Object Models

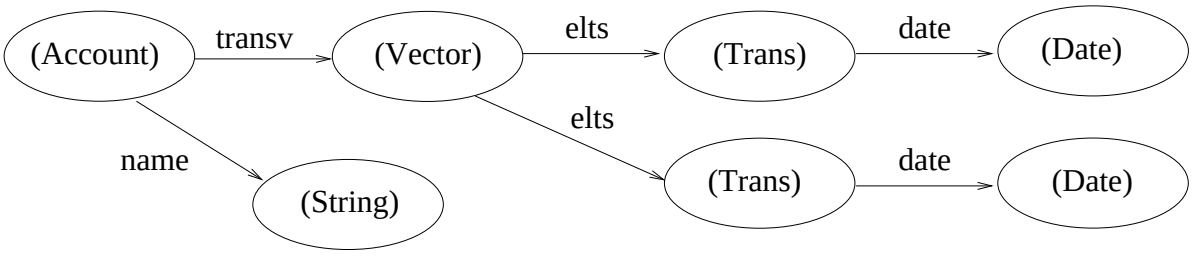
6.170 Lecture 6

Fall 2005

Today we turn to the subject of *object models*: a diagrammatic notation that allows us to capture sets of snapshots. Until now, we've had to illustrate the heap configurations of a program by showing individual snapshots using object diagrams. Object models are far more powerful, because they allow us to describe an infinite set of snapshots all at once. As the term progresses, we'll make more and more use of object models. not only for describing snapshots of programs, but also for describing more abstract properties of a problem domain.

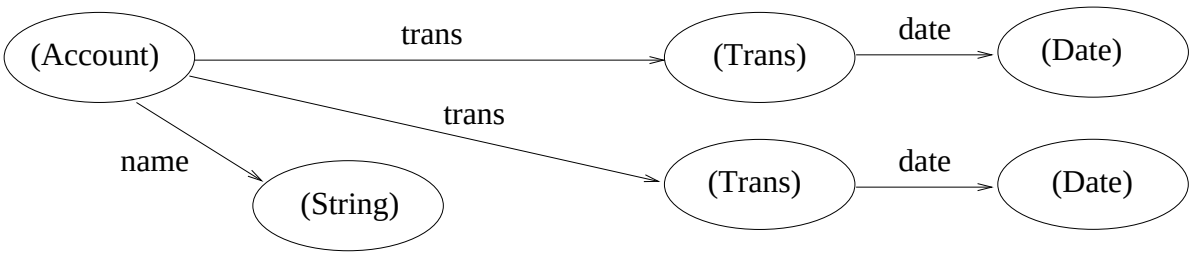
1 Object Diagrams

In our last lecture, we used *object diagrams* to show particular configurations of objects, or 'snapshots'. Here is an object diagram showing a snapshot of an `Account` object and the objects reachable from its fields:



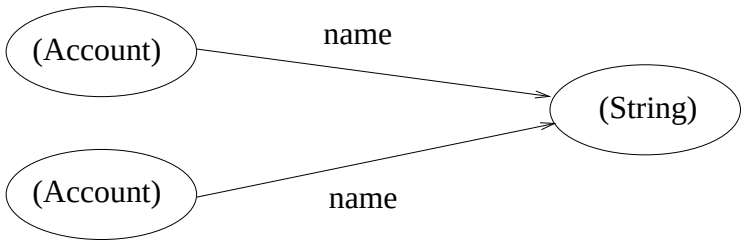
We're no longer concerned now to explain how assignment and so on works; instead, we're going to focus on structural constraints on object configurations. That's why the `Account` object isn't referenced by a variable in the diagram. It's not that the object is not reachable, but just that we're not concerned about how it's reached in the code. Note that we've also omitted the primitive values (the `balance` field of `Account` and the `amount` field of `Trans`), since we're primarily concerned about objects and their relationships.

