Chapter 2 Static Models Object Attributes and Invariants

Models can be divided into static, dynamic, and interactive parts dealing with, respectively, what is known about an object at any one moment, how this information changes dynamically with events, and how objects interact with one another. This chapter discusses the static part of a model, in which you characterize the state of an object by describing the information known about it at any point in time. It uses the type model diagram to capture the static model and snapshot diagrams to show instantaneous configurations of object state.

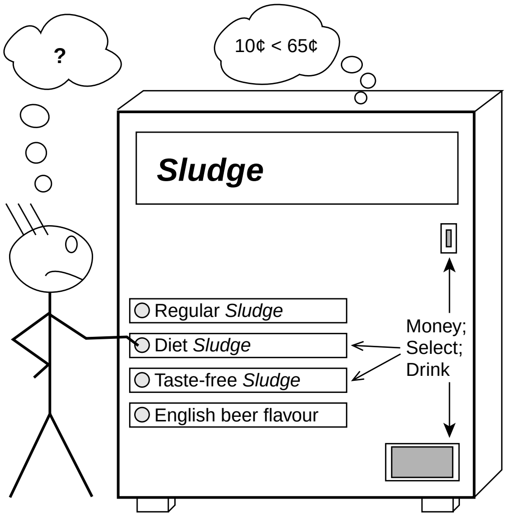
The first section is an overview of what a static model is about. Section 2.2 introduces objects, their attributes, and snapshots and distinguishes the concept of object identity from object equality. The attributes that model an object's state can be implemented in very different ways. Section 2.3 outlines some implementation variations using Java, a relational database, and a real-world implementation.

Section 2.4 abstracts from individual objects and snapshots of their attribute values to a *type model*, which characterizes all objects having these attributes. Here, we introduce parameterized attributes; graphical associations between objects, collections of objects, and type constants; and type combination operators.

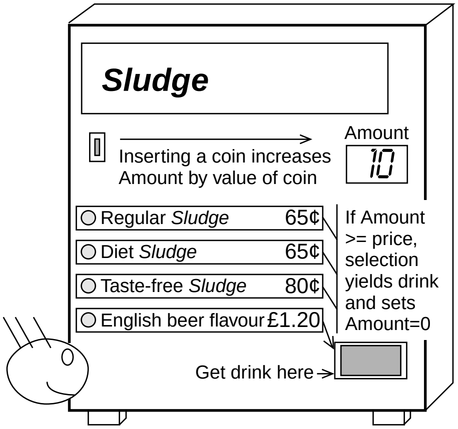
Not all combinations of attributes values are legal. Section 2.5 introduces static invariants as a way of describing integrity constraints on the values of attributes, shows some common uses of such invariants, and outlines how these invariants appear in the business domain as well as in code.

The same model of object types and attributes could describe situations in the real world, for a software specification, or even of code. Section 2.6 introduces the dictionary as a mechanism for documenting the relationship between model elements and what they represent.

2.1 *What Is a Static Model?*



Much of how an object responds to any interaction with its surroundings depends on what has happened to it up to now. Your success in checking into a hotel, for example, depends on whether you have previously called to arrange your stay. The hotel with which you've successfully gone through this preliminary courtesy will welcome you with open arms, whereas others on the same night might well turn you out into the cold. The response to your arrival depends on the previous history of your interactions. So it is with many other encounters in life: the beverage machine that will not yield a drink until you have inserted sufficient money; the car that will not respond to the gas pedal unless you have previously turned the starter key and provided that you have not since switched it off; the file that yields a different character every time you apply the read operation. The response to each interaction you have with any of these objects depends on what interactions it has already had.



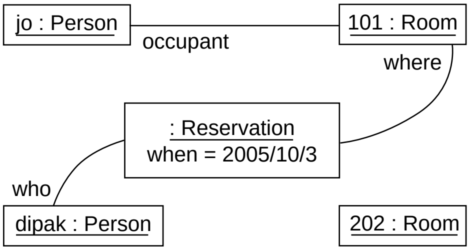
To simplify our understanding of this potentially bewildering behavior, we invent the mental notion of *state*. The hotel has a reservation for me, the machine is registering 20¢, the car's engine is running, the file is open and positioned at byte 42. The idea of state makes it easier to describe the outcome of any interaction because instead of talking about all the previous interactions it might have had, we merely say (a) how the outcome depends on the current state and (b) what the new state will be.

It doesn't matter much whether the user can observe the state directly through a display on a machine, by an inquiry with a person, or by calling a software function. To provide such a facility is often useful, but even if it isn't there the model still fulfills its main purpose: to help the client understand the object's behavior. If you take away the numeric display in the cartoon but leave the instructions and the crucial state attribute Amount, the machine is still more usable than with no such model.

Nor does it matter how the state is realized. The hotel reservation might be a record in a computer, a piece of paper, or a knot in the manager's tie. The same principle applies inside software; the client objects should not care how an object implements its state. State is a technique that helps document the behavior of an object as seen by the outside world.

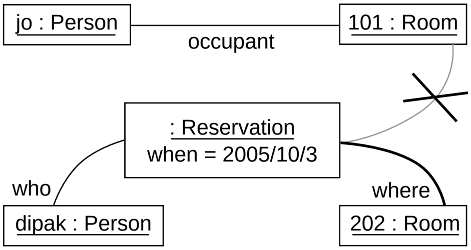
In fact, it is important that a client not depend on how the state is implemented. Back in the olden days of programming when a team would write a software system from scratch, every part of the system was accessible to every other part. You just had to stick your head above the partition to shout across at whoever was designing the bit whose state you wanted to change. But in recent times, software has joined the real business world of components that are brought together from many sources, and you shouldn't interfere with another object's internal works any more than you should write directly on the hotel's reservation book (or the manager's tie). It would be wrong and inflexible to make assumptions about how they work.

2.1.1 Snapshots: Drawing Pictures of States



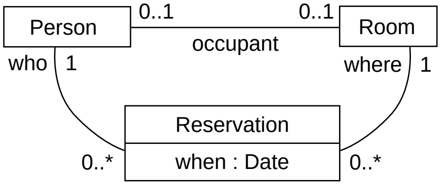
To illustrate a given state, we use *snapshots* (see Section 2.2.2). The objects represent things or concepts; the links represent what we know about them at a particular time. In this example, we can see that Jo is currently occupying Room 101 and Dipak is currently scheduled to occupy Room 101 next week.

Actions can be illustrated by showing how the attributes (drawn as lines or written in the objects) are affected by the action. Here, the rescheduling operation has been applied to shift Dipak to Room 202 next week. (Although we say that a snapshot illustrates a particular moment in time, it can include current information about something planned or scheduled for the future and also a current record of relevant things that have happened previously.)



These drawings first and foremost represent states. The same kind of drawing can be used to represent specific implementations of a state. For example, we could decide that each link represents a row in a relation in a relational database; or they might be pointers in main memory; or they might be rows in a chart at the hotel's front desk. But at the start of a design the most useful way is to say that we don't care yet: we're interested in describing the states and not the detail of how they're implemented. We will make the less important representation decisions in due course as the design proceeds.

2.1.2 Static Models: Which Snapshots Are Allowed



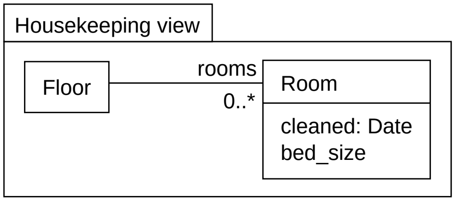
Whereas snapshots illustrate specific sample situations, what we need to document are the interesting and allowed states: the objects, links, and labels that will be used in the snapshots. This documentation is the purpose of a static model, which comprises a set of type diagrams and surrounding documentation. Notice the difference in the appearance of type boxes and object-instance boxes: the headers of instances are underlined and contain a colon (:). This example summarizes the ways in which rooms and people can be related.

Most analysts and designers are familiar with the idea of establishing a project glossary at an early stage as a way to get everyone using the same words for the same things. In Catalysis, the type diagrams are the central part of the glossary: they represent a vocabulary of terms and make plain the important relationships between them. That vocabulary can then be used in all the documents surrounding the project and in the program code itself.

2.1.3 Using Static Models

Static models have different uses in different parts of the development life cycle. If we decided to start from scratch in providing software support for a hotel's booking system, our analyst's first deliverable would be a description of how the hotel business works, and a type model would be an essential part of it, formalizing the vocabulary. Later in the life cycle, the objects in the software can be described in the same notation.

When applied to analyzing the real world, modeling is never complete. There's more to say about a Person than which room he or she is in; every type diagram can always be extended with more detail. This is just as true within software. We saw earlier that as a client of an object, you are interested in a model that helps explain the behavior you expect of it—but you don't care how it is actually implemented. The model can omit implementation detail and have a completely different structure from that of the implementation as long as the client gets an understanding to which the actual behavior conforms.



Some tools and authors use the term *class diagram*. We reserve *class* for the most detailed level of design, representing what's actually in the code. A class box (marked «class») shows all the directly stored attributes and links of its instances. A *type* is more general: Its attributes and associations represent information that can be known about any of its members, without stating how. We use types much more than classes in analysis, and both during design.

A model may focus on one view among many possible views. For example, the housekeeping staff may be interested in recording when a room was last cleaned. When we come to implement the software, we will need it to cope with all these different views, so we must combine them at some stage. Conversely, part of our overall implementation might be to divide the system into components that deal with different aspects, in which case we must do the reverse. (We'll discuss both operations in Part III.)

It's important to realize that the simplified model is still a true statement about the complex implementation. The attributes and associations tell us about what information is there; they do not tell us how it is represented. Different models can be written at different levels of detail, and we can then relate them together to ensure consistency; you'll find more about this in Chapter 6, Abstraction, Refinement, and Testing.

A model of object state is used to define a vocabulary of precise terms on which to base an analysis, specification, or design. A well-written document should contain plenty of narrative text in natural language along with illustrative diagrams of all kinds, but the type models are used to make sure that there are no gaps or misunderstandings. More on this in Chapter 5, Effective Documentation.

