Chapter 3 Static models — Object Attributes and Invariants

Outline

Models can be divided into static, dynamic, and interactive parts: dealing with what is known about an object at any one moment; how this information changes dynamically with events; and how objects interact with each other.

This chapter deals with the static part of a model, characterizing the state of an object by describing the information known about it at any point in time.

The first section is an overview of what a static model is about. Section 3.2 then introduces objects, their attributes, and “snapshots”, and distinguishes the concept of object identity from object equality.

The attributes that model an object’s state may be implemented in very different ways. Section 3.3 outlines some implementation variations, using Java, a relational database, and a “physical” real-world implementation.

Section 3.4 abstracts from individual objects and snapshots of their attribute values, to a *type-model* which characterizes all objects with (possibly different implementations of) these attributes. It introduces 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 3.5 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.

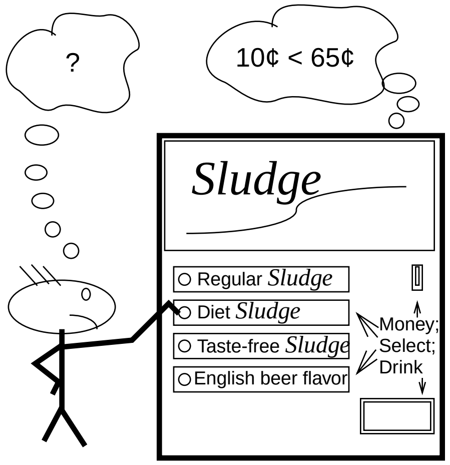
The same model of object types and attributes could describe situations in the real world, for a software specification, or even of code. Section 3.6 introduces the Dictionary as a mechanism for documenting the relation between model elements and what they represent.

3.1 *What is a static model?*

3.1.1 What is ‘state’ about?

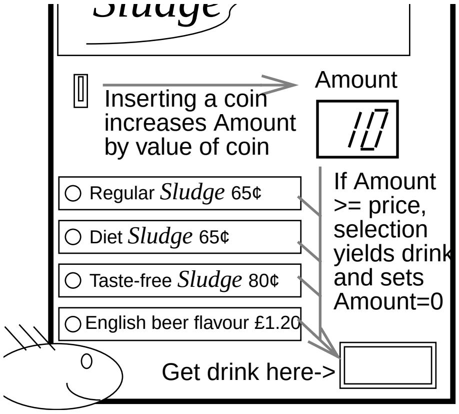
An object’s behavior depends on history.

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, while others on the same night might well turn you back into the cold. The response to your turning up depends on the previous history of your interactions. So it is with so many other encounters in life: the drinks machine that will not yield 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 you have not since switched 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.



State encapsulates history.

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 20p, 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 just say (a) how the outcome depends on current state, and (b) what the new state will be.



State need not be directly observable.

It doesn’t matter much whether the user can observe the state directly — through a display on a machine or by an enquiry 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. 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.

State is independent of implementation.

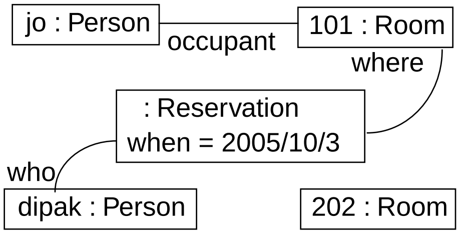
Nor doesn it matter how the state is realised. The hotel reservation might be a record in a computer, or a piece of paper, or a knot in the manager’s tie. The same applies inside software as well: the client objects should not care how an object implements its state. State is a technique that helps document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

In fact, it is quite important that a client should 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. 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 oughtn't to 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.

State is encapsulated in objects.

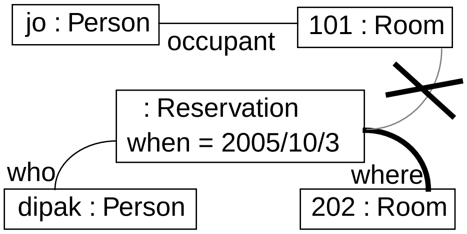
3.1.2 Drawing pictures of states

To illustrate a given state, we use "snapshots". 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.



Snapshots show example states.

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. (Notice that although we say 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.)



Dynamic behavior: changes in the states.

(You might like to draw the effect when Dipak checks in, making a new labeled link and deleting some others. What rule governs where you draw the new link when a person checks in? Can Jo or Chris check in? What must we see on the drawing before a person can check in?)

Exercise

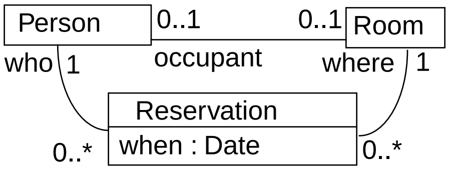
These drawings first and foremost represent states. The same kind of drawing can be used to represent specific implementations of a state: for example, we could decide that each link represents a row in a relation in a relational database; or they might be pointers in main memory; or they might be rows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

Models describe states, not implementations.

3.1.3 Documenting what states are interesting

Static model describes all possible snapshots.

While snapshots illustrate specific example situations, what we need to document are what the interesting and allowed states are: what objects, links and labels are going to be used in the snapshots. This 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 summarises the ways in which rooms and people may be related.



A static model is a glossary.

Most analysts and designers are familiar with the idea of establishing a project glossary at an early stage, so as 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.

3.1.4 Using static models

Static models used in different parts of design

Static models have different uses in different parts of the development lifecycle. 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, formalising the vocabulary. Later in the lifecycle, the objects in the software can be described in the same notation.

Models are abstract.

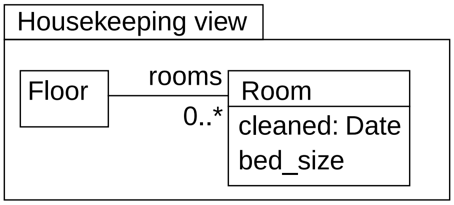
When applied to analysing the real world, modeling is never complete: there's more to say about a Person than which room they're in; every type diagram can always be extended with more detail. This is just as true within software: we saw earlier that as a client of some object, you are interested in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

'Class' and 'type.'

Some tools and authors use the term 'class diagram'. We reserve 'class' for when we're using the diagram at the most detailed level of design, to represent what's actually in the code. A class box (marked «class») shows all the attributes and links its instances have. 'Types' are more general, and represent information that has to be represented somewhere, but we're not telling how. Because this book is about analysis and design, separating the important decisions out from the coding detail, most of the diagrams will be type diagrams.

Views.

A model can just focus on one view: 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 have to combine them at some stage. Conversely, part of our overall implementation might be to divide the system up into components that deal with different aspects, in which case we will have to do the reverse. We'll discuss both these operations later.



Associations and attributes are not program variables.

It's important to realise 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; more of this in Chapter 14, *Refining Models.*

Formal models complement informal narrative.

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 still contain plenty of narrative text in natural language, and illustrative diagrams of all kinds; but the type models are used to make sure there are no gaps or misunderstandings. More on this in Chapter 6, *Documentation Style.*

3.2 *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, and 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.

3.2.1 Objects

Every individual thing is an object.

