# Lecture 9: Dependences and Decoupling

A fundamental concern in designing in a program is how the modules depend on one another. Roughly speaking, a module A depends on a module B if A won't work properly without B working properly. Dependence is a liability, because it makes it hard to modify the program locally. If you change B, you may have to make a compensating change to A (and then to whatever depends on A, and so on). For this reason, dependences are often described as *coupling*, and the attempt to reduce the dependences between modules as *decoupling*.

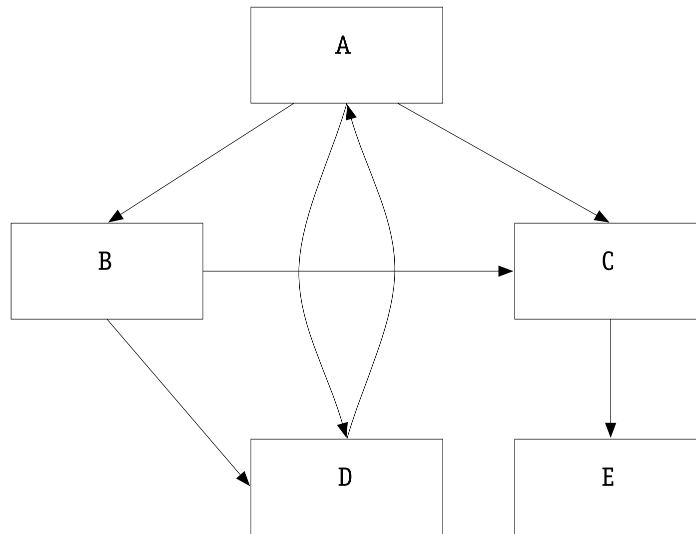
In this lecture, I'll explain the basic ideas of dependences, and present a notation for capturing the dependences of a program. This representation, called the *module dependency diagram* (MDD), is invaluable: it let's you see at a glance some crucial properties of the program, and is generally small and fairly easy to construct. You should draw MDD's as part of your design process before you start writing code, and you will often find it helpful to draw an MDD for a program that you're maintaining or using in some way if one doesn't already exist.

## 9.1 Parnas's Uses Relation

The idea of module dependences was first articulated by David Parnas in a seminal paper entitled *Designing Software for Ease of Extension and Contraction* (IEEE Transactions on Software Engineering, Vol. SE-5, No 2, 1979). Parnas defined the *uses* relation amongst modules as follows: a module A uses a module B if "correct execution of ''B may be necessary for A to complete the task described in its specification". He goes on: "That is, A uses B if the correct of functioning of A depends upon the availability of a correct implementation of B".

Note that he says *may* be necessary and not *is* necessary. Dependency relations, like modifies clauses (which we learnt about earlier in our study of procedure specifications) are actually more useful for what they omit than for what they state: if A uses B, we can't say very much, but if A does *not* use B, then we know that we can safely make changes to B (or remove it entirely) without affecting A.

Here's an abstract example of a uses diagram:




To illustrate its application, let's see how we can answer some questions about development tasks by looking at the uses diagram alone.

- *Reusable subsets*. Which groups of modules can we take and reuse in another program? Suppose we want the functionality that C provides. Then since C uses E we will need to take E also. So the set of modules {C,E} forms what Parnas calls a 'reusable subset'. If we want B, we need everything; that is, there is no reusable subset that contains B that isn't the whole program.
- *Testing and Order of Construction*. What order should we build the program in, and where will we need testing stubs? To test E, no other modules are needed. To test C we need E. This suggests an ordering: we construct the leaves first, and then the modules on which they depend, and so on. If the program is layered, this strategy will allow us to avoid writing stubs entirely. Or in the case of a program like ours, it will tell us which modules must be implemented together or which stubs we must write. For example, if after C we implement D, we will need a stub for A, since D uses it.
- *Division of Labour*. The uses diagram helps figure out how to allocate the task of implementing modules to different teams. Suppose we have two teams. It's clear that dividing into {A,B,D} and C,E is a better choice than dividing into {A,E} and {B,C,D}, for example, because it allows subprograms to be put together that can be tested independently with fewer stubs.

