

Lecture 7: Object Modelling Notation

7.1 Quote of the Day

The demand for software has grown far faster than our ability to produce it. Furthermore, the Nation needs software that is far more usable, reliable, and powerful than what is being produced today. We have become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways.

Information Technology Research: Investing in Our Future
President's Information Technology Advisory Committee (PITAC)
Report to the President, February 24, 1999
Available at <http://www.ccic.gov/ac/report/>

7.2 Context

What you'll learn: the basics of object models, a notation for representing state relationally.

Why you should learn this: this kind of notation has become universal, so as a software designer or implementor, you need to be able to read it; in designing your own systems, object modelling gives you a real edge.

What I assume you already know: what sets and relations are.

See the 'Object modelling crib sheet' available online for a summary of the notation.

7.3 A Relation View of State

When we talked about specifications, I introduced the idea of viewing the state of an executing Java program in terms of relations. This idea is in fact quite fundamental, and has much wider application. Not only program state, but also the state of the world can be expressed relationally. This viewpoint became popular with relational databases, but it is now central to object oriented development too.

Almost all software development methods have a notation for describing state relationally. These notations derive from the *entity relationship diagram*, which was the earliest form of diagram used to represent the relations in a problem domain, as the first step in designing a database. Almost all object-oriented programmers use class diagrams to sketch the structure of their code; although these at first appear just to be a diagrammatic form of code, they are in fact relational too. So understand relational notation is an issue of basic literacy for a software developer.

We'll use the common term *object model* to describe the diagram that represents a system's state relationally. Object models can be constructed at different levels of abstraction, and of different things. In tomorrow's lecture, we'll look at how object models are used to describe the world around the system – the problem domain. And in Wednesday's lecture, we'll see how they are used to describe code.

When you design a software system, you need to be able to articulate your ideas before they appear in their final form in code. Many developers just sketch class outlines, but this is a cumbersome way to work, and obscures the key structures. You experience this yourself as you saw me present the outlines of classes for an instant messaging server last week. I suspect you felt dissatisfied by my piecemeal presentation, and wanted something that gave a better global overview. That's what the object model does, and we'll look at an object model for that system tomorrow.

The object model helps you get at the central issues quickly and easily. There are few design tools that are as easy to use and as helpful.

7.4 Atoms, Sets and Relations

The structures of our models will be built from sets, relations and atoms. An atom is a primitive entity that is

- *indivisible*: it can't be broken down into smaller parts;
- *immutable*: its properties don't change over time; and
- *uninterpreted*: it doesn't have any built-in properties, the way numbers do, for example.

Elementary particles aside, very few things in the real world are atomic. But that won't stop us from modelling them as atomic. In fact, our modelling approach has no built-in notion of composites at all. To model a part x that consists of parts y and z , we'll treat x , along with y and z , as atomic, and represent the containment by an explicit relation between them.

A set is just a collection of atoms, with no notion of repetition count or order. A relation is a structure that relates atoms. Mathematically, it's a set of pairs, each pair consisting of two atoms, in a specified order. You can think of a relation as a table with two columns, in which each entry is an atom. The order in which the columns appear is important, but the order of the rows is irrelevant. Each row must have an entry in every column.

There are also relations with more than two columns. A ternary relation has three columns, and is a set not of pairs but of *triples*.

7.5 Classification

Each set in an object model is denoted by a box labelled with the name of the set.

The sets of an object model are organized by *classification*. A ‘fat arrow’, which has a large, unfilled triangle as its arrowhead, denotes the subset relationship: an arrow from set A to set B says that A is a subset of B: every element of A is also an element of B. This relationship is sometimes called is-a, because the arc can be read ‘every A is a B’.

By default, all the subsets of a set are mutually disjoint – that is, they have no elements in common. To indicate that a subset is not necessarily disjoint from the others, the arc is marked with the keyword *in*, and arcs representing subsets that are mutually disjoint are marked *extends*. The keyword *extends* is the default, and is assumed when omitted.

If the label is prefixed by the keyword *abstract*, or is italicized, the set is exhausted by the subsets that extend it. This means that there are no elements that belong only to the superset and to none of the extending subsets.

Examples. (Written textually, but drawn in lecture. Don’t you wish you’d been there?).

A file system:

```
set FileSystemObject
set Directory extends FileSystemObject
set File extends FileSystemObject
set Link extends File
set Root extends Directory
```

A company’s organization:

```
set Employee
set Manager extends Employee
set RecentHire extends Employee
set Department
```

A linked list implementation:

```
set Object
set List extends List
set EmptyList extends List
set NonEmptyList extends List
```

Note that the classification of atoms into sets can be *dynamic*: an atom’s membership can change over time. For example, whether an Employee belongs to the set RecentHire will change as time passes.

7.6 Multiplicity Keywords

For both sets and relations, we'll want to express some fundamental counting properties. For this, we define some useful keywords:

lone ? zero or one
some + one or more
one ! exactly one
set * any number

Each keyword is followed by an abbreviation: thus ?, for example, is short for lone. You can think of lone as short for 'less than or equal to one'. The first three of these can also be used as quantifiers when writing constraints, in addition to all and no. For example,

lone x: e | F

says that there is at most one value of x, an element drawn from the set e, such that F holds.

Note that set isn't really a multiplicity at all: it just represents the absence of a constraint, and is used because the default is sometimes one.

7.7 Multiplicities Applied to Sets

Multiplicities can be used to indicate the size of a set. You either write the keyword before the label, or the symbol after it. If missing, the default is set. A set with multiplicity of lone or one can optionally be drawn as an oval rather than a rectangle; the default multiplicity for an oval is one.