Anything that can be identified as an individual thing, physical or conceptual, can be modeled as an object; if you can count them, distinguish them from one other, or tell when they are created, they are objects. All of the following are valid objects, drawn as boxes with underlined names for each object; the ":" is optionally followed by a type name for the object, as we explain later.

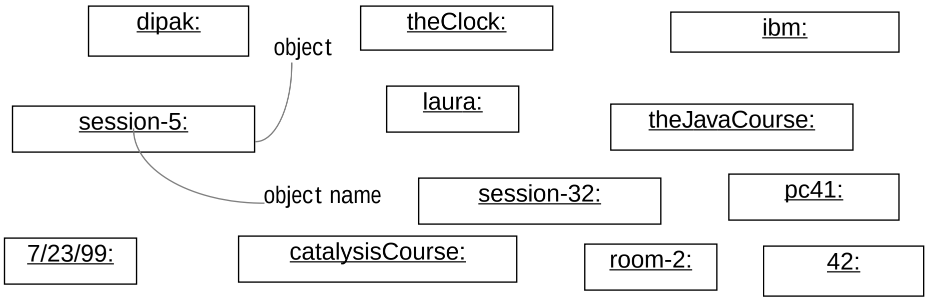


Figure 37: Some objects

Not all objects are interesting.

Of course, not all valid objects are interesting. As we will see, the behaviors that we wish to describe will determine which objects and properties are relevant.

3.2.2 Attributes and Snapshots

An attribute is an abstraction of some object state.

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 we need to say everything we need to say about the object.

For example, session-5 (in Figure 38 below) has attributes of startDate, instructor, course, and client. The value of an attribute is the identity of another object — whether a big changeable object like IBM or a simple thing like 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.

Attributes refer to other objects.

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.

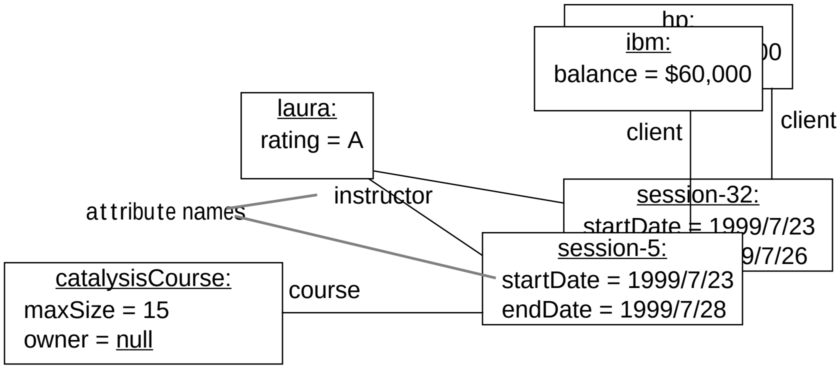


Figure 38: Snapshot depicts attribute values of objects

The predefined name null or ∅ refers to a special object; the value of any unconnected attribute or link is null. In Figure 38, the catalysisCourse does not currently have an owner.

null is a special attribute value

These depictions are called *snapshots* or *instance diagrams*. They are useful for illustrating a given situation, and we will use them for showing the effects of actions.

3.2.3 Alternative ways of drawing a snapshot

The links drawn between the objects and the attributes written inside the objects are actually just 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 attributes where the value is a type of object imported from elsewhere, such as numbers and other primitives. Just to emphasise this point, let’s look at some alternative ways of drawing Figure 38.

Links and attributes are equivalent

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:

Text equivalent

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

A snapshot shows a network of linked objects

Alternatively, we could draw the boxes, but write all the links as attributes inside the boxes:

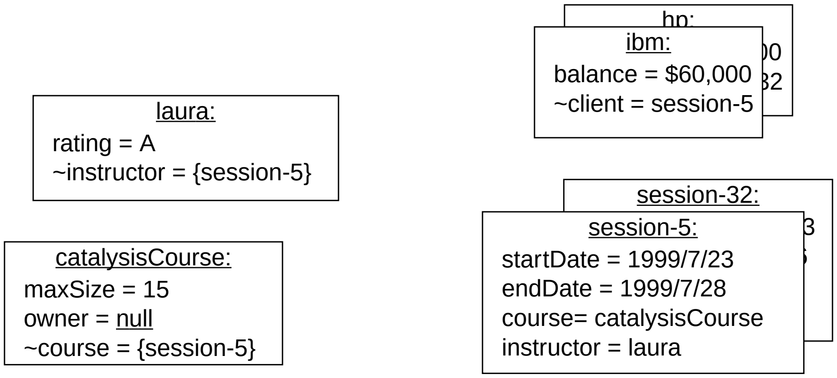
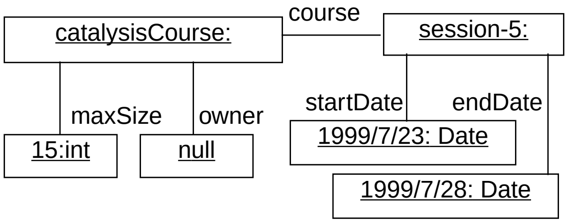


Figure 39: Snapshots with links written as attributes

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 design them into the software directly is another question.

Links depict attribute values in pictures

Now let's go to the other extreme and draw all the attributes as links. Picking out part of Figure 38, 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 where we've 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.



Numbers are objects

We regard as objects, primitive concepts like dates, numbers, and the two boolean values. You can alter an attribute (like maxSize in the example below) 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 etc., but for modeling, there is no point in this extra complication.)

3.2.4 Navigation

"Navigating" object attributes

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

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

In the above expressions, session-5, ibm, 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 “.” Usually the “names” that will refer to objects will be 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.

Expressions refer to objects

Every object has a potentially huge number of attributes. Every Instructor has a pair of shoes, each of which has a colour, each of which may be associated with a set of people who particularly like it. How does one know which ones to document? We’ll deal with this question when we come to *actions*, used to model interesting behaviors; but the short answer is: *only those that help describe the behaviors of interest*.

Not all attributes are interesting

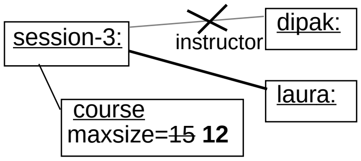
3.2.5 Object Identity

Every object has an identity: that is, some means of identification that allows it to be distinguished from others. This might be realised in all sorts of different ways: a memory pointer, or a database key, or a reference number or name of some sort, or just 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.

Object identity is a fundamental concept.

(If you’ve done any database design, you’ll be familiar with the idea that every entity must have some sort of unique key, which it is up to you to decide. 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, they provide this for you; otherwise, you make it explicit when you get to coding.)

Identity is implicit.



An object identity can be assigned to a suitable attribute or program variable. Changing an attribute value to refer to a different object — 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: the maxSize for the session’s course may change, but that session is still of the same course.

Two different navigation paths, may refer to the same object: e.g. the object referred to as “my boss” may be the same object as that refered to as “my friend’s wife”. In Figure 38, 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.

Different paths may lead to the same object.

“x == y” or “x = y” mean “x and y refer to the *same* object”; “x <> y” means that x and y refer to different objects.

“=” and “<>” are based on identity

But we have to be careful about what relationships like ‘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’ while their courses, dates, and clients are the same. But we know 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.