2.2 *Object State: Objects and Attributes*

In this section we introduce the basics of objects and their attributes. Before going any further, we'll introduce an example that will run through the rest of the chapter. IndoctriSoft Inc. is a seminar company that develops and delivers courses and consulting services. The company has a repertoire of courses and a payroll of instructors. Each session (that is, a particular presentation of a course) is delivered by a suitably qualified instructor using the standard materials for that course, usually at a client company's site. Instructors qualify to teach a course initially by taking an exam and subsequently by maintaining a good score in the evaluations completed by session participants.

2.2.1 Objects

Anything that can be identified as an individual thing, physical or conceptual, can be modeled as an object; if you can count it, distinguish it from another, or tell when it is created, it is an object. All the things in Figure 2.1 are valid objects, drawn as boxes with underlined names for each object. Of course, not all valid objects are interesting. As we will see, the behaviors that we wish to describe determine which objects and properties are relevant.

© ***object*** Any identifiable individual or thing. It may be a concrete, touchable thing, such as a car; an abstract concept, such as a meeting; or a relationship, a number, or a computer system. Objects have individual identity, characteristic behavior, and a (perhaps mutable) state. In software, an object can be represented by a combination of stored state and executable code.

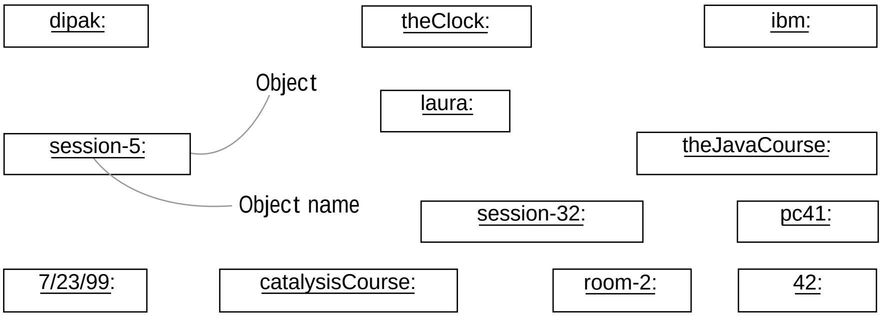


Figure 2.1 Some objects.

2.2.2 Attributes and Snapshots

The state of an object, the information that is encapsulated in it, is modeled by choosing suitable *attributes*. Each attribute has a label and a value; the value may change as actions are performed. In constructing a model, we choose all the attributes that we need to say everything we need to say about the object.

© ***attribute*** A named property of an object whose value describes information about the object. An attribute's value is itself the identity of an object. In software, an attribute may represent stored or computable information. An attribute is part of a model used to help describe its object's behavior and need not be implemented directly by a designer.

For example, session-5 (in Figure 2.2) has the attributes startDate, instructor, course, and client. The value of an attribute is the identity of another object, whether a big, changeable object such as IBM or a simple thing such as 1999/7/23. The attributes of an object link it to other objects. Some attributes are mutable (that is, they can be altered to refer to other objects); others are unchanging, defining lifetime properties of the object. For session-5, the value of its startDate and instructor attributes will change as scheduling needs change; but its client attribute will remain unchanged for the lifetime of session-5. These snapshots, or instance diagrams, are useful for illustrating a given situation, and we will use them for showing the effects of actions.

© ***snapshot*** A depiction (usually as a drawing) of a set of objects and the values of some of their attributes at a particular point in time.

The predefined name *null* or ∅ refers to a special object; the value of any unconnected attribute or link is *null*. In Figure 2.2, the catalysisCourse does not currently have an owner.

2.2.3 Alternative Ways of Drawing a Snapshot

The links drawn between the objects and the attributes written inside the objects are different ways of drawing the same thing. We tend to draw links where the target objects are an interesting part of our own model and draw attributes where the value is a type of object

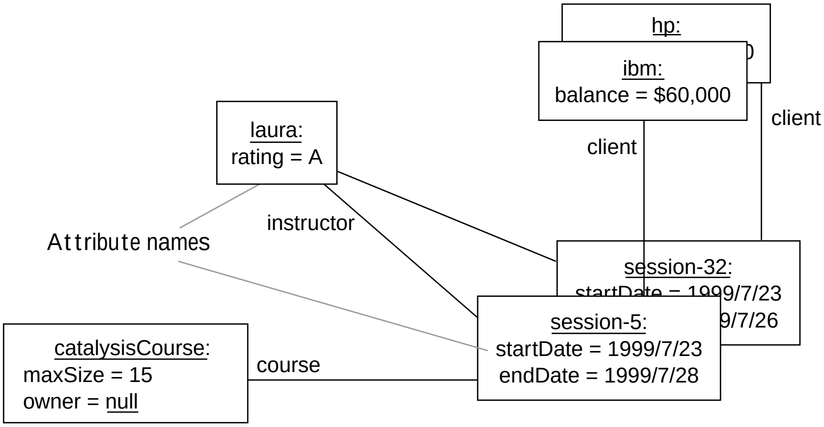
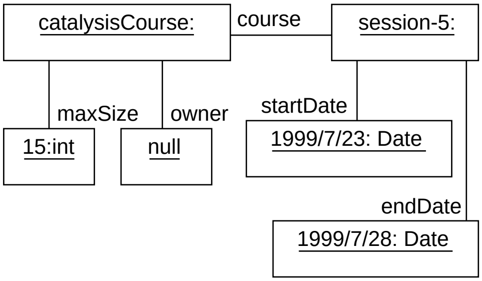


Figure 2.2 Snapshot depicts attribute values of objects.

imported from elsewhere, such as numbers and other primitives. To emphasize this point, let's look at some alternative ways of drawing Figure 2.2.

It's worth remembering, especially if you are designing support tools, that a diagram is a convenient way of showing a set of statements. There is always an equivalent text representation:

session-32 . instructor = laura
session-32 . startDate = 1999/7/23
session-32 . endDate = 1999/7/26
session-32 . client = hp
session-5 . instructor = laura
session-5 . course = catalysisCourse
session-5 . client = ibm
session-5 . startDate = 1999/7/23
laura . rating = A
catalysisCourse . maxSize = 15
catalysisCourse . owner = null
ibm . balance = $60,000



Alternatively, we could draw the boxes but write all links as attributes (see Figure 2.3). Notice that every link has an implied reverse attribute: if the instructor for session-5 is Laura, then by implication Laura has an attribute (by default called ~instructor) modeling the sessions for which Laura is the instructor, which must include session-5. Attributes in a model are about the relationships between things; whether we choose to implement them directly in the software is another question.

Now let's go to the other extreme and draw all the attributes as links. Picking out part of Figure 2.2, we could have just as well have shown Dates, and even the most primitive numbers, as separate objects. An attribute is generally written inside the box when we've

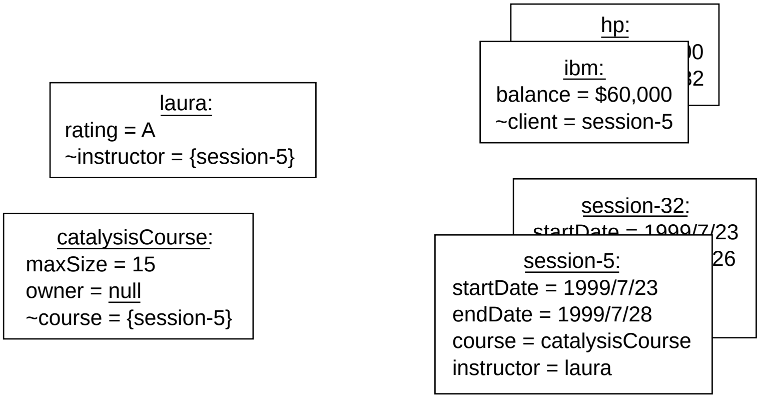


Figure 2.3 Snapshots with links written as attributes.

nothing interesting to say about the structure of the object it refers to. Everyone knows what numbers and dates are, so we have no need to show them in detail.

We regard as objects primitive concepts such as dates, numbers, and the two Boolean values. You can alter an attribute (such as maxSize) to refer to a different number, but the number itself does not change. Useful basic and immutable attributes of numbers include "next" and "previous," so that, for example, 5.next = 6. (Many tools and languages like to separate primitives from objects in some fundamental way. The separation is useful for the practicalities of databases and the like, but for most modeling there is no point in this extra complication.)

2.2.4 Navigation

Given any object(s), you can refer to other related objects by a *navigation* expression using a dot (".") followed by an attribute name. The value of a navigation expression is another object, so you can further navigate to its attributes:

```
session-5 . course = catalysisCourse
session-5 . course . maxSize = 15
session-5 . client . balance = $60,000
```

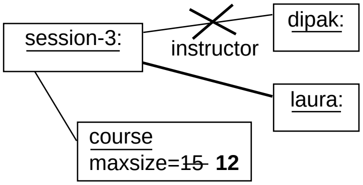
In the preceding and the earlier expressions, session-5, ibm, and 1999/7/23 are names that refer to specific objects—only names of objects can start off navigation expressions; startDate, instructor, course, and client are attributes—they occur to the right of the dot. Usually the "names" that refer to objects are variable names, such as formally named parameters to actions or local variables, and constants, such as 1999/7/23, A, and 15. We build navigation expressions from names and attributes.

2.2.5 Object Identity

Every object has an *identity:* a means of identification that allows it to be distinguished from others. An identity might be realized in all sorts of ways: a memory pointer, a database key, a reference number or name of some sort, or a physical location. Once again, the point about making a model is to defer such questions until we get down to the appropriate level of detail.

If you've done any database design, you'll be familiar with the idea that every entity must have a unique key, which it is up to you to assign. The key is an explicit combination of the entity's attributes, and any two entities with the same attribute values are actually the same one. But in object-oriented design, we always assume an implicit unique key. If you implement in an OO language or on an OO database, it provides the key for you; otherwise, you make it explicit when you get to coding.