Often, we'll want to elide collection objects such as the vector and only show the more interesting user-defined objects. The `Vector` is part of the *representation* of the `Account` object, and a client that calls the methods of `Account` sees only a `Trans` and not the `Vector` itself. So we might draw this diagram:

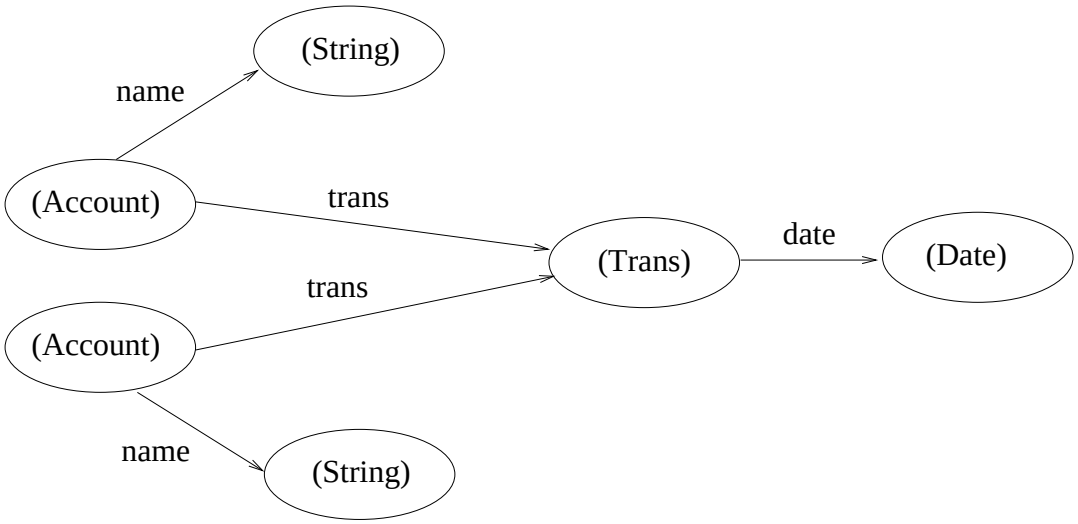


in which the abstract field arrow labelled trans from the `Account` object to the `Trans` object hides the `Vector`. This isn't actually new; we did the same thing with `Vector` itself, not showing that it was implemented with an array.

Many states can be created using these classes, but not all of them will be desirable. Here are two bad states. In the first, two `Account` objects share a name; this will mean that names cannot be used as unique identifiers:



In the second, a transaction belongs to two accounts:



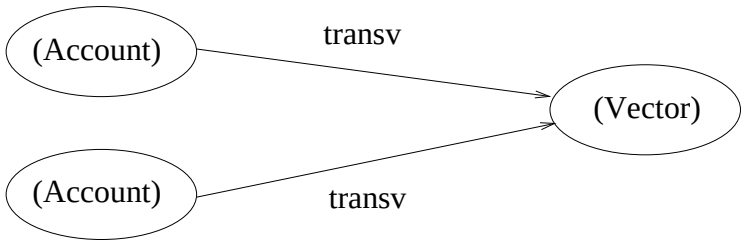
Whether these are in fact problematic depends on the intent of the designer of these classes, and the properties of the problem domain. For example, a transaction may be shared between two accounts if it represents a transfer; two accounts may share the same name if there is some other way to identify the accounts; it may cause no problems for two transactions to be recorded as occurring simultaneously.

So we cannot say for sure whether these configurations are right or wrong. What's important is to understand that such constraints do arise in practice, and you need a way to record them.

The crucial point is that the code itself does not indicate how it is to be used, so in addition to succinctly summarizing some code features (such as which classes and fields there are), the object model adds constraints about potential clients of the code.

In these diagrams, we've omitted the variable bindings and the primitive values, which are less relevant than the objects. It will often be convenient to draw object diagrams and object models that correspond to only part of the state, and sometimes we will even omit objects.

Sometimes we will want to talk about the representations of objects, and for these we will not want to elide intermediate objects, such as the vector. The diagram below shows a problematic state that might arise. This may be created by clients of the code we have seen, but should not be.