Distinct objects can be similar or “equal”

Similarity or equality relationships have to 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',
    equal (another:Session) =               -- we 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     )
```

In some cases, equal might be defined to mean identical, but this is by no means general. Suppose in 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”. Now, 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 while 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 10.7, “Templates for Equality and Copying,” on page 419 discusses this in detail, and provides templates for their use.

[1] Anecdote heard from Ken Auer of KSC.

3.3 *Implementations of Object State*

Snapshots describe the information in some system. It might be about a business or a piece of hardware or a software component; we might be analysing an existing situation, or designing a new one. Whatever the case, we'll call the description a 'model', and the concrete realisation an 'implementation'. Notice that this includes both program code and human organisation: the implementation of a company model is in the staff's understanding of each others' 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++.

"Implementation" includes software, hardware, and business.

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 some scheme to implement object identity.

A implementation must represent attributes and identities.

3.3.1 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.

Java uses classes and instance variables

```
class Session {
    // each session contains this data
    Date startDate;
    Date endDate;
    // a client, instructor, and course
    Client client;
    Instructor instructor;
    Course course;
    // and some status information
    boolean confirmed;
    boolean delivered;
}
```

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

Figure 40: Java implementation of object state

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, and some additional status attributes. Similarly 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.

Identity and attributes are directly represented

The methods the object provides will use, and possibly modify, these instance variables. Thus, if Session provides a confirm() method, that method may set the confirmed flag, and seek out 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 ();
    }
```

3.3.2 Relational Data-Base Implementation

A relational database uses tables and columns

In a relational data-base, we might have separate tables, Session, Instructor, and Client. Each object is one row in its corresponding table.

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
22	2001 / 7 / 23	2001 / 7 / 28	3	11	2

Figure 41: Object state in a relational database

Identity and links are indirectly represented

Object attributes are represented by columns in the table. Each session, instructor, and client is assigned a unique identification tag, ID, 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 data-bases, which are concerned primarily with storing the attributes and links between objects. The database can be driven with something like SQL, but the queries and commands are not encapsulated with specific relations.

3.3.3 Business World “Implementation”

A business world uses forms, databases, ...

In a non-computerized seminar business, all the objects we have discussed still exist, except not in a computer system. We may keep a large calendar on a wall, with the sessions drawn on that calendar as bars with the client, course, and instructor names as “links” to these other objects. Clients and instructors are all recorded in an address book.

... with its own scheme for identity, attributes, links, etc.

If we get two instructors with the same name, we may add their middle initial to remove ambiguity — a scheme for object identity. The balance for each client may be written into a ledger, or may be totalled from the unpaid purchase-orders for that client in some folder. And actions are procedures followed by the active objects — mostly human roles in this case — in carrying out their jobs.

3.3.4 And any other implementation...

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 the objects and their attributes. Clearly, we need a way to describe our objects and attributes independent of implementation, for those times when the implementation is as yet unknown, or is irrelevant to level of modeling at hand.

We abstract away implementation variations.

3.4 *Modeling Object State — Types, Attributes, Associations*

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

3.4.1 Types describe Objects

A type model generalizes snapshots

Objects and snapshots are very concrete depictions, and we will make good use of them in the chapters ahead, but each one just shows 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.

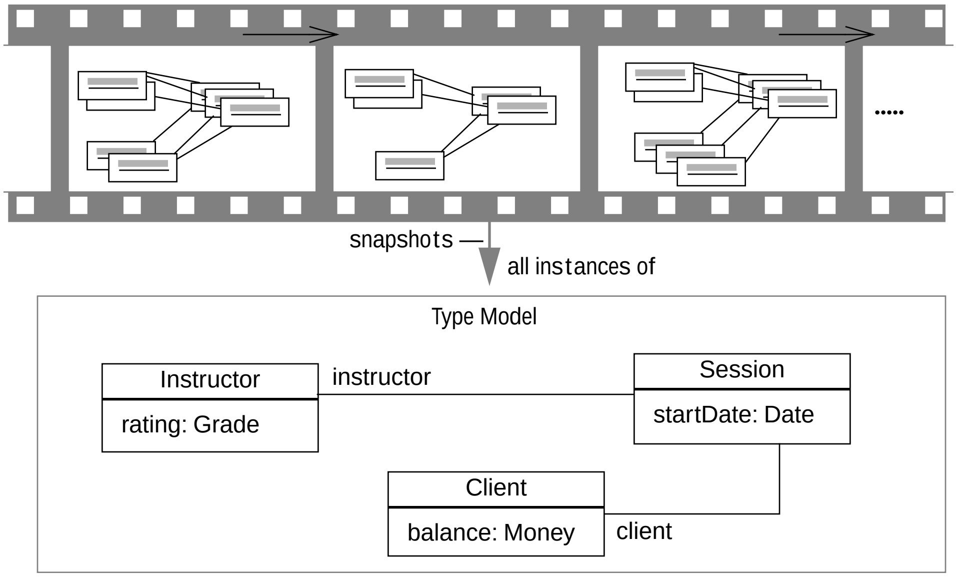
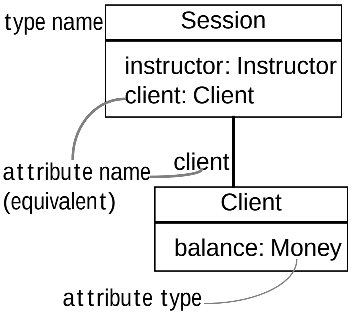


Figure 42: Type diagrams generalize snapshots

A type model summarizes attributes of its members

The boxes in these diagrams are object types (no ":" or underlining in the header). A type is a set of objects that share some characteristics: their attributes and behavior (though we'll just focus 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.



Association names.

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, called leader and helper. But we do sometimes use the convention that, if there is only one attribute linking two types, the attribute name is a lower-case version of the name of the type, and don't bother writing it in.

Attribute types define what snapshots are illegal

The attributes in a type model define which snapshots are legal. As shown in Figure 43, the course attribute of a session must link to a valid Course. A given snapshot need not depict all attributes of an object; unconnected attributes must be marked with a null value.

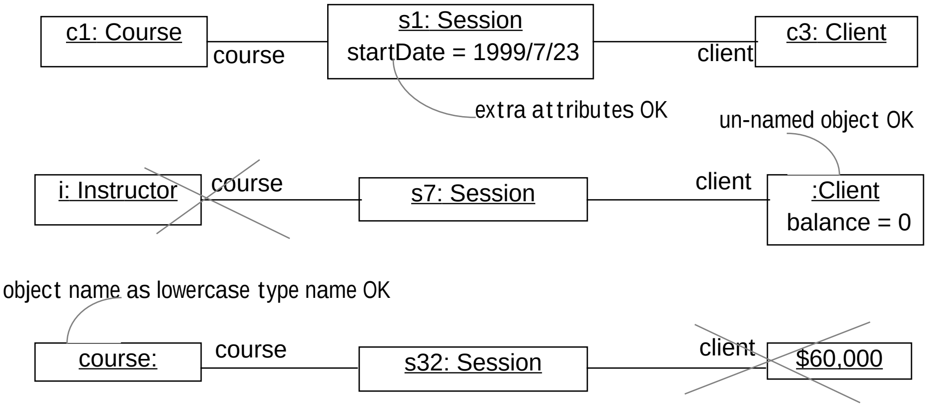


Figure 43: Attribute types define some snapshots as illegal

3.4.2 Attributes: model and reality

An attribute is not necessarily stored information.