An object identity can be assigned to a suitable attribute or program variable. Changing an attribute value to refer to a different object—for example, the session's instructor is changed from dipak to laura—is different from having an attribute refer to the same object whose state has changed; for example. The maxSize for the session's course may change, but that session is still of the same course, and the number 15 itself has not changed.



Two different navigation paths may refer to the same object; for example, the object referred to as "my boss" may be the same object as that referred to as "my friend's wife." In Figure 2.2, session-5.instructor and session-32.instructor both refer to the same object, laura. If both names refer to the same object, they both see the same attribute values and changes to those values. "x = y" means x and y refer to *same* object; "x <> y" means that x and y refer to different objects. These symbols are based on the Object Constraint Language (OCL) in UML; in C++ and Java we'd use == and !=.

But we must be careful about what relationships such as "equal" mean. session-5 and session-32 may be the same course, for the same client, starting on the same day, and yet they are two different sessions. The seminar company might choose to call two courses equal if their courses, dates, and clients are the same. But we know that they're different objects because operations applied to one don't affect the other: if session-5 is rescheduled, session-32's date remains unchanged.

Similarity or equality relationships must be defined separately for each type depending on the concerns of the business. We can attach a definition to each type in the model we build, picking out the attributes of interest:

```
Session::                          -- For any individual Session-instance ‘self’, we
    equal (another:Session) =      -- define “equal to another Session” to mean ...
         (      self.startDate = another.startDate
        and     self.endDate = another.endDate
        and     self.course = another.course
        and     self.client = another.client      )
```

The meaning of identity for an object type is defined, to a great extent, by constraints on whether two distinct objects can have the same attribute values. For example, in a navigation application we may deal with a type Location consisting of a latitude and longitude. If part of the definition of this type includes a constraint that no two distinct location objects can have identical latitudes and longitudes, then an identity check would suffice to determine same locations; otherwise, we would need to define Location::equal (another: Location): Boolean, an equality check that would compare the two attributes.

So in some cases "equal" might be defined to mean "identical," but this is by no means general. Suppose you're dining at a restaurant. When your waiter comes up to take your order, you point at the next table and say, "I'll have what she is having." If the waiter interprets your request in terms of object identity rather than your intended "equality" or "similarity," he need not expect a tip from either of you![1]

So even though the concept of object identity is fundamental to the object-oriented world view, there are usually also separate business-defined concepts of similarity or equality that depend on the values of particular attributes. Section 9.7, Templates for Equality and Copying, discusses these concepts in detail and provides templates for their use.

2.3 *Implementations of Object State*

Snapshots describe the information in a system. It might be about a business or a piece of hardware or a software component; we might be analyzing an existing situation or designing a new one. Whatever the case, we'll call the description a *model* and the concrete realization an *implementation*. Notice that this includes both program code and human organization: the implementation of a company model is in the staff's understanding of one anothers' roles. We could do an analysis, abstracting a model of the business by questioning the staff, and then do a software implementation, coding some support tools by implementing the model in C++.

An implementation must somehow represent information pertaining to the attributes of each object to describe its properties, status, and links to other objects. To represent the links between objects, the implementation must also provide a scheme to implement object identity.

2.3.1 Java Implementation

In a Java implementation, every object is an instance of a class. The class defines a set of *instance variables*, and each instance of that class stores its own value for that instance variable, as shown here:

```
class Session {
    // each session contains this data
    Date startDate;
    Date endDate;
    // a client, instructor, and course
    Client client;
```

```
class Client {
    String name;
    int balance;
}
class Instructor {
    String name;
```

[1] Anecdote heard from Ken Auer of KSC.

```
        Instructor instructor;
        Course course;
        // and some status information
        boolean confirmed;
        boolean delivered;
    }
```

```
        char rating;
    }
    class Course {
        String name;
        int maxSize;
    }
```

According to this code, every session (an instance of the class Session) has its own instance variable values for startDate, endDate, client, instructor, and course along with some additional status attributes. A similar approach is taken for the other objects.

Object identity is directly supported by the language and is not otherwise visible to the programmer. Thus, the link from a session to its instructor is represented as a direct reference to the corresponding instructor via the instance variable instructor, implemented under the covers by some form of memory address.

The methods provided by the object will use, and possibly modify, these instance variables. Thus, if Session provides a confirm() method, it may set the confirmed flag and seek an appropriate instructor to assign to itself. The keyword this represents the current session instance that is being confirmed.

```
    class Session {
        ....
        confirm () {
            this.confirmed = true;
            this.instructor = findAppropriateInstructor ();
        }
```

2.3.2 Relational Database Implementation

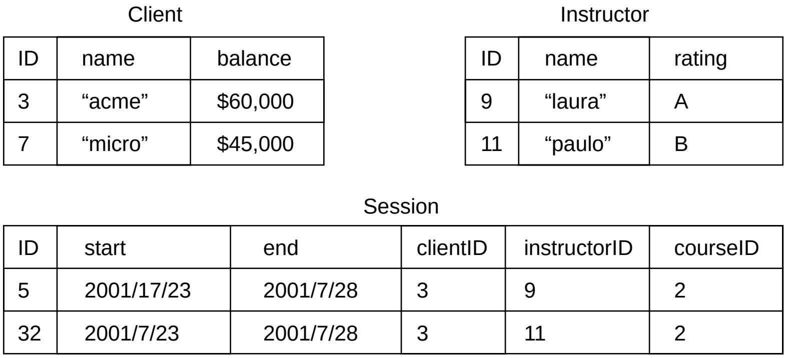
In a relational database, we might have separate tables, Session, Instructor, and Client. Each object is one row in its corresponding table (see Figure 2.4).

Object attributes are represented by columns in the table. Each session, instructor, and client is assigned a unique identification tag, ID, which is used to implement links between the objects. Links between objects are represented by columns that contain the ID of the corresponding linked object.

Object behaviors have no clear counterpart in this world of relational databases, which are concerned primarily with storing the attributes and links between objects. The database can be driven with something such as SQL, but the queries and commands are not encapsulated with specific relations.

2.3.3 Business World Implementation

In a noncomputerized seminar business, all the objects we have discussed still exist but not in a computer system. We might keep a large calendar on a wall, with the sessions drawn as bars and positions on the calendar determining the date attributes. Handwritten client, course, and instructor names would serve as “links”; clients and instructors would be recorded in an address book.



Client

ID	name	balance
3	“acme”	$60,000
7	“micro”	$45,000

Instructor

ID	name	rating
9	“laura”	A
11	“paulo”	B

Session

ID	start	end	clientID	instructorID	courseID
5	2001/17/23	2001/7/28	3	9	2
32	2001/7/23	2001/7/28	3	11	2

Figure 2.4 Object state in a relational database.

If we get two instructors having the same name, we could add their middle initials to remove ambiguity—a scheme for object identity. The balance owed by each client could be written into a ledger or totaled from the client’s unpaid purchase orders. Actions would be procedures followed by the active objects—mostly human roles, in this case—in carrying out their jobs.

2.3.4 Other Implementations

The objects and their attributes are common to all implementations even though the specific representation mechanisms may differ. Even within a specific implementation technology, such as Java, there are many different ways to represent objects and their attributes. For those times when the implementation is as yet unknown or is irrelevant to the level of modeling at hand, we need a way to describe our objects and attributes independent of implementation.

2.4 *Modeling Object State: Types, Attributes, and Associations*

This section explains how to describe objects and their attributes independent of any particular implementation.

2.4.1 Types Describe Objects

Objects and snapshots are concrete depictions, and we will make good use of them in the chapters ahead, but each one shows only a particular situation at a given moment in time. To document a model properly, we need a way of saying what all the possible snapshots are. This is what type diagrams are for (see Figure 2.5).

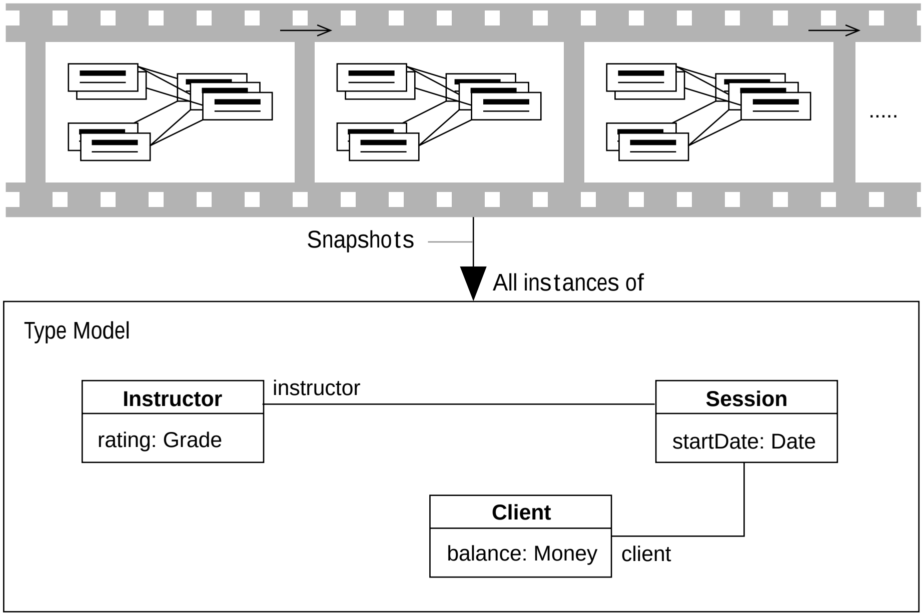
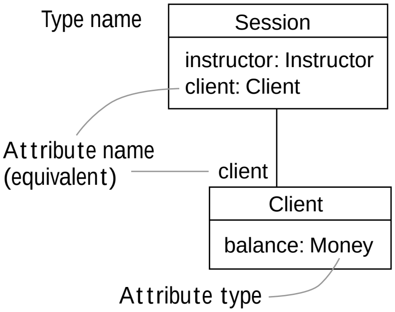


Figure 2.5 Type diagrams generalize snapshots.



