Lecture 1: Introduction

1.1 What 6170's About

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 you only really learn an idea when you try and use it.

You will learn:
- how to design software: powerful abstraction mechanisms; patterns that have been found to work well in practice; how to represent designs so you can communicate them and critique them
- how to implement in Java
- how to get it right: dependable, flexible software.

Not hacking
- how to be an architect, not just a low-level coder
- how to avoid spending time debugging

1.2 Admin & Policies

Course staff intros:
- Lecturers: Daniel Jackson and Rob Miller
- TAs: you'll meet in review session *next* week
- LAs: you'll meet in clusters
- Hours: see website. Lecturers don't have fixed office hours but happy to talk to students: just send email or drop by.

Materials:
- course text by Liskov; read according to schedule in general info handout
- lecture notes: usually published the day of the lecture
- 'Gang of Four' design patterns book: recommended
- 'Effective Java' by Bloch: recommended
- Java tutorial: see general information handout for details

Recommended texts are really superb; will be good references, and will help you become a good programmer faster. Special deal if you buy as package.

Course organization:

- First half of term: lectures, weekly exercises, reviews, quiz
- Second half of term: team project. More to say on this later.

A change from previous terms: no need to worry now about who will be in your team. Expect to switch TA's at half term.

Reviews:
- Weekly sessions with TAs will be used for *review* of student work
- Initially, TAs will pick fragments of your work to focus on
- Whole section will discuss in a constructive and collaborative way
- Absolutely essential part of course: opportunity to see how ideas from lecture get applied in practice

Learning Java:
- It's up to you, but we try and help
- Use Sun's Java tutorial and do exercises
- Great team of lab assistants on hand in clusters to help you

Collaboration and IP policy:
- see general info
- in short: you can discuss, but written work must be your own; includes spec, design, code, tests, explanations
- you can use public domain code
- in team project, can collaborate on everything

Quizzes:
- two in-class quizzes, focusing on lecture material

Grading:
- 70% individual work = 25% quizzes + 45% problem sets
- 30% final project, all in team get same grade
- 10% participation extra credit
- *no late work will be accepted*

1.3 Why Software Engineering Matters

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 Conspiracy*, Mark Minasi, McGraw Hill, 2000).

Role in infrastructure:

- not just the Internet
- transportation, energy, medicine, finance

Software is becoming pervasive in embedded devices. New cars, for example, have between 10 and 100 processors for managing all kinds of functions from music to braking.

Cost of software:
- Ratio of hardware to software procurement cost approaches zero
- Total cost of ownership: 5 times cost of hardware. Gartner group estimates cost of keeping a PC for 5 years is now $7-14k

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

1.3.1 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'

1.3.2 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, in: *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

You might think that we'd learn from this and such a disaster would never happen again. But…

- International Atomic Energy Agency declared 'radiological emergency' in Panama on 22 May, 2001
- 28 patients overexposed; 8 died, of which 3 as result; 3/4 of surviving 20 expected to develop 'serious complications which in some cases may ultimately prove fatal'
- Experts found radiotherapy equipment 'working properly'; cause of emergency lay with data entry
- If data entered for several shielding blocks in one batch, incorrect dose computed
- FDA, at least, concluded that 'interpretation of beam block data by software' was a factor
- see http://www.fda.gov/cdrh/ocd/panamaradexp.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

The Ariane accident is more typical of most software disasters than the radiotherapy machine accidents. It's quite rare for bugs in the code to be the cause; usually, the problem goes back to the requirements analysis, in this case a failure to articulate and evaluate important environmental assumptions.

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

The London Ambulance disaster was really a managerial one. The managers who produced the software were naive, and accepted a bid from an unknown company that was many times lower than bids from reputable companies. And they made the terrible mistake of trying to go online abruptly, without running the new and old systems together for a while.

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

1.3.3 Software Quality

One measure: bugs/kloc
- measured after delivery
- industry average is about 10
- high quality: 1 or less

Praxis CDIS system (1993)
- UK air-traffic control system for terminal area
- used precise spec language, very similar to the object models we'll learn
- no increase in net cost
- much lower bug rate: about 0.75 defects/kloc
- even offered warranty to client!

Of course, quality isn't just about bugs. You can test software and eliminate most of the bugs that cause it crash, but end up with a program that's impossible to use and fails much of the time to do what you expect, because it has so many special cases. To address this problem, you need to build quality in from the start.

1.4 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 (and it's cheap!)
- can't add quality at the end: 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, hard to add new features

It's a funny thing that computer science students are often 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 (a former 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.

1.4.1 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 pondering 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 soon became the dominant browser 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 came from a desire to go back to developing in small teams. Without clean and simple interfaces, it's impossible to divide up the work into parts that are independent of one another.

Netscape set aside 2 months to re-architect the browser, but it wasn't long enough. So they decided 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.

In the end, Microsoft won the browser war, and AOL acquired Netscape. Of course this is not the entire story of how Microsoft's browser came to dominate Netscape's. Microsoft's business practices didn't help Netscape. And 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.

I've excerpted this section from an excellent book about Netscape and its business and technical strategies. You can read the whole story there:

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.

1.5 Advice

Course strategy
- don't get behind: pace is fast!
- attend lectures: material is not all in textbook
- think in advance: don't rush to code
- de-sign, not de-bug

Can't emphasize enough importance of starting early and thinking in advance. Of course I don't expect you to finish your problem sets the day they're handed out. But you'll save yourself a lot of time in the long run, and you'll get much better results, if you make *some* start on your work early. First, you'll have the benefit of elapsed time: you'll be mulling problems over subconsciously. Second, you'll know what additional resources you need, and you'll be able to get hold of them while it's easy and in good time. In particular, take advantage of the course staff – we're here to help! We've scheduled LA cluster hours and TA office hours with the handin times in mind, but you can expect more help if it isn't the night before the problem set is due when everyone else wants it…

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, but very relevant to the design of programs

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
- 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.

Optimization rule
- Don't do it
- For experts only: Don't do it yet

(from Michael Jackson, *Principles of Program Design*, Academic Press, 1975).

1.6 Parting Shots

Reminders:
- Tomorrow is a *lecture* not a review session
- Complete online registration form by midnight tonight
- Get started on learning Java now!
- Exercise 1 is due next Tuesday

Check this out:
- http://www.170systems.com/about/our_name.html

Lecture 2: Decoupling I

A central issue in designing software is how to decompose a program into parts. In this lecture, we'll introduce some fundamental notions for talking about parts and how they relate to one another. Our focus will be on identifying the problem of *coupling* between parts, and showing how coupling can be reduced. In the next lecture, we'll see how Java explicitly supports techniques for decoupling.

A key idea that we'll introduce today is that of a *specification*. Don't think that specifications are just boring documentation. On the contrary, they are essential to decoupling and thus to high-quality design. And we'll see that in more advanced designs, specifications become design elements in their own right.

Our course text treats the terms *uses* and *depends* as synonyms. In this lecture, we'll distinguish the two, and explain how the notion of *depends* is a more useful one than the older notion of *uses*. You'll need to understand how to construct and analyze dependency diagrams; uses diagrams are explained just as a stepping stone along the way.

2.1 Decomposition

A program is built from a collection of parts. What parts should there be, and how should they be related? This is the problem of *decomposition*.

2.1.1 Why Decompose?

Dijkstra pointed out that if a program has N parts, and each has a probability of correctness of c – that is, there's a chance of 1-c that the developer gets it wrong – then the probability that the whole assemblage will work is c^N. If N is large, then unless c is very close to one, c^N will be near zero. Dijkstra made this argument to show how much getting it right matters – and the bigger the program gets, the more it matters. If you can't make each part almost perfect, you have no hope of getting the program to work.

(You can find the argument in the classic text *Structured Programming* by Dahl, Dijkstra and Hoare, Academic Press, 1972. It's a seductive and elegant argument, but perhaps a bit misleading. In practice, the probability of getting the whole program completely correct is zero anyway. And what matters is ensuring that certain limited,

but crucial, properties hold, and these may not involve every part. We'll return to this later.)

But doesn't this suggest that we shouldn't break a program into parts? The smaller N is, the higher the probability that the program will work. Of course, I'm joking – it's easier to get a small part right than a big one (so the parameter c is not independent of N). But it's worth asking what benefits come from dividing a program into smaller parts. Here are some:

- *Division of labour*. A program doesn't just appear out of thin air: it has to be built gradually. If you divide it into parts, you can get it built more quickly by having different people work on different parts.
- *Reuse*. Sometimes it's possible to factor out parts that different programs have in common, so they can be produced once and used many times.
- *Modular Analysis*. Even if a program is built by only one person, there's an advantage to buidling it in small parts. Each time a part is complete, it can be analyzed for correctness (by reading the code, by testing it, or by more sophisticated methods that we'll talk about later). If it works, it can be used by another part without revisiting it. Aside from giving a satisfying sense of progress, this has a more subtle advantage. Analyzing a part that is twice is big is much more than twice as hard, so analyzing about a program in small parts dramatically reduces the overall cost of the analysis.
- *Localized Change*. Any useful program will need adaptation and extension over its lifetime. If a change can be localized to a few parts, a much smaller portion of the program as a whole needs to be considered when making and validating the change.

Herb Simon made an intriguing argument for why structures – whether man-made or natural – tend to be build in a hierarchy of parts. He imagines two watchmakers, one of whom builds watches in one go, in a single large assembly, and one of who builds composite subassemblies that he then puts together. Whenever the phone rings, a watchmaker must stop and put down what he as currently working on, spoiling that assembly. The watchmaker who builds in one go keeps spoiling whole watch assemblies, and must start again from scratch. But the watchmaker who builds hierarchically doesn't lose the work he did on the completed subassemblies that he was using. So he tends to lose less work each time, and produces watches more efficiently. How do you think this argument applies to software?

(You can find this argument in Simon's paper *The Architecture of Complexity*.)

2.1.2 What Are The Parts?

What are the parts that a program is divided into? We'll use the term 'part' rather than

'module' for now so we can keep away from programming-language specific notions. (In the next lecture, we'll look at how Java in particular supports decomposition into parts). For now, all we need to note is that the parts in a program are *descriptions*: in fact, software development is really all about producing, analyzing and executing descriptions. We'll soon see that the parts of a program aren't all executable code – it's useful to think of specifications as parts too.

2.1.3 Top Down Design

Suppose we need some part A and we want to decompose into parts. How do we make the right decomposition? This topic is a large part of what we'll be studying in this course. Suppose we decompose A into B and C. Then, at the very least, it should be possible to build B and C, and putting B and C together should give us A.

In the 1970's, there was a popular approach to software development called *Top-down Design*. The idea is simply to apply the following step recursively:

- If the part you need to build is already available (for example, as a machine instruction), then you're done;
- Otherwise, split it into subparts, develop them, and combine them together.

The splitting into subparts was done using 'functional decomposition'. You think about what function the part should have, and break that function into smaller steps. For example, a browser takes user commands, gets web pages, and displays them. So we might split *Browser* into *ReadCommand*, *GetPage*, *DisplayPage*.

The idea was appealing, and there are still people who talk about it with approval. But it fails miserably, and here's why. The very first decomposition is the most vital one, and yet you don't discover whether it was good until you reach the leaves of the decomposition tree. You can't do much evaluation along the way; you can't test a decomposition into two parts that haven't themselves been implemented. Once you get to the bottom, it's too late to do anything about the decompositions you made at the top. So from the point of view of risk – making decisions when you have the information you need, and minimizing the chance and cost of mistakes – it's a very bad strategy.

In practice, what usually happens is that the decomposition is a vague one, with the hope that the parts become more clearly defined as you go down. So you're actually figuring out what problem you're trying to solve as you're structuring the solution. As a result, when you get near the bottom, you find yourself adding all kinds of hacks to make the parts fit together and achieve the desired function. The parts become extensively *coupled* to one another, so that the slightest alteration to one isn't possible without changing all the others. If you're unlucky, the parts don't fit together at all. And,

finally, there's nothing in top-down design that encourages reuse.

(For a discussion of the perils of top-down design, see the article with that title in: *Software Requirements and Specifications: A Lexicon of Software Principles, Practices and Prejudices*, Michael Jackson, Addison Wesley, 1995.)

This isn't to say, of course, that viewing a system hierarchically is a bad idea. It's just not possible to develop it that way.

2.1.4 A Better Strategy

A much better strategy is to develop a system structure considering of multiple parts at a roughly equal level of abstraction. You refine the description of every part at once, and analyze whether the parts will fit together and achieve the desired function before starting to implement any of them. It also turns out that it is much better to organize a system around data than around functions.

Perhaps the most important consideration in evaluating the decomposition into parts is how the parts are coupled to one another. We want to minimize coupling – to *decouple* the parts – so that we can work on each part independently of the others. This is the topic of our lecture today; later in the course, we'll see how we can express properties of the parts and the details of how they interact with one another.