We've already said that an attribute *need* not correspond directly to stored data in an implementation[1]. 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 added up selected purchase orders in a data-base.

But a model should surely tell us something about the business or design: if we're allowed to do things any old how, what's the difference between a system that conforms to the model and one that doesn't?

An attribute is an abstraction (*model*) of any representation.

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 wierd format the designer has represented it in. These 'abstraction' or 'retrieval' functions are a valuable aid to both documentation and debugging. (More in Chapter 13, *Refinement*.)

(The strict answer is that the static model, without actions, doesn't tell us anything. The only conformance test is whether the system we're modelling behaves (responds to actions) as a client would expect from reading the whole model, actions and all; while the static part just sets a vocabulary for the rest. This strict view allows some implementations to conform that might not otherwise. For example, suppose we

[1] Unless you've marked the types as «classes», in which case you really are documenting your code.

never specified any actions that used the balance. By the ‘retrieval function’ rule, we would still have to implement that attribute; but it would actually make no perceptible difference to clients whether it was implemented or not.)

The power of seeing attributes as abstractions is that you can simplify a great many aspects of a system, losing detail, but not accuracy. The idea corresponds to the way we think of things in everyday life: whether you can buy a new carpet depends, in 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, quite irrespective of whether your bank chooses to store it as such. The attribute pictures we draw in Catalysis show the concepts — which are useful; and not their implementations — which are less relevant.

An Advanced Topic

3.4.3 Parameterized Attributes

A parameterized attribute value depends on a parameter

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 which 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 is not required to be directly implemented.

It also abstracts many implementations

client-3 has some balance due, but different amounts are due on different dates. As shown in Figure 44, this can be described by an attribute parameterized by the due date: balanceDueOn (Date): Money. The snapshot shows attribute values for specific interesting parameter values explicitly. Parameterized attributes abstract many possible implementations — an implementation must be capable of determining the balance due on any applicable dates.

Similary, client-3 had a favourite course last year, and a (possibly different) one previously. A second parameterized attribute — depicted as a link — models this information.

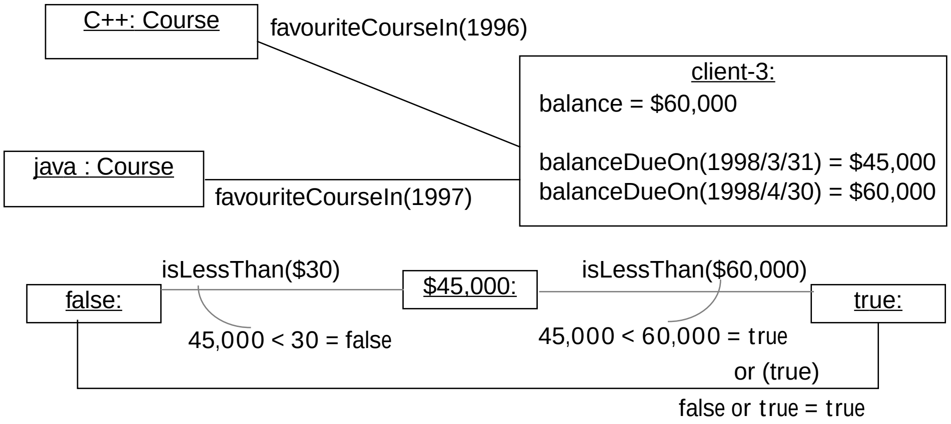


Figure 44: Parameterized attributes

Parameterized attributes provide an effective abstraction tool to avoid the often-inappropriate data-normalization style otherwise required[1].

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 like:

Basic relationships like "<", "and", "or" are parameterized attributes

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

Or, using a more conventional syntax:

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

Clearly these constraints merely navigate parameterized attributes; we could use the more conventional syntax "a < b" instead of "a.isLessThan(b)". Date, a pre-defined type of object, has an parameterized attribute "< (Date): Boolean", which yields true or false for any given compared date. Similarly for the Money type.

Such constraint operators are just attributes on predefined types of objects

The primitive types of numbers, sets, and so on can be defined "axiomatically": that is, by a set of key assertions showing the relationships between their operations, as outlined in the Appendix. They can be taken for granted by most users. Other immutable types e.g. Dates, can be modelled 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.

Even primitive are fully defined within Catalysis

It is easy to envisage both immutable and mutable versions of many types: for example a Date object whose attributes you can change, or a set you can move things in and out of. Often a reasonable model could be built with either. However, people modeling a mutable Date object often really intended to use a mutable object like Clock or Today, that refers to different date objects as time passes.

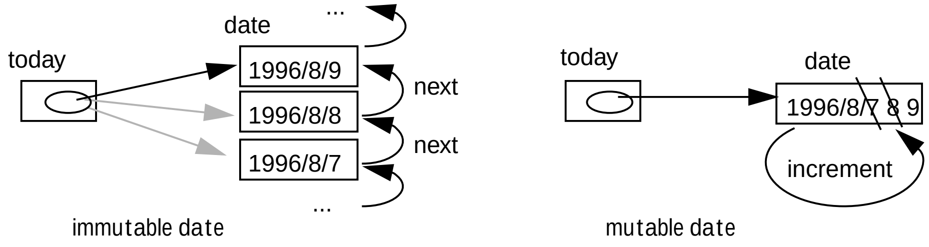


Figure 45: Immutable or mutable models of a Date type

Most interesting domain or "business" objects however, are naturally mutable e.g. Customer, Machine, Clock.

For objects such as dates, we may determine whether d1 < d2 without explicitly storing all the dates that are less than d1, by using a clever representation of dates e.g. a single number representing the time elapsed since some reference date. The values of

Attributes like "<" use efficient encodings

[1] Data normalization might define a relationship between Client and Date, and describe balanceDue as a relationship attribute.

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 any number of other value-type objects are fixed.

3.4.4 Drawing Associations

Associations are inverse attributes drawn pictorially

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 upon completion (not before); each Evaluation is for precisely one Session. This eliminates certain snapshots, as shown in Figure 46.

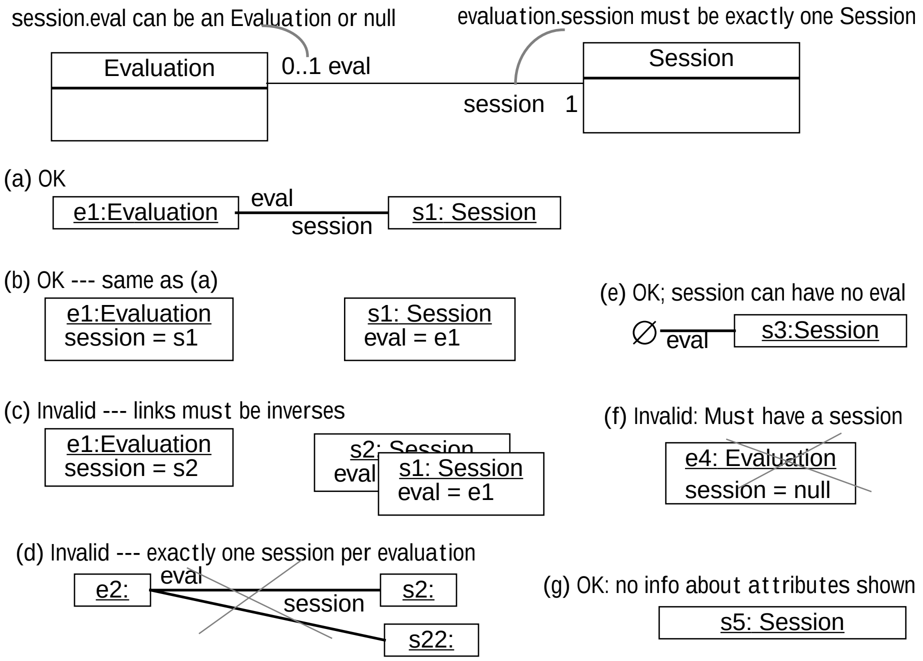


Figure 46: Associations define rules about valid snapshots

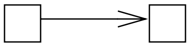
Association attributes are automatically inverses

