

6170: Lab in Software Engineering

Note on Code Object Models

1 Introduction

Code object diagrams and models are graphical notations for describing the structure of the state in a program. A particular state is described with an *object diagram*; a collection of states is described with an *object model*. The meaning of an object model can thus be viewed as a set, usually infinite, of object diagrams, in the same way that the meaning of the Java grammar is the set of all well-formed Java programs.

When talking about an entire program, the object model describes the set of states that might arise during execution. When talking about a fragment of a program, such as a class or a package, the object model describes the set of states that are *intended* to arise when the fragment is embedded in a larger program. In this case, the model conveys more than just the semantics of the code: it also incorporates constraints, telling how the code is to be used. Object models can also be more abstract than the code, hiding details of representation that are uninteresting.

In the last few years, object models have become widely used. Each object oriented method has its own object modelling notation. The Unified Modelling Language, developed by a consortium of companies led by Rational, and now an Object Management Group standard, includes an elaborate object modelling notation. Our notation, called Alloy, is much simpler and has a precise semantics (which UML does not). It is almost identical to the notation used in Catalysis, a UML variant.

Fluency in object modelling is important for any software engineer. You'll need to be able to read object models to read the literature on design patterns, the most popular idea in software development at the moment. Increasingly, designers talk about the structure of code using object models because they are so much more convenient than code itself, more informative than class hierarchies, and more precise than informal sketches.

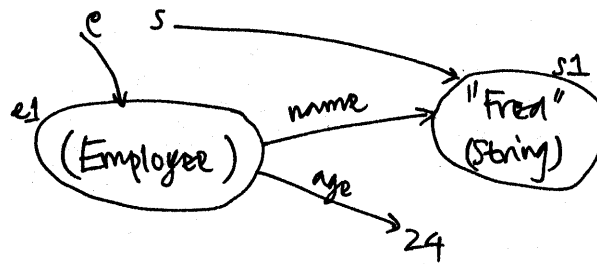
A basic result of theoretical computer science is that no non-trivial property of a program can, in general, be determined automatically. Object models are no exception (and are certainly non-trivial!). So it's impossible to write a tool that extracts a correct object model from a program. However, it's possible to obtain a pretty good approximation, and there are several tools that do this. The SuperWomble tool, developed as part of an MIT research project, is freely available for your use, and is more powerful than the commercial tools. It is very easy to use: you simply select a bunch of class files, choose the features that you would like to be shown, and it generates the object model.

Later in the course, we'll see how object models can be used to describe problems, rather than code. We'll be able to use exactly the same diagrams by giving them a more abstract interpretation. For now though, you should focus on understanding the meaning of an object model in concrete terms.

2 Object Diagrams

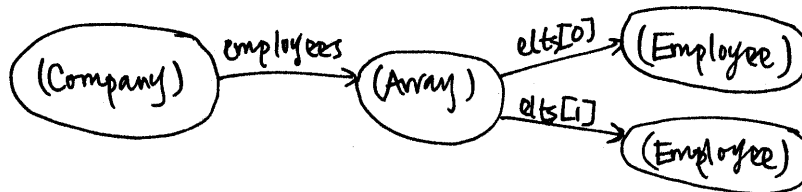
An object diagram shows a particular state or object configuration. Its elements are ovals, which represent objects, and arrows between ovals which represent fields of objects. Field arrows are labelled with the name of the field. The diagram may also include the values of variables, shown as arrows from a variable name to an object. One can optionally label the ovals with identifiers, for talking about particular objects, and show the type of an object in parentheses.

Here is a sample diagram that shows two objects, an *Employee* object *e1* and a *String* object *s1*, where *e1* has a field *name* holding a reference to *s1*, and a field *age* holding the primitive value 24, with two variables *e* and *s* holding references to the objects *e1* and *s1* respectively:

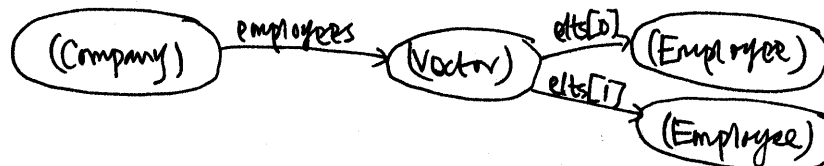


Note that primitive values are distinguished from objects by not being enclosed in ovals. The value of the string is shown informally as a label inside the string object.