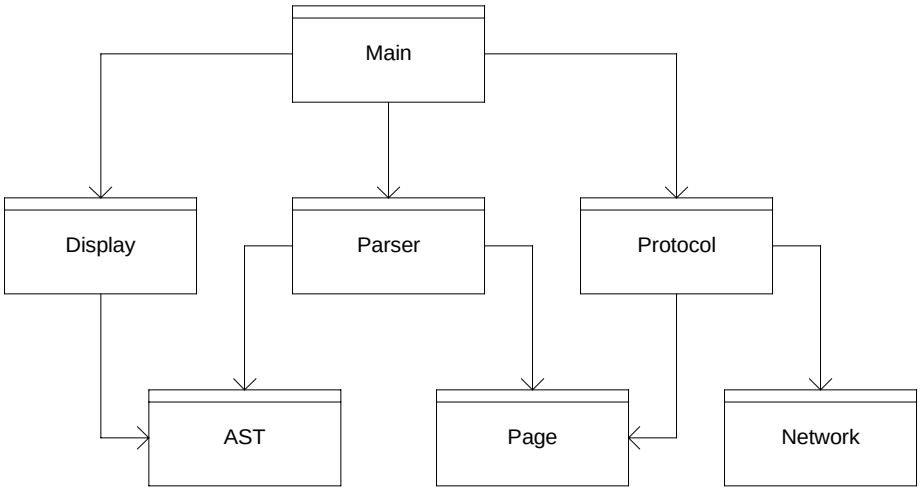
2.2 Dependence Relationships

2.2.1 Uses Diagram

The most basic notion relationship between parts is the *uses* relationship. We say that a part A uses a part B if A refers to B in such a way that the meaning of A depends on the meaning of B. When A and B are executable code, the meaning of A is its behaviour when executed, so A uses B when the behaviour of A depends on the behaviour of B.

Suppose, for example, we're designing a web browser. The diagram shows a putative decomposition into parts:

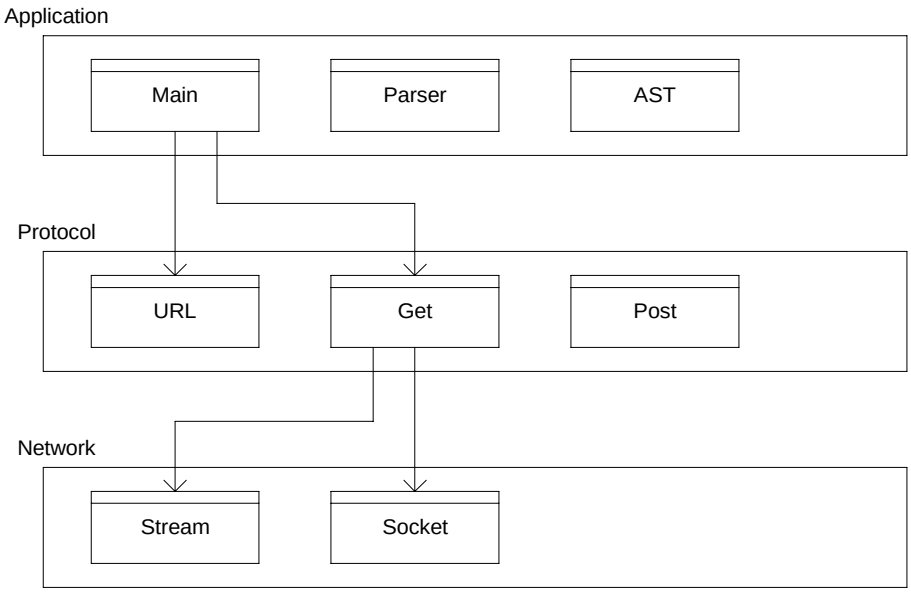
The *Main* part uses the *Protocol* part to engage in the HTTP protocol, the *Parse* part to parse the HTML page received, and the *Display* part to display it on the screen. These parts in turn use other parts. *Protocol* uses *Network* to make the network connection and to handle the low-level communication, and *Page* to store the HTML page



received. *Parser* uses the part *AST* to create an abstract syntax tree from the HTML page – a data structure that represents the page as a logical structure rather than as a sequence of characters. *Parser* also uses *Page* since it must be able to access the raw HTML character sequence. *Display* uses a part *Render* to render the abstract syntax tree on the screen.

Let's consider what kind of shape a uses graph may take.

- *Trees*. First, note that when viewed as a graph, the uses-diagram is not generally a tree. Reuse causes a part to have multiple users. And whenever a part is decomposed into two parts, it is likely that those parts will share a common part that enables them to communicate. *AST*, for example, allows *Parser* to communicate its results to *Display*.
- *Layers*. Layered organizations are common. A more detailed uses-diagram of our browser may have several parts in place of each of the parts we showed above. The *Network* part, for example, might be replaced by *Stream*, *Socket*, etc. It is sometimes useful to think of a system as a sequence of layers, each providing a coherent view of some underlying infrastructure, at varying levels of abstraction. The *Network* layer provides a low-level view of the network; the *Protocol* layer is built on top of it, and provides a view of the network as an infrastructure for processing HTTP queries, and the top layer provides the application user's view of the system, which turns URLs into visible web pages. Technically, we can make any uses-diagram layered by assigning each part to a layer so that no *uses* arrow points from a part in

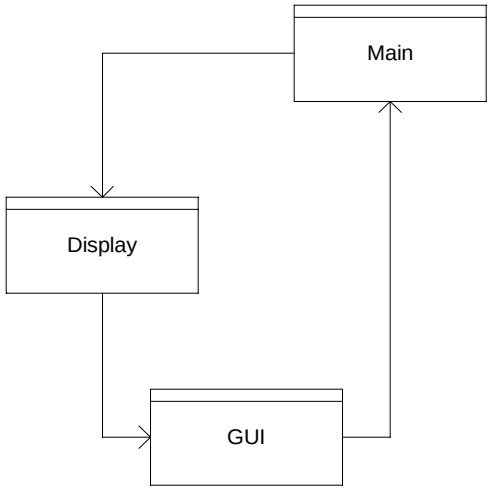


some layer to a part in a higher layer. But this doesn't really make the program layered, since the layers have no conceptual coherence.

- *Cycles*. It's quite common to have cycles in the uses-diagram. This doesn't have to mean that there is recursion in the program. Here's how it might arise in our browser design. We haven't considered how *Display* will work. Suppose we have a *GUI* part that provides functions for writing to a display, and handles input by making calls (when buttons are pressed, etc) to functions in other parts. Then *Display* may uses *GUI* for output, and *GUI* may use *Main* for input. In object-oriented designs, as we'll see, cycles often arises when objects of different classses interact strongly.

What can we do with the uses-diagram?

- *Reasoning*. Suppose we want to determine whether a part *P* is correct. Aside from *P* itself, which parts do we need to examine? The answer is: the parts *P* uses, the parts they use, and so on – in other words all parts reachable from *P*. In our browser example, to check that *Display* works we'll need to lok at *Render* and *AST* also. Conversely, if we make a change to *P*, which parts might be affected? The answer is all parts that use *P*, the parts that use them, and so on. If we change *AST* for example, *Display*, *Parser* and *Main* may all have to change. This is called *impact analysis* and it's important during maintenance of a large program when you want to make sure that the consequences of a change are completely known, and you want to avoid retesting every part.



- *Reuse*. To identify a subsystem – a collection of parts – that can be reused, we have to check that none of its parts use any other parts not in the subsystem. The same determination tells us which how to find a minimal subsystem for initial implementation. For example, the parts *Display*, *Render* and *AST* form a collection without dependences on other parts, and could be reused as a unit.
- *Construction Order*. The uses diagram helps determine what order to build the parts in. We might assign two sets of parts to two different groups and let them work in parallel. By ensuring that no part in one set uses a part in another set, we can be sure that neither group will be stalled waiting for the other. And we can construct a system incrementally by starting at the bottom of the uses diagram, with those parts that don't use any other parts, and then move upwards, assembling and testing whenever we have a consistent subsystem. For example, the *Display* and *Protocol* parts could be developed independently along with the parts they use, but not *Display* and *Parser*.

Thinking about these considerations can shed light on the quality of a design. The cycle we mentioned above, (Main–Display–GUI–Main), for example, makes it impossible to reuse the *Display* part without also reusing *Main*.

There's a problem with the uses diagram though. Most of the analyses we've just discussed involve finding all parts reachable or reaching a part. In a large system, this may be a high proportion of the parts in a system. And worse, as the system grows, the problem gets worse, even for existing parts which refer directly to no more parts than they did before. To put it differently, the fundamental relationship that underlies *uses*

is transitive: if *A* is affected by *B* and *B* is affected by *C*, then *A* is affected by *C*. It would be much better if reasoning about a part, for example, required looking at only at the parts it refers to.

The idea of the uses relation, and its role in thinking about software structure, was first described by David Parnas in *Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering, Vol. SE-5, No 2, 1979.

2.2.2 Dependences & Specifications

The solution to this problem is to have instead a notion of dependence that stops after one step. To reason about some part *A*, we will need to consider only the parts it depends on. To make this possible, it will be necessary for every part that *A* depends on to be complete, in the sense that its description completely characterizes its behaviour. It cannot itself depend on other parts. Such a description is called a *specification*.

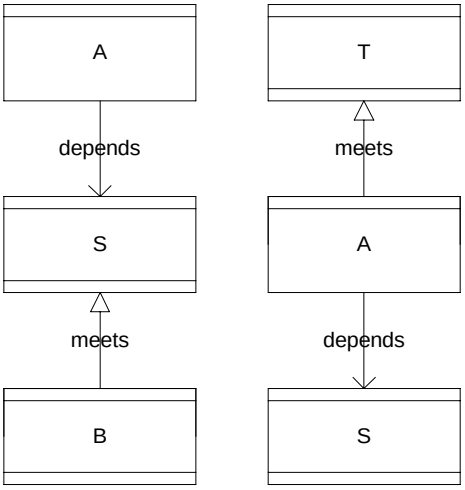
A specification cannot be executed, so we'll need for each specification part at least one implementation part that behaves according to the specification. Our diagram, the *dependency diagram*, therefore has two kinds of arcs. An implementation part may *depend* on a specification part, and it may *fulfill* or *meet* a specification part.

In comparison to what we had before, we have broken the *uses* relationship between two parts *A* and *B* into two separate relationships. By introducing a specification part *S*, we can say that *A* depends on *S* and *B* meets *S*. The diagram on the left illustrates this; note the use of two double lines to distinguish specification parts from implementation parts.

Each arc incurs an obligation. The writer of *A* must check that it will work if it is assembled with a part that satisifies the specification *S*. And 'works' is now defined by explicitly by meeting specifications: *B* will be usable in *A* if it works according to the specification *S*, and *A* will be deemed to work if it meets whatever specification is given for its intended uses – *T* say. The diagram on the right shows this. It's the same depends-meet chain centered on an implementation part rather than a specification part.

This is a much more useful and powerful framework than *uses*. The introduction of specifications brings many advantages:

- *Weakened Assumptions.* When *A* uses *B*, it is unlikely to rely on every aspect of *B*. Specifications allow us to say explicitly which aspects matter. By making specifications much smaller and simpler than implementations, we can make it much easier to reason about correctness of parts. And a weak specification gives more opportu-

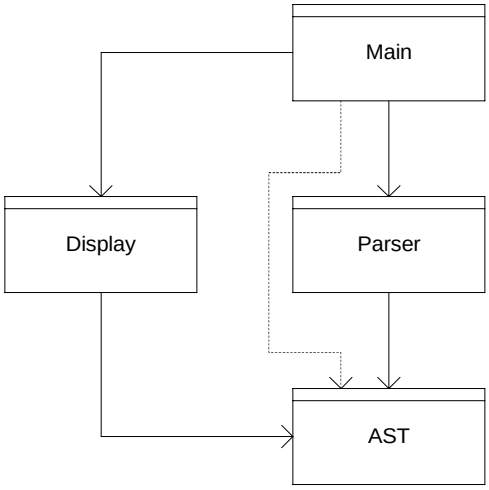


nities for performance improvements.

- *Evaluating Changes*. The specification *S* helps limit the scope of a change. Suppose we want to change *B*. Must *A* change as well? Now this question doesn't require looking at *A*. We start by looking at *S*, the specification *A* requires of the part it uses. If the new *B* will still meet *S*, then no change to *A* will be needed at all.
- *Communication*. If *A* and *B* are to be built by different people, they only need to agree upon *S* in advance. *A* can ignore the details of the services *B* provides, and *B* can ignore the details of the needs of *A*.
- *Multiple Implementations*. There can be many different implementation parts that meet a given specification part. This makes a market in interchangeable parts possible. Parts are marketed in a catalog by the specifications they meet, and a customer can pick any part that meets the required specification. A single system can provide multiple implementations of a part. The selection can be made when the system is configured, or as we shall see later in the course, during execution of the system.

Specifications are so useful that we'll assume that there is a specification part corresponding to every implementation part in our system, and we'll conflate them, drawing dependences directly from implementations to implementations. In other words, a dependence arc from *A* to *B* means that *A* depends on the specification of *B*.

So whenever we draw a diagram like the one of our browser above, we'll interpret it as a dependence diagram and not as a uses diagram. For example, it will be possible to have teams build *Parser* and *Protocol* in parallel as soon as the *specification* of *Page* is



complete; its implementation can wait.

Sometimes, though, specifications are design elements in their own right and we'll want to make explicit their presence. Java provides some useful mechanisms for expressing decoupling with specifications, and we'll want to show these. Design patterns, which will be studying later in the term, make extensive use of specifications in this way.

2.2.3 Weak Dependences

Sometimes a part is just a conduit. It refers to another part by name, but doesn't make use of any service it provides. The specification it depends on requires only that the part exist. In this case, the dependence is called a *weak dependence*, as is drawn as a dotted arc.

In our browser, for example, the abstract syntax tree in AST may be accessible as a global name (using the Singleton pattern, which we'll see later). But for various reasons – we might for example later decide that we need two syntax trees – it's not wise to use global names in this way. An alternative is for the *Main* part to pass the *AST* part from the *Parse* part to the *Display* part. This would induce a weak dependence of *Main* on *AST*. The same reasoning would give a weak dependence of *Main* on *Page*.

In a weak dependence of A on B, A usually depends on the name of B. That is, it not only requires that there be some part satisfying the specification of B, but it also requires that it be called B. Sometimes a weak dependence doesn't constrain the name.