Drawing an association says more than simply defining two attributes. Two attributes would be independent of each other, and the last snapshot in Figure 46 would be legal. With an association, the attributes are defined to be inverses of each other i.e. s.eval = e if (and only if) e.session = s. Section 3.5 on page 115 will formalize this concept of *invariant*.

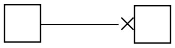
There are default attribute names defined.

If an association is only named in one direction — e.g. eval — then the attribute in the opposite direction is named ~eval by default. If no name is written on the association in either direction, the name of the type at the other end (but with a lower-case initial) can be used — so the default name for s1.eval would be s1.evaluation. But two associations can connect the same pair of types, this can lead to ambiguity. It is good to name the attributes explicitly in both directions if you intend to refer to that direction for any reason.

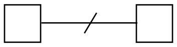
There are several other adornments available for any association:



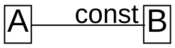
In a design or implementation, used to show that navigation in this direction is much easier than in the other. Further explicit annotation can mark it as a stored field.



No defined navigation towards 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. It should be supported with an invariant.



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

3.4.5 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 explicit about what kind of collection we want. Useful kinds of collections include:

Attribute values can be collections of objects

- 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 the average of the in his last five sessions. The type model is illustrated in Figure 47.

Associations and attributes are defined accordingly

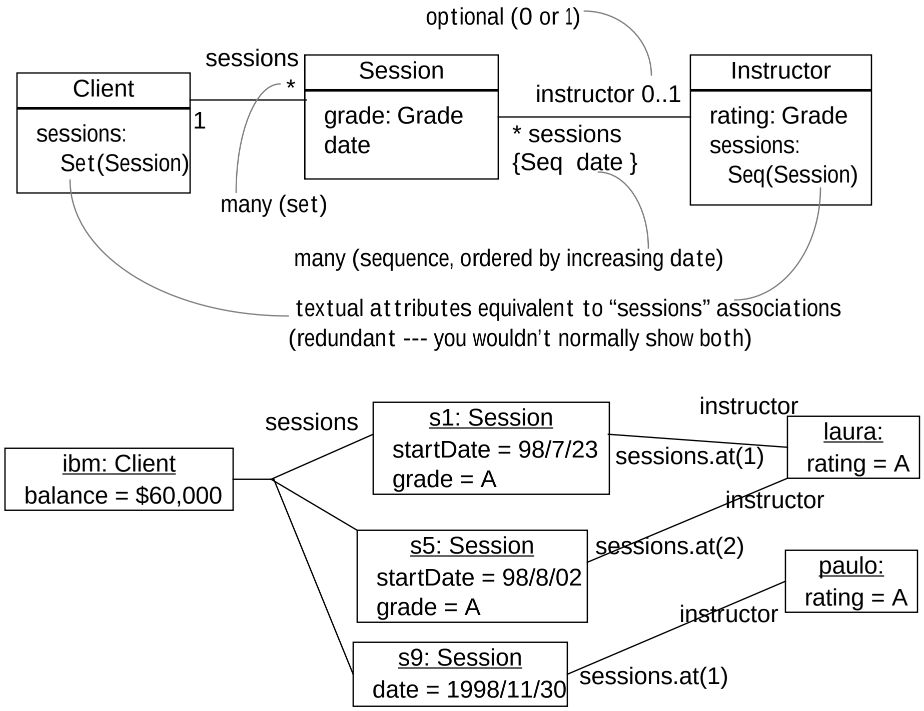


Figure 47: Collections: Type Model and Snapshot

Collections are navigable

Here are some examples of useful navigations on this snapshot. Try reading them as ways to *refer to* particular objects, rather than as operations that are executed on software objects i.e. they are precise, implementation-free definitions of terms. The operators are summarised in Figure 49 on page 117.

- The set of sessions for client3, written: Set {element1, element2, ...}

  client3.sessions = Set { s1, s5, s9 }

- The instructors that have taught client3:

  client3.sessions.instructor = Set { laura, paulo }   -- Sets have no duplicates

- The number of sessions for client3.

  client3.sessions->count = 3

- Sessions for client3 starting after 1998 / 8 / 1; here are 2 equivalent forms, where 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

  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 those 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)

Collection combinations

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 which are not taught by Marty:

  laura.sessions.course – marty.sessions.course

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

We can refer to attributes of a collection *or* its elements

A “.” 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[1] used on a collection evaluates some 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 are specifically defined to apply to collections of numbers — like sum and average.

Collection attributes are still an abstraction of many implementations

Collections are so widely used in modelling that there is a standard package of generic types, extensible by an experienced modeler, as detailed in Appendix A. Collections themselves are immutable objects in Catalysis, 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.

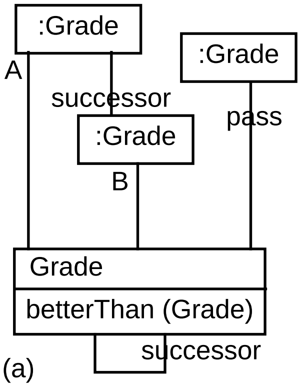
3.4.6 Type constants

A type constant is a shared constant object

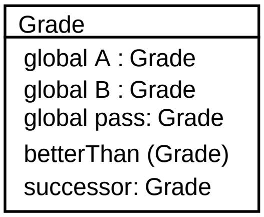
It is often convenient to define a constant object or value, and associate it with a particular type. Often we want to associate the constant with the type of which the constant is a member. For example, the Number type has a constant 0, a number; our Grade type has constants pass, A, B, etc. , each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

It is still an attribute, but with a shared value

A type-constant is still an attribute of type members, such that all the members share the same constant value. An implementation would most likely not store this constant in each member of that type. So session-5.grade.pass = session-23.grade.pass: traversing the pass link from any Grade always takes you to a single object.



(a)



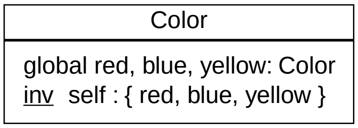
(b)

Defining a type-constant is one of the ways in which it is permissable to mix object-instances (usually seen in snapshots) and types. Figure (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. 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 (b).

If you want to refer to a type-constant but don't have a member of that type handy, just use the name of the 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.

Type constants can describe enumerated types

Type constants can be used to describe what are traditionally treated as enumerated types. To introduce a type Color, whose only legal values are red, blue, yellow, we use three type constants and an invariant saying every color is one of these three.


