6170 Lecture Notes
Daniel Jackson, Fall 2k

Lecture 1: Introduction

1.1 Course overview

Course is actually three courses in one:
- crash course in object-oriented programming
- software design in the medium
- studio course on team construction of software

Emphasis is on design. Programming is included because it's a prerequisite; the project is included because at this stage, you don't really understand designs until you implement them.

You will learn:
- how to design software using some powerful abstraction mechanisms and a collection of patterns that have been found to work well in practice;
- how to get it right, by construction and by modular reasoning;
- how to articulate your design ideas and critique other people's designs;

and on the way:
- how to to think about a problem
- how to code in Java
- how to work in a team

Materials
- new textbook by Liskov
- lecture notes
- other recommended stuff on website (eg, Sun's Java documentation)

Prerequisites
- 6001, or some experience programming in a high-level language

1.1.1 Course organization and policy

Course in two halves. First half of term is lectures and weekly assignments done individually; second half is team project. Problem sets for first half build up to implementation of MapQuick, a local version of MapQuest using the US Census Bureau Database. Team project is Gizmoball, with some new and exciting features.

What we expect from you:
- attend lectures and recitations;
- complete readings in advance of lectures;
- do problem sets weekly for first half of term;
- do closed-book quiz at half term;

- attend project reviews;
- complete design and implementation of project.

What you can expect from us:
- lectures that explain new ideas and show how to apply them;
- recitations that offer practice and critique;
- problem sets that increase your skill with minimal grunt work;
- timely grading of your work;
- openness of lecturers to chat (drop by, and send email);
- availability of TA's during office hours;
  availability of lab assistants online;

Collaboration and IP policy:
- you may talk about course material with your colleagues;
- you may not share insights into weekly assignments;
- you may use any code we provide, and any code in the standard Java library;
- you may copy code and algorithms from textbooks or from general online sources;
- you may not copy each other's code, or use code written in 6170 by previous students;
- in the team project, you may share everything but should all contribute to all aspects.

Grading
- 75% on individual work: 45% problem sets, 25% quiz, 5% recitation participation
- 25% on team project
- we reserve the right to normalize across sections

Late policy
- no credit for late work
- but one slack weekend: can hand in Monday at noon instead of Friday at noon
- can't use slack on final project

Completion credit
- for problem sets 1 to 5, some part of the grade is for completing the implementation
- that means documenting, testing too
- can get this credit later if you weren't awarded it the first time
- you'll need work from earlier problem sets in later ones

Programming diagnostic
- to help us gauge your background
- due on Friday (September 8)
- not graded, but you cannot take the course unless you complete it

Signup sheet
- due at end of class today

1.1.2 Why does software engineering matter?

Software's contribution to US economy (1996 figures)
- greatest trade surplus of exports

- $24B software exported, $4B imported, $20B surplus
- compare: agriculture 26-14-12, aerospace 11-3-8, chemicals 26-19-7, vehicles 21-43-(22), manufactured goods 200-265-(64)
- from *Software Consipracy*

Role in infrastructure
- not just the Internet
- transportation, energy, medicine, finance

How good is our software?
- failed developments
- accidents
- poor quality software

1.2 Development failures

IBM survey, 1994
- 55% of systems cost more than expected
- 68% overran schedules
- 88% had to be substantially redesigned

Advanced Automation System (FAA, 1982-1994)
- industry average was $100/line, expected to pay $500/line
- ended up paying $700-900/line
- $6B worth of work discarded

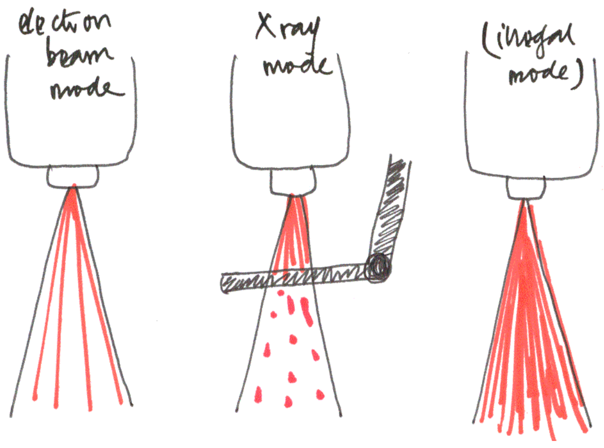
Bureau of Labor Statistics (1997)
- for every 6 new systems put into operation, 2 cancelled
- probability of cancellation is about 50% for biggest systems
- average project overshoots schedule by 50%
- 3/4 systems are regarded as 'operating failures'

1.3 Accidents

"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in. We're computer professionals. We cause accidents."
Nathaniel Borenstein, inventor of MIME
Programming as if People Mattered: Friendly Programs, Software Engineering and Other Noble Delusions (Princeton University Press, Princeton, NJ, 1991)

Therac-25 (1985-87)
- radiotherapy machine with software controller
- hardware interlock removed, but software had no interlock
- software failed to maintain essential invariants:
    - either electron beam mode
    - or stronger beam and plate intervening, to generate X-rays

- several deaths due to burning
- programmer had no experience with concurrent programming
- see: http://sunnyday.mit.edu/therac-25.html



Ariane-5 (June 1996)
- European Space Agency
- complete loss of unmanned rocket shortly after takeoff
- due to exception thrown in Ada code
- faulty code was not even needed after takeoff
- due to change in physical environment: undocumented assumptions violated
- see: http://www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html

London Ambulance Service (1992)
- loss of calls, double dispatches from duplicate calls
- poor choice of developer: inadequate experience
- see: http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las.html

In the short term, these problems will become worse because of the pervasive use of software in our civic infrastructure. PITAC report recognized this, and has successfully argued for increase in funding for software research:

“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/

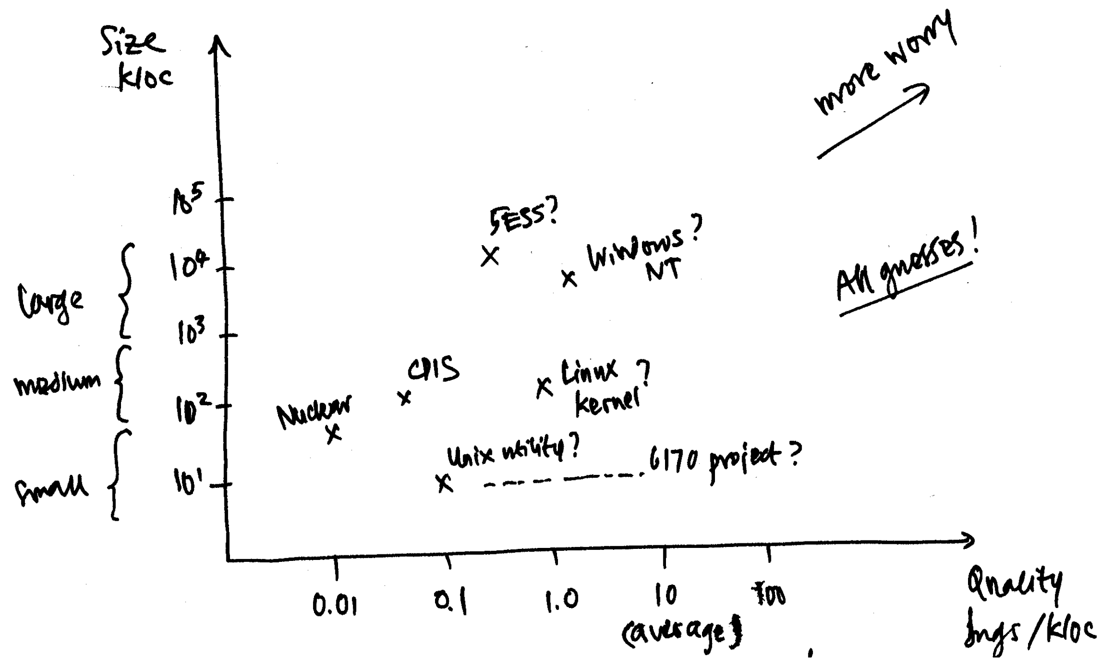
RISKS Forum
- collates reports from press of computer-related incidents

- http://catless.ncl.ac.uk

1.4 Software Quality

One measure: bugs/kloc
- measured after delivery
- industry average is about 10
- high quality: 0.1 or less

Some rough guesses:



Praxis CDIS system (1993)
- UK air-traffic control system for terminal area
- used formal methods: precise specification
- no increase in net cost
- much lower bug rate: about 0.75 defects/kloc
- even offered warranty to client!

Sample contracts:
- Cosmotronic Software Unlimited Inc. does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error-free. However, Cosmotronic Software Unlimited Inc. warrants the diskette(s) on which the program is furnished to be of black color and square shape under normal use for a period of ninety (90) days from the date of purchase.

- We don't claim Interactive EasyFlow is good for anything – if you think it is, great, but it's up to you to decide. If Interactive EasyFlow doesn't work: tough. If you lose a million because Interactive EasyFlow messes up, it's you that's out of the million, not us. If you don't like this disclaimer: tough. We reserve the right to do the absolute minimum provided by law, up to and including nothing. This is basically the same disclaimer that comes with all software packages, but ours is in plain English and theirs is in legalese.

ACM Software Engineering Notes, Vol. 12, No. 3, 1987.

1.5 Why Design Matters

"You know what's needed before we get good software? Cars in this country got better when Japan showed us that cars *could* be built better. Someone will have to show the industry that software can be built better."

John Murray, FDA's software quality guru
quoted in Software Conspiracy, Mark Minasi, McGraw Hill, 2000

That's you!

Our aim in 6170 is to show you that 'hacking code' isn't all there is to building software. In fact, it's only a small part of it. Don't think of code as part of the solution; often it's part of the problem. We need better ways to talk about software than code, that are less cumbersome, more direct, and less tied to technology that will rapidly become obsolete.

Role of design and designers
- thinking in advance always helps!
- contrast with reliance on testing: more effective, much cheaper
- makes delegation and teamwork possible
- design flaws affect user: incoherent, inflexible and hard to use software
- design flaws affect developer: poor interfaces, bugs multiply

It's a funny thing that computer science students areoften resistant to the idea of software development as an engineering enterprise. Perhaps they think that engineering techniques will take away the mystique, or not fit with their inherent hacker talents. On the contrary, the techniques you learn in 6170 will allow you to leverage the talent you have much more effectively.

Even professional programmers delude themselves. In an experiment, 32 NASA programmers applied 3 different testing techniques to a few small programs. They were asked to assess what proportion of bugs they thought were found by each method. Their intuitions turned out to be wrong. They thought black-box testing based on specs was the most effective, but in fact code reading was more effective (even though the code was uncommented). By reading code, they found errors 50% faster!

Victor R. Basili and Richard W. Selby. Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering. Vol. SE-13, No. 12, December 1987, pp. 1278–1296.

For infrastructural software (such as air-traffic control), design is very important. Even then, many industrial managers don't realize how big an impact the kinds of ideas we teach in 6170 can have. See the article that John Chapin (another 6170 lecturer) and I wrote that explains how we redesigned a component of CTAS, a new air-traffic control system, using ideas from 6170:

Daniel Jackson and John Chapin. Redesigning Air-Traffic Control: An Exercise in Software Design. IEEE Software, May/June 2000. Available at http://sdg.lcs.mit.edu/~dnj/publications.

1.6 The Netscape Story

For PC software, there's a myth that design is unimportant because time-to-market is all that matters. Netscape's demise is a story worth understanding in this respect.

The original NCSA Mosaic team at the University of Illinois built the first widely used browser, but they did a quick and dirty job. They founded Netscape, and between April and December 1994 built Navigator 1.0. It ran on 3 platforms, and quickly became the browser of choice on Windows, Unix and Mac. Microsoft began developing Internet Explorer 1.0 in October 1994, and shipped it with Windows 95 in August 1995.

In Netscape's rapid growth period, from 1995 to 1997, the developers worked hard to ship new products with new features, and gave little time to design. Most companies in the shrink-wrap software business (still) believe that design can be postponed: that once you have market share and a compelling feature set, you can 'refactor' the code and obtain the benefits of clean design. Netscape was no exception, and its engineers were probably more talented than many.

Meanwhile, Microsoft had realized the need to build on solid designs. It built NT from scratch, and restructured the Office suite to use shared components. It did hurry to market with IE to catch up with Netscape, but then it took time to restructure IE 3.0. This restructuring of IE is now seen within Microsoft as the key decision that helped them close the gap with Netscape.

Netscape's development just grew and grew. By Communicator 4.0, there were 120 developers (from 10 initially) and 3 million lines of code (up a factor of 30). Michael Toy, release manager, said:

"We're in a really bad situation ... We should have stopped shipping this code a year ago. It's dead... This is like the rude awakening... We're paying the price for going fast."

Interestingly, the argument for modular design within Netscape in 1997 was driven by the desire to go back to small team development. Without clean and simple interfaces, it becomes impossible to divide up the work into independent groups.

Netscape set aside 2 months to re-architect the browser, but it wasn't long enough. So they planned to start again from scratch, with Communicator 6.0. But 6.0 was never completed, and its developers were reassigned to 4.0. The 5.0 version, Mozilla, was made available as open source, but that didn't help: nobody wanted to work on spaghetti code. So Microsoft won the browser war, and AOL acquired Netscape.

This is not the entire story, by the way. Platform independence was a big issue right from the start. Navigator ran on Windows, Mac and Unix from version 1.0, and Netscape worked hard to maintain as much platform independence in their code as possible. They even planned to go to a pure Java version ("Javagator"), and built a lot of their own Java tools (because Sun's tools weren't ready). But in 1998 they gave up. Still, Communicator 4.0 contains about 1.2 million lines of Java.

You can read the whole story in:

Michael A. Cusumano and David B. Yoffie. Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft, Free Press, 1998.

See especially Chapter 4, *Design Strategy.*

Note, by the way, that it took Netscape more than 2 years to discover the importance of design. Don't be surprised if you're not entirely convinced after one term; some things come only with experience.

1.7 Advice

Course strategy

· don't get behind: first week especially is very fast!
· attend lectures: material is not all in textbook
· do the readings on time

Life strategy

· think in advance: don't rush to code
· design is more fun than debugging!
· focus on ideas, not embodiments
·   don't be blinded by technology
·   you should master Java, but realize that it will become obsolete

Be simple:

"I gave desperate warnings against the obscurity, the complexity, and over-ambition of the new design, but my warnings went unheeded. I conclude that there are two ways of constructing a software design: One way is to make it so simple there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."

Tony Hoare, Turing Award Lecture, 1980
talking about the design of Ada

How to 'Keep it simple, stupid' (KISS)

· avoid skating where ice is thin: avoid clever hacks, complex algorithms & data structures
· don't use most obscure programming language features
· don't optimize until proven necessary
· be skeptical of complexity
· don't be overambitious: spot 'creeping featurism' and the 'second system effect'.

Remember that it's easy to make something complicated, but hard to make something truly simple.

1.8 Closing Admin

What you must do:

- Hand in sign-up sheet before you leave.
- Complete and hand in PS0 (programming diagnostic) by Friday.

Recitation: tomorrow, we'll send section assignments to you by email.

Help getting going with Java

- Electronic classrooms open Sunday and Tuesday, with TA's on hand to help

Lecture 2: Object Semantics

In today's lecture, we'll focus on the heap semantics of Java: what the state looks like, in terms of variables, objects, and fields. We'll see the distinction between mutable and immutable objects; later on in the course, we'll return to these ideas in more detail. I'll illustrate particular states using object diagrams, which are our first step towards object models. Our aim is to be able to talk about properties of the state precisely, without having to show code. Diagrams are less cumbersome, easier to grasp (once you understand them), and can convey information that isn't in the code about the intended use of a class.

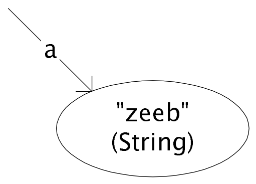
2.1 Variables, Declarations and Objects

Some types of objects can be created with literals. What happens when you run this?

String a = "zeeb";
System.out.println (a);

It prints *zeeb*. The first statement is a declaration (of the variable *a*) and an assignment (to *a*) in one. The expression "zeeb" is called a string literal. The second statement is a procedure call that prints a representation of *a* to standard out; don't worry for now about its details.

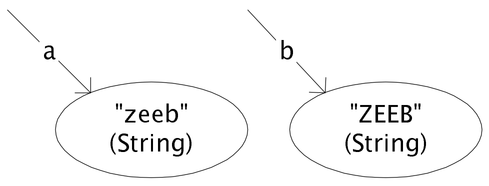
We can draw the result of the first statement as an object diagram, showing that *a* is a reference to an object of type String:



What happens when you run this?

String a = "zeeb";
String b = a.toUpperCase ();
System.out.println (b);

It prints *ZEEB*. The second statement is a call to the method *toUpperCase*. The method is applied to the object referenced by *a*, sometimes called the 'receiver', and results in the creation of a fresh string that is then bound to the variable *b*. Here's the object diagram:



What would this do?

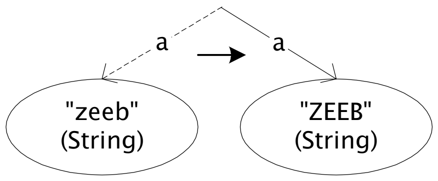
String a = "zeeb";
a.toUpperCase ();
System.out.println (a);

It prints *zeeb*. The second statement creates a fresh string that gets thrown away since it is bound to no variable. The string object referenced by *a* is not changed: strings are *immutable*.

What about this?

String a = "zeeb";
a = a.toUpperCase ();
System.out.println (a);

Again, no object changes. But *a* is made to refer to the new object by the second assignment, so the result is *ZEEB*.

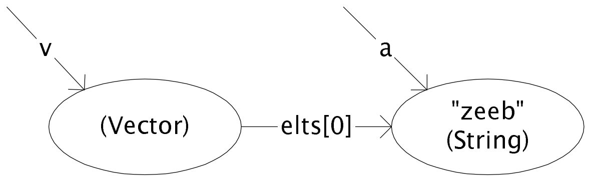


2.2 Aliasing, Mutability and Equality

Another example:

Vector v = new Vector ();
String a = "zeeb";
v.addElement (a);
System.out.println (v.lastElement ());

This prints *zeeb*, because the third statement inserts (a reference to) the string into the vector (referred to by) *v*:



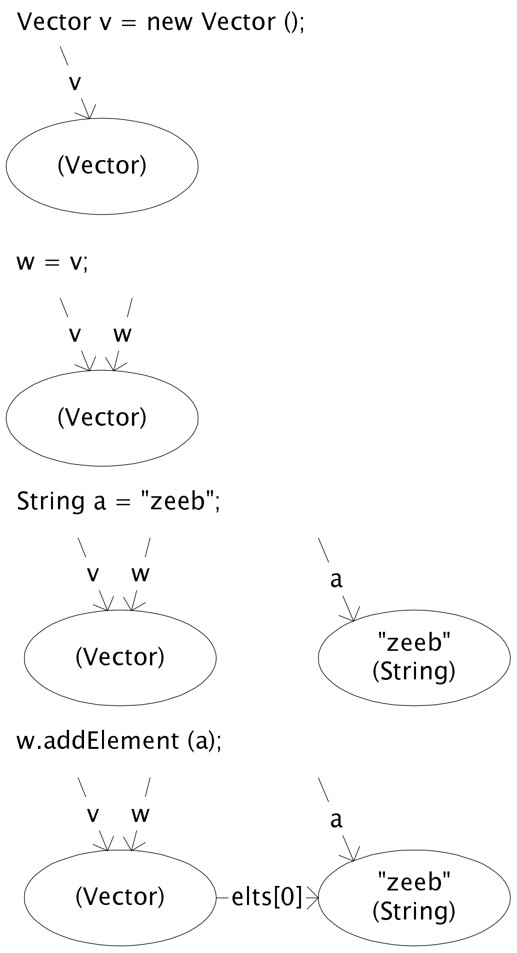
and the last statement extracts this element and prints it. Unlike *toUpperCase*, the method *addElement* does not return an object, but changes its argument object. Vectors are said to be *mutable*.

To see this, consider:

Vector v = new Vector ();
Vector w = v;
String a = "zeeb";

```
w.addElement (a);
System.out.println (v.lastElement ());
```

This prints *zeeb*, since the two variables *w* and *v* are names for the same vector object: they are said to be *aliases*. The call to *addElement* mutates the vector object, and the change is seen through both variables:



Aliasing is pervasive in languages like Java, and very useful. But it adds a lot of complexity. For one thing, it breaks the rule that a statement 'affects only the variables it mentions'. Just because *v* isn't mentioned in *w.addElement(a)* doesn't mean that the value test involving *v* alone won't change.

How can we observe the aliasing? By testing equality:

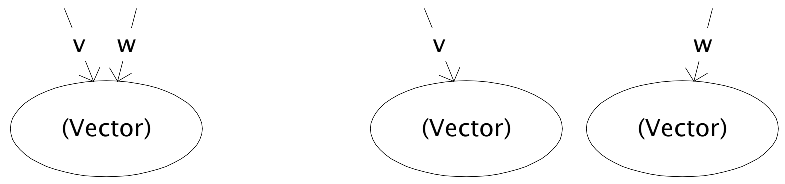
```
Vector v = new Vector ();
Vector w = v;
if (v == w)
    System.out.println ("same object");
```

which results in *same object* being printed. The built-in == test tells you whether two references are for the same object.

What does this do?

```
Vector v = new Vector ();
Vector w = new Vector ();
if (v == w)
    System.out.println ("same object");
```

It prints nothing, because *v* and *w* are distinct objects. Here are the object diagrams for the two cases:



A puzzle: what does this do?

```
String a = "zeeb";
String b = "zeeb";
if (a == b)
    System.out.println ("same object");
```

Strangely, this prints *same object*, because the Java virtual machine automatically 'interns' string literals: if it can tell that two string literals have the same sequence of characters, it only allocates one object. You'd be right to think this is a bit confusing; it's a performance optimization.

To determine whether two immutable objects have the same value, you use the equality *method*. The String class provides a method *equals* which tells you whether two strings contain the same sequence of characters or not. This code

```
String a = "zeeb";
String b = a.toUpperCase ();
if (b.equals ("ZEEB"))
    System.out.println ("same characters");
if (b == "ZEEB")
    System.out.println ("same object");
```

prints *same characters.*

Break for questions:

· Would you expect that generally *x == y* implies *x.equals (y)*? *Yes, it should. Because the equals method can be user-defined, just like any other method, you could make it behave in any way you wanted. On a mutable type, it might even mutate the object! But that would be bad form: there's a generic contract that clients expect equals to obey. More on this later.*

· Why would a language have immutable types? *Because aliasing is complicated, and when you use immutable types, the issue doesn't arise. Also, code built with immutable types can sometimes be more efficient.*

2.3 Null References

What does this do?

```
String a = null;
System.out.println (a);
```

It prints *null*. The keyword *null* denotes a value that can be taken on my an object reference. It means that the reference does not in fact refer to any object. There is no null object, by the way!

But note that

```
String a = null;
String b = a.toUpperCase ();
System.out.println (b);
```

throws a *NullPointerException*. What's the difference? Every method call must have a receiver, and because *a* is null, there is no receiver in the *toUpperCase* method call. But in the previous example, the call to *System.out.println* is OK, since the null reference is an *argument*, and you can write a method that tests whether an argument is null and does something appropriate.

Dereferencing null is a common programming mistake in Java. To avoid it, you can check whether a reference is null before you attempt to call a method:

```
if (arg == null)
    System.out.println ("error")
else {
    String x = arg.toUpperCase ();
    ...
```

A better solution is to avoid creating null references in the first place. You'll learn about that when we discuss representation invariants. Sometimes you can't avoid it, and then it's important to document where the null references may occur. That's one reason specifications are important: they can spare you runtime errors and unnecessary checks.

Question:

- In general, would you expect a.equals (b) and b.equals (a) to have the same effect? *No, because when a is null and b is not, the first will throw an exception, and the second will (usually) return false.*

2.4 Instance Variables or Fields

Let's make an object of our own.

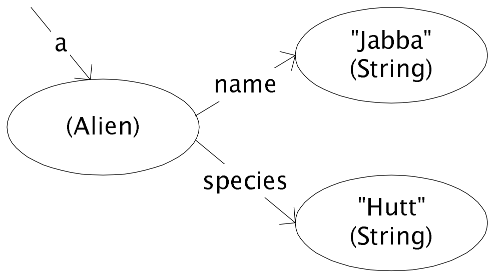
```
class Alien {
    String name;
    String species;
    }
```

The variables *name* and *species* are called 'fields', or 'instance variables'. The class describes a collection of of objects, each of which has a *name* and a *species*.

What does this do?

```
Alien a = new Alien ();
a.name = “Jabba”;
a.species = “Hutt”;
System.out.println (a.name);
```

It prints *Jabba*. How many objects are there here? Three: two string objects, and an alien object. The object diagram looks like this:



The ‘setter’ statement

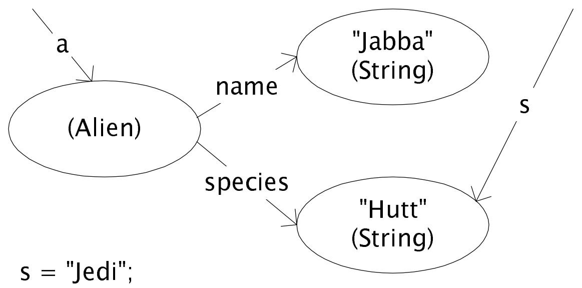
```
a.name = “Jabba”;
```

mutates the object referenced by *a*, by making its *name* field point to a different object. This is not an assignment. Assignments change the bindings of variables and can never mutate objects; mutations change the values of fields, and thus mutate objects. Suppose that after the four lines of code above, we execute

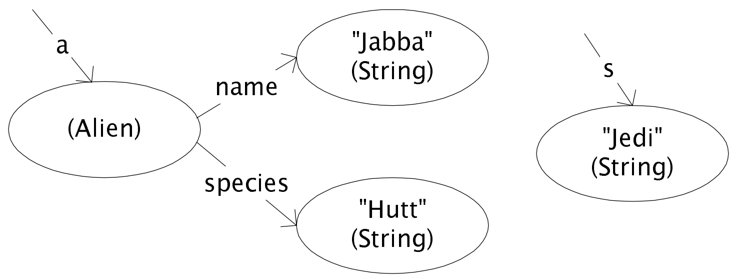
```
String s = a.species;
s = “Jedi”;
System.out.println (a.species);
```

If you think this makes Jabba a Jedi, you haven’t understand setter statements. It prints *Hutt*, because the effect of the first two statements is this:

String s = a.species;



s = "Jedi";



Note that the expression *a.name* means completely different things on the left and on the right. In fact, because of this, it's best not to call it an expression. On the right, it really is an expression; it denotes the value of the *name* field of the variable *a*. But on the left, it acts as a setter; the statement

e1.name = e2;

says 'make the name field of the object denoted by expression e1 point to the object denoted by the expression e2'. So in a statement of the form

e.f.g = h;

e.f is an expression, but *e.f.g* is not.

2.5 Semantics of Method Call

Here's our first method definition:

```
class Alien {
    String name;
    String species;
    void print () {
        System.out.println (name + " the " + species);
        }
    }
```

Now we can write:

```
Alien a = new Alien ();
a.name = “Jabba”;
a.species = “Hutt”;
a.print ();
```

and the result will be *Jabba the Hutt*.

Note that in the body of *print*, the expressions *name* and *species* are short for *this.name* and *this.species*, where *this* is a reference to the receiver object that is passed implicitly. So they’re actually getters. Here’s a method that sets a field:

```
void mutate (String s) {
    species = s;
    }
```

Don’t be confused: *species* here is short for *this.species*, so the statement is a setter equivalent to

```
this.species = s;
```

even if it looks like the assignment of a local variable.

Finally, here’s a constructor:

```
Alien (String n, String s) {
    name = n; species = s;
    }
```

and some code that uses it:

```
Alien a = new Alien (“Jabba”, “Hutt”);
a.mutate (“Ewok”);
a.print ();
```

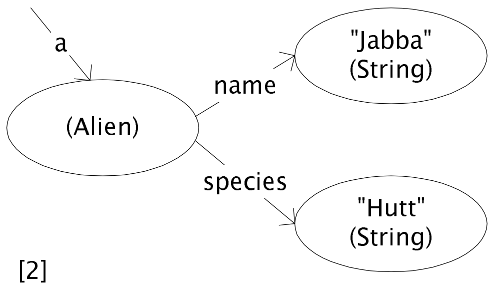
which yields *Jabba the Ewok*. The object diagram cartoon below shows how this happens. Diagram 1 shows the state resulting from the constructor call. Diagram 2 shows the state inside the *mutate* method, just after the call to *mutate*; the variable *this* has been assigned to the receiver, which can no longer be accessed by the variable *a* (since it’s out of scope), and the variable *s* has been bound to the string argument. Diagram 3 shows the result of the setter statement, which mutates the Alien object. Diagram 4 shows the state just after the call to *mutate* has returned, with the names *this* and *s* no longer in scope.

(A Java hint: if you try and now create an Alien object with *new Alien ()*, the compiler will reject it. The default constructor is only available when no explicit constructor has been declared. Now that we have declared a constructor, it’s the only one we can use.)

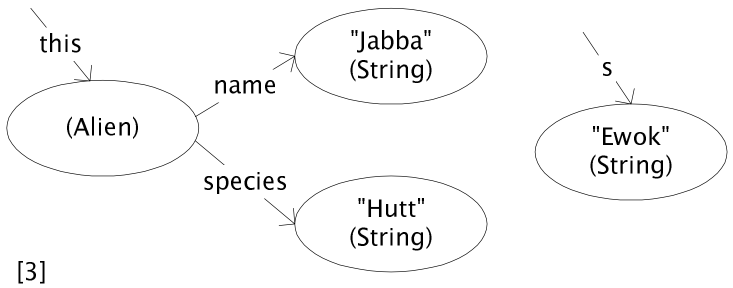
The parameter passing mechanism is sometimes referred to as ‘call by sharing’, since the formal arguments (such as *this* and *s*) share the objects with the variables of the calling context. But it can equally be viewed as call by value, in which the values that are passed are *references* to objects.

To check that you understand the mechanism, consider this method

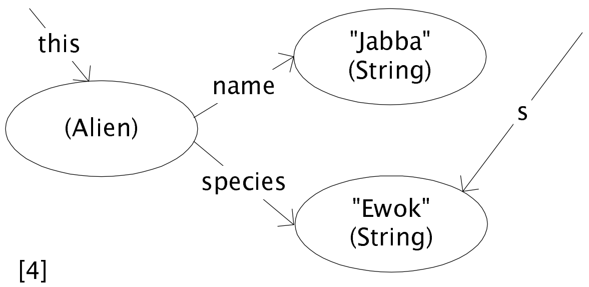
[1]



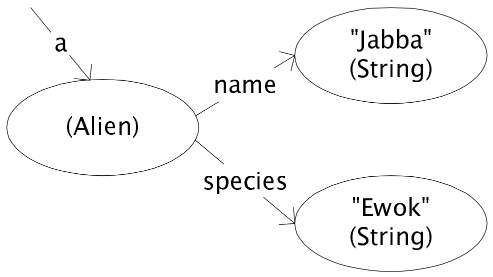
[2]



[3]



[4]



```
void marry (Alien spouse) {
    String save = name;
    name = spouse.name;
    spouse.name = save;
    }
```

and figure out the effect of the following code by drawing some object diagram cartoons:

```
    Alien a = new Alien ("Jabba", "Hut");
    Alien b = new Alien ("Wicket", "Ewok");
    a.marry (b);
    a.print ();
```

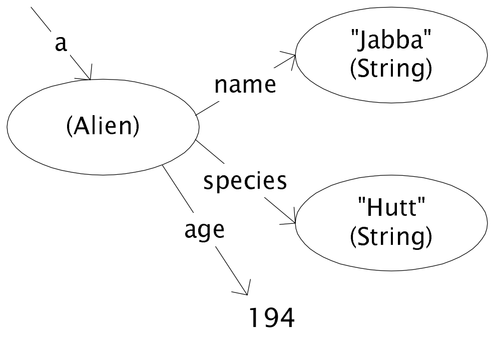
What do aliens do when they marry? As a sign of affection, they exchange names. So the Alien previously called *Jabba The Hutt* is now called *Wicket the Hutt*.

2.6 Primitive Values

So far, all our variables and fields have had values that are objects (or more precisely, references to objects). Java also has primitive values. Our alien, for example, may have an age:

```
class Alien {
    String name;
    String species;
    int age;
    }
```

The age field takes on an integer value. The lowercase name for the type is a Java convention that signals a primitive type. We can show the primitive value in an object diagram like this:



You can't apply a method to a primitive value. In the expression 124 + 1, the plus sign is a built-in arithmetic operator. But in the expression "Jabba" + "Hutt", the plus sign is short for a call to a concatenation method.

There are a handful of other primitive types in Java; these include other integral types (such as long, a 64-bit integer), floating point types (float and double), and boolean. Java has some (surprising) implicit conversions amongst numeric types. You can read about them in the Java Language Specification.

Sometimes,you need to make objects for primitive values. There is a class corresponding to each primitive type. The Boolean class, for example, has two objects, which can be referred to as Boolean.TRUE and Boolean.FALSE. But these are objects, and so both of these

```
    System.out.println (Boolean.TRUE.equals (true));
```

System.out.println (Boolean.TRUE == true);

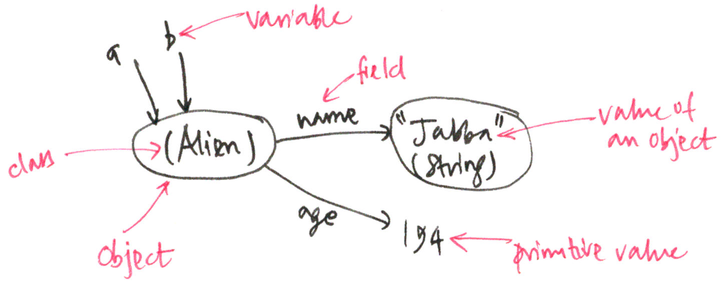
will be rejected by the compiler, since the boolean literal *true* is not an object. (More precisely, we should say that *true* is not an object *reference*. As explained above, *null* is not an object, but we can compare an expression to null with ==. The object equality operator takes two object references, and returns true if they are references to the same object.)

2.7 Summary

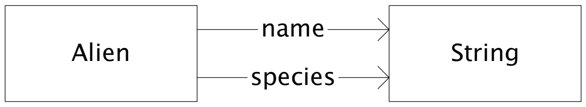
In this lecture, we've introduced a number of fundamental notions:

- variables, objects and fields
- constructors and methods
- setters and getters
- mutable vs. immutable objects
- object equality (==) vs. value equality (.equals)

We've drawn a bunch of diagrams to illustrate Java states, of this form:



Occasionally, these are useful to make a point about a particular state. But usually we want to talk about sets of states, and we'll need a more powerful notation for that. Next time we'll see how the sets of states that can arise from the code we wrote can be described by a diagram like this:



Lecture 3: Classes, subclasses & inheritance

In today's lecture, we'll look at the notion of subclassing, in which the implementation of one class can implicitly incorporate code from another class, using a mechanism known as *inheritance*. We'll talk about interfaces and the role they play in Java next week, when we consider namespace issues.