In this case, *A* depends only the existence of some part satisfying the specification of *B*, and *A* will refer to such a part using the name of the specification of *B*. We will see how Java allows this kind of dependence. In this case, it's useful to show the specification of *B* as a separate part with its own name.

For example, the *Display* part of our browser may use a part *UI* for its output, but need not know whether the *UI* is graphical or text-based. This part can be a specification part, met by an implementation part *GUI* which *Main* depends on (since it creates the actual GUI object). In this case, *Main*, because it passes an object whose type is described as *UI* to *Display*, must also have a weak dependence on the specification part *UI*.

2.3 Techniques for Decoupling

So far, we've discussed how to represent dependences between program parts. We've also talked about some of the effects of dependences on various development activities. In every case, a dependence is a *liability*: it expands the scope of what needs to be considered. So a major part of design is trying to minimize dependences: to *decouple* parts from one another.

Decoupling means minimizing both the quantity and quality of dependences. The quality of a dependence from *A* to *B* is measured by how much information is in the specification of *B* (which, recall from above, is what *A* actually depends on). The less information, the weaker the dependence. In the extreme case, there is no information in the dependence at all, and we have a weak dependence in which *A* depends only on the existence of *B*.

The most effective way to reduce coupling is to design the parts so that they are simple and well defined, and bring together aspects of the system that belong together and separate aspects that don't. There are also some tactics that can be applied when you already have a candidate decomposition: they involve introducing new parts and altering specifications. We'll see many of these throughout the term. For now, we'll just mention some briefly to give you an idea of what's possible.

2.3.1 Facade

The facade pattern involves interposing a new implementation part between two sets of parts. The new part is a kind of gatekeeper: every use by a part in the set *S* of a part in the set *B* which was previously direct now goes through it. This often makes sense in a layered system, and helps to decouple one layer from another.

In our browser, for example, there may be many dependences between parts in a protocol layer and parts in a networking layer. Unless all the networking parts are platform-independent, porting the browser to a new platform may require replacing the networking layer. Every part in the protocol layer that depends on a networking part may have to be examined and altered.

To avoid this problem, we might introduce a facade part that sits between the layers, collects together all the networking that the protocol layer needs (and no more), and presents them to the protocol layer with a higher-level interface. This interface is, of course, a new specification, weaker than the specifications on which the protocol parts used to rely. If done right, it may now be possible to change the parts of the networking layer while leaving the facade's specification unchanged, so that no protocol parts will be affected.

2.3.2 Hiding representation

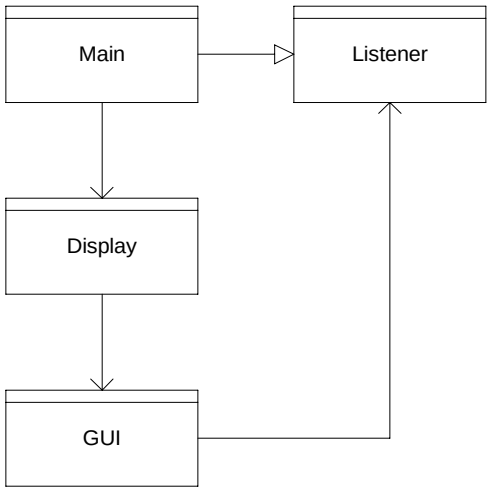
A specification can avoid mentioning how data is represented. Then the parts that depend on it cannot manipulate the data directly; the only way to manipulate the data is to use operations that are included in the specification of the used part. This kind of specification weakening is known as 'data abstraction', and we'll have a lot to say about it in the next few weeks. By eliminating the dependence of the using part *A* on the representation of data in the used part *B*, it makes it easier to understand the role that *B* plays in *A*. It makes it possible to change the representation of data in *B* without any change to *A* at all.

In our browser, for example, the specification part associated with *Page* might say that a web page is a sequence of characters, hiding details of its representation using arrays.

2.3.3 Polymorphism

A program part *C* that provides container objects has a dependence on the program part *E* that provides the elements of the container. For some containers, this is a weak dependence, but it need not be: *C* may use *E* to compare elements (eg, to check for equality, or to order them). Sometimes *C* may even use functions of *E* that mutate the elements.

To reduce the coupling between *C* and *E*, we can make *C* polymorphic. The word 'polymorphic' means 'many shaped', and refers to the fact that *C* is written without any mention of special properties of *E*, so that containers of many shapes can be produced according to which *E* the part *C* uses. In practice, pure polymorphism is rare, and *C*



will at least rely on equality checks provided by *E*. Again, what's going on is a weakening of the specification that connects *C* to *E*. In the monomorphic case, *C* depends on the specification of *E*; in the polymorphic case, *C* depends on a specification *S* that says only that the part must provide objects with an equality test. In Java, this specification *S* is the specification of the *Object* class.

In our browser, for example, the data structure used for the abstract syntax tree might use a generic *Node* specification part, which is implemented by an *HTMLNode* part, for much of its code. A change in the structure of the markup language would then affect less code.

2.3.4 Callbacks

We mentioned above how, in our browser, a *GUI* part might depend on the *Main* part because it calls a procedure in *Main* when, for example, a button is pressed. This coupling is bad, because it makes intertwines the structure of the user interface with the structure of the rest of the application. If we ever want to change the user interface, it will be hard to disentangle it.

Instead, the *Main* part might pass the *GUI* part at runtime a reference to one of its procedures. When this procedure is called by the *GUI* part, it has the same effect it would have had if the procedure had been named in the text of the *GUI* part. But since the association is only made at runtime, there is no dependence of *GUI* on *Main*. There will be a dependence of *GUI* on a specification (*Listener*, say) that the passed procedure

must satisfy, but this is usually minimal: it might say, for example, just that the procedure returns without looping forever, or that it does not cause procedures within *GUI* itself to be called. This arrangement is a *callback*, since *GUI* 'calls back' to *Main* against the usual direction of procedure call. In Java, procedures can't be passed, but the same effect can be obtained by passing a whole object.

2.4 Coupling Due to Shared Constraints

There's a different kind of coupling which isn't shown in a module dependency diagram. Two parts may have no explicit dependence between them, but they may nevertheless be coupled because they are required to satisfy a constraint together.

For example, suppose we have two parts, *Read*, which reads files, and *Write*, which writes files. If the files read by *Read* are the same files written by *Write*, there will be a constraint that the two parts agree on the file format. If the file format is changed, both parts will need to change.

To avoid this kind of coupling, you have to try to localize functionality associated with any constraint in a single part. This is what Matthias Felleisen calls 'single point of control' in his novel introduction to programming in Scheme (*How to Design Programs, An Introduction to Programming and Computing*, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, MIT Press, 2001).

David Parnas suggested that this idea should form the basic of the selection of parts. You start by listing the key design decisions (such as choice of file format), and then assign each to a part that keeps that decision 'secret'. This is explained in detail with a nice example in his seminal paper *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No. 12, December 1972 pp. 1053–1058.

2.5 Back to Dijkstra: Conclusion

Dijsktra's warning that the chance of getting a program right will drop to zero as the number of parts increases is worrying. But if we can decouple the parts so that each of the properties we care about is localized within only a few parts, then we can establish their correctness locally, and be immune to the addition of new parts.

Lecture 3: Decoupling II

In the last lecture, we talked about the importance of dependences in the design of a program. A good programming language allows you to express the dependences between parts, and control them – preventing unintended dependences from arising. In this lecture, we'll how the features of Java can be used to express and tame dependences. We'll also study a variety of solutions to a simple coding problem, illustrating in particular the role of interfaces.

3.1 Review: Module Dependency Diagrams

Let's start with a brief review of the module dependency diagram (MDD) from the last lecture. An MDD shows two kinds of program parts: implementation parts (classes in Java) shown as boxes with a single extra stripe at the top, and specification parts shown as boxes with a stripe at the top and bottom. Organizations of parts into groupings (such as packages in Java) can be shown as contours enclosing parts in Venn-diagram-style.

A plain arrow with an open head connects an implementation part A to a specification part S, and says that the meaning of A depends on the meaning of S. Since the specification S cannot itself have a meaning dependent on other parts, this ensures that a part's meaning can be determined from the part itself and the specifications it depends on, and nothing else. A dotted arrow from A to S is a weak dependence; it says that A depends only the existence of a part satisfying the specification S, but actually has no dependence on any details of S. An arrow from an implementation part A to a specification part S with a closed head says that A meets S: its meaning conforms to that of S.

Because specifications are so essential, we will always assume they are present. Most of the time, we will not draw specification parts explicitly, and so a dependence arrow between two implementation parts A and B should be interpreted as short for a dependence from A to the specification of B, and a meets arrow from B to its specification. We will show Java interfaces as specification parts explicitly.

3.2 Java Namespace

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.

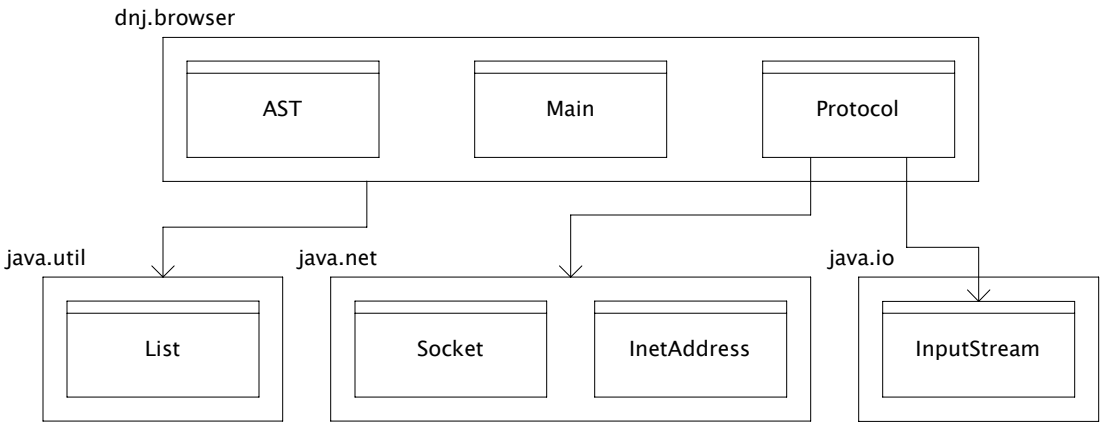
Here's how the Java naming system works. The key named components are classes and interfaces, and they have named methods and named fields. Local variables (within methods) and method arguments are also named. Each name in a Java program has a scope: a portion of the program text over which the name 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. For example, the class *dnj.browser.Protocol* would be in a file called *Protocol.java* in the directory *dnj/browser*.

We can show this structure in our dependence diagram. The classes and interfaces form the parts between which dependences are shown. Packages are shown as contours enclosing them. It's convenient sometimes to hide the exact dependences between parts in different packages and just show a dependence arc at the package level. A dependence from a package means that some class or interface (or maybe several) in that package has a dependence; a dependence on a package means a dependence on some class or interface (or maybe several) in that package.

3.3 Access Control

Java's access control mechanisms allow you to control dependences. In the text of a class, you can indicate which other classes can have dependences on it, and to some extent you can control the nature of the dependences.



A class declared as *public* can be referred to by any other class; otherwise, it can be referred to only by classes in the same package. So by dropping this modifier, we can prevent dependences on the class from any class outside the package.

Members of a class – that is, its fields and methods – may be marked *public*, *private* or *protected*. A *public* member can be accessed from anywhere. A *private* member can be accessed only from within the class in which the field or method is declared. A *protected* member can be accessed within the package, or from outside the package by a subclass of the class in which the member is declared – thus creating the very odd effect that marking a member as *protected* makes it more, not less, accessible.

Recall that a dependence of A on B really means a dependence of A on the specification of B. Modifiers on members of B allow us to control the nature of the dependence by changing which members belong to B's specification. Controlling access to the fields of B helps give representation independence, but it does not always ensure it (as we'll see later in the course).

3.4 Safe Languages

A key property of a program is that one part should only depend on another if it names it. This seems obvious, but in fact it's a property that only holds for programs written in so-called 'safe languages'. In an unsafe language, the text in one part can affect the behaviour of another without any names being shared. This leads to insidious bugs that are very hard to track down, and which can have disastrous and unpredictable effects.

Here's how it happens. Consider a program written in C in which one module (in C,

just a file) updates an array. An attempt to set the value of an array element beyond the bounds of the array will sometimes fail, because it causes a memory fault, going beyond the memory area assigned to the process. But, unfortunately, more often it will succeed, and the result will be to overwrite an arbitrary piece of memory – arbitrary because the programmer does not know how the compiler laid out the program's memory, and cannot predict what other data structure has been affected. As a result, an update of the array *a* can affect the value of a data structure with the name *d* that is declared in a different module and doesn't even have a type in common with *a*.

Safe languages rule this out by combining several techniques. Dynamic checking of array bounds prevents the kind of updating we just mentioned from occurring; in Java, an exception would be thrown. Automatic memory management ensures that memory is not reclaimed and then mistakenly reused. Both of these rely on the fundamental idea of *strong typing*, which ensures that an access that is declared to be to a value of type *t* in the program text will always be an access to a value of type *t* at runtime. There is no risk that code designed for an array will be mistakenly applied to a string or an integer.

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.

Some safe languages guarantee type correctness at compile time – by 'static typing'. Others, such as Scheme and LISP, do their type checking at runtime, and their type systems only distinguish primitive types from one another. We'll see shortly how a more expressive type system can also help control dependences.

If reliability matters, it's wise to use a safe language. In lecture, I told a story here about use of unsafe language features in a medical accelerator.

3.5 Interfaces

In languages with static typing, one can control dependences by choice of types. Roughly speaking, a class that mentions only objects of type *T* cannot have a dependence on a class that provides objects of a different type *T'*. In other words, one can tell from the types mentioned in a class which other classes it depends on.