Examples.

A file system:

```
set FileSystemObject
set Directory extends FileSystemObject
set File extends FileSystemObject
set Link extends File
one Root extends Directory
```

An application's undo feature:

```
set Command
lone LastCommand extends Command
```

7.8 Relations

Thin arrows, which have small, filled triangles as their arrowhead, represent relations. An arc with a thin arrow from set A to set B denotes a relation whose domain is contained in A and whose range is contained in B. The label gives the name of the relation. The label can include multiplicity keywords; a label of the form

$m R n$

on an arc from A to B says that each atom in A is mapped by R to n atoms in B, and m atoms in A map to each atom in B.

(By the way, you've already seen this notation textually, when we wrote – in the lecture notes on specification – the declaration

(Object -> **lone** Object) map;

to say that the specification field map of a hash table, for example, maps each key to at most one value. This declaration takes the form

(A m -> n B) R;

In the Alloy modelling language, from which this notation derives, the declaration is written

R: A m -> n B

which is more similar to conventional mathematical usage.)

The default multiplicity is always set, so if no multiplicity symbols are used, there are no implicit multiplicity constraints.

(In class we talked about how certain multiplicities correspond to standard mathematical terminology: ->? for a partial function, ->! for a total function, ?->? for an injection, !->! for a bijection.)

An arc can have multiple labels. This is one reason that it's convenient to make the multiplicity symbols part of the label and not write them as annotations on the ends of the arc: two relations with the same domain and range can share the same arc even if they have different multiplicities.

Examples.

A file system:

```
set FileSystemObject
set Directory extends FileSystemObject
set File extends FileSystemObject
set Link extends File
one Root extends Directory
```

dir: FileSystemObject ->**lone** Directory
to: Link ->**one** FileSystemObject

(We had some discussion in class about the file system example. It's an interesting one, because different file systems enforce different invariants. In Unix, a file can be in multiple directories, because a hard link is indistinguishable from the file it points to. Some students wanted to regard the presence of 'dot' in a Unix directory as self-containment, which would imply that every directory has a parent. I'm not convinced by this though; dot is more of a convenience for the syntax of shell commands than a real self-containment.)

A company's organization:

set Employee
set Manager **extends** Employee
set RecentHire **extends** Employee
set Department
contains: Department **lone** -> Department
runs: Manager **one** -> **some** Department
worksFor: Employee -> **one** Manager
worksIn: Employee -> **one** Department

A linked list implementation:

set Object
set List **extends** List
set EmptyList **extends** List
set NonEmptyList **extends** List
element: NonEmptyList ->**one** Object
next: NonEmptyList **lone**->**one** List

An application's undo feature:

set Command
lone LastCommand
precedes: Command **lone**->**lone** Command

7.9 Semantics

What exactly is the meaning of an object model? An object model denotes a set of *snapshots*. Each snapshot is a graph consisting of some atoms, which we can draw as round nodes, and labelled arcs between them representing relationships. The nodes are labelled with names of sets that the atom belongs to. Although each snapshot is finite, an object model will in general denote an infinite set of snapshots.

(In class, we drew a snapshot for the file system example, and considered a small change might invalidate it, making it no longer a snapshot) of the object model.

7.10 Expressions and Higher-Arity Relations

You can show any expression that denotes a set or a binary relation in a relational diagram. Just use the expression as the label of the node or arc instead of a set or relation name.

A relation with more than two columns cannot be shown directly. But you can usually say everything you need to say by creating one or more arcs and labelling them with appropriate expressions. For example, suppose you have a relation

R: A \rightarrow B \rightarrow C

which relates atoms in A, B and C. You might show this as an arc from B to C labelled A.R; or from A to B labelled R.C.

To show the multiplicity of a relation like R, however, what you often want is instead to show an archetypal expression such as a.R, where a is a scalar in the set A. Labelling a relation arc

expr1 (var: expr2)

means that whatever constraints are implicit in the arc apply for the expression expr1 for whatever value var takes from the set given by expr2. So, for example, an arc labelled

a.R (a: A) !

(consisting of such a label combined with a multiplicity marking) from B to C says that a.R is a function that maps every B to one C, for every a.

In practice, the bound is almost always over a named set, and this notation is a bit clumsy. So as a convenient shorthand, a variable that ranges over the set S is written instead as <S>. For the last example, then, we'd write the label just as <A>.R.

Examples.

Suppose you want to model the relationship between a person, a company she works for, and her salary. If she works for more than one company, a single binary relation won't work. You could either model it by introducing Job as a set:

```
set Job
set Person
set Company
set Salary
company: Job ->one Company
```

salary: Job ->**one** Salary

or with a ternary relation

set Job

set Person

set Company

set Salary

<person>.worksFor: Company ->**one** Salary

As another example, in the execution of a Java program, an implementation of java.util.Map (such as a hash table) can be represented with a ternary relation

map: Map -> Object -> Object

that associates maps, keys and values. The triple $m \rightarrow k \rightarrow v$ is in the relation when map m maps key k to value v . This relation can be shown as an arc labelled map? from Object to itself.

Another common case is when you have a relation called r say, of type $A \rightarrow B \rightarrow C$, and want to show it diagrammatically as an arc from A to C . In that case, it's convenient to use an informal shorthand, and label the arc $r[]$. For example, the map relation that associates an array of elements of type T with its elements has type

map: $T[] \rightarrow \mathbf{int} \rightarrow T$

but can be shown by an arc from $T[]$ to T labelled $\text{map}[]$.

© 2004, Daniel Jackson