Our focus will continue to be on the structure of the state. In a later lecture, we'll address the question of the behaviour of subclasses, in particular whether a subclass's behaviour must in some way conform to the behaviour of its superclass.

We'll express the structure of the state using *object models*. An object model is a description of a (usually infinite) set of states, each being a configuration of objects. In our last lecture, we used object diagrams to show individual states; once we have object models, we'll only need to fall back on object diagrams occasionally when illustrating a particular state.

Object models can be used both to describe the structure of the state at the level of the executing Java code, and at a more abstract level of the problem domain. We'll see how to use object models for characterizing problems later, and for now, we'll give them a very concrete interpretation.

Although the elements of object models are very simple to understand, there are several important subtleties that give the notation more power than it might at first seem to have. Read the technical note on code object models available on the class website for a summary. We'll be using object models extensively throughout the course, so it's important that you master them early on.

Because we focus on the fundamental notions, and their expression with object models, we will not be able to survey all the details of Java's class mechanism. You should read one of the recommended Java texts to learn about the subtleties of subclassing. Constructors are an especially tricky (and often confusing) topic that you should make sure you understand.

3.1 Classes, fields & methods

Here's a simple class for a bank transaction with a constructor and two fields:

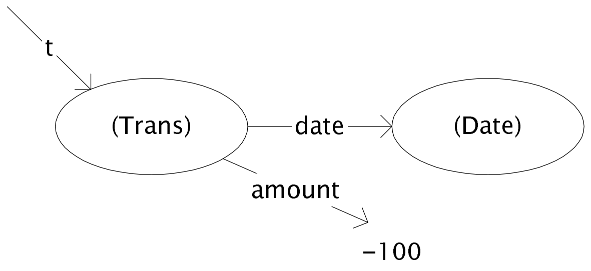
```
class Trans {
    int amount;
    Date date;
    Trans (int a, Date d) {
        amount = a; date = d;
        }
    }
```

The class *Date* is part of the Java library; it belongs to the package *java.util*, along with *Vector*. To refer to it without the package name, you'll actually need an import statement. We'll discuss this issue in the lecture on namespace.

To create a new object we call the constructor with the special keyword *new*. Here's a withdrawal of $100, done now:

```
Trans t = new Trans (-100, new Date ())
```

which results in the state:



Note the call to the *Date* constructor also: this will create a date object corresponding to the date and time at which the call is executed. Here's a more interesting class:

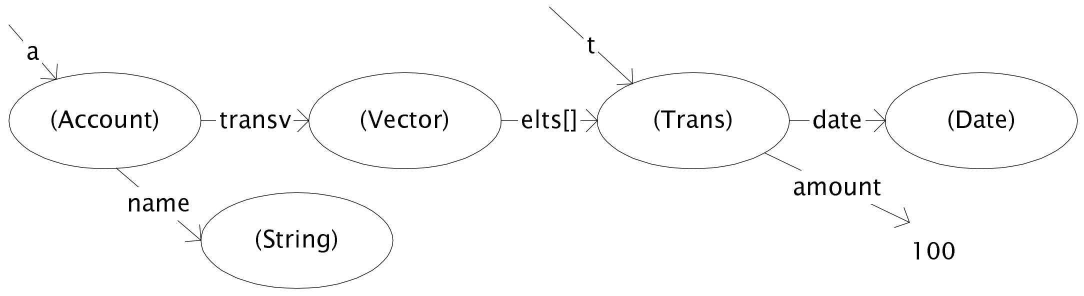
```
class Account {
    String name;
    Vector transv;
    int balance;
    Account (String n) {
        transv = new Vector ();
        balance = 0;
        name = n;
        }
    boolean checkTrans (Trans t) {
        return (balance + t.amount >= 0);
        }
    void post (Trans t) {
        transv.addElement (t);
        balance += t.amount;
        }
    }
```

In addition to the constructor, we have a method for posting a transaction, and a method (to be called first) that checks whether a transaction is allowed. The transactions are stored in a vector.

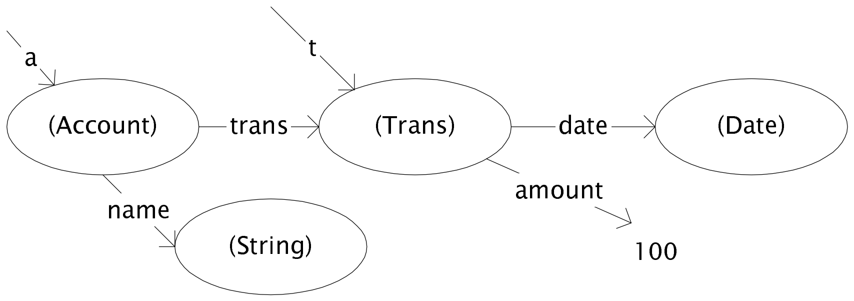
What's the result of this code?

```
Account a = new Account ("Zeeb");
Trans t = new Trans (100, new Date ());
if (a.checkTrans (t))
    a.post (t);
```

It results in a state shown in this object diagram:



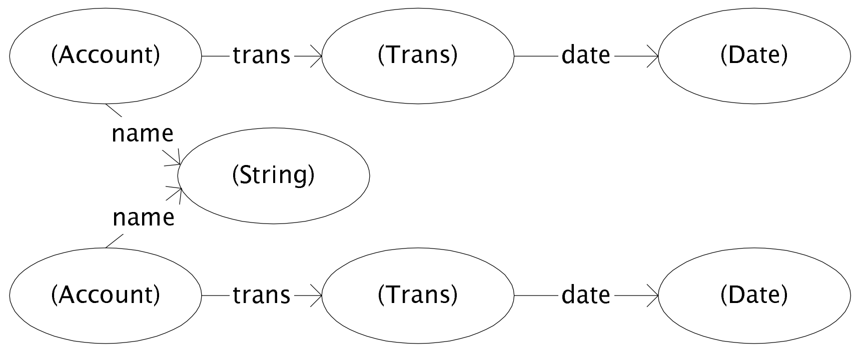
Often, we'll want to elide collection objects such as the vector and only show the more interesting user defined objects. The *Vector* is part of the *representation* of the *Account* object, and a client that calls the methods of *Account* sees only a *Trans* and not the *Vector* itself. So we might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

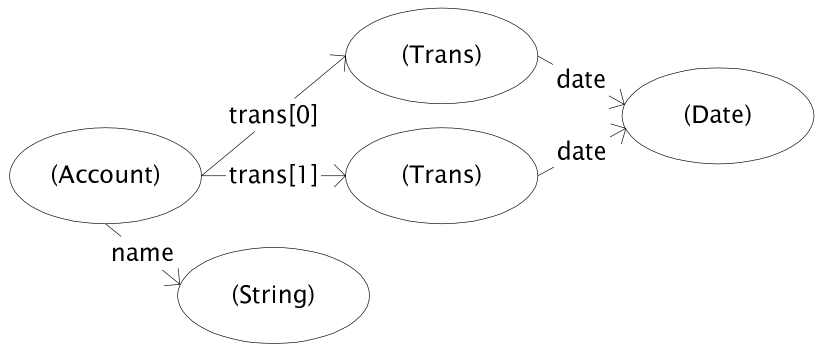


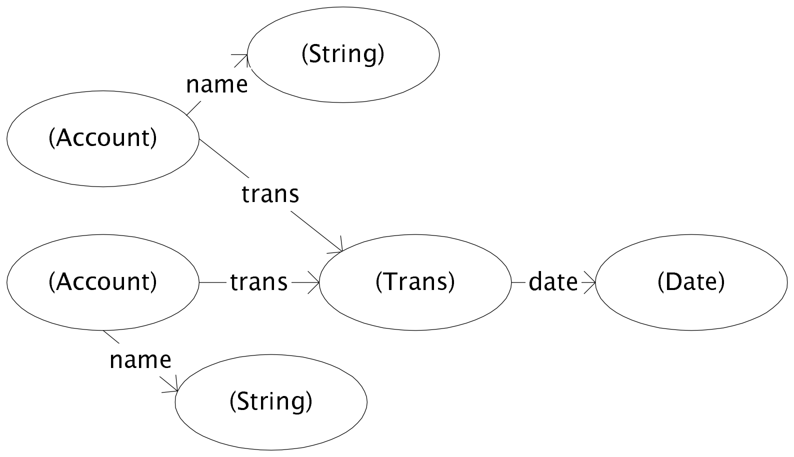
in which the abstract field arrow labelled *trans* from the *Account* object to the *Trans* object hides the *Vector*. This isn't actually new; we did the same thing with *Vector* itself, not showing that it was implemented with an array.

Many states can be created using these classes, but not all of them will be desirable. Three are shown below. In the first, two Accounts share a name; this will mean that names cannot be used as unique identifiers. In the second, two transactions in an account have the same Date object (and are thus recorded as happening simultaneously). In the third, 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. So we can't say for sure whether these configurations are right or wrong. Our interest in this lecture is just raising the issue, and showing how we can *record* a decision in an object model. 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, I'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.



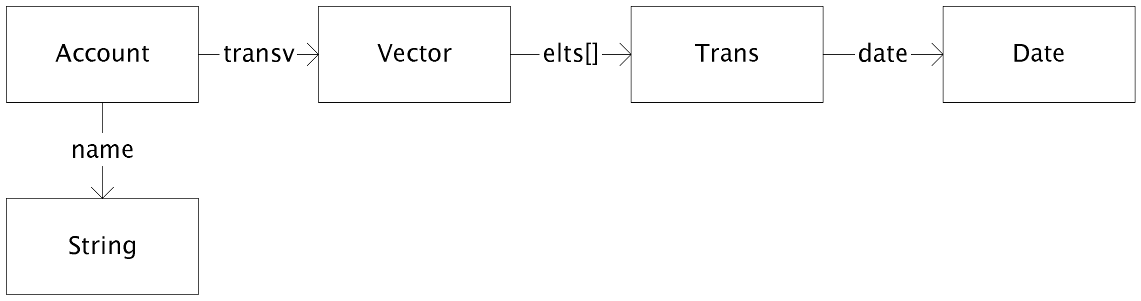




Sometimes we *will* want to talk about the representations of objects, and for these we will not want to elide intermediate objects, such as the vector. The diagrams below show two problematic states that might arise. Both can be created by clients of the code we have seen, but both should not be. The first embodies a rather subtle problem. If two Account objects share a vector, an execution of the *post* method on one will cause a transaction to tbe added to the vector, 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. The second is simpler: it shows a violation of the representation invariant that the transaction vector should only hold Trans objects. Both of these states can be prevented by using Java language mechanisms, and by coding the Account class more carefully.

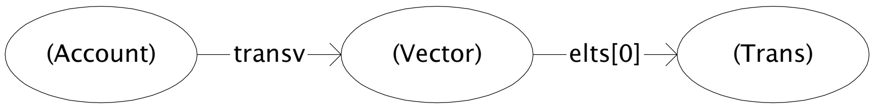
3.2 Object Models of Code

We can summarize a set of states with an *object model*:

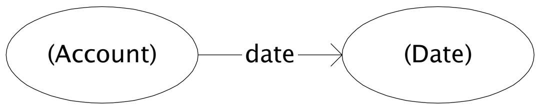


Each box corresponds to a class: it represents the set of all objects that belong to that class (in some given state). The arrows are *relations*, sometimes called *associations*, and they represent the fields that connect classes. For example, the arrow labelled *transv* from Account to Vector shows that each object of the Account class has a field whose value is a Vector object.

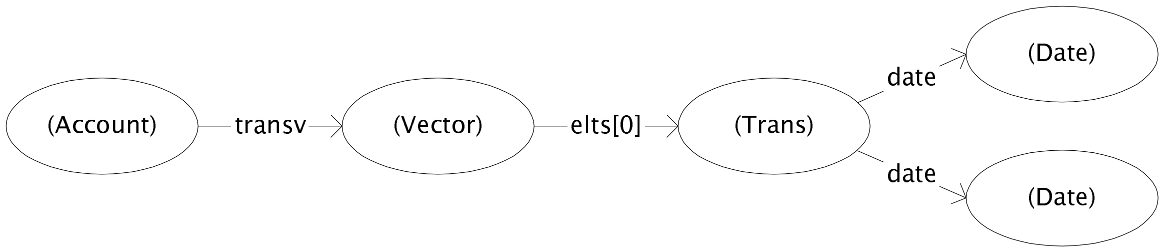
The object model specifies a set of object diagrams, by imposing the constraint that each object in the diagram belong to one of the object model boxes, and that any field arrow in the object diagram connect objects from the appropriate boxes. So our object model allows, for example, this state:



but not this state:

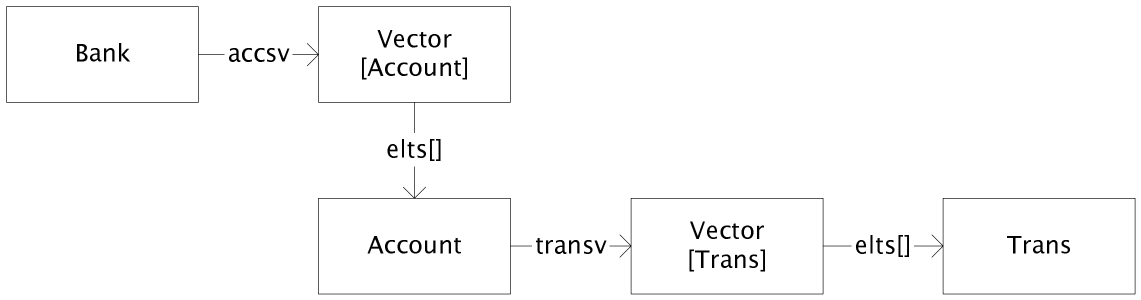


Note that the object model (at least without multiplicity constraints, which we'll talk about below) can't require that any object or field be present; it says what *cannot* be present. The model above places no constraints on the relative numbers of objects, so it admits crazy states such as:



which the code clearly rules out.

Even this simple object model is a bit more subtle than it might first appear to be. Although Account represents all objects of that class, Vector will not represent all objects of the class Vector. Different vectors hold different types of object. We'll come back to this later; it's called *polymorphism*. So while we would never expect to see two boxes labelled "Account" in an object model, we might have two boxes labelled "Vector". For example, we might have a Bank class with a field *accsv* holding a vector of Account objects*:*



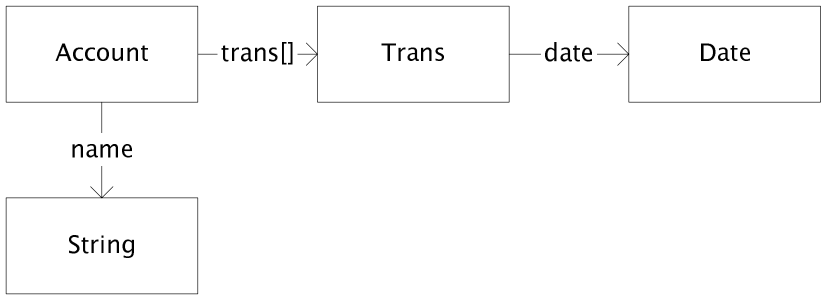
When this happens, it's convenient to annotate the polymorphic class with the class of its elements: Vector [Account], for example, means that the box represents Vectors that hold Account objects.

How do we know that the Vector associated with the Account object always holds Trans objects and not any other kinds of object? Last time we saw that we can call *addElement* on a String, for example. Well, we see that the only call to *addElement* in the code of Account takes an object that has been declared to be a Trans. That doesn't actually clinch it though. It's possible to add an object to the vector from the outside:

Account a = new Account ("Zeeb");
a.transv.addElement ("Zork");

This is bad, because presumably we're planning to add other methods to Account that will do things to the transactions, and such methods will fail if there are elements of the vector that aren't transactions. In a future lecture, we'll talk about how to prevent this kind of thing from happening; it'll rely both on Java language mechanisms and a strategy for building abstract datatypes.

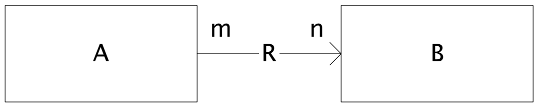
In the same way that we abstracted away Vectors in the object diagram, we can abstract away the Vector class in the object model:



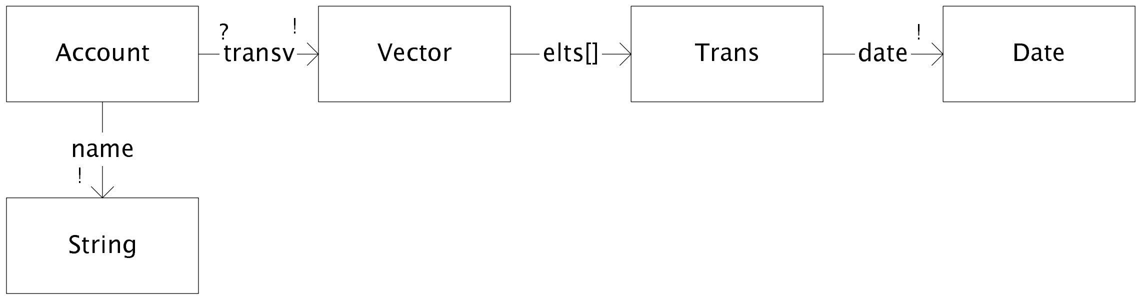
3.3 Multiplicity

This 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:

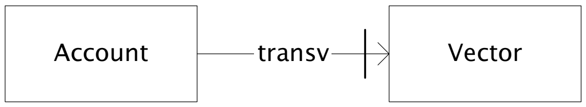
- Account to Vector. The ? on the tail of the arrow from Account to Vector tells us that each of these Vector objects is associated with at most one Account object. That is, Account objects don’t share Vectors: two different Account objects must have different *transv* fields.
- Vector to Trans. The 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. Again, the lack of a symbol says that two Trans objects may share the same Date.

Sometimes multiplicity constraints are enforced by the code. In this case, none of these are . It is possible to write code that uses the classes we have defined to create states that violate these constraints. As a simple example, we can set the *date* field of a Trans to null. We’ll see later how we can enforce some of these constraints in the code of the classes themselves by using mechanisms built into Java: to enforce this one, we can make the *date* field private so that only the constructor sets it, and it guaranteed never to be assigned the null reference.

There are some constraints that cannot be enforced like this however, and for these the object model is even more valuable (because it says something that can’t be said in the code). 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 can 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.

3.4 Mutability

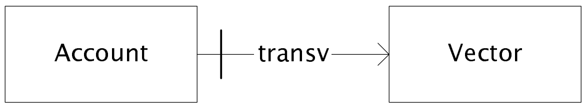
An object model can also show mutability information: how the relationships between objects are allowed to change. We expect transactions to be added to an account, but we don’t expect the vector holding the transactions to be replaced. This can be shown in the object model by marking the end of the 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 right end of a relation is hatched, the relation is said to be *right static*. Should the name relation from Account to String be right 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 left 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 satisifies this, since the constructor always creates a fresh vector rather than reusing an old one.

3.5 Extending a class with inheritance

Suppose we want to implement a new kind of account that allows overdrafts. We might call it AccountPlus, and code it like this:

```
class AccountPlus extends Account {
    int creditLimit;
    AccountPlus (String n, int c) {
        super (n);
        creditLimit = c;
        }
    boolean checkTrans (Trans t) {
        return (balance + creditLimit + t.amount >= 0);
        }
    void bump (int i) {
        creditLimit += i;
        }
    }
```

The keyword *extends* indicates that the implementation of AccountPlus extends the implementation of Account by adding some new features. AccountPlus is said to *inherit* features from Account; AccountPlus is a *subclass* of Account, and Account is a *superclass* of AccountPlus.

There is a new field, creditLimit, and a new method, bump, which increases the credit limit. Because a new AccountPlus object needs to have the new field initialized, AccountPlus must have its own constructor; this actually calls the constructor of Account (see Java text for details of this slightly strange syntax). All the other methods and fields of Account are implicitly present in AccountPlus.

The method checkTrans appears again in AccountPlus, with different code in its body. This is called *overriding*. When the code *acc.checkTrans* is executed, which method actually gets called will depend on whether the object referenced by *acc* is an Account object or an AccountPlus object. The method call is said to be *dynamically resolved*.

At runtime, each object has a type, equal to the class whose constructor created it. A variable that appears in the code also has a type, given by its declaration at compile-time. At runtime, a variable can refer to an object whose type is *not* the variable's type; it is sufficient that the object type be a subclass of the variable type.

(For now, by the way, we're using the term *type* to mean classification by class name, to distinguish it from the term *class* which usually carries the connotation of the code in the class too. Later in the course, we'll be more precise about what *type* means.)

Sometimes, it will be clear in the code what type an object will have at runtime:

```
AccountPlus acc = new AccountPlus ("Zeeb", 100);
Trans t = new Trans (100, new Date ());
if (acc.checkTrans (t))
    acc.post (t);
```

In this case, since acc is declared to be of type AccountPlus, and AccountPlus has no subclasses, we know that the method of AccountPlus will be called.

Suppose we want to handle a collection of accounts. We might have a Bank class, implemented something like this:

```
class Bank {
    Account [] accounts;
    …
    void chargeMonthlyFee () {
        for (int i = 0; i < accounts.length; i++) {
            Trans fee = new Trans (-1, new Date ());
            if (accounts[i].checkTrans (fee))
                accounts[i].post (fee);
            }
        }
    …
    }
```

A bank object holds an array of Account objects. An array is an object just like a Vector, but it can't grow or shrink dynamically. This Bank is unusual: it doesn't hit you when you're down. If deducting the monthly fee would take you below your limit, it won't do it.

The method chargeMonthlyFee works whether the accounts in the array are regular accounts (in the Account class), or special accounts (in the AccountPlus class). The reason is that the *declared type* given in the code says only that the object at runtime will belong to that class or one of its subclasses. But at runtime, which method is selected will depend on the *runtime type* of the object. This code is said to be 'polymorphic', meaning 'many shapes', since the same piece of code text can handle different types of account. If the accounts array contains two objects of the class Account, and a third object of class AccountPlus, the first and second time round the loop the call to the method *checkTrans* will execute code from Account, but the third time round, it will execute the code from AccountPlus. The call to *post* will always call the same code, since it appears only once (in Account), although sometimes it will be called for an Account object, and sometimes an AccountPlus object.

Question: which constraint of the object model (of those we discussed) would be violated if the statement

```
Trans fee = new Trans (-1, new Date ());
```

were hoisted out of the loop, and done just once at the start of *chargeMonthlyFee*?

3.6 A Template Method

Instead of making the client of the account class call the checkTrans method, we could call it inside the post method like this:

```
    boolean post (Trans t) {
        if (!checkTrans (t)) return false;
        transv.addElement (t);
        balance += t.amount;
        return true;
        }
```

Look at the context this method sits in:

```
class Account {
    boolean post (Trans t) {...}
    boolean checkTrans (Trans t) {...}
    }
class AccountPlus extends Account {
    boolean checkTrans (Trans t) {...}
    }
```

Which checkTrans method gets called inside post? It depends on the runtime type of the receiver. Although post belongs to the class Account, we cannot assume that *self* will be an object whose dynamic type is Account. Since the post method is not overridden in the subclass, executing acc.post when acc is an AccountPlus object will cause the post method of Account to be executed; inside it, the checkTrans method of AccountPlus will then be called. So although the post method only appears in the code once, it actually behaves differently for AccountPlus and Account objects.

3.7 Downcasting

Arrays aren't very convenient to program with, since they can't grow or shrink. Suppose we implement Bank with a vector or accounts instead:

```
// bad code!
class Bank {
    Vector accounts;
    ...
    void chargeMonthlyFee () {
        for (int i = 0; i < accounts.size(); i++) {
            Trans fee = new Trans (-1, new Date ());
            if (accounts.elementAt (i).checkTrans (fee))
            accounts.elementAt (i).post (fee);
        }
    }
...
}
```

Vectors are provided as part of the standard Java library, but they're not part of the language itself. So there's no special syntax to access a vector element: you have to call a method (here, elementAt (i) to get the ith element). Also, when you declare a Vector, you can't say what it's a vector *of*. The elementAt method has this signature:

```
Object elementAt (int i)
```

It returns an object of class Object, the superclass of all classes.

So there's no way to know that the expression accounts.elementAt(i) will actually evaluate to an Account or an AccountPlus object. For this reason, the code above will actually be rejected by the Java compiler. Instead we have to write this:

```
void chargeMonthlyFee () {
    for (int i = 0; i < accounts.size(); i++) {
        Trans fee = new Trans (-1, new Date ());
        if (((Account) accounts.elementAt (i)).checkTrans (fee))
                ((Account) accounts.elementAt (i)).post (fee);
        }
```

or better

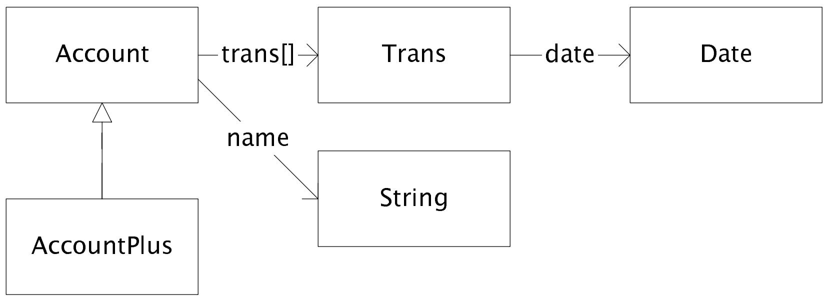
```
void chargeMonthlyFee () {
    for (int i = 0; i < accounts.size(); i++) {
        Trans fee = new Trans (-1, new Date ());
        Account acc = (Account) accounts.elementAt (i);
        if (acc.checkTrans (fee))
            acc.post (fee);
        }
```

The *(Account)* on the fourth line is called a *downcast*. At runtime, it checks that the object returned by the expression belongs to Account or one of its subclasses. If it does, execution continues normally; if it does not, the program is terminated with a ClassCastException. We'll talk about exceptions in our next lecture. For now, it's important just to understand that if execution continues at the next line, the object bound to acc is guaranteed to be of class Account or AccountPlus, and must therefore have the post method. So the Java compiler will accept this code, since the presence of the downcast ensures that there will be no attempt to call a method that does not exist.

People are often confused about downcasts, and think that some kind of conversion is taking place. This is not true. The downcast is simply a test; no change to the object occurs.

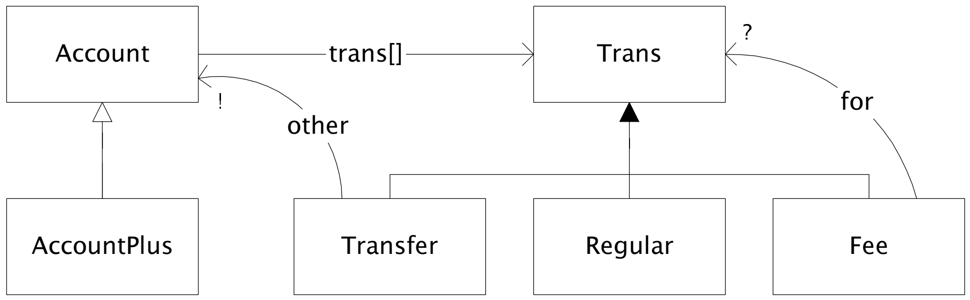
3.8 Subclassing in the Object Model

Here's 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.

3.9 Static class members

Suppose we want to have a maximum credit limit allowed in special accounts. We can declare a static field, maxCreditLimit, in AccountPlus like this:

```
class AccountPlus extends Account {
    static int MAX_LIMIT;
    int creditLimit;
    ...
    }
```

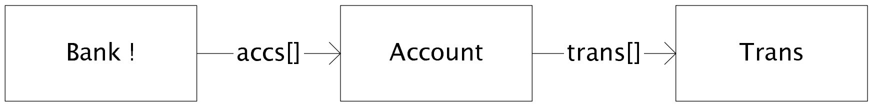
Being *static* means that there will be one value of *MAX_LIMIT* for the class as a whole; the field creditLimit, in contrast, has a value for each object of the class.

Sometimes, we only want to create a *singleton object*: we want it to belong to a class, since the object has code associated with it, but the class should contain no other objects. Here's an example:

```
class Bank {
    static Bank theBank = new Bank ();
    Vector accounts;
    ...
    }
```

The static variable theBank is initialized by creating a single Bank object and assigning it to the variable. We can now refer to this object as Bank.theBank; we'll see later that Java provides mechanisms that will allow us to hide the constructor, so that it's not even possible for a client of the Bank class to create other Banks.

To show this singleton in the object model, we use a multiplicity markings in the box labelled Bank, to show that it contains exactly one object:



Other multiplicity constraints on classes are illustrated in the technical note on Code Object Models.

3.10 Summary

We have had a brief tour of some fundamental notions of object-oriented programming:
· subclassing and inheritance
· dynamic resolution of method calls
· downcasts
· static members

We have also seen all the elements of code object models:
· sets and subsets
· field arrows, concrete and abstract
· multiplicity
· mutability

We saw how the object model captures succinctly some properties that are evident from the code, but can also express constraints that are intended by the designer of a class (or collection of classes), but which cannot be enforced using language mechanisms. This is another reason that object models are a more useful design notation than code or code sketches.

Lecture 4: Exceptions

In this lecture, we'll look at Java's exception mechanism. As always, we'll focus more on design issues than the details of the language, which can be found in one of the recommended Java texts. We'll discuss the general issue of making your code more robust with runtime assertions; how to use exceptions for communicating back to callers of a method, even in non-failure situations; and how to decide whether to use checked or unchecked ex ceptions.

4.1 Defensive Programming

Suppose you are designing a compiler. Since the compiler is under constant development, and is large and complex, it is likely to contain some bugs despite your best efforts. *Defensive programming* is a way to mitigate the effects of bugs without knowing where they are. Adopting defensive strategies is not an admission of incompetence. Although any large program is likely to have bugs in it, even a bug-free program can benefit from defensive programming. You may have to deal with a database that may occasionally be corrupted. It would be too hard to predict every possible form of corruption, so you take a defensive strategy instead.

Here's how defensive programming works. When you're writing some code, you figure out conditions that you expect to hold at certain points in the code; these are called *invariants* and we'll have a lot more to say about them later. Then, rather than just assuming that these conditions hold, you test for them explicitly. If a condition is false, you abort the computation.

For example, suppose that in your compiler you maintain a vector of symbols, and some other vectors, each of which contains objects holding information about symbols.

```
class Symbols {
    Vector symbols;
    Vector types;
    Vector positions;
    ...
    }
```

To access the information for a given symbol, you have to access the element of the information vector with the same index. Given a symbol *sym*, to obtain its type for example, we take the element of the *types* vector in the same position:

```
int i = symbols.indexOf (sym);
Type t = types.get (i);
```

This code will only work if the symbol being looked up is indeed in the *symbols* vector, and if the two vectors have the same length, so that the indexing in the second statement succeeds. If the second statement fails, then something has gone wrong, and it is unlikely that the compiler will be able to generate correct code. So it would be best to waste no more resources and terminate the program immediately. In fact, we can do better and notice the problem after the first statement.

We do this by inserting a *runtime assertion:*

```
int i = symbols.indexOf (sym);
if (! (i >= 0))
    System.exit (1);
...
Type t = types.get (i);
```

If the call to *indexOf* returns -1, indicating that the symbol is not in the vector, we terminate the entire compiler by calling the special method *exit* of the System class.

To make runtime assertions easy to write, most programmers define a procedure, so they can write, for example:

```
assert (i >= 0)
```

This also demonstrates the documentation value of assertions. Even if they are not executed, they help someone reading the code immeasurably to understand what it's doing.

Two important questions arise:

· Where and what kind of runtime assertions are best?
· How should you abort execution?

4.2 Runtime Assertions

We'll have much more to say later in the course about what runtime assertions to write (when we study preconditions and representation invariants). There are some useful general guidelines though.

First, runtime assertions shouldn't be used as a crutch for bad coding. You want to make your code bug-free in the most effective way. Defensive programming doesn't mean writing lousy code and peppering it with assertions. If you don't already know it, you'll find that in the long run it's much less work to write good code from the start; bad code is often such a mess it can't even be fixed without starting over again.

When should you write runtime assertions? As you write the code, not later. When you're writing the code you have invariants in mind anyway, and writing them down is a useful form of documentation. If you postpone it, you're less likely to do it.

Runtime assertions are not free. They can clutter the code, so they must be used judiciously. Obviously you want to write the assertions that are most likely to catch bugs. Good programmers will typically use assertions in these ways:

· At the start of a procedure, to check that the state in which the procedure is invoked is as expected. This makes sense because a high proportion of errors are related to misunderstandings about interfaces between procedures. You'll understand this better when we walk about preconditions.
· At the end of a complicated procedure, to check that the result is plausible. In a procedure that computes square roots for example, you might write an assertion that squares the result to check that it's (roughly) equal to the argument. This kind of assertion is sometimes called a *self check*.

- When an operation is about to be performed that has some external effect. For example, in a radiotherapy machine, it would make sense to check before turning on the beam that the intensity is within reasonable bounds.

Runtime assertions can also slow execution down. Novices are usually much more concerned about this than they should be. The practice of writing runtime assertions for testing the code but turning them off in the official release is like removing seat belts from a car after the safety tests have been performed. A good rule of thumb is that if you think a runtime assertion is necessary, you should worry about the performance cost only when you have evidence (eg, from a profiler) that the cost is really significant.

Nevertheless, it makes no sense to write absurdly expensive assertions. Suppose, for example, you are given an array and an index at which an element has been placed. It would be reasonable to check that the element is there. But it would not be reasonable to check that the element is nowhere else, by searching the array from end to end: that might turn an operation that executes in constant time into one that takes linear time (in the length of the array).

4.3 Responding to Failure

Now we come to the question of what to do when an assertion fails.

You might feel tempted to try and fix the problem on the fly. This is almost always the wrong thing to do. It makes the code more complicated, and usually introduces even more bugs. You're unlikely to be able to guess the cause of the failure; if you can, you could probably have avoided the bug in the first place.

On the other hand, it often makes sense to execute some special actions irrespective of the exact cause of failure. You might log the failure to a file, and/or notify the user on the screen, for example. In a safety critical system, deciding what actions are to be performed on failure is tricky and very important; in a nuclear reactor controller, for example, you probably want to remove the fuel rods if you detect that something is not quite right.

