Lecture 7 Summary

Code Object Models

Spring 2002: Devadas/Guttag

7.1. Introduction

An object model is a description of a collection of configurations. In today's lecture, we'll look at object models of *code*, in which the configurations are states of a program. But we'll see later in the course that the same notation can be used more generally to describe any kind of configuration -- such as the shape of a file system, a security hierarchy, a network topology, etc. These *problem object models* are very useful for conceptual modelling and will be the subject of a future lecture. Stay tuned!

The basic notions that underlie object models are incredibly simple: sets of objects and relations between them. What students find harder is learning how to construct a useful model: how to capture the interesting and tricky parts of a program, and not to get carried away modelling irrelevant parts, and end up with a huge and unwieldy model, or to say so little that you end up with an object model that is worthless.

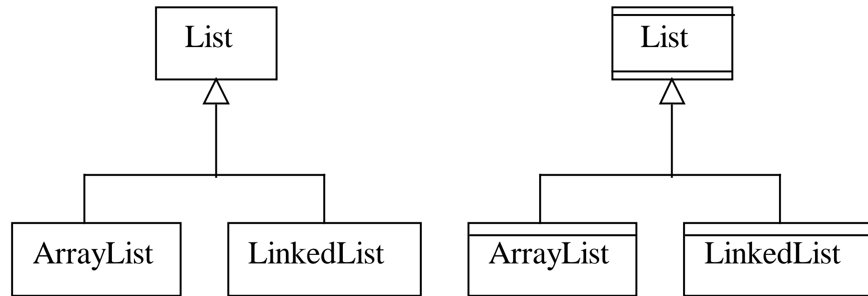
Object models and module dependency diagrams both contain boxes and arrows. The similarity ends there. The MDD is about syntactic structure -- what textual descriptions there are, and how they are related to one another. The OM is about semantic structure -- what configurations are created at runtime and what properties they have.

7.2. Object Models and MDDs

An object model expresses two kinds of properties: classification of objects, and relationships between objects. To express classification, we draw a box for each class of objects. In an object model of code, these boxes will correspond to Java classes and interfaces; in a more general setting, they just represent arbitrary classifications.
An arrow with a fat, closed head from class *A* to class *B* indicates that *A* denotes a subset of *B*: that is, every *A* is also a *B*. To show that two boxes represent disjoint subsets, we have them share the same arrowhead. In the diagram shown, *LinkedList* and *ArrayList* are disjoint subsets of *List*.

In Java, every *implements* and *extends* declaration results in a subset relationship in an object model. This is a property of the type system: if an object *o* is created

with a constructor from class *C*, and *C* extends *D* say, then *o* is regarded as also having type *D*.

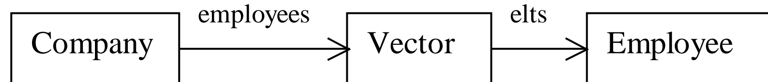


The diagram above shows the object model on the left. The diagram on the right is a module dependency diagram. Its boxes represent textual descriptions -- the code of the classes. Its arrows, you will recall, denote the 'meets' relation. So the arrow from *ArrayList* to *List* says that the *ArrayList* code meets the specification *List*. In other words, objects of the class *ArrayList* behave like abstract lists. This is a subtle property and is true only because of the details of the code. As we will see later in the lecture on subtyping, it's easy to get this wrong, and have a class extend or implement another without there being a 'meets' relation between them. (The sharing of the arrowhead has no significance in the MDD.)

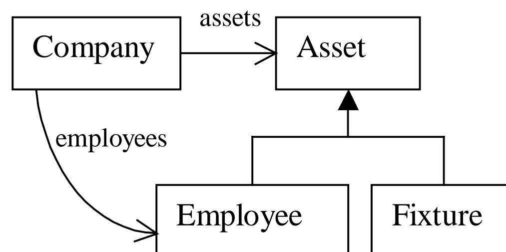
7.3. Object Model Basics

The elements of an object model are square boxes and arrows. The boxes denote sets of objects and correspond in the code to classes and interfaces. An arrow with an open head models a field; an arrow with a closed head says that one set of objects is a subset of another, and corresponds in the code to *extends* or *implements*.

