulated in Figure 72 on page 158. Specifying requirements for a complete software system (with a user interface etc) is the topic of Chapter 16 *How to Specify a Component*, p 581; specifying the interface to a substantial component is in Chapter 11 *Components and Connectors*, p 437.)

4.3.1 Objects and Actions are not just about software

Our terms ‘object’ and ‘action’ cover a broad range:

‘Object’ includes whole systems

- ‘Object’ includes not just individual programming-language objects, but also software components, programs, networks, relations, records; as well as hardware, people, organizations --- anything that presents a definable encapsulated behavior to the world around it.

‘Action’ includes dialogues

- ‘Action’ includes not just individual programming-language messages or procedure calls, but also complete dialogues between objects of all kinds. But 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.

Model is of real world...

The diagram in Figure 55 can be seen in two ways. Firstly, it can be a picture of the real-world. The objects represent human instructors, scheduled sessions, etc. 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.

... or of software system

Alternatively, the diagram may be about what a particular object knows about the world outside it (which may, of course, be inaccurate, out of date, or inconsistent with some other object’s view). In particular, it can be a model of the state of a computer system (that is, application, suite, component, etc.) that we intend to build.

Action could be real-world event ...

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 and negotiates the dates and fees for a new session of the course.

... or dialog with system

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.

This could describe system state, state changes, and behaviors

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

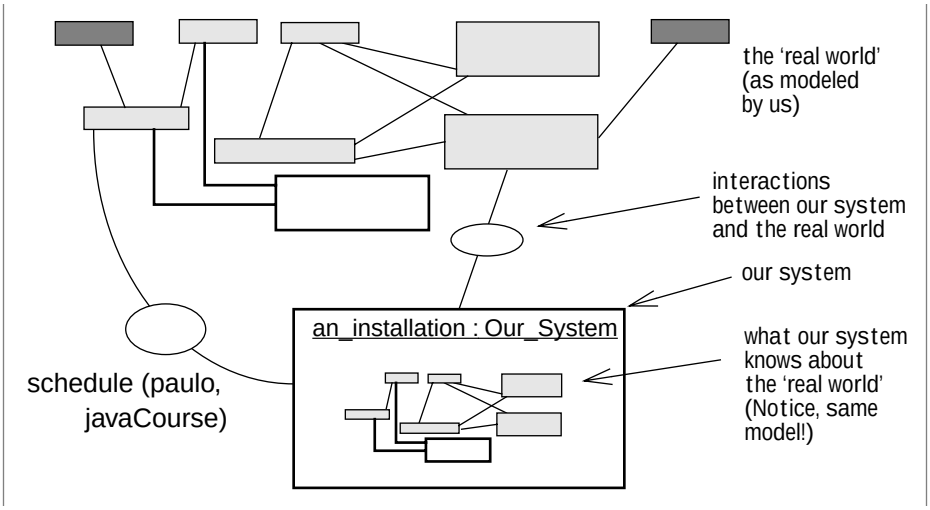


Figure 59: Models of Domain and System

We've already said that these drawings are only models. The meaning of a link in the system model is subject to interpretation (and so must be documented in the Dictionary) just as much as a link in the domain model. A link or attribute only represents a piece of information that can be got out of the system somehow — via a GUI, a query, or indirectly by its effect on other actions — but it will be the job of the designer to decide how it is represented inside.

The attributes do not prescribe an implementation

The same goes for the actions. Schedule in the domain means the achievement of whatever situation is represented by the existence of the session object. Schedule, the system action, means whatever dialogue has to be conducted with the system, and whatever algorithms have to execute within the system, to achieve the state of the code represented in our model by the existence of the session object in the model of the system's internal state. Only once the actual implementation of the object has been defined, can the detailed code to implement this operation be fixed.

Nor do the actions.

4.3.2 Types

Different objects will react in different ways to the same interaction. But rather than describe each object separately, we group objects into *types:* sets of objects that have some (not necessarily all) behavior in common. A type is described by a type specification, that tells what the effects of some actions are, on the internal state of the object; and conversely, what effect the state has on the outcome of actions.

Types group objects by behavior

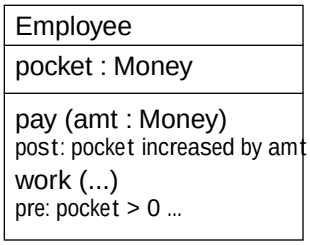
Types are generally partial descriptions. They say "if you do X to one of my members, the resulting response will have this property and that property". But they don't tell you about what will happen if you try doing actions that aren't mentioned; and they don't always tell you everything there is to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

Types can be combined

Types describe roles played for others

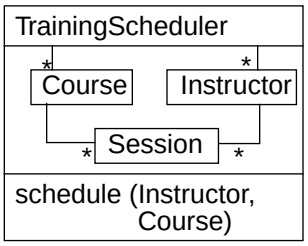
Types correspond to real-world descriptions, as OO concepts always should. An Employee is something that does work when you give it money; a Parent is something that does work and gives you money; a Shopkeeper gives you things when you give it money. These descriptions are all partial, focusing on the behavior that interests certain other objects that interact with them. In object design, we build systems from interacting objects, and so these partial perspectives are crucial. And each object is likely to play several roles, so we need to be able to easily talk about Employee * Parent, i.e. someone who does work when paid (by the appropriate other person), and will also provide money (ditto).

Interesting behaviors need model attributes



Because these descriptions are 'black box' ("I don't care how my parent gets the money, just so longs as s/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 they keep their money; it is just a model, a device to explain the relationship between the actions of payment and request to work.

The type model can be pictorial.

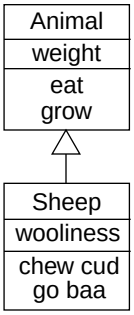


In a complex model, we find a few attributes like pocket insufficient, and tend to use pictorial attributes instead. But the idea is the same: the model is principally there to explain the effects of the actions. And we can use the same principle to describe both small simple objects and large complex 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.

4.3.3 Subtypes and type extensions

A subtype specifies a subset of objects

Because a type-spec is a just description of behavior, an object can be a member of many types. In fact, an object is a member of every type whose specification it conforms to — even if that type specification was written after the object was created. In other words, it can play several roles. 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 — i.e. conform to the behavior specification for that type — may exhibit the additional properties of Sheep.



A subtype is often defined as an extension of a super-type

Putting more into a specification, raising the expectations, reduces the set of objects satisfying 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.

4.3.4 State Charts

State charts are a powerful pictorial tool for envisioning the effects of different sequences of actions, most useful where there are distinct changes of behavior in different states. The states in the chart represent boolean expressions, and transitions between them represent actions. State charts are covered in more detail in Section 4.9.

Postconditions can be drawn on state charts.

4.3.5 Actions and Operations Defined

Action. An action-occurrence is an interaction involving specific participant objects between two particular points in time, bringing about a change of state in some or all of them. An action-spec denotes a set of action occurrences by specifying the effects of an action. 'Action' or 'action-type' means any or all the action-occurrences that conform to a given action-spec; action is sometimes used to refer to an action occurrence in this text, where the meaning is clear.

An action specifies a set of action occurrences

Over a period of time, objects are affected by several actions, and will progress through a series of changes as shown in Figure 60.

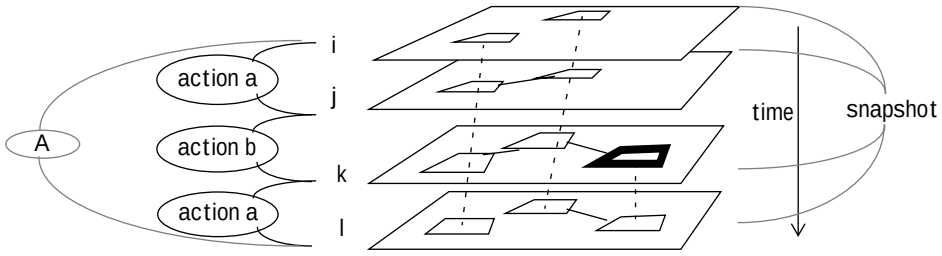


Figure 60: Snapshot history over time

Actions may be composed of finer-grained actions; the sequence of three action occurrences a,b,a in Figure 60 may be modeled as a single more abstract action, A. Function-calls, rendezvous, hardware signals, and messages are all varieties of action. So are use-cases, sessions, remote procedure calls, and complete dialogues.[1]

Actions abstract detailed interactions

An object's history can be envisaged as the column that threads its states in successive snapshots; the history lists all the actions it is involved in, and the resulting states. A type is a set of all the possible histories conforming to a given set of action specs.

Two kinds of Actions

There are two main kinds of actions we are concerned with in this book. One kind — an *localized action*; at the level of implementation code this is often called an *operation* — is an action which an individual object is requested to perform, and responds accordingly. The other kind — a *joint action* — is an action in which a group of objects participate, and is always an abstraction of a more detailed dialog between these par-

Object have individual and joint behaviors

1. Your reading this entire book is itself an action; so is the reading of each chapter, and examination of each footnote. If only we could be sure about the postconditions....

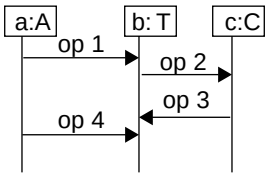
ticipants. *Use-cases* are an example of joint actions, subject to refinement to a particular dialog of interaction between objects. At the level of program code, actions will correspond to operations. This chapter focuses on the former kind of action.

An *operation* is performed by one object

Operation. When describing the behavior of an individual object, our focus is on the individual operations that may be requested of that object, and the effect of each such operation, without consideration of the initiator of the action. You can recognize operations by their focus on a single distinguished object type:

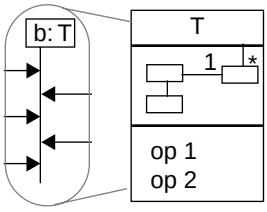
action Type::operationName (...) ...

Program code is at the level of operations



At the level of program code, one object requests another to perform an *operation*; as a result of the operation there is a state change, and some outputs produced. The interactions are illustrated with a *sequence diagram*: individual objects are vertical lines, and each operation-request is shown with an arrow. The objects a,b,c could be objects in the business such as client, seminar company, instructor; or instances of classes in software, like session, calendar, instructor; or large-grained software components like a course-qualifier, room-allocator, and scheduler.

But the idea applies to more abstract objects and services.



For a business entity like a seminar company, the operations it provides as services to outside objects include: scheduleCourse, courseEnquiry, etc. A much smaller object within one of its software systems, a Calendar, provides its own operations like: addEvent and removeEvent. The operations cause state changes in, and outputs from, the object of interest. In both cases, the operations are part of the type specification for that object type. This is the subject of the current chapter.

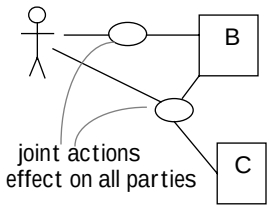
A *joint action involves multiple objects*

Joint Action. To describe behavior and interactions of a group of objects, we focus on what the interactions between multiple objects achieves, and how to specify the effect of those higher-level actions on 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 receiver are designated, but the action effect is still described in terms of all participants.

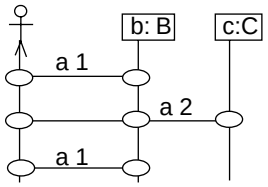
A joint action, or 'use-case', involves multiple participants



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 comprise an abstract purchaseCourse action. This has a net effect on both client and seminar company: not only has the seminar company delivered a service and gained some revenue, but the client has paid some fees and gained knowledge. In software, it may take a sequence of low-level operations via the UIs of multiple applications and databases, to complete a scheduleCourse operation. Such an action is called a *joint action* or *use-case*. We describe its effect on all participat-

ing objects, abstracting individual interactions. All joint actions must be refined eventually into operations to be implemented in any popular OO programming language. This is the subject of the next chapter.

Each occurrence of such an abstracted 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.



4.4 *Example: two implementations*

Two 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': it will be easier to understand what the specification means, if we can see the kinds of implementation that it can have. So let's start the time-honored way: we'll hack the code first, and write up the spec afterwards![1]

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

Both support the same interface

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

addEvent:	add a new event to the current calendar schedule
isFree:	determine if an instructor is free on given dates
removeEvent:	delete an existing event from the schedule
calendarFor:	return the scheduled events for a particular instructor. It is returned as an Enumeration i.e. a small object that has operations to step through the collection until the end.

4.4.1 Calendar Implementation A

The first implementation maintains events by instructor

This implementation keeps a separate un-ordered vector of events for each instructor in a hashtable, keyed by the instructor. Its internal interactions are described in the sequence diagram in Figure 61, 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 existed. The new event is added to this vector, and the hashtable updated. The Java code for this design is listed below.

1. If this offends your sense of decency, please skip straight on to the next section. Should you wish to avert your eyes from the bare code on display, it's three page-turns ahead.