But the uses diagram is lacking in some key respects:

- *Transitivity*. By definition, it's transitive. In other words, if P uses Q and Q uses R, then P uses R too. But perhaps the whole reason for inserting Q between P and R was to decouple them from one another. The problem here is that the uses relation doesn't say *how* one module uses another, so in this scenario, it's not possible to distinguish

the modifications to R that will be propagated through to Q (and further through Q to P) from those that won't. This makes the uses diagram unsuited for reasoning about change propagation, and also for answering questions about which modules should be examined to trace a bug in the behaviour of one module (or to verify that the module operates correctly).

- *Modern programming features*. It's not obvious how to draw a uses diagram for a Java program, in which there are modules that are actually specifications and not executable code (ie, interfaces), and in which relationships between modules can arise on the fly at runtime (because passing an object essentially passes a handle on the code of its class).

## 9.2 Liskov's MDD

The MDD notation described in your course text addresses the transitivity problem. A module A is said to depend on a module B 'if a change to the *specification* of B might causes a change in A'. Every module is assumed to have a specification, and to be required to satisfy it.

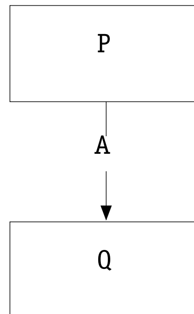
So now it's clear that some kinds of changes don't propagate. If we want to change B, but only the implementation and not the specification, then A need not change. If we want to change the specification of B, then A is likely to need changing. Oddly, this may not require a change to the implementation of B; we might be weakening the specification, for example, to admit a more efficient implementation at some later point. Questions about bug tracking are also easier to address now. A bug in B may indeed cause A to fail to meet its specification; whether it does will depend on whether B now fails to meet *its* specification.

Even this new notation, however, is not perfect. It doesn't allow us to say that A depends on B via a specification that is *not* the complete specification of B – that is, the one its implementor contracted to. This turns out to be a crucial distinction in many object oriented programs. When we study design patterns, we'll see that many of them rely on the introduction of a new specification between two modules that weakens the coupling between them.

This term, we'll therefore use a new dependency notation that I have developed. It's not perfect, but it does seem to give us the extra expressive power we need without becoming too complicated. You'll be using this notation in your own work, so you should make sure you understand it well!

## 9.3 The 3-Element Model

The MDD notation we'll use is based on a relation whose instances relate three things: two modules and an *assumption*. We'll draw an arrow from module P to module Q labelled A, and say 'P A-uses Q', when P was designed with an assumption A in mind, and as configured in the program, this assumption is discharged by Q.




Such an edge says that P depends on Q, but not necessarily through Q's standard specification. P may only depend on some small property that Q provides. There may in fact be several modules depending on Q, and they can depend on it through different specifications.

I use the term 'assumption' because it may not be a specification in the traditional sense that we use in 6170. Think, for example, about the relationship between a scheduler S and a process P that it schedules by calling a main procedure of P repeatedly. Which depends on which? In terms of procedure call, S uses P. But of course it's S that provides the service to P, and the designer of P that makes assumptions about how often it is scheduled for execution. So we might want to say that P uses S with an assumption about frequency of scheduling. Nevertheless in almost all cases we consider in 6170, the assumption on a dependency relationship will be a specification. Sometimes it will be part of a specification; an important precondition, for example, can be shown as an assumption from called to caller module.

What's a module? We will only treat executable code as modules. So for Java, the classes will be the modules. An interface is a specification, so it can appear as an assumption, but it cannot appear as a module. Not all assumptions, even those that are specifications, will correspond to Java interfaces. The *Object contract*, for example, which is an important specification that all classes in Java should obey, but which is not represented by any interface, will appear frequently as an assumption. Since specifications don't appear as modules, we won't have to worry about whether one specification can depend on another. (This usually isn't a major concern in software design).

Why this business about how the program is configured? The point is that a module may be designed with an assumption in mind, but may be written in such a way that which

module discharges that assumption is postponed until runtime. The Java HashMap class, for example, assumes that the implementation of its keys obeys the Object contract. So in fact we will show a dependence of HashMap on every class that might be a key for it in the particular program under consideration – and there may be several such classes.