This model for example, shows three classes, *Company*, *Vector*, and *Employee*, with the fields that relate them:

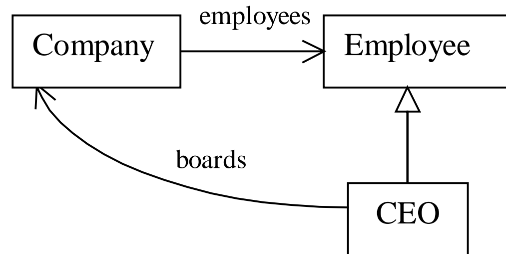


Suppose *Company* has an additional field, *assets*, that is a vector objects of interface *Asset*, which is implemented by *Employee* and some new class *Fixture*. This would shown like this:



Note that in the above object model, we have elided *Vector* hiding the representation of *Company*. A code object model does not have to expose the implementation, indeed, in many cases, it is better not to. Also, the closed arrow is filled, or *exhausive*. This says that the subsets *Employee* and *Fixture* "fill" the set *Asset*; there are no *Asset* objects that are not either *Employee* or *Fixture* objects. You may wonder whether every employee is regarded also as an asset (so that, in every object configuration, if an *Employee* object is in the *employees* vector it is also in the *assets* vector). An object model can record this kind of property too, as a textual constraint written adjacent to the model.

Subclassing is shown with the closed arrow. A subclass *CEO* of *Employee* might have, in addition to the fields of *Employee*, a field, *boards*, which associates a *CEO* object with a collection of Company objects (those companies on whose boards the CEO sits):

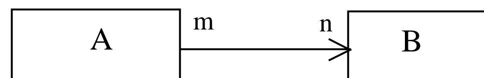


Note that in this case, the closed arrow is not filled, i.e., is not exhaustive. This says that there may be *Employee* objects that are not *CEO* objects.

7.4. Multiplicities

The object model can constrain how many objects there are in a set, and how many objects are related to a given object.

Suppose we have a field arrow from *A* to *B*, with multiplicities *m* and *n* on the source and target ends: Then there are *n B*'s associated with each *A*, and *m A*'s are mapped to each *B*.



There are 4 multiplicity symbols:

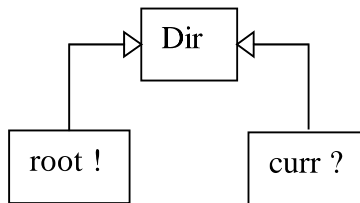
- `! means exactly one`
- `? means zero or one`
- `+ means one or more`
- `* means zero or more`

Because zero or more in not actually a constraint, omitting a symbol is the same as using a star. Multiplicity symbols can be used in three places: after the label in

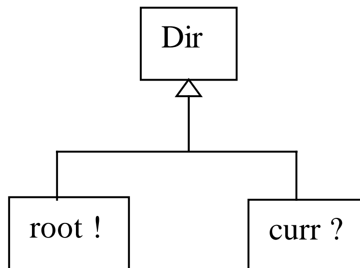
a box, in which case the symbol indicates the number of objects in the set, or on the source or target end of a field arrow.

For example, a *Company* has several employees, but each *Employee* works for exactly one *Company*:

A set of limited cardinality usually models a static field of a class. In a file system, we might have a root directory and a current directory, subclasses *root* and *curr* of the class *Dir*. This model shows that the *curr* field but not the *root* field may be null (because there need not be a current directory):

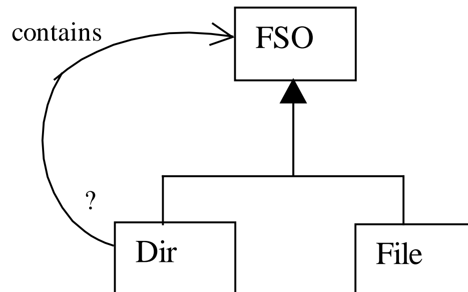


Note that in this model, the two subsets do not share an arrow. This says that the subsets are not necessarily disjoint. If we had drawn instead:



we would be asserting that *root* and curr cannot be *equal*; that is, the root directory cannot be the current directory also.