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

[1] Sorry, but the choice of operator symbol was not ours.

Use type-constants sparingly for mutable types. A type-constant says that there is only one object of this name wherever this type is understood, in every implementation in which it is used. The only practical way to implement this is if the shared object itself is immutable, and can be copied permanently in every implementation: for example, the relationship of Grade.A to Grade.B is always fixed, just like Integer.0 and Integer.1, and Color.red, Color.blue.

An alternative implementation becomes feasible for mutable types when you really are prepared to organise worldwide access to the object. The global attribute is constant in that it always refer to one object; but the attributes of the target object itself can change. For example, URL.register could model the unique and mutable worldwide registry of internet addresses.

3.4.7 Type operators

The Set operators also apply to Types, which are treated as sets of objects. Instead of defining a Type in a box, it is sometimes useful to define it with an expression. This can be written in the Dictionary, or embedded in the narrative of your model:

```
Status = enum { on, off, broke }          -- enumerated type
CollegeMember = Professor + Student
GradStudent = Student – Undergrad
```

3.5 *Static Invariants*

Some attribute combinations are illegal

Not all combinations of attribute values are legal. We have already seen how the type diagram constrains the snapshots that are allowed (Figure 43 and Figure 46 showed some examples). Those constraints were all about the type of object an individual attribute refered to.

But sometimes we need to disallow certain combinations of attribute values. To do this we can write *invariants*.

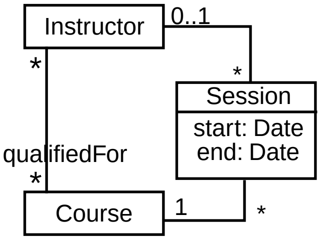
An invariant must be satisfied by every snapshot

An *invariant*[1] is a boolean (true / false) expression which must be true for every permitted snapshot. (We will scope the snapshots by the set of actions to which this applies later, in Chapter 4).

3.5.1 Writing an Invariant

Some invariants are described explicitly in text

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



First of all, we must model the idea of a set of courses that 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 upon this model, here is a snapshot that we would not want to admit:

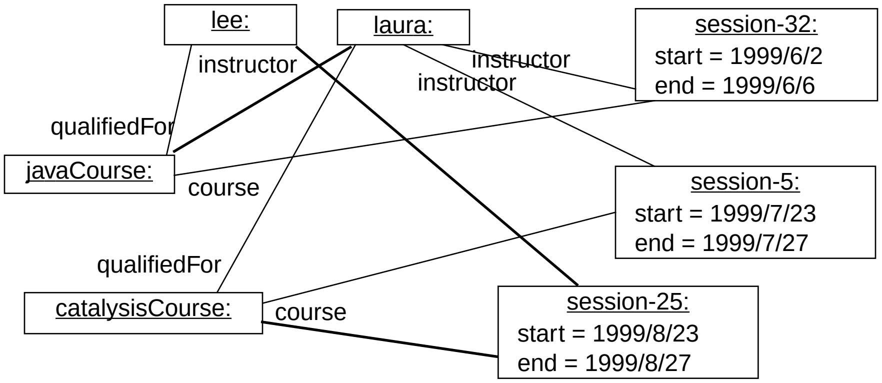


Figure 48: An illegal snapshot requiring an explicit invariant

[1] Specifically, a *static* invariant; we will introduce dynamic invariants later

The problem is that session-25, a Catalysis course, is scheduled to be taught by Lee, who is not qualified to teach it. We'd like to ensure that an instructor is never assigned to a session unless qualified to teach that course i.e. qualifiedFor — the set of courses you get to by following the qualifiedFor link — includes all the courses of its sessions. Put more formally,

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

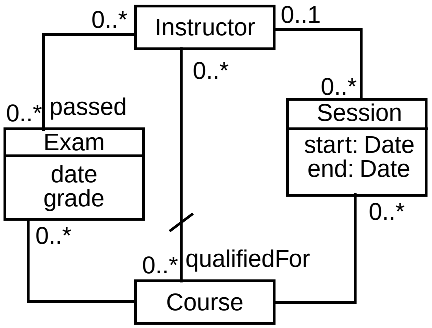
or if you prefer it round the other way,

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

Invariants can abstract away much detail

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

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



inv Instructor ::
qualifiedFor = exams.course

3.5.2 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. Figure 49 displays them. A more detailed explanation is given in [OCL].

Some expressions may have undefined values — for example, attributes of null, or daft arithmetic expressions like 0/0, or parameterised 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 round you write the operands — we're not writing a program.)

Operators in Assertions

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 a is false, False if b is false.
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

Collection operators

s1->size		The number of elements in s1
s1->intersection(s2)	s1 * s2	The set containing just 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 refered 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 such that order of evaluation does not matter.

General expressions in assertions

<u>let</u> x = expr1 <u>in</u> 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

Figure 49: General operators for assertions

3.5.3 About being formal

Formal and informal complement each other

We said in the introductory chapters that Catalysis provides for a variable degree of precision, and it is as well to repeat it here, down among the high-precision stuff. The notation gives you a way of being as precise as you like about a domain or a system or component, without 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.

The precision flags problems early that would otherwise be ignored

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 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 on pays off 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 is not necessarily to be a contract between you and the end-users; 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 mungeing by quick-fix maintainers.

More on how the formal dovetails with the informal in Chapter 6, *Documentation Style*.

3.5.4 The context operator

Some of our examples have attached an Evaluation to a Session; but this only makes sense once the session has actually taken place. This can be said in informal prose; we make it more precise and testable.

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

Read this as: "for every object of type session — let's call it 'self' — it is always true that its eval attribute is not null exactly when its completed attribute is true".

Meaning of ::

The context operator "::" says "the following is true for any member of this type (or set), which we'll call 'self'". self is a special name, that is always the default starting point for navigation expressions. We could have written

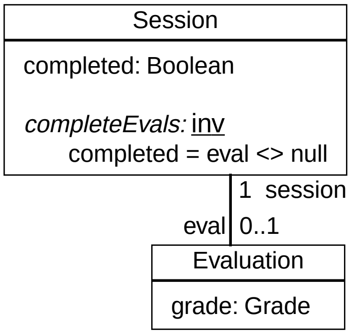
```
inv Session :: completed = (eval <> null)
```

The context operator is short for an explicit forall:

```
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.

You can only refer to known attributes



The invariant is shown on the type model for the type Session: either within the type, or separately after a context operator 'Session::', or in the Dictionary. Recall that attribute types themselves define implicit invariants; explicit invariants play the same role, and are documented together.

The pre-defined notation directly captures certain common invariants. Declaring the course attribute of a Session to be of type Course in Figure 42 is equivalent to:

The notation defines some invariants directly

```
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 46 implicitly defines attribute types and an inverse invariant:

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

Derived parameterized attributes can also be defined by invariants. If the balance attributes in Figure 44 were defined in terms of some session history, we might have:

```
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
```

3.5.5 Invariants: Code vs. 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 busines rule, such as *assignQualified*, could look a lot more complex when expressed against an optimized implementation. For example, the assignment of instructors to courses may be represented by a complex data structure indexed by both date and course, to efficiently find replacement instructors as availability changes.