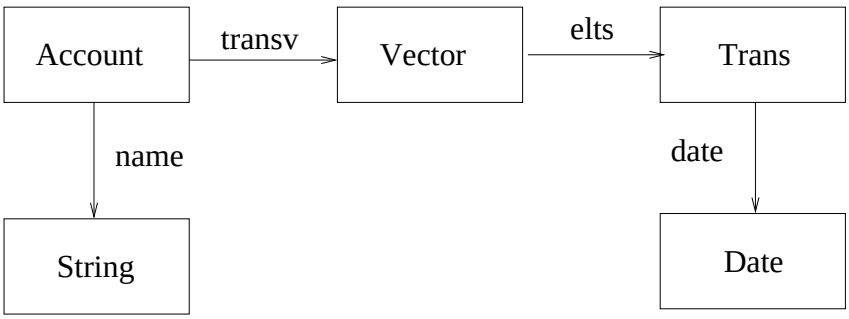


If two `Account` objects share a vector, an execution of the `post` method on one will cause a transaction to to be added to the vector (see Lecture 3), but the balance of only one of the accounts to be updated. This will violate a representation invariant that the balance should always be the sum of the amounts of the transactions in the vector. Later, we will describe this problem as a representation exposure, in which part of the representation of an `Account` object – its vector – has leaked out, and become accessible from the outside, thus compromising the invariant.

2 Object Model Basics

We've drawn lots of object diagrams showing various configurations. When we want to talk about a program and the configurations that can arise, it would be tedious if we had to draw object diagrams to illustrate different cases. So instead we'll draw *object models*. An object model is related to an object diagram the way a grammar is related to a sentence: while an object diagram denotes a single snapshot, an object model denotes a set of snapshots, usually infinite.

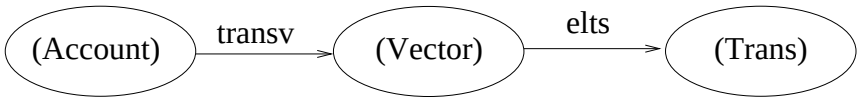
Here's an object model that corresponds to our class `Account`, whose code appears in Lecture 3:



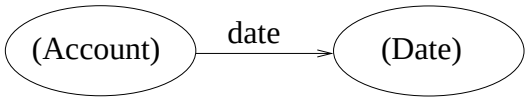
Each box corresponds to a class: it represents the set of all objects that belong to that class (in some given state). The arrows are relations, sometimes called associations, and they represent the

fields that connect classes. For example, the arrow labelled `transv` from `Account` to `Vector` shows that each object of the `Account` class has a field whose value is a `Vector` object.

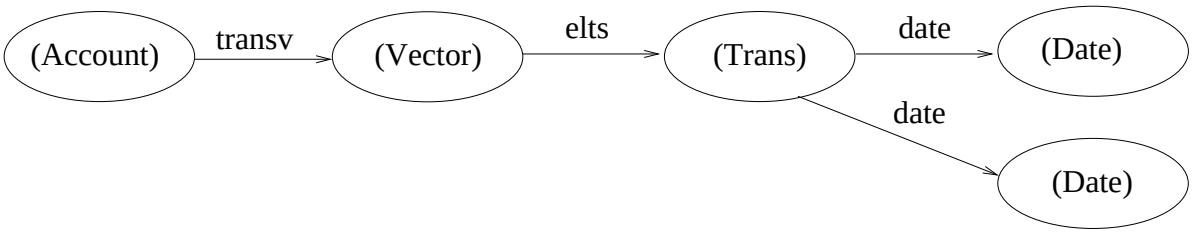
The object model specifies a set of object diagrams, by imposing the constraint that each object in the diagram belong to one of the object model boxes, and that any field arrow in the object diagram connect objects from the appropriate boxes. So our object model allows, for example, this state:



but not this state:

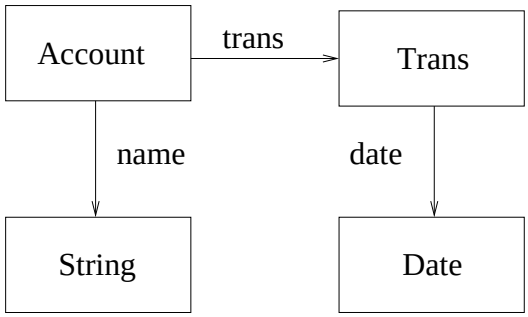


Note that the object model (at least without multiplicity constraints, which we'll talk about below) can't require that any object or field be present; it says what cannot be present. The model above places no constraints on the relative numbers of objects, so it admits crazy states such as:



which the code clearly rules out.