The boxes in these diagrams are object types (there is no colon or underlining in the header). A type is a set of objects that share some characteristics—their attributes and behavior—although we'll focus only on attributes for now. The diagram tells us that every Session has a startDate, an attribute that always refers to a Date; and an instructor attribute, which—a piece of imaginative naming, this—always refers to an object belonging to the type Instructor.

Attributes drawn as links on a type diagram are usually called *associations*. In these examples the association labels might seem a bit redundant, but associations are not always named for the type of the target object. We might decide, for example, to have two instructors associated with each session and call them leader and helper.

The attributes in a type model define which snapshots are legal. As shown in Figure 2.6, the course attribute of a session must link to a valid Course. A given snapshot need not depict all attributes of an object, so if you omit an attribute from a snapshot you have said nothing about its value. If you want the attribute to be unconnected, it must be marked with a null value.

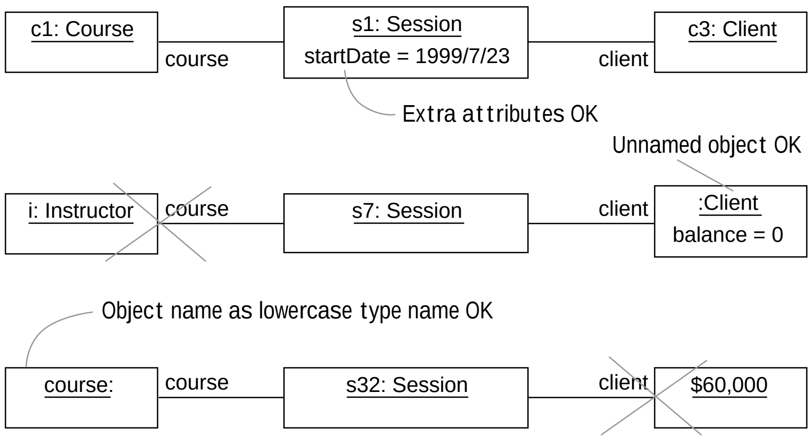


Figure 2.6 Attribute types define some snapshots as illegal.

2.4.2 Attributes: Model and Reality

We've already said that an attribute *need* not correspond directly to stored data in an implementation.[2] A client is described as having a balance, but its implementation could be anything from a number tallied by hand in a ledger to a macro that summed selected purchase orders in a database.

But a model should surely tell us something about the business or design: if we're allowed to do things any old way, what's the difference between a system that conforms to the model and one that doesn't? The practical answer is that the information represented by each attribute should be in there somewhere. It should be possible to write a read-only function or procedure that retrieves the information from whatever weird format the designer has represented it. These *abstraction* or *retrieval* functions are a valuable aid to both documentation and debugging. (You'll find more about this in Chapter 6, Abstraction, Refinement, and Testing.)

The strict answer is that the static model, without actions, does not tell us enough. The only real test is whether the system we're modeling behaves (responds to actions) as a client would expect from reading the whole model, actions and all; the static part merely sets a vocabulary for the rest. This strict view allows some implementations to conform that might not otherwise. For example, suppose we never specified any actions that used the balance. By the retrieval function rule, we would still have to implement that attribute even though it would make no perceptible difference to clients whether or not it was implemented.

The power of seeing attributes as abstractions is that you can simplify a great many aspects of a system, deferring detail but not losing accuracy. The idea corresponds to the way we think of things in everyday life: whether you can buy a new carpet depends, in

[2] Unless you marked the boxes as «classes» when you are documenting your code.

detail, on the history of your income and your expenditure. But it can all be boiled down to the single number of the bank balance: that number determines your decision irrespective of whether your bank chooses to store it as such. The attribute pictures we draw in Catalysis focus on the concepts, which are useful, and not just their implementations, which may be easier done in code.

2.4.3 Parameterized Attributes

Once you realize that attributes do not directly represent stored information, the unconventional concept of a parameterized attribute is a natural extension. A *parameterized attribute* is one that has a defined value for each of many different possible values of its parameter(s). Like attributes generally, it is best thought of as a *query,* or read-only, function that has been hypothesized for some purpose; it need not be directly implemented.

© ***parameterized attribute*** An attribute with parameters such as priceOf(Product). Its value is a function from a list of parameters to an object identity. Unlike an operation, a parameterized attribute is used only as an ancillary part of a state description and need not be implemented directly.

client-3 has a balance due, with amounts due on different dates. Figure 2.7 shows an attribute parameterized by the due date: balanceDueOn (Date): Money; the snapshot explicitly shows attribute values for specific interesting parameter values. Similary, client-3 had a favorite course last year and a (possibly different) one previously; it is modeled by a second parameterized attribute (depicted as a link). Parameterized attributes abstract many implementations,[3] and an implementation must be capable of determining the balance due on any applicable date.

Assuming that objects such as 1997/7/23 and $60,000 have appropriate attribute definitions for isLessThan, you can use this notion of parameterized attributes to write useful statements such as this one:

```
session-5.startDate.isLessThan (today)
session-5.client.balanceDueOn (1998/3/31).isLessThan (someLimit)
```

Here is the same thing in a more conventional syntax:

```
session-5.startDate < today
session-5.client.balanceDueOn(1998/3/31) < someLimit
```

These constraints simply navigate parameterized attributes; we could use the more conventional syntax a < b instead of a.isLessThan(b). Thus, a predefined type Date has a parameterized attribute < (Date): Boolean, which yields true or false for any given compared date.

The primitive types of numbers, sets, and so on can be defined axiomatically—that is, by a set of key assertions about the relationships between them—as outlined in Appendix

[3] Traditional data modeling would use data normalization to define a relationship between Client and Date and describe balanceDue as a relationship attribute; parameterized attributes avoid the need for such data normalization and result in simpler models and more-natural specifications.

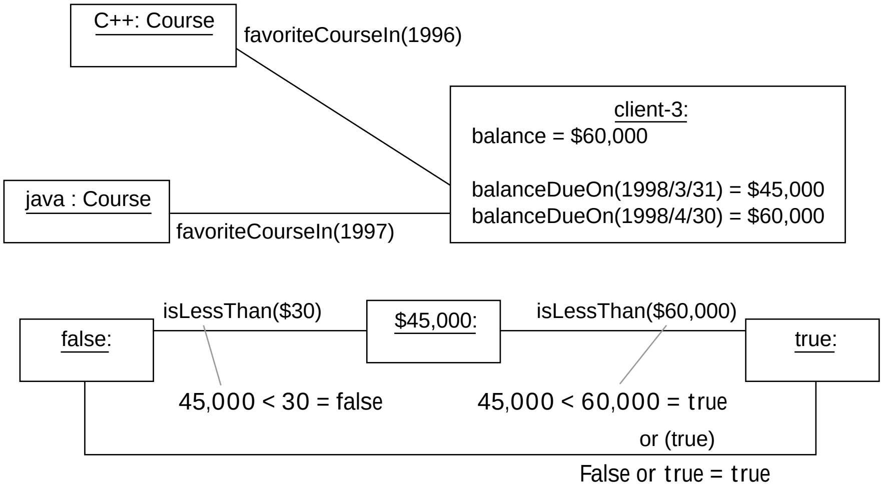


Figure 2.7 Parameterized attributes.

A. Thus, 1 + 3 could be specified by a parameterized attribute, Number:: + (other: Number): Number, with the appropriate definitions constraining this navigation. These definitions can be taken for granted by most users. Other immutable types, such as dates, can be modeled using primitive attributes and operations. The read-only operations of immutable types should not be confused with attributes: the former are publicly accessible in any implementation.

It is easy to envisage both immutable and mutable versions of many types: a Date object whose attributes you can change, or a set you can move things in and out of (see Figure 2.8). Often a reasonable model could be built with either type. However, models of a mutable Date object often should instead use a mutable object, such as Clock, whose today attribute refers to different date objects as time passes.[4] Most interesting domain or business objects are naturally mutable—for example, Customer, Machine, Clock.

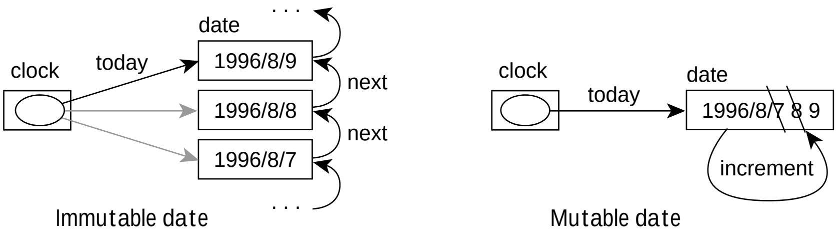


Figure 2.8 Immutable or mutable models of a Date type.

[4] It's much more plausible to say, "The clock stopped" than "Time stopped."

For objects such as dates, we can determine whether d1 < d2 without explicitly storing all the dates that are less than d1 by using a clever representation of dates, such as a single number representing the time elapsed since some reference date. The values of these numbers effectively encode the information about all dates that are less than d1; we simply compare the numbers for the two dates. This technique is often used for a value-type object whose links to other value-type objects are fixed. The 2's complement bit string "0110" in your running program is a clever encoding of a *reference* to the number 6 and, implicitly, to its links to the numbers 5 and 7; the numbers 6, 5, and 7 themselves existed before the bit strings appeared in your program.

2.4.4 Associations

An association is a pair of attributes that are inverses of each other, drawn as a line joining the two types on a type model. For example, each Session has a corresponding Evaluation on completion (not before); each Evaluation is for precisely one Session. This arrangement eliminates certain snapshots, as shown in Figure 2.9.

Drawing an association says more than simply defining two attributes. If defined by two independent attributes, the snapshot in Figure 2.9(c) would be legal. With an association, the attributes must be inverses of each other: s.eval = e if (and only if) e.session = s.