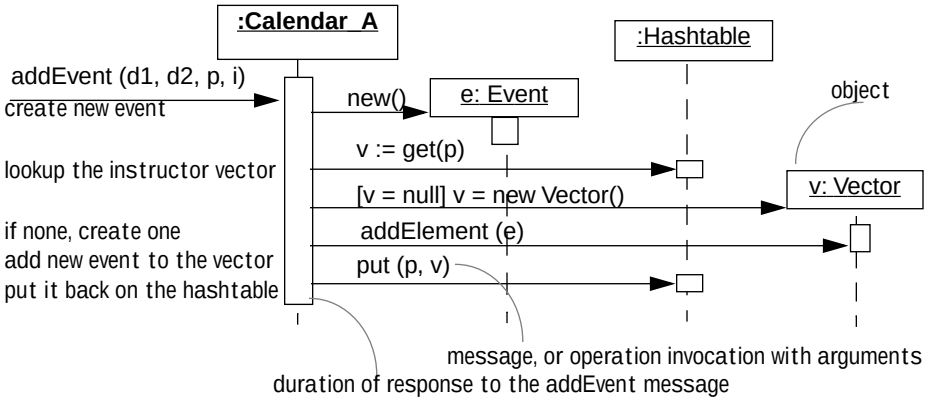


Figure 61: Internal Design interactions of Calendar A

```
import java.util.*;

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

    // provided no schedule conflict, this creates and records new event
    public Event addEvent (Dated1,Date d2, Instructorp, 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 thesedates
    // i.e.does 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(asan enumeration)
    public Enumeration calendarFor (Instructor i) {
        return ((Vector) instructorSchedule.get(i)).elements();
    }
}

// internal details irrelevant here
class Instructor { }

// represents one session
// Just two public operation: 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 (Dated1,Date d2, Instructorw, Object i) {
        from = d1;
        to = d2;
        who = w;
        info = i;
    }

    // doesthis event overlap the given dates?
    boolean overlaps (Date d1, Dated2){
        return false;
    }

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

Figure 62: Java code for Calendar implementation A

4.4.2 Calendar Implementation B

The second implementation maintains events by date range

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. (The art of object oriented programming is to delegate as much as possible to another designer:-) The internal interactions for this calendar implementation are shown in Figure 63, and the Java code is listed below.

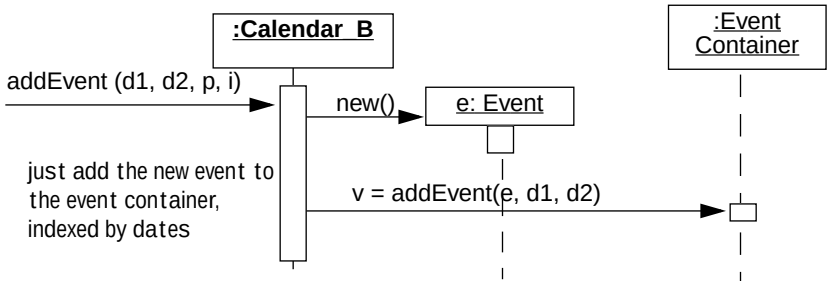


Figure 63: Internal Design interactions of Calendar B

```
import java.util.*;

// Organizes eventsby 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 thosedates?
    // i.e.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(asan enumeration)
    public Enumeration calendarFor (Instructor i) {
        // implementation not shown
        // presumably less efficient, sincetuned 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 explicitlyrecorded; containermaintains 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);
}
```

Figure 64: Java code for Calendar implementation B

4.5 *Example: Specification covering all Implementations*

A client could use either implementation of calendar: both implement the same *type*. We need to describe this type so that a client can use either implementation based solely on the type specification, admitting both implementations as correct implementations of that type, yet ruling out incorrect implementations.

Both implementations meet client needs

Ordinarily, we wouldn't bother with a very precise specification of such a trivial thing: but specification *is* important for the cases where there are several potential implementations: for example where many components can be 'plugged in' to another. Specification is important in the world of reusable pieces, even for small ones.

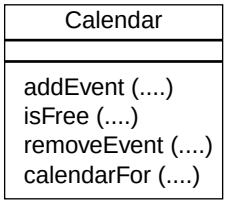
4.5.1 What a Client should (and should not) know

We first document this type by listing the operations it provides. Naturally, the client needs to understand a good bit more than just the list of operation names and signatures: they need a specification of the provided behavior.

Client must understand inputs, outputs, assumptions, and guarantees

- What are the inputs and outputs?

  Calendar operations expect inputs of type Date, Object, Instructor, and Event. The client must provide arguments of the appropriate type i.e. objects that implement the stated types.

  The addEvent operation returns an Event object, which provides just one public operation to the client: delete(). There are no other externally visible outputs.

- What assumptions is the implementer permitted to make?

  These could be many. He expects specific input parameter types; in the case of Java, these assumptions happen to be checked by the compiler. The add and isFree operations could assume that d1 < d2; they may not check this explicitly, and may behave incorrectly if the dates are improper. For a removeEvent operation, the calendar may assume that the input event is one that already exists within this calendar. A implementation may or may not support overlapping events for an instructor.

- What else can the client rely upon for the outcome of the operation?

  Are the events returned by calendarFor ordered by increasing dates? What does a delete() on an event object do? Is a separate call to removeEvent on the corresponding calendar required? If so, which one should be done first?

Besides inputs and outputs, the client needs an abstract model of the "state" of the calendar independent of a specific implementation, even if that state is not directly accessible. Whenever addEvent has been called, there is indisputably a new event in the calendar, even though the way in which it is represented differs from one to the other. isFree must be based on the events currently in that calendar; and delete removes it from the calendar — even though the implementations differ. We already know how to build such a model using attributes.

Client needs an abstract model of calendar state — attributes

Internal details should be hidden

Figure 61 and Figure 63 illustrate that the internal representation and interactions differ widely between the two implementations. Our description of object behavior must adequately specify operations, input parameters, change of state of the object, and returned values or other outputs produced, without getting into irrelevant “internal” interactions.

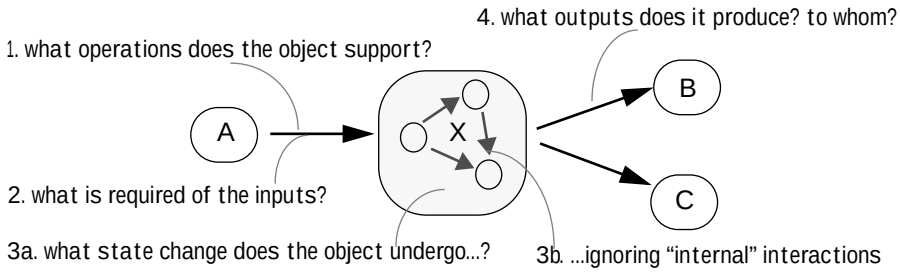


Figure 65: Object behavior description needs a clear boundary

Omit objects the client should not be aware of.

In a specification, we only include interactions with objects that *the client should be aware of*[1]. In this case, the vector, hashtable, and event container are entirely internal to the calendar, and a client need not even know they exist; so we summarize interactions with them into a net change on abstract attributes that characterize either implementation. What we are really describing is not just the single calendar instance, but its grouping with its internal supporting objects, as shown in Figure 66. The same rea-

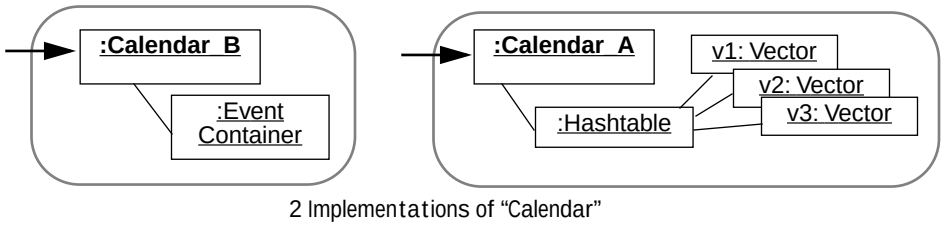


Figure 66: External view hides internal objects and interactions

soning applies at the business level; the seminar company can be considered a single object from the outside, and all its internal roles and interactions are only visible in an internal view.

4.5.2 From Attributes to Operation Specification

A simple sequence of steps to write a type spec

Here is a sequence of steps to arrive at a precise type specification of the Calendar. [2]

We will assume that the calendar may return output values to the client, and that all other interactions are internal details that should not be known to the client.

1. **List operations** — addEvent, isFree, removeEvent, calendarFor

1. Although intuitively reasonably, this can be tricky due to *“aliasing”* i.e. two different paths in an object graph leading to a shared object.
2. Thanks to Larry Wall for pulling apart the steps involved

2. **Informal operation descriptions** of each one:

Start informal

addEvent	a new event is created with the properties provided, and added to the calendar schedule
isFree	returns true if the instructor is free in the date-range provided
removeEvent	the event is removed 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 (Figure 68 on page 152), 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 inputs and outputs.** At the level of individual operations in code these are usually straightforward, perhaps already known.

Identify information exchanged

addEvent	(date1, date2, instructor, info): Event
isFree	(instructor): Boolean
removeEvent	(event)
calendarFor	(instructor): Enumeration

4. **Snapshots.** 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, ✕ for objects or links that do not exist in the "after" snapshot, and name the input and output parameters to the action occurrence consistently with the snapshots.

Find underlying attribute terms

After an addEvent, the highlighted objects and links are created; the output is e3:

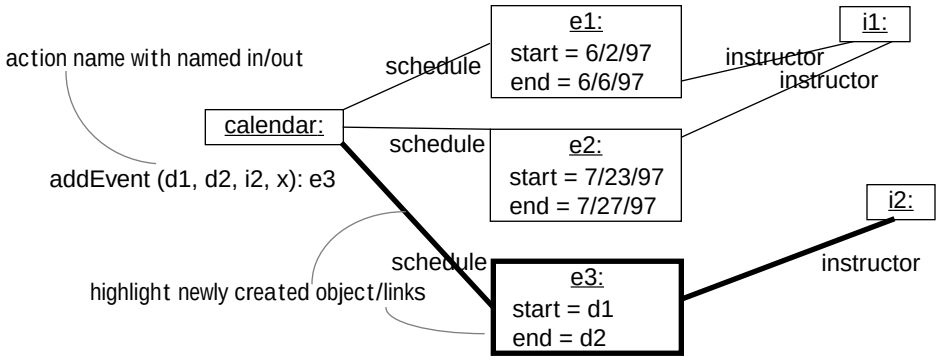


Figure 67: A snapshot-pair for an action occurrence

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 like isFree, check there is some way the information could be extracted from every snapshot.

5. **Static type diagram** of the object being specified. Draw a type diagram that generalises all the snapshots you have drawn. (That is, they are all valid instances of it.) The attributes mentioned by each operation are listed below, and summarized in Figure 68.[1]

Attributes form the basis for specifying multiple operations

addEvent	Calendar schedule represents events currently in the calendar. Each event has attributes instructor and start, 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 needed; schedule on calendar suffices.
calendarFor	Use a schedule attribute on instructor; note that the externally provided operation is different from the attribute that models the necessary state.

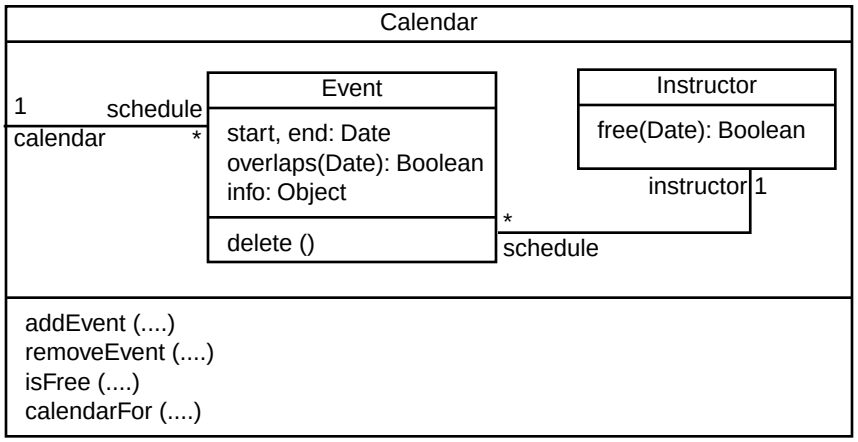


Figure 68: Type model attributes will be used to specify operations

Write attribute types and invariants

6. **Invariants.** Consider whether any invariants should be written with the model. More will become apparent as you work through it. The choice of attributes and associations may change too, because their purpose is to make the actions easy to specify.

```
-- 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 )
```

This requires an overlaps attribute on event.

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

Formalize the operation specs

7. **Specify operations.** Make the operation specs more precise. This step may be followed to a greater or lesser extent in different situations, depending on the project. The precision helps uncover gaps in the model, and also defines test specs for the implementation; it does take some effort, and is greatly aided by a decent

1. Not every tool can draw one type inside another. An alternative is given in §4.13, p.192

tool. At the very least we should improve the informal specifications, as in step §9, p.154.

The notation for referring to attributes was introduced in the last chapter; additional constructs for operation specifications are used here and will be explained in detail in the following section.

```
-- 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]
```

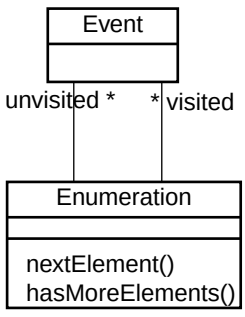
A function is an operation that may return a result.