Sometimes, it's best *not* to abort execution at all. When our compiler fails, it makes sense to abort completely. But consider a failure in a word processor. If the user issues a command that fails, it would be much better to signal the failure and abort the command but not close the program; then the user can mitigate the effects of the failure (eg, by saving the buffer under a different name, and only then closing the program).

How do you structure an abort of a command in the code? With the mechanisms we've seen so far, it's extremely tedious. You'd need to give each procedure a special return value that indicates whether it succeeded or failed, then at every call you'd need to check this value. When a procedure call fails, the calling procedure must then itself return a failure value. Your code will look something like this:

```
boolean p (...) {
    boolean success = q ( ...);
    if (!success) return false;
    success = r (...);
```

```
        if (! success) return false;
        ...
        }
```

Aside from being tedious (and depriving you of additional return values -- a big problem in a language that only allows one return value), this approach has a serious modularity problem. Suppose there is some part of the program that is expected not to fail, and is thus not coded in this special style. Now you make some change deep in the procedure call hierarchy, introducing a procedure that might fail. Now its caller, and the caller of its caller, and so, all the way to the top, must be modified. Aaagh!

4.4 Non-local Jumps

What we need is a mechanism for making a *non-local jump*. Look at this code:

```
class Symbols {
    Vector symbols;
    Vector types;
    Vector positions;
    Type getType (Symbol s) {
        int i = symbols.indexOf (s);
        if (! (i >= 0))
            throw new RuntimeException ("getType");
        }
    ...
    }
class Compiler {
    Symbols symbols;
    ...
    void compile () {
        try{
            parse ();
            typecheck ();
            optimize ();
            generate ():
            }
        catch (RuntimeException e) {
            System.out.println ("Failed at: " + e.getMessage ());
            }
        }
    void typeCheck () {
        ...
        Type t = symbols.getType (s);
        ...
        }
    }
```

This code shows how exceptions can be used in our compiler to do abort without killing the entire execution. Even in the compiler this is useful, because it lets us print an informative error message. If the assertion in *getType* fails, an exception is 'thrown'. This not only causes the execution of *getType* to abort, but also the execution of *typeCheck*, its caller. Execution of each caller in the call stack is aborted, until a method such as *compile* is reached that has a *handler* for the exception. Now control is transferred to the handler, labelled *catch*, and its code is executed.

The throw command doesn't just transfer control: it passes an object along too. This allows information about a failure to be passed. The constructor for the exception takes a message string which is extracted in the handler. Here we've used it to convey information about where in the code the failure occurred. This is a common strategy; you can make it more useful by adding more details, such as the particular symbol that was looked up.

4.5 Exceptions for Special Results

Exceptions are not just for handling failures. They can be used to improve the structure of code that involves procedures with special results.

A standard way to handle special results is to return special values. Lookup operations in the Java library are often designed like this: you get an index of -1 when expecting a positive integer, or a null reference when expecting an object. This approach is OK if used sparingly. It has two problems though. First, it's tedious to check the return value. Second, it's easy to forget to do it. We'll see that with exceptions you can get help from the compiler in this.

(Why is Java designed like this? Probably because its inventors viewed exceptions as things to be used in exceptional cases, and not routinely. CLU, the language in which 6170 used to be taught, invented here at MIT by Barbara Liskov, supported a more pervasive use of exceptions. Programs in CLU used exceptions whenever a procedure's behaviours could not be easily accommodated by a single type of return value. CLU implemented exceptions very efficiently, so that when they were not thrown, the overhead was minimal. It also had a very lightweight syntax for exception handlers. Unfortunately, exceptions are expensive both in runtime and in programmer effort in Java. A friend of mine measured the cost of writing a loop that iterates over the elements of an array in two ways: one terminating by checking the index with a conditional expression, the other terminating by catching an array out of bounds exception. Throwing and catching the exception cost about the equivalent of 500 loop iterations.)

Sometimes its not easy to find a 'special value'. Suppose we have a *BirthdayBook* class with a lookup method. Here's one possible method signature:

Date lookup (String name)

What should the method do if the birthday book doesn't have an entry for the person whose name is given? Well, we could return some special date that is not going to be used as a real date. Bad programmers have been doing this for decades; they would return 9/9/99, for example, since it was *obvi-*

ous that no program written in 1960 would still be running at the end of the century.

Because of this, some pundits predicted failures on September 9th last year. The Wall Street Journal reported that 9/9/99 was a non-event, although there were apparently a few incidents that may be attributable to this bug, including a banking failure. See the following entries in the RISKS forum:

- http://catless.ncl.ac.uk/Risks/20.55.html#subj5.1
- http://catless.ncl.ac.uk/Risks/20.60.html#subj8.1

Here's a better approach. The method throws an exception:

```
Date lookup (String name) throws NotFoundException {
    ...
    if // not found
        throw new NotFoundException ();
    ...
```

and the caller handles the exception with a catch clause. Now there's no need for any special value, nor the checking associated with it.

Novice Java programmers often make the mistake of using null references as special values. Be warned that this is a bad idea, and can lead to code riddled with null dereferencing problems.

4.6 Checked and Unchecked Exceptions

We've seen two different purposes for exceptions: failures and special results. Java provides two different kinds of exception for these two purposes. They behave the same at runtime; the only difference is what kind of checking the compiler provides.

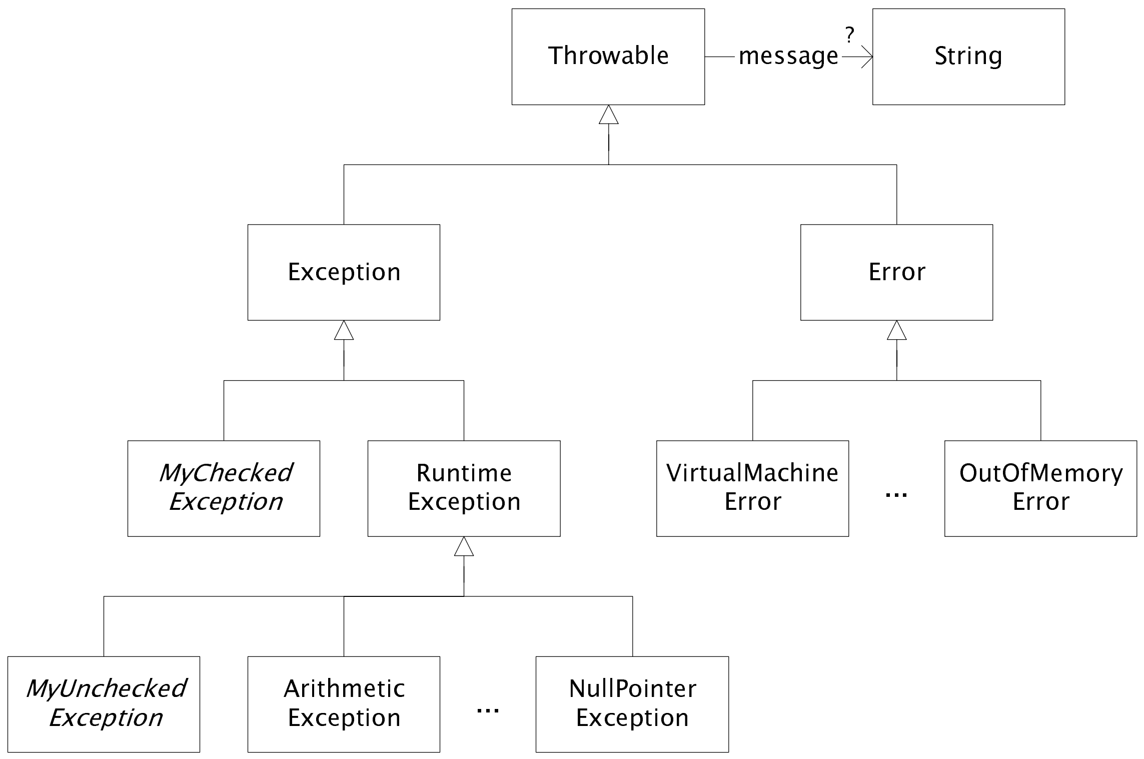
If a method might throw a *checked* exception, the possibility must be declared in its signature. *NotFoundException* would be a checked exception, and that's why the signature ends *throws NotFoundException.* If a method calls another method that may throw a checked exception, it must either handle it, or declare the exception itself (since if it isn't caught locally it will be propagated).

So if you call the *lookup* method and forget to handle the exception, the compiler will reject your code. This is very useful, because it ensures that exceptions that are expected to occur should be handled. On the other hand, exceptions that correspond to failures are not expected to be handled except at the top level, and for reasons of modularity, we wouldn't want to declare the possibility of failure at every level.

For an *unchecked* exception, in contrast, the compiler will not check for try-catch or a throws declaration. Java still allows you write a throws clause as part of a signature for an unchecked exception, but this has no effect (and is thus a bit funny, and I don't recommend doing it).

How do you create these two kinds of exception? By using different built-in classes, or subclassing them to create your own exception classes. The object model is shown below.

The superclass of all errors and exceptions is Throwable. It has two direct subclasses, *Exception* and *Error.* All subclasses of *Exception*, bar *RuntimeException* and its subclasses, are checked. The rest -- that is, *Error* and *RuntimeException* and their subclasses -- are unchecked. You may wonder why



there isn't a simply division into two classes at the top level, one checked and one unchecked. The reason is that *Error* is designed to be used for exceptions a program is not expected to catch, since recovery is impossible: these included virtual machine errors, such as running out of memory. *RuntimeException*, on the other hand, is used for bad things that you might well catch, such as arithmetic overflows and class cast errors.

All errors and exceptions may have a message associated with them. If not provided in the constructor, the reference to the message string is null.

4.7 Built-in Java Exceptions

The subclasses of Error will not generally concern you, since there's not much you can do about them. If you run out of memory, you might try just increasing the heap size at the command line. On the other hand, you may actually be using more memory than you really need, because you have a 'conceptual memory leak', in which you're creating lots of objects that never get used after some point, but which can't be garbage collected because they are reachable (eg, because you put them in a table that is reachable).

The subclasses of RuntimeException are more relevant in 6170, and worth studying, because they represent common bugs:

- ArithmeticException: eg, if you divide by zero
- IndexOutOfBoundsException: accessing an array or vector with a bad index
- NullPointerException: calling a method with a null reference as receiver

- ClassCastException: thrown by ((T) e), when expression e does not evaluate to a T

4.8 User-Defined Exceptions

You will want to define your own exceptions. Let's take a second look at the compiler example. We threw RuntimeException on assertion failure, and caught it at the top level. But suppose somewhere in our code an ArithmeticException is thrown, and is not caught anywhere else. Our top level handler for RuntimeException will catch this, since ArithmeticException is a subclass of RuntimeException. This is called "capture": our handler is unexpectedly catching other exceptions. In this case, it may not be too bad, but in general, we want to make sure that inadvertent capture does not occur.

We therefore define our own exceptions, and make the handlers more specific. So instead of throwing RuntimeException in this case, we would thrown our own failure exception:

```
class Symbols {
    Vector symbols;
    Vector types;
    Vector positions;
    Type getType (Symbol s) {
        int i = symbols.indexOf (sym);
        if (! (index >= 0))
            throw new FailureException ("getType");
        }
    ...
    }
class Compiler {
    Symbols symbols;
    ...
    void compile () {
        try{
            typecheck ();
            ...
            }
        catch (FailureException e) {
            System.out.println ("Failed at: " + e.getMessage ());
            }
        }
    }
```

defined like this:

```
class FailureException extends RuntimeException {}
```

4.9 Handling Exceptions

A try statement can have several catch clauses:

```
try {
    statement;
    }
catch (Exception1 e1) {handler1;}
...
catch (ExceptionN eN) {handlerN;}
```

If the statement raises an exception, each catch clause is tested in turn until one is found whose exception class matches the class of the thrown exception object. That is, if the thrown exception has class E, we find the kth clause such that Ex ceptionK is a superclass of E. Equivalently, we find the first handler variable to which the exception object can be assigned. The exception is bound to the variable, and the handler is executed. If no handler is found, the exception is 'reflected' -- propagated to the next enclosing try statement, or by having the method throw the exception again. There is also a *finally* clause that holds code to be executed after any handler: read about it in a Java text.

A handler can itself throw an exception. This is often appropriate, when the exception being handled is not at the right level of abstraction for the caller. For example, a caller of a method called *lookup* should receive a *NotFoundException*, not a *NullPointerException*. Often the handler 'masks' the exception, by causing control to continue normally (eg, by printing a message and carrying on).

4.10 Design Considerations

The rule we have given -- use checked exceptions for special results, and unchecked exceptions to signal failures -- makes sense, but it isn't the end of the story. The snag is that exceptions in Java aren't as lightweight as they might be (compared to CLU, a language which, like Algol-60, was 'an improvement on most of its successors').

Aside from the performance penalty, exceptions in Java incur another (more serious) cost: they're a pain to use. If you design a method to have its own exception, you have to create a new class for the exception. If you call a method that can throw a checked exception, you have to wrap it in a try-catch statement (even if you know the exception will never be thrown). This latter stipulation creates a dilemma. Suppose, for example, you're designing a queue abstraction. Should popping the queue throw a checked exception when the queue is empty? Suppose you want to support a style or programming in the client in which the queue is popped (in a loop say) until the exception is thrown. So you choose a checked exception. Now some client wants to use the method in a context in which, immediately prior to popping, the client tests whether the queue is empty and only pops if it isn't. Maddeningly, that client will still need to wrap the call in a try-catch statement.

This suggests a more refined rule (see course text, p 73):

- You should use an unchecked ex ception only if you expect that clients will usually write code that ensures the exception will not happen, because there is a convenient and inexpensive way to avoid the exception, or because the exception reflects unexpected failures;
- Otherwise you should use a checked exception.

The cost of using exceptions in Java is one reason that many Java API's use the null reference as a special value. It's not a terrible thing to do, so long as it's done judiciously, and carefully specified.

4.11 Safe Languages

Exceptions are an important part of the design of a 'safe' programming language. Safe languages aren't (unfortunately) ones in which buggy programs can't be written. They're simply languages that are designed so that errors of a certain sort are never masked: they are either caught by the compiler (which is ideal, because then they can never occur), or caught at runtime as soon as they happen.

There is a ctually one respect in which safe languages do prevent the writing of buggy code. Memory management errors, such as deallocating the storage associated with an object prior to its last use, cannot occur in Java because memory is managed automatically by the virtual machine, and the progammer does no explicit deallocation.

An important consequence of safety is that the behaviour of the code can always be predicted. In Java, when you're coding a method that takes an argument of class X, you can assume that at runtime the object bound to the argument will belong to a subclass of X. The type system guarantees it. But in the C programming language, such assumptions are invalid. Because code can write to arbitrary memory locations (eg, by exceeding the bounds of an array), there are no guarantees that objects are well formed. Strings are by convention terminated with a special value in C, but you cannot assume that a string argument is correctly terminated. In Java, on the other hand, a string argument will always be a well formed string, on which the string methods behave predictably.

The errors that are prevented by safe languages are most of the errors that are due to program 'anomalies': that is, they are faults in the code that can be identified without any knowledge of what the program is supposed to do. They include: type errors, such as adding a string to an integer, or calling a method on an object that doesn't exist; array bounds errors; returning from a method without a value when one is required; failing to initialize a variable; etc.

To find errors at compile time, the compiler must be *conservative*, which means that sometimes it will reject programs that would not have failed. For example, a Java compiler won't allow this

```
int m () {code_later ();}
void code_later () {throw new RuntimeException ("unimplemented");}
```

and will complain that *m* does not return a value, even though the call in the method *m* will always cause the exception to be thrown so it should not matter. To find errors at runtime, the compiler will usually insert checks. Your downcasts, for example, will be checked at runtime. Because of an oddity in the design of arrays in Java, it also turns out that to enforce safety (for a reason we may discuss later in the context of subtyping and subclassing), a downcast is performed every time an array element is stored.

In practice, these are small prices well worth paying for the increase in reliability they produce. If an error does occur, they also tend to make it less damaging (by terminating the program before damage is done), and easier to trace.

Safe languages have been around since 1960. Famous safe languages include Algol-60, Pascal, Modula, LISP, CLU, Ada, ML, and now Java. It's interesting that for many years industry claimed that the costs of safety were too high, and that it was infeasible to switch from unsafe languages (like C++) to safe languages (like Java). Java benefited from a lot of early hype about applets, and now that it's widely used, and lots of libraries are available, and there are lots of programmers who know Java, many companies have taken the plunge and are recognizing the benefits of a safe language.

4.12 A Cautionary Tale

Exceptions are not a panacea. The Ariane 5 incident is a sad story in this respect. An arithmetic exception was thrown after launch of the European unmanned rocket by a piece of code that was still running, but was used only during launch itself. Because there was no handler, the exception propagated to the top and terminated the entire control program. As a result, the takeoff was aborted and the rocket was destroyed.

Had the code been written in an unsafe language (and not in Ada), it looks as if the disaster would not have occurred, since no exception would have been raised and the irrelevant code would have continued without interfering with the rest of the system, despite producing bad results.

Read about it at: http://www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html

4.13 Summary

Today, we've seen how exceptions can provide non-local jumps. These allow failures to be handled gracefully, and support a common style of defensive programming. Exceptions are also often a better way to convey results than special values.We looked at some features specific to Java, in particular the distinction between checked and unchecked ex ceptions, and how to define your own exceptions.

Lecture 5: Namespace

This lecture is about the gross organization of programs, with a particular emphasis on names and how they are used. We'll look at the mechanisms that Java provides to name parts of the code, and to structure the code of a large system hierarchically. Using Java's access control mechanisms, we'll see how one can prevent access to certain methods and fields outside certain scopes. The hierarchical namespace makes it possible to see what other parts of the code a particular method interacts directly with. Access control takes this further and allows local reasoning; we'll now be able to guarantee some of the object model properties of our banking example from Lecture 3 without knowing about the clients that use that code.

The notion of dependence is essential for doing good software design. We'll define dependences, and see how the coupling they bring about causes problems by making a program less flexible and harder to reason about. We'll examine three language constructs that can be used to reduce dependences: procedures, global variables and Java interfaces. Interfaces will be the most interesting and important of the three.

We'll see how, in an object-oriented language such as Java, the control flow at runtime does not necessarily match the compile-time structure. This will be exposed as mismatches between the object model and the module dependence diagram. It's a tricky notion, but a useful one, since it let's you use dynamic configuration to obtain flexibility. Java interfaces are crucial support for this. Also, many of the design patterns we'll see later rely on this.

As always, our emphasis is on fundamental underlying notions, with an eye to design issues. You'll need to study a Java text to make sure you understand the subtleties of the Java features we discuss in this lecture: packages and access control, static fields, interfaces and abstract classes. We don't discuss inner classes, but you should know how to use them, as they are especially useful when generating objects to satisfy given interfaces.

5.1 Package Structure

Like any large written work, a program benefits from being organized into a hierarchical structure. When trying to understand a large structure, it's often helpful to view it top-down, starting with the grossest levels of structure and proceeding to finer and finer details. Java's naming system supports this hierarchical structure.

It also brings another important benefit. Different components can use the same names for their subcomponents, with different local meanings. In the context of the system as a whole, the subcomponents will have names that are qualified by the components they belong to, so there will be no confusion. This is vital, because it allows developers to work independently without worrying about name clashes.

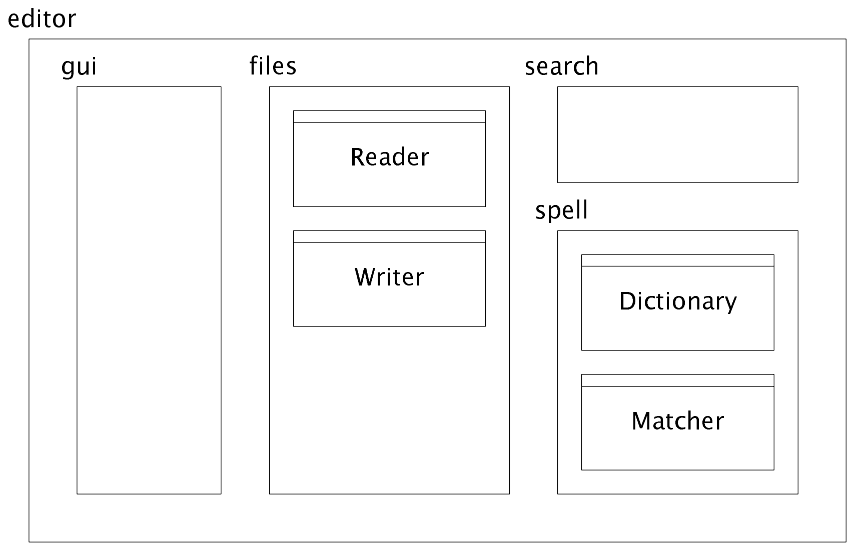
Just because it's useful to view software top-down doesn't mean that it should be *produced* top-down. Early on, people advocated designing software by successively refining a description, building a structure gradually by expanding the nodes of a tree. As you went down the tree, you moved closer to implementation issues; the leaves of the tree were program statements. We now know that this strategy tends not to work at all in practice, since it's usually impossible to make the first decomposition correctly. You don't find out you've made a mistake until you get to the bottom, and then you find you can't build the smallest pieces that are needed to complete the structure, or that they are more complicated than you'd hoped, and interact in complex ways. By then, it's usually too late to change the high-level structure. So top-down *development* a nd top-down *description* are very different things.

Here's how the Java naming system works. The key named components are classes and interfaces, and they have named methods and named fields. Local variables (within methods) and method arguments are also named. Each name in a Java program has a scope: a portion of the program text over which the name is valid and bound to the component. Method arguments, for example, have the scope of the method; fields have the scope of the class, and sometimes beyond. The same name can be used to refer to different things when there is no ambiguity. For example, it's possible to use the same name for a field, a method and a class; see the Java language spec for examples.

A Java program is organized into *packages*. Each class or interface has its own file (ignoring inner classes, which we won't discuss). Packages are mirrored in the directory structure. Just like directories, packages can be nested arbitrarily deeply. To organize your code into packages, you do two things: you indicate at the top of each file which package its class or interface belongs to, and you organize the files physically into a directory structure to match the package structure.

Suppose you're building a text editor for example. You might name the outermost package *editor*. This package may have subpackages *gui* for the user interface, *files* for code managing reading and writing of files, *edit* for code that handles editing functions, and so on. These packages themselves may have subpackages; *edit*, for example, may have *search* for search and replace, *spell* for spelling checks, and so on. Each package may contain classes and interfaces as well as packages, but in many programs the actual code is only in the leaves of the package tree. So the *spell* package may contain classes such as *Dictionary* and *Matcher*, and the *files* package may contain *Reader* and *Writer*.

Here's a diagram that illustrates this structure:



Each component has a *simple name* and a *qualified name*. The simple name of the *Dictionary* class, for example, would be just *Dictionary*; its qualified name woul d be *editor.edit.spell.Dictionary.* To place it in its appropriate package, it would be declared like this:

```
package editor.edit.spell;
public class Dictionary {...}
```

You can always use the qualified name. The simple name is valid within the same package as the component. In *Matcher*, for example, you can refer to *Dictionary.* Sometimes, code needs to refer to many components in another package, and it becomes a nuisance to use qualified names. Java lets you *import* the contents of a package. For example, if you write

```
package editor.edit.spell;
import editor.files.Reader;
public class Matcher {...}
```

then inside the *Matcher* class, you can refer to the *Reader* class of the package *editor.files* by its simple name. To import all the classes of the package, you'd write

```
import editor.files.*;
```

Java calls this 'import on demand' because it's as if an import declaration were added whenever a particular class in the package is used.

Package naming has a lot in common with naming files and directories in a file system where path names are like qualified names. But it's not quite the same. You can't refer to the class *spell.Matcher* by importing *editor.edit*, for example: the import statement imports *components*, and doesn't introduce a new name scope in which relative package names can be used. This import would be legal

```
import editor.edit.*;
```

but its effect is to import all the components that belong to the package *editor.edit*, and this has no effect on how components of packages within *editor.edit* can be subsequently referred to.

5.2 Standard Java Packages

The Java language itself defines some standard packages that come with the standard Java distribution. The basic ones used by all applications are:

· *java.lang.* The root of the class hierarchy, *Object*, has the qualified name java.lang.Object. You'll rarely see this though, since every Java program implicitly imports java.lang.*. The package also contains the classes (such as Integer) that correspond to the builtin types, exceptions such as NullPointerException, the class Math that provides mathematical routines (such as sin and cos), Thread and other classes used to implement concurrency, and so on.

· *java.util.* Contains all the basic collection classes, such as *Vector.* Because this package is not automatically imported, to use vectors you need to import them explicitly.

· *java.io.* Contains classes for doing basic input and output. The object *System.out* belongs to a class in this package (although *System* itself belongs to java.lang).

The package *java.util* has a class *Dictionary* that is an abstract superclass of classes that map keys to values (such as Hashtable). In fact, it's now deprecated (which means it will be made obsolete in a future version of Java). But it serves our point: that there's no problem using our class, editor.spell.Dictionary, and the standard Java class in the same program, because we can distinguish them with qualified names. If you import two collections of classes whose names overlap, the compiler will reject the ambiguous names.

Then there are packages with more specialized uses, such as *java.net* for network communications, and *java.swing* for graphical user interfaces.

Sun also provides what it calls 'standard extensions' to Java. These have package names that start with *javax*. The package *javax.mail*, for example, provides the kinds of components you need to build applications that deal with email.

5.3 Unique Package Names

The package naming system can be used to ensure that software that is widely distributed has a unique name, thereby avoiding the problem of name clashes. The Java convention is to use the standard internet naming system in reverse. So for example, an editor package created here at MIT might be called

edu.mit.editor

Another advantage of this scheme is that it gives a company a little advertisement in the source code. If you see

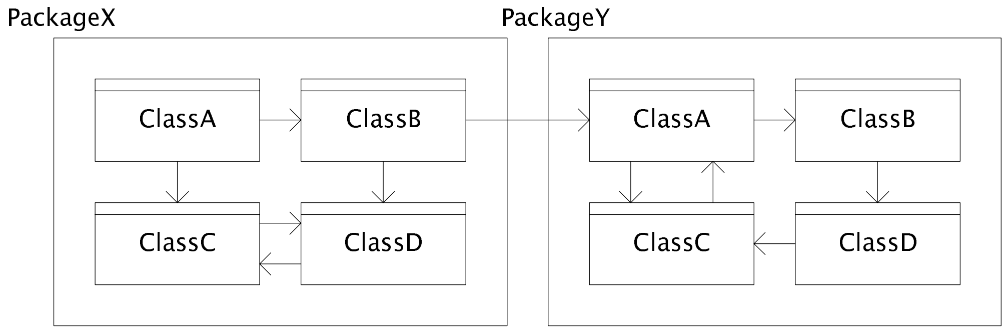
import com.genlogic.;*

you can guess where on the web to find the company that supplied these widgets (in this case for diagramming).

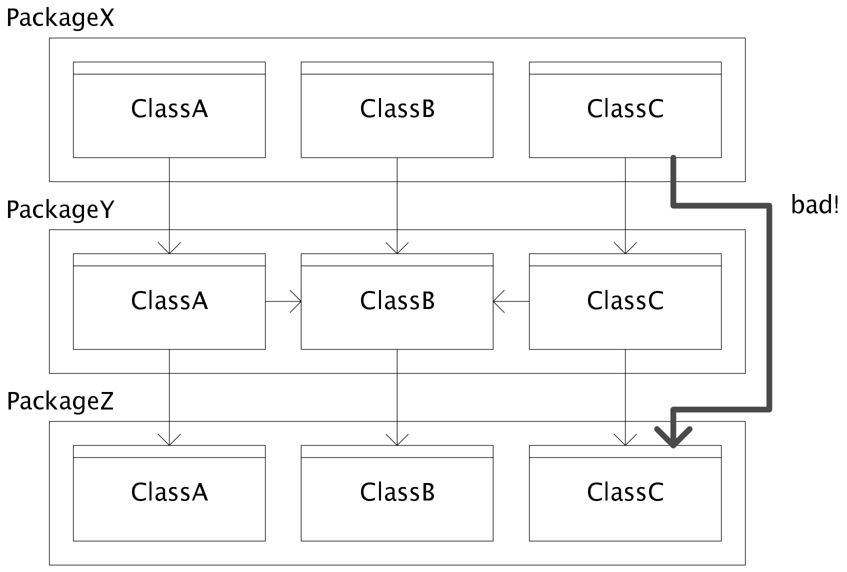
5.4 Dependences & Architecture

What makes a good package organization? A good test to check if your package structure is reasonable is to see what names each class uses. Draw an arrow from a class A to a class B if class A mentions class B in its code. These arrows are called *dependences.* Now if most of the arrows are within packages, and don't cross package boundaries, the package organization is fulfilling a useful function. It's grouping together classes that work together. If you need to change some code, most of your work may be confinable to a single package.

Here's a module dependency diagram to illustrate this kind of organization:



This isn't the only criterion for a reasonable division. You may organize your program into layers, and then you might expect many arrows between higher-level layers and the layers below them. But in this case, you wouldn't expect arrows to cross more than one layer.



You can see that the package structure alone doesn't tell you much about how the software is put together. The dependences can show you much more. As we study them more, we'll see how dependences can be used as a guide for good design, and can expose lurking problems.

5.5 Access Control

In Java, you can limit the scope in which components can be accessed. By marking components with access control keywords, you can make them more or less accessible. This is useful because it allows you to prevent certain dependences from arising when you're writing a component. For example, if you don't declare the Dictionary class of editor.edit.spell to be public, it will not be accessible outside the package. A Java compiler would reject this code:

package editor.edit.search;
import editor.edit.spell;
class Finder {... Dictionary...}

because the non-public *Dictionary* class cannot be accessed from the package *editor.edit.search*.

There are three keywords: *public*, *private* and *protected*.

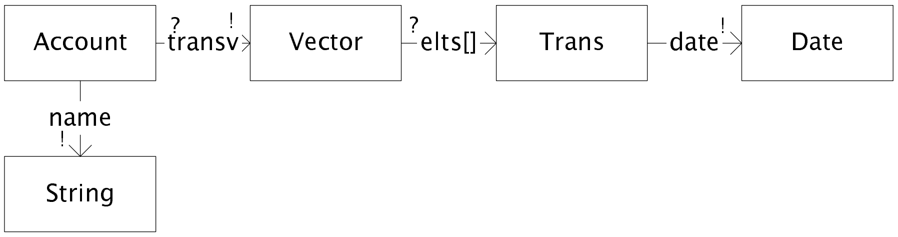
Here are their rules, taken from the Java language spec:

- If a class or interface is declared to be public, it can be accessed whenever its package can be accessed, otherwise it can only be accessed within its own package.
- A member (field or method) or a constructor of a class or interface can only be accessed if the class or interface can be accessed, and the member of constructor permits access.
- If a member or constructor is declared public, then access is permitted. All members of interfaces are implicitly public.
- If a member or constructor is declared private, it can only be accessed within its class.
- If a member or constructor is declared without an access keyword, it can only be accessed within its package.
- If a member or constructor is declared protected, it can only be accessed either within its package, or in a subclass of the class in which it is declared.

Notice that the *protected* keyword has the odd property that adding it to an unannotated member or constructor makes it *more* accessible. Without the keyword, the default access says that you can't access outside the package. When you add the keyword, you can access it in a subclass that's outside the package. The rules are a bit trickier than this suggests: see the JLS (section 6.6.2) for details.

5.6 Enforcing Object Model Properties with Access Control

Let's return to our bank example of Lecture 3 and see how we can make use of access control to ensure that our object model properties hold. Recall that we wanted each Account object to have a non-null reference to its transaction vector through the *transv* field, so that we don't get null pointer errors when we post a transaction. This property was indicated in the object model by the multiplicity marking (!) on the target end of the *transv* field arrow:



We observed that this invariant can be broken simply by setting the vector to be null:

Account a = new Account ();
a.transv = null;

Now we can prevent this. By declaring the field to be private, we can ensure that it is not accessed outside the class. In general, it's good practice to make all fields private:

```
import java.util.*;
package bank.accounts;
public class Account {
    private String name;
    private Vector transv;
    private int balance;
    Account (String n) {
        transv = new Vector ();
        balance = 0;
        name = n;
        }
    boolean checkTrans (Trans t) {
        return (balance + t.amount >= 0);
        }
    void post (Trans t) {
        transv.addElement (t);
        balance += t.amount;
        }
    }
```

The access control makes *local reasoning* possible. Since the field *transv* is assigned to only once (in the constructor), and is always given a non-null value, we can be sure that the invariant is maintained. Similarly, we can see that the *transv* vector only contains *Trans* objects, by examining the *post* method.

We can also check the more subtle property, that transaction vectors are not shared between accounts. This relies on noticing that the assignment to *transv* is always a new vector.

Access control won't solve all our problems though. We can't establish the object model property that *Trans* objects aren't shared between (the transaction vectors of) *Account* objects, for example, since a *Trans* object is passed into the *post* method and we can't check what other object is referring to an object when it's passed in.

The problem of a possibly null date in the *Trans* object is an interesting variant. Access control alone won't work, since the date is passed in to the constructor. However, we can test the date argument and throw an exception if it's null:

```
public class Trans {
    private int amount;
    private Date date;
    Trans (int a, Date d) {
        if (d == null) throw new NullPointerException ();
        amount = a; date = d;
        }
    }
```

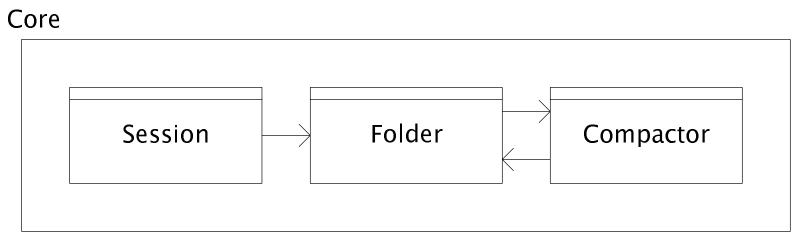
5.7 Example: Instrumenting a Program

For the remainder of the lecture, we'll study some decoupling mechanisms in the context of an example that's tiny but representative of an important class of problems.

When we discussed runtime assertions last lecture, we noted that the failure of an assertion might be reported to the user or logged by some I/O operation. What should the relationship be between the code that checks the assertion and the code that calls the low-level I/O operation? Surprisingly, this is much more subtle and interesting than you might imagine.

Let's consider a slightly more general variant of this problem in which we want to report incremental steps of a program as it executes by displaying progress line by line. For example, in a compiler with several phases, we might want to display a message when each phase starts and ends. In an email client, we might display each step involved in downloading email from a server. This kind of reporting facility is useful when the individual steps might take a long time or are prone to failure (so that the user might choose to cancel the command that brought them about). Progress bars are often used in this context, but they introduce further complications (marking the start and end of an activity, and calculating proportional progress) which we won't worry about.