However, in languages with subtyping, something interesting is possible. Suppose class *A* mentions only the class *B*. This does *not* mean that it can only call methods on objects created by class *B*. In Java, the objects created by a subclass *C* of *B* are regarded as also having the type *B*, so even though *A* can't create objects of class *C* directly, it can be passed them by another class. The type *C* is said to be a *subtype* of the type *B*, since a *C* object can be used when a *B* object is expected. This is called 'substitutability'.

Subclassing actually conflates two distinct issues. One is subtyping: that objects of class *C* are to be regarded as having types compatible with *B*, for example. The other is inheritance: that the code of class *C* can reuse code from *B*. Later in the course we'll discuss some of the unfortunate consequences of conflating these two issues, and we'll see how substitutability doesn't always work as you might expect.

For now, we'll focus on the subtyping mechanism alone, since it's what's relevant to our discussion. Java provides a notion of *interfaces* which give more flexibility in subtyping than subclasses. A Java interface is, in our terminology, a pure specification part. It contains no executable code, and is used only to aid decoupling.

Here's how it works. Instead of having a class *A* depend on a class *B*, we introduce an interface *I*. *A* now mentions *I* instead of *B*, and *B* is required to meet the specification of *I*. Of course the Java compiler doesn't deal with behavioural specifications: it just checks that the types of the methods of *B* are compatible with the types declared in *I*. At runtime, whenever *A* expects an object of type *I*, an object of type *B* is acceptable.

For example, in the Java library, there is a class *java.util.LinkedList* that implements linked lists. If you're writing some code that only requires that an object be a list, and not necessarily that it be a linked list, you should use the type *java.util.List* in your code, which is an interface implemented by *java.util.LinkedList*. There are other classes, such as *ArrayList* and *Vector* that implement this interface. So long as your code refers only to the interface, it will work with any of these implementation classes.

Several classes may implement the same interface, and a class may implement several interfaces. In contrast, a class may only subclass at most one other class. Because of this, some people use the term 'multiple specification inheritance' to describe the interface feature of Java, in contrast to true multiple inheritance in which can reuse code from multiple superclasses.

Interfaces bring primarily two benefits. First, they let you express pure specification parts in code, so you can ensure that the use of a class *B* by a class *A* involves only a dependence of *A* on a specification *S*, and not on other details of *B*. Second, interfaces let you provide several implementation parts that meet a single specification, with the

selection being made at compile time or at runtime.

3.6 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.

Suppose 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*.

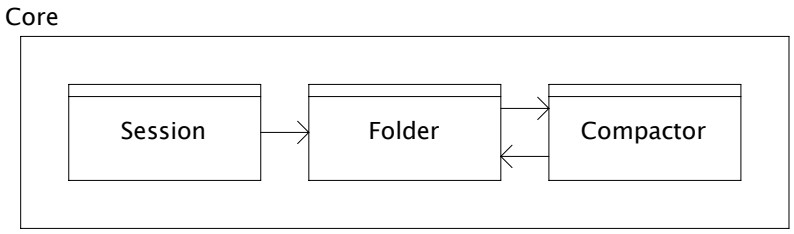
We'll look at a variety of ways to implement our instrumentation facility, and we'll study the advantages and disadvantages of each. Starting with the simplest, most naive design possible, we might intersperse statements such as

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

throughout the program.

3.6.1 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);
        }
    }
```

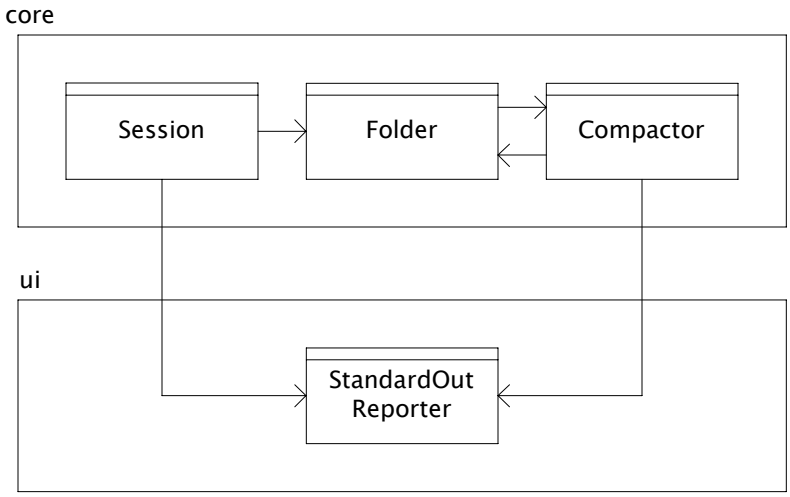
Now the change can be made at a single point in the code. We just modify the procedure:

```
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, such as

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 instrumenta-



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

3.6.2 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" );
    ...
    }
```

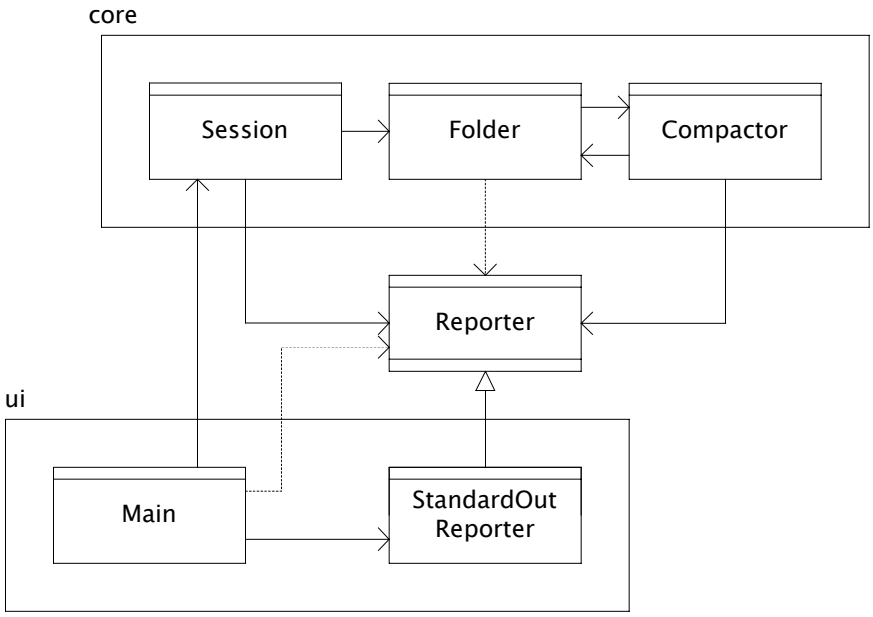
Now we define a class that will actually implement the reporting behaviour. Let's use standard out as our example as it's simpler:

```
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.

Look at the module dependency diagram. Recall that an arrow with a closed head from *A* to *B* is read '*A* meets *B*'. *B* might be a class or an interface; the relationship in Java may be implements or extends. Here, the class *StandardOutReporter* meets 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.

Recall that 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.

3.6.3 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.

3.6.4 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:

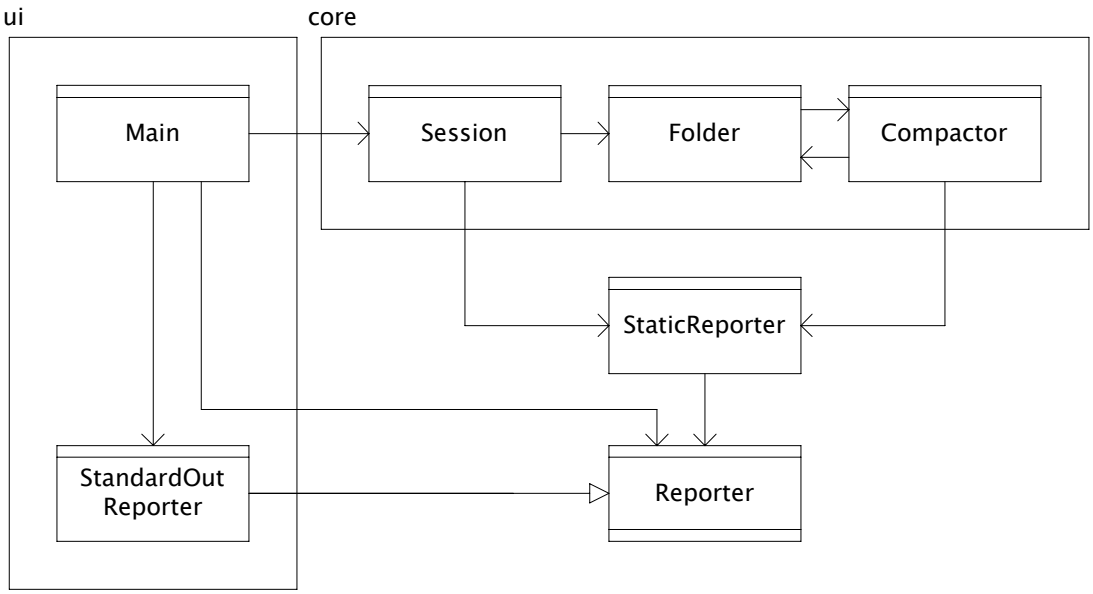
```
void download (...) {
    StaticReporter.report ("Starting downloading" );
    ...
    }
```

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 *StaticReporter.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 com-



ponents 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.

Lecture 4: Procedure Specifications

4.1. Introduction

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 behavior 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 decoupling and dependences in the last two lectures. 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.

4.2. Behavioral 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 behavior:

- 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 behavior 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*

4.3. 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.

4.4. 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 behavior in all cases, the specification makes implementation easier. The term non-determinism suggests that the implementation should exhibit all possible behaviors 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 behavior, but having chosen to model the observers as a set, there is no way to specify an order anyway.

4.5. 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* in a forthcoming lecture.

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.

4.6. 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.

4.7. 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).

4.8. 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.

4.9. 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 5: Abstract Types

5.1. Introduction

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.

5.2. 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.

5.3. 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).

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.

5.4. 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.

5.5. Designing an Abstract Type

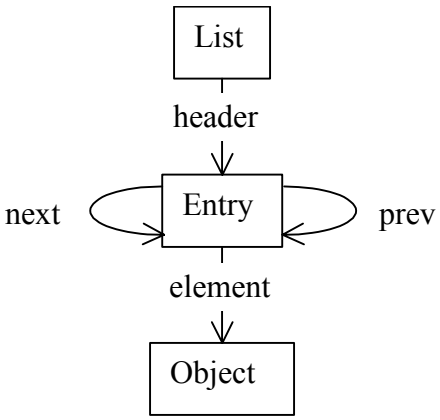
Designing an abstract type involves choosing good operations and determining how they should behave. A few rules of thumb:

- 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 behavior 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.

5.6. 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

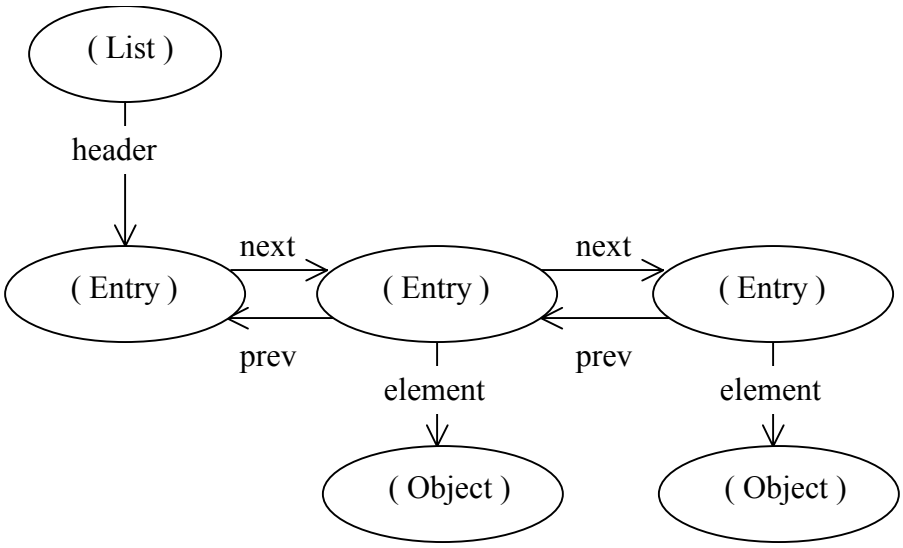


model shows a linked list implementation similar (but not identical to) the *LinkedList* class in the standard Java library:

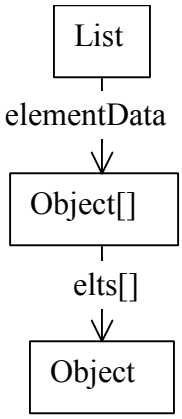
The list object has a field *header* that references an *Entry* object. An *Entry* object is a record with three fields: *next* and *prev* which may hold references to other *Entry* objects (or be null), and *element*, which holds a reference to an element object. The *next* and *prev* fields are links 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:

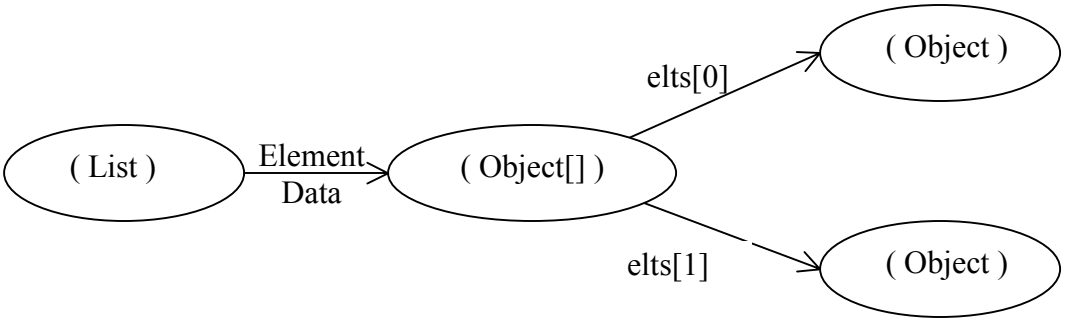


Another, different representation of lists uses an array. The following object model



shows how lists are represented in the class *ArrayList* in the standard Java library:

Here's a list with two elements in its 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.