```
-- 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. **Parameter models.** Describe (by a type model) any input and output parameter types as well, and their attributes and operations, to the extent the client and implementor need to understand and agree on these.

Also describe other related types

The Enumeration returned by calendarFor could also be modeled explicitly. It provides two operations, informally specified below. These 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. The client clearly needs to know the effect of this operation e.g. 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)
```

Because this is equivalent to removeEvent which we have already specified, we could also have written it more concisely, as discussed in Section 4.7.6:

```
-- 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@pre.removeEvent (self)
```

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

Don't forget the complementary narrative!

9. **Write a Dictionary** and improve your informal specifications. Even if the invariants and operation specifications will not be formalized, you can now concisely define the terminology of types and attributes, and consequently the operation requirements. Contrast the updated informal operation specifications below with the ones we started out with.

Instructor	the person assigned to a scheduled event
schedule	the set of events the instructor is currently scheduled for
free	an instructor is free on a date means that no event on his schedule overlaps 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	an event overlaps a date means that date lies within (inclusive) the range of dates of the event

Figure 69: A Dictionary of Terms

```
-- 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 the range of dates
    post:    -- a new event is on the calendar schedule 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:   -- the return is true if that 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 re-factoring e.g. we can remove the redundant constraint of d1<d2 by introducing a DateRange type, with attributes start,end dates, overlaps(date), and an invariant on these attributes. Multiple places in our design will become simplified as a result.

Re-factor to improve things.

4.5.3 The resulting object type specification

So we have specified our Calendar requirements in such a way that it can be fulfilled by either implementation — or indeed any other that behaves suitably. The actions have been listed and we have described the effect of each on our model of the state of the Calendar. The main products of the specification task are shown in Figure 70.

The next section details how actions are defined.

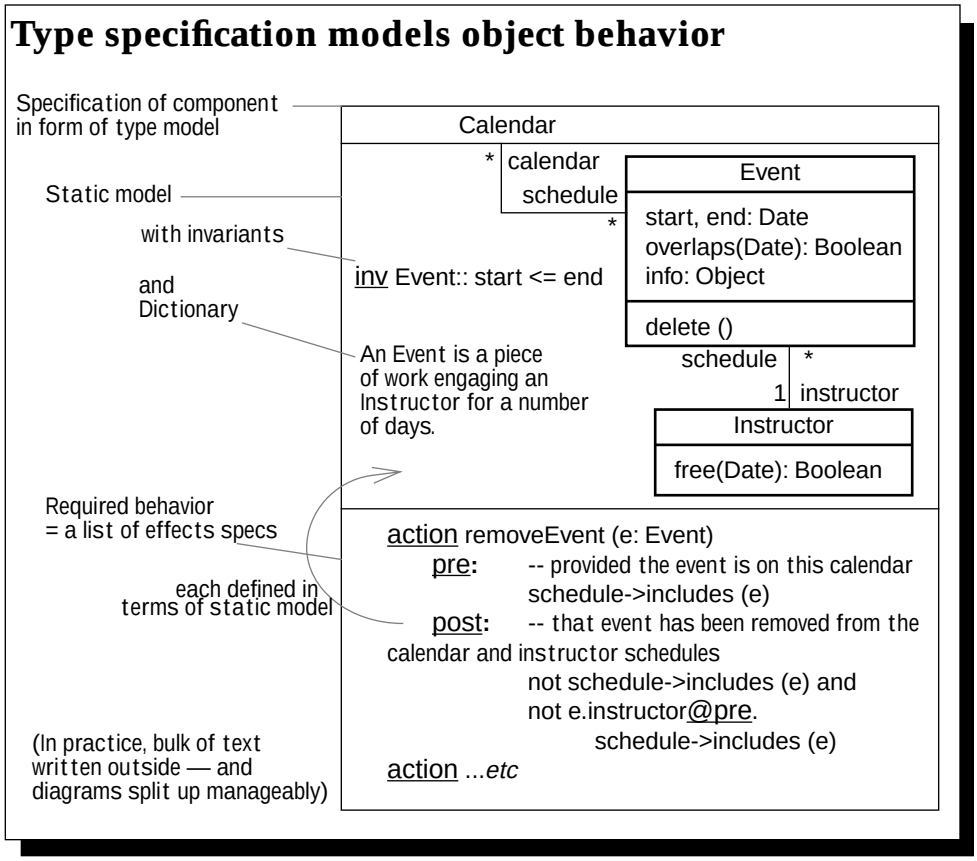


Figure 70: The product of Behavior Modeling is an object Type Spec

4.6 *Specifying actions*

Attributes have different values at different times

If you imagine objects and their links as a diagram on a two-dimensional surface, then you have to extend it to three dimensions to understand what an operation specification is about. Every object has any number of previous states, as we saw in Figure 60. Each operation request to a "receiver" object starts with an initial snapshot of the

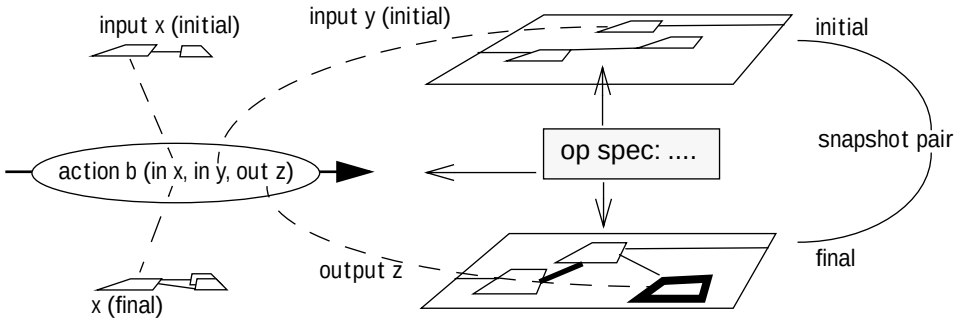


Figure 71: Operation spec relates parameters and initial / final attributes

receiver and of all the input parameters to that request, and ends with a final snapshot with changed attributes, newly created objects, and some generated outputs. Each attribute has a *initial* and *final* value. The operation specification is a relation between the inputs, initial values, final values, and outputs, written as a boolean expression i.e. an operation specification specifies is a *test* condition*;* any implementation will either pass or fail that test.

An operation spec focuses on *before* and *after*

An operation specification describes the effect of invoking that operation, using a pair of *pre-condition/post-condition* expressions called an *effect spec*. Within a single effect spec, two moments are singled out — immediately *before* the operation, and immediately *after*.

```
action Calendar::addEvent (d1: Date, d2: Date, i: Instructor, o: Object) : Event
    pre:    -- provided dates are ordered, and there is no overlap caused by the new event
            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]
```

An alternate syntax puts the receiver type name with the signature; it is more consistent withthe more general ideas we'll meet later, but less conventional:

```
action addEvent <=
        -- the action 'addEvent" is specified, among other things, by the following effects clause:
    effect
        Calendar:: ( d1: Date, d2:Date, i:Instructor, o:Object)
                -- for any combination of a calendar, called "self" in the assertions,
                -- together with two Dates and an instructor and something else ;
        pre: ....  -- and provided they all meet the conditions here stipulated ;
                -- then we reckon that when an action conformig to this spec happens,
        post: ...  -- this should be the outcome.
```

Action specs

Defines a requirement on one or more actions. 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 by the end of the operation.

• May also define a result type and a receiver type.

pre: *condition*

Precondition 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 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 refered to using @pre.

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

rely: *condition*

If the rely clause ever becomes false during the execution of the action, the specification no longer applies.

guarantee: *cond*

The action will maintain this true while executing.

Pre, post, rely and guarantee conditions are called 'assertions'. A further two assertions appear as part of 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.

Figure 72: Effect spec clauses

Special terms in a postcondition

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

x@pre	The value of x's prior state. (There is no need to use them in a pre-condition.) E.g. moving rooms: jo.room@pre — jo's old room jo.room@pre.isDirty@pre — previous state of jo's old room jo.room@pre.isEmpty — current state of jo's old room jo.room.isEmpty@pre —previous state of jo's new room
new	The set of all objects that exist in the later state that did not exist in the earlier. T*new=(T – T@pre). Common usages with new: Egg*new — all new Eggs Egg*new [size>5] — all new Eggs satisfying the filter Egg.new — the only new Egg Egg.new [size>5] — the only new Egg, and its size>5 Egg[size>5].new — the only new (Egg whose size>5) (The .new notation denotes the new object; but as a side condition, specifies that there are no new others.)
[[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. E.g.: action square_root (x : int) post: x = result * result ... y = square_root (64); // y == 8

Figure 73: Postconditions

Op-spec generalizes snapshot-pairs

An operation specification generalizes all occurrences of that operation i.e. it should hold true of every snapshot pair (Figure 74), much like a type model generalizes all snapshots in Figure 68 on page 152.

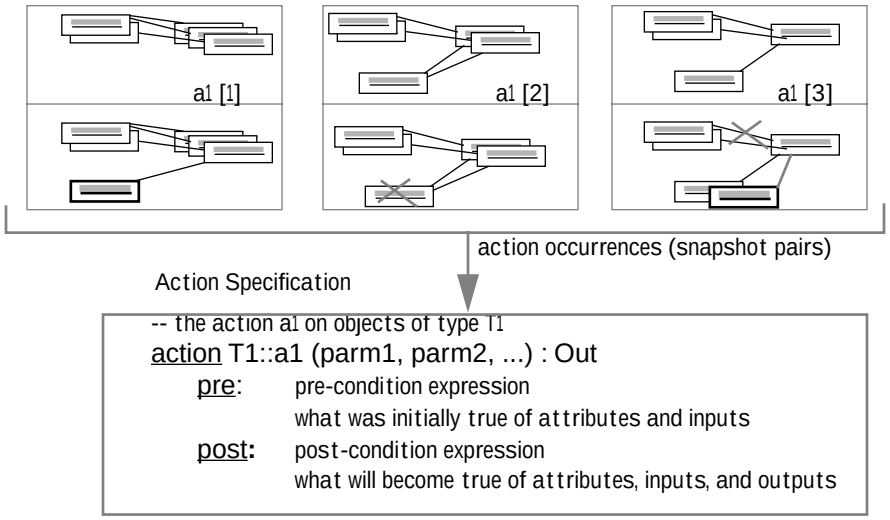


Figure 74: Operation specification generalizes snapshot pairs

An op-spec prohibits certain snapshot-pairs

Given a type T with operation M whose operation spec has a pre-condition P and post-condition 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 before that occurrence, then Q should have become true after that invocation.

4.7.1 An Action Spec is not an Implementation

Op-spec is a boolean, not an algorithm

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
          (with a supporting definition of path, and of the length of a path)

Any implementation must meet the spec

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 through an implementation, where the test data met the pre-conditions of the specification, we would expect the post-condition to be satisfied; if not, we have found some bug either in the implementation or the specification.

The spec is like a test

A good operation specification is much like a test specification. With a little infrastructure support — e.g. 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 often be mapped to test code that is executable at run-time, at least during testing and / or debugging.

Though some specs are easier to test than others!

The operation specs will often map directly to tests. Thus, the spec for squareRoot above easily translates into test code; so does schedule_course, once we write some query functions to determine attributes related to instructor qualification and availability in terms of the concrete implementation. Other specifications may need to be re-factored a bit to be tested effectively. A naive usage of takeShortestPath as a test specification would require generating all possible paths to show that the computed path is the shortest; not a very practical test strategy!

4.7.2 Parameter types

Parameter types are part of pre / post

Parameter types are an implicit part of pre / post-conditions. Our spec of squareRoot could be re-written 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

A shorthand for parameter types

We permit a shorthand for parameter types. A parameter which is not explicitly typed has a name which is a lower-case version of its type name. The spec below implicitly types all three parameters:

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

The effect of an operation can be specified with an explicit pre / post pair of conditions, or with a single postcondition. The main difference is that within the expression starting with pre: all references are implicitly to the initial values of attributes; within an effect clause we have to explicitly indicate initial values using x@pre. These two are 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 simply have nothing to say about the outcome.

Each is useful in different situations

The pre/post style is well suited to documenting an operation at the implementation level as a single pre / post specification in the form of a "contract": the caller is responsible for only invoking this operation when the pre-condition is true; the implementor can assume the pre-condition is satisfied, and must then guarantee the post-condition. If a pre / post pair is written, the post-condition should cover all those cases that are permitted by the pre-condition, otherwise the specification is not well formed. By convention, if only the post-condition is written, we treat it like an implicit pre-condition that is derived (not always easily) from the post-condition.

The single post form is sometimes more convenient. It also lets us write specifications that do not have to be of the form P@pre implies Q. Chapter 13 will describe how this is useful to factor out parts of the specification of a complex action; however, it then takes more effort to extract an explicit pre-condition.

4.7.3 Partial action specs

Pre and post-conditions are always paired

An operation post-condition is always paired with a corresponding pre-condition. Thus, the stated post-condition of squareRoot should be guaranteed provided the corresponding pre-condition was true when it was invoked. Similarly, the stated outcome of schedule_course is guaranteed provided that there was an available and qualified instructor for this request.