If an association is named only in one direction—for example, eval—then by default the attribute in the opposite direction is named ~eval. If no name is written on the association in either direction, you can use the name of the type at the other end (but with a lowercase letter); the default name for s1.eval would be s1.evaluation. Because two associations can connect the same pair of types, this practice can lead to ambiguity. It is good to name the attributes explicitly in both directions if you intend to refer to them.

© ***association*** A pair of attributes that are inverses of each other, usually drawn as a line connecting two types.

There are several other adornments available for any association (see Figure 2.10).

2.4.5 Collections

Many attributes have values that are collections of other objects. By default, the meaning of the * cardinality is that the attribute is a set, but we can be more explicit about the kind of collection we want, including the following:[5]

- Set: a collection of objects without any duplicates
- Bag: a collection with duplicates of elements
- Seq: a sequence—a bag with an ordering of its elements

For example, each client has some number of sessions, each with one instructor. Each instructor teaches many sessions in a date-ordered sequence. The rating of an instructor is

[5] Based on the Object Constraint Language (OCL) part of UML 1.1.

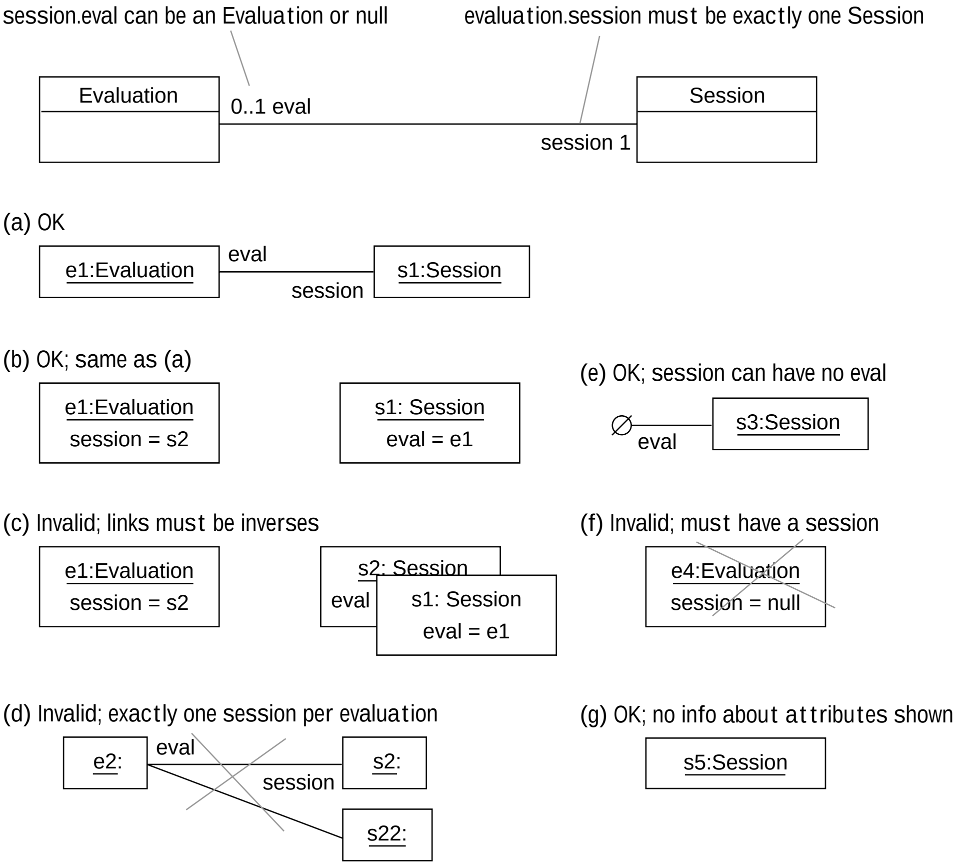
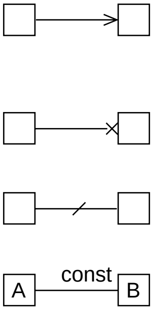


Figure 2.9 Associations define rules about valid snapshots.



In a design or implementation, used to show that navigation in this direction is much easier than in the other. Further explicit annotation can mark it as a stored field.

No defined navigation toward the x; equivalent to an attribute from the right to the left but not left to right.

A derived or redundant association; can be expressed as a function of others.

If a:A, then a.b refers to the same object throughout the life of a.

Figure 2.10 Options for associations.

the average of the rating in his or her last five sessions. The type model is illustrated in Figure 2.11.

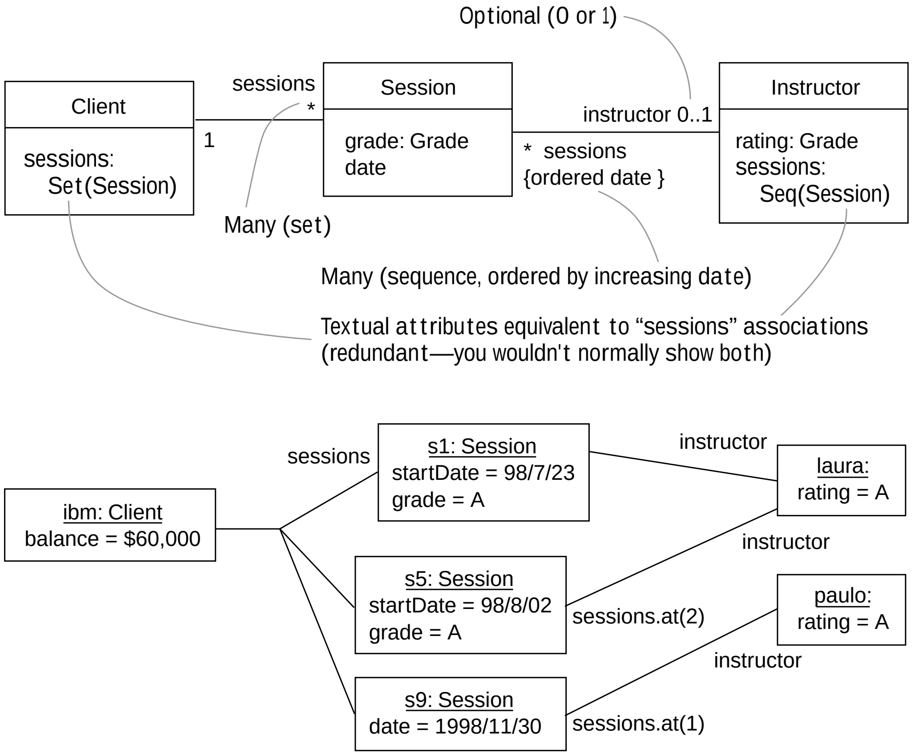


Figure 2.11 Collections: type model and snapshot.

Here are some examples of useful navigations on this snapshot.[6] Try reading them as ways to *refer to* particular objects rather than as operations that are executed on software objects; they are precise, implementation-free definitions of terms. The operators are summarized later.

- The set of sessions for client3, written Set {element1, element2, ...}
  client3.sessions = Set { s1, s5, s9 }
- The instructors who have taught client3.
  client3.sessions.instructor = Set { laura, paulo } -- Sets have no duplicates
- The number of sessions for client3.
  client3.sessions->count = 3

[6] These are also based on the UML's Object Constraint Language.

- Sessions for client3 starting after 1998/8/1; in the following two equivalent forms, the second form does an implicit select from the set.

  client3.sessions->select (sess | sess.startDate > 1998/8/1) = Set { s5, s9 }
  client3.sessions [startDate > 1998/8/1] = Set { s5, s9 }

- Has laura taught courses to ibm (does the set of clients associated with the set of sessions that Laura has taught include IBM)?

  laura.sessions.client -> includes (ibm) -- long version
  ibm : laura.sessions.client -- short version "ibm belongs to laura's sessions' clients"

  Equivalently, has ibm been taught any courses by laura?

  ibm.sessions.instructor -> includes (laura)

(ibm.sessions is a set of sessions; following the instructor links from all of them gives a set of instructors; is one of them Laura?)

- Every one of laura's session grades is better than pass.

  laura.sessions.grade -> forAll (g | g.betterThan(Grade.pass))

- At least one of laura's session grades is a Grade.A.

  laura.sessions.grade -> exists (g | g = Grade.A)

Mathematicians use special symbols to combine sets, but we keep to what's on your keyboard.

- The courses taught by either Laura or Marty:

  laura.sessions.course + marty.sessions.course

- The courses taught by both Laura and Marty:

  laura.sessions.course * marty.sessions.course

- The courses taught by Laura that are not taught by Marty:

  laura.sessions.course – marty.sessions.course

(+, *, and – can also be written -> union(...), -> intersection(...), and -> difference(...).)

A dot (".") operator used on a collection evaluates an attribute on every element of the collection and returns another collection. So laura.sessions is a set of Sessions; evaluating the grade attribute takes us to a set of Grades. If the resulting collection is a single value, it can be treated as a single object rather than a set.

The –> operator used on a collection evaluates an attribute on the collection itself rather than on each of its elements. Several operations on collections—select, forAll—take a *block* argument representing a single argument function evaluated on each element of the collection. Some operators, such as sum and average, are specifically defined to apply to collections of numbers.

Collections are so widely used in modeling that there is a standard package of generic types, extensible by an experienced modeler, as detailed in Appendix A. In Catalysis, collections themselves are immutable objects, although they are not usually explicitly shown on type models. As usual, collection attributes do not dictate an implementation but are used simply to make terms precise; they are an abstraction of any implementation.

2.4.6 Type Constants

It is often convenient to define a fixed object or value and associate it with a particular type.[7] Often, we want to associate the constant with the type of which the constant is a member. For example, the Number type has a constant 0, a number; our Grade type has constants pass, A, B, and so on, each of which is a grade.

A type constant is still an attribute of type members so that all the members share the same constant value; so myGrade.pass = yourGrade.pass. An implementation would store this constant only once rather than in each member of that type.

If you want to refer to a type constant you can use the name of the type without any specific member of that type. Grade is the set of all objects that conform to the Grade type specification; Grade.pass takes you to the single object they're all linked to with that attribute.