5.7. 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 behaviors 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)

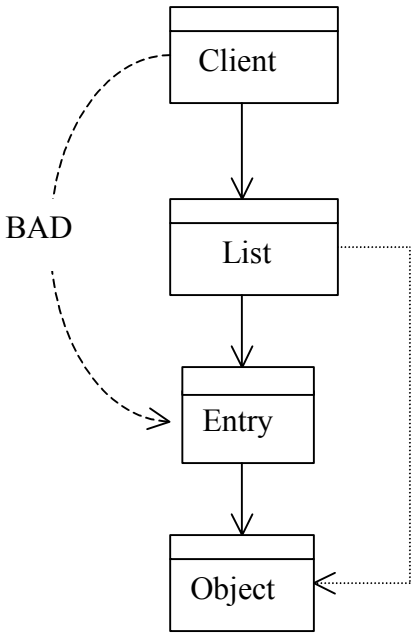
Our rationale is that if there are many calls to *set* on the same index, this will save the linear search of repeatedly obtaining the element. Instead of

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

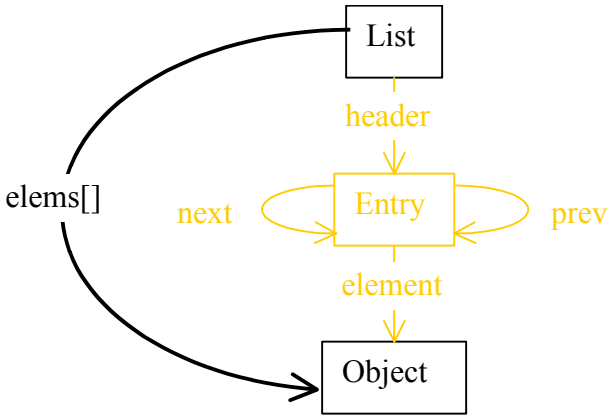
we can now write

```
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:



In the *Entry* example we have exposed the representation. 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, suppose that *List* 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 *size* field was based on the index at which a null reference first appears) a modification to this array may break the computation of size.

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

Once size is computed wrongly, all hell breaks loose: subsequent operations may behave in arbitrary ways.

5.8. Language Mechanisms

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

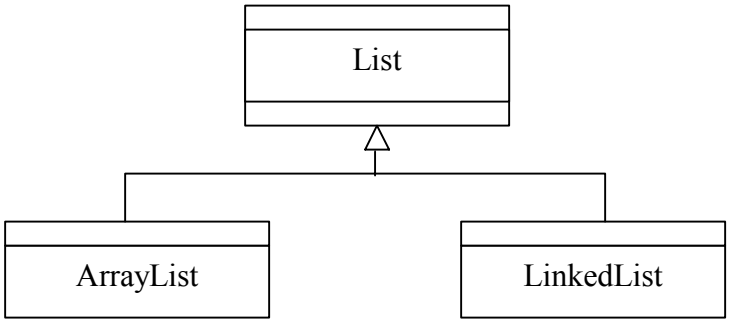
```
v.copyInto (l.elementData);
```

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. 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.

5.9. 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 6: Representation Invariants and Abstraction Functions

6.1 Introduction

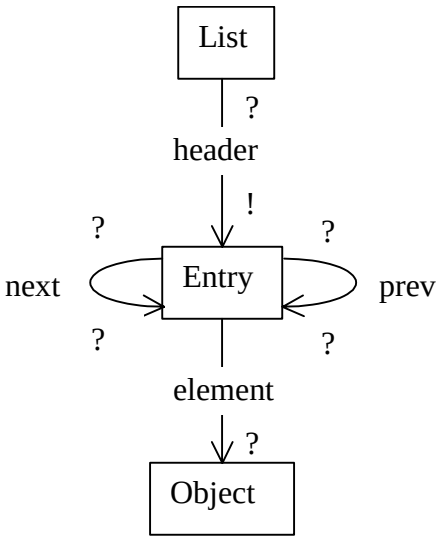
In this lecture, we describe two tools for understanding abstract data types: the representation invariant and the abstraction function. The representation invariant describes whether an instance of a type is well formed; the abstraction function tells us how to interpret it. Representation invariants can amplify the power of testing. 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.

6.2 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

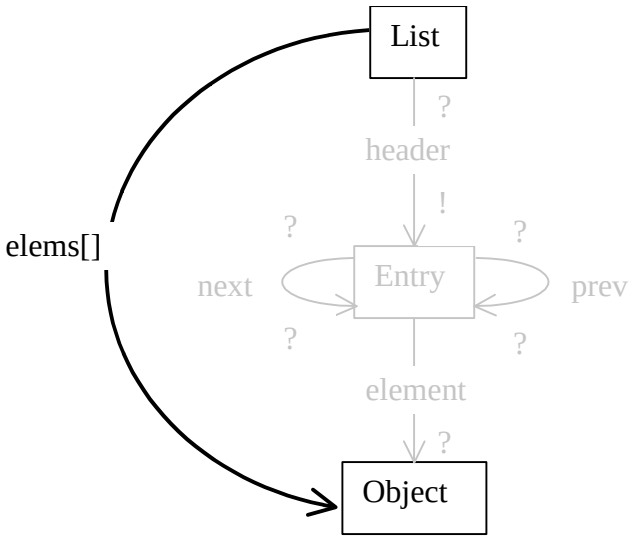
Consider the linked list implementation that we discussed last time. Here was its object model:



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 gold (*Entry* and its incoming and outgoing arcs) hidden:



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 ! symbol denotes exactly one.)
- The multiplicities ? on the target end of the next and prev arrows say that each of the next and prev arrows point to at most one entry. (The ? symbol denotes zero or one.)
- 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's *next* field, and by at most one other entry's *prev* field. (The ? symbol denotes zero or one.)
- The multiplicity ? on the target end of the element field says that each Entry points to at most one Object.

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.

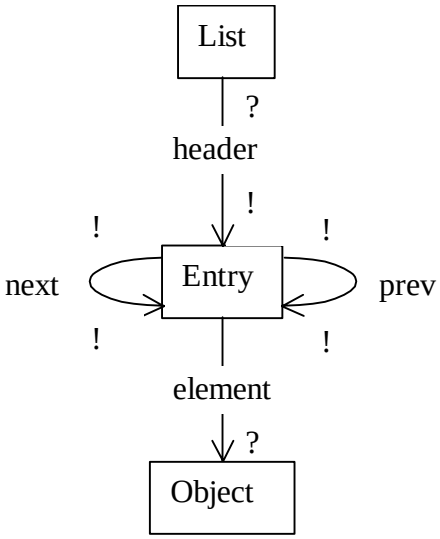
There are properties of the representation invariant which are not shown in the graphical object model:

- When there are two *e1* and *e2* entries in the list, if *e1.next* = *e2*, then *e2.prev* = *e1*.

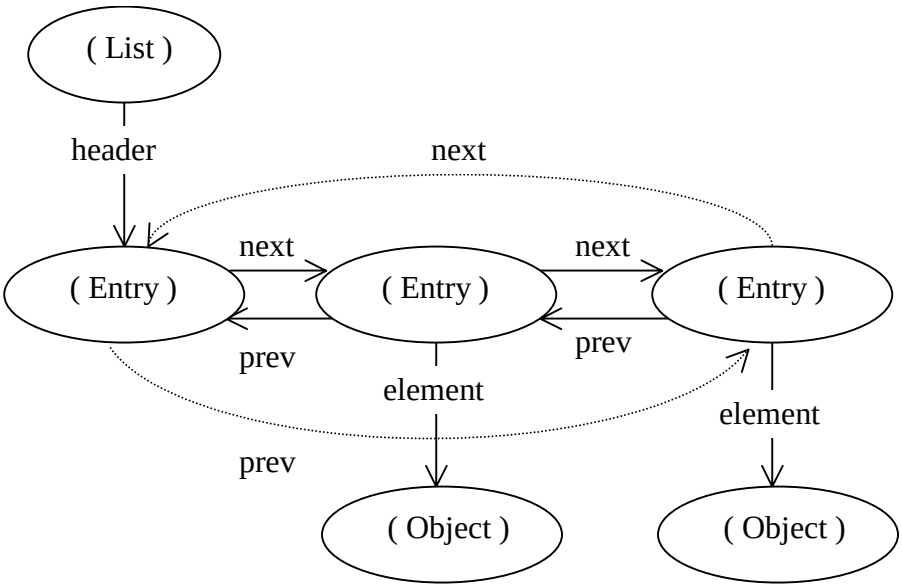
- The dummy entry at the front of the list has a null *element* field.

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 multiplicities on the next and prev arrows. 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 |
    && p.header = p.header.next^(p.size + 1)
    && all e in 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 |
    && p.header = p.header.next^(p.size + 1)
    && all e in 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 other than the header 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.

6.3 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, i.e., this new entry becomes the last entry in the *next* chain, 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.prev* 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.prev*, 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.

6.4 Interpreting the Representation

Consider the mutator *add* again, 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 $p.next^{i+1}.element$*)*

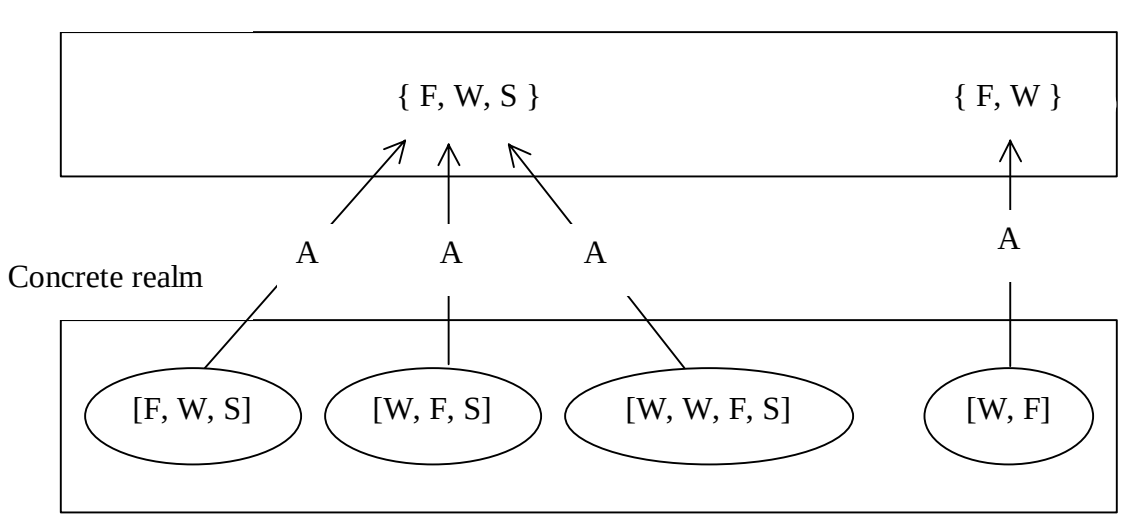
6.5 Abstract and Concrete Objects

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.S)) ...

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 {}) $\cup$
(if s.boolarr[1] then {W} else {}) $\cup$
(if s.boolarr[2] then {S} else {}) $\cup$
(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 {}) $\cup$
(if s.boolarr[1] then {U} else {}) $\cup$
(if s.boolarr[2] then {F} else {}) $\cup$
(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.

6.6 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 =>lateNights

that!

courseSix => 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 => sixOneSeventy) (sixOneSeventy => lateNights) (! (courseSix => 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}{ !b,c}}

is equivalent to the conventional formula

a Λ (!b V c)

Our formula above would be represented in CNF as

{ {! courseSix,sixOneSeventy}, {! sixOneSeventy, lateNights}, {courseSix}, {! 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:

$$A(f) = true \wedge C(f.clauses.elts[0]) \wedge \ldots \wedge C(f.clauses.elts[(size(f.clauses) - 1])$$

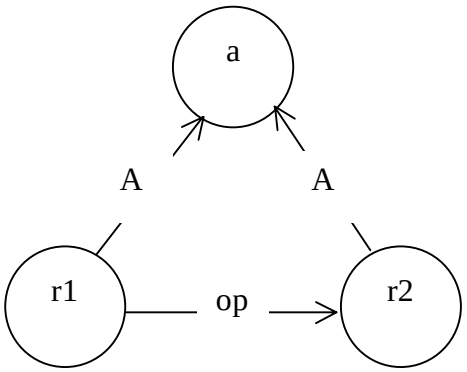
where C(c) = false V c.elts[0] V ... V c.elts[0]

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.

6.7 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.

For example, 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.

6.8 Summary

Why use rep invariants? Recording the invariant can actually save 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.

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 7: Iteration Abstraction and Iterators

7.1 Introduction

In this lecture, we describe iteration abstraction and iterators. Iterators are a generalization of the iteration mechanism available in most programming languages. They permit users to iterate over arbitrary types of data in a convenient and efficient way.

For example, an obvious use of a set is to perform some action for each of its elements:

for all elements of the set
do action

In this lecture we discuss how we can specify and implement iteration abstraction. We also describe representation exposure as related to iterators.

7.2 Reading

Read Chapter 6 of Liskov and Guttag before moving on. The first half of the material of this lecture is based on Chapter 6 of the book, and will not be duplicated here.

7.3 Representation Exposure in Iterators

Consider the implementation of an iterator for IntSet. The general structure of the class IntSet would look like this:

```
public class IntSet {
   private Vector els;   // the rep
  private int size;  // the rep

  // constructors, etc go here, see p. 88 of Liskov
        ...
   public Iterator elems() {
      return new IntGen(this); }

   // static inner class
   private static class IntGen implements Iterator {

        public boolean hasNext() {  ... }
        public Object next() throws NoSuchElementException { ... }
        public void remove(Object o) { ... }

   } // end of IntGen
}
```

Notice the additional method *remove()* in the *IntGen* class. This method is not required to be implemented, it is optional. The method allows one to remove an element from *IntSet* while iterating over the elements of the set. It has to be implemented very carefully!

Note that in Liskov, modifications to the object being iterated over (i.e., *IntSet* in our example) are not allowed. However, the Java iterator interface includes the optional *remove()* method.

Now we wish to implement *IntGen*. We notice that *IntSet* is represented by the *Vector els*, and the *Vector* class has a method that returns an iterator, so we could conceivably implement our method *elems()* like this:

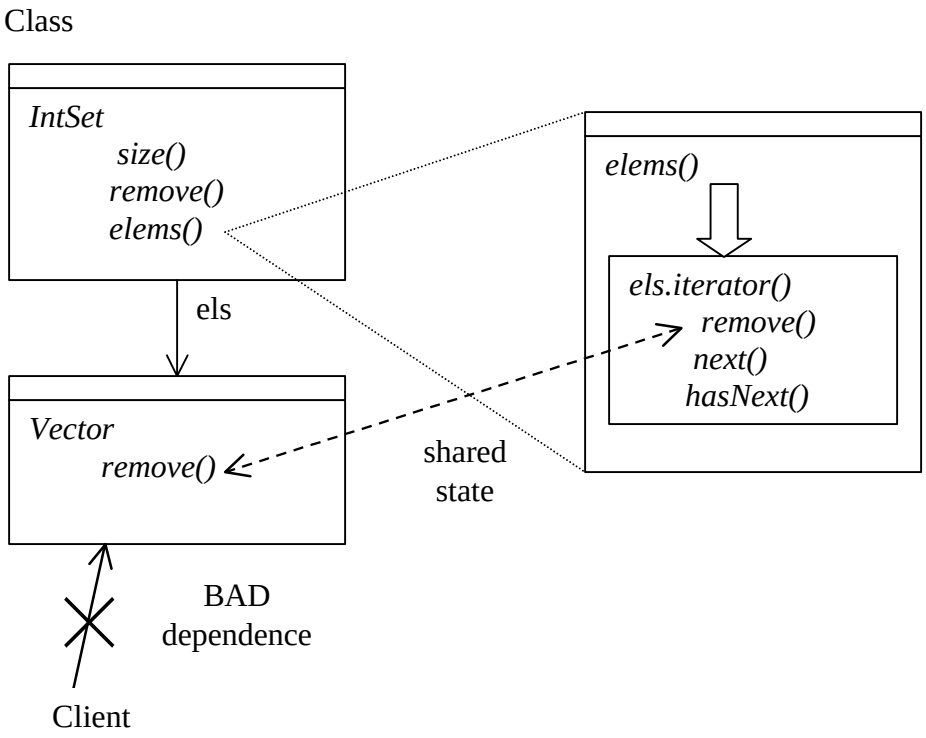
```
public class IntSet {

      ...
   public Iterator elems() {
      return els.iterator();  }
}
```

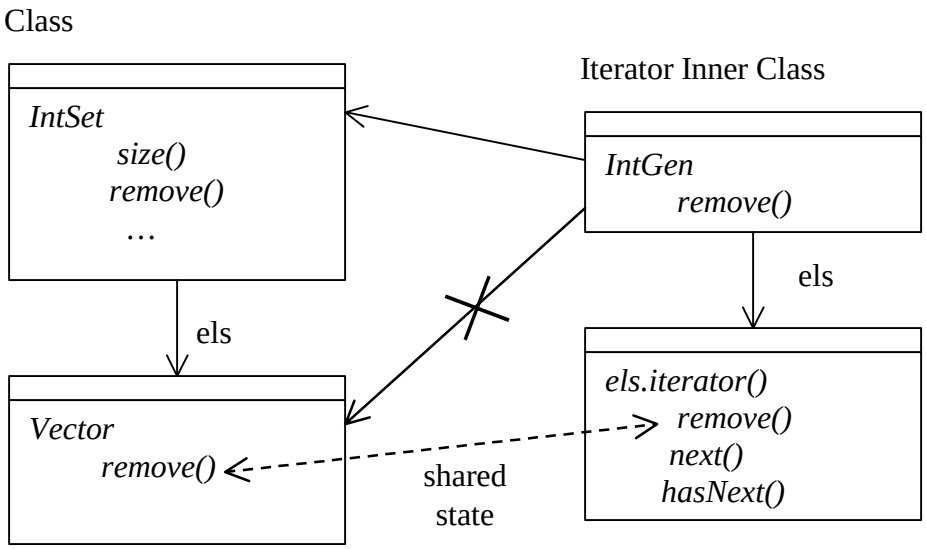
The returned generator *els.iterator()* provides the *next()*, *hasNext()* and *remove()* methods. This saves us a lot of work, but unfortunately causes a subtle form of rep exposure!

We have already discussed a simple form of rep exposure relating to the *remove()* methods in *IntSet* and *Vector*. *IntSet* implements a *remove()* method which may affect the *size()* method. The Vector *remove()* method does not know about the size of *IntSet*. So if a client calls the *Vector remove()* directly, then bad things can happen, e.g., *size* will be computed incorrectly.

Similarly, in the iterator class, if the client directly uses *g.remove()*, where *g* = *els.iterator(),* since there is shared state between the *els.iterator()* and the *Vector els*, bad things can happen. We summarize this pictorially below.



What should we do? We could obviously turn off *IntGen remove()* or not ever call it, but that is a cop-out. We need *IntGen* to implement the *remove()* method so it does the things that *IntSet remove()* does, and this is the only method that the client can call. *IntGen remove()* can call *g.remove()*, where *g = els.iterator(),* which manipulates the underlying representation while the iterator is being used. This is summarized pictorially below.



Note that implementing *IntGen remove()* by calling *Vector els.remove()* is also not a good idea, it might break the iterator with respect to the *next()* or *hasNext()* methods.

Lecture 8: Object Models & Invariants

This lecture consolidates many of the fundamental ideas of the previous lectures on objects, representations and abstraction. We will explain the graphical object modelling notation in detail and revisit representation invariants, abstraction functions and rep exposure. After reading this lecture, you may want to return to the previous lectures and look them over again, as they contain more details about the examples discussed here.

8.1 Object Models

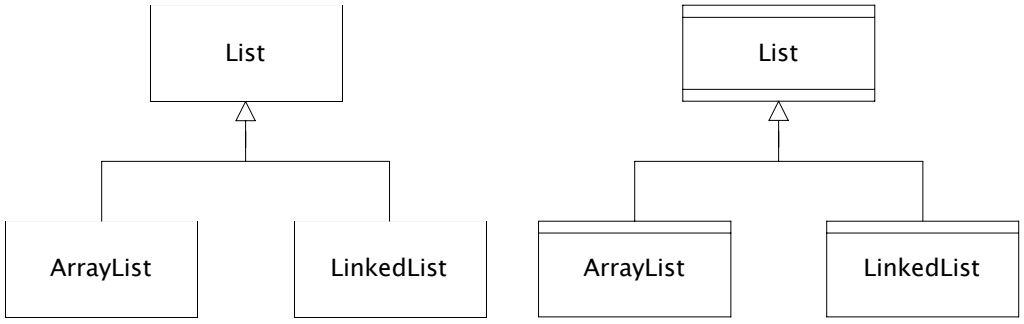
An *object model* is a description of a collection of configurations. In today's lecture, we'll look at object models of code, in which the configurations are states of a program. But we'll see later in the course that the same notation can be used more generally to describe any kind of configuration – such as the shape of a file system, a security hierarchy, a network topology, etc.

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

Object models and module dependency diagrams both contain boxes and arrows. The similarity ends there. Well, OK, I'll admit there are some subtle connections between the OM and MDD of a program. But at a first cut it's best to think of them as completely different. The MDD is about syntactic structure – what textual descriptions there are, and how they are related to one another. The OM is about semantic structure – what configurations are created at runtime and what properties they have.

8.1.1 Classification

An object model expresses two kinds of properties: classification of objects, and relationships between objects. To express classification, we draw a box for each class of objects. In an object model of code, these boxes will correspond to Java classes and interfaces; in a more general setting, they just represent arbitrary classifications.



An arrow with a fat, closed head from class *A* to class *B* indicates that *A* denotes a subset of *B*: that is, every *A* is also a *B*. To show that two boxes represent disjoint subsets, we have them share the same arrow head. In the diagram shown, *LinkedList* and *ArrayList* are disjoint subsets of *List*.

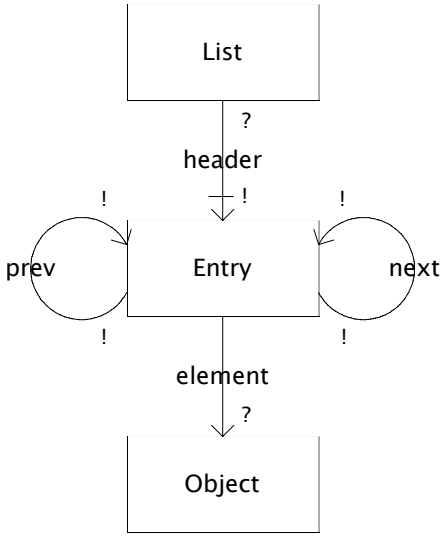
In Java, every *implements* and *extends* declaration results in a subset relationship in an object model. This is a property of the type system: if an object *o* is created with a constructor from class *C*, and *C* extends *D* say, then *o* is regarded as also having type *D*.

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

8.1.2 Fields

An arrow with an open head from *A* to *B* indicates a relationship between objects of *A* and objects of *B*. Because there may be many relationships between two classes, we name these relationships and label the arrows with the names. A field *f* in a class *A* whose type is *B* results in an arrow from *A* to *B* labelled *f*.

For example, the following code produces structures that can be illustrated by the dia-



gram shown overleaf (ignoring, for the moment, the markings on the ends of the arrows):