As a concrete example, consider an email client that has a package *core* that contains a class *Session* that has code for setting up a communication session with a server and downloading messages, a class *Folder* for the objects that models folders and their contents, and a class *Compactor* that contains the code for compacting the representation of folders on disk. Assume there are calls from *Session* to *Folder* and from *Folder* to *Compactor*, but that the resource intensive activities that we want to instrument occur only in *Session* and *Compactor*, and not in *Folder*.



The module dependency diagram shows that *Session* depends on *Folder*, which has a mutual dependence on *Compactor*.

We'll look at a variety of ways to implement our instrumentation facility, and we'll study the advantages and disadvantages of each. Starting with the simplest, most naive design possible, we might intersperse statements such as

System.out.println ("Starting download");

throughout the program.

5.8 Abstraction by Parameterization

The problem with this scheme is obvious. When we run the program in batch mode, we might redirect standard out to a file. Then we realize it would be helpful to timestamp all the messages so we can see later, when reading the file, how long the various steps took. We'd like our statement to be

```
System.out.println ("Starting download at: " + new Date ());
```

instead. This should be easy, but it's not. We have to find all these statements in our code (and distinguish from other calls to System.out.println that are for different purposes), and alter each separately.

Of course, what we should have done is to define a procedure to encapsulate this functionality. In Java, this would be a static method:

```
public class StandardOutReporter {
    public static void report (String msg) {
        System.out.println (msg);
        }
    }
```

Now the change can be made at a single point in the code. We just modify the procedure:

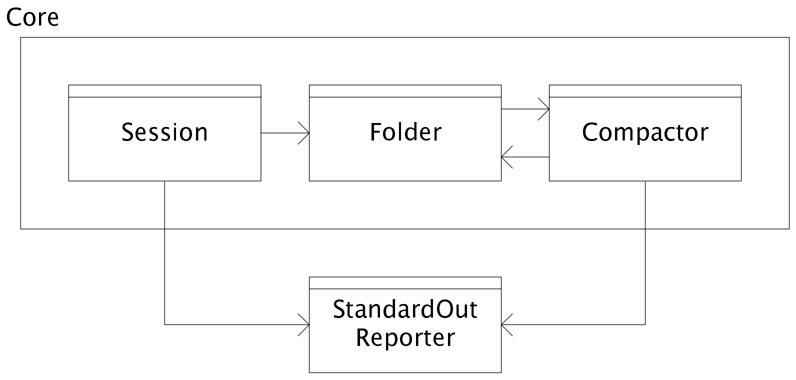
```
public class StandardOutReporter {
    public static void report (String msg) {
        System.out.println (msg + " at: " + new Date ());
        }
    }
```

Matthias Felleisen calls this the 'single point of control' principle. The mechanism in this case is one you're familiar with: what 6001 called *abstraction by parameterization*, because each call to the procedure

```
StandardOutReporter.report ("Starting download");
```

is an instantiation of the generic description, with the parameter *msg* bound to a particular value.

We can illustrate the single point of control in a module dependence diagram. We've introduced a single class on which the classes that use the instrumentation facility depend:all the client classes now depend:



Note that there is no dependence from *Folder* to *StandardOutReporter*, since the code of *Folder* makes no calls to it.

5.9 Decoupling with Interfaces

This scheme is far from perfect though. Factoring out the functionality into a single class was a good idea, but the code still has a dependence on the notion of writing to standard out. If we wanted to create a new version of our system with a graphical user interface, we'd need to replace this class with one containing the appropriate GUI code. That would mean changing all the references in the core package to refer to a different class, or changing the code of the class itself, and now having to handle two incompatible versions of the class with the same name. Neither of these is an attractive option.

In fact, the problem's even worse than that. In a program that uses a GUI, one writes to the GUI by calling a method on an object that represents part of the GUI: a text pane, or a message field. In swing, Java's user interface toolkit, the subclasses of *JTextComponent* have a *setText* method. Given some component named by the variable *outputArea*, for example, the display statement might be:

```
outputArea.setText (msg)
```

How are we going to pass the reference to the component down to the call site? And how are we going to do it without now introducing swing-specific code into the reporter class?

Java interfaces provide a solution. We create an interface with a single method *report* that will be called to display results.

```
public interface Reporter {
    void report (String msg);
    }
```

Now we add to each method in our system an argument of this type. The *Session* class, for example, may have a method *download*:

```
void download (Reporter r, ...) {
    r.report ("Starting downloading" );
    ...
    }
```

Now we define a class that will actually implement the reporting behaviour. Let's use standard out as our example as it's simpler:

```
public class StandardOutReporter implements Reporter {
    public void report (String msg) {
        System.out.println (msg + " at: " + new Date ());
        }
    }
```

This class is not the same as the previous one with this name. The method is no longer static, so we can create an object of the class and call the method on it. Also, we've indicated that this class is an implementation of the *Reporter* interface. Of course, for standard out this looks pretty lame and the creation of the object seems to be gratuitious. But for the GUI case, we'll do something more elaborate and create an object that's bound to the particular widget:

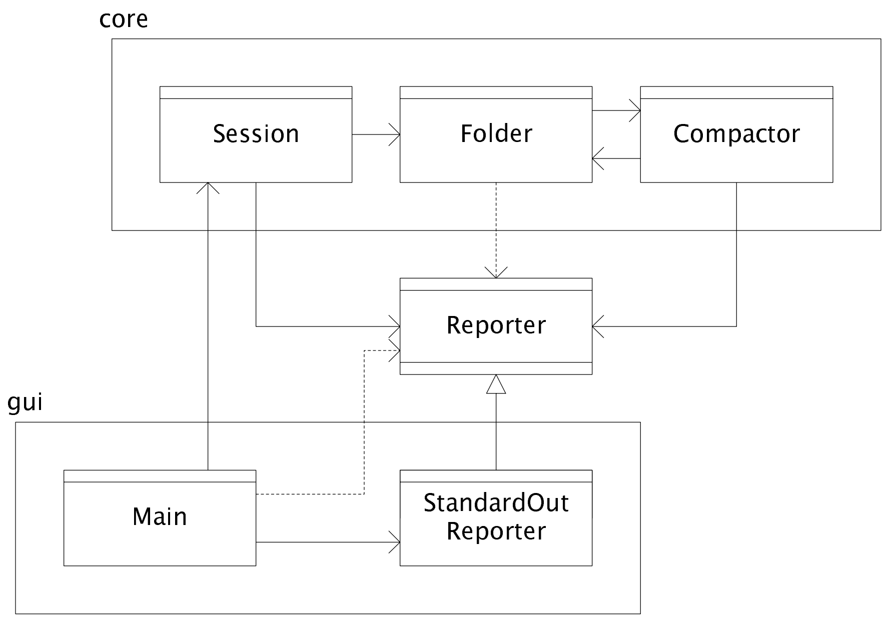
```
public class JTextComponentReporter implements Reporter {
    JTextComponent comp;
    public JTextComponentReporter (JTextComponent c) {comp = c;}
    public void report (String msg) {
        comp.setText (msg + " at: " + new Date ());
        }
    }
```

At the top of the program, we'll create an object and pass it in:

```
s.download (new StandardOutReporter (), ...);
```

Now we've achieved something interesting. The call to report now executes, at runtime, code that involves *System.out*. But methods like download only depend on the interface *Reporter*, which makes no mention of any specific output mechanism. We've successfully decoupled the output mechanism from the program, breaking the dependence of the core of the program on its I/O.

The module dependency diagram looks like this::



An arrow with a closed head from A to B is read 'A satisifies B'. B might be a class or an interface; the relationship in Java may be *implements* or *extends*. Here, the class *StandardOutReporter* satisfies the interface *Reporter*.

The key property of this scheme is that there is no longer a dependence of any class of the *core* package on a class in the *gui* package. All the dependences point downwards (at least logically!) from *gui* to *core*. To change the output from standard output to a GUI widget, we would simply replace the class StandardOutReporter by the class *JTextComponentReporter*, and modify the code in the main class of the *gui* package to call its constructor.

on the classes that actually contain concrete I/O code. This idiom is perhaps the most popular use of interfaces, and is well worth mastering.

The dotted arrows are weak dependences. A weak dependence from A to B means that A references the name of B, but not the name of any of its members. In other words, A knows that the class or interface B exists, and refers to variables of that type, but calls no methods of B, and accesses no fields of B.

The weak dependence of *Main* on *Reporter* simply indicates that the *Main* class may include code that handles a generic reporter; it's not a problem. The weak dependence of *Folder* on *Reporter* is a problem though. It's there because the *Reporter* object has to be passed via methods of *Folder* to methods of *Compactor.* Every method in the call chain that reaches a method that is instrumented must take a *Reporter* as an argument. This is a nuisance, and makes retrofitting this scheme painful.

5.10 Interfaces vs. Abstract Classes

You may wonder whether we might have used a class instead of an interface. An *abstract class* is one that is not completely implemented; it cannot be instantiated, but must be extended by a subclass that completes it.

Abstract classes are useful when you want to factor out some common code from several classes. Suppose we wanted to display a message saying how long each step had taken. We might implement a *Reporter* class whose objects retain in their state the time of the last call to *report*, and then take the difference between this and the current time for the output. By making this class an abstract class, we could reuse the code in each of the concrete subclasses *StandardOutReporter, JTextComponentReporter,* etc.

Why not pass make the argument of download have this abstract class as its type, instead of an interface? There are two related reasons. The first is that we want the dependence on the reporter code to be as weak as possible. The interface has no code at all; it expresses the minimal specification of what's needed. The second is that there in no multiple inheritance in Java: a class can only extend at most one other class. So when you're designing the core program, you don't want to use the opportunity to subclass prematurely. A class can implement any number of interfaces, so by choosing an interface, you leave it open to the designer of the reporter classes how they will be implemented.

5.11 Static Fields

The clear disadvantage of the scheme just discussed is that the reporter object has to be threaded through the entire core program. If all the output is displayed in a single text component, it seems annoying to have to pass a reference to it around. In dependency terms, every method has at least a weak dependence on the interface *Reporter.*

Global variables, or in Java *static fields*, provide a solution to this problem. To eliminate many of these dependences, we can hold the reporter object as a static field of a class:

```
public class StaticReporter {
    static Reporter r;
    static void setReporter (Reporter r) {
        this.r = r;
        {
    static void report (String msg) {
        r.report (msg);
        }
    }
```

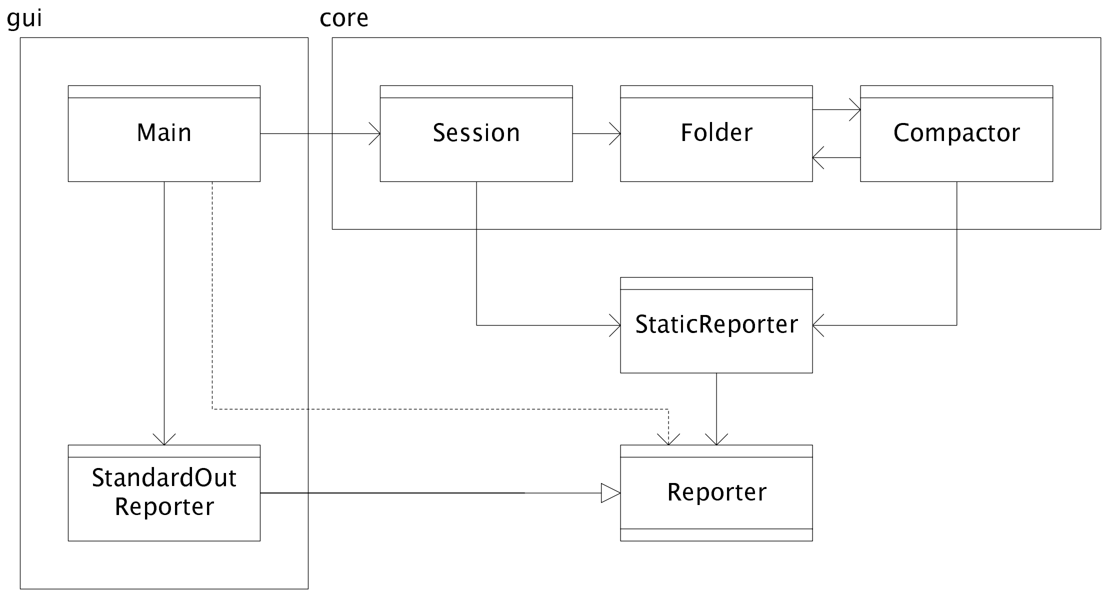
Now all we have to do is set up the static reporter at the start:

```
StaticReporter.setReporter (new StandardOutReporter ());
```

and we can issue calls to it without needing a reference to an object:

```
void download (...) {
    StaticReporter.report ("Starting downloading" );
    ...
    }
```

In the module dependency diagram, the effect of this change is that now only the classes that actually use the reporter are dependent on it:



Notice how the weak dependence of Folder has gone. We've seen this global notion before, of course, in our second scheme whose *StandardOutReporter* had a static method. This scheme combines that static aspect with the decoupling provided by interfaces.

Global references are handy, because they allow you to change the behaviour of methods low down in the call hierarchy without making any changes to their callers. But global variables are dangerous. They can make the code fiendishly difficult to understand. To determine the effect of a call to *Stati-*

cReporter.report, for example, you need to know what the static field *r* is set to. There might be a call to *setReporter* anywhere in the code, and to see what effect it has, you'd have to trace executions to figure out when it's executed relative to the code of interest.

Another problem with global variables is that they only work well when there is really one object that has some persistent significance. Standard out is like this. But text components in a GUI are not. We might well want different parts of the program to report their progress to different panes in our GUI. With the scheme in which reporter objects are passed around, we can create different objects and pass them to different parts of the code. In the static version, we'll need to create different methods, and it starts to get ugly very quickly.

Concurrency also casts doubt on the idea of having a single object. Suppose we upgrade our email client to download messages from several servers concurrently. We wouldn't want the progress messages from all the downloading sessions to be interleaved in a single output.

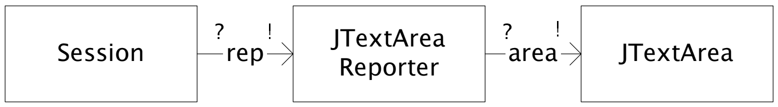
A good rule of thumb is to be wary of global variables. Ask yourself if you really can make do with a single object. Usually you'll find ample reason to have more than one object around.

This scheme goes by the term *Singleton* in the design patterns literature, because the class contains only a single object.

5.12 Dynamic Configuration

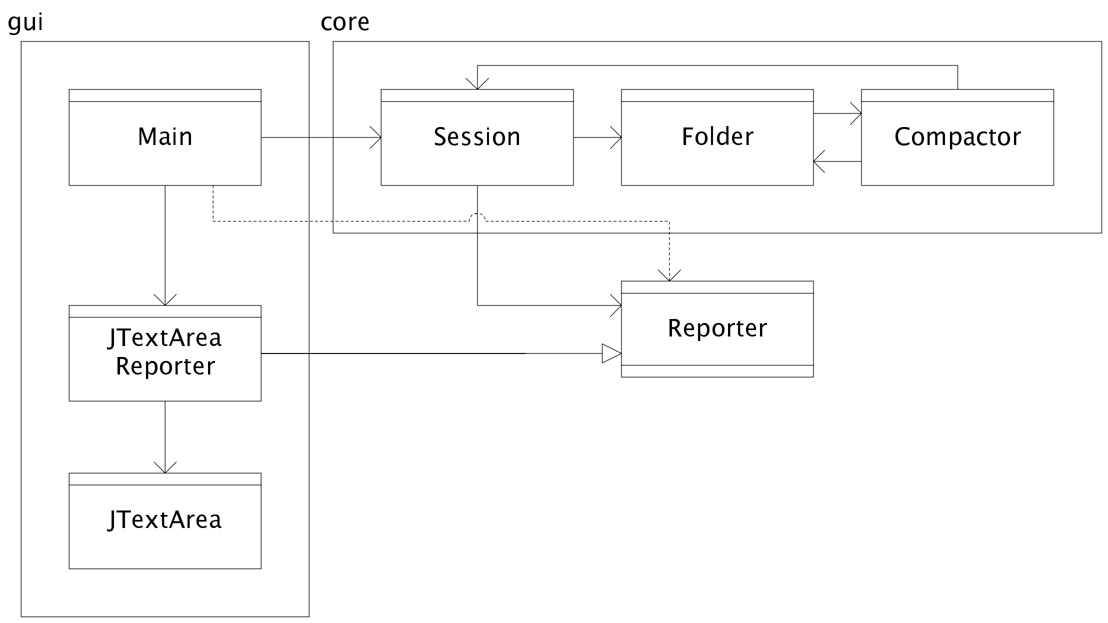
Our final scheme is a hybrid. We want the advantage of separate reporter objects, but we don't want to have to pass these objects in every method call. So we create several *Reporter* objects, and bind them to key objects in the core computation.

For example, we might bind *Reporter* objects to *Session* objects. Each *Session* object handles a single session in which email messages are downloaded. Its state might include details of the server, statistics, etc. Now we add to this object a field that refers to its reporter; the reporter is passed to the constructor, and then all the code that acts on a session object can access the reporter. Because the session object holds its own reporter, we can easily ensure that each session reports in a separate output panel. Suppose we use Swing's *JTextArea* objects. Then our object model might be:



whose multiplicities indicate that each session has its own *JTextAreaReporter* object, and each of these is connected to a separate *JTextArea* swing object.

Let’s look at this in the context of the module dependency diagram:



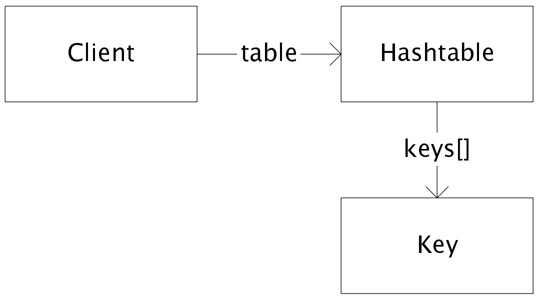
The difference is subtle but important. At runtime, each *Session* object is associated with a *JTextAreaReporter* object. But the code of *Session* shows no dependence on *JTextAreaReporter.* The association is created at runtime: the system is dynamically configured.

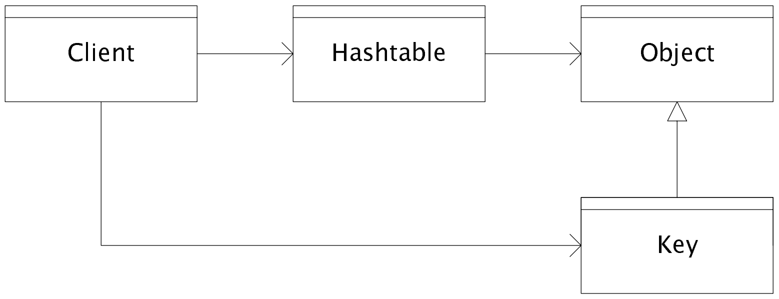
Note that I’ve added a dependence from *Compactor* to *Session*. *Compactor* now reports its progress by calling a method of *Session*, which then makes use of its own *Reporter* object. This is a common feature of this style of design: the objects become highly interdependent, forming collaborations rather than more traditional layering relationships in which mutual dependences are rare.

Dynamic configuration is very powerful. It adds some complexity in order to provide syntactic decoupling. It is widely used in design patterns (eg, *Observer*).

This example also illustrates why we need both the object model and the dependence diagram. The object model is about runtime configurations. It tells us about the structure of the state. The dependence diagram is about syntactic relationships at compile time. It tells us about the structure of the code. Often, the two diagrams will be very similar. But their divergences expose the subtleties of a design.

This same idiom arises with all container types in Java. The *Hashtable* class will only work correctly if the code of the class of the objects inserted as keys have *equals* and *hashCode* methods that behave appropriately. If not, the *get* method may fail to return a value for a given key even when there’s a matching key in the table. How do we explain this? Does the *Hashtable* class depend on your particular concrete key class? The object model and dependence diagram give the answer:

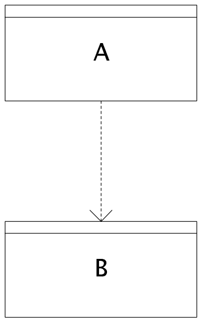




The table is asociated dynamically with the keys (and thus executes the code in the class *Key*). But Hashtable has only a dependence on *Object*. The obligation of the *Key* class is expressed by its relationship in the dependence diagram to *Object*, whose specification dictates the behaviour of *hashCode* and *equals*.

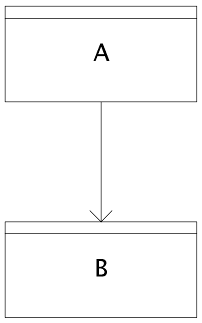
5.13 Module Dependence Diagram

The module dependence diagram has a syntax even simpler than the object model. Its elementary boxes are classes (horizontal stripe at the top) and interfaces (stripes at top and bottom). Packages are shown as larger boxes without stripes. There are two kinds of dependence arrow. A dotted line with an open arrow head denotes a *weak dependence*: an arrow from A to B means that A mentions B.

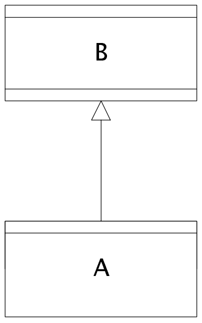


A solid line with an open arrow head denotes a *strong dependence*: an arrow from A to B means that

A mentions members of B (fields or methods).



An arrow with a closed head from A to B denotes a satisfaction relation; in Java, this is always due to *extends* or *implements.*



Our course text includes additional notation for static methods and iteration abstractions (*enumerations*, in Java terminology).

5.14 Summary

We have studied how various features of Java are used to support the organization of programs, and the decoupling of one part from another. We've seen that runtime and compile-time coupling are distinct: the control flow and object structure do not always follow syntactic dependences. The module dependency diagram allows us to express succinctly the dependence structure of a program. We'll use it throughout the course to show overall syntactic structure, to expose unwanted dependences, and to explain strategies for decoupling.

Lecture 6: Procedure Specifications

In this lecture, we'll look at the role played by specifications of methods. Specifications are the linchpin of team work. It's impossible to delegate responsibility for implementing a method without a specification. The specification acts as a contract: the implementor is responsible for meeting the contract, and a client that uses the method can rely on the contract. In fact, we'll see that like real legal contracts, specifications place demands on both parties: when the specification has a precondition, the client has responsibilities too.

Many of the nastiest bugs in programs arise because of misunderstandings about behaviour at interfaces. Although every programmer has specifications in mind, not all programmers write them down. As a result, different programmers on a team have different specifications in mind. When the program fails, it's hard to determine where the error is. Precise specifications in the code let you apportion blame (to code fragments, not people!), and can spare you the agony of puzzling over where a fix should go.

Specifications are good for the client of a method because they spare her the task of reading code. If you're not convinced that reading a spec is easier than reading code, take a look at some of the standard Java specs and compare them to the source code that implements them. Vector, for example, in the package java.util, has a very simple spec but its code is not at all simple.

Specifications are good for the implementor of a method because they give her freedom to change the implementation without telling clients. Specifications can make code faster too. Sometimes a weak specification makes it possible to do a much more efficient implementation. In particular, a precondition may rule out certain states in which a method might have been invoked that would have incurred an expensive check that is no longer necessary.

This lecture is related to our discussion of dependences in the last lecture. There, we were concerned only with whether a dependence existed. Here, we are investigating the question of what form the dependence should take. By exposing only the specification of a procedure, its clients are less dependent on it, and therefore less likely to need changing when the procedure changes.

6.1 Behavioural Equivalence

Consider these two methods. Are they the same or different?

```
static int findA (int [] a, int val) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == val) return i;
        }
    return a.length;
    }
static int findB (int [] a, int val) {
    for (int i = a.length -1 ; i > 0; i--) {
        if (a[i] == val) return i;
        }
    return -1;
    }
```

Of course the code is different, so in that sense they are different. Our question though is whether one could substitute one implementation for the other. Not only do these methods have different code; they actually have different behaviour:

· when *val* is missing, *findA* returns the length and *findB* returns -1;
· when *val* appears twice, *findA* returns the lower index and *findB* returns the higher.

But when *val* occurs at exactly one index of the array, the two methods behave the same. It may be that clients never rely on the behaviour in the other cases. So the notion of equivalence is in the eye of the beholder, that is, the client. In order to make it possible to substitute one implementation for another, and to know when this is acceptable, we need a specification that states exactly what the client depends on.

In this case, our specification might be

requires: *val occurs in a*
effects: *returns result such that a[result] = val*

6.2 Specification Structure

A specification of a method consists of several clauses:

· a precondition, indicated by the keyword *requires*;
· a postcondition, indicated by the keyword *effects*;
· a frame condition, indicated by the keyword *modifies*.

We'll explain each of these in turn. For each, we'll explain what the clause means, and what a missing clause implies. Later, we'll look at some convenient shorthands that allow particular common idioms to be specified as special kinds of clause.

The precondition is an obligation on the client (ie, the caller of the method). It's a condition over the state in which the method is invoked. If the precondition does not hold, the implementation of the method is free to do anything (including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc).

The postcondition is an obligation on the implementor of the method. If the precondition holds for the invoking state, the method is obliged to obey the postcondition, by returning appropriate values, throwing specified exceptions, modifying or not modifying objects, and so on.

The frame condition is related to the postcondition. It allows more succinct specifications. Without a frame condition, it would be necessary to describe how all the reachable objects may or may not change. But usually only some small part of the state is modifed. The frame condition identifies which objects may be modified. If we say *modifies x*, this means that the object *x*, which is presumed to be mutable, may be modified, but no other object may be. So in fact, the frame condition or *modifies clause* as it is sometimes called is really an assertion about the objects that are *not* mentioned. For the ones that are mentioned, a mutation is possible but not necessary; for the ones that are not mentioned, a mutation may not occur.

Omitted clauses have particular interpretations. If you omit the precondition, it is given the default value *true*. That means that every invoking state satisfies it, so there is no obligation on the caller. In this case, the method is said to be *total*. If the precondition is not true, the method is said to be *partial*, since it only works on some states.

If you omit the frame condition, the default is *modifies nothing*. In other words, the method makes no changes to any object.

Omitting the postcondition makes no sense and is never done.

6.3 Declarative Specification

Roughly speaking, there are two kinds of specifications. *Operational* specifications give a series of steps that the method performs; pseudocode descriptions are operational. *Declarative* specifications don't give details of intermediate steps. Instead, they just give properties of the final outcome, and how it's related to the initial state.

Almost always, declarative specifications are preferable. They're usually shorter, easier to understand, and most importantly, they don't expose implementation details inadvertently that a client may rely on (and then find no longer hold when the implementation is changed). For example, if we want to allow either implementation of *find*, we would not want to say in the spec that the method 'goes down the array until it finds val', since aside from being rather vague, this spec suggests that the search proceeds from lower to higher indices and that the lowest will be returned, which perhaps the specifier did not intend.

Here are some example of declarative specification. The class *StringBuffer* provides objects that are like *String* objects but mutable. The methods of StringBuffer modify the object rather than creating new ones: they are *mutators*, whereas String's methods are *producers*. The *reverse* method reverses a string. Here's how it's specified in the Java API:

```
public StringBuffer reverse()
// modifies: this
// effects: Let n be the length of the old character sequence, the one contained in the string buffer
//      just prior to execution of the reverse method. Then the character at index k in the new
//      character sequence is equal to the character at index n-k-1 in the old character sequence.
```

Note that the postcondition gives no hint of how the reversing is done; it simply gives a property that relates the character sequence before and after. (We've omitted part of the specification, by the way: the return value is simply the string buffer object itself.) A bit more formally, we might write

```
effects:
    length (this.seq) = length (this.seq')
    all k: 0..length(this.seq)-1 | this.seq'[k] = this.seq[length(this.seq)-k-1]
```

Here I've used the notation this.seq' to mean the value of the character sequence in this object after execution. The course text uses the keyword *post* as a subscript for the same purpose. There's no precondition, so the method must work when the string buffer is empty too; in this case, it will actually leave the buffer unchanged.

Another example, this time from *String*. The *startsWith* method tests whether a string starts with a particular substring.

```
public boolean startsWith(String prefix)
// Tests if this string starts with the specified prefix.
// effects:
//   if (prefix = null) throws NullPointerException
//   else returns true iff exists a sequence s such that (prefix.seq ^ s = this.seq)
```

I've assumed that String objects, like StringBuffer objects, have a specification field that models the sequence of characters. The caret is the concatenation operator, so the postcondition says that the method returns true if there is some suffix which when concatenated to the argument gives the character sequence of the string. The absence of a modifies clause indicates that no object is mutated. Since String is an immutable type, none of its methods will have modifies clauses.

Another example from String:

```
public String substring(int i)
// effects:
//   if i < 0 or i > length (this.seq) throws IndexOutOfBoundsException
//   else returns r such that
//        some sequence s | length(s) = i && s ^ r.seq = this.seq
```

This specification shows how a rather mathematical postcondition can sometimes be easier to understand than an informal description. Rather than talking about whether *i* is the starting index, whether it comes just before the substring returned, etc, we simply decompose the string into a prefix of length *i* and the returned string.

Our final example shows how a declarative specification can express what is often called non-determinism, but is better called 'under-determinedness'. By not giving enough details to allow the client to infer the behaviour in all cases, the specification makes implementation easier. The term non-determinism suggests that the implementation should exhibit all possible behaviours that satisfy the specification, which is not the case.

There is a class BigInteger in the package java.math whose objects are integers of unlimited size. The class has a method similar to this:

```
public boolean maybePrime ()
// effects: if this BigInteger is composite, returns false
```

If this method returns false, the client knows the integer is not prime. But if it returns true, the integer may be prime or composite. So long as the method returns false a reasonable proportion of the time, it's useful. In fact, as the Java API states: the method takes an argument that is a measure of the uncertainty that the caller is willing to tolerate. The execution time of this method is proportional to the value of this parameter.' We won't worry about probabilistic issues in this course; we mention this spec simply to note that it does not determine the outcome, and is still useful to clients.

Here is an example of a truly underdetermined specification. In the *Observer* pattern, a set of obejects known as 'observers' are informed of changes to an object known as a 'subject'. The subject will belong to a class that subclasses *java.util.Observable*. In the specification of *Observable*, there is a specification field *observers* that holds the set of observer objects. This class provides methods to add an observer

```
public void addObserver(Observer o)
// modifies: this
// effects: this.observers' = this.observers + {o}
```

(using + to mean set union), and to notify the observers of a change in state:

```
public void notifyObservers()
// modifies the objects in this.observers
// effects: calls o.notify on each observer o in this.observers
```

The specification of notify does not indicate in what order the observers are notified. What order is chosen may have an effect on overall program behaviour, but having chosen to model the observers as a set, there is no way to specify an order anyway.

6.4 Exceptions and Preconditions

An obvious design issue is whether to use a precondition, and if so, whether it should be checked. It is crucial to understand that a precondition does not require that checking be performed. On the contrary, the most common use of preconditions is to demand a property precisely because it would be hard or expensive to check.

As mentioned above, a non-trivial precondition renders the method partial. This inconveniences clients, because they have to ensure that they don't call the method in a bad state (that violates the precondition); if they do, there is no predictable way to recover from the error. So users of methods don't like preconditions, and for this reason the methods of a library will usually be total. That's why the Java API classes, for example, invariably throw exceptions when arguments are inappropriate. It makes the programs in which they are used more robust.

Sometimes though, a precondition allows you to write more efficient code and saves trouble. For example, in an implementation of a binary tree, you might have a private method that balances the tree. Should it handle the case in which the ordering invariant of the tree does not hold? Obviously not, since that would be expensive to check. Inside the class that implements the tree, it's reasonable to assume that the invariant holds. We'll generalize this notion when we talk about *representation invariants* next week.

The decision of whether to use a precondition is an engineering judgment. The key factors are the cost of the check (in writing and executing code), and the scope of the method. If it's only called locally in a class, the precondition can be discharged by carefully checking all the sites that call the method. But if the method is public, and used by other developers, it woul d be less wise to use a precondition.

Sometimes, it's not feasible to check a condition without making a method unacceptably slow, and a precondition is often necessary in this case. In the Java standard library, for example, the binary search methods of the *Arrays* class require that the array given be sorted. To check that the array is sorted would defeat the entire purpose of the binary search: to obtain a result in logarithmic and not linear time.

Even if you decide to use a precondition, it may be possible to insert useful checks that will detect, at least sometimes, that the precondition was violated. These are the runtime assertions that we discussed in our lecture on exceptions. Often you won't check the precondition explicitly at the start, but you'll discover the error during computation. For example, in balancing the binary tree, you might check when you visit a node that its children are appropriately ordered.

If a precondition is found to be violated, you should throw an *unchecked* exception, since the client will not be expected to handle it. The throwing of the exception will not be mentioned in the specification, although it can appear in implementation notes below it.

6.5 Shorthands

There are some convenient shorthands that make it easier to write specifications. When a method does not modify anything, we specify the return value in a *returns* clause. If an exception is thrown, the condition and the exception are given in a *throws* clause. For example, instead of

```
public boolean startsWith(String prefix)
// effects:
//   if (prefix = null) throws NullPointerException
//   else returns true iff exists a sequence s such that (prefix.seq ^ s = this.seq)
```

we can write

```
public boolean startsWith(String prefix)
// throws: NullPointerException if (prefix = null)
// returns: true iff exists a sequence s such that (prefix.seq ^ s = this.seq)
```

The use of these shorthands implies that no modifications occur. There is an implicit ordering in which conditions are evaluated: any throws clauses are considered in the order in which they appear, and then returns clauses. This allows us to omit the else part of the if-then-else statement.

Our 6170 JavaDoc html generator produces specifications formatted in the Java API style. It allows the clauses that we have discussed here, and which have been standard in the specification community for several decades, in addition to the shorthand clauses. We won't use the JavaDoc *parameters* clause: it is subsumed by the postcondition, and is often cumbersome to write.

6.6 Specification Ordering

Suppose you want to substitute one method for another. How do you compare the specifications?

A specification A is at least as strong as a specification B if
- A's precondition is no stronger than B's
- A's postcondition is no weaker than B's, for the states that satisfy B's precondition.

These two rules embody several ideas. They tell you that you can always weaken the precondition;

placing fewer demands on a client will never upset him. You can always strengthen the postcondition, which means making more promises. For example, our method *maybePrime* can be replaced in any context by a method *isPrime* that returns true if and only if the integer is prime. And where the precondition is false, you can do whatever you like. If the postcondition happens to specify the outcome for a state that violates the precondition, you can ignore it, since that outcome is not guaranteed anyway.

These relationships between specifications will be important when we look at the conditions under which subclassing works correctly (in our lecture on subtyping and subclassing).