An array is an object in its own right. Its elements are shown as a collection of field arrows, like this:



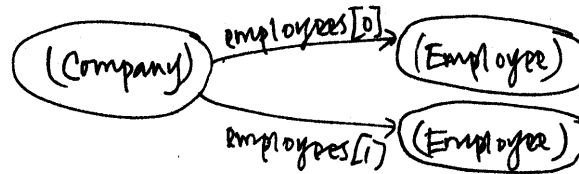
This shows a company object with a field *employees* that is an array of *Employee* objects. The same indexed notation can be used for the elements of a vector, or any other object that is conceptually a sequence; here is a different representation using a vector rather than an array:



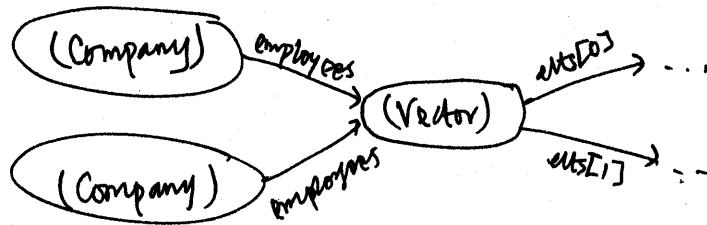
Note that these field arrows are *abstract*. Vectors are not built-in to the Java language, but are provided in the standard library. They therefore have a representation in terms of more primitive objects which we have chosen not to show. (Challenge: find out what that representation is, and show an

example as an object diagram.)

We can be even more abstract than this, and hide the fact that the *Company* object holds its employees using a vector:



Knowing when to use abstract field arrows requires some judgment. It depends on what you're using the object diagram for. If you want to show, for example, that the same vector of employees may be shared by two company objects -- probably a design error -- you'll need to be able to draw diagrams like this:

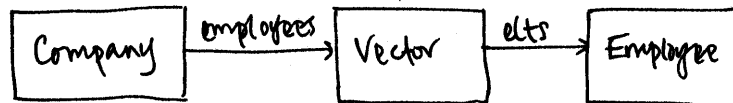


But if you only want to show that a *Company* is related to some *Employees*, you probably would do without the intervening vector object.