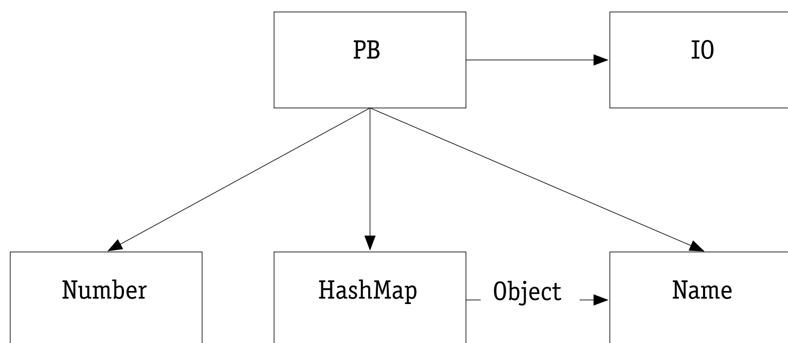
## 9.4 Example: Phone Book

To see how the MDD notation works, and the insight it provides into program design, let's consider a toy program that provides a phone book facility. The user enters names and phone numbers at the console; the program records their association, and responds to queries. There is no persistent storage of the phone book; when the program exits, all information is lost.

We might have the following modules in our program:

- PB, the main class.
- Name, a class implementing an abstract data type for names. It will provide methods to parse a name into a first name and surname, a toString method to format a name for output, an equals method to check whether two names are the same, and perhaps other observers that implement looser matches.
- Number, a class implementing an abstract data type for phone numbers. It will provide methods to parse a number into area code, number, extension, etc, and a toString method to format a number for output.
- IO, a class which encapsulates all the input-output facilities we'll need.
- HashMap, the standard Java hashtable-based implementation of maps that we'll use to store the name/number association.

Here is an MDD of a reasonable design:



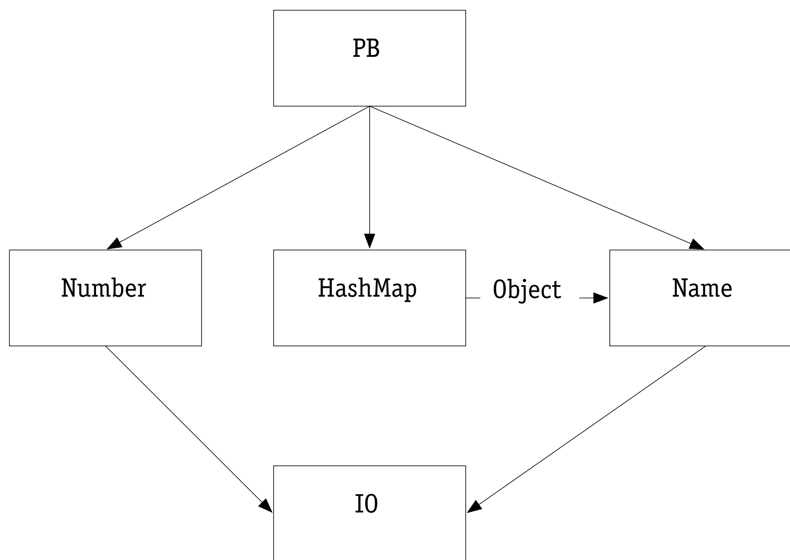

The main class PB uses all the other classes, not surprisingly. Name and Number are not dependent on anything; they're standalone data abstractions that can be reused.

HashMap has a dependence on Name through the Object contract, because if Name's equals and hashCode methods don't obey the specification given in Object, the hashing

mechanism may fail. Note that HashMap is in fact standalone, even though it depends on Name, and it certainly wasn't designed with Name in mind! The assumption labelled Object tells us it can be reused in any program that provides a class satisfying the Object contract – not a very arduous demand. There is no dependence of HashMap on Number, because the HashMap calls methods only on its keys and not its values.

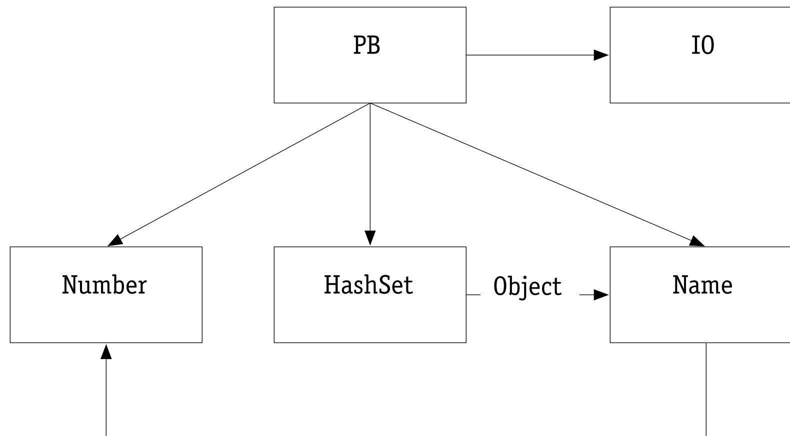
Note that only the dependence of HashMap on Name is labelled. All the other dependences have assumptions that are the specifications of their target classes. PB's dependence on Name, for example, can assume all the properties of the Name class.