Some conditions may be left unspecified in a spec

What should squareRoot do in the case of a negative input? Or schedule_course in the case when a qualified instructor was not available? Our specs, as written, do not cover those other conditions; we leave those behaviors unspecified. If we said nothing further about these operations, then the implementations could ignore those other conditions completely.

But, an operation may be constrained by multiple specs

But, an operation can have multiple specifications; this is very common in higher-level requirements where we want to separately specify different aspects of an operation. Different aspects of an operation like schedule_course — scheduling policies, qualification criteria, associated production of course materials, performance requirements — can be specified separately. At the very least, these may appear in separate sections of a document. This allows us to factor a specification into more coherent bits, and makes it easier to separate exceptions and variations. The implementor of an

operation must meet the conjunction of all specifications for that operation[1] — every one of the individual guarantees must hold independently of the others. Thus, we could add the following specification to the squareRoot operation:

```
action squareRoot (in x: Real, out y: Real)
    pre:    x < 0
    post:   y = -1.0
```

A spec may also constrain outcomes, without fully determining them

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 exclued one result, we only have to 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, just that there should be no other path with a shorter length.

Multiple specs vs. a single — example

A operation may be constrained by multiple specifications, each with its own pre / post pair (or, its own effects clause). Alternately, the multiple specs can be combined into a single specification with a more complex pre-condition and post-condition. 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)
    pre:    Provided there is an instructor qualified for this course
            who is free on this date, for the length of the course.
    post:   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)
    pre:    Provided the client is above some loyalty threshold
    post:   The client is sent a certificate for a free course
```

Any reasonable tool should related 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)
    pre:    instructor available or client is above loyalty threshold
    post:   (instructor available)@pre implies single new session for that course, ...
            and (client above threshold)@pre implies client is sent free certificate ...
```

The single pre / post form becomes a bit awkward and redundant in the combined specification. It must deal with multiple conditional outcomes with overlapping conditions, which is done by **and**'ed clauses in the post-condition. In addition, the pre-condition has to specify under what conditions the post-condition is applicable. It is simpler to instead write out the multiple pre / post pairs in one action spec:

```
action Scheduler::schedule_course (client, course, date)
    (   pre:    instructor available
```

1. Chapter 13 describes other ways to *join* two operation specifications

```
        post:   single new session for that course, ...
    ) and
    (   pre:    client is above loyalty threshold
        post:   single is sent free certificate ...
    )
```

The equivalent effect form is very similar, except it requires explicit @pre:

```
action Scheduler::schedule_course (client, course, date)
    post:   (instructor available)@pre implies single new session for that course, ... and
            (client above threshold)@pre implies client is sent certificate ...
```

Pre / post only constrains occurrences where pre was true.

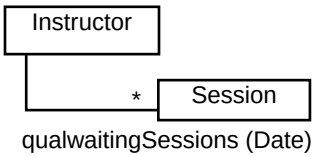
When we write a pre / post pair, the interpretation is simply that any occurrence of the action in which the pre-condition was true must result in the post-condition becoming true. That particular spec does not state the outcome if the pre-condition was not true, hence it does not constrain such occurrences. When we write an effect clause, every occurrence of the operation must satisfy the effects clause; which, so far, has simply reflected the pre / post form.

4.7.4 "Convenience" attributes simplify specs

Introduce new attributes to simplify specs

Since invariants can simplify action specifications, and attributes themselves simply represent a precise terminology for use in action specs, we can sometimes greatly simplify the specs by introducing suitable attributes and defining them using invariants.

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



Finding this specification somewhat complex, we seek underlying terms to simplify it. The pre-condition seems to refer to a set of sessions which need an instructor for a particular range of dates — in this case, the dates of the session being cancelled. Why not introduce a parameterized attribute qualWaitingSessions(dates) and simplify the operation spec?

```
inv Instructor:: qualWaitingSessions (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 there were any sessions waiting for that instructor on those dates
        s.instructor.qualWaitingSessions (s.dates)->notEmpty & s.confirmed
post:   -- then the instructor is assigned to one such session
        ....
```

Judiciously chosen auxiliary attributes such as this can be very effective in simplifying specs, by introducing precisely defined terms that express the requirement in a natural way, in terms close to what a client might use (despite the formal syntax).

4.7.5 Effects

Effect: local function with @pre

Another kind of convenience function is called an "effect": this is a function that can use @pre, and so can be used to factor out the parts that are common between different action specs.

We want to define effects regardless of the action

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; however, we would not list these as actions. Instead, we define these as *named effects*; referring to them by name is exactly equivalent to writing out their specifications directly.

```
-- 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]

-- saying that a schedule_unfirmed_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]
```

Named effects can be used to specify actions

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, since the postcondition covers all cases
    post:   -- either a confirmed course has been scheduled
            schedule_confirmed_course (course, date)
            -- or a unconfirmed course has been scheduled
    or      schedule_unconfirmed_course (course, date)
```

Pre=>post vs pre&post

Esoteric topic

In the example above, 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. That 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 in a style that 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 was no available instructor initially ...
post    not (instructorAvailable@pre (course, date))
            -- then an unconfirmed session is created
            => Session.new [unconfirmed]
```

And the composition:

```
-- You must always confirm or unconfirm a course depending on instructor availability:
action Scheduler::schedule_course (course, date)
    pre:    true -- no precondition, since the postcondition covers all cases
    post:   -- either a confirmed course has been scheduled
            when_instructor_available_confirm (course, date)
            -- or a 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.[1] Experience suggests:

- Write effects in a (pre => post) style when you wish to ensure that there is no getting out of the contract, that if the precondition is true, then the postcondition will be met. Then combine them into actions using AND.

  This is generally better when combining several separately-defined requirements — for example when building an component that conforms to the interfaces expected by several different clients.

- Write effects 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 building a specification model from different parts within the same document. These effects have to be combined with open eyes: none of them makes any guarantees that the outcome it describes will be met.

1. This was the biggest difference between the formal specification languages Z and VDM.

4.7.6 Quoting Action Specs within effects clauses

Each action introduces a named effect

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..." i.e. re-use the effect specification of another action. Specifications can be quoted within others' postconditions (like calling subroutines in code). Suppose you want to say "if I am told about a phone number change, I alter my address book":

```
action Person::notify_phone_change (who:Person, n:PhoneNum)
    pre:        who : friends              — this spec only about people I know
    post:       addressBook[who].number = n
```

[[action]] uses the effect of one action inside another

Now "if I move house, I move my furniture to the new address, and get my new phone number entered in each of my friends' address books".

```
action Person::moveHouse (newAddr, newNum:PhoneNum)
    post:       furniture.location = newAddr      —  shift my chattels
        &  (    f:friends,                 — for each of my friends, call it f,
                [[ f.notify_phone_change(self, newNum) ]]
                                  — get f to do whatever they do with phone changes
           )
```

It does not require the action to be invoked; just its effect to be achieved

The 'quoting' syntax [[...]] is a predicate, possibly involving initial and final states, that says "this action achieves whatever notify_phone_change would have achieved, with these parameters", but not necessarily by invoking that action. It doesn't say how this must be achieved. The obvious thing, of course, is invoke the notify_phone_change operation on each friend; but we leave the decision open to our designer, who might know of another way to achieve the same effect.

If you want to 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 & self substitutions. These two variations mean the same:

```
action Person ::moveHouse (newAddr, newNum:PhoneNum)
    post:       furniture.location = newAddr
        &  (    f : friends,          — for each of my friends,
                self : f.friends          — if I am one of their friends,...
                ⇒  f.addressBook[self].number = newNum )

action Person::moveHouse (newAddr, newNum:PhoneNum)
    post:       furniture.location = newAddr      —  shift my chattels
        &  (    f: friends,                — for each of my friends, call it f,
                [[ f.notify_phone_change(self, newNum) ]]
                                  — get the effect of f's notify phone change
           )
```

[[–> action]] actually invokes the action

If you decide that a part of 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: [[–> f.notify_phone_change(...)]]. This alters the postcondition to mean "the notify_phone_change operation has been performed on each friend". The end result is no different, but we're now pinning down how to achieve it.

4.8 *Specifying component types*

Now let's put together the actions and the static model. We are writing a type to specify an interface to an object; it may cover all the operations that object can perform, or just the ones used by a particular client.

A type is a specification of the response an object has to a set of operations. The operation specs are therefore the primary purpose of the spec: without them, it doesn't mean much. The operation specs share a single static model. A simple object has just one or two attributes; a complex one has a whole diagram full of them (Figure 75). (Of course, it is usually too big to fit in one box like this! See Chapter 6 *Effective Documentation,* p 237.)

4.8.1 Variables an action spec may use

An action spec tells about the outcome of a named action happening to a set of parameter-objects and (for localised actions) a receiver.

The spec of an action may use:

- self, refering to the receiver object, whose type is written
- the parameter names, refering to those objects
- a result object
- the attributes of the type of self, drawn as associations inside or outside the box
- the attributes of the parameters and the result

4.8.2 Actions need not duplicate Invariants

An example type model with invariants

Operation specifications can be simplified by taking advantage of constraints in the type model. Consider the type model of Scheduler in Figure 75 with these invariants:

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

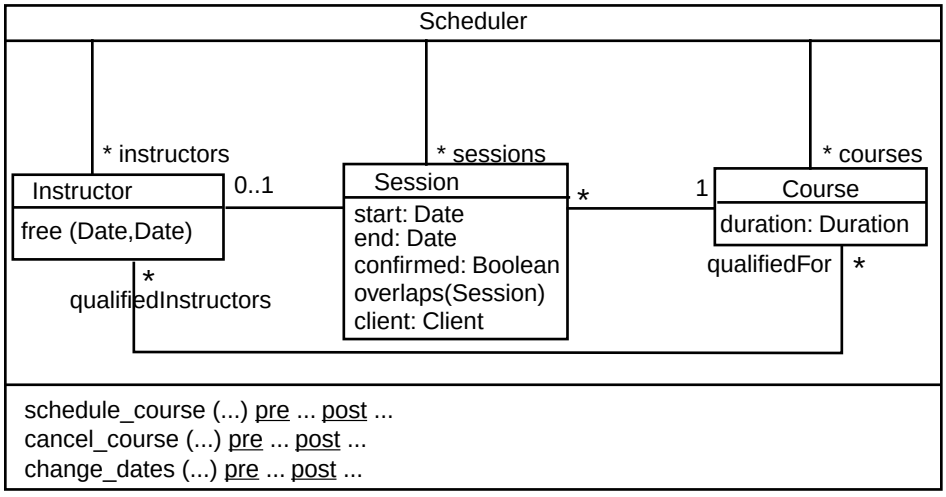


Figure 75: Scheduler type model with invariants

Let us try to define some operations against this model.

We can write lots of details in a spec

```
-- 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 unnecessary?

Some of these are implied by param types and invariants

1. One cannot change the dates of a session from the past. *Necessary.*
2. 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. *Unnecessary.*
3. The post-condition refers to start + course.duration, which only makes sense if duration was a valid Duration. Once again, the type model already stipulates that every course has a duration attribute which is a valid Duration. *Unnecessary.*
4. This is the essential part of the post-condition. *Necessary.*
5. It seems reasonable that the end-date of the course is also changed. However, the relationship between the start / end dates and is 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. *Unnecessary.*

Each of the unnecessary items is already implied by an invariant. The definitions of types themselves introduce invariants, such as the multiplicity or types of attributes. The unnecessary parts would not be incorrect, just redundant. This leaves us with a much simpler operation specification:

Invariants simplifies the spec

```
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
            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 post-condition are redundant, and the last line in this specification can be omitted. Other than that, we don't bother to give any further requirement here: so at present we're allowing different implementations to choose among the available qualified instructors in different ways.

Advanced Topic

4.8.3 Redundant specifications can be useful

It can be useful to write redundant specifications

We have seen how certain elements of an operation specification are implied by the invariants, hence become redundant in the op-specs themselves. Writing them would not be incorrect; just redundant. It can still be useful to write them down; note the change_dates example above. 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.

Distinguished from necessary specs using /pre, /post, /inv

Just as we can introduce *derived* attributes — those marked with a "/" that could be omitted since they defined entirely in terms of other attributes — we can also have introduce *derived specifications* — significant properties which we claim would automatically be true of any correct implementation of the non-derived 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: atttribute 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 only write those claims we consider important to explicitly point out.

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

4.8.4 Meaning of invariants

Invariants have a special meaning with actions

We introduced invariants as constraints on legal snapshots, in Section 3.5, “Static Invariants,” on page 115. We have described actions in terms of the relationship between two snapshots, before and after the action; and now understand an object in terms of its history of action occurrences and snapshots. What do invariants mean in this temporal view of changing objects?

Invariant holds before and after every action in some range

An invariant is implicitly conjoined (**and**’ed) to both the precondition and the post-condition 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.

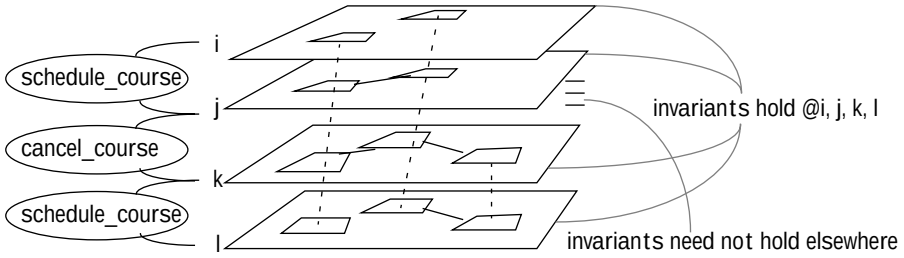


Figure 76: Invariants hold before and after a range of actions

Thus,