There are static invariants in the code as well as the business

Here is an perfectly reasonable objection to the type model, debated by *Marty*, a *real good developer*™, and *Laura*, the dreaded *type modeler.*

Lets hear both sides of the picture...

> *Laura:* There! We have a nice precise constraint on session completion.
>
> *Marty:* I see no need for this completed attribute. We already have eval, we can simply check if it is null to determine session completion. It is as fast, and saves space as well....

Laura: Maybe. But the accounts-receivable folks want to see completed sessions.

Marty: Let's tell them to drop completed, and to instead check eval <> null.

Laura: We could try. Do you think they will go for that?

Marty: Bill, over in Accounts, keeps going on about how he really wants to trigger invoicing as soon as a session is completed. He adamantly refuses to hear any thing related to session evaluations. *Oh well. I suppose we could add it in. But don't say I did not tell you about the space cost.*

Laura: Let's step back from implementation for a moment; treat this diagram as a glossary. Bill needs a definition of completed, and the folks in course-QA want eval. So we include both here, related with an invariant. You can still decide how to represent things effeciently...

Implementation must map correctly to requirement

Marty: ..So I can just provide a computed completed? Bill would never know the difference, as long as my implementation is consistent about eval and completed.

Laura: Exactly. And no matter what your implementation, you'd *better* have some mapping for each model attribute.

Marty: Could we still mark this attribute as derived? I know that I will be computing it, not storing it.

Laura: It is derived. Let's write "/completed" since this attribute is fully defined in terms of others. But this is true even if you decided to store it.

Marty (anxious to get back to some real work™): I'm supposed to get the code done and debugged. I'd better get started ...

Laura (uncannily): You can use this invariant as part of your tests and debugging as well. After running any test case just check if any of your session objects are inconsistent with respect to the invariant. Seems trivial for this one, but it could be very useful for more complex ones.

Marty: Come to think of it, the Instructor Scheduling folks caught some nasty algorithm bugs by checking the instructor assignment rules invariants.

(pauses briefly): You really should talk to John and Wilkes about this stuff. I use their communication API, and they keep talking buffers and interrupt handlers. Makes no sense to me. They say they have no choice, writing in assembler for this embedded device and all that. Most distressing. Seems like they could talk to me in my language, instead of those low-level details.

Laura: You bet. Let's have a word with them.

Laura's diplomatically explained position boils down to this:

> What does a "client" need to say about a design or requirement? State this using terms natural to that client. Make sure all the underlying terms are well defined in a glossary. Then make that glossary precise using attributes and invariants in a type model. Re-state what you wanted to say more precisely in terms of this type model. Lastly, make sure your implementation has a consistent mapping to these abstract attributes and invariants.

3.5.6 Invariants in Code

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

Invariants must hold in the implemented code

Consider the invariant *assignQualified* in Section 3.5.1 on page 115. If we choose to represent the qualifiedFor(Course) attribute by storing in each instructor a list of the qualified course, and the sessions attribute by storing a list of sessions:

Depending on the representation...

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

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

...the invariant expands differently in code

```
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 operation on the object. Such a function provides a valuable sanity check on the state of a running application. Together with operation specifications, they provide the basis for both testing and debugging.

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

3.5.7 Common Uses of Invariants

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

1. Derived attributes: the value of one attribute may be fully determined by other attributes e.g. the completed attribute in Section 3.5.1 on page 115. 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

Derived attributes are defined as a function of others such attributes in terms of others, and optionally use a “/” to indicate they can be derived from others.

Suppose we need to refer to the clients that a given instructor has taught in the past, and to the instructors that 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....
```

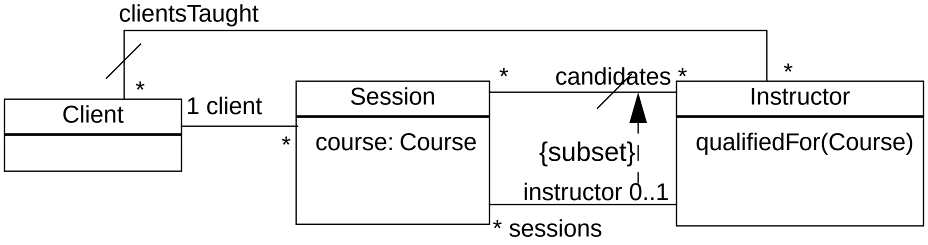


Figure 50: Derived attributes and subset-constraints

Attributes are often constrained to be in some set

2. 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 50. It could have been written out explicitly instead:

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

Ssubtypes can constrain supertype attributes

3. Subtype constraints: a supertype may introduce attributes that apply to several subtypes, where each subtype imposes specific constraints on those attributes. An example is illustrated in Section 7 on page 22.

Constraints may be state-specific

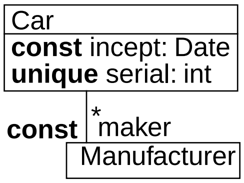
4. State-specific constraints: being in a specific state may imply constraints on some other attributes of an object. From Section 3.5.1 on page 115:

```
inv Session:: -- any confirmed session must have an Instructor
    confirmed implies instructor <> null
```

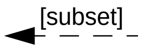
There are some other forms of invariants that are common enough to merit special symbols:

const — This attribute refers to same object through the life of the ‘owner’.

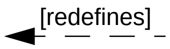
unique — No other object of this owner type has the same attribute value. unique can apply to a tuple of attributes.



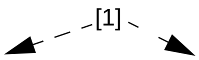
Association constraints (between the ends of two or more associations):



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] etc.

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

Attributes and invariant define static model.

3.6 *The Dictionary*

Pictures and symbols are not self-explanatory!

When a link is drawn in a snapshot from a Course to a Session, does that mean the session has happened, or that it will happen, or 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? And what is a Course anyway? Does it include courses that are being prepared, or just those ready to run?

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

We can be precise about symbols

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 only allows you to make precise relationships between *symbols*. "fris > bee" is fearlessly uncompromising and precise in what it states about these two things — provided someone will please tell us what fris and bee are supposed to represent. We cannot either support or refute the statement unless we have an 'interpretation' of the symbols.

Figure 51 could just as well be a model of any of a seminar business, or 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 itself never schedule such a thing?

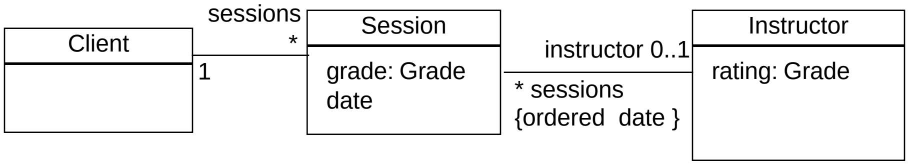


Figure 51: Is this a model of a business? Database? Application?

The Dictionary relates symbols to the world

The *Dictionary* relates symbolic names to the real world. So if I told you 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, *now* you can definitely find out whether fris > bee is true or false. If the model above was of a database, the dictionary would relate the model elements to tables, columns, etc.