Here are examples of variant designs that are less attractive. The first embeds IO functionality inside the Name and Number types. Name, for example, may have a method getNameFromConsole that reads characters at the console and returns an abstract Name object:




This is a bad design, because Name and Number can no longer be reused independently of IO. It is also likely to create a different kind of dependency which our MDD does not represent, called a *sharing constraint*. We are likely to want Name and Number to prompt for input and handle input errors in a consistent manner. If we change how it's done in one class, we'll need to change the other. This isn't a dependence of the sort we've been describing, since neither class uses the other, but it is a dependence nonetheless. In the original design, decisions about how to prompt for input and handle input errors are localized within a single module PB.

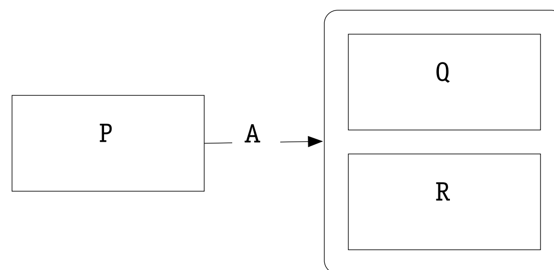
The second bad design uses a HashSet instead of a HashMap to allow Name objects to be looked up, and associates Names and Numbers directly by putting a number field inside the Name class:



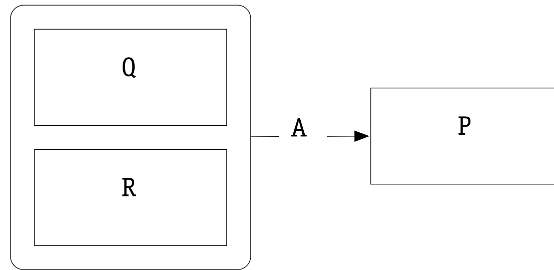

This is bad because it hardwires the relationship between the `Name` and `Number` types inside `Name`. We can't reuse one without the other. And suppose we want to implement different relationships: to allow several numbers to be associated with a single name, or to allow reverse lookup from numbers to names. This design will be much harder to adapt.

## 9.5 Grouping

Sometimes a module will use several other modules under the same assumption. A nice way to draw the diagram that avoids cluttering it is to make a contour around these modules, and draw one arrow to them. In this MDD, for example, `P` has an `A`-use of `Q` and `R`:




Similarly, if we want to show that several modules have the same dependence on some other module, we can show an arrow coming from a contour:




You can group modules with contours in any way you like, even partially overlapping them (although it can get incomprehensible if you have too many overlappings). And you can draw an arc from a contour to a contour. The meaning of the diagram is always clear: you just replace each arc by a set of arcs, so that `P` gets an `A-use` to `Q` if `P` is within a contour `c` and `Q` is within a contour `c'`, and there's an arc marked `A` from `c` to `c'`.

This kind of graph is called a *hypergraph*. The idea of using hypergraphs in models of software is due to David Harel. See: D. Harel, *On Visual Formalisms*, Communication of the ACM. Vol. 31, No. 5, 1988, pp. 514–530.

## 9.6 Assumption Annotations

What kinds of annotations can appear on dependency edges as assumptions? Here are some examples of the kinds of annotation that can be written:

- *Interfaces*. An assumption can be the name of a Java interface. As in Java itself, the name implies not only what can be checked by the compiler, but also whatever properties the designer chooses to associate with the interface. For example, the interface `Map` is accompanied in the Java library documentation by informal comments that describe how `put` and `get` and so on behave. Sometimes an interface constrains behaviour only very weakly, requiring the methods to have certain algebraic properties: see the interface `java.lang.Comparable`, for example, which describes informally some key properties that the `compareTo` method must have, but doesn't pin down its behaviour.
- *Class names*. An assumption that is the name of a class represents the full specification associated with that class. For example, a dependence on the class `HashMap` under the assumption `HashMap` allows the using class to assume that the class it uses is not just a `Map` but actually a `HashMap`. You can also use the name of a superclass to use its specification. The name `Object`, for example, implies an assumption of the specification of the `Object` class, from which all classes are descended. If you look in the Java documentation, you'll see that this specification gives a lot of detail about how methods that override the methods of `Object` must behave.
- *Method names*. An assumption may list some method names, qualifying the name of a class or interface, implying that the using class relies only on these methods. For

example, if we write `Map.put` on a dependency, it means that the using class only makes use of the `put` operation of the map it uses. We'll write `C.new` to denote the constructor for a class `C`.
- *Specification names*. To associate an arbitrary specification with a dependence that does not correspond to the specification of a class or an interface, we'll just invent a name and provide a separate note explaining the meaning of the name. For example, we might use the name `Subject` to refer to the specification of operations common to subject classes in the Observer pattern, even if we don't have an abstract class or interface corresponding to it.

An important property of the MDD, as we mentioned above, is that it is *conservative*: it says what dependences might be there, and may therefore overestimate the dependences. This is a vital freedom. It may turn out that a designer's assumptions about a module are too generous, and the implementor doesn't actually need all the assumed services. More commonly, we simply won't want to express the dependences in their most detailed form – it's too much work, and can result in a cluttered MDD.

This means that when you draw an MDD you have the freedom to omit certain bits of information, increasing the set of dependences described:
- You can omit the assumption label on a dependency edge. In this case, the dependence is assumed to be on the full specification of the used class.
- You can use a class or interface name without specifying which methods are relevant; this implies that potentially all methods may be used.
- You can draw a dependence arc to or from a contour when in fact it should go to or from a particular subset of the boxes within the contour, thus adding dependences between modules that may not actually have dependences.

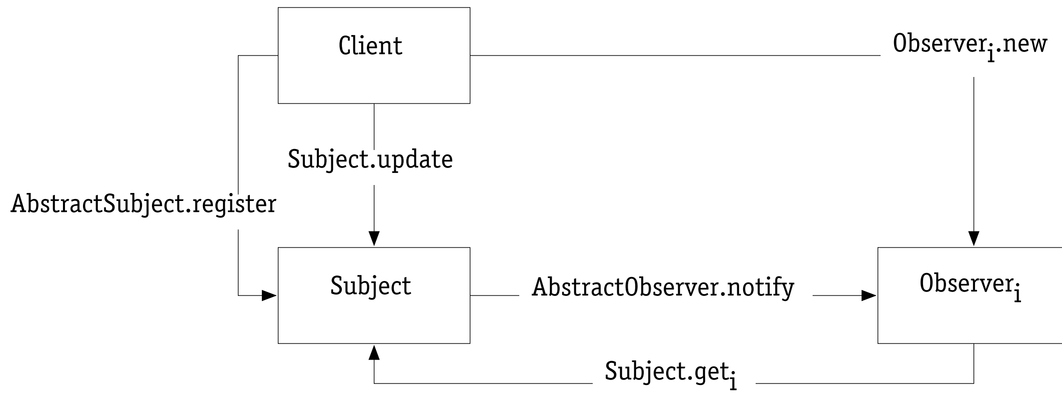
## 9.7 Name Dependences

Sometimes a module mentions another module by name, but doesn't actually make *any* assumptions about it. A class may have a method that takes an argument of another class, for example, and just passes the object of that class to a method of a third class. This is a *name dependence*, or, as Liskov calls it, a *weak dependence*, and it can be shown by a dotted line without a label.