```
action      FlightMaster:: assign_pilot (p:Pilot, f.Flight)
   pre:     -- the pilot must be at the departure location on the flight date
            p.locationOn (f.date) = f.departureLocation
   post:    f.pilot = p     -- pilot has been assigned to flight
```

combines with inv Flight:: pilot <> copilot (pilot and co-pilot cannot be the same) to form a complete operation specification:

```
action      FlightMaster:: assign_pilot (p:Pilot, f.Flight)
   pre:     f.pilot <> f.copilot     -- true of all flights, one of which we focus on
      &     p.locationOn (f.date) = f.departureLocation
   post:    f.pilot <> f.copilot
      &     f.pilot = p
```

There may be actions outside its range where invariants are “bent”

However, the private operations of any implementation may see situations in which the invariant is ‘untrue’. For example, suppose the user assigns a pilot to a flight for which she is already copilot. One acceptable implementation would be to deallocate the copilot duty — but in the actual code, this might happen after assigning as pilot. Thus the invariant is temporarily broken between *internal* actions in the code.

4.8.5 Effect Invariants

Advanced Topic

Some effects are invariant across all actions

If some actions share common effects — i.e. changes between before and after states — we can specify a named effect and refer to it by name in the post-conditions of those actions. In contrast, an ordinary invariant is required to hold before and after the actions in its range; i.e. it is a *static* invariant, true of every applicable snapshot.

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. We could write a named effect, but we would have to explicitly reference it in every operation:

```
effect invoke () post: count += 1
```

These are effect invariants

Recalling that static invariants are implicitly applied to all specs, we introduce an *effect invariant* — an effect that is invariant across all actions in its range, and is implicitly **and**'ed to the post-conditions 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 invoke post: count += 1
```

An effect invariant is **and**'ed to the post-condition 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
```

We can define rules that span all actions

By using conditions in the post-conditions of an effect invariant, we can describe effects that apply selectively to any action that meets the condition. For example, to keep a count of all actions that either create or delete an event on the schedule:

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

Advanced Topic

4.8.6 Context of an invariant

Context of invariant can be type, class, collaboration

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

Invariants are only true outside the interface

Any object claimed to conform to the type should always make it look to clients as if the invariant is always true. While the client is waiting for any operation to be accomplished, the invariant can be broken behind the interface that the type describes; but it must always be true again once the operation is complete. Behind that interface, are components of the design that have their own nested contexts, and invariants that govern them.

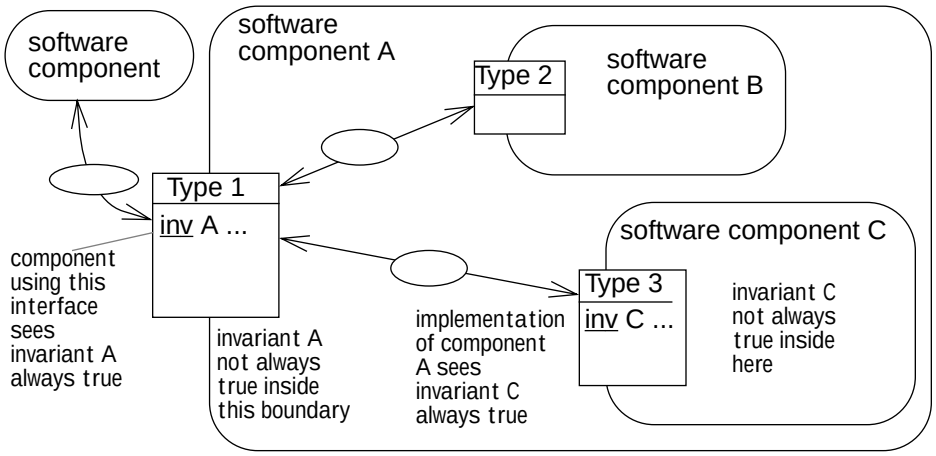


Figure 77: Invariants outside their contexts

Invariants should refer to the attributes of the type in hand

It is possible to write invariants that cannot be satisfied by an implementor — just like any specification. In your spec of a Sludge vending machine, you may write an invariant that the weather is always clement in Northern England; but I cannot deliver such a device.[1] But if you write an effect invariant that the machine's cashbox will always fill as the can stack decreases, then I believe I can do that.

Invariants need encapsulation

In order to achieve it, I must employ certain techniques in my design. For example, if I forget to include a stout metal case around the outside, then 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 in software: the developer of an alphabetically sorted list of customers cannot guarantee that the list will remain sorted, if other designers' code is able directly to update the customers' names. You can only guarantee what you have control over.

Collaborations are fundamental to design

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

1. Although, now I come to think of it, it depends what's in the cans ....

4.9 *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 alternate view of the spec.

4.9.1 States as Attributes and Invariants

Some objects progress through distinct states

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 either confirmed or delivered it is considered sold. From another perspective, the session may be pendingInvoice, invoiced, or paid.

A state-chart defines boolean attributes and invariants

States are often drawn in a *state chart*, showing the states and relationships between them, as in Figure 78. Each state is a boolean attribute[1]: 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.

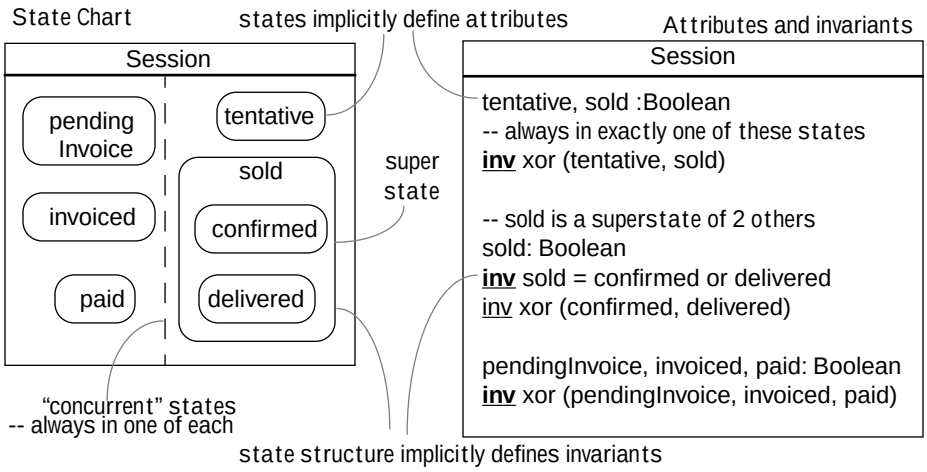


Figure 78: A state-chart defines state attributes and invariants

State can be exclusive,

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

inclusive,

- A state chart may 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 one of its states (or one from each of its concurrent sections) must be true. This is the or invariant defining sold.

or independent.

- A state chart may be divided into concurrent sections by a dashed line. Each of these is a separate simple state chart. The object is simultaneously in one state

1. We qualify the attribute name with superstates names to deal with nested states

from each of the sections. No explicit invariants are needed, since 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 why two such statements should not be true at the same time. We can separately introduce invariants that eliminate certain combinations, to represent business rules. For example,

We can also define explicit state constraints

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

Since 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

and relate states to other attributes.

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

thereby tying the state to the existence or not of a link to another object.

4.9.2 State-Transitions as Actions

In addition to defining state attributes and their invariants, state charts also depict transitions between states. An example state chart for Session is shown in Figure 79.

Transitions are partial action specs

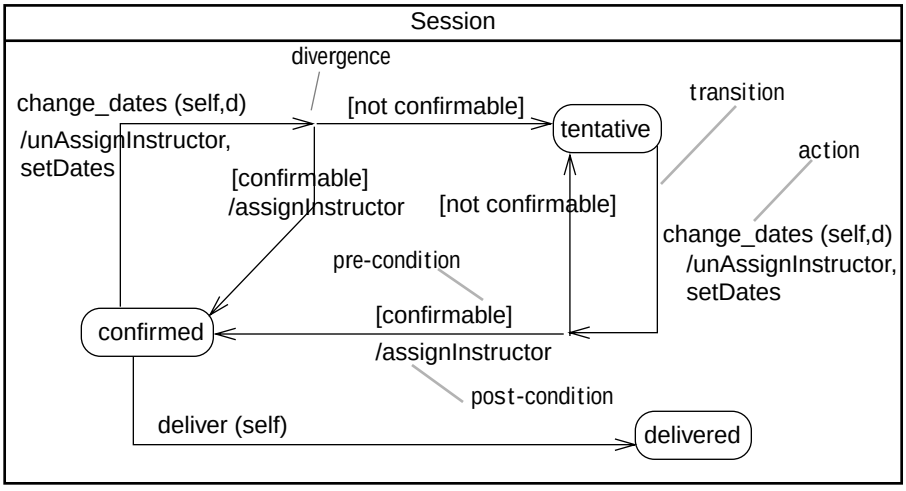


Figure 79: Session state-chart with transitions

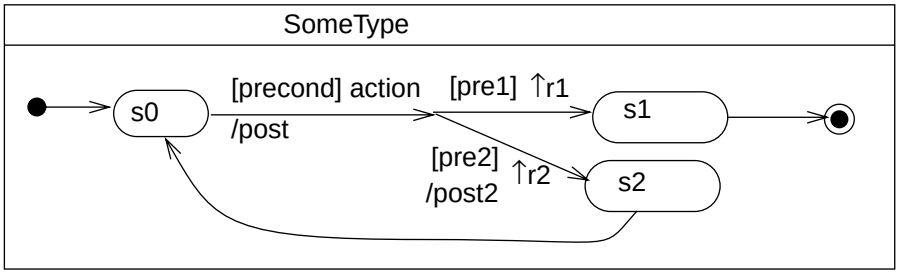
The change_dates action has multiple transitions, which translate into multiple partial action specifications. The transition out of the confirmed state is translated below. For brevity, the state chart omitted the parameters used in pre- and post-conditions, but we fill these in the textual action specs.

The translation is simple

```
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) => s.confirmed & assignInstructor (s, d) &
not confirmable (d) => s.tentative
```

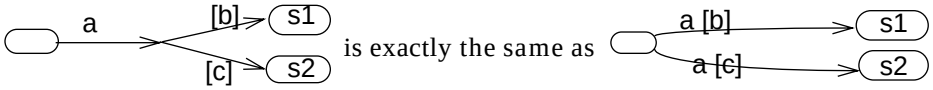
The elements of state chart notation is described below:



A state chart is part of the model of a type. A round-cornered box represents a state — the truth of a predicate. Each state implicitly defines an attribute of the type. The states in simple state chart are mutually exclusive; nested and concurrent super-states have different rules. Normally, the states in one diagram focus on one object.

Start and end are distinguished 'pseudo'-states

The arrows indicate what sequences of transitions are possible. "●" indicates which state 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 they become undefined i.e. 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 divergence is not a branch point in the programming sense, but just says there are two possible outcomes from this action: its postcondition includes (s1 or s2). Without the preconditions, either outcome would satisfy this spec.

Transitions specify pre/post conditions

State transitions share most of the machinery of effect clauses, described in Section 4.3.4. State transitions can be used to specify actions as well as named effects.

"[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.

"/post" — some effect achieved as part of executing this transition.

"↑action" — a (more abstract) action completed as part of executing this transition. We'll have more to say about this in the Chapter 14, *Refinement*.

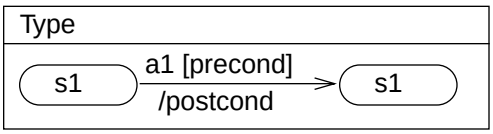
[[receiver.action]] — part of the effect of this transition is the same as the documented effect of action on receiver (which is self, the state chart object, by default).

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

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. Statecharts 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 have a definition in terms of the other attributes and associations of the type.

Multiple transitions define conjoined specs

When 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.

Superstates work as expected

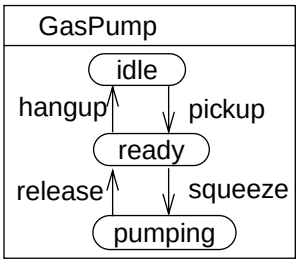
Transitions indicate the *completion* of the actions with which they are labelled. Although the actions may take time to accomplish, the transitions themselves are instantaneous; this will become more significant in Chapter 14, *Refinement.*

Transitions define action completions

4.9.3 State Charts of Specification Types

When 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 actions specs like:

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



A single state chart can describe only the simplest behaviors graphically

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 like Figure 75 cannot be described on a single state chart, except with the most trivial states (e.g. “exists”), with 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.

Complex objects have many state components

The technique we use here is to 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.

Describe their states via multiple state charts

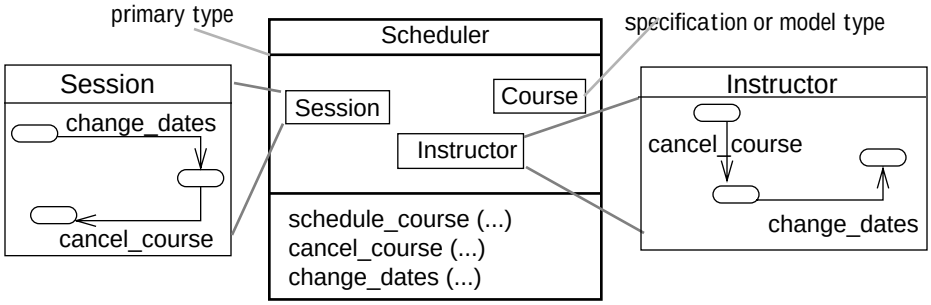


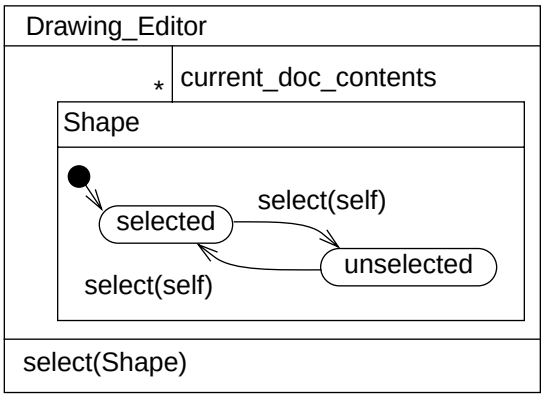
Figure 80: State Charts of Specification Types

The states and transitions are projections of the primary type and its actions

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

This is a useful and general technique

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 like this; it is slightly more convenient to use a single effect clause than separate the pre / post style here:

```
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 ⇒ unselected)
                  -- if it's the target and was not selected, select it
         &        ((s.unselected && s=shape)@pre ⇒ selected)
             )
```

4.9.4 Underdetermined transitions

Advanced Topic

Sometimes a state chart will be deliberately vague about the outcome of an action. The reason is usually to allow subtypes to make different choices, while the supertype gives the broad constraints, or to simply define a minimal partial constraint.

Transitions need not be fully deterministic

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 any precondition it is labelled with was true. You are allowed to write a state chart for which there are several feasible transitions at any moment, called an underdetermined set of transitions. When this is the case, what state will we end up in?

What does it mean to be under-determined?

The answer is that we 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 — just that there are forces at work of which you, based solely on the current spec, are unaware. 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.

Based on this spec you cannot assume a unique outcome

For example, dialling a phone number — dial_number — has several possible outcomes. As users we are unaware of the factors that will influence the outcome.

Dialling a phone has many outcomes

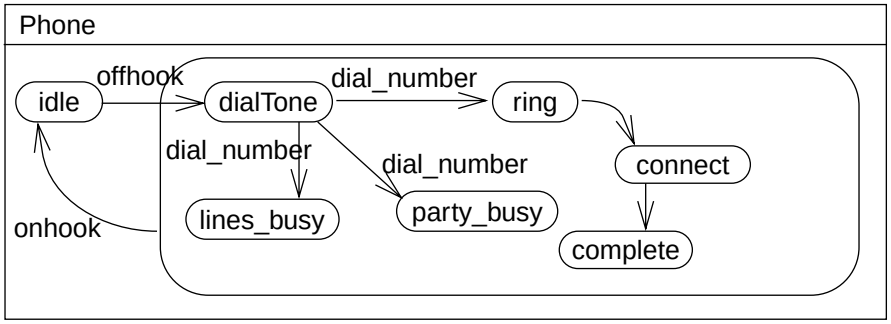


Figure 81: Underdetermined transitions

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.

At some level all outcomes are determined

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 cause these outcomes, and even when they don't, there is always a cause that we, the designers of the phone system, know about.

So why leave them under-specified?

Well, in this case, perhaps. 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, just what the possible outcomes can be. This might happen if we allowed our phone instrument to be connected to a new kind of switching system. As long we have a way of re-using this under-specified model, and adding to it in another context, this is a worthwhile separation to make.

Because the precise causes may differ in a different context

Advanced Topic

Silent transitions

A transition may even have an unknown trigger

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 you knowing why or when, and without you doing anything to it. Again, this does not necessarily imply pure randomness, just that either we don't know what might cause the change (as when 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 hanging on to see if the phone will be answered).

Silent transitions also let us describe systems that are not purely re-active, since the partial descriptions permit transitions with unknown causes.

4.9.5 Ancillary Tables

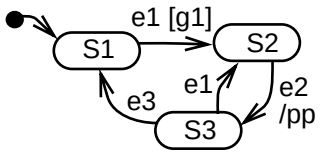
Two tables make the state view a better complement

State charts themselves provide a useful and different view of behavior from action specifications. They 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. There are two related tables that can be very helpful in conjunction with state charts, to check for completeness and consistency.

State transition matrix

A state chart can be represented by a matrix:

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



X = shouldn't happen
I = nothing happens
[ ] = 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 automatically generated from the state charts themselves; it better highlights combinations that may have been overlooked.

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, so are easy to show in a table. 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	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, < 2	<> null		0 < attr1 < 2 **&** assn1 <> null

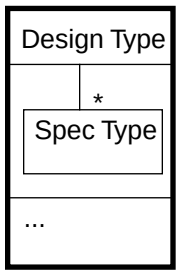
4.10 *Specification Types vs. Design Types*

Advanced Topic

The reason we are interested in specifications is to describe how a client should use a component (whether a software component or part of an organization; and no matter whether a complete software system or a large or small part of one). And what we really want to say about a component is *what it does* — its behavior, the actions it takes part in.

An implementation can look different internally from the external spec

We have seen how to specify the externally visible behavior of a type by specifying actions in terms of a type model of attributes. Since a type doesn't necessarily have to be designed along the same lines as its model, it may be that the implementation does not explicitly represent distinct and separate objects that belong to the types used within that model. When designing, some people like to distinguish those types they have decided to implement — "design types" — from those that are just there to help write a specification — "spec types". Design types are drawn with a heavier border.



Clients only need to know external behavior

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

> "This command generates a print job" — *what's a job?* "It's a thing with pointers to a user and a file" — *what can I do with it?* "Oh, you can't get hold of one yourself, but you can list all the jobs there are, and cancel any of your own" — *so there is one big list of jobs in there?* "Yup." — *isn't that a bit inefficient, considering there are so many different printers all over the Department? Wouldn't it be better to send each job to its own printer?* "Uh, well yes, that's how it's really done of course; and actually, there's really no such thing as a job, we just append the user name to the file. And actually, there's this hash table,..." — *But if I think of it as a big list of jobs, I'll understand how to use the system?* "Yes" — *thanks, that's what I need.*

Some terms and concepts are hypothesized

In this scenario, not only the attributes of the printing system — the list of jobs — but also the type of objects they contain (Job) is a convenient fiction hypothesized to describe its behavior aside from all the implementation complexities. Job's attributes also might or might not be directly implemented. Job is called a *specification type*, or a *model type* — it is only there for the purpose of modeling.

Others must be explicitly implemented

Types that are 'really' there (in the sense that they are separable and take part in actions and we intend to implement them) are called *design types.* Many types are used for both purposes — for example, Date is often used in specifications and also has many implementations. Also, in some situations the specification requires an implementation not just of a primary type, but also of related types as required for input and output parameters e.g. §8, p.153.

The key distinction is who participates in actions

Typically, a design type will be specified with a model drawn inside it, using modeling 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 4.10.1.

However, there is nothing to stop a type that happens to have an implementation somewhere also being used in a model. The more important design decision is how the types in an implementation will be used, and those decisions are recorded in collaboration diagrams.

4.10.1 Factoring to Specification Types

Action effects may depend on spec types

There are many cases when the outcome of an action depends on the type of the object or objects to which it is applied; indeed, this is one of the mainstays of the object-oriented approach.

e.g. different financial instruments

For example, if a financial institution keeps a variety of instruments under your name — life-insurance, mortgage, savings account, pension, etc. — what happens when you notify their system that you have died? (Well — perhaps someone does it for you!)

But the action is still on a containing type

The outcome is different for each different type of Account: the life insurance pays out and stops expecting regular pay-ins, the mortgage demands immediate repayment of its outstanding balance, the savings account does nothing, and the pension stops regular payouts. Yet the action itself is simply on the financial system itself; committing to more would be internal design decisions.

We can factor the effects onto spec types

The easiest way to deal with this is to factor the action spec as effects into the model's types (the short (pre:–post) here is just for brevity):

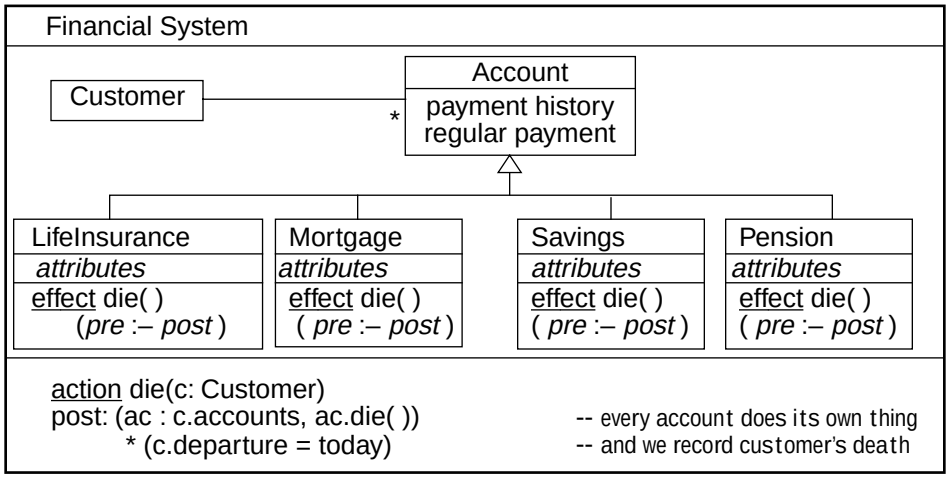


Figure 82: Factoring the effects of an action to spec types

This is not design; just factoring of a spec

Does this mean we are doing some design — assigning responsibilities and deciding internal interactions to the modeling types? Not really — we're just distributing the action spec among the concepts with which it deals, much as we did with state charts in Section 4.9.3. If there happens to be an implementation of, say, Mortgage, we are not referring to that implementation and the particular properties of its code: just the specification.

Yes, we could decide to design an object that is a member of the LifeInsurance type of our model; and yes, in that case it could very likely have a 'die' operation conforming to spec as given in the LifeInsurance type-box. But it's likely that the only design deci-

sion we've really taken so far is to implement FinancialSystem; and perhaps not even that — this model may be just a factored part of some larger one. It would still be possible to represent the information about a Customer's various policies in some spread-around way, with no single item corresponding entirely to any one type of account.

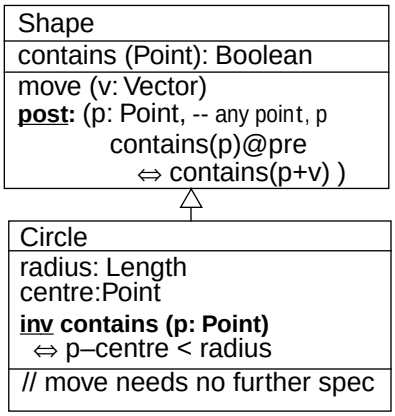
Design decisions are recorded separately, as refinements and ultimately as code. In a support tool, you know that a type is (to be) implemented if it has a link to one or more refinements (even if they aren't fully filled in yet). Some people like to draw with a bolder outline a type which has been, or is intended to be, implemented.

Design decisions will be recorded in refinements

Not all types will be used in the design. If a collaboration documents messages being sent to members of a given type, then clearly that type will have to be implemented in some way.

4.10.2 Factoring — specify effects abstractly

The 'die' example placed very different outcomes in each type. But where possible, it pays to look for something common between the supertypes. For example, moving a Shape in a graphical editor is very different in terms of the attributes of each subtype; but we can express the required effect in common terms, of what happens to the points contained within each Shape.



Even if action effects vary across subtypes, find abstract attributes and common spec

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

4.11 *Outputs of Actions*

Modeling outputs is a conscious abstraction

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 comes out of an action back to the invoker, or any output signals or requests that are generated to other objects. There are several approaches to this.

4.11.1 Return values

return values

An action can have a return type, the return value being the identity of some new or pre-existing object, used by the sender. Within the postcondition, **result** or **return** is the conventional name given to this value. Actions with return values are usually functions i.e. they have no other side effects.

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

4.11.2 Out parameters

Out parameters bind to attributes; ordinary ones bind to objects.

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. While out parameters can be broadly considered similar to return values, the details are somewhat different. The post-condition 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 some 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, like Java and Eiffel, do not have these features, which shows that it is possible to do without them in an implementation language. However, the idea of having multiple return values is itself very convenient in both specification and implementation.

4.11.3 Raised actions

A post-condition can specify invocations

It is also possible to state as part of a postcondition that another action has been invoked, either:

Synchronous

- Synchronously — the sent action will be completed as part of the sender. Its post-condition can be considered part of this one. 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. Asynchronous
  - Request sent to a specific receiver, and the 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]]

4.11.4 Specifying sequences of raised actions

Advanced Topic

When specifying raised actions, it is sometimes necessary to specify that they happen in a particular sequence — to describe the protocol of a dialogue. You might want that an "open_comms" action sends certain messages to a modem object in a particular order. We wish to specify this while retaining our basic premise of just using initial and final states in post-conditions.

Output actions sometimes must be sequenced

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

Some receiver must accept that sequence

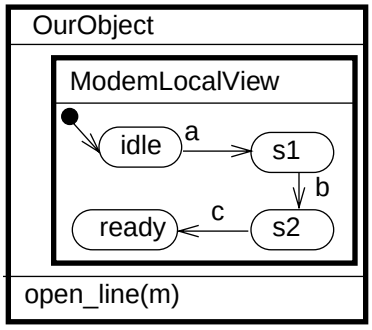
This is equivalent to saying that I have been asked to get it into the state of "having received message c". I haven't been told what that state might signify as far as the receiver is concerned; just that I've got to get it there. But also, you tell me that I must first send message b: in other words,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

...specified as a local view of that receiver

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:

The post-condition simply achieves the target state

```
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 sequences that are linear, branching, and concurrent.