6.7 Judging Specifications

What makes a good method? Designing a method means primarily writing a specification. There are no infallible rules, but there are some useful guidelines:

- The specification should be *coherent*: it shouldn't have lots of different cases. Deeply nested if-statements are a sign of trouble, as are boolean flags presented as arguments.
- The results of a call should be *informative*. Java's HashMap class has a put method that takes a key and a value and returns a previous value if that key was already mapped, or null otherwise. HashMaps allow null references to be stored, so a null result is hard to interpret.
- The specification should be *strong enough*. There's no point throwing a checked exception for a bad argument but allowing arbitrary mutations, because a client won't be able to determine what mutations have actually been made.
- The specification should *be weak enough*. A method that takes a URL and returns a network connection clearly cannot promise always to succeed.

6.8 Summary

A specification acts as a crucial firewall between the implementor of a procedure and its client. It makes separate development possible: the client is free to write code that uses the procedure without seeing its source code, and the implementor is free to write the code that implements the procedure without knowing how it will be used. Declarative specifications are the most useful in practice. Preconditions make life hard for the client, but, applied judiciously, are a vital tool in the software designer's repertoire.

Lecture 7: Abstract Types

In this lecture, we look at a particular kind of dependence, that of a client of an abstract type on the type's representation, and see how it can be avoided. We also discuss briefly the notion of specification fields for specifying abstract types, the classification of operations, and the tradeoff of representations.

7.1 User-Defined Types

In the early days of computing, a programming language came with built-in types (such as integers, booleans, strings, etc.) and built-in procedures, eg. for input and output. Users could define their own procedures: that's how large programs were built.

A major advance in software development was the idea of abstract types: that one could design a programming language to allow user-defined types too. This idea came out of the work of many researchers, notably Dahl (the inventor of the Simula language), Hoare (who developed many of the techniques we now use to reason about abstract types), Parnas (who coined the term 'information hiding' and first articulated the idea of organizing program modules around the secrets they encapsulated), and here at MIT, Barbara Liskov and John Guttag, who did seminal work in the specification of abstract types, and in programming language support for them (and developed 6170!).

The key idea of *data abstraction* is that a type is characterized by the operations you can perform on it. A number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on. In a sense, users could already define their own types in early programming languages: you could create a record type *date*, for example, with integer fields for day, month and year. But what made abstract types new and different was the focus on operations: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers. All that matters is the operations.

In Java, as in many modern programming languages, the separation between built-in types and user-defined types is a bit blurry. The classes in java.lang, such as Integer and Boolean are built-in; whether you regard all the collections of java.util as built-in is less clear (and not very important anyway). Java complicates the issue by having primitive types that are not objects. The set of these types, such as int and boolean, cannot be extended by the user.

7.2 Classifying Types and Operations

Types, whether built-in or user-defined, can be classified as mutable or immutable. The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results. So *Vector* is mutable, because you can call *addElement* and observe the change with the *size* operation. But *String* is immutable, because its operations create new string objects rather than changing existing ones. Sometimes a type will be provided in two forms, a mutable and an immutable form. *StringBuffer*, for example, is a mutable version of *String* (although the two are certainly not the same Java type, and are not interchangeable).

As we discussed in lecture 2, immutable types are generally easier to reason about. Aliasing is not an issue, since sharing cannot be observed. And sometimes using immutable types is more efficient, because more sharing is possible. But many problems are more naturally expressed using mutable types, and when local changes are needed to large structures, they tend to be more efficient.

The operations of an abstract type are classified as follows:

· *Constructors* create new objects of the type. A constructor may take an object as an argument, but not an object of the type being constructed.
· *Producers* create new objects from old objects; the terms are synonymous. The *concat* method of String, for example, is a producer: it takes two strings and produces a new one representing their concatenation.
· *Mutators* change objects. The *addElement* method of *Vector*, for example, mutates a vector by adding an element to its high end.
· *Observers* take objects of the abstract type and return objects of a different type. The *size* method of *Vector*, for example, returns an integer.

We can summarize these distinctions schematically like this:

constructor: t -> T
producer: T, t -> T
mutator: T, t -> void
observer: T, t -> t

These show informally the shape of the signatures of operations in the various classes. Each *T* is the abstract type itself; each *t* is some other type. In general, when a type is shown on the left, it can occur more than once. For example, a producer may take two values of the abstract type; string *concat* takes two strings. The occurrences of *t* on the left may also be omitted; some observers take no non-abstract arguments (eg, *size*), and some take several.

This classification gives some useful terminology, but it's not perfect. In complex data types, there may be operations that are producers and mutators, for example. Some people use the term 'producer' to imply that no mutation occurs.

Another term you should know is *iterator.* An iterator usually means a special kind of method (not available in Java) that returns a collection of objects one at a time -- the elements of a set, for example. In Java, an iterator is a *class* that provides methods that can then be used to obtain a collection of objects one at a time. Most collection classes provide a method with the name *iterator* that returns an iterator.

7.3 Example: List

Let's look at an example of an abstract type: the *list*. A list, in Java, is like an array. It provides methods to extract the element at a particular index, and to replace the element at a particular index. But unlike an array, it also has methods to insert or remove an element at a particular index. In Java, *List* is an interface with many methods, but for now, let's imagine it's a simple class with the following methods:

```
public class List {
    public List ();
    public void add (int i, Object e);
    public void set (int i, Object e);
    public void remove (int i);
    public int size ();
    public Object get (int i);
    }
```

The *add*, *set* and *remove* methods are mutators; the *size* and *get* methods are observers. It's common for a mutable type to have no producers (and an immutable type certainly cannot have mutators).

To specify these methods, we'll need some way to talk about what a list looks like. We do this with the notion of *specification fields*. You can think of an object of the type as if it had these fields, but remember that they don't actually need to be fields in the implementation, and there is no requirement that a specification field's value be obtainable by some method. In this case, we'll describe lists with a single specification field,

```
seq [Object]    elems;
```

where for a list *l*, the expression *l.elems* will denote the sequence of objects stored in the list, indexed from zero. Now we can specify some methods:

```
public void get (int i);
// throws
//     IndexOutOfBoundsException if i < 0 or i > length (this.elems)
// returns
//     this.elems [i]
public void add (int i, Object e);
// modifies this
// effects
//     throws IndexOutOfBoundsException if i < 0 or i > length (this.elems)
//     else this.elems' = this.elems [0..i-1] ^ <e> ^ this.elems [i..]
public void set (int i, Object e);
// modifies this
// effects
//     throws IndexOutOfBoundsException if i < 0 or i >= length (this.elems)
//     else this.elems' [i] = e and this.elems unchanged elsewhere
```

In the postcondition of *add*, I've used *s[i..j]* to mean the subsequence of *s* from indices *i* to *j*, and *s[i..]* to mean the suffix from i onwards. The caret means sequence concatenation. So the postcondition says that, when the index is in bounds or one above, the new element is 'spliced in' at the given index.

7.4 Designing an Abstract Type

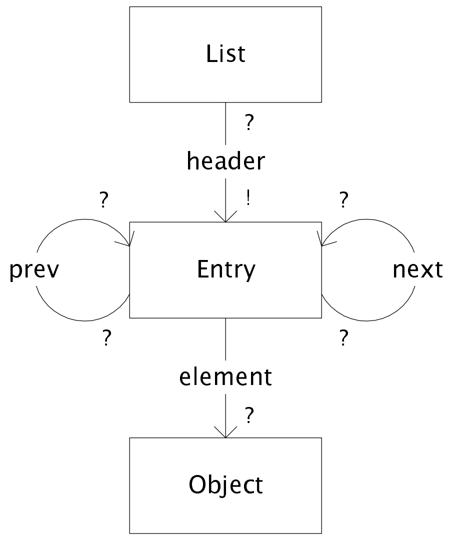
Designing an abstract type involves choosing good operations and determining how they should behave. A few rules of thumb:

- It's better to have a few, simple operations that can be combined in powerful ways than lots of complex operations.
- Each operation should have a well-defined purpose, and should have a coherent behaviour rather than a panoply of special cases.
- The set of operations should be *adequate*; there must be enough to do the kinds of computations clients are likely to want to do. A good test is to check that every property of an object of the type can be extracted. For example, if there were no *get* operation, we would not be able to find out what the elements of the list are. Basic information should not be inordinately difficult to obtain. The *size* method is not strictly necessary, because we could apply get on increasing indices, but this is inefficient and inconvenient.
- The type may be generic: a list or a set, or a graph, for example. Or it may be domain-specific: a street map, an employee database, a phone book, etc. But it should not mix generic and domain-specific features.

7.5 Choice of Representations

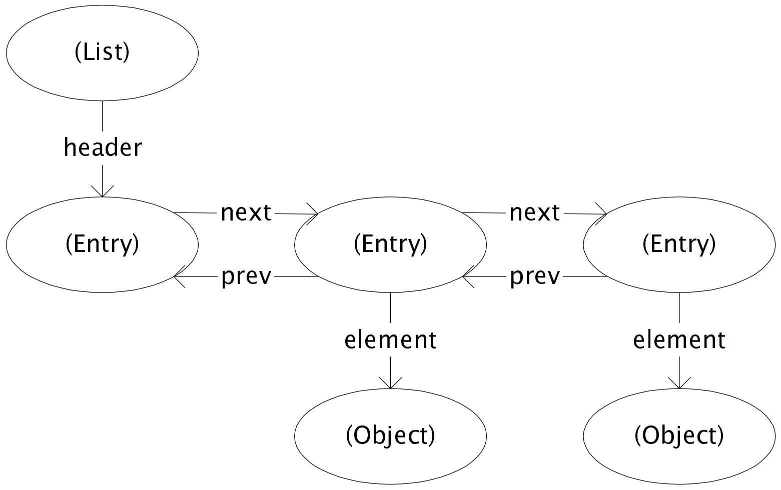
So far, we have focused on the characterization of abstract types by their operations. In the code, a class that implements an abstract type provides a *representation*: the actual data structure that supports the operations. The representation will be a collection of fields each of which has some other Java type; in a recursive implementation, a field may have the abstract type but this is rarely done in Java.

Linked lists are a common representation of lists, for example. The following object model shows a linked list implementation similar (but not identical to) the *LinkedList* class in the standard Java library:

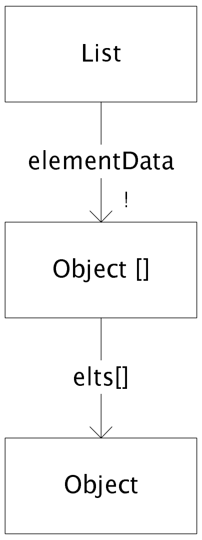


The list object has a field *header* that references an *Entry* object. An *Entry* object is a record with three fields: *next* and *prev* which may hold references to other *Entry* objects (or be null), and *element*, which holds a reference to an element object. The *next* and *prev* fields are links that point forwards and backwards along the list. In the middle of the list, following *next* and then *prev* will bring you back to the object you started with. Let's assume that the linked list does not store null references as elements. There is always a dummy *Entry* at the beginning of the list whose element field is null, but this is not interpreted as an element.

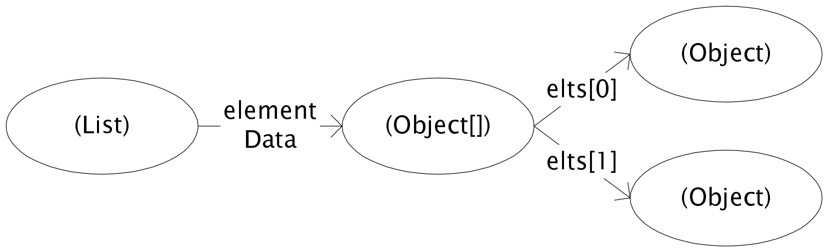
The following object diagram shows a list containing two elements::



Another, different representation of lists uses an array. The following object model shows how lists are represented in the class *ArrayList* in the standard Java library:



Here's a list with two elements in this representation:



These representations have different merits. The linked list representation will be more efficient when there are many insertions at the front of the list, since it can splice an element in and just change a couple of pointers. The array representation has to bubble all the elements above the inserted element to the top, and if the array is too small, it may need to allocate a fresh, larger array and copy all the references over. If there are many *get* and *set* operations, however, the array list representation is better, since it provides random access, in constant time, while the linked list has to perform a sequential search.

We may not know when we write code that uses lists which operations are going to predominate. The crucial question, then, is how we can ensure that it's easy to change representations later.

7.6 Representation Independence

Representation independence means ensuring that the use of an abstract type is independent of its representation, so that changes in representation have no effect on code outside the abstract type itself. Let's examine what goes wrong if there is no independence, and then look at some language mechanisms for helping ensure it.

Suppose we know that our list is implemented as an array of elements. We're trying to make use of some code that creates a sequence of objects, but unfortunately, it creates a Vector and not a List. Our List type doesn't offer a constructor that does the conversion. We discover that Vector has a method copyInto that copies the elements of the vector into an array. Here's what we now write:

```
List l = new List ();
v.copyInto (l.elementData);
```

What a clever hack! Like many hacks it works for a little while. Suppose the implementor of the List class now changes the representation from the array version to the linked list version. Now the list *l* won't have a field *elementData* at all, and the compiler will reject the program. This is a failure of representation independence: we'll have to change all the places in the code where we did this.

Having the compilation fail is not such a disaster. It's much worse if it succeeds and the change has still broken the program. Here's how this might happen.

In general, the size of the array will have to be greater than the number of elements in the list, since otherwise it would be necessary to create a fresh array every time an element is added or removed. So there must be some way of marking the end of the segment of the array containing the elements. Suppose the implementor of the list has designed it with the convention that the segment runs to the first null reference, or to the end of the array, whichever is first. Luckily (or actually unluckily), our hack works under these circumstances.

Now our implementor realizes that this was a bad decision, since determining the size of the list requires a linear search to find the first null reference. So he adds a *size* field and updates it when any operation is performed that changes the list. This is much better, because finding the size now takes constant time. It also naturally handles null references as list elements (and that's why it's what the Java *LinkedList* implementation does).

Now our clever hack is likely to produce some buggy behaviours whose cause is hard to track down. The list we created has a bad size field: it will hold zero however many elements there are in the list (since we updated the array alone). *Get* and *set* operations will appear to work, but the first call to *size* will fail mysteriously.

Here's another example. Suppose we have the linked list implementation, and we include an operation that returns the *Entry* object corresponding to a particular index.

```
public Entry getEntry (int i)
```

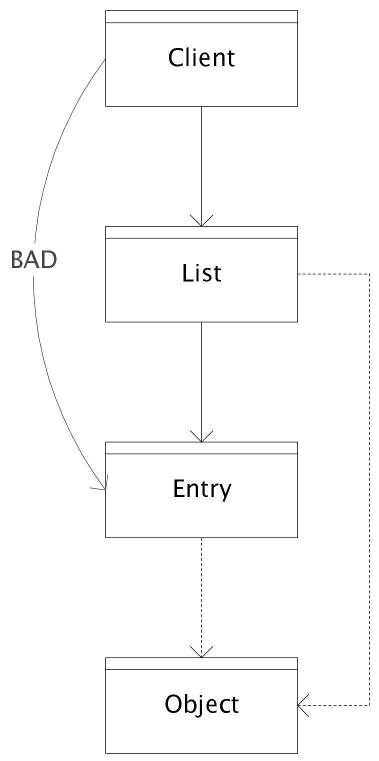
Our rationale is that if there are many calls to *set* on the same index, this will save the linear search of repeatedly obtaining the element. Instead of

```
l.set (i, x); ... ; l.set (i, y)
```

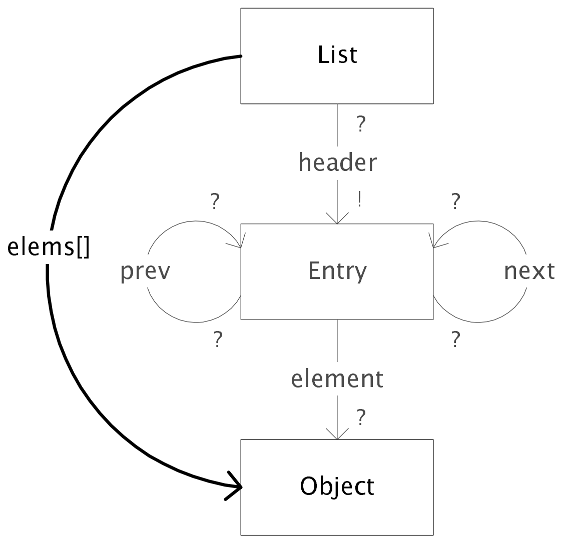
we can now write

```
Entry e = l.getEntry (i);
e.element = x;
...
e.element = y;
```

This also violates representation independence, because when we switch to the array representation, there will no longer be *Entry* objects. We can illustrate the problem with a module dependency diagram:



There should only be a dependence of the client type *Client* on the *List* class (and on the class of the element type, in this case *Object*, of course). The dependence of *Client* on *Entry* is the cause of our problems. Returning to our object model for this representation, we want to view the Entry class and its associations as internal to List. We can indicate this informally by colouring the parts that should be inaccessible to a client red (if you're reading a black and white printout, that's *Entry* and all its incoming and outgoing arcs), and by adding a specification field *elems* that hides the representation:



7.7 Language Mechanisms

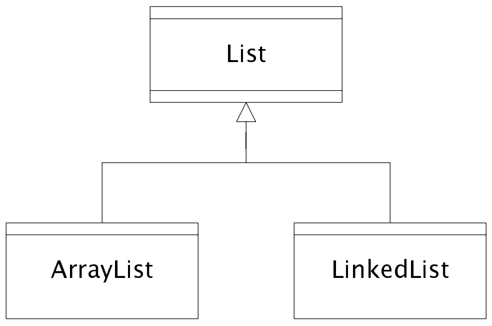
To prevent access to the representation, we can make the fields private. This eliminates the array hack; the statement

v.copyInto (l.elementData);

would be rejected by the compiler because the expression *l.elementData* would illegally reference a private field from outside its class.

The *Entry* problem is not so easily solved. There is no direct access to the representation. Instead, the List class returns an Entry object that belongs to the representation. This is called *representation exposure*, and it cannot be prevented by language mechanisms alone. We need to check that references to mutable components of the representation are not passed out to clients, and that the representation is not built from mutable objects that are passed in. In the array representation for example, we can't allow a constructor that takes an array and assigns it to the internal field.

Interfaces provide another method for achieving representation independence. In the Java standard library, the two representations of lists that we discussed are actually distinct classes, *ArrayList* and *LinkedList*. Both are declared to extend the *List* interface. As we saw in our last lecture, the interface breaks the dependence between the client and another class, in this case the representation class:



This approach is nice because an interface cannot have (non-static) fields, so the issue of accessing the representation never arises. But because interfaces in Java cannot have constructors, it can be awkward to use in practice, since information about the signatures of the constructors that are shared across implementation classes cannot be expressed in the interface. Moreover, since the client code must at some point construct objects, there will be depedendences on the concrete classes (which we will obviously try to localize). The *Factory* pattern, which we will discuss later in the course, addresses this particular problem.

7.8 Summary

Abstract types are characterized by their operations. Representation independence makes it possible to change the representation of a type without its clients being changed. In Java, access control mechanisms and interfaces can help ensure independence. Representation exposure is trickier though, and needs to be handled by careful programmer discipline.

Lecture 8: Rep invariants

This lecture begins a series of three lectures on the subject of *getting it right*. In these lectures, we'll study a variety of intellectual tools that help in the construction of high quality code. First we'll look at representation invariants and abstraction functions, which are key tools for designing and implementing abstract data types. Then we'll look at testing and code review strategies.

The use of these tools results in more reliable code. Representation invariants and abstraction functions, because they are applied earlier, tend to improve the structure of the code, which makes all subsequent activities easier. Representation invariants can also amplify the power of testing. All of the techniques can reduce the amount of time spent debugging, and if you are reduced to debugging, can make it more focused.

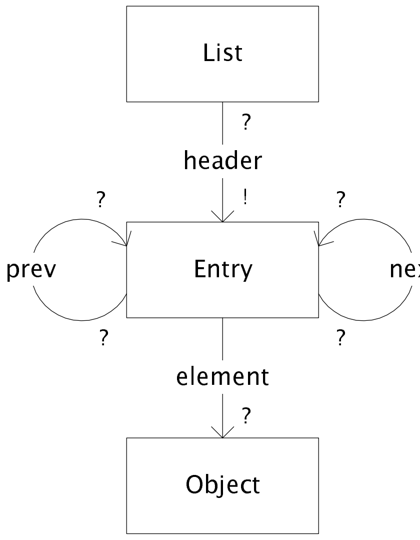
Today's lecture is about representation invariants. We'll discuss what they are, and how they're used in the design, documentation and testing of abstract types.

8.1 What is a Rep Invariant?

A representation invariant, or *rep invariant* for short, is a constraint that characterizes whether an instance of an abstract data type is well formed, from a representation point of view. Mathematically, it is a formula over the representation of an instance; you can view it as a function that takes objects of the abstract type and returns true or false depending on whether they are well formed:

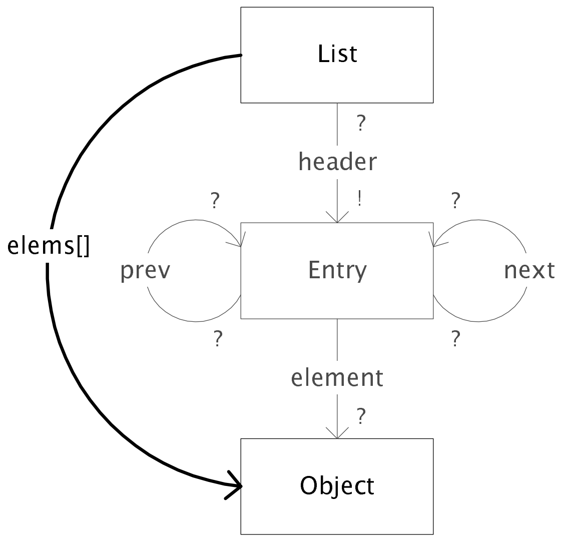
RI : Object -> Boolean

Consider the linked list implementation that we discussed last time. Here was its object model:



The *LinkedList* class has a field, *header*, that holds a reference to an object of the class *Entry*. This object has three fields: *element*, which holds a reference to an element of the list; *prev*, which points to the previous entry in the list; and *next*, which points to the next element.

This object model shows the *representation* of the data type. As we have mentioned before, object models can be drawn at various levels of abstraction. From the point of view of the user of the list, one might elide the box *Entry*, and just show a specification field from *List* to *Object*. This diagram shows that object model in black, with the representation in red (*Entry* and its incoming and outgoing arcs) hidden:



In tomorrow's lecture, we'll study the relationship between the representation and the abstract view; today, our concern is solely with the representation.

The representation invariant is a constraint that holds for every instance of the type. Our object model already gives us some of its properties:

- It shows, for example, that the *header* field holds a reference to an object of class Entry. This property is important but not very interesting, since the field is declared to have that type; this kind of property is more interesting for the contents of polymorphic containers such as vectors, whose element type cannot be expressed in the source code.
- The multiplicity marking on the target end of the *header* arrow says that the *header* field cannot be null.
- The multiplicities on the source end of the *next* and *prev* arrows say that each entry is pointed to by at most one other entry.

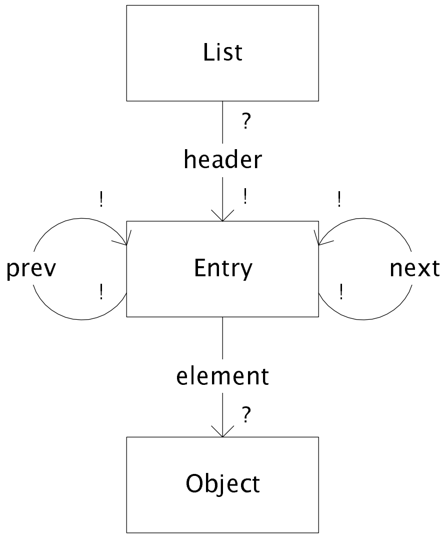
Some properties of the object model are not part of the representation invariant. For example, the fact that entries are not shared between lists (which is indicated by the multiplicity on the source end of the *header* arrow) is not a property of any single list.

There are properties of the representation invariant which are not shown in the graphical object model:

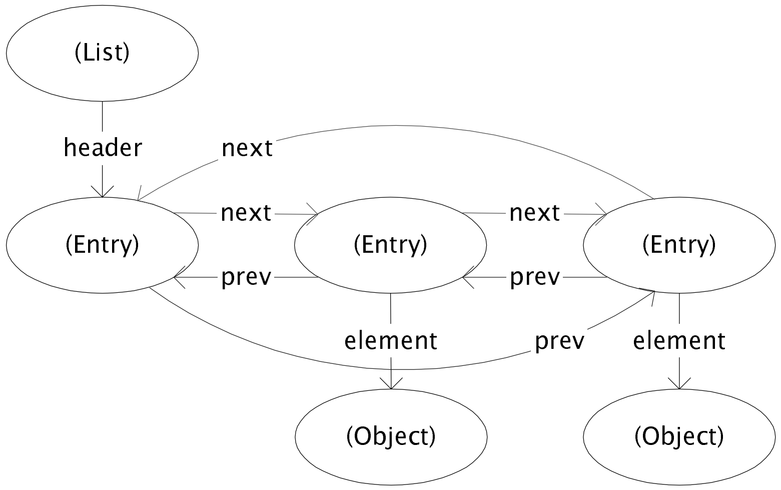
- When there are two *e1* and *e2* entries in the list, if *e1.next* = *e2*, then *e2.prev* = *e1*.
- The dummy entry at the front of the list has a null *element* field.

There are also properties that do not appear because the object model only shows objects and not primitive values. The representation of LinkedList has a field *size* that holds the size of the list. A property of the rep invariant is that *size* is equal to the number of entries in the list representation, minus one (since the first entry is a dummy).

In fact, in the Java implementation java.util.LinkedList, the object model has an additional constraint, reflected in the rep invariant. Every entry has a non-null *next* and *prev:*



Note the stronger multicities on the *next* and *prev* arrows. The list always forms a circle; when the list contains no elements, the dummy entry is connected to itself. Here is a sample list of two elements (and therefore three entries, including the dummy):



When examining a representation invariant, it is important to notice not only what constraints are present, but also which are missing. In this case, there is no requirement that the *element* field be non-null, nor that elements not be shared. This is what we'd expect: it allows a list to contain null references, and to contain the same object in multiple positions.

Let's summarize our rep invariant informally:

```
for every instance of the class LinkedList
    the header field is non-null
    the header field has a null element field
    there are (size + 1) entries
    the entries form a cycle starting and ending with the header entry
    for any entry, taking prev and then next returns you to the entry
```

We can also write this a bit more formally:

```
all p: LinkedList |
    p.header != null
    && p.header.element = null
    && p.size + 1 = | p.header.*next |
    && all e: p.header.*next | e.prev.next = e
```

To understand this formula, you need to know that

· for any expression *e* denoting some set of objects, and any field *f*, *e.f* denotes the set of objects you get if you follow *f* from each of the objects in *e*;
· *e.*f* means that you collect the set of objects obtained by following *f* any number of times from each of the objects in *e*;
· | e | is the number of objects in the set denoted by e.

So *p.header.*next* for example denotes the set of all entries in the list, because you get it by taking the list *p*, following the *header* field, and then following the *next* field any number of times.

One thing that this formula makes very clear is that the representation invariant is about a single linked list *p.* Another fine way to write the invariant is this:

```
R(p) =
    p.header != null
    && p.header.element = null
    && p.size + 1 = | p.header.*next |
    && all e: p.header.*next | e.prev.next = e
```

in which we view the invariant as a boolean function. This is the point of view we'll take when we convert the invariant to code as a runtime assertion.

The choice of invariant can have a major effect both on how easy it is to code the implementation of the abstract type, and how well it performs. Suppose we strengthen our invariant by requiring that the *element* field of all entries is non-null. This would allow us to detect the *header* entry by comparing its element to null; with the current invariant, operations that require traversal of the list must count entries instead or compare to the header field. Suppose, conversely, that we weaken the invariant on the *next* and *prev* pointers and allow *prev* at the start and *next* at the end to have any values. This will result in a need for special treatment for the entries at the start and end, resulting in less uniform code. Requiring *prev* at the start and *next* at the end both to be null doesn't help much.

8.2 A Variety of Invariants

Expert programmers use many different kinds of rep invariants. As a rough rule of thumb, the presence of strong invariants is a sign of expert programming; and conversely, the omission of invariants is oftena sign of incompetence.

Here is an example of how we might elaborate the invariant on our linked list. For the fields that we have, aside from requiring elements to be non-null, it's hard to strengthen the invariant. But we might add a hashmap that maps elements of the list to their entries:

Hashmap entries;

The rep invariant would now include the constraint

*all e: p.header.*next | p.entries[e.element] = e*

saying that for every entry in the list, looking up the element of that entry in the hashmap yields the entry itself, and also the constraint

*all k | k in dom(p.entries) ⇒ k in p.header.*next.element*

which says that if a key is mapped by the hashmap, then it must be an element of the list. This invariant now allows us to do constant time lookups. This use of the hashmap is an example of a large class of invariants that constrain a *redundant* component of the representation. The hashmap is redundant in the technical sense that it can be obtained from the list at any point. It's not redundant from a performance point of view, of course.

Caching constraints are another form of redundancy. Sometimes its convenient to store only part of the representation redundantly. For example, suppose the *get* operation of our linked list, which returns the element at a given index, is frequently called many times ina row with the same index. If we simply add two fields containing the last index requested and its element

Object lastElementRequested;
int lastIndexRequested;

we will be able to provide the result in constant time every call after the first in a series. The rep invariant will say that the given element indeed occurs at the given index, and it must be reset whenever the list is modified. Will these fields always be defined? The rep invariant will have to answer this question. Perhaps when the list is empty, the fields will be ignored:

size > 0 ⇒ e.element = lastElementRequested where e is the entry with index lastIndexRequested

This will require the fields to be set up as soon as there is an insertion, and it will require the fields to be reset whenever the list is mutated to hold information about some entry in the list. Instead, we might use a special value to indicate that no element is cached:

lastIndexRequested >= 0 ⇒
    e.element = lastElementRequested where e is the entry with index lastIndexRequested

This will allow the *lastIndexRequested* field to be initialized to -1; we can set the value when a *get* occurs, and simply invalidate the field again on a mutation.

The same representation may be used for different abstract types. For a set implemented as a list, the following additional invariants might be appropriate:

- The list may be *free of duplicates*. That is, we might ensure that no two entries contain the same element. Duplication usually has no utility for a set, and merely complicates the code and wastes space (although it may make insertion easier).
- The list may be *sorted*. This would allow a method that extracts the minimal element to be easily and efficiently implemented. It also marginally improves the efficiency of membership tests.

Perhaps the most important class of constraints are simply those that assert that a reference is non-null. As a rule of thumb, you should avoid ever having null references in your representations; they almost always lead to trouble. A particularly horrible error occasionally made by novices is to use a null reference as if it were a legitimate instance of the type: for example, as the empty list. This makes life very unpleasant for the client, since every method call on a list must be wrapped in a test to see if the reference is null. More programmers make this error inside a representation. Suppose a set is represented as field *elemvec* that is a vector of elements. If you allow the field to have a null value to represent the empty set, you'll end up peppering your code with null-reference tests. Far better to impose the rep invariant that the reference is non-null, and use an empty vector for an empty set.

8.3 Inductive Reasoning

The rep invariant makes *modular reasoning* possible. To check whether an operation is implemented correctly, we don't need to look at any other methods. Instead, we appeal to the principle of *induction*. We ensure that every constructor creates an object that satisfies the invariant, and that every mutator and producer *preserves* the invariant: that is, if given an object that satisfies it, it produces one that also satisfies it. Now we can argue that every object of the type satisfies the rep invariant, since it must have been produced by a constructor and some sequence of mutator or producer applications.

To see how this works, let's look at some sample operations of our *LinkedList* class. The representation is declared in Java like this:

```
public class LinkedList {
    Entry header;
    int size;
    class Entry {
        Object element;
        Entry prev;
        Entry next;
        Entry (Object e, Entry p, Entry n) {element = e; prev = p; next = n;}
        }
    …
```

Here's our constructor:

```
public LinkedList () {
    size = 0;
    header = new Entry (null, null, null);
```

```
        header.prev = header.next = header;
        }
```

Notice that it *establishes* the invariant: it creates the dummy element, forms the cycle, and sets the size appropriately.

The mutator *add* takes an element and adds it to the end of the list:

```
    public void add (Object o) {
        Entry e = new Entry (o, header.prev, header);
        e.prev.next = e;
        e.next.prev = e;
        size++;
        }
```

To check this method, we can assume that the invariant holds on entry. Our task is to show that it also holds on exit. The effect of the code is to splice in a new entry just before the *header* entry, so we can see that the constraint that the entries form a cycle is preserved. Note that one consequence of being able to assume the invariant on entry is that we don't need to do null reference checks: we can assume that *e.previous* and *e.next* are non-null, for example, because they are entries that existed in the list on entry to the method, and the rep invariant tells us that all entries have non-null *prev* and *next* fields.

Finally, let's look at an observer. The operation *getLast* returns the last element of the list or throws an exception if the list is empty:

```
    public Object getLast () {
        if (size == 0) throw new NoSuchElementException ();
        return header.prev.element;
        }
```

Again, we assume the invariant on entry. This allows us to dereference *header.next*, which the rep invariant tells us cannot be null. Checking that the invariant is preserved is trivial in this case, since there are no modifications.

8.4 Rep Exposure

The notion of rep invariants that we have espoused offers a systematic method for checking the correctness of abstract data type implementations. (We haven't yet considered how to ensure that a mutator or producer generates the *right* instance of the type, only that it produces a well-formed instance. Tomorrow, when we study abstraction functions, we'll look at that issue.)

Our method says that we can consider the operations one by one, and then appeal to induction to show that every instance will be well formed. A crucial aspect of this method is local reasoning: we can examine the operations individually, and certainly don't need to look at client code.

In fact, this method is not always sound. It has a proviso: that the representation must not be *exposed.* Representation exposure is a nasty problem, because it can arise unexpectedly, and have disastrous effects that are hard to pin down.

The simplest form of rep exposure involves allowing client code to manipulate the representation directly. This is easy to rule out, however, by making all fields of the abstract type private, so we don't usually even regard it as a kind of exposure.

Instead, the rep exposure we'll be concerned with arises because an object inside the representation is accessible from the outside, through a different path. Two common ways in which this happens are

· that a reference to an object that is part of the rep is passed out, as the result of an operation;
· or that an object is passed in and made part of the rep despite being accessible by an existing reference from the outside.

We saw an example of rep exposure in our last lecture. If we were so foolish as to provide a method

```
public Entry getEntry (int i)
```

that returns the entry at index *i* in the list, subsequent modifications to the entry could break the invariant.