Defining a type constant is one of the ways in which it is permissible to mix object instances (usually seen in snapshots) and types. Figure 2.12(a) indicates that following the A link from any Grade always takes you to a specific object, which itself happens to be a Grade. It has an attribute successor, which takes you to the next grade in the list, which happens to be the type constant Grade.B. Instead of drawing the links between the type and the objects, you can write an attribute in the type box with the modifier global, as in Figure 2.12(b).

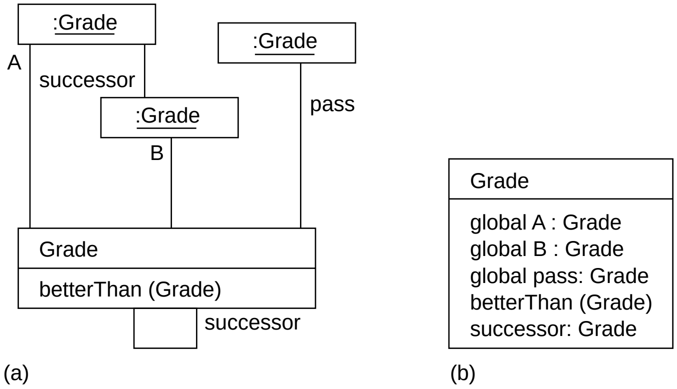


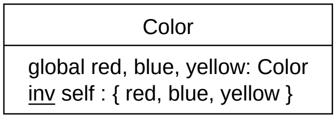
Figure 2.12 Mixing object instances and types.

© ***type constant*** A named member of the type—for example, "7" is a type constant of integer. Type constants can be globally referred to by type_name.member_name.

Type constants can be used to describe what are traditionally treated as enumerated types. To introduce a type Color whose legal values are red, blue, and yellow,

[7] Corresponding to final static class variables in Java (const static pointers in C++).

use three type constants and an invariant specifying that there are no other colors (we permit a syntactic shortcut, enum).


Color
global red, blue, yellow: Color
inv self : { red, blue, yellow }

Be careful not to use type constants when a regular attribute of a higher-level object is needed. For example, rather than use Course::catalog as a global type constant to model the catalog of courses, it is much better to build an explicit object for the seminar company itself and use Company::catalog as a normal attribute on the company.

Use type constants sparingly for mutable types. A type constant says that there is only one object of this name across every implementation in which the type is used. If the shared object itself is immutable, it can be copied permanently in every implementation. For example, the relationship of Grade.A to Grade.B is always fixed, just as with Integer.0 and Integer.1 and Color.red and Color.blue; these shared objects can safely be replicated along with their (immutable) relationships to others. With mutable types you should be prepared to organize worldwide access to the shared object or replicate the object but have a fancy scheme to update the cache.

The global attribute is constant in that it always refers to one object even if the attributes of the target object can change. For example, URL.register could model the unique and mutable worldwide registry of Internet addresses.

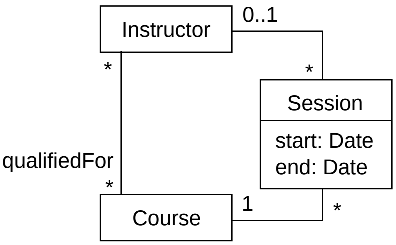
2.5 *Static Invariants*

Not all combinations of attribute values are legal. We have already seen how the type diagram constrains the snapshots that are allowed (Figure 2.6 and Figure 2.9 showed some examples). Those constraints were all about the type of object an individual attribute referred to.

But sometimes we need to disallow certain combinations of attribute values. To do this we can write an invariant: a Boolean (true/false) expression that must be true for every permitted snapshot. (We will scope the snapshots by the set of actions to which this applies in Section 3.5.5, Context and Control of an Invariant.)

© ***static invariant*** A predicate, forming part of a type model, that should hold true on every permitted snapshot—specifically, before and after every action in the model. Some static invariants are written in text; other common ones, such as attribute types and associations as inverse attributes, have built-in notations.

2.5.1 Writing an Invariant



A graphical notation cannot cover all possible constraints and rules. For example, we have several rules about which instructor can be assigned to a particular session: An instructor must be qualified to teach any session he or she is assigned to.

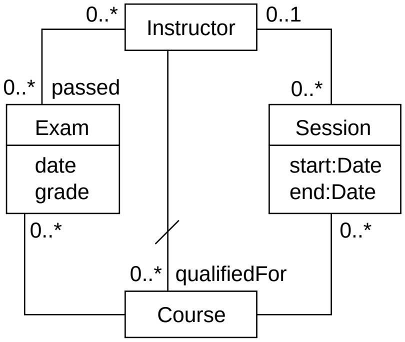
First, we must model the set of courses an instructor is qualified to teach; that has been missing from the model so far. It is easily modeled with a many-many association between instructors and courses.

Based on this model, Figure 2.13 shows a snapshot that we would not want to admit. The problem is that session-25, a Catalysis course, is scheduled to be taught by Lee, who is not qualified to teach it. To ensure that an instructor is never assigned to a session unless qualified to teach it, qualifiedFor—the set of courses you get to by following the qualifiedFor link—includes all the courses of its sessions. Here is how to put it more formally.

inv Instructor:: qualifiedFor -> includesAll (sessions.course)

Or you can write it the other way.

inv Instructor :: sessions.course <= qualifiedFor
--the courses I teach are a subset of the ones I'm qualified for



inv Instructor :: qualifiedFor = exams.course

Notice that we have deferred details of the rules that determine whether an instructor is qualified for a course. These details will have to be captured somewhere, but we might defer it for now if we are not yet considering concepts such as qualification exams and course evaluations.

To add some of these details later, we would enrich the model. Then we can define the less detailed attribute in terms of the new details by using an invariant. Now it's clear that being qualified for a course means having passed an exam for it.

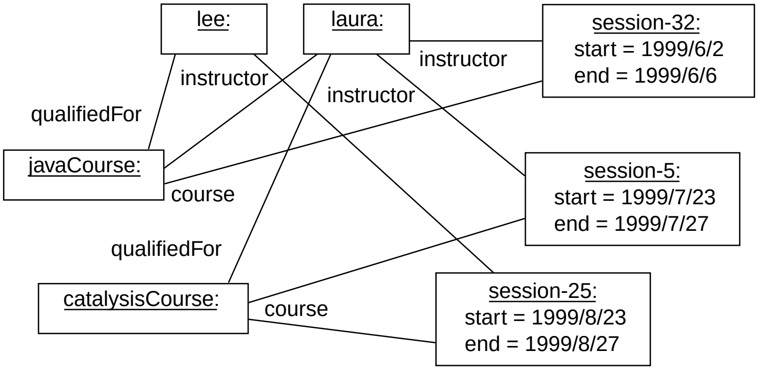


Figure 2.13 An illegal snapshot requiring an explicit invariant.

2.5.2 Boolean Operators

An invariant is a Boolean expression. The usual Boolean operators (as used in programming languages) are available; there are different ways of writing them depending on your preferences. Tables 2.1, 2.2, and 2.3 display them.

Some expressions may have undefined values—for example, attributes of null, daft arithmetic expressions such as 0/0, or parameterized attributes whose precondition is false. Generally, an expression is undefined if any of its subexpressions is undefined. However, some operators do not depend on one of their inputs under certain circumstances: 0 * n is well defined even if you don't know n; so is n * 0. The same applies to (true | b) and (false & b), again no matter what the order of the operands. (This works no matter which way you write the operands—we're not writing a program.)

2.5.3 About Being Formal

We said in the introductory chapters that Catalysis provides for a variable degree of precision, and it is good to repeat it here, among the high-precision stuff. The notation gives you a way of being as precise as you like about a domain, a system, or a component with-

Table 2.1 Boolean Operators

Long	Short	Explanation
and	&	False if either operand is false.
or	|	True if either operand is true.
a implies b	a ==> b	True if whenever a is true, so is b.
not a	! a	
aSet -> forall (x | P(x))	x: aSet :: P(x)	For every member (call it x) of aSet, the Boolean expression P(x) is true.
aSet -> exists (x | P(x))	exist x: aSet, P(x)	There is at least one member (call it x) of aSet for which P(x) is true.

Table 2.2 Collection Operators

Long	Short	Explanation
s1->size		The number of elements in s1.
s1->intersection(s2)	s1 * s2	The set containing only those items in both sets (math s1 ∩ s2).
s1->union(s2)	s1 + s2	The set of all items in both (s1 ∪ s2).
s1 – s2	s1 – s2	Those items of s1 that are not in s2.
s1->symmetricDifference(s2)		Those items only in one or the other.
s1->includes(item)	item : s1	Item is a member of s1.
s1->includesAll(s2)	s2 <= s1	Every item in s2 is also in s1 (s2 ⊆ s1).
s1->select(x|bool_expr)	s1[x | bool_expr]	Filter: the subset of s1 for which bool_expr is true. Within bool_expr, each member of s1 is referred to as x.
s1->select(bool_expr)	s1[bool_expr]	Same as s1[self | bool_expr]. Less general — gets self mixed up with self in the context.
s1.aFunction		The set obtained by applying aFunction to every member of s1.
s1->iterate (x, a= initial value | function_using(x,a)) E.g., scores: Set(integer); -- some attribute; Set(integer) :: average = (self->iterate (x, a= 0 | x+a)) / self->size; scores->average -- meaning now defined		The closure of the function. It is applied to every member x of s1. The result of each application becomes the a argument to the next. The final value is the overall result. Write the function so that order of evaluation does not matter.

Table 2.3 General Expressions in Assertions

Expression	Explanation
let x = expr1 in expression	In expression, x represents expr1's value
Type	The set of existing members of Type
x = y	x is the same object as y

out going into all the detail of program code. But you also have the option to use only informal descriptions or to freely intermix the two. However, it is the experience of many designers who've tried it that writing precise descriptions at an early stage of development tends to bring questions to the fore that would not have been noticed otherwise. Granted, the specification part of the process goes into a bit more depth and takes longer than it takes to prepare a purely text document. But it gets more of the work done and tends to bring the important decisions to the earlier stages of development, leaving the less important detail until later. The extra effort early pays off later in a more coherent and less bug-prone design.