Used in this way, states shown as separate in this local view can turn out not to be separate in an implementation of the modem; they are, however, separate in our object's implementation, since it must generate 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 there are no operations for actually finding out its state, that's OK.

Sequential outputs are often better dealt with as refinements, as described in Chapter 14. State charts are also are used to describe a collaboration refinement (in which 'zooming in' on an action shows it to be a dialogue of smaller ones).

Advanced Topic

4.11.5 Sequence expressions

Textual sequence expressions are better drawn as state charts

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

```
action stay (c: Customer, h: Hotel, dd: Dates)
    post:
        let bill: Money = h.rate * dd.days in
            [[   make_reservation (c,h,dd)   ;
                 change_dates (res, dd)*   ;
                 check_in (res);
                 check_out(res, bill) ]]
```

In other words, a stay in a hotel is a sequence of making a reservation, possibly changing the details (any number of times), checking in, then checking out (including paying the appropriate bill).

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

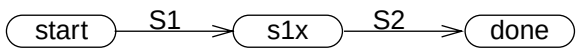
S1 ; S2	S1 always precedes S2
S1 | S2	S1 or S2
S1 *	Any number of repetitions of S1
S1 || S2	S1 concurrent with S2

All sequences [[...]] are an abbreviation for a state model with the implication:

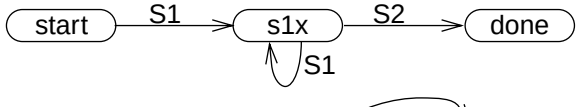
```
(start => done)
```

where the two states are defined by a state model, whose exact nature depends on the sequence expression:

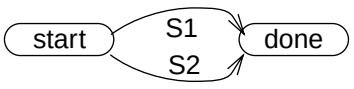
[[S1 ; S2]]



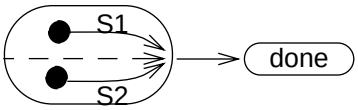
[[S1*; S2]]



[[S1 | S2]]



[[S1 || S2]]



4.12 *Subtypes with Attributes and Invariants*

A type specifies a set of objects

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 can extend the definition of a supertype

A subtype extends the specification of its supertype. It "inherits" all properties (attributes and invariants) of the supertype, and adds its own specifics. Since all supertype properties still apply to it, and its members have to conform to all properties, every member of a subtype is also a member of its supertype i.e. a subtype's members are a subset of its supertype members.

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

All properties are inherited

The types CourseEngagement and ConsultingEngagement could both be subtypes of ServiceEngagement. Objects of the CourseEngagement type have a total of 6 attributes defined on them; these objects may have different implementations of these attributes, so 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.

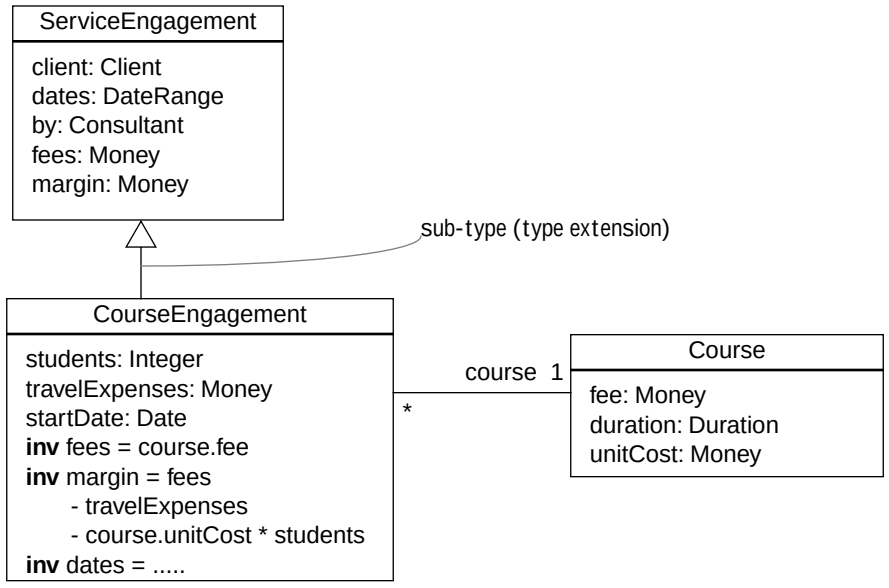


Figure 83: Subtype extends definition of supertype

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. On closer consideration, both of them follow the same basic rule for their margin:

margin = fees - travelExpenses - additionalCosts

Common specs can be refactored to the supertype.

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 are due to the per-student production costs; additional costs for consulting may be due to 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 just once in the supertype.

A type is not a class

A type is not a class. A class is an OOP construct for defining the common implementation — stored data and executed methods — of some objects, while 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[1].

A subtype does not imply a subclass

A subtype is not a subclass. Specifically, a subtype in a model does not imply that an OOP class which 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*; just extension.

Not all subtype are interesting

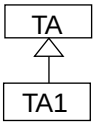
As with objects and attributes, there are many ways of partitioning subtypes. ServiceEngagements could be viewed based on their geographic location (domestic vs. international), taxation status (taxable or not), nature of service provided (consulting vs. training), etc. Which of these are relevant is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

4.12.1 Common Pictorial Type Expressions

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

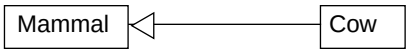
4.12.1.1 Subtype

TA1 extends TA — it inherits all its attributes and action-specs. TA1 may add more action specs, for the same or different actions. Viewed as sets of objects:



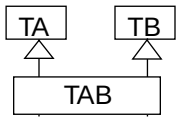
$TA1 \subseteq TA$

Example:



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

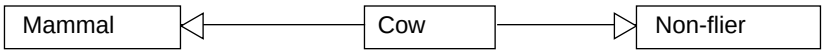
4.12.1.2 Multiple supertypes



TAB 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:

$TAB \subseteq TA * TB$

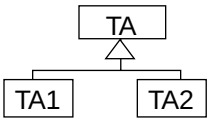
Example:



Subtype is strictly additive

The subtype conforms to all the expectations that any client could have based solely on the guarantees of one supertypes. Further requirements particular to this subtype may be added. It's perfectly possible to combine two types that have conflicting requirements, so that you just couldn't implement the result.

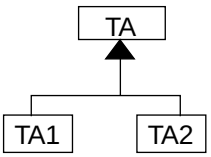
4.12.1.3 Type exclusion



The two types are mutually exclusive i.e. no object is a member of both.

$TA1 * TA2 = \varnothing$

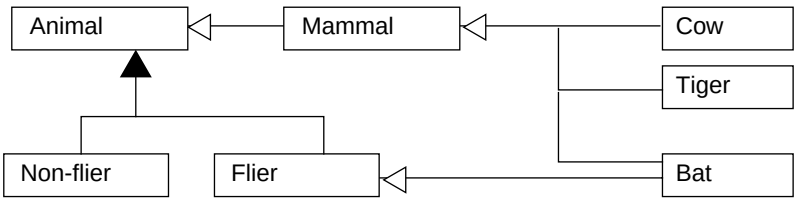
4.12.1.4 Type partitioning.



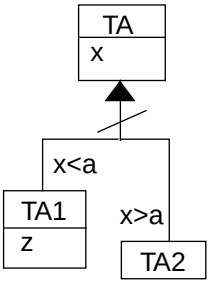
Every member of TA is a member either of TA1 or TA2, but not both. There may be more than one partitioning of a type, each drawn with a separate triangle.

$TA = TA1+TA2$

Example:

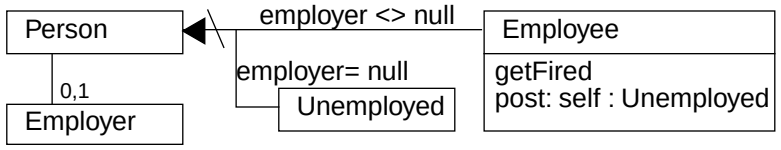


4.12.1.5 State-types.



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 4.13). 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. This construct is a convenient way to define conditional type membership, of the form: *a TA is also a TA1 if it has property x<a; in which case it also has these other properties (z, etc).*

Example:



This is a form of checked subtype relation, providing an implicit classification facility. It lets you use predicates as type names, and is helps to bridge some common and intuitive usage of terminology that may not map to a strict meaning of types. Consider what state types of Person would permit the following:

State types let you use very natural names

```
Club::admit (t: Teenager)
    post:   t.stipend.depleted
            [[ t.hasEnjoyedSelf() ]]

BabySitter::admit (b: Baby)
    pre:    b.diaper.unsoiled
    post:   [[ b.hasEnjoyedSelf() ]]
```

4.12.2 General Type Expressions

Advanced Topic

Since types define sets of objects, we can use set operations to combine types. These expressions can be useful in assertions. In that context, the name of a type refers to its current set of existing objects.

jo : Student	Object jo is a member of the type Student, conforming to the behavioral requirements set by Student.
Tutor * Student	the type whose members are in both these types. Maybe laura : (Tutor * Student) .
Tutor + Student	the type whose members are in either or both type. We might define a type: CollegeMember = Professor + Student
Person – Pilot	the type whose members behave according to the first type's definition, but have behavior inconsistent with the second.
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 are of this form.

4.13 *Subjective model — the meaning of "containment"*

Type model 'containment' means two things

When we specify a type, we will often depict its attributes and specification types with a distinguished "root" type by using a form of visual containment, as shown in Figure 84(a) (or with a distinguished type node marked with «root», in (b); the former has the advantage of permitting multiple levels of nesting, to which these rules apply uniformly). This is more than a cosmetic choice; it has a specific semantic meaning.

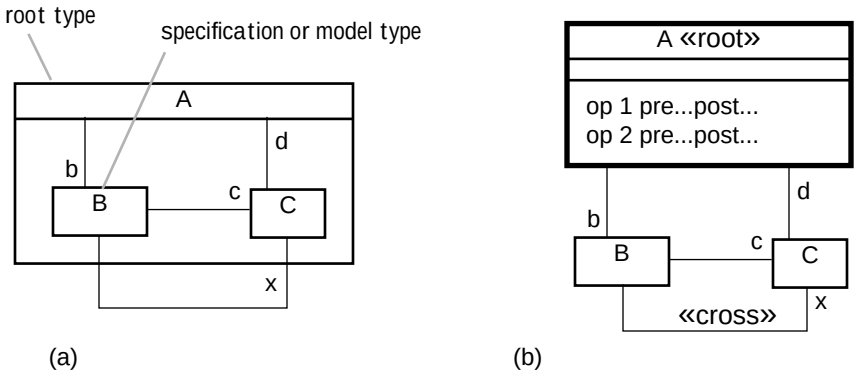


Figure 84: Type model: (a) with "containment" (b) with "root"

There are two meanings to a type diagram with a "root" type. Both are illustrated by a traslation to this explicit model:

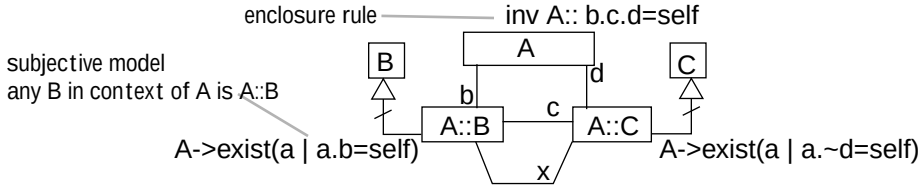


Figure 85: Interpretation of "containment"

Enclosure

All paths from A back to A that use only associations defined within the box (or, that do not use any links marked «cross») are guaranteed to get back to the same instance of A i.e. all links lie within the tree of objects that is rooted at self:A. By contrast, though 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 they are within the same vehicle. This rule works as expected across multiple levels of nesting.