```
class LinkedList implements List {
    Entry header;
    ...
    }

class Entry {
    Entry next;
    Entry prev;
    Object elt;
    ...
    }
```

8.1.3 Multiplicity

So far, we have seen *classification* of objects into classes, and *relations* that show that objects in one class may be related to objects in another class. A fundamental question about a relation between classes is *multiplicity*: how many objects in one class can be related to a given object in another class.

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 field f from class A to class B, there are n members of class B associated by f with each A. It works the other way round too; if there is a marking m at the A end of a field f from A to B, each B is mapped to by m members of class A.

At the target end of the arrow – the end with the arrowhead – the multiplicity tells you how many objects an instance variable holds. For now, we have no use for * and +, but we'll see how they are used later for abstract fields. The choice of ? or ! depends on whether a field can be null or not.

At the source end of the arrow, the multiplicity tells you how many objects can point to a given object. In other words, it tells you about sharing. Let's look at some of the arrows and see what their multiplicities mean:

- For the field *header*, the ! on the target of the arrow says that every object in the class *List* is related to exactly one object in the class *Entry* by the field *header*. The ? on the source says that each *Entry* object is the header object of at most one *List*.
- For the field *element*, the ? on the target says that the *element* field of an *Entry* object points to zero or one objects in the class *Object*. In other words, it may be null: a *List* may store null references. The lack of a symbol on the source, says that an object may be pointed to by the *element* field of any number of *Entry* objects. In other words, a *List* may store duplicates.
- For the field *next*, the ! on the target and the source says that the *next* field of every *Entry* object points to one *Entry* object, and every *Entry* object is pointed to by the *next* field of one *Entry* object.

8.1.4 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, but for now we'll con-

sider only a limited form in which an optional bar may be marked crossing the target end of a field arrow. When present, this mark says for a given object, the object it is related to by the field must always be the same. In this case, we say the field is *immutable*, *static*, or more precisely *target static* (since later we'll give a meaning to a bar on the source end of the arrow).

In our diagram, for instance, the bar on the target end of the *header* relation says that a *List* object, once created, always points via its *header* field to the same *Entry* object.

An object is immutable if all its fields are immutable. A class is said to be immutable if its objects are immutable.

8.1.5 Instance Diagrams

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

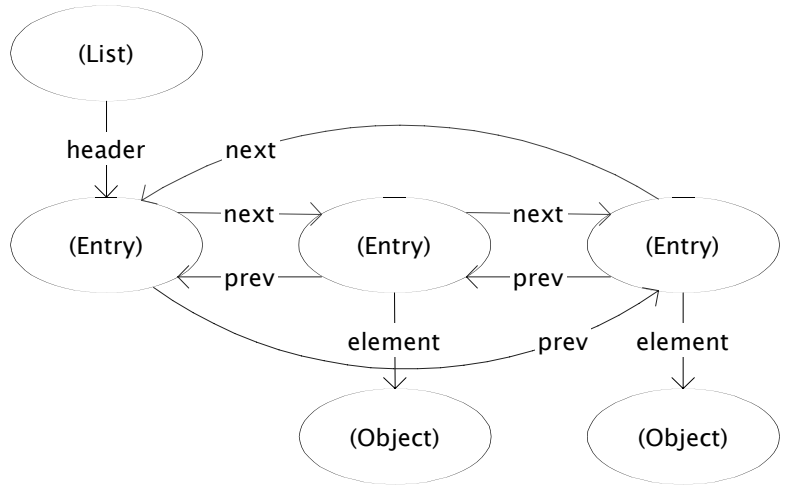
The relationship between a snapshot and an object model is just like the relationship between an object instance and a class, or the relationship between a sentence and a grammar.

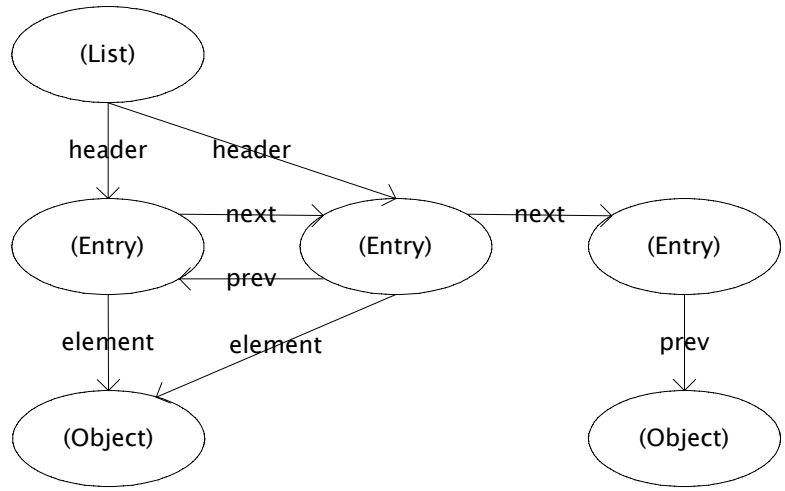
The figure below shows one legal snapshot (belonging to the collection denoted by the object model) and one illegal snapshot (not belonging). There are, of course, an infinite number of legal snapshots, since you can make a list of any length.

A useful exercise to check that you understand the meaning of the object model is to examine the illegal snapshot and determine which constraints it violates. The constraints are the multiplicity constraints and the constraints implicit in the placement of the arrows. For example, since the *header* field arrow goes from *List* to *Entry*, a snapshot containing a reference arrow labelled field from an *Entry* to an *Entry* must be wrong. Note that the mutability constraints are not relevant here; they tell you which transitions are permitted.

8.2 Whole Program Models

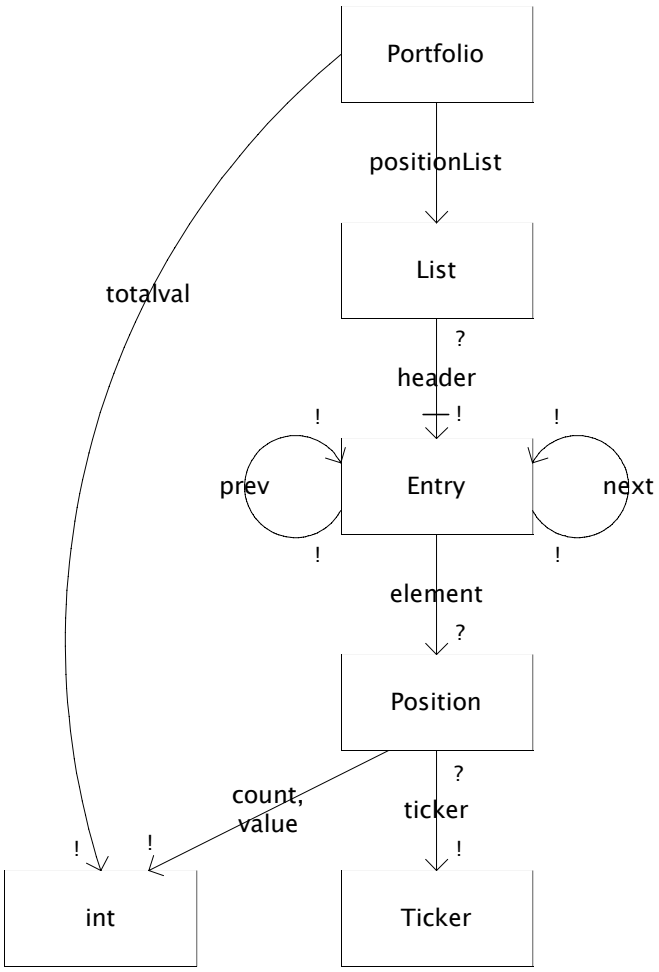
An object model can be used to show any portion of the state of a program. In the *List* example above, our object model showed only the objects involved in the representation of the *List* abstract type. But in fact, object models are most useful when they





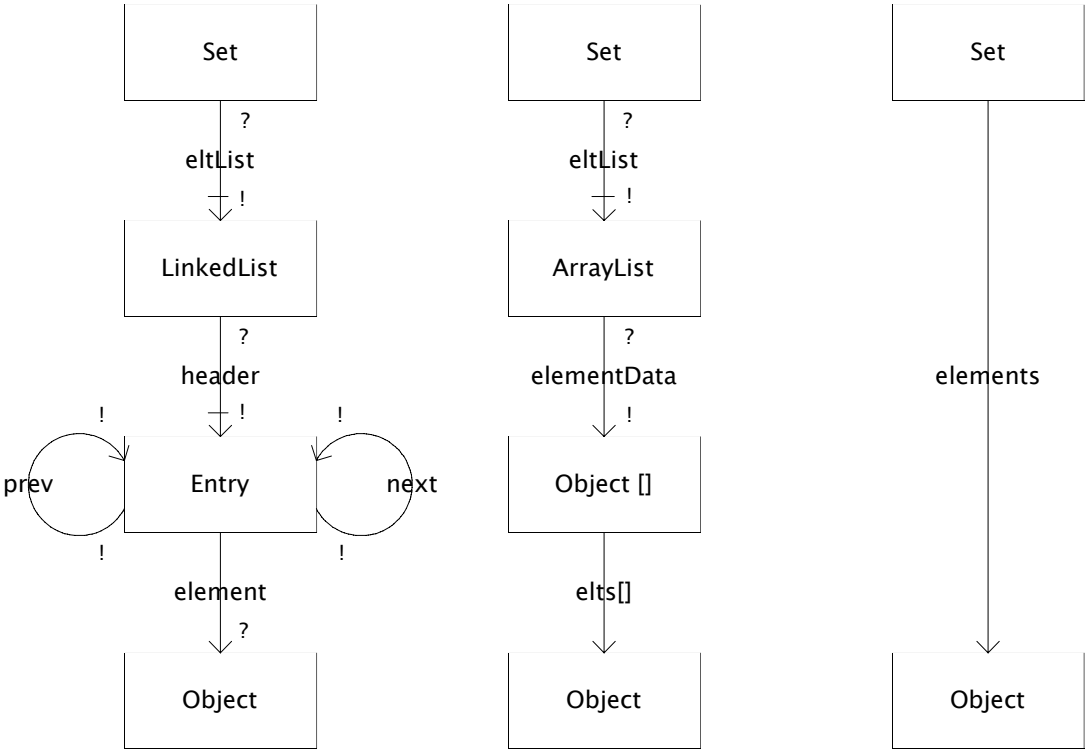
include the objects of many types, since they capture the web of relationships between objects which is often the essence of an object-oriented design.

Suppose, for example, we're building a program for tracking stock prices. We might design a *Portfolio* datatype that represents a stock portfolio. A *Portfolio* contains a list of *Position* objects, each holding a *Ticker* symbol for a stock, a count of the number of



shares held in that stock, and a current value for the stock. The *Portfolio* object also holds the total value of all the *Positions*.

The object model below shows this. Note how the *Entry* objects are now shown pointing to *Position* objects: they belong to a *List* of *Positions*, not an arbitrary list. We must allow several boxes in the same diagram with the label *List* that correspond to different kinds of *List*. And consequently, we have to be a bit careful about how we interpret the constraints implicit in a field arrow. The arrow marked *element* from *Entry* to *Position* in our diagram, for example, does not mean that every *Entry* object in the program points to a *Position* object, but rather that every *Entry* object contained in a *List*

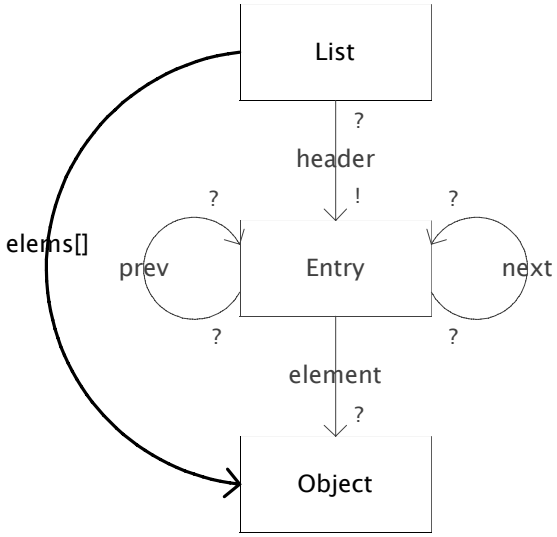


that is contained in a *Portfolio* points to a *Position*.

8.3 Abstract and Concrete Viewpoints

Suppose we want to implement a set abstract data type. In some circumstances – for example when we have a lot of very small sets – representing a set as a list is a reasonable choice. The figure overleaf shows three object models. The first two are two versions of a type *Set*, one represented with a *LinkedList* and one with an *ArrayList*. (Question for the astute reader: why is the *header* field in *LinkedList* immutable, but the *elementData* field in *ArrayList* is not?).

If our concern is how the *Set* is represented, we might want to show these object models. But if our concern is the role that *Set* plays in a larger program, and we don't want to be concerned about the choice of representation, we would prefer an object model that hides the difference between these two versions. The third object model, on the right hand side, is such a model. It replaces all the detail of the representation of *Set*

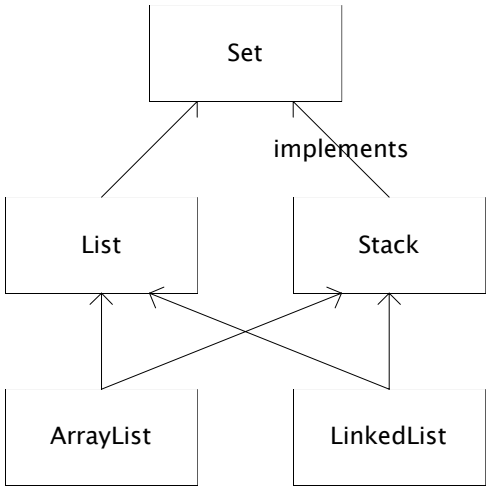


with a single field *elements* that connects *Set* objects directly to their elements. This field doesn't correspond to a field that is declared in Java in the *Set* class; it's an *abstract* or *specification* field.

There are therefore many object models that can be drawn for the same program. You can choose how much of the state to model, and for that part of the state, how abstractly to represent it. There is a particular level of abstraction that can claim to be normative, though. This is the level of abstraction that is presented by the methods in the code. For example, if some method of the class *Set* returns an object of type *LinkedList*, it would make little sense to abstract away the *LinkedList* class. But if, from the point of view of a client of *Set*, it is impossible to tell whether a *LinkedList* or *ArrayList* is being used, it would make sense to show the abstract field *element* instead.

An abstract type can be represented by many different representation types. Likewise, a type can be used to represent many different abstract types. A linked list can be used to implement a stack, for example: unlike the generic *List* interface, *LinkedList* offers *addLast* and *removeLast*. And, by design, *LinkedList* directly implements the *List* interface, which represents an abstract sequence of elements. We can therefore view the *LinkedList* class itself more abstractly with a field *elems*[] bypassing the internal entry structure, in which the [] indicate that the field denotes an indexed sequence.

The figure below shows these relationships: an arrow means "can be used to represent". The relationship is not symmetrical of course. The concrete type generally has more information content: a list can represent a set, but a set cannot represent a list, since it



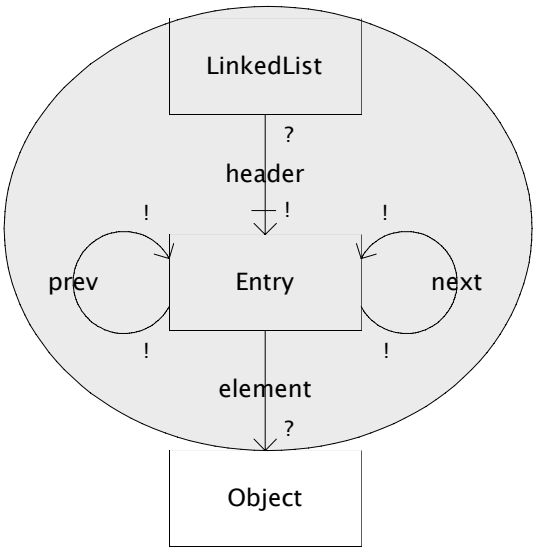
does not retain ordering information or allow duplicates. Note also that no type is inherently 'abstract' or 'concrete'. These notions are relative. A list is abstract with respect to a linked list used to represent it, but concrete with respect to a set it represents.

8.3.1 Abstraction Functions

For a particular choice of abstract and concrete type, we can show how the values of the concrete type are interpreted as abstract values using an abstraction function, as explained in an earlier lecture. Remember that the same concrete value can be interpreted in different ways, so the abstraction function is not determined by the choice of the abstract and concrete types. It's a record of a design decision, and determines how the code is written for the methods of the abstract data type.

In a language without mutable objects, in which we don't have to worry about sharing, we can think of the abstract and concrete 'values' as being just that – values. The abstraction function is then a straightforward mathematical function. Think, for example, of the various ways in which integers are represented as bitstrings. Each can be described as an abstraction function from bitstring to integer. An encoding that places the least significant bit first, for example, may have a function with mappings such as:

A (0000) = 0
...
A (0001) = 8
A (1001) = 9



…

But in an object-oriented program in which we have to worry about how mutations to an object via one path can affect a view of the object via another path, the 'values' are actually little subgraphs. The most straightforward way to define the abstraction function in these circumstances is to give a rule for each abstract field, explaining how it is obtained from concrete fields. For example, for the *LinkedList* representation of *Set*, we can write

*s.elements = s.list.header.*next.element*

to say that for each object *s* in the class, the objects pointed to by the abstract field *elements* are those obtained by following *list* (to the *List* object), *header* (to the first *Entry* object), then zero or more traversals of the *next* field (to the remaining *Entry* objects) and, for each of these, following the *element* field once (to the object pointed to by the *Entry*). Note that this rule is itself a kind of object model invariant: it tells you where it's legal to place arrows marked *elements* in a snapshot.

In general, an abstract type may have any number of abstract fields, and the abstraction function is specified by giving a rule for each one.

In practice, except for very subtle container types, abstraction functions are generally more trouble than they are worth. Understanding the idea of an abstraction function is valuable, however, because it helps you crystallize your understanding of data

abstraction. And you should be ready to write an abstraction function if the need arises. The *Boolean Formula in CNF* example of Lecture 6 is a good example of an abstract type that really needs an abstraction function. In that case, without a firm grasp of the abstraction function, it's hard to get the code right.

8.3.2 Representation Invariants

An object model is a kind of invariant: a constraint that always holds during the lifetime of a program. A representation invariant, or 'rep invariant', as discussed in Lecture 6, is a particular kind of invariant that describes whether the representation of an abstract object is well-formed. Some aspects of a rep invariant can be expressed in an object model. But there are other aspects that cannot be expressed graphically. And not all constraints of an object model are rep invariants.

A rep invariant is a constraint that can be applied to a single object of an abstract type, and tells you whether its representation is well formed. So it always involves exactly one object of the abstract type in question, and any objects in the representation reachable from it.

We can draw a contour around the part of the object model that a particular representation invariant can talk about. This contour groups the objects of a representation together with their abstract object. For example, for the rep invariant of the *LinkedList* viewed as a *List* (that is, an abstract sequence of elements), this contour includes the *Entry* elements. Not surprisingly, the classes within the contour are exactly those skipped over by the abstract field *elems*[]. And similarly, the rep invariant for the *ArrayList* covers the contained *Array*.

The details of the rep invariants were discussed in Lecture 6: for *LinkedList*, for example, they include such constraints as the entries forming a cycle, the *header* entry being always present and having a null *element* field, etc.

Let us recall why the rep invariant is useful – why it's not just a theoretical notion, but a practical tool:

- The rep invariant captures in one place the rules about how a legal value of the rep is formed. If you are modifying the code of an ADT, or writing a new method, you need to know what invariants have to be reestablished and which you can rely on. The rep invariant tells you everything you need to know; this is what is meant by *modular* reasoning. If there's no explicit rep invariant recorded, you have to read the code of every method!
- The rep invariant captures the essence of the design of the rep. The presence of the header entry and the cyclic form of the Java *LinkedList*, for example, are clever

design decisions that make the methods easy to code in a uniform way.

- As we shall see in a subsequent lecture, the rep invariant can be used to detect bugs at runtime in a form of 'defensive programming'.

8.3.3 Representation Exposure

The rep invariant provides modular reasoning so long as the rep is modified only within the abstract data type's class. If modifications by code outside the class are possible, one needs to examine the entire program to ensure that the rep invariant is maintained.

This unpleasant situation is called *rep exposure*. We've seen in earlier lectures some straightforward and more subtle examples. A simple example occurs when an abstract data type provides direct access to one of the objects within the rep invariant contour. For example, every implementation of the *List* interface (in fact the more general *Collection* interface) must provide a method

public Object [] toArray ()

which returns the list as an array of elements. The specification of this method says

> *The returned array will be "safe" in that no references to it are maintained by this collection. (In other words, this method must allocate a new array even if this collection is backed by an array). The caller is thus free to modify the returned array.*

In the *ArrayList* implementation, the method is implemented thus:

```
private Object elementData[];
...
public Object[] toArray() {
    Object[] result = new Object[size];
    System.arraycopy(elementData, 0, result, 0, size);
    return result;
    }
```

Note how the internal array is copied to produce the result. If instead, the array were returned immediately, like this

```
public Object[] toArray() {
    return elementData;
    }
```

we would have a rep exposure. Subsequent modifications to the array from outside the

abstract type would affect the representation inside. (In fact, in this case, there's such a weak rep invariant that a change to the array cannot break it, and this would produce only the rather odd effect of seeing the value of the abstract list change as the array is modified. But one could imagine a version of *ArrayList* that does not store null references; in this case, an assignment of null to an element of the array would break the invariant.)

Here's a much more subtle example. Suppose we implement an abstract data type for lists without duplicates, and we define the notion of duplication by the *equals* method of the elements. Now our rep invariant will say, for a linked list rep, for example, that no two distinct entries have elements that test true for equality. If the elements are mutable, and the *equals* method examines internal fields, it is possible that a mutation of an element will cause it to become equal to another one. So access to the elements themselves will constitute a rep exposure.

This is actually no different from the simple case, in the sense that the problem is access to an object within the contour. The invariant in this case, since it depends on the internal state of the elements, has a contour that includes the element objects. Equality creates particularly tricky issues; we'll pursue these further in tomorrow's lecture.

Lecture 9: Equality, Copying and Views

9.1 The Object Contract

Every class extends *Object*, and therefore inherits all of its methods. Two of these are particularly important and consequential in all programs, the method for testing equality:

public boolean equals (Object o)

and the method for generating a hash code:

public int hashCode ()

Like any other methods of a superclass, these methods can be overridden. We'll see in a later lecture on subtyping that a subclass should be a *subtype*. This means that it should behave according to the specification of the superclass, so that an object of the subclass can be placed in a context in which a superclass object is expected, and still behave appropriately.

The specification of the *Object* class is rather abstract and may seem abstruse. But failing to obey it has dire consequences, and tends to result in horrible obscure bugs. Worse, if you don't understand this specification and its ramifications, you are likely to introduce flaws in your code that have a pervasive effect and are hard to eliminate without major reworking. The specification of the *Object* class is so important that it is often referred to as 'The Object Contract'.

The contract can be found in the method specifications for *equals* and *hashCode* in the Java API documentation. It states that:

- *equals* must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive;
- *equals* must be consistent: repeated calls to the method must yield the same result unless the arguments are modified in between;
- for a non-null reference *x*, *x.equals (null)* should return false;
- *hashCode* must produce the same result for two objects that are deemed equal by the *equals* method;

9.2 Equality Properties

Let's look first at the properties of the *equals* method. Reflexivity means that an object always equals itself; symmetry means that when *a* equals *b*, *b* equals -a*; transitivity means that when* a *equals* b *and* b *equals* c*,* a *also equals* c*.*

These may seems like obvious properties, and indeed they are. If they did not hold, it's hard to imagine how the *equals* method would be used: you'd have to worry about whether to write *a.equals(b)* or *b.equals(a)*, for example, if it weren't symmetric.

What much less obvious, however, is how easy it is to break these properties inadvertently. The following example (taken from Joshua Bloch's excellent *Effective Java: Programming Language Guide*, one of the course recommended texts) shows how symmetry and transitivity can be broken in the presence of inheritance.

Consider a simple class that implements a two-dimensional point:

```
public class Point {
    private final int x;
    private final int y;
    public Point (int x, int y) {
        this.x = x; this.y = y;
        }
    public boolean equals (Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;
        }
    ... }
```

Now suppose we add the notion of a colour:

```
public class ColourPoint extends Point {
    private Colour colour;
    public ColourPoint (int x, int y, Colour colour) {
        super (x, y);
        this.colour = colour;
        }
    ...
}
```

What should the *equals* method of *ColourPoint* look like? We could just inherit *equals*

from *Point*, but then two *ColourPoints* will be deemed equal even if they have different colours. We could override it like this:

```
public boolean equals (Object o) {
    if (!(o instanceof ColourPoint))
        return false;
    ColourPoint cp = (ColourPoint) o;
    return super.equals (o) && cp.colour.equals(colour);
    }
```

This seemingly inoffensive method actually violates the requirement of symmetry. To see why, consider a point and a colour point:

```
Point p = new Point (1, 2);
ColourPoint cp = new ColourPoint (1, 2, Colour.RED);
```

Now *p.equals(cp)* will return true, but *cp.equals(p)* will return false! The problem is that these two expressions use different *equals* methods: the first uses the method from *Point*, which ignores colour, and the second uses the method from *ColourPoint*.

We could try and fix this by having the equals method of *ColourPoint* ignore colour when comparing to a non-colour point:

```
public boolean equals (Object o) {
    if (!(o instanceof Point))
        return false;
    // if o is a normal Point, do colour-blind comparison
    if (!(o instanceof ColourPoint))
        return o.equals (this);
    ColourPoint cp = (ColourPoint) o;
    return super.equals (o) && cp.colour.equals (colour);
    }
```

This solves the symmetry problem, but now equality isn't transitive! To see why, consider constructing these points:

```
ColourPoint p1 = new ColourPoint (1, 2, Colour.RED);
Point p2 = new Point (1, 2);
ColourPoint p2 = new ColourPoint (1, 2, Colour.BLUE);
```

The calls *p1.equals(p2)* and *p2.equals(p3)* will both return true, but *p1.equals(p3)* will return false.

It turns out that there is no solution to this problem: it's a fundamental problem of inheritance. You can't write a good *equals* method for *ColourPoint* if it inherits from *Point*. However, if you implement *ColourPoint* using *Point* in its representation, so that a *ColourPoint* is no longer treated as a *Point*, the problem goes away. See Bloch's book for details.

Bloch's book also gives some hints on how to write a good *equals* method, and he points out some common pitfalls. For example, what happens if you write something like this

public boolean equals (Point p)

substituting another type for *Object* in the declaration of *equals*?

9.3 Hashing

To understand the part of the contract relating to the *hashCode* method, you'll need to have some idea of how hash tables work.

Hashtables are a fantastic invention – one of the best ideas of computer science. A hashtable is a representation for a mapping: an abstract data type that maps keys to values. Hashtables offer constant time lookup, so they tend to perform better than trees or lists. Keys don't have to be ordered, or have any particular property, except for offering *equals* and *hashCode*.

Here's how a hashtable works. It contains an array that is initialized to a size corresponding the number of elements that we expect to be inserted. When a *key* and a *value* are presented for insertion, we compute the hashcode of the key, and convert it into an index in the array's range (eg, by a modulo division). The value is then inserted at that index.

The rep invariant of a hash table includes the fundamental constraint that keys are in the slots determined by their hash codes. We'll see later why this is important.

Hashcodes are designed so that the keys will be spread evenly over the indices. But occasionally a *conflict* occurs, and two keys are placed at the same index. So rather than holding a single value at an index, a hashtable actually holds a list of key/value pairs (usually called 'hash buckets'), implemented in Java as objects from class with two fields. On insertion, you add a pair to the list in the array slot determined by the hash code. For lookup, you hash the key, find the right slot, and then examine each of the pairs until one is found whose key matches the given key.

Now it should be clear why the *Object* contract requires equal objects to have the same

hash key. If two equal objects had distinct hash keys, they might be placed in different slots. So if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may fail.

A simple and drastic way to ensure that the contract is met is for *hashCode* to always return some constant value, so every object's hash code is the same. This satisfies the *Object* contract, but it would have a disastrous performance effect, since every key will be stored in the same slot, and every lookup will degenerate to a linear search along a long list.

The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the determination of equality (usually by calling the *hashCode* method of each component), and then combining these, throwing in a few arithmetic operations. Look at Bloch's book for details.

Most crucially, note that if you don't override *hashCode* at all, you'll get the one from *Object*, which is based on the address of the object. If you have overridden *equals*, this will mean that you will have almost certainly violated the contract. So as a general rule:

Always override hashCode when you override equals.

(This is one of Bloch's aphorisms. The whole book is a collection of aphorisms like it, each nicely explained and illustrated.)

Last year, a student spent hours tracking down a bug in a project that amounted to nothing more than mispelling *hashCode* as *hashcode*. This created a method that didn't override the *hashCode* method of *Object* at all, and strange things happened. Another reason to avoid inheritance…

9.4 Copying

The need to make a copy of an object often arises. For example, you may want to do a computation that requires modifying the object but without affecting objects that already hold references to it. Or you may have a 'prototype' object that you want to make a collection of objects from that differ in small ways, and it's convenient to make copies and then modify them.

People sometimes talk about 'shallow' and 'deep' copies. A shallow copy of an object is made by creating a new object whose fields point to the same objects as the old object. A deep copy is made by creating a new object also for the objects pointed to by the fields, and perhaps for the objects they point to, and so on.

How should copying be done? If you've diligently studied the Java API, you may assume that you should use the *clone* method of *Object*, along with the *Cloneable* interface. This is tempting, because *Cloneable* is a special kind of 'marker interface' that adds functionality magically to a class. Unfortunately, though, the design of this part of Java isn't quite right, and it's very difficult to use it well. So I recommend that you don't use it at all, unless you have to (eg, because code you're using requires your class to implement *Cloneable*, or because your manager hasn't taken 6170). See Bloch's book for an insightful discussion of the problems.

You might think it would be fine to declare a method like this:

```
class Point {
    Point copy () {
    ...
        }
    ...
    }
```

Note the return type: copying a point should result in a point. Now in a subclass, you'd like the *copy* method to return a subclass object:

```
class ColourPoint extends Point {
    ColourPoint copy () {
    ...
        }
    ...
    }
```

Unfortunately, this is not legal in Java. You can't change the return type of a method when you override it in a subclass. And overloading of method names uses only the