In the same way that we abstracted away Vectors in the object diagram, we can abstract away the `Vector` class in the object model:



3 Multiplicity

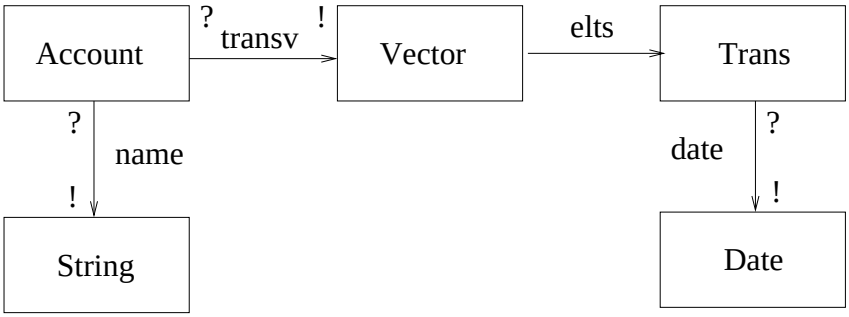
Our object model doesn't constrain how many objects there are of each class in relation to one another. For example, we might want to say that an `Account` can have several `Trans` objects, but a `Trans` can have only one `Date`. Multiplicity annotations let us do this.

The multiplicity symbols are:

- * (zero or more);
- + (one or more);
- ? (zero or one); and
- ! (exactly one).

When a symbol is omitted, * is the default (which says nothing). The interpretation of these markings is that when there is a marking `n` at the `B` end of a relation R from class `A` to class `B`, there are `n` objects of class `B` associated by R with each `A`. It works the other way round too; if there is a marking `m` at the `A` end of a relation R from `A` to `B`, each `B` is mapped to by `m` objects of class `A`.

Now we can add multiplicity constraints to our model:



Let's look at each of the multiplicity symbols and see what it tells us. We'll start with those on the target ends, because they're easier to understand:

- `Account` to `Vector`. The ! on the head of the arrow from `Account` to `Vector` tells us that in any legal state, there is exactly one `Vector` object associated with each `Account` object. In other words, the `transv` field is never null.
- `Vector` to `Trans`. The lack of a symbol, equivalent to *, tells us that there are zero or more `Trans` objects in the `Vector`.
- `Trans` to `Date`. The ! tells us that the date field of `Trans` is non-null.

Now let's consider the symbols on the sources of the arrows:

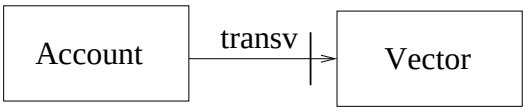
- `Account` to `Vector`. The ? on the tail of the arrow from `Account` to `Vector` tells us that each of these `Vector` objects is associated with at most one `Account` object. That is, `Account` objects don't share Vectors: two different `Account` objects must have different `transv` fields.
- `Vector` to `Trans`. The lack of a symbol says that each `Trans` may belongs to any number of Vectors, so even though each `Vector` belongs to at most one `Account`, a `Trans` may be shared between accounts.
- `Trans` to `Date`. The lack of a symbol would mean that two `Trans` objects may share the same `Date`. Since there is a ?, each `Date` object is associated with at most one `Trans` object.

Some multiplicity constraints can be enforced in a straightforward way in the code. A `Trans` object will certainly have at most one `date`, for example, because the field can hold at most one, and by making it private, constructing a `Trans` with a `date` that is checked to be non-null, and by allowing no subsequent mutations, we can ensure that the `date` field is non-null.

Others are much harder to enforce, and may require constraints on the clients of these classes. For example, we might decide that no two `Trans` objects can occur at exactly the same moment, so that we can always say of two `Trans` objects which is earlier. We express this by placing a ? on the source end of the `date` arrow from `Trans` to `Date`. Similarly, we might want `Account` objects to be uniquely identified by their `name` fields. In both cases, these constraints cannot be conveniently enforced within the classes themselves. Instead, we will ensure that they hold by careful design of collaborations amongst objects.