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' i.e. those members of B (or C) who happen to be contained within an A. Brought outside the boundary of A, they have prefixes to their names. In a different context, every B might not be linked to a C. For example, to the Invoicing Dept, Customers are linked to

Products, while Warehousing knows only that Products have Parts. Containment represents a localized view.

The local usage of a type is a state-type of the common usage. In addition, local usages can also directly refer to attributes or operations on their container[1]. This makes it simple to write localized specifications of actions and effects, as discussed in Section 4.9.3.

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 will also need to describe operations on the input or output parameters of these operations, as shown in Figure 68.

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

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

4.14 *Programming Language: Classes and Types*

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

Class prescribes internal structure

A class is an implementation unit that prescribes the internal structure of any object that is created as an instance of it. An object belongs to one class throughout its life.

Classes can be implemented in any language

'Class' is an OO programming concept — which doesn't necessarily mean 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 if, for example, you need to write in a traditional language like Fortran or assembler — perhaps for especial control of performance. That way, you still get the benefits of OO design (modularity, re-use, etc). Of course, OO programming languages best support object design.

Classes also cover persistent objects

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 an object in filestore a message. If you can secure a good OO database you're in luck; but otherwise, you're typically stuck with plain old files or a relational database, and need to think how to encode the objects. Your class-layer design should intially defer the question of how objects are distributed between hosts and media.

The *class layer* of design

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 like Java, Eiffel, or C++; or otherwise by judicious application of class-to-non-class patterns.

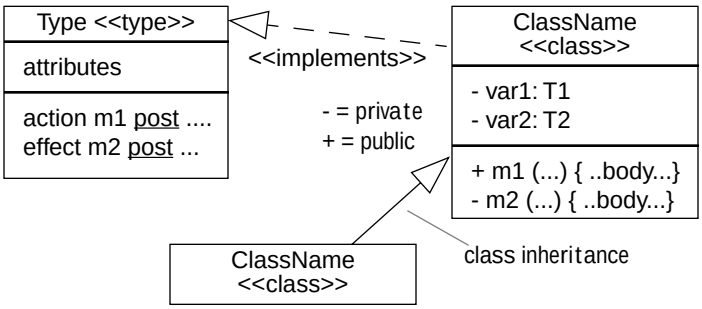


Figure 86: Class models

Not all 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.

4.14.1 Messages and operations

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

Classes implement operations

- *Message* — an invocation of an operation, consisting of the name of the operation, the identity of the recipient, and a set of arguments. Like a procedure call, except 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!
- *Operation* — a procedure, function, 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.
- *Receiver* — the object distinguished as determining which operation will be invoked by a given message. Normally 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.

4.14.2 Internal variables and messages

There are 4 primary kinds of variables in an object-oriented program:

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

Operations use object variables

Operations read and write variables local to each object. A variable refers to an object: it is important to distinguish variables from objects, 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 (e.g. 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 may 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 analysing the code.
- *Access* — which methods can get at it. 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 (= group of classes designed together), or at a finer grain. This makes sense, as each package is the

responsibility of one designer or team; and any changes within the package can readily be accommodated elsewhere within the same package. Encapsulation is only important 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: that is, one class is tied to using one specific other — not a generic design. So 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.

4.14.3 Class extension

Class inheritance is one mechanism to re-use implementation.

Inheritance, derivation, or extension mean that the definition of one class is based upon 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.

Multiple *class* inheritance is tricky.

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.

Class inheritance overhyped.

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

Re-use implementation only if you intend to re-use spec.

In general, if you want to base your code on someone else’s, it is best to use your favorite editor to copy and paste. Unless the spec of your code is closely coupled to theirs, it’s quite unlikely that you’d want to inherit any modifications they make. In fact, the big benefit of O-O design comes more from polymorphism — conformance of many classes to one spec; if in some cases this is achieved partly by sharing some code, then that’s nice, but not necessary. Arbitrary code-sharing just couples designs that ought to be independent.

4.14.4 Abstract classes

Inheriting classes should behave compatibly

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.

So abstract operations need behavior specs.

That raises the question of what a Shape is expected to do — which 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 just list the messages it expects. The rest is defined by each subclass in its own way.

Abstract classes approximate types.

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

Classes with subtypes shouldn't have instances

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).

Classes should depend only on other types — not other classes.

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

Use factories to create objects.

But there is one case in which this does not work so well: when you want to create an object, you must say what class you want it to belong to. However, there are a number of patterns such as Factory that help localise 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.

4.14.5 Types

Type should still be a specification

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

Java has interfaces

Java. A Java *interface* is a pure type -- i.e. client visible behavior. You define interfaces for the major categories of clients you expect to have. You factor your services into different interfaces, and can define some interfaces as extensions of other interfaces (i.e. subtypes), to offer suitable client views. This provides those clients with a pluggable type requirement, where any object which 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, just a class.

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

C++ has pure abstract classes

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;
    public 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 {
    ....
}
```

Smalltalk can use message categories

Smalltalk. 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 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.

- A client expects a 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 we subclass or not; we could just as well cut / paste the methods, delegate, or code it all ourself. In Smalltalk the only check is a run-time 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 may be represented by systematically following programming conventions that use a “message protocol” or “message category” as a type.

4.14.6 Generic types

A generic generates many defintions.

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 substituted 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, many well-thought out experimental languages, and — with any luck — 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 10, *Model Frameworks and Template Packages* (p.389)). We argue that this is a very important feature of component-based design.

  Either single classes or groups of related classes can be made generic.

- *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 and when 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 have to pass the source code around. By contrast, the generic parameters of FOOPS [Goguen] come with "parameter assumptions" about such properties. The compiler will check that you have made all such assumptions explicit when first compiling the generic; and can guarantee that it will work for all conforming argument classes.

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

Catalysis frameworks can map to generics

4.14.7 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. Although it is most commonly just the compiler that makes use of these facilities, 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.

A class can itself be an object.

Java also supports such a reflexive layer: classes, interfaces, methods, and instance variables can all be manipulated as run-time entities.

In open systems design, it is important that an object should be able to engage in a dialogue 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 have to have the facility stuck on: C++ has recently aquired a limited form of such a feature with RTTI (run-time type identification).

Open systems should be reflexive

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.

4.14.8 Specifications in classes

4.14.8.1 'implements' assertions

A class implements types

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 this class.

Java supports this directly

In Java, types are represented by interfaces and abstract classes. Even though the complete specification of the type (pre and postconditions etc) is not understood by the compiler, the clause

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

documents the designer's intention to satisfy the expectations of anyone who has read the spec associated with the interface Food. Java allows many classes to implement one interface, directly or through class extension.

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

should mean that the class implements the type represented by its superclass — as well as extedning the definition of its code.

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

C++ rules are slightly different

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 above); but the usual recommendation is to use an internal variable of the proposed base type instead.

4.14.8.2 Constructors

Constructors initialize new instances

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 i.e. satisfy the expected invariants.

4.14.8.3 Retrieval

Type model attributes describe state of implementation

A fully-documented implementation claim is backed up by a justification; the minimal version is a set of retrieve functions (Section 7.7, "Refinement #4: Operation conformance," on page 315). Writing these often exposes bugs.

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

```
interface Shape {
    attribute[1]  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) { ... }
}
```

4.14.8.4 Operation specs

Operations can be specified precisely

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).

Some languages support such specs.

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.

1. 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.

4.15 *Using type specifications*

This chapter has dealt in detail with the business of specifying actions — what happens in some world, or in some system — without going into the detail of how it happens. Indeed, we have seen an example (§4.4, p.144) of how two different implementations can have the same behavioral specification.

The action specifications use the terms defined in a static model (as inthe previous chapter). The static and action models together make up the specification of a complete type:

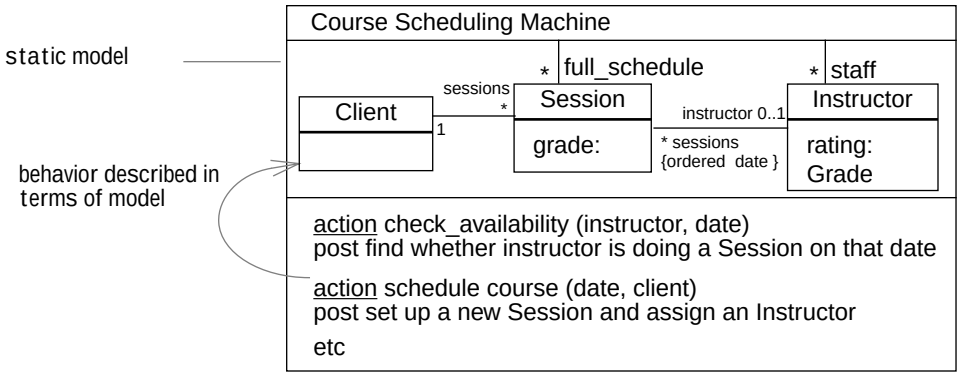


Figure 87: Complex model — pictorial

In the chapters that follow, we will use these ideas to build specifications of complete software systems and interfaces to components.

But first we should deal with the interactions that go on between objects: both inside the object we have specified, as part of its implementation; and also between our object and others, to understand how it is used by our (software or human) clients.

4.16 *Behavior Specifications — summary*

- An object's behavior (or part of it) may 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 take place.
- A precondition defines when the associated postcondition is applicable.

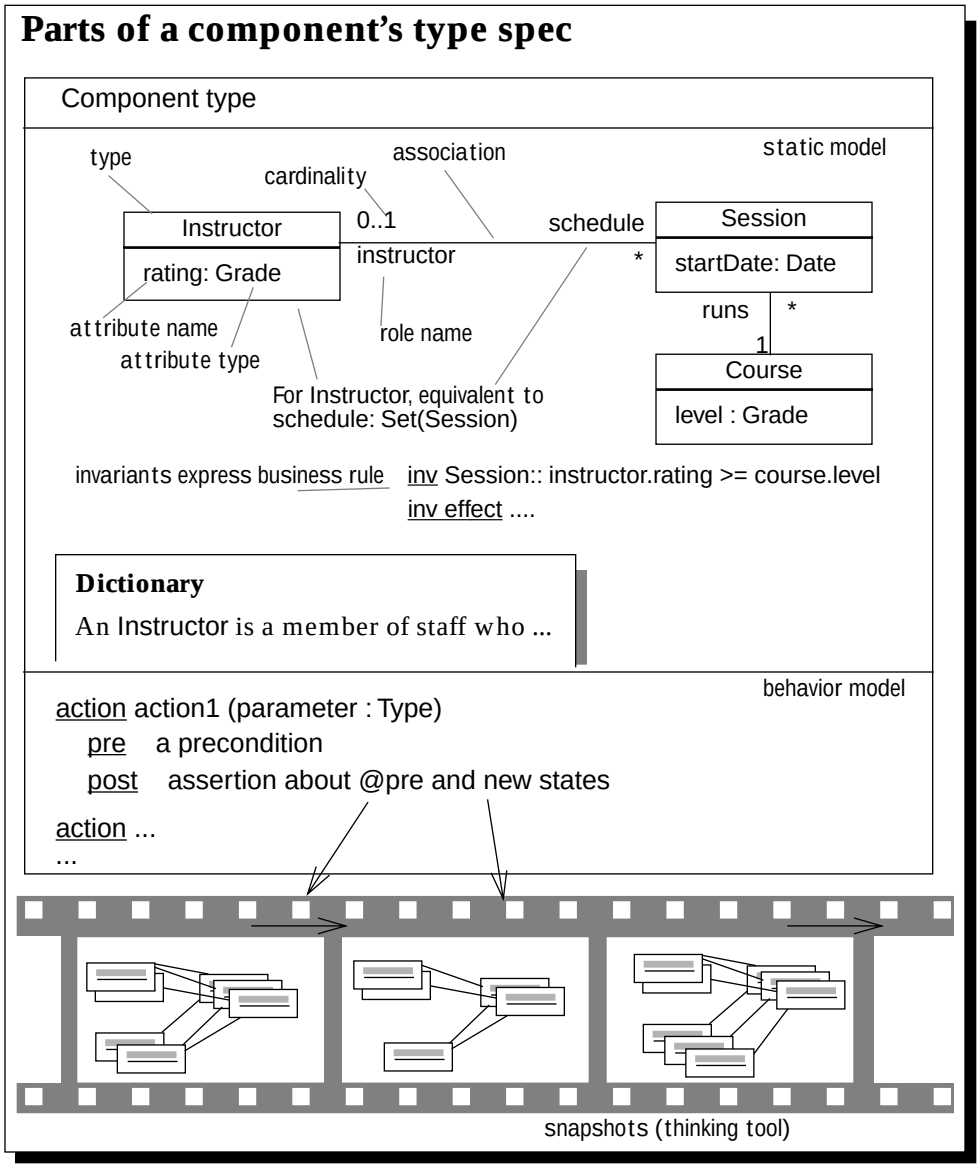


Figure 88: Type models