Suggestive names help, of course. But they can also be misleading since 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![1]

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

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

The Dictionary contains named definitions for object types, attributes (including associations), invariants, action-types and parameters, and more (which we'll discuss later). A typical dictionary is shown below; some of its contents are automatically derived from the models themselves:

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:* an summary of recent instructor results	deliverCourse, passExam
	sessions	*Attribute:* The sessions assigned this instructor	scheduleCourse
	assignQualified	*Invariant*: Only qualified instructors assigned to a session sessions->forall (s | self.qualifiedFor (s.course))	
Session	One scheduled delivery of a course		scheduleCourse
	date	Start date of the session.	rescheduleCourse

3.7 *Models of business; models of components*

So far, we have used static modeling to describe what objects there are 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 that was really the ulterior motive for making a static model.

Here is a model of a simple type:

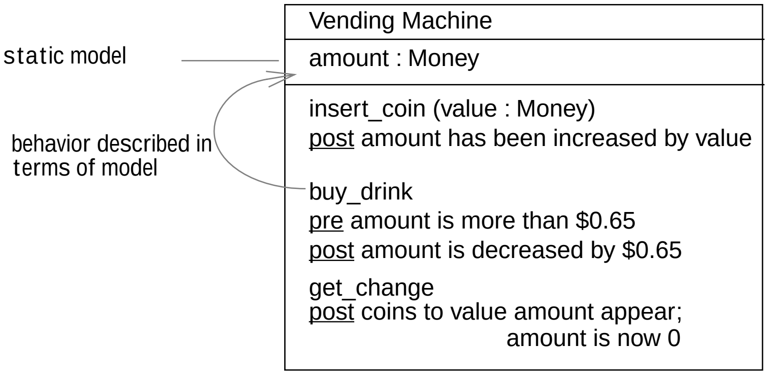

static model
behavior described in terms of model
Vending Machine
amount : Money
insert_coin (value : Money)
post amount has been increased by value
buy_drink
pre amount is more than $0.65
post amount is decreased by $0.65
get_change
post coins to value amount appear;
amount is now 0

Figure 52: Simple type model — one attribute

This is the type of a system or component. The ‘amount’ represents the money stored inside the machine towards the current sale. We don’t know how it is represented inside — maybe it keeps the coins in a separate container; or maybe it just counts the coins as they go into its takings pool. So Money isn’t a type representing real coins: it’s the type of this component’s internal state.

To represent the type of a more complex component, we could just add attributes. But the easier way is 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 will form the model of the component’s state, and we can then go ahead and define the actions in those terms.

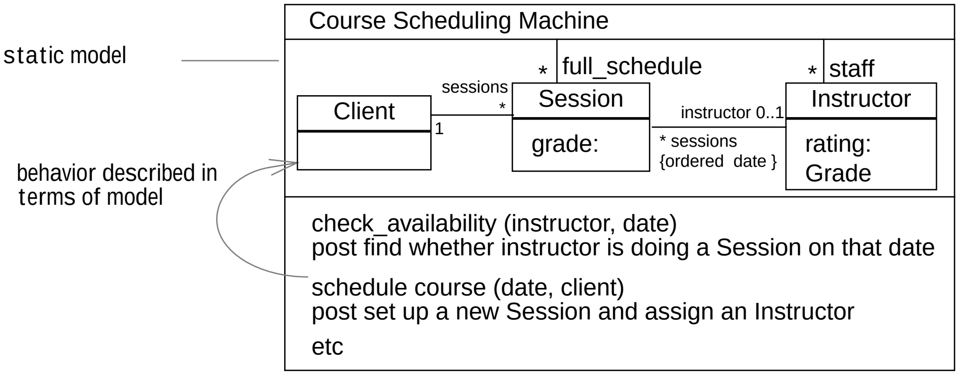
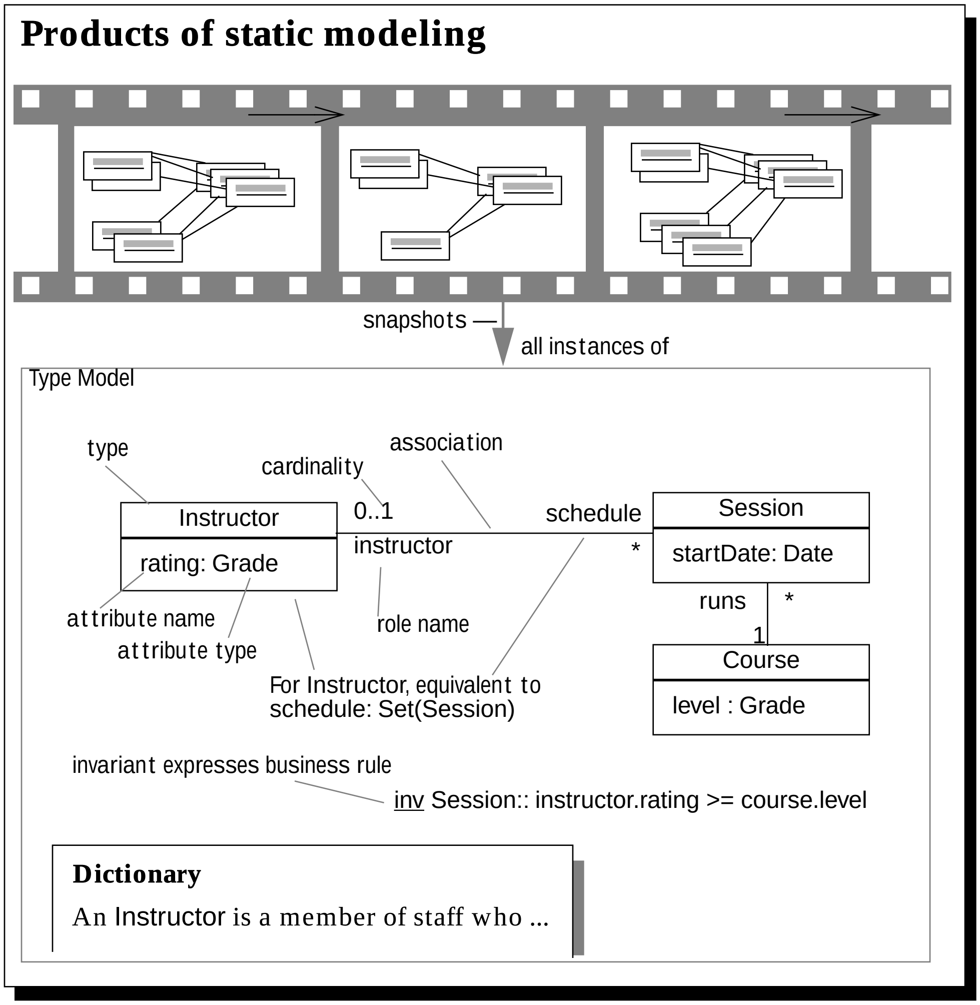

Course Scheduling Machine
static model
behavior described in terms of model
* full_schedule
* staff
Client
sessions *
1
Session
grade:
instructor 0..1
* sessions {ordered date }
Instructor
rating: Grade
check_availability (instructor, date)
post find whether instructor is doing a Session on that date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

Figure 53: Complex model — pictorial

3.8 *Static Models — 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 the 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, or between users and software, or between objects inside the software.
- We use snapshots to represent specific situations, which helps to develop the static model. Snapshots are an important 'thinking tool', though are not fully general descriptions, and therefore play only an explanatory role in documentation.


Products of static modeling

Dictionary

An Instructor is a member of staff who ...

Figure 54: Static models