The formal parts are not necessarily readable on their own by the end users of a software product. But the purpose of formal description is not necessarily to be a contract between you and the end users; rather, it is to give a clear understanding between your client, you, and your colleagues of what you are intending to provide. It is a statement of your overall vision of the software, and writing it down prolongs the life of that vision, making it less prone to disfigurement by quick-fix maintainers.

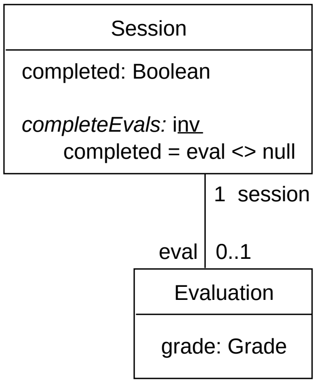
2.5.4 The Context Operator

Some of our examples have attached an evaluation to a session, but this makes sense only after the session has taken place. This rule can be expressed in informal prose; let's make it more precise and testable.

```
-- A session has an evaluation exactly when it is completed.
inv Session:: self.completed = (self.eval <> null)    --'self' is optional
```

The context operator (::) is short for an explicit forall. It says "the following is true for any member of this type (or set), which we'll call self":

```
inv Session  -> forall (self | self.completed = (self.eval <> null))
```



To capture this invariant we must define the term completed; we do so by simply adding a Boolean attribute to Session—and, of course, defining its real-world meaning in the dictionary.

Writing the invariant within the type box is equivalent to writing it separately after a context operator. The predefined notation directly captures certain common invariants. Declaring the course attribute of a Session to be of type Course in Figure 2.5 is equivalent to

```
inv Session:: -- for every session, its course must be an
   object of type Course
self.course : Course -- ":" is set-membership; Course ->
includes (self.course)
```

Similarly, the association shown in Figure 2.9 implicitly defines attribute types and an inverse invariant that is equivalent to this text version:

```
inv Evaluation:: -- for every evaluation
     self.session : Session -- its session must be a Session
     & self.session.eval = self -- whose 'eval' attribute refers back to me
```

2.5.5 Invariants: Code versus Business

An invariant captures a consistency rule about a required relationship between attributes. For example, at the business level, an instructor should never be assigned to a course unless qualified. For a given implementation, this means that certain combinations of stored data should never occur. An invariant representing a business rule, such as *assignQualified*, could look a lot more complex when expressed against an optimized implementation. For example, to efficiently find replacement instructors as availability changes, the assignment of instructors to courses may be represented by a complex data structure indexed by both date and course.

Laura's diplomatically explained position boils down to this: The bottom line when doing type modeling is this: What does a client need to say about a design or requirement? State this using terms natural to the client. Make sure that all the underlying terms are well defined in a glossary. Then make the glossary precise using attributes and invariants in a type model. Restate what you wanted to say more precisely in terms of this type model. Then make sure that your implementation has a consistent mapping to these abstract attributes and invariants.

2.5.6 Invariants in Code

Although an implementation can choose any suitable representation, every attribute in a type model must have a mapping from that representation. Hence, all invariants in a type model will have a corresponding constraint on the implemented state.

Consider the invariant *assignQualified* in Section 2.5.1. Suppose that we choose to represent the qualifiedFor(Course) attribute by storing in each instructor a list of the qualified courses and to represent the sessions attribute by storing a list of sessions:

```
class Instructor {
    Vector qualifiedForCourses;
    Vector sessions;
```

Then the *assignQualified* invariant corresponds to the following Boolean function, which should evaluate to true upon completion of any external operation invocation. Note that the code form is the same as that of the type model invariant, with each reference to an attribute in the type model expanded to its corresponding representation in the implementation.

```
boolean assignQualified () {
    -- for every session that I am assigned to
    for (Enumeration e = session.elements(); e.hasMoreElements;) {
        Course course = ((Session) e.nextElement()). course();
        -- if I am not qualified to teach that course
        if (! qualifiedForCourses.contains (course))
            return false; -- then something is wrong!
    }
    return true;
}
```

The combination of all invariants for a class can be used in a single ok() function, which should evaluate to true after any external invocation of an operation on the object. Such a function provides a valuable sanity check on the state of a running application. Together with operation specifications, it provides the basis for both testing and debugging.

```
boolean ok () {
    assignQualified() = true
    & notDoubleBooked() = true
```

```
            & ......
        }
```

2.5.7 Common Uses of Invariants

There are several common uses of invariants in a type model.

• *Derived attributes:* the value of one attribute can be fully determined by other attributes—for example, the completed attribute in Section 2.5.1. Because an attribute merely introduces a term for describing information about an object and does not impose any implementation decision, we are free to introduce redundant attributes to make our descriptions more clear and concise. However, we define such attributes in terms of others and optionally use a forward slash (/) to indicate that they can be derived from others.

Suppose we need to refer to the clients taught by a given instructor in the past and to the instructors who are qualified candidates for a session. We introduce simple derived attributes on Instructor and Session, defined by an invariant:

```
inv Instructor:: -- clients taught = clients of past sessions I have taught
    clientsTaught = sessions[date < today].client
inv Session:: -- my candidate instructors are those qualified for my course
    candidates = Instructor[qualifiedFor (self.course)]
```

We can now directly use these attributes to write clearer expressions:

```
instructor.clientsTaught... or session.candidates....
```

• *Derived parameterized attributes:* these can also be defined by invariants. If the balance attributes in Figure 2.7 were defined in terms of some session history, we might have the following:

```
inv Client:: -- for every client
    -- the balance due for that client on any date is...
    balanceDueOn (d: Date)
        -- the sum of the fees for all sessions in the preceding 30 days
        = sessions [date < d and d > d - 30] . fees ->sum
```

• *Subset constraints:* the object(s) linked via one attribute must be in the set of those linked via another attribute. For example, the instructor assigned to a session must be one of the candidates qualified to teach that session. This form of invariant is quite common and has a special graphical symbol shown in Figure 2.14. It could have been written explicitly instead:

```
inv Session:: -- my assigned instructor must be one of my qualified candidates
    candidates->includes (instructor)
```

• *Subtype constraints:* a supertype can introduce attributes that apply to several subtypes; each subtype imposes specific constraints on the attributes. An example is illustrated in Section 3.7, Subtypes and Type Extension.

• *State-specific constraints:* being in a specific state may imply constraints on some other attributes of an object. From Section 2.5.1:

```
inv Session:: -- any confirmed session must have an Instructor
```

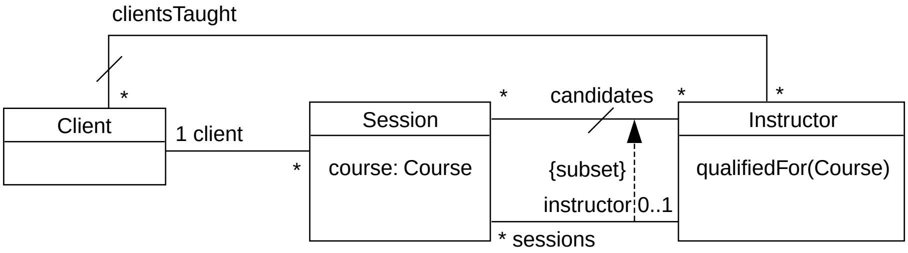


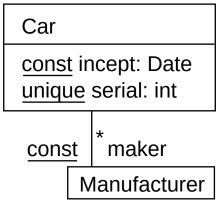
Figure 2.14 Derived attributes and subset constraints.

```
confirmed implies instructor <> null
```

Other forms of invariants are common enough to merit special symbols:

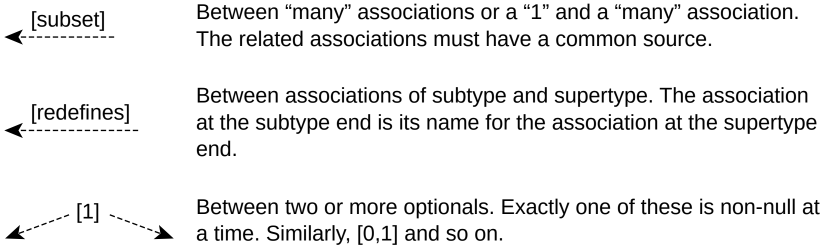
The const attribute refers to the same object throughout the life of the "owner."

The unique attribute means that no other object of this owner type has the same attribute value. unique can apply to a tuple of attributes.



Association constraints (between the ends of two or more associations) are shown in Figure 2.15.

© ***static model*** A set of attributes, together with an invariant, constitutes the static part of a type model. The invariant says which combinations of attribute values make sense at any one time and includes constraints on the existence, ranges, types, and combinations of individual attributes.



Between "many" associations or a "1" and a "many" association. The related associations must have a common source.

Between associations of subtype and supertype. The association at the subtype end is its name for the association at the supertype end.

Between two or more optionals. Exactly one of these is non-null at a time. Similarly, [0,1] and so on.

Figure 2.15 Association constraints.

2.6 *The Dictionary*

When a link is drawn in a snapshot from a course to a session, does it mean that the session has happened, that it will happen, or that it is happening? Or does it mean that it might happen if we get enough customers? Does it mean that this session is an occurrence of that course or that it is some other event, intended for people who have previously attended the course? What is a course, anyway? Does it include courses that are being prepared or only those ready to run?

It's important to realize that the diagrams mean nothing without definitions of the intended meanings of the objects and attributes.

The rather precise notation in this chapter gives you a way of making unambiguous statements about whatever you want to model or design: you can be just as precise with requirements as you can with a programming language in code but without most of the complications. But the notation allows you to make precise relationships only between *symbols*. fris > bee is fearlessly uncompromising and precise in what it states about these two things, provided that someone will please tell us what fris and bee are supposed to represent. We can neither support or refute the statement unless we have an interpretation of the symbols.