A *Dir* contains zero or more *FSO*s (file system objects), while an *FSO* is contained by zero or more *Dir*s:

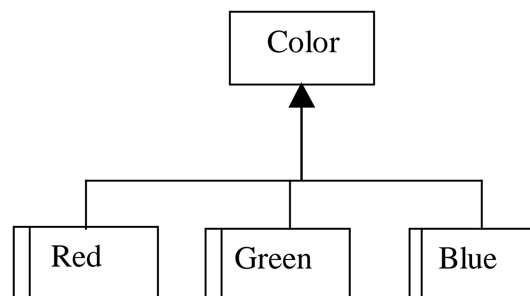


Conceptually, the two ends of the arrow are equally easy to understand. But in the code, the source end is more subtle. Putting a multiplicity of * or + on the target end just implies that there must be some sort of collection class that hasn't been shown. The exactly one symbol says that the field cannot be null. But multiplicity constraints on the source end say what sharing is possible – that no *Employee* can be shared by two *Company* objects, and that no *FSO* can be shared by two *Dir* objects. A source end multiplicity of ! or + says something about reachability too: the ! in the employee example says that each *Employee* object can be reached from a *Company* object.

The typical cases are * on the source end (showing that sharing may be possible), and ? on the target end, showing that the field is a possibly null reference to an object.

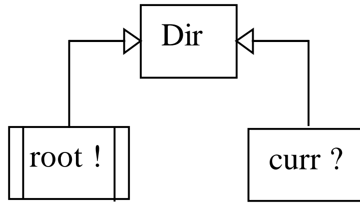
7.5. Mutabilities

Object models can also describe what changes are possible. A single vertical stripe on the left indicates that the set is *static*; that is, the objects in the set remain in that set for extent of their existence. The contents of the set can change; over time, new objects may come into existence, and old ones can die. For example, we might implement an abstract *Color* class, with three subclasses *Red*, *Blue* and *Green*. If the color of an object remains fixed for all time and cannot change, then we would draw:

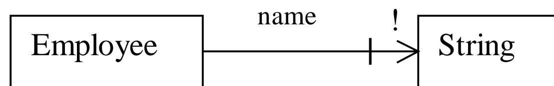


where the set *Red* can change, i.e., it might contain different objects, but an object cannot change color from red to green, or vice versa.

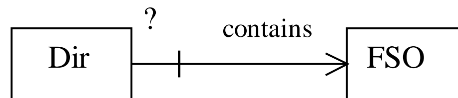
Putting stripes on either side of a box says that the set of objects denoted by that box is *fixed*. That is, the contents of the set do not change throughout the lifetime of the system. To show that the *curr* set can be changed, but not the *root* set would draw:



To show mutability of objects, we put mutability markings on *fields*. Putting a hatch mark on the target end of a field arrow says that a source object, once created, will always have the same target or targets. For example, this says that an *Employee* is created with a *name* that cannot be changed:

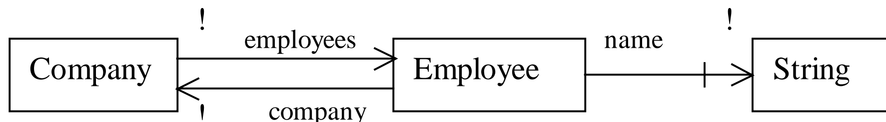


Putting a hatch mark on the source end says that which objects point to an object cannot change. This is a more subtle notion. If we put a hatch mark on the source end of the *contains* field, for example,



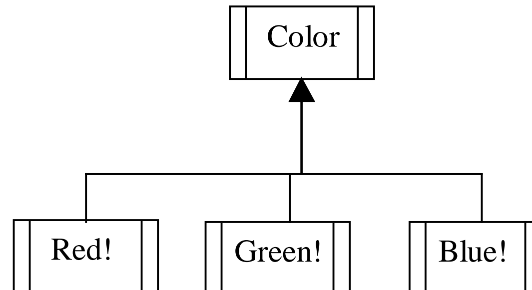
we would be asserting that an *FSO* object cannot be moved between *Dir* objects. We can associate a new *FSO* object with a *Dir* (by adding to a vector, e.g., if contains is implemented as such), and we can disconnect *FSO* objects when no longer needed, but we cannot transfer an *FSO* from one *Dir* to another. This model implies that to support the moving of file system objects between directories, fresh objects must be created.