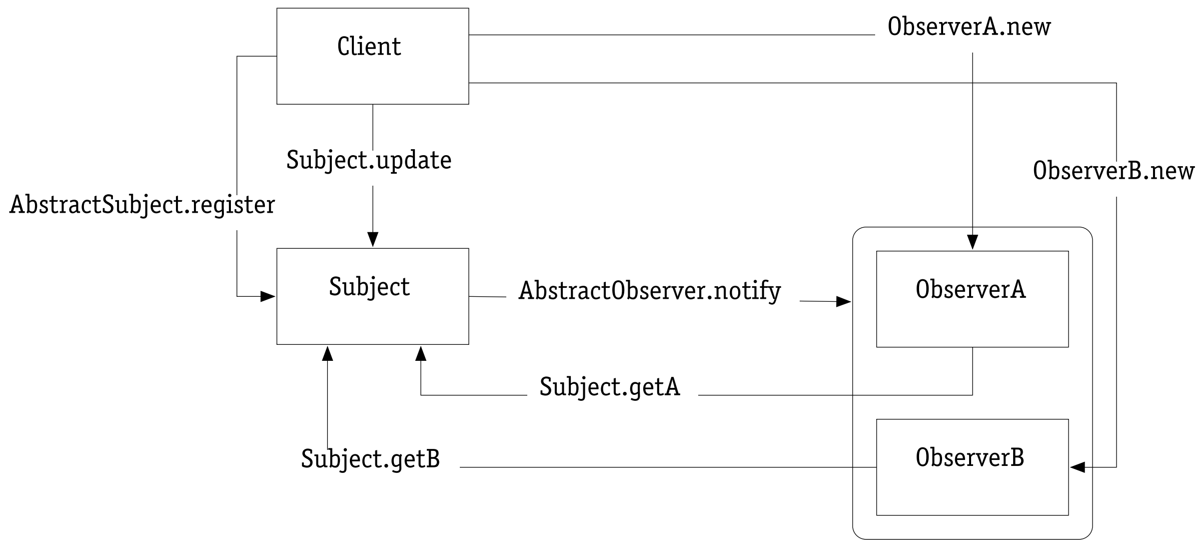
## 9.8 Families and Patterns

Sometimes we'll want to draw an MDD that describes not one particular program, but a family of programs. As a convenient shorthand, we'll use subscripts on module names and assumptions to indicate templates that can be replicated; each separate subscript can replicated independently.

For example, the Observer pattern, which we'll study later, can be described by an MDD that looks like this:




The subscript i indicates that there may be several distinct concrete Observer classes. The Client class has specific constructor dependences on these. And each of these Observer classes may call a different get method of Subject. An example of an instantiated MDD that this generic MDD represents is:

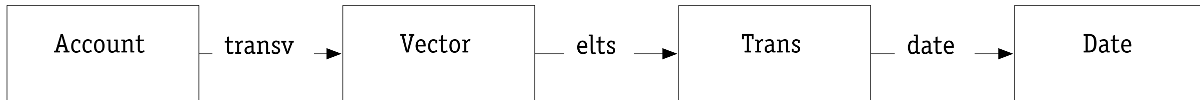



## 9.9 Comparison to Object Models

Don't confuse MDD's with object models (OM's). Superficially they look similar: both use boxes and arrows, and often the names in the boxes are names of classes. But remember that the boxes of an MDD are syntactic units, whereas the boxes of an object model represent sets of objects. And the arrows of an MDD represent dependences that

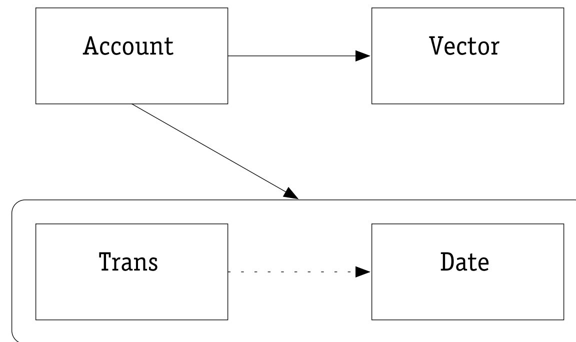
are usually associated with method calls, whereas the arrows of an object model represent relations, often implemented as fields.

For example, recall our object model describing configurations involving objects in a banking application:




The object model says that each Account is mapped by the transv relation to a Vector, which has elements that are Trans objects, each having a field date associating the transaction with a Date.

Here's a possible MDD for the same program:




Note that it has edges that are missing in the OM. There's a dependence of Account on Trans, and also on Date, for example, because there are methods in Account that manipulate transactions according to their dates.

And there are edges in the OM that don't correspond to edges in the MDD. The Vector class, for example, has no dependence on the Trans class: transactions are simply inserted into the vector and taken out again. (If we made use of the indexOf method of Vector, which uses the equals method of the element to find the index of a given element, we would have an Object-use of Trans by Vector). The dependence of Trans on Date is only a name dependence, since no methods of Trans actually use properties of Date: this represents a decision to treat Trans as a wrapper, rather than providing it, for example, with a method that would order transactions by date.