A more plausible exposure, which is quite common, arises from implementing a method that returns a collection. When the representation already contains a collection object of the appropriate type, it is tempting to return it directly. For example, *LinkedList* has a method *toArray* that returns an array of elements corresponding to the elements of the list. If we had implemented the list itself as an array, we might just return the array itself. If the rep invariant requires some other part of the rep to be related to the array (eg, a *size* field to correspond to the index at which a null reference first appears) a modification to this array may break the invariant:

```
a = p.toArray ();        // exposes the rep
a[i] = null;             //ouch! breaks invariant
p.get (i);               // now behaves unpredictably
```

Once the invariant is violated, all hell breaks loose: subsequent operations may behave in arbitrary ways.

A more subtle variant of this problem arises with iterators. Many Java classes have a method that returns an iterator. Building an iterator is work, so we might be tempted to use one that's already provided by the Java library. Suppose our representation includes a field *vec* that holds a set of elements, and we want to implement a method

```
public Iterator elements ()
```

that returns an iterator over these elements. Noticing that the *Vector* class provides its own method that returns an iterator, we implement our method like this:

```
public Iterator elements () {
    return vec.iterator ();
    }
```

Unfortunately, this is a rep exposure. Classes that implement the *Iterator* interface in Java must offer *add* and *remove* methods. So the result of this method is an object that can actually be modified outside the abstract type. Since the *iterator* method of *Vector* returns an iterator that shares state with the vector it is called on, this object will be part of the state of our representation.

Another variety of rep exposure can happen *between* two objects of a type. This is a rather strange kind of exposure, since it involves violations due to calls to abstract operations. Suppose, for example, we have a linked list whose representation is

```
public class LinkedList {
    Entry header;
    int size;
    class Entry {
        Entry next;
        Object element;
        Entry (Entry n, Object e) {next = n; element = e;}
        }
    ...
```

We want a *cdr* operation (as in Scheme) that returns a list containing all but the first element. Here is a bad implementation:

```
public LinkedList cdr () {
    LinkedList p = new LinkedList ();
    p.header.next  = this.header.next.next;
    p.size = this.size - 1;
    return p;
    }
```

We create a new list object, with its own dummy entry, and we make the *next* field of this dummy entry point to the second element of the original list. If we now make a call to *remove* on one of the lists, the rep invariant of the other list will be violated, since the size field will no longer correspond to the number of elements in the list. We should have copied the list instead.

Object models can help expose representation invariants. Two arrows pointing at the same box indicate potential sharing in the heap, and thus potential exposure. Whenever the source end of an arrow is not marked with a multiplicity, you should be concerned that sharing may lead to a rep exposure between objects of the type.

8.5 Element Equality & Rep Exposure

Rep exposure is actually a very subtle notion, because it is not always clear what belongs to the rep. Is it a rep exposure for list operations to return elements of the list? Let's see why it might be.

Suppose our list representation has the invariant that there are no duplicates (eg. because the list represents a set). Furthermore, let's say that our notion of equality is based on the contents of the elements. For example, if the elements are themselves lists, we'll regard two elements to be equal if they contain the same sequence of elements. Now we have a rep exposure: we can create duplicates simply by modifying the elements of the list from outside, making two equal when they were not equal previously.

The root of the problem here is not the passing in or out of the element objects: that can't be avoided. Rather, it's the notion of equality. If our set determined equality of the elements using reference equality, so that two elements are equal when they are the same object, the rep exposure would not arise. That would itself lead to problems though, since it would result in two strings that represent the same sequence of characters being regarded as distinct. The best approach, therefore, is to have the set call the *equals* method of the element type, and for *equals* of every type to be *reference* equality when the type is mutable.

Unfortunately, the collection classes in Java are not designed in this way. Two *LinkedList* objects, for example, are regarded as equal if they contain equal elements, even if they are distinct list objects. Now if we insert such lists as keys into a *Hashmap* or *Hashtable*, a subsequent modification to a list can break the hash table invariant. This can lead to very strange behaviour:

```
LinkedList k;
Hashmap m;
...
m.put (k, v);        // insert the list as a key into the hash map
k.add (e);           // mutate the key; breaks the rep invariant of m
x = m.get (k);       // now x may not be v
```

The problem is that the key is stored in a fixed slot in the hash table according to its computed hash-code. Mutating the key may change its hashcode, so that when looked up a second time, the hash table code looks in the wrong slot.

To work around this problem, you should either wrap objects before you insert them into hashtables (so that they have an *equals* method you define yourself), or you should make sure you never mutate keys. Neither of these is convenient though.

8.6 Rep Invariants as Assertions

Many rep invariants can be translated straightforwardly into code. In our *LinkedList* implementation, for example, it's easy to check that the *prev* and *next* pointers commute, that the header is non-null, etc. Even if the check is expensive, it's worth coding it up, since expensive properties tend to be tricky ones that are more likely to be violated.

The rep invariant assertion checker can be coded as a method *checkRep* of no arguments that throws an exception if the rep invariant is violated, with a message indicating which constraint is broken. Calls to *checkRep* can be placed at the start and end of every public mutator and at the end of every constructor. Although our induction argument suggests that placing it at the start of mutators should not be necessary, there is a risk of representation exposure which would cause the rep to change between the end of one operation and the start of the next. For an observer, the call to *checkRep* should be placed at the start, and at the end also if it changes the rep (see the discussion of 'benevolent side effects' in the next lecture), and maybe even if it supposedly doesn't (since you may be wrong).

If the check is very expensive, you'll probably want to comment it out or turn it off in the release version. Otherwise it makes sense to leave it in. Be careful in how you judge performance here; novices are often much too ready to worry about performance improvements that turn out to be negligible. Before dropping a check, you should have some evidence that it's expensive, such as an analysis with a profiler showing that indeed the check is a hotspot, or a theoretical argument, for example that the check turns a constant time operation into a linear time one.

Checking rep invariants is especially useful because it helps to localize bugs. Suppose you forget to update the *size* field in one of the mutator operations of the linked list. This bug will not cause problems until a subsequent operation tries to make use of the size, and even then some operations may succeed. A call to get with a low index, for example, might work just fine. When an operation finally fails, the bug will be obscure. Perhaps *get* with a high index fails because the implementation counts back from the high end of the list. It will take some debugging to discover that there is no fault with the *get* operation at all, but that it was passed a bad object. With checks on the rep invariant inserted, however, the bug would be noticed as soon as the offending operation executed, and the programmer's attention would be drawn to the operation that actually contains the error, not the operation that first fails.

8.7 Summary

Why use rep invariants? Recording the invariant may seem like extra work, but actually it saves work:

· It makes modular reasoning possible. Without the rep invariant documented, you might have to read all the methods to understand what's going on before you can confidently add a new method.
· It helps catch errors. By implementing the invariant as a runtime assertion, you can find bugs that are hard to track down by other means.

Moreover, choosing a strong invariant is often a good design decision:

· It tends to result in cleaner code (eg, because there are no null reference checks).
· It can help avoid errors. For example, you might allow a linked list to contain unused elements beyond the end, and use the *size* field to avoid running over the end, but this is asking for trouble (and will prevent the unused elements from being garbage collected).
· It can improve performance (eg, because of caching or redundancy).

You should therefore design and record the rep invariant as part of the design of the representation, before you start coding. When you're trying to understand an abstract data type, writing down the rep invariant is a good place to start.

Lecture 9: Abstraction Functions

In this lecture, we turn to our second tool for understanding abstract data types: the abstraction function. The rep invariant describes whether an instance of a type is well formed; the abstraction function tells us how to interpret it. It's impossible to code an abstract type or modify it without understanding the abstraction function at least informally. Writing it down is useful, especially for maintainers, and crucial in tricky cases.

9.1 Interpreting the Representation

Recall from last lecture the mutator *add*, which takes an element and adds it to the end of the list:

```
public void add (Object o) {
    Entry e = new Entry (o, header.prev, header);
    e.prev.next = e;
    e.next.prev = e;
    size++;
    }
```

We checked that this operation preserved the rep invariant, by correctly splicing a new entry into the list. What we didn't check, however, was that it was spliced into the right position. Is the new element inserted into the start or the end of the list? It looks as if it's at the end, but that assumes that the order of entries corresponds to the order of elements. It would be quite possible (although perhaps a bit perverse) for a list *p* with elements *o1, o2, o3* to have

```
p.header.next.element = o3;
p.header.next.next.element = o2;
p.header.next.next.element = o1;
```

To resolve this problem, we need to know how the representation is *interpreted*: that is, how to view an instance of LinkedList as an abstract sequence of elements. This is what the abstraction function provides. The abstraction function for our implementation is:

```
A(p) =
    if p.size = 0 then
        <>  (the empty list)
    else
        <p.header.next.element, p.header.next.next.element, ...>
        (the sequence of elements with indices 0.. p.size-1 whose ith element is $p.next^{i+1}$.element)
```

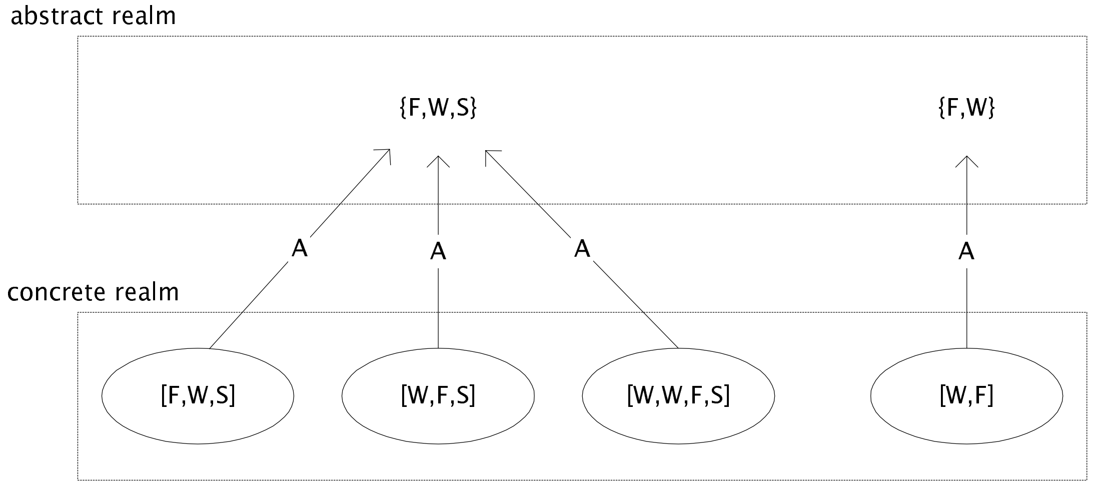
9.2 Abstract and Concrete Objects

In thinking about an abstract type, it helps to imagine objects in two distinct realms. In the concrete realm, we have the actual objects of the implementation. In the abstract realm, we have mathematical objects that correspond to the way the specification of the abstract type describes its values.

Suppose we're building a program for handling registration of courses at a university. For a given course, we need to indicate which of the four terms *Fall*, *Winter*, *Spring* and *Summer* the course is offered in. In good MIT style, we'll call these *F*, *W*, *S* and *U*. What we need is a type *SeasonSet* whose values are sets of seasons; we'll assume we already have a type *Season*. This will allow us to write code like this:

if (course.seasons.contains (Season.SUMMER)) ...

There are many ways to represent our type. We could be lazy and use *java.util.ArrayList*; this will allow us to write most of our methods as simple wrappers. The abstract and concrete realms might look like this:



The oval below labelled [*F,W,S*] denotes a *concrete* object containing the array list whose first element is *F*, second is *W*, and third is *S*. The oval above labelled {*F,W,S*} denotes an *abstract* set containing three elements *F*, *W* and *S*. Note that there may be multiple representations of the same abstract set: {*F*, *W*, *S*}, for example, can also be represented by [*W,F*, *S*], the order being immaterial, or by [*W,W,F*, *S*] if the rep invariant allows duplicates. (Of course there are many abstract sets and concrete objects that we have not shown; the diagram just gives a sample.)

The relationship between the two realms is a function, since each concrete object is interpreted as at most one abstract value. The function may be partial, since some concrete objects -- namely those that violate the rep invariant -- have no interpretation. This function is the *abstraction function*, and is denoted by the arrows marked *A* in the diagram.

Suppose our SeasonSet class has a field *eltlist* holding the *ArrayList*. Then we can write the abstraction function like this:

A(s) = {s.eltlist.elts [i] | 0 <= i <= size(s.eltlist)}

That is, the set consists of all the elements of the list.

Different representations have different abstraction functions. Another way to represent our SeasonSet is using an array of 4 booleans. Here the abstraction function may, for example, map

[true, false, true, false]

to {*F*,*S*}, assuming the order *F, W, S, U* for the elements of the array. This order is the information conveyed by the abstraction function, which might be written, assuming the array is stored in a field *boolarr* as

```
A(s) =
    (if s.boolarr[0] then {F} else {})∪
    (if s.boolarr[1] then {W} else {})∪
    (if s.boolarr[2] then {S} else {})∪
    (if s.boolarr[3] then {U} else {})
```

We could equally well have chosen a different abstraction function, that orders the seasons differently:

```
A(s) =
    (if s.boolarr[0] then {S} else {})∪
    (if s.boolarr[1] then {U} else {})∪
    (if s.boolarr[2] then {F} else {})∪
    (if s.boolarr[3] then {W} else {})
```

An important lesson from this last example is that 'choosing a representation' means more than naming some fields and selecting their types. The very same array of booleans can be interpreted in different ways; the abstraction function tells us which. Likewise, in our linked list example, the abstraction function tells us how the order of entries corresponds to the order of elements. It is a common error of novices to imagine that the abstraction function is obvious, since you can always guess what it is from the declarations in the code. Unfortunately, this is often not true: it takes careful reading of the linked list code to discover that the first entry is a dummy entry, for example.

9.3 Example: Boolean Formulas in CNF

Let's look at an example of a simple representation with a tricky abstraction function. A boolean formula is a mathematical formula constructed from *propositions* (symbols that can be assigned the values true and false) and logical *operators*. For example, the formula

courseSix ⇒ *sixOneSeventy*

uses two propositions, *courseSix* and *sixOneSeventy*, and the logical implication operator. It says that if *courseSix* is true, *sixOneSeventy* is true also. A boolean formula is *satisfiable* if there is some assignment of boolean values to the propositions that makes the formula true. This formula is satisfiable, since we can set *courseSix* to false, or we can set both propositions to true.

An algorithm that determines whether a formula is satisfiable, and if so returns satisfying values for the propositions is called a *SAT solver*. SAT solvers have many applications, and their technology has advanced dramatically in the last decade. They are used in design tools for checking design constraints, in planners for finding plans, in testing tools for finding tests that expose particular classes of error, and so on. A SAT solver can also be used to check a proof. Suppose we assert that it follows from

courseSix ⇒ *sixOneSeventy*

and

$sixOneSeventy \Rightarrow lateNights$

that

$courseSix \Rightarrow lateNights$

This is elementary reasoning using modus ponens, of course, but let's see how to check it with a SAT solver. We simply conjoin the premises to the negation of the conclusion:

$(courseSix \Rightarrow sixOneSeventy) \wedge (sixOneSeventy \Rightarrow lateNights) \wedge \neg(courseSix \Rightarrow lateNights)$

and present this formula to the solver. The solver will find it not satisfiable, and will have demonstrated that it is impossible to have the premises be true and not the conclusion: in other words, the proof is valid.

Most SAT solvers use a representation of boolean formulas known as *conjunctive normal form*, or *CNF* for short. A formula in CNF is a set of clauses; each clause is a set of literals; a literal is a proposition or its negation. The formula is interpreted as a conjunction of its clauses and each clauses is interpreted as a disjunction of its literals. A more helpful name for CNF is *product of sums*, which makes it clear that the outermost operator is product (ie., conjunction).

For example, the CNF formula

$\{\{a\}\{\neg b,c\}\}$

is equivalent to the conventional formula

$a \wedge (\neg b \vee c)$

Our formula above would be represented in CNF as

$\{\{\neg courseSix,sixOneSeventy\}, \{\neg sixOneSeventy, lateNights\} \{courseSix\}\{\neg lateNights\}\}$

Let's consider now how we might build an abstract data type that holds formulas in CNF. Suppose we already have a class *Literal* for representing literals. Here is one reasonable representation that uses the Java library *ArrayList* class:

```
public class Formula {
    private ArrayList clauses;
    ...
    }
```

The *clauses* field is an *ArrayList* whose elements are themselves *ArrayLists* of literals.

Our representation invariant might then be

```
R(f) =
    f.clauses != null &&
    all c: f.clauses.elts |
        c instanceof ArrayList && c != null &&
            all l: c.elts | c instanceof Literal && c != null
```

I've used the specification field *elts* here to denote the elements of an ArrayList. The rep invariant says that the elements of the *ArrayList* clauses are non-null *ArrayLists*, each containing elements that are non-null *Literals*.

Here, finally, is the abstraction function:

$A(f) = true \wedge C\ (f.clauses.elts[0]) \wedge \ldots \wedge C(f.clauses.elts[(size(f.clauses)\ \text{-}1])$
 $\quad$ $where\ C(c) = false \vee c.elts[0] \vee \ldots \vee c.elts[0]$

Note how I've introduced an auxiliary function *C* that abstracts clauses into formulas. Looking at this definition, we can resolve the meaning of the boundary cases. Suppose *f.clauses* is an empty *ArrayList*. Then *A(f)* will be just true, since the conjuncts on the right-hand side of the first line disappear. Suppose *f.clauses* contains a single clause *c*, which itself is an empty ArrayList. Then *C(c)* will be false, and *A(f)* will be false too. These are our two basic boolean values: true is represented by the empty set of clauses, and false by the set containing the empty clause.

9.4 Specification Fields

The abstract values of many abstract data types have a tuple structure at the top-level. For example, a line is a pair of points; a mailing address is a number, a street, a city and a zipcode; a URL is a protocol, a host name, and a resource name.

In these cases, one can specify a single function that maps representation objects to tuples. This is the approach followed by our textbook. It's convenient, and perhaps more natural, to break the function into several separate functions, each viewed as an *specification field*.

For example, we might represent a *Card* datatype, used in card game program, by a single integer in a field *index*. The rep invariant requires *index* be in the range 0..51. We might have two specification fields defined as follows:

c.suit = S(c.index div 13)
c.val =V (c.index mod 13)
 where
  S(0) = Hearts, S(1) = Spades, S(2) = Clubs, S(3) = Diamonds
  V(1) = Ace, V(2) = 2, ..., V(11) = Jack, V(12) = Queen, V(0) = King

so that a *Card* object with *index* field of 3, for example, would correspond to the 3 of Hearts; 14 corresponds to the Ace of Spades.This abstraction function maps each representation object *c* to a pair (*c.suit, c.val*), but rather than writing it as a single function, we've specified it as two separate ones, one for each specification field.

This scheme is so convenient that we'll use it even when there is only one specification field. We've actually seen this many times before. When we referred to the ith element of a vector *v* as *v.elts[i]*, this used a specification field *elts* that is a mathematical sequence. It allowed us to talk about the elements of the vector without mentioning the representation. Without the specification field, we would have to write *A(v)* to denote the vector's element sequence, to distinguish it from the value of *v* itself -- a Java object reference.

Note that we've actually already used specification fields, whenver we needed to refer to the abstract value denoted by an object. In giving the abstraction function for *SeasonSet*

A(s) = {s.eltlist.elts [i] | 0 <= i <= size(s.eltlist)}

for example, in order to refer to the element at index *i* of the *ArrayList*, we needed to first obtain the abstract sequence represented by the list. The expression *s.eltlist* is a reference to a Java *ArrayList* object; *s.eltlist.elts* is the sequence of elements it represents. This definition thus actually mixed two styles; we should write either

A(s) = {A(s.eltlist) [i] | 0 <= i <= size(s.eltlist)}

using the single abstraction function, or

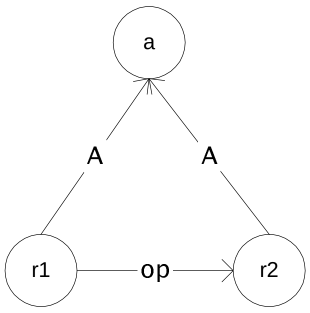
s.elems = {s.eltlist.elts [i] | 0 <= i <= size(s.eltlist)}

using specification fields.

9.5 Benevolent Side Effects

What is an *observer* operation? In our introductory lecture on representation independence and data abstraction, we defined it as an operation that does not mutate the object. We can now give a more liberal definition.

An operation may mutate an object of the type so that the fields of the representation change, will maintaining the abstract value it denotes. We can illustrate this phenomenon in general with a diagram:



The execution of the operation *op* mutates the representation of an object from *r1* to *r2*. But *r1* and *r2* are mapped by the abstraction function *A* to the same abstract value *a*, so the client of the datatype cannot observe that any change has occurred.

We have in fact already seen an example of such an observer. We mentioned in the last lecture that the *get* method of *LinkedList* may cache the last element extracted, so that repeated calls to *get* for the same *index* will be speeded up. This writing to the cache (in this case just the two fields) certainly changes the rep, but it has no effect on the value of the object as it may be observed by calling operations of the type. The client cannot tell whether a lookup has been cached (except by noticing the improvement in performance).

In general, then, we can allow observers to mutate the rep, so long as the abstract value is preserved. We will need to ensure that the rep invariant is not broken, and if we have coded the invariant as a method *checkRep*, we should insert it at the start and end of observers.

9.6 Idioms for Expressing Abstraction Functions

Writing abstraction functions is difficult, because we don't have a language for talking about abstract values. Such a language, called a formal specification language, would also allow us to specify our operations more precisely. But in practice, formal specification languages are not easy to use, so they are usually reserved for critical projects.

Nevertheless, with a little practice, it's possible to write fairly clear and simple abstraction functions. Here are some tips.

First, figure out what the abstract values looks like. If an abstract value is a tuple of several values, examine each of these values and define a separate specification field for it. Can the abstract value be described with standard mathematical sets, sequences and relations? If so, you're in luck: you can now use whatever notation you're comfortable with for constructing and manipulating those mathematical values. If not, you might decide to specify the abstraction function by example: that's what I did above with the auxiliary functions *S* and *V* in the *CardSet* example, lacking a standard way to model values such as 'Hearts'.

Second, pick a strategy for mapping concrete objects to the abstract values you've chosen. Here are some common idioms:

- *Comprehension*. The set comprehension expression *{x | P(x)}* denotes the set of all elements *x* that satisfy the property *P*. Suppose we have a set of integers in the range 0..255 represented as an array of booleans (ie, a bit string) stored in a field *bitvec*. Then we might write the abstraction function like this:

  s.elems = {i | s.bitvec[i]}

- *Recursion*. Recursive representations are often well treated with recursive functions. Suppose we have a set represented as a binary tree, with fields *val* (the value of the tree node, null if the tree is empty), *left* (the left tree, null for a leaf) and *right* (the right tree, null for a leaf).Then we might write the abstraction function like this:

  s.elems = if (t.val = null) then {} else {t.val} $\cup$ *t.left.elems* $\cup$ *t.right.elems*

- *Projection*. Sometimes it's easiest to write a formula that relates a part, or a projection, of the concrete object to a part of the abstract value. Suppose we have a sequence datatype represented as an array stored in a field *eltarray,* with a field *max* giving the index of the highest array element that corresponds to an element of the abstract sequence. Then, using a specification field elts that gives the abstract sequence of elements, we might write the abstraction function as two constraints:

  size (s.elts) = s.max + 1
  for i: 0.. size (s.elts) | s.elts [i] = s.eltarray [i]

- *By example*. Finally, you can always fall back on simply illustrating the abstraction function on an example, and hoping that the reader infers the generalization correctly. This is what I did when showing the abstraction for *LinkedList*:

```
p.elts =
    if p.size = 0 then
        <>  (the empty list)
    else
        <p.header.next.element, p.header.next.next.element, ...>
        (the sequence of elements with indices 0.. p.size-1 whose ith element is $p.next^{i+1}$.element)
```

9.7 Summary

The abstraction function specifies how the representation of an abstract data type is interpreted as an abstract value. Together with the representation invariant, it allows us to reason in a modular fashion about the correctness of an operation of the type.

In practice, abstraction functions are harder to write than representation invariants. Writing down a rep invariant is always worthwhile, and you should always do it. Writing down an abstraction function is often useful, even if only done informally. But sometimes the abstract domain is hard to characterize, and the extra work of writing an elaborate abstraction function is not rewarded. You need to use your judgment.

Lecture 11: Problem Object Models

In the next two lectures, we'll be looking at a new topic: how to model problems. It turns out that one of the hardest problems in software development is figuring out what the problem is. If you don't understand the problem, there's a serious risk that you'll build the wrong system. Moreover, if you understand the problem well, you're much more likely to build a system that's well structured and flexible.

In these lectures, we'll study *object models* as a tool for analyzing problems. An object model is a lightweight but precise way to characterize a potentially infinite set of configurations. Many problems involve configurations, so object models have widespread application. In constructing an object model, one is forced to answer a variety of simple but important questions; this process often exposes inadequate understanding of a problem. At the end, the object model that results is a succinct description of an important aspect of the problem, and can be used to guide design.

We've already seen object model notation for describing configurations in code. I presented the notation as if it were designed for describing heap structures in Java. But in fact the notation is far more general, and the same notation can be used at many levels of abstraction, from object configurations in code to the most abstract properties of a problem domain.

In today's lecture, we'll revisit the object modelling notation, and give it a more precise (and more abstract interpretation). Tomorrow, we'll focus more on the process of modelling, and we'll investigate a particular problem using object modelling.

11.1 Three Problems

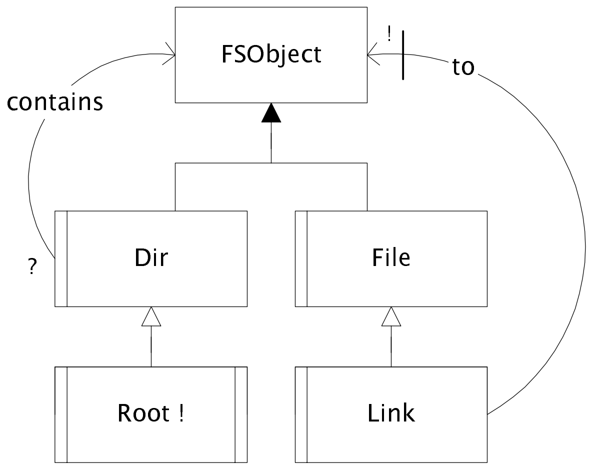
To illustrate the variety of artifacts that can be modelled, we'll construct object models for three rather different domains:

- *A file system*. In the first, we'll express some of the essential structure of a typical file system with a directory hierarchy and links.
- *Air traffic control.* In the second, we'll look at something less familiar. Rather than modelling an existing computer system, we'll model an organizational structure. We'll describe how airspace is divided into sectors, and how controllers are associated with aircraft.
- *A model of Java itself.* In the third, we'll build a generic model that shows how variables, references and objects are related in a Java program.

As we consider these examples, you should bear in mind that our purpose is to show how one can construct *some* model of a problem or a system. In general, there is no single, best model; there are usually many ways to model the same problem. Which one you choose depends not so much on personal taste as on which properties matter for the task at hand. Since we don't have particular tasks, we can't make these judgments. In your problem set and in your final project, however, you will have particular tasks, and you'll have to take these into account. For example, when building a model of street maps for your MapQuick system, you will want to bear in mind which features of the map are relevant to finding and displaying directions.

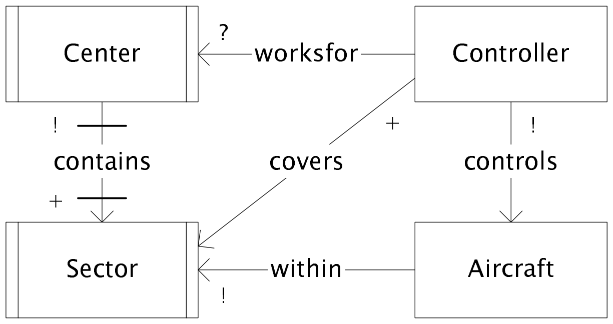
11.2 Example 1: File System

Here is an object model of a simple file system. It shows a classification of file system objects (*FSObject*) into directories (*Dir*) and files (*File*). There is a root directory (Root), and some of the files are links (*Link*). Links are associated with file system objects, and directories contain file system objects.



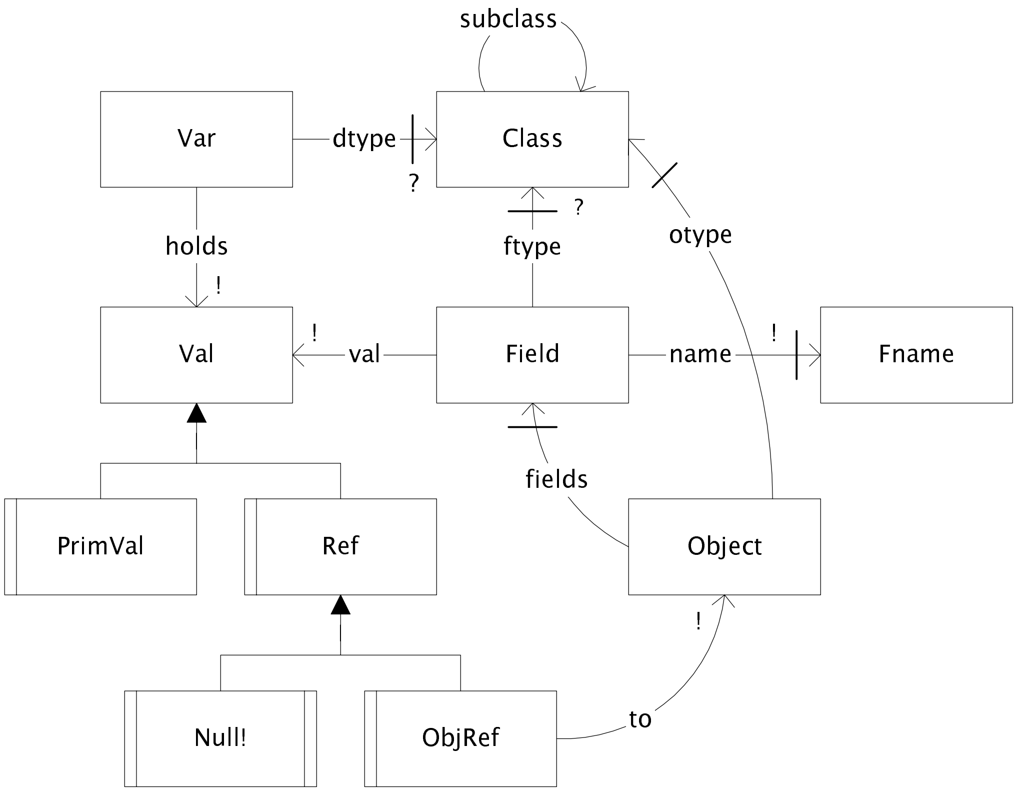
11.3 Example 2: Air Traffic Control

Our object model of air-traffic control captures only a few vital notions. It shows that there are centers (*Center*), each of which contains one or more sectors (*Sector*). Air-traffic controllers (*Controller*) work for centers, and cover sectors. An aircraft (Aircraft) is controlled by a controller and flies within a sector.



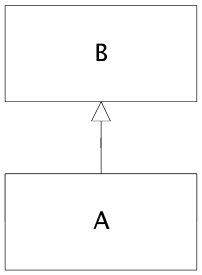
11.4 Example 3: Java Objects

In a Java program, there are variables (*Var*), which have declared types that are classes (*Class*). Classes are related by subclassing. Variables hold values (*Val*), divided into primitive values (*PrimVal*) and references (*Ref*); references are further divided into the null reference (*Null*) and references to objects (*ObjRef*), which point to objects (Object). An object has fields (*Field*), each of which has a name (*Fname*) and a value. Objects have types too. The model ignores interfaces and the types associated with primitive values.



11.5 Sets and Classification

The backbone of the object model is a classification hierarchy. The hierarchy may have many roots; these are the *domains*, and they constitute the coarsest classification of objects. Each box in an object model denotes a set of objects. An arrow with a closed head from A to B



denotes a subset relationship: A is a subset of B. These edges are sometimes called 'is-a' edges, because they can be read 'every A is a B'. A box without an outgoing arrow is a domain.

When two subsets share an arrow, they are disjoint. When the arrowhead is filled, the subsets exhaust the superset: that is, there are no members of the superset that are not members of one of the subsets.

In this case, the subsets form a *partition*: every member of the superset belongs to exactly one subset.

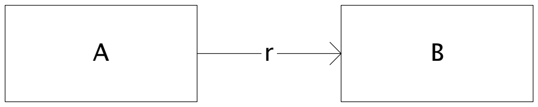
Thus, in the file system example, there is a single domain, *FSObject*. This domain is partitioned into *File* and *Dir*. *Root* is a subset of *Dir*, and *Link* is a subset of *File*: that is, every *Root* is a *Dir*, and every *Link* is a *File*.

In the air-traffic control example, each set is a domain, and there is no further classification.

In the Java example, the domains are *Var*, *Val*, *Object*, *Class*, *Field*, and *Fname*. The domain *Val* is partitioned into *PrimVal* and *Ref*; *Ref* is then partitioned into *Null* and *ObjRef*.

11.6 Relations

Arrows with open heads denote relations. A relation is a set of pairs. When an arrow marked *r* is drawn from box *A* to box *B*



this means that there is a relation that associates members of A with members of B. The relation r is a set of pairs whose first elements are drawn from A, and whose second elements are drawn from B. The relation is ordered: we need to know that it goes from A to B, and not from B to A. For example, in the file system, a directory d1 containing a directory d2 is different from a directory d2 containing a directory d1. In our code object models, we always ordered the relation to match the direction of the reference in Java, but in general, no notion of navigability is implied, and, when implemented, a relation need not be realized as a field in the code.

In the file system example, *to* is a relation that associates links with the file system objects they point to; the pair *(l,o)* will belong to the relation if link *l* points to object *o*. In the air-traffic control example, *controls* associates a controller with an aircraft when that controller has responsibility for giving clearances to that aircraft. In the Java example, *holds* relates a variable *x* and a value *v* when *x* holds the value of *v*.

When a relation arrow connects a superset, it admits elements of a subset. For example, the *contains* relation in the file system model maps *Dir* to *FSObject*. Because *Link* is a subset of *FSObject*, this means that a directory may contain a link. A connection to a subset does not admit elements of the superset however. So because *contains* maps directories, it does not in general map file system objects: a file system object that is not a directory (that is, a file) cannot contain other objects. Thus moving a relation down in the classification hierarchy makes the object model more precise.

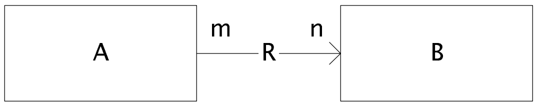
11.7 Multiplicity

The multiplicity symbols are:

- * (zero or more)
- + (one or more)
- ? (zero or one)

· ! (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* members 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* members of class A.