If all the arrows coming out of a box are marked with hatches at the target end, this means that none of the fields can be changed. So the box represents a set of *immutable* objects. The value of this notation, using hatches on arrows, rather than marking sets as mutable or immutable, is that we can describe partial immutability (by hatching some targets and not others), and talk about whether an object may be passed from one object's field to another (with a source hatch). In this model, the *Employee* object's name field cannot change, but an additional field that maps an *Employee* to a *Company* can change:



Don't confuse a fixed set with a set of immutable objects. If we striped the *Employee* sets we would be claiming (rather oddly) that over the execution of the program, the set of *Employee* objects is fixed: none are created or garbage

collected. Fixed sets are most often used to describe singletons (implemented as static fields), but they are also useful for describing classes that would have been implemented using an enumeration type (which Java does not provide). For example, in our abstract *Color* class, we might create singletons *Color.Red*, *Color.Blue* and *Color.Green*:



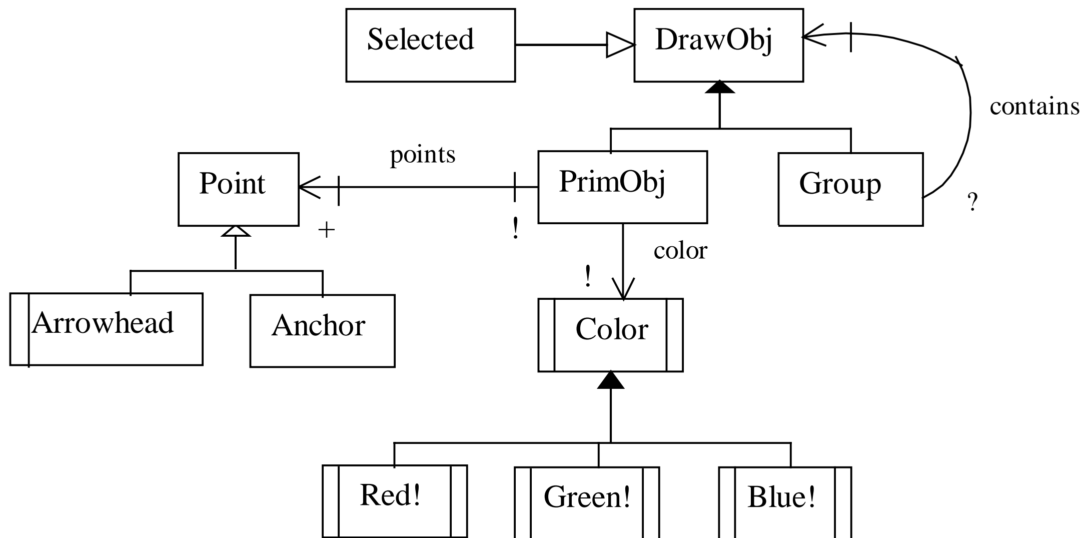
The class *Color* is striped because after initialization, no *Color* objects are created: the program simply reuses the three immutable objects as if they were primitive values.