Figure 2.16 could just as well be a model of a seminar business, a database, or a Java application. When we describe the *assignQualified* invariant, are we saying that an unqualified instructor never teaches in the business or that some piece of software should never schedule such a thing?

The *dictionary* relates symbolic names to the real world. So if I told you that fris is the name I use for my age and bee is how the age of the current British Prime Minister is referred to in my household, then you could find out whether fris > bee is true or false. If the model in Figure 2.16 were of a database, the dictionary would relate the model elements to tables, columns, and so on.

Suggestive names help, but they can also be misleading because readers readily make silent assumptions about familiar names. Should you be dealing with aileronAngle or fuelRodHeight you might want to be a little more careful than usual about definitions![8]

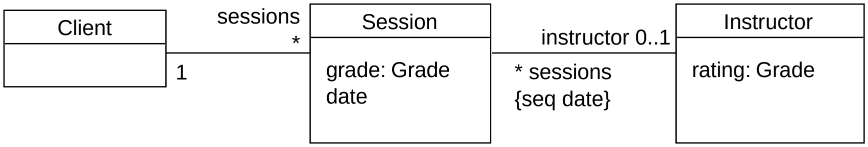


Figure 2.16 Is this a model of a business? a database? an application?

To use a precise language properly, you must first define your terms; once that's done, you can use the definition to avoid any further misunderstandings. There's no avoiding the possibility of mistakes with our dictionary definitions, but we can hope to make them as simple as possible and then get into the precise notation to deal with the complex relationships between the named things.

[8] Safety-critical systems place much more stringent demands on precise definitions.

Type	Description (narrative, with optional formal expressions)		Created by (actions)
	Attr, Inv...	Description (narrative, with optional formal expressions)	written by (actions)
Instructor	The person assigned to a scheduled event		hireInstructor
	rating	*Attribute:* a summary of recent instructor results	deliverCourse, passExam
	sessions	*Attribute:* The sessions assigned this instructor	scheduleCourse
	assignQualified	*Invariant:* Only qualified instructors are assigned to a session **sessions->forall (s I self.qualifiedFor (s.course))**	
Session	One scheduled delivery of a course		scheduleCourse
	date	Start date of the session	rescheduleCourse

Figure 2.17 A typical dictionary.

Our dictionary contains named definitions for object types, attributes (including associations), invariants, action types and parameters, and other elements (which we'll discuss later). A typical dictionary is shown in Figure 2.17. Some of its contents are automatically derived from the models themselves.

© ***dictionary*** The collected set of definitions of modeling constructs. The definitions must include not only the formal modeling and specification bits (relating the formal names and symbols to each other) but also the (usually informal) descriptions that relate the symbols and names to things in the problem domain. Dictionary definitions are scoped according to package scope rules.

2.7 *Models of Business; Models of Components*

So far, we have used static modeling to describe the objects that exist in some world, but we can also use a static model to describe the state of a complete system. We said at the beginning of this chapter that this was the ulterior motive for making a static model. Figure 2.18 shows a model of a simple type.

This is the type of a system or component. The amount represents the money stored inside the machine and credited toward the current sale. We don't know how it is represented inside. Maybe it keeps the coins in a separate container, or maybe it counts the coins as they go into its takings pool. So Money isn't a type representing real coins; rather, it's the type of this component's internal state.

To represent the type of a more complex component, we could just add attributes. It may be easier to do it pictorially. For example, a system that helps schedule instructors for courses would clearly need to know all the concepts we have been discussing in the training business. That knowledge will form the model of the component's state, and we can then define the actions in those terms (see Figure 2.19).

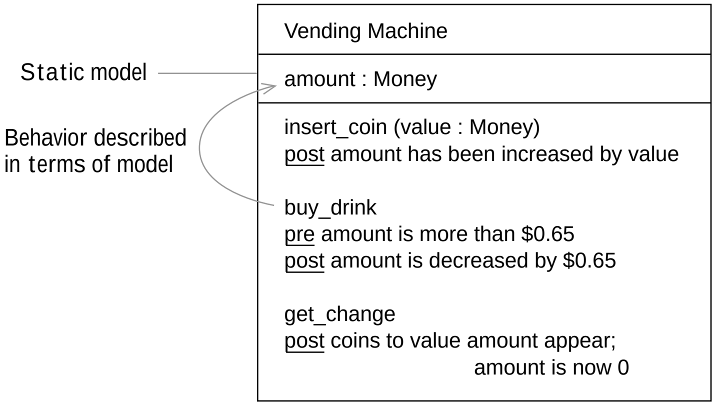

Vending Machine
Static model
amount : Money
Behavior described
in terms of model
insert_coin (value : Money)
post amount has been increased by value
buy_drink
pre amount is more than $0.65
post amount is decreased by $0.65
get_change
post coins to value amount appear;
amount is now 0

Figure 2.18 Simple type model with one attribute.

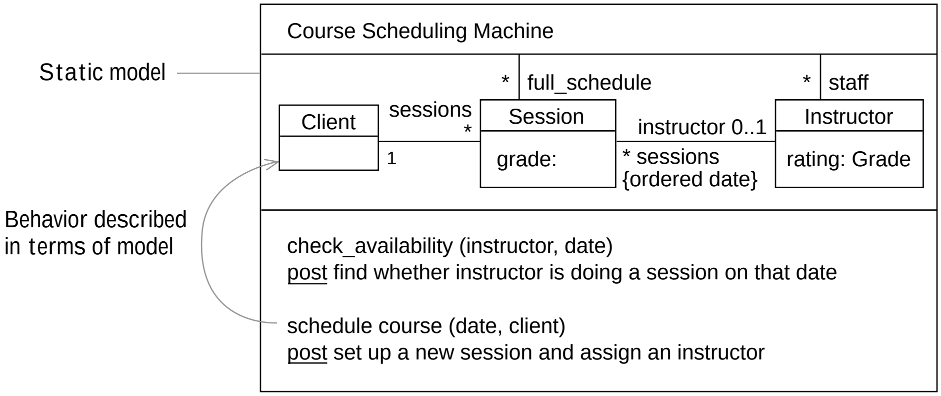

Course Scheduling Machine
Static model
* full_schedule
* staff
Client
sessions
*
1
Session
grade:
instructor 0..1
* sessions
{ordered date}
Instructor
rating: Grade
Behavior described
in terms of model
check_availability (instructor, date)
post find whether instructor is doing a session on that date
schedule course (date, client)
post set up a new session and assign an instructor

Figure 2.19 Complex model—pictorial.

2.8 *Summary*

A static model describes the state of the business or the component(s) we are interested in. Each concept is described with a type, and its state is described with attributes and associations; these lead in turn to other types. The formally defined types are related to the users' world in a dictionary.

Invariants express constraints on the state: combinations of values that should always be observed. They can represent some categories of business rules.

The main purpose of a static model is to provide a vocabulary in which to describe actions, which include interactions in a business, between users and software, or between objects inside the software. We use snapshots to represent specific situations, and that helps to develop the static model (see Figure 2.20). Snapshots are an important thinking

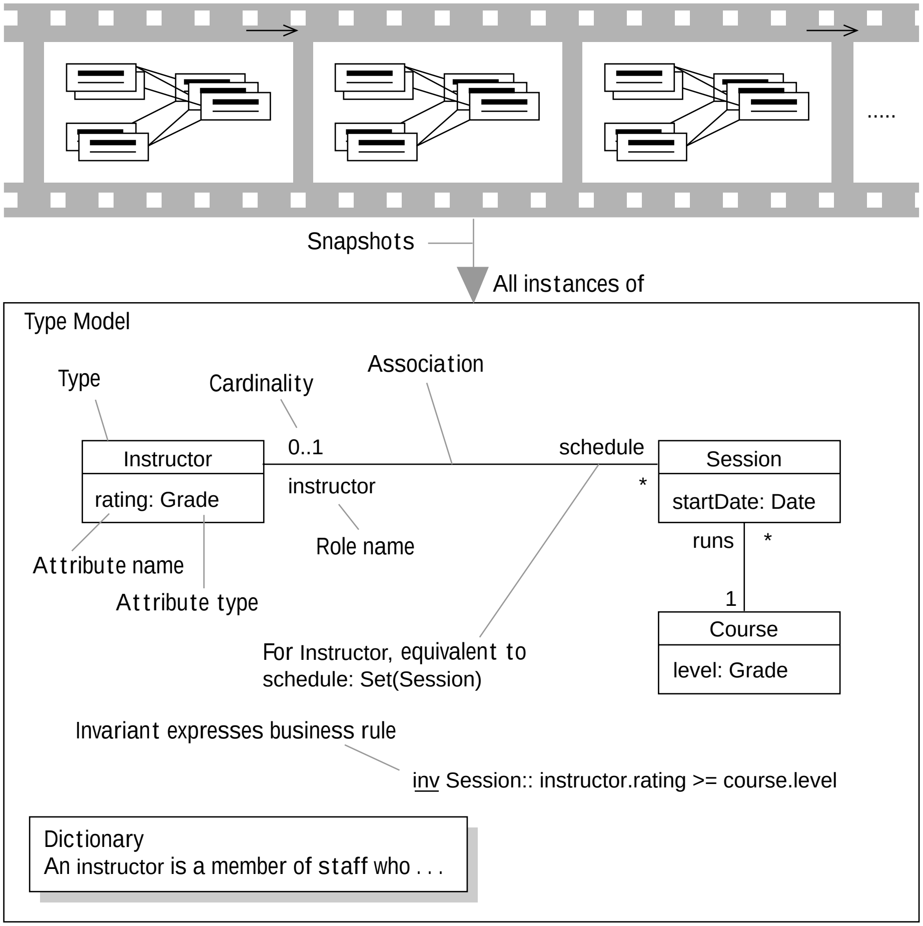


Figure 2.20 Snapshots and the static type model.

tool, but they are not fully general descriptions and therefore play only an explanatory role in documentation.