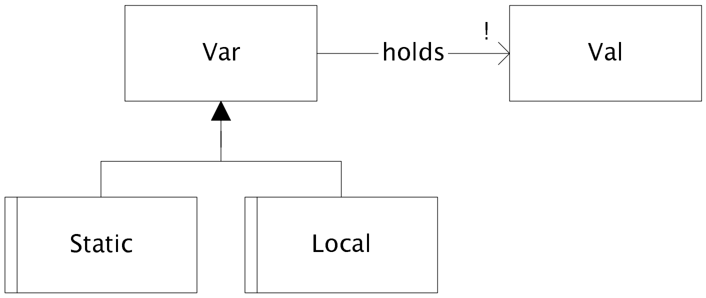
In the file system example, the multiplicity on the source end of the *contains* arrow tells us that a file system object is contained by at most one directory. The multiplicity on the target end of the *to* arrow says that a link points to exactly one object.

In the air-traffic control example, the multiplicities say that a controller works for at most one center; that a center contains at least one sector, and each sector is contained by exactly one center; that an aircraft is always in exactly one sector (that is, they don't overlap or have gaps), and that each aircraft is controlled by exactly one controller. A sector is covered by at least one controller.

In the Java example, the model says that every variable and every field have a value (although it may be null). Every object reference points to an object. These are fundamental properties of a safe language. But note that not every variable has a declared type, since we are only modelling types that are classes.

Multiplicity symbols can be used to constrain the size of sets too. So, in the file system example, the ! after *Root* says that there is exactly one root object. In the Java example, *Null!* says there is exactly one null reference (which does not mean that there is only one variable or field with that value,but merely that there aren't two different reference values that are both the null reference).

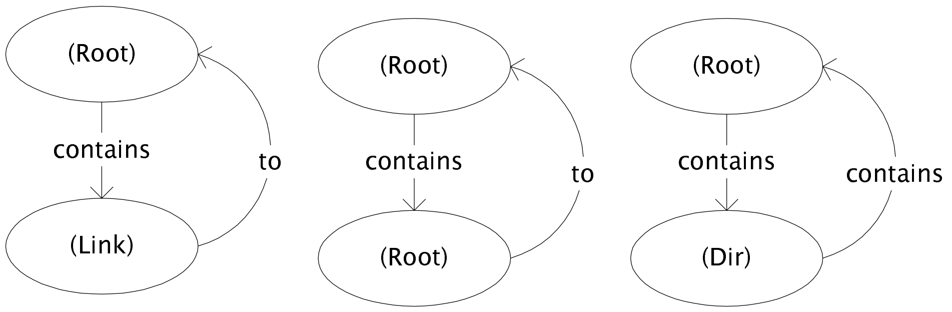
Multiplicity constraints on relations apply to all members of the relevant sets, and therefore to members of their subsets too. For example,suppose we classified variables into static variables and local variables;



then the constraint that every variable holds a value, given by the ! on the target end of the *holds* relation, would apply to both kinds of variable.

11.8 Semantics

The meaning of an object model is given by the set of configurations that the model admits. We've used object diagrams to illustrate these configurations before. Here, for example, are three object diagrams that are candidate configurations of the file system:



The object model admits the first. It rejects the second, since the multiplity of Root prevents there from being two Root objects. It admits the third, even though it is undesirable, because it is only a partial model and doesn't express the constraint that the directory structure should be acyclic.

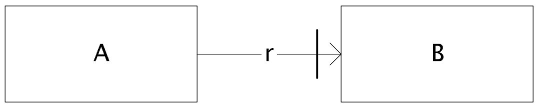
11.9 Mutability

So far, all the features of the object model that we have described constrain individual states. Mutability constraints describe how states may change. To show that a multiplicity constraint is violated, we only need to show a single state, but to show that a mutability constraint is violated, we need to show two states, representing the state before and after a global state change.

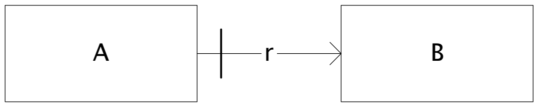
Mutability constraints can be applied to both sets and relations. A stripe on both sides of a box indicates that the set is *fixed*: it always denotes the same set of elements. The root in the file system is fixed, which indicates that we cannot change which object is the root, or destroy the root or create another one. Sectors and centers are fixed in the air-traffic control example, which means that we are taking the structure of airspace to be fixed.

A stripe on the left side of a box indicates that the set is *static*. A static set represents a static classification of elements. An element cannot belong to the set at one point and not belong at the next, or vice versa. Note that this does not mean that the set cannot change, since elements can be created and destroyed. *Dir*, *File* and *Link* are static in the file system, because an object cannot change its classification. A file cannot become a directory, for example. But new files and directories can be created, and objects can be deleted from the file system. Likewise, in the Java model, a value cannot be a primitive value at one time and an object reference at another.

The mutability of a relation is marked with hatch marks on either end. A relation with the hatch mark on the right end, or target end



is said to be *right static* or *target static*. The above diagram says that the set of B's associated with a given member of A by the relation *r* cannot change. Conversely, a hatch mark on the other end



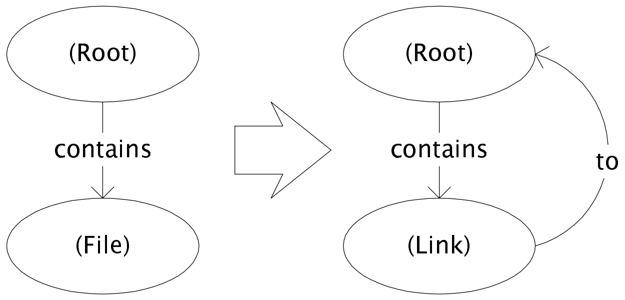
makes the relation *left static* or *source static*. The above diagram says that the set of A's that map to a given B by the relation *r* cannot change.

In the file system model, the relation *to* is right-static, since when a link is created, it is bound to a particular object. You can change which links point to an object by creating and deleting links that point to that object (and thus the relation is not left static), but you can't change which object a given link points to. The *contains* relation is neither left nor right static, since objects can be moved freely amongst directories.

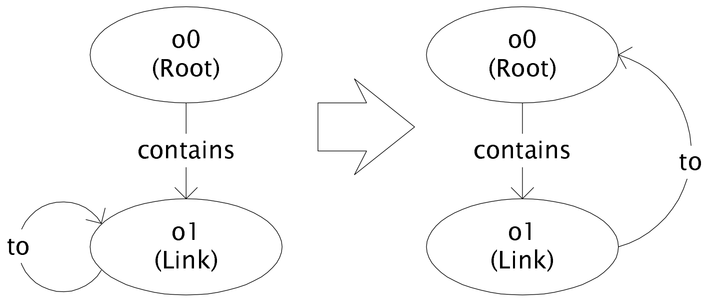
In the air-traffic control model, the relation between centers and sectors is left and right static. Since the sets are fixed, this implies that the relation is fixed too.

In the Java model, the mutability marking on *dtype* says that the declared type of a variable cannot change. The mark`ng on otype says that the type of an object cannot change either; objects cannot migrate between classes. An object cannot change what fields it has, and the name of a field cannot change.

The semantics of mutability is explained in terms of *pairs* of configurations: mutability constraints say which state changes are allowed. So the file system example admits



in which a file is deleted and a link is created, but not



in which the a link's target is altered.

These notions of mutability are powerful but rather abstract and subtle. One common confusion is worth noting. Marking a set as static has nothing to do with whether the class that implements it is mutable or not. (Nevertheless, it is true that if a set is implemented as a class, and its outgoing relations are implemented as fields, then it may be immutable if all the outgoing relations are right-static.) Static classifications are what programmers are familiar with in languages such as Java; a dynamic classification has to be implemented outside the class mechanism, eg with boolean fields.

It may help to see a more formal definition of mutability. Suppose the object model declares a set A to be a subset of a set B. Let's use the names A' and B' to refer to the values these sets have after a change of global state. Then, if A is fixed, we know that the state change must satisify the constraint

$$A' = A$$

If A is static though, it need only satisfy

$$\mathit{all}\ b \in B \cap B' \mid b \in A \Leftrightarrow b \in A'$$

That is, for every *b* that belongs to the set B before and after the state change, *b* either remains inside A or outside A, but it cannot be reclassified.

Now suppose we have a relation *r* from A to B that changes to *r'*. The relation is right-static when for any *r'* that *r* may change to

$$\mathit{all}\ a \in A \cap A' \mid a.r = a.r'$$

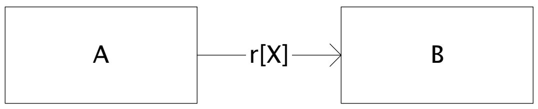
that is, when for an *a* that belongs to A before and after, the set of elements of B it maps to is unchanged. The relation is left-static when

$$\mathit{all}\ b \in B \cap B' \mid b.\sim r = b.\sim r'$$

that is, when for a *b* that belongs to B before and after, the set of elements of A that map to it is unchanged. In these last two formulas, I have used the notation *x.r* to mean the set of elements that *r* maps *x* to, and *~r* to denote the transpose (mirror image) of *r*.

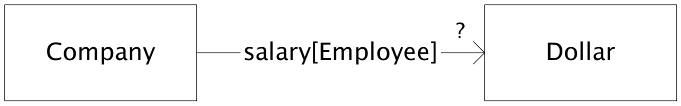
11.10Indexed Relations

Sometimes we want to relate more than just pairs of elements. The label *r[X]* in the following diagram

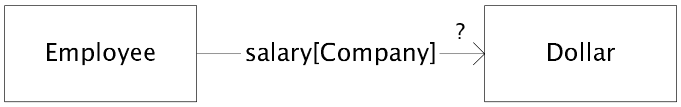


declares an *indexed relation. X* is an indexing set; for each element *x* of this set; *r[x]* denotes a relation.

In an employment database, for example, we might want to describe the relationship between employees, the companies they work for, and their salaries. Such a relationship cannot be described with one binary relation, since it involves three entities. An employee might work for several companies; we want to allow a distinct salary for each. Using an indexed relation, we could show the relationship like this:

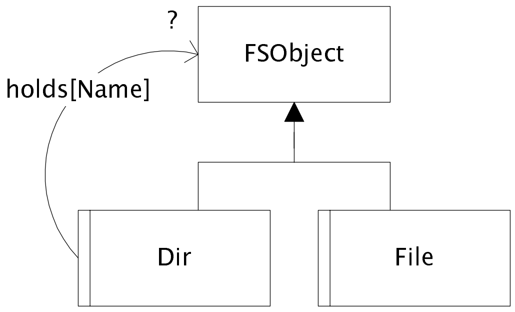


in which for each employee *e, salary[e]* maps a company to the salary it pays *e*, or alternatively like this:



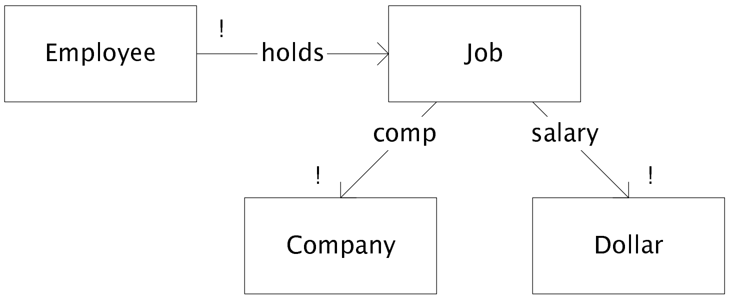
in which for each company *c, salary[c]* maps an employee to the salary she earns at *c.*

In our file system example,instead of using the contains relation to map directories to their contents, we could declare a relation *holds* indexed on names, and then use a multiplicity constraint to say that a directory holds at most one object with a given name:

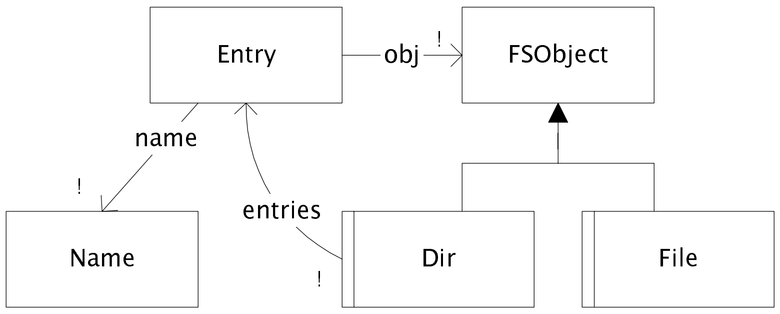


For some name *n*, the relation *holds[n]* maps each directory to the objects that it holds that have the name *n*.

A three-way relationship can also be modelled by introducing an additional set. If we introduce the notion of a *Job*, for example, we can model the employment problem like this:

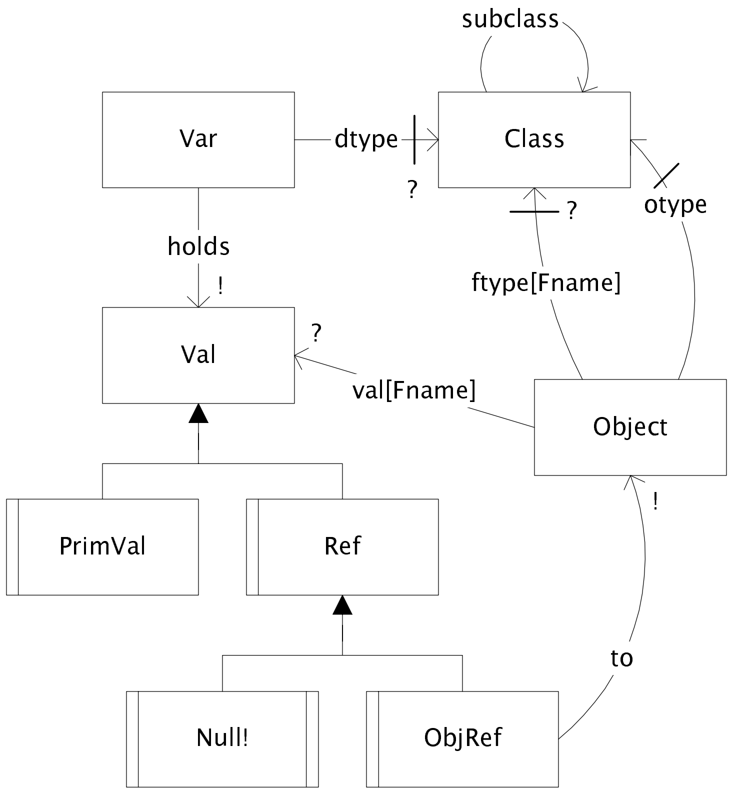


Similarly, in our file system example, we could introduce a notion of a directory entry:



Adding a new set is useful when you might want to hand other relations off the set, or to classify. For example, we might classify *Job* into *Temporary* and *Fulltime*, and we might associate positions with jobs. Using the indexed relation approach has the advantage that it allows tighter constraints to be expressed graphically. In the file system example, the multiplicity shows that there is at most one object with a given name in a directory; the version that uses the *Entry* set cannot express this (except as an additional, textual constraint). A good test is whether the set corresponds naturally to something in the problem domain; *Job* is more plausible than *Entry* in this respect.

Using indexed relations, our Java model could be drawn instead like this:



Note that the *Field* set has gone. Instead, there is an indexed relation from *Object* to *Val*, indexed on *Fname*. For each fieldname *fn*, *val[fn]* maps an object o to a value *v* if *o* has a field with the name *fn* that has a value *v*. The same kind of indexed relation is used to associate types with object fields. Note that we had to change to the multiplicities too, since not every object has every field. The multiplicities on an arrow labelled with an indexed relation apply to every relation in the collection. So the ? on the target end of *val*[*Fname*], for example, says that for every field name *fn*, *val[fn]* maps each object to zero or one values.

11.11 Textual Constraints

Not all constraints can be expressed graphically in an object model. Additional constraints can be expressed textually, as informal comments accompanying the model. Here are some examples of constraints we might want to express:

- In the file system model, that the root directory is not contained by another directory; that no directory contains itself, directly or indirectly; that links do not point to themselves, directly or indirectly; that all file system objects are reachable from the root by the contains relation, and so on.
- In the air-traffic control model, that if an aircraft is flying in a sector, then it is being controlled by a controller that covers that sector; that controllers only cover sectors contained by centers they work for.
- In the Java model, we can express the fundamental property guaranteed by the Java type system as an object model constraint: that if a variable (or field) holds a reference to an object, the type of the object is the same class or a subclass (directly or indirectly) of the declared type of the variable or field.

11.12Summary

Object models a lightweight but precise notation for describing complex configurations. Such configurations lie at the heart of most software systems, and they play a role from early specification through to implementation. The same object modelling notation can be used very abstractly to capture the essence of a problem, and much more concretely, as we have seen before, to describe the structure of objects in a running program.

Lecture 12: Problem Analysis

In this lecture, we'll see how object modelling is used to analyze a problem. The problem we'll consider is representative of many software development problems: apparently simple at first, but actually rather complex, and if not clarified early on, likely to derail the entire development. Of course, if the problem is poorly understood, the resulting implementation will do the wrong thing. But that's not the only danger.

In practice, it's almost impossible to recover from bad early decisions. If the system is built on shaky foundations, it becomes painfully difficult to go back and fix them. Confusion about the problem tends to lead to spurious complexity spread throughout the system. In contrast, if you analyze the problem well, you can eliminate spurious complexities, and invent robust and flexible abstractions on which to base the subsequent design.

Another danger of failing to analyze the problem properly is that the resulting system appears to the user to have no coherent underlying model. We've all been frustrated by programs that appear to be flexible and powerful, but turn out to be a mass of special cases and ad hoc extensions, whose behaviour is impossible to predict. Such programs are unusable.

Finally, for critical software -- such as infrastructural systems (air traffic control, energy distribution, communications, etc) -- if there is no clear articulation of the problem, there can be no way to assess the implementation rigorously. The first step in building software that behaves correctly is to have a characterization of what 'correct' means.

12.1 True Confessions

I picked this problem because of a continuing frustration with my email client. Like several modern email clients, instead of just having a table that maps aliases to sets of email addresses, it provides an elaborate address book with aliases, nicknames, full names, multiple email addresses for an individual, groups distinct from individuals, etc. Address book entries can be constructed automatically from messages, and fields entered when composing a message are automatically completed. But these features behave in a largely unpredictable way and lead to frequent failures.

So I decided that I would figure out what an email client *should* do: what features it should have in this respect, and how it should behave. In analyzing the problem a bit further, it rapidly became clear that the problem is actually tantalizingly difficult. I suspect that the designers of my email client have not paid much attention to these issues, and probably designed it from the user interface inwards. But I now at least have more sympathy for the problems they face, and I am skeptical that there is any very simple solution to the problem.

12.2 The Problem

Here, in short, is the problem:

· *How should the recipient of an email message be named?*

Standard email clients offer many options here. You can indicate a recipient by typing an email address explicitly; you can give a nick name; you can reply to the sender of a previous message; you can type a person's name and hope that a match will be found in an address book; and so on.

What I'd like to do is illustrate, using object modelling, how we can analyze this problem, by looking at the fundamental elements -- messages, aliases, addresses, etc -- and how they are related to one another. We'll expand the problem slightly to include also about how we associate names with the senders of incoming messages.

12.3 The Game Plan

Our aim is to not just to end up with an object model, but to illuminate as we construct it the issue involved so that we have a better understanding at the end than we had at the outset. Of course, our intent will be that the model embodies this understanding, but it won't embody it entirely; we will have discovered, for example, that some apparently plausible notions are not viable, or that certain complexities arise but should be ignored because the cost of addressing them is too high.

Our strategy has the following steps:
- Identify and define the key domains;
- Consider classifications of the domain elements that might result in introducing new sets;
- Analyze the relationships between the sets;
- Investigate multiplicity constraints;
- Investigate mutability constraints.

The strategy will be applied iteratively until it converges. At any point, we may invent new abstractions (and represent them as sets), or may change how we model the basic elements of the problem.

12.4 Domains

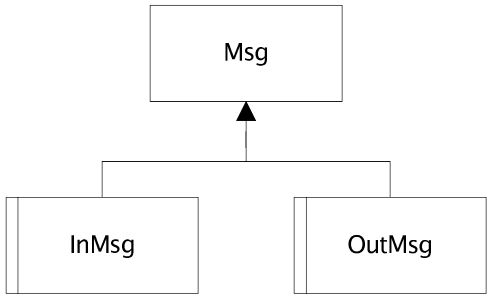
First, we list candidate domains. Some obvious domains for this problem are
- *Msg*, the set of messages;
- *Person*, the set of persons to and from which messages are sent;
- *Name*, the set of names of persons;
- *Address*, the set of email addresses;
- *Alias*, the set of nicknames of aliases we choose as shorthands.

We'll see as we develop our model that additional domains are useful. At this stage we might consider some domains but reject them. We might wonder whether we need an AddressBook domain, for example. As a general rule of thumb, if a domain has only one element, it's not worth including. Since we as yet have no good reason to introduce the (rather implementation-oriented) notion of an address book, we won't introduce it as a domain.

12.5 Classification

Next, we ask whether we might refine our classification of objects. We look at each domain, and ask whether there are any obvious subsets.

It seems likely that we'll want to distinguish incoming and outgoing messages. So let's introduce subsets *InMsg* and *OutMsg* of *Msg*:



Note that I've chosen to make the classification exhaustive, and to make the subsets static. This is a bit fishy, since it rules out the possibility of there being messages that can't be immediately classified as incoming or outgoing. When might this happen? When messages are imported from a file, for example, or when you send a message to yourself. We decide not to address this complication, so we record it (so we don't forget about it), but carry on without loosening the model.

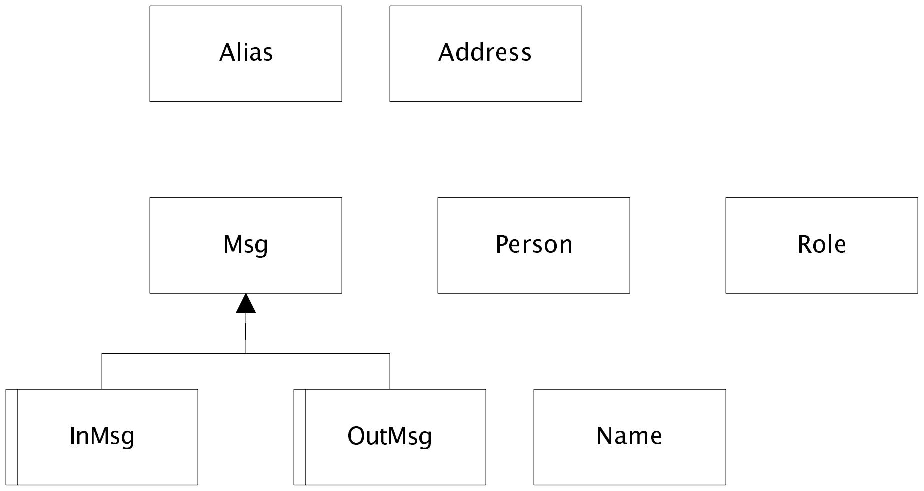
Should we classify elements of *Person*? Might we treat family members and close friends differently from business colleagues? Probably not. We do, however, treat people in certain official positions differently. For example, we might have messages that we send to our dean, teaching assistant, system administrator, congressional representative, etc, that are not personal, and are sent only to that person in his or her official capacity. It seems likely that these kinds of targets will have different properties. We might expect their email address to change frequently, for example; and we expect their names to change too. One could even imagine a mechanism whereby changes are made automatically (for example, an alias *lab-assistant* that is resolved in the client depending on the time of day to a different person).

On further reflection, we realize that this notion is not in fact a subset of *Person* at all. There is a notion of a role, and people fulfill roles. So we take a step back and introduce a new domain *Role*.

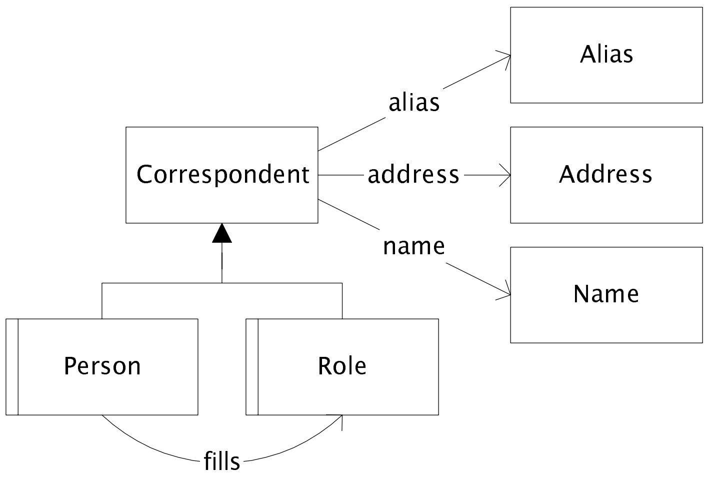
Can *Address* be refined? Should we distinguish local addresses from addresses at other institutions? And *Alias*? Does it have refinements? We'll assume not.

12.6 Relations

Now things start getting interesting. We look at the sets we have and we wonder for each pair of sets whether there might be a relation between them. In skeletal form, our object model so far looks like this:



Some relations are easy to see. A person has a name and an email address for example, so we can introduce a relation *name* from *Person* to *Name*, and a relation *address* from *Person* to *Address*. A person fills a role, so we introduce a relation *fills* from *Person* to *Role*. Roles have names also, so we consider adding a relation *rname* from *Role* to *Name*. We notice the similarity in the treatment of persons and roles, and we wonder whether we might treat them as subsets of the same set. This seems reasonable, so we introduce a new domain *Correspondent* and we make *Person* and *Role* its subsets. We posit that aliases are also associated with correspondents. The right hand side of our model now looks like this:

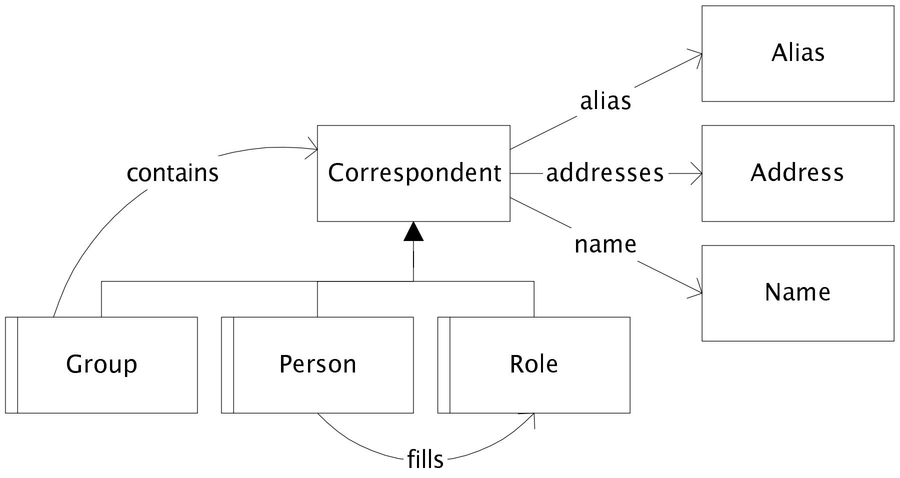


How is the set *Msg* related to these sets?

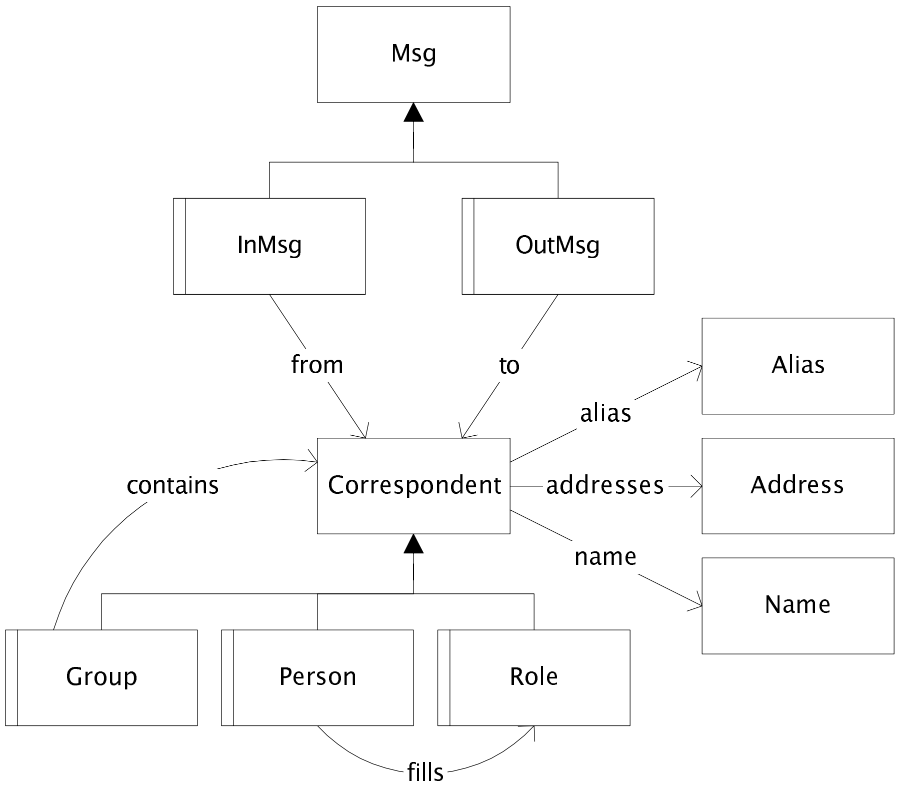
We remember that people are associated with messages in many different ways: as senders, receivers, cc's and so on. In the context of this lecture,we'll ignore all but senders and receivers. Thinking about the relationships between messages and persons, we remember that there are groups: both those that appear to the client as one account (ie, where the group is defined externally, on some other machine), and those that appear as a collection of persons or addresses. Let's call these external and internal groups.

Can an external group be treated as a Correspondent? If so, is it a Person or a Role? Given the bitter experiences of messages intended for individuals being sent to huge mailing lists, it seems that it would be wise to treat such a group as different from a Person or a Role; our email client could then support useful functions such as blocking listserv messages, or warning before sending to a mailing list. On the other hand, external groups do have addresses and names, and it seems fine to give them aliases. So we might accommodate them by adding a new subset to Correspondent. But we decide that we need more evidence that this complication is worthwhile before doing that, so we record the issue, and decide that for now we'll treat an external groups as if it were a role filled by some unknown persons.

How about internal groups? We'd like to be able to form groups which themselves contain groups. For example, I'd I may want to define groups for students I advise in different years, but also have agroup for all students I advise. It's tempting then to introduce a set, *Group* say, that's another subset of Correspondent, with a *relation* contains from *Group* to *Correspondent*. In design pattern jargon, this forms a 'composite': we can make trees of arbitrary depth by including groups in groups. But this makes us slightly uncomfortable, since Persons and Groups are related to addresses in very different ways. Recognizing this, we decide to loosen the notion inherent in the *address* relation; we rename it *addresses*, and take it to map a group to a *computed* set of addresses of all group members. The rig-ght-hand side of our model now looks like this:



Now finally we get back to the question of relating messages to these sets. Let's bravely claim that each incoming message is from a correspondent; each outgoing message is to some correspondents. Then our object model takes this form:



This decision implies that we will somehow need to determine for an incoming message who its sender is, based on the information we have about correspondents and their addresses.

In considering the relations amongst sets, note that we ended up introducing a new abstraction, *Correspondent*, and a new classification, *Group*.

12.7 Multiplicities

We now consider adding multiplicity constraints to our object model. We immediately hit some tricky questions:

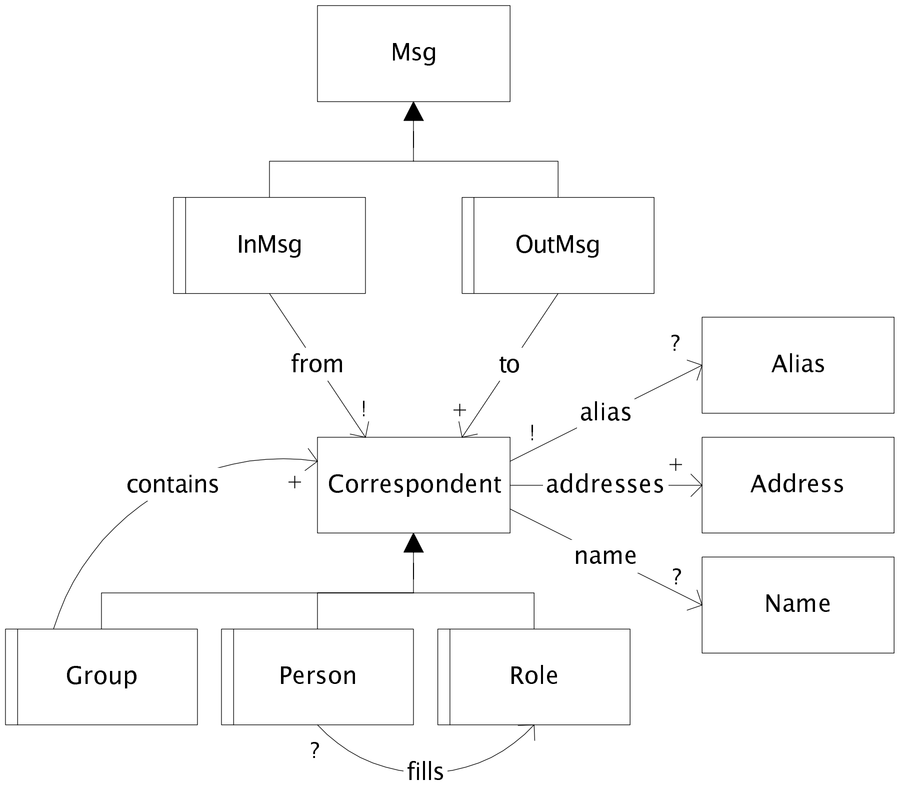
- Can a Group contain no Correspondents? We'll say yes, to eliminate the kinds of confusion that will arise from sending a message to nobody.
- Must every Correspondent have a Name? An Alias? Sometimes we reply to a message without even knowing the name of the person who sent it. Clearly, then, having associated messages with correspondents, we'll have to allow correspondents to be anonymous. It seems reasonable to use a role to represent an anonymous correspondent. We might record a textual constraint that every Group and Person must have a Name, but a Role need not.
- Can a Correspondent have several Addresses? We've already noted that a group will general have several addresses associated with it (computed fromthe addresses of its member correspondents). But we also know that some people have multiple email addresses, and it would be a horrible hack

to treat such people as groups. In this respect, a Person is really different from other Correspondents; w e might want to record a primary email address, and backup addresses to use if the primary fails. Lacking evidence that such a feature is necessary, however, we record the issue, and carry on under the assumption that a Person has exactly one Address, which we record as a textual constraint.