7.6. Program Model Example

Here is an object model for (part of) the state of a drawing program. It illustrates most features of the notation:

- **Subsets**. The objects that belong to the classes *PrimObj* (for primitive drawn objects) and *Group* (for groupings of objects) also belong to the set *DrawObj*, which may be implemented as an interface or class. *Anchor* is a subclass of *Point*, and *Red*, *Green*, *Blue* are subclasses of *Color*. The selected set is a static field of *DrawObj*.
- **Disjoint subsets**. The sharings of open arrowheads say that no object is both a *PrimObj* and a *Group*, or both *Red* and *Green*, etc.
- **Exhaustiveness**. The filled arrowheads show exhaustiveness: that every *DrawObj* is either a *PrimObj* or a *Group*, and that every *Color* is *Red*, *Blue* or *Green*. The unfilled arrowheads say that there are *Point* objects that are not *Anchor* or *Arrowhead* objects, and that selected objects may belong to the classes *PrimObj* or *Group*.
- **Multiplicities**. Each *DrawObj* belongs to at most one *Group*. The subclasses of *Color* are singletons. Each *PrimObj* has at least one *Point* object associated with it, and a *Point* belongs to one *PrimObj*. Every *PrimObj* has a color.
- **Static sets**. The *Arrowhead* subclass and the *Anchor* subclass form disjoint subsets of *Point*. *Arrowhead* is a static set, i.e., an *Arrowhead* object cannot become an *Anchor* object.
- **Fixed sets**. The *Color* class and its subclasses are fixed; colors are not created on the fly.

- **Mutability of fields**. The color of a *PrimObj* may be changed. Which *Point* objects are associated with a given *PrimObj* may be changed, but a *Point* may not be moved from *PrimObj* to another. Which *DrawObj* objects belong to a *Group* cannot be changed, but a *DrawObj* object can belong to different *Group* objects at different times.

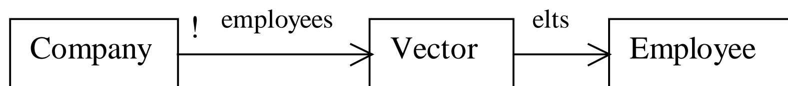


7.7. Instance Diagrams

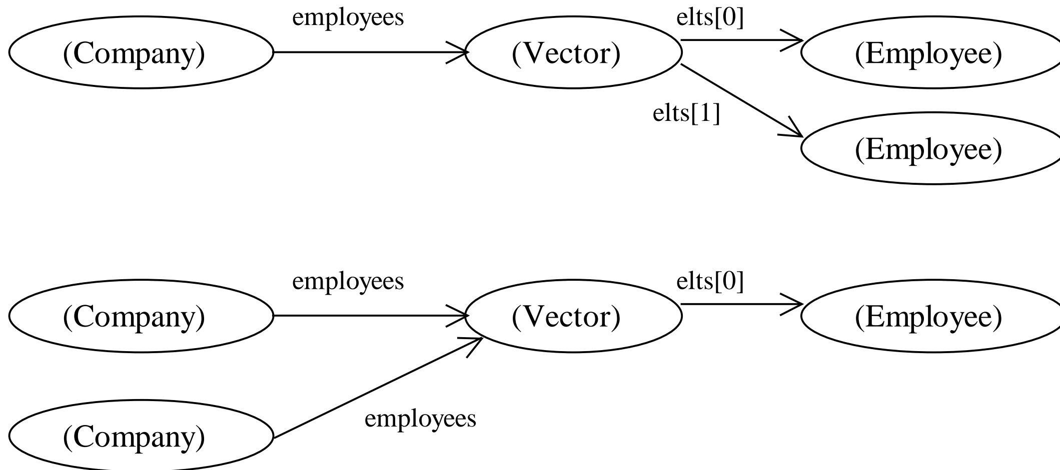
The meaning of an object model is a collection of configurations – all those that satisfy the constraints of the model. These configurations can be represented in instance diagrams or snapshots, which are simply graphs consisting of objects and references connecting them. Each object is labelled with the (most specific) class it belongs to. Each reference is labelled with the field its represents.

The relationship between a snapshot and an object model is just like the relationship between an object instance and a class, or the relationship between a sentence and a grammar.

Consider the object model below:



Some instance diagrams are shown below.



The first is a legal instance diagram, but not the second, because it violates the multiplicity constraint that each employee works for exactly one company. There are, of course, an infinite number of legal snapshots, since you can make an array of any length.

7.8. Object Model Glossary