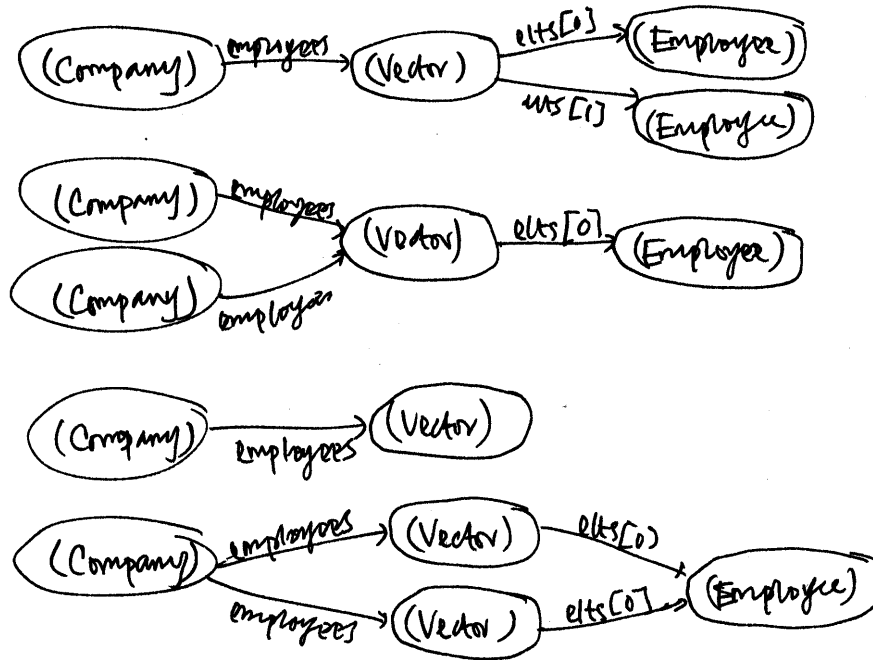
3 Object Model Basics

The elements of an object model are square boxes and two kinds of arrow. 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*. (We shall see that static fields can also be represented with sets and subset arrows, but don't worry about that for now.)

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



Note that the *age* field of *Employee* has been omitted. Primitives are usually less important in the design of a program, and so it often makes sense to omit them. This object model describes an infinite set of object configurations, such as:

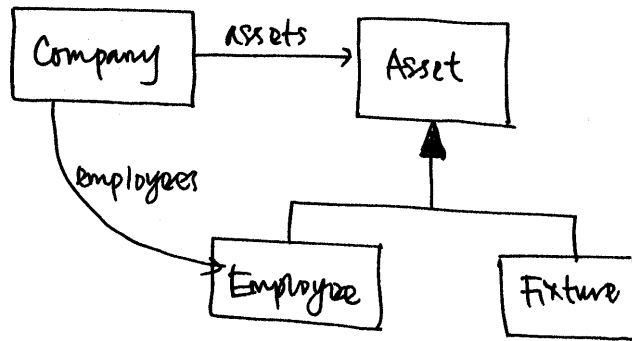


The second and fourth are not what we intend; we will rule these out with multiplicity constraints, described below. As in object diagrams, we can use abstract fields. We might elide the vector, for example, hiding the representation of *Company*:



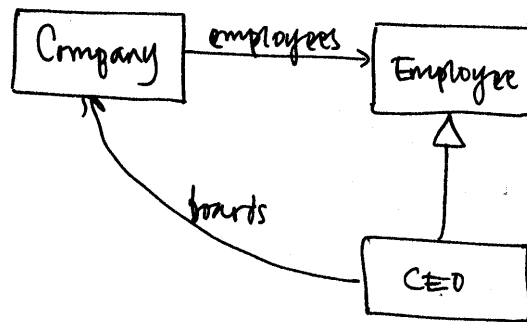
Suppose *Company* has an additional field, *assets*, that is a vector objects of interface *Asset*, which is

implemented by *Employee* and some new class *Fixture*. This would be shown like this:



Note that the closed arrow is filled. This says that the subsets *Employee* and *Fixture* 'fill' the set *Asset*; there are no *Asset* objects that are not either *Employee* or *Widget* 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. When we study representation invariants we'll see how important these kinds of constraints are.

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. This says that there may be *Employee* objects that are not *CEO* objects.

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.

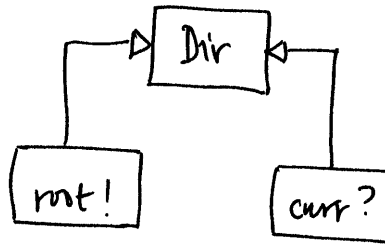
There are 4 multiplicity symbols:

- ! means exactly one

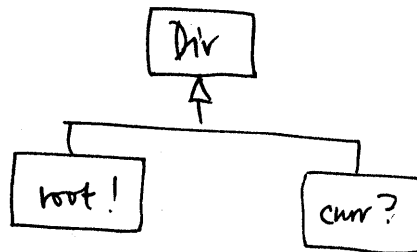
- ? means zero or one
- + means one or more
- * means zero or more

Because 'zero or more' is 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.