Because of the use of roles to model anonymous correspondents, we will certainly not want to require each role to be filled by a person. There seems to be no reason to allow a role to be filled by more than one person, so we record that multiplicity. In pondering this question, we stumble across an interesting issue. A role defined in isolation may have an address given for it by the user; when a role is filled by a person, however, that person's address will determine the address of the role. Or will it? Such a person may have a separate address in their capacity filling that role. This is a tricky issue, and we'll leave it aside for now.

Looking at the other ends of the arrows, we can ask about identification. Does a name uniqely identify a correspondent? Perhaps not, but an alias certainly should.

Our model looks like this:



12.8 Mutability

Finally, we consider changes that might occur by examining each set and relation and considering its mutability constraints. We've already marked all the subsets as static, so we'll look only at the relations. Most are straightforward. Clearly, we'll want to be able to change the alias, address and name associated with a person or role, change the contents of a group, and change how persons fill roles.

Much trickier is the question of whether can change the correspondents associated with a message. What if a correspondent for an incoming message was found by finding a person whose address matches the address in the header of the message, and we now change the person's address? Should the message be reclassified? What if we delete a correspondent that a message was sent to?

Note that we are really doing design here, albeit of a rather abstract sort. One solution to these dilemmas might be to allow reclassifications, and to make use of anonymous roles when no matching correspondent is found (as we would have to for an incoming message from an unrecognized address anyway). A deficiency of this scheme is that if correspondents change their email addresses, our program will lose the association between old messages and their correspondents. It might, therefore, be sensible to prevent reclassifications that would turn the correspondent of a message from a person to an anonymous role, for example.

By doing this, we can avoid imposing any mutability constraints on the relations, and the object model will be unchanged. This will allow the user to make arbitrary changes to names, addresses, etc, and allows the program to do fancy reclassification of messages.

By considering mutability, we've forced ourselves to address some of the hardest issues involved in this problem. The lack of mutability constraints on the relations indicates that we have been unsuccessful in making these issues go away.

12.9 Summary

The particular decisions that I made in this analysis are not that important. You may disagree with them. But what's important is the *process* by which I went about analyzing the problem incrementally, and how I used the *notation*. The object modelling notation is more than just a format for recording-final decisions; it gives you a way to express fragmentary ideas, and stimulates you to think about subtle aspects of the problem.

In the process of developing our object model, we engaged in two different activities. One was understanding the problem domain as it already exists: we took into account, for example, the notions of groups and roles, which represent how people communicate with each other. At the same time, we invented new abstractions, usually by generalization: the notion of a correspondent, for example, was introduced to allow a uniform treatment of groups, persons and roles. Generalizations are dangerous (as one wit observed, they're generally wrong), so we must be aware of the risks they introduce along with their benefits. In the case of correspondents, the complexity we introduced was that some correspondents have addresses defined by the user; others have addresses that are computed from the addresses of other correspondents. In a real development, we would analyze this issue more deeply.

Lecture 18: Design Strategy

This lecture puts together some of the ideas we have discussed in previous lectures: object models of problems and code, module dependency diagrams, and design patterns. Its aim is to give you some general advice on how to go about the process of software design. I'll explain some criteria for evaluating designs, and give a handful of heuristics that help in finding a design to solve a given problem.

18.1 Process Overview & Testing

The development process has the following major steps:
· Problem analysis: results in an object model and a list of operations.
· Design: results in a code object model, module dependency diagram and module specs.
· Implementation: results in executable code.

Testing should ideally be performed throughout the development, so that errors are found as soon as possible. In a famous study of projects at TRW and IBM, Barry Boehm found that the cost of fixing an error can rise by a factor as great as 1000 when it is found later rather than earlier. We've only used the term 'testing' to describe evaluation of code, but similar techniques can be applied to problem descriptions and designs if they are recorded in a notation that has a semantics. (In my research group, we've developed an analysis technique for object models). In your work in 6170, you'll have to rely on careful reviewing and manual exercise of scenarios to evaluate your problem descriptions and designs.

As far as testing implementations goes, your goal should be to test as early as possible. Extreme programming (XP), an approach that is currently very popular, advocates that you write tests before you've even written the code to be tested. This is a very good idea, because it means that test selection is less likely to suffer from the same conceptual errors that tests are intended to find in the first place. It also encourages you to think about specs up front. But it is ambitious, and not always feasible.

Instead of testing your code in an ad hoc way, you should build a systematic test bed that requires no user interaction to execute and validate. This will pay dividends. When you make changes to code, you'll be able to quickly discover fresh bugs that you've introduced by rerunning these 'regression tests'. Make liberal use of runtime assertions, and check representation invariants.

18.2 Problem Analysis

The main result of problem analysis is an object model that describes the fundamental entities of the problem and their relationships to one another. (In the course text, the term 'data model' is used for this.) You should write short descriptions for each of the sets and relations in the object model, explaining what they mean. Even if it's clear to you at the time, it's easy to forget later what a term meant. Moreover, when you write a description down, you often find it's not as straightforward as you thought. My research group is working on the design of a new air-traffic control component; we've discovered that in our object model the term *Flight* is a rather tricky one, and getting it right clearly matters.

It's helpful also to write a list of the primary operations that the system will provide. This will give you a grip on the overall functionality, and allow you to check that the object model is sufficient to support the operations. For example, a program for tracking the value of stocks may have operations to create and delete portfolios, add stocks to portfolios, update the price of a stock, etc.

18.3 Design Properties

The main result of the design step is a code object model showing how the system state is implemented, and a module dependency diagram that shows how the system is divided into modules and how they relate to one another. For tricky modules, you will also want to have drafted module specifications before you start to code.

What makes a good design? There is of course no simple and objective way to determine whether one design is better than another. But there are some key properties that can be used to measure the quality of the design. Ideally, we'd like a design to do well on all measures; in practice, it's often necessary to trade one for another.

The properties are:

- *Extensibility.* The design must be able to support new functions. A system that is perfect in all other respects, but not amenable to the slightest change or enhancement, is useless. Even if there is no demand for additional features, there are still likely to be changes in the problem domain that will require changes to the program.
- *Reliability.* The delivered system must behave reliably. That doesn't only mean not crashing or eating data; it must perform its functions correctly, and as anticipated by the user. (This means by the way that it's not good enough for the system to meet an obscure specification: it must meet one that is readily understood by the user, so she can predict how it will behave.) For a distributed system, availability is important. For real-time systems, timing is important: usually this means not that the system is fast, but that it completes its tasks in predictable times. How reliability is judged varies greatly from system to system. A browser's failure to render an image precisely is less serious than the same failure in a desktop publishing program. Telephone switches are required to meet extraordinarily high standards of availability, but may misroute calls occasionally. Small timing delays may not matter much for an email client, but they won't do in a nuclear reactor controller.
- *Efficiency.* The system's consumption of resources must be reasonable. Again, this depends of course on the context. An application that runs on a cell phone can't assume the same availability of memory as one that runs on a desktop machine. The most concrete resources are the time and space consumed by the running program. But remember that the time taken by the development can be just as important (as Microsoft has demonstrated), as well as another resource not to be ignored -- money. A design that can be implemented more economically may be preferable to one that does better on other metrics but would be more expensive.

18.4 Overview of Strategy

How are these desirable properties obtained?

18.4.1 Extensibility

- *Object model sufficiency.* The problem object model has to capture enough of the problem. A common obstacle to extending a system is that there is no place for the new function to be added, because its notions aren't expressed anywhere in the code. An example of this can be seen in Microsoft Word. Word was designed on the assumption that paragraphs were the key document structuring notion. There was no notion of text flows (physical spaces in the document through which text is threaded), nor of any kind of hierarchical structure. As a result, Word doesn't smoothly support division into sections, and it can't place figures. Its important to be very careful not to optimize the problem object model and eliminate substructure that appears to be unnecessary. Don't introduce an abstraction as a replacement for more concrete notions unless you're really sure that it's well founded. As the motto goes, generalizations are generally wrong.
- *Locality and decoupling*. Even if the code does end up embodying enough notions onwhich to hang new functionality, it may be hard to make the change you need to make without altering code all over the system. To avoid this, the design must exhibit *locality*: separate concerns should, to the greatest extent possible, be separated into distinct regions of the code. And modules must be *decoupled* from one another as much as possible so that a change doesn't cascade. We saw examples of decoupling in the lecture on name spaces, and more recently in the lectures on design patterns (in *Observer*, for example). These properties can be judged most easily in the module dependency diagram: this is why we construct it. Module specifications are also important for achieving locality: a specification should be *coherent*, with a clearly bounded collection of behaviours (without special ad hoc features), and a clear division of responsibility amongst methods, so that the methods are largely orthogonal to one another.

18.4.2 Reliability

- *Careful modelling*. Reliability cannot be easily worked into an existing system. The key to making reliable software is to develop it carefully, with careful modelling along the way. Most serious problems in critical systems arise not from bugs in code but from errors in problem analysis: the implementor simply never considered some property of the environment in which the system is placed. Example: Airbus failure at Warsaw airport.
- *Review,analysis, and testing*. However careful you are, you will make errors. So in any development, you have to decide in advance how you will mitigate the errors that you will inevitably make. In practice, peer review is one of the most cost-effective methods for finding errors in any software artifact,whether model, specification or code. So far, you've only been able to exploit this with your TA and the LA's; in your final project, you should take full advantage of working in a team to review each other's work. It'll save you a lot of time in the long run.

  More focused analyses and testing can find more subtle errors missed by peer review. Some useful and easy analyses you can apply are simply to check that your models are consistent: does your code object model support all the states of the problem object model? do the multiplicities and mutabilities match appropriately? does the module dependency diagram account for all the edges in the object model? You can also check you code against the models. The Womble tool, available at http://sdg.lcs.mit.edu, automatically constructs object models from bytecode. We have found many bugs in our code by examining extracted models and comparing them to the intended models. You should check the crucial properties of your object model in the code by asking youself how you know that the properties are maintained. For example, suppose your model asserts that a vector is never shared by two bank account objects. You should be able to make an argument for why the code ensures this. Whenever you have a constraint in your object model that wasn't expressible graphically, it's especially worth checking, as it is likely to involve relationships that cross object boundaries.

18.4.3 Efficiency

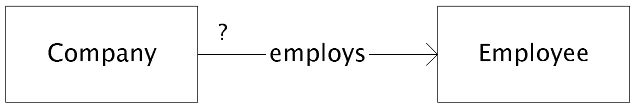
- *Object Model.* Your choice of code object model is crucial, because it's hard to change. So you should consider critical performance targets early on in the design. We'll look later in the lecture at some sample transformations that you can apply to the object model to improve efficiency.
- *Avoid bias*. When you develop your problem object model, you should exclude any implementation concerns. A problem model that contains implementation details is said to be *biased*, since it favours one implementation over another. The result is that you have premuaturely cut down the space of possible implementations, perhaps ruling out the most efficient one.
- *Optimization.* Optimization is misnamed; it invariably means that performance gets better but other qualities (such as clarity of structure) get worse. And if you don't go about optimization carefully, you're likely to end up with a system that is worse in every respect. Before you make a change to improve performance, make sure that you have enough evidence that the change is likely to have a dramatic effect. In general, you should resist the temptation to optimize, and put your efforts into making your design clean and simple. Such designs are often the most efficient anyway; if they're not, they are the easiest to modify.
- *Choice of reps.* Don't waste time on gaining small improvements in performance,but focus instead on the kinds of dramatic improvement that can be gained by choosing a different rep for an abstract type, for example, which may change an operation from linear to constant time. Many of you have seen this in your MapQuick project: if you chose a representation for graphs that required time proportional to the size of the entire graph to obtain a node's neighbours, search is completely infeasible. Remember also that sharing can have a dramatic effect, so consider using immutable types and having objects share substructure. In MapQuick, *Route* is an immutable type; if you implement it with sharing, each extension of the route by one node during search requires allocation of only a single node, rather than an entire copy of the route.

Above all, remember to aim for *simplicity.* Don't underestimate how easy it is to become buried under a mass of complexity, unable to achieve any of these properties. It makes a lot of sense to design and build the simplest, minimal system first, and only then to start adding features.

18.5 Object Model Transformations

In problem and code object models, we've seen two very different uses of the same notation. How can an object model describe a problem and also describe an implementation? To answer this question, it's helpful to think of interpreting an object model in two steps. In the first step, we interpret the model in terms of abstract sets and relations. In the second step, we map these sets and relations either to the entities and relationships of the problem, or to the objects and fields of the implementation.

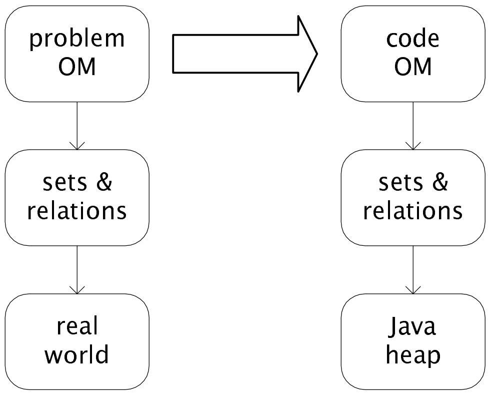
For example, suppose we have an object model with a relation *employs* from *Company* to *Employee*.



Mathematically, we view it as declaring two sets and a relation between them. The multiplicity constraint says that each *employee* is mapped to under the *employs* relation by at most one *company.* To interpret this as a problem object model, we view the set *Company* as a set of companies in the real world, and *Employee* as a set of persons who are employed. The relation *employs* relates *c* and *e* if the actual company *c* employs the person *e*.

To interpret this as a code object model,we view the set *Company* as a set of heap-allocated objects of the class *Company*, and *Employee* as a set of heap-allocated objects of the class *Employee*. The relation *employs* becomes a specification field, associating *c* and *e* if the object *c* holds a reference to a collection (hidden in the representation of *Company*) that contains the reference *e*.

Our strategy is to start with a problem object model, and transform it into a code object model. These will generally differ considerably, because what makes a clear description of the problem is not generally what makes a good implementation.

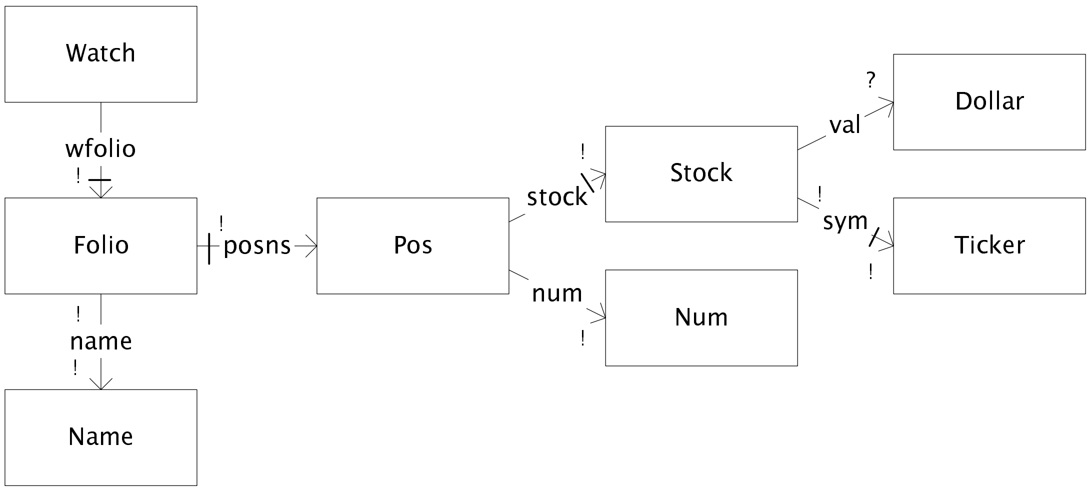


How is this transformation accomplished? One way is to brainstorm and play with different code model fragments until they coalesce. This can be a reasonable way to work. You need to check that the code object model is faithful to the problem object model. It must be capable of representing at least all the information in the states of the problem model, so you can add a relation for example, but you can't remove one.

Another way to go about the transformation is by systematically applying a series of small transformations. Each transformation is chosen from a repertoire of transformations that preserve the information content of the model, so that since each step keeps the model sound, the entire series must also. Nobody has yet figured out a full repertoire of such transformations -- this is a research problem -- but there are a handful we can identify that are the most useful. First let's introduce an example.

18.6 Folio Tracker Example

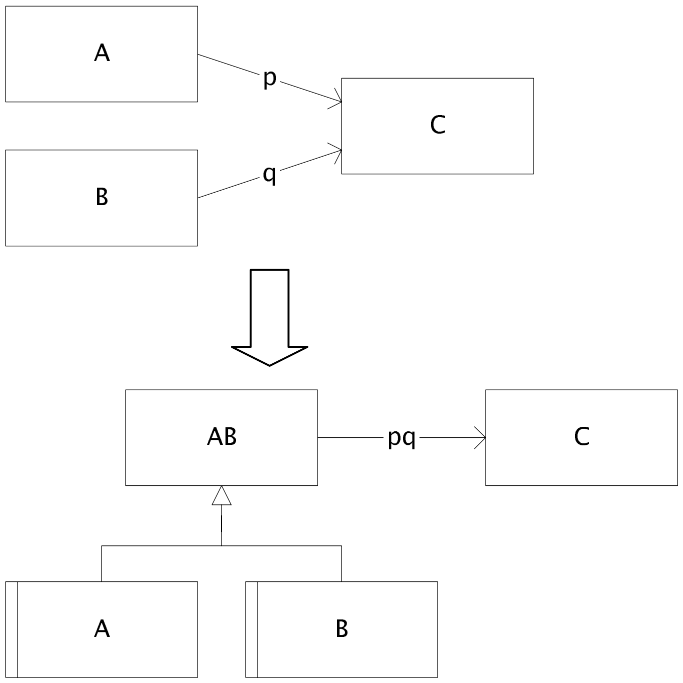
Consider designing a program for tracking a portfolio of stocks. The object model describes the elements of the problem. *Folio* is the set of portfolios, each with a *Name*, containing a set of positions *Pos*. Each position is for a particular *Stock*, of which some number are held. A stock may have a value (if a quote has been recently obtained), and has a ticker symbol that does not change. Ticker symbols uniquely identify stocks. A *Watch* can be placed on a portfolio; this causes information about the portfolio to be displayed when certain changes to the portfolio occur.



18.7 Catalog of Transformations

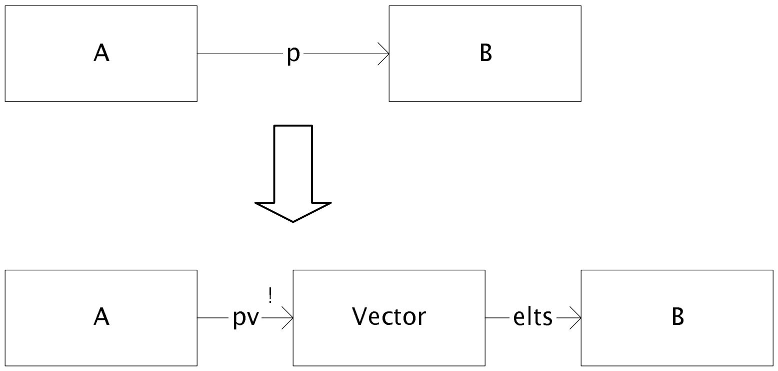
18.7.1 Introducing a Generalization

If *A* and *B* are sets with relations *p* and *q*, of the same multiplicity and mutability, to set *C*, we can introduce a generalization *AB* and replace *p* and *q* by a single relation *pq* from *AB* to *C*. The relation *pq* may not have the same source multiplicity as *p* and *q*.

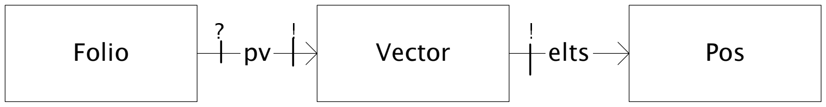


18.7.2 Inserting a Collection

If a relation *r* from *A* to *B* has a target multiplicity that allows more than one element, we can interpose a collection, such as a vector or set between *A* and *B*, and replace *r* by a relation to two relations, one from *A* to the collection, and one from the collection to *B*.



In our Folio Tracker example, we might replace interpose a vector in the relation *posns* between *Folio* and *Pos*. Note the mutability markings; the collection is usually constructed and garbage collected with its container.

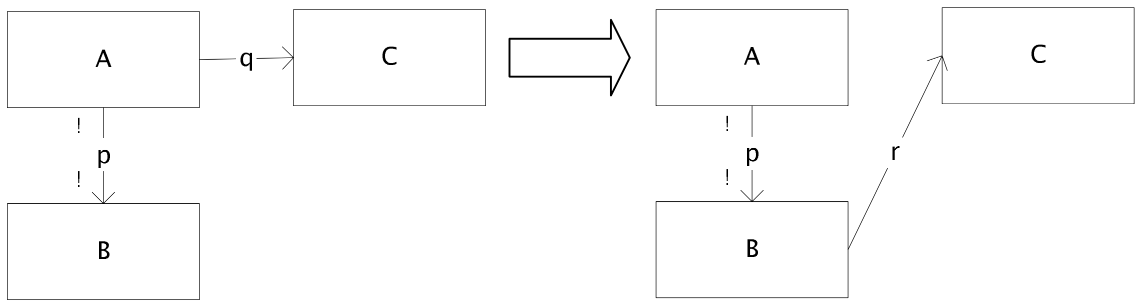


18.7.3 Reversing a relation

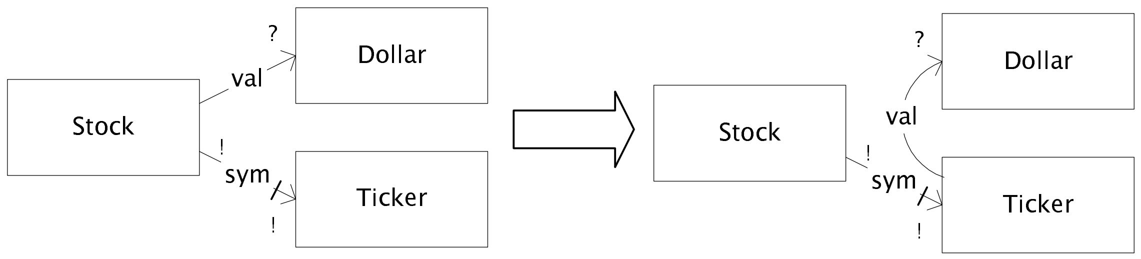
Since the direction of a relation doesn't imply the ability to navigate it in that direction, it is always permissible to reverse it. Eventually, of course, we will interpret relations as fields, so it is common to reverse relations so that they are oriented in the direction of expected navigation. In our example, we might reverse the *name* relation, since we are likely to want to navigate from names to folios, obtaining a relation *folio*, say.

18.7.4 Moving a Relation

Sometimes the target or source of a relation can be moved without loss of information. For example, a relation from *A* to *C* can be replaced by a relation from *B* to *C* if *A* and *B* are in one-to-one correspondence.

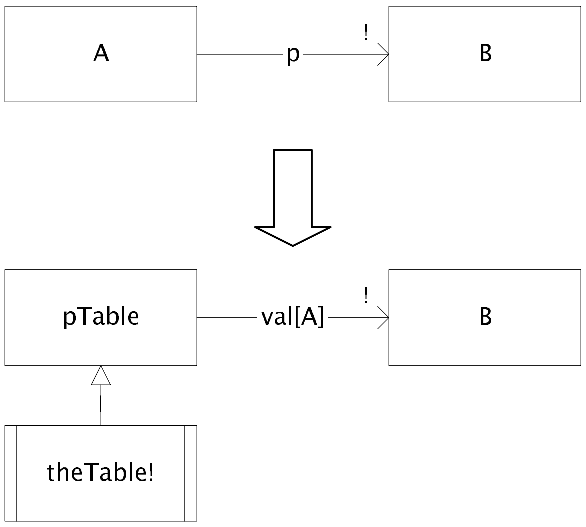


In our example, we can replace the *val* relation between *Stock* and *Dollar* by a relation between *Ticker* and *Dollar*. It's convenient to use the same name for the new relation, although technically it will be a different relation.

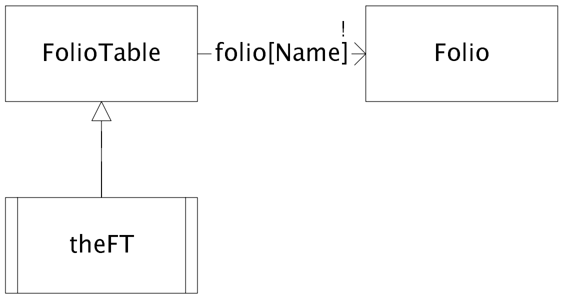


18.7.5 Relation to Table

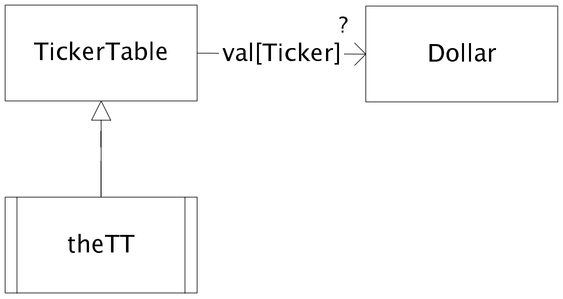
A relation from *A* to *B* with a target multiplicity of *exactly one* or *zero or one* can be replaced by a table. Since only one table is needed, the singleton pattern can be used so that the table can be referenced by a global name. If the relation's target multiplicity is *zero or one*, the table must be able to support mappings to null values.



In FolioTracker, for example, we might convert the relation *folio* to a table to allow folios to be found by a constant-time lookup operation. This gives:



It would make sense to turn the relation *val* from *Ticker* to *Dollar* into a table too, since this will allow the lookup of values for ticker symbols to be encapsulated in an object distinct from the portfolio. In this case, because of the zero-or-one multiplicity, we'll need a table that can store null values.



18.7.6 Adding Redundant State

It is often useful to add redundant state components to an object model. Two common cases are adding the transpose of a relation, and adding the composition of two relations. If *p* maps *A* to *B*, we can add the transpose *q* from *B* to *A*. If *p* maps *A* to *B*, and *q* maps *B* to *C*, we can add the composition *pq* from *A* to *C*.

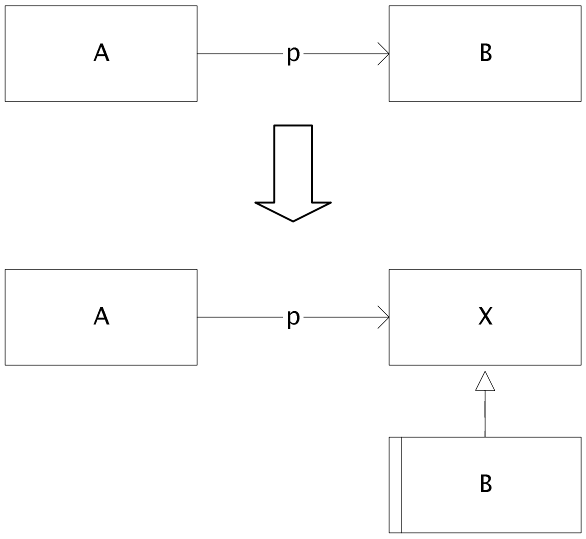
18.7.7 Factoring out Mutable Relations

Suppose a set *A* has outgoing relations *p*, *q* and *r*, of which *p* and *q* are right-static. If implemented directly, the presence of *r* would cause *A* to be mutable. It might therefore be desirable to factor out the relation *r*, eg by using the *Relation to Table* transform, and then implementing *A* as an immutable datatype.

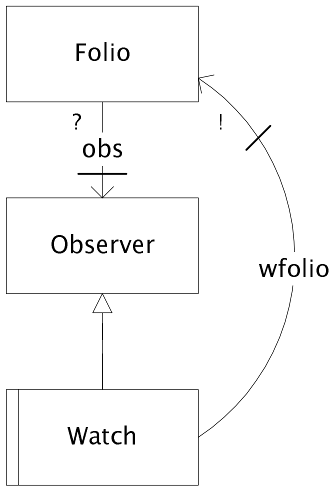
In our example, the factoring out of the *val* relation fits this pattern, since it renders *Stock* immutable. The same idea underlies the *Flyweight* design pattern, by the way.

18.7.8 Interpolating an interface

This transformation replaces the target of a relation *R* between a set *A* and a set *B* with a superset *X* of *B*. Typically, *A* and *B* will become classes and *X* will become an abstract class or interface. This will allow the relation *R* to be extended to map elements of *A* to elements of a new set *C*, by implementing *C* as a subclass of *X*. Since *X* factors out the the shared properties of its subclasses, it will have a simpler specification than *B*; *A*'s dependence on *X* is therefore less of a liability than its prior dependence on *B*. To make up for the loss of communication between *A* and *B*, an additional relation may be added (in a further transformation) from *B* back to *A*.

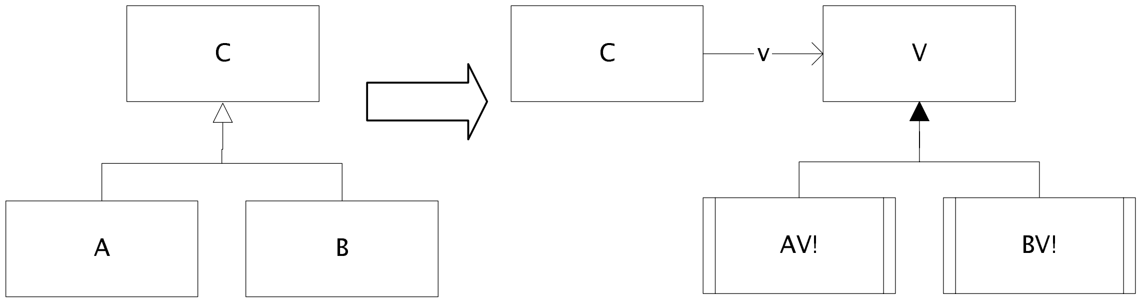


The Observer design pattern is an example of the result of this transformation. In our example, we might make the *Watch* objects observers of the *Folio* objects:



18.7.9 Eliminating Dynamic Sets

A subset that is not static cannot be implemented as a subclass (since objects cannot migrate between classes at runtime). It must therefore be transformed. A classification into subsets can be transformed to a relation from the superset to a set of classifier values

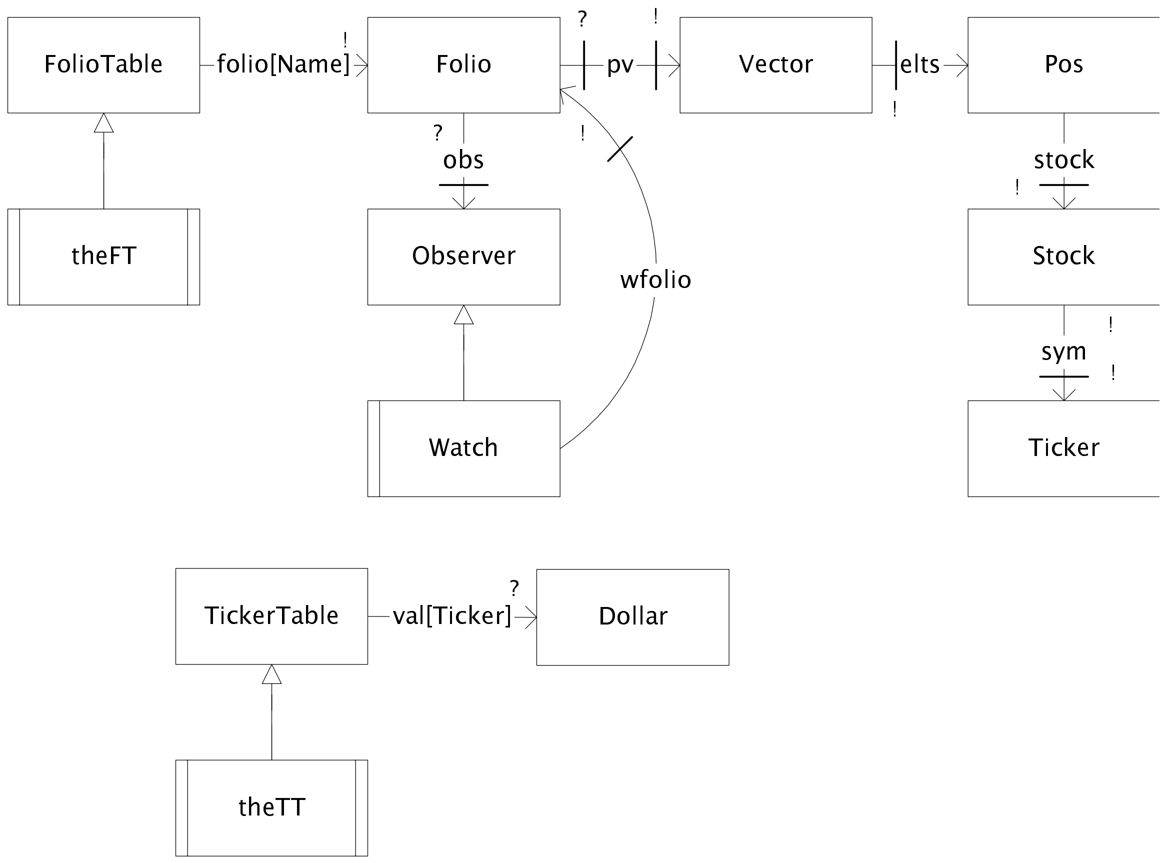


Where there is only one or two dynamic subsets, the classifier values can be primitive boolean values.

The classification can also be transformed to several singleton sets, one for each subset.

18.8 Final OM

For our Folio Tracker example, the result of the sequence of transformations that we have discussed is shown below. At this point, we should check that our model supports the operations the system must perform, and use the scenarios of these operations to construct a module dependency diagram to check that the design is feasible. We will need to add modules for the user interface and whatever mechanism is used to obtain stock quotes. We would also want to add a mechanism for storing folios persistently on disk. For some of this work, we may want to go back and construct a problem object model, but for other parts it will be reasonable to work at the implementation level. For example, if we allow users to name files to store folios in, we will almost certainly need a problem object model. But to resolve issues of how to parse a web page to obtain stock quotes, constructing a problem object model is unlikely to be productive.



18.9 UML and Methods

There are many methods that prescribe a detailed approach to object-oriented development. They tell you what models to produce, and in what order. In an industrial setting, standardizing on a method can help coordinate work across teams. Although you won't learn about any particular method in 6170, the notions you learn in 6170 are the foundation of most methods, so you should be able to pick up any particular method easily. Almost all methods use object models; some also use module dependency diagrams. If you'd like to learn more about methods, I'd recommend Catalysis, Fusion and Syntropy; a google search on these names will direct you to online materials and books.

In the last few years, there's been an attempt to standardize notations. The Object Management Group has adopted the Unified Modeling Language (UML) as a standard notation. It's actually a large collection of notations. It includes an object modelling notation that is similar to ours (but much more complicated).