4 Mutability

An object model can also show mutability information: how the relationships between objects are allowed to change. We expect transactions to be added to an account, but we don't expect the vector holding the transactions to be replaced. This can be shown in the object model by marking the end of the arrow from `Account` to `Vector` with a small hatch:

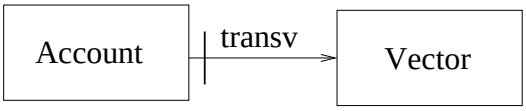


This means that the `Vector` associated with a given `Account` is fixed over the lifetime of that `Account`: it's set on creation (in the constructor) and not changed subsequently. Of course, the contents of the vector can change: those are determined by the arrow from `Vector` to `Trans` instead.

When the target end of a relation is hatched, the relation is said to be *target static*. The decision to make `transv` target static is a fundamental one about how the `Account` class is implemented. Since vectors are extensible, there's no need to replace the vector once allocated. But if we'd used an array instead, we would have had to replace it when it could no longer hold as many transactions as required. In that case, the relation from `Account` to `Array` would certainly not be target static:

Should the name relation from `Account` to `String` be target static too? There's nothing in the code of `Account` that prevents the name from being changed, but it seems like a reasonable constraint to impose.

Relations can be *source static* too. We can put a hatch on the left end of the arrow from `Account` to `Trans`, like this:

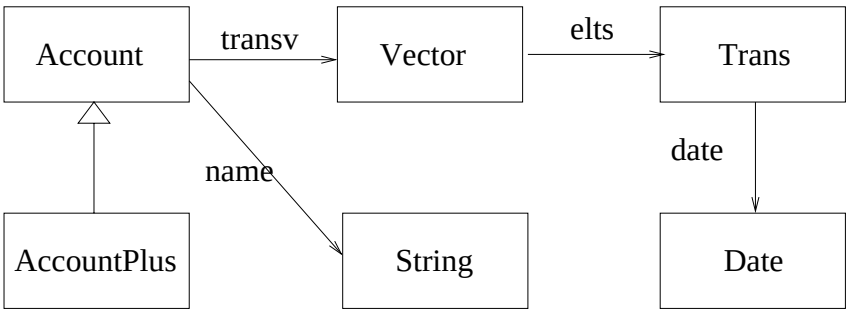


This would say that which `Account` is associated with a given `Vector` does not change over the lifetime of the `Vector`. This is subtly different from the constraint implied by the hatch on the other

end of the arrow: it means that if an `account` object is no longer used (and subsequently garbage collected by the Java runtime environment), the vector cannot be reused. Our implementation (see Lecture 3) satisifies this, since the constructor always creates a fresh vector rather than reusing an old one.

5 Subclassing in the Object Model

We can draw an object model that includes the relationship between `Account` and `AccountPlus`. A closed arrowhead from `A` to `B` says that every `A` is a `B`, or that the set of objects denoted by `A` is a subset of the set denoted by `B`. This can arise either because `A` is a subclass of `B`, or because `A` implements the interface `B` (more on that later).



Because every `AccountPlus` is an `Account`, the field from `Account` to `Trans` also implicitly associates `AccountPlus` objects with `Trans` objects. So the model allows states in which `AccountPlus` objects have trans fields. Likewise, the multiplicity constraints are 'inherited'. If we have a constraint that says that no two `Account` objects can share a `Trans`, then this will mean that no two `AccountPlus` objects can either, nor an `Account` object and an `AccountPlus` object.

In a more elaborate system, we might subclass `Trans` too: This object model shows that different subclasses may have different fields. Transfer transactions hold a reference to the other `Account`; Fee transactions may point to another transaction for which the fee was charged; Regular transactions have neither. The filled in arrowhead indicates that in this case the subsets exhaust the superset: namely that every transaction belongs to one of the subclasses. In implementation terms, this will imply that `Trans` is an interface or an abstract class.

6 Conclusion

The obejct model notation is a simple but surprisingly powerful one. We have seen how an object model can help bridge the gap between the problem domain and the program: we can articulate in the object model, for example, very directly and succinctly, that the name of an account should be a unique identifier, or that transactions should not be simultaneous. These properties can be quite subtle when we consider how they are enforced in the code. the object model describes *global* properties of the state; the tricky task that faces us is to establish these properties *locally*, within the appropriate class.