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, held as static fields *root* and *curr* of the class *Dir*. This model shows that the *root* field but not the *curr* field may be null (because there need not be a current directory):

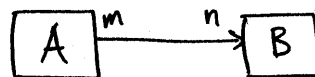


In other words, the field *Dir.curr* can be null, but *Dir.root* cannot. Note that in this model, the two subsets do not share an arrow. This says that the subsets are not necessarily disjoint. If we'd drawn instead



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

Suppose we have a field arrow from *A* to *B*, with multiplicities *m* and *n* on the source and target ends:

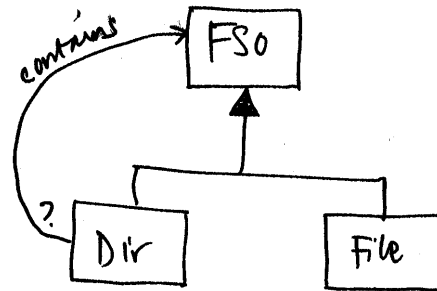


Then there are *n* *B*'s associated with each *A*, and *m* *A*'s are mapped to each *B*. For example, a *Company*

has several *Employees*, but each *Employee* works for exactly one *Company*:



and a *Dir* contains zero or more *FSo*s (file system objects), while an *FSo* is contained by zero or one *Dirs*:



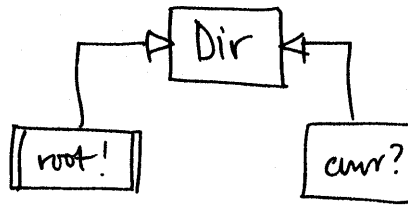
Conceptually, the two ends of the arrow are equally easy to understand. But in the code, the source end is more subtle. Putting a multiplicity of * or + on the target end just implies that there must be some sort of collection class that hasn't been shown. The exactly one symbol says that the field cannot 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 might be possible), and ? on the target end, showing that the field is a possibly null reference to an object.

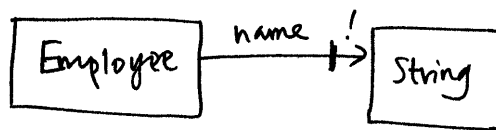
5 Mutability

Object models can also describe what kinds of changes are possible. Putting stripes on either side of a box says that the set of objects denoted by that box is *fixed*. This is useful for static fields. To show

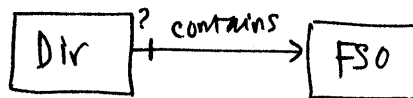
that the *curr* field of *Dir* can be changed, but not the *root* field, we'd draw:



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:



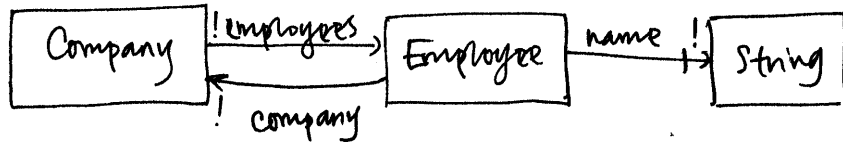
Putting a hatch mark on the source end says that which objects point to an object cannot change. This is a more subtle notion. If we put a hatch mark on the source end of the *contains* field, for example,



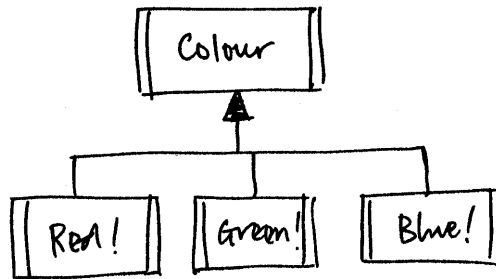
we would be asserting that an *FSO* object cannot be moved between *Dir* objects. We can associate a new *FSO* object with a *Dir* (by adding to a vector, eg, if *contains* is implemented as such), and we can disconnect *FSO* objects when no longer needed, but we cannot transfer an *FSO* from one *Dir* 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 *Com-*

pany can change:



Don't confuse a fixed set with a set of immutable objects. If we striped the *Employee* set 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, we might implement an abstract *Colour* class, with three subclasses *Red*, *Blue*, *Green*, and create singletons *Colour.Red*, *Colour.Blue* and *Colour.Green*:



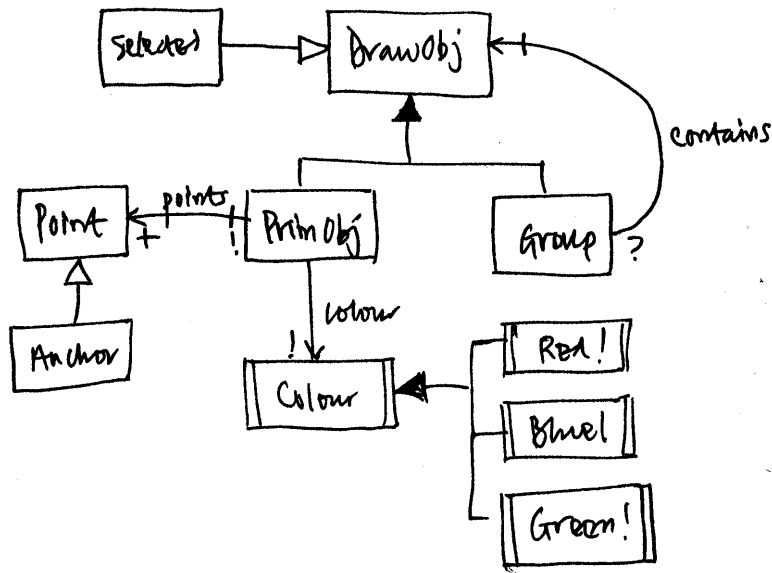
The class *Colour* is striped because after initialization, no *Colour* objects are created: the program simply reuses the three immutable objects as if they were primitive values.

6 Example

Here is an object model for (part of) the state of a drawing program. It illustrates every feature 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 *Colour*. 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 *Colour* is *Red*, *Blue* or *Green*. The unfilled arrowheads say that there are *Point* objects that are not *AnchorPoint* 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 *Colour* are singletons. Each *PrimObj* has at least one *Point* object associated with it, and a *Point* belongs to one *PrimObj*. Every *PrimObj* has a *Colour*.
- *Fixed sets.* The *Colour* class and its subclasses are fixed; colours are not created on the fly.
- *Mutability of fields.* The colour 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 one *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 References

7.1 Tools

<http://sdg.lcs.mit.edu/~womble>

Womble extracts object models automatically from bytecode.

<http://www.mit.edu/~6.170>

Microsoft has generously donated Visio 2000, a drawing tool, to 6170; you can download it from the class website. There is also a Visio template that we have developed for drawing object models and object diagrams.

7.2 Textbooks

Most object-oriented methods use some form of object model. If you take a job in industrial software development, you should be aware of these methods. In 6170, you learn essential notions of object modelling that appear in all of them; if you deepen your grasp of object modelling, you'll understand the strong and weak points of these methods and you'll know how to respond to a col-

league or manager who wants to use one.

There are many methods, and a huge number of books about them. We recommend:

- Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.
- Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.
- Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994.

The Unified Modeling Language has been adopted as a standard by the Object Management Group. It's a rather ungainly hotch-potch of notations from different methods, and most of the notations are not well-defined. It's taken quite seriously though, since it tries to overcome the proliferation of incompatible notations, and because it will be adopted in many companies despite its deficiencies.

- Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Second edition. Addison Wesley, 2000.
- James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
- <http://www.rational.com/uml>

Object models are used to show the structure of design patterns. It's well worth buying the 'Gang of Four' book:

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, October 1994.

7.3 Research Papers

<http://sdg.lcs.mit.edu/~dnj/publications.html#womble>

Describes Womble and how it works. Most of the paper should be accessible to a 6170 student.

<http://sdg.lcs.mit.edu/~dnj/publications.html#alloy>

Describes the full Alloy modelling notation, which includes a textual constraint notation. Rather technical and probably only of interest to students considering research in this area. Contains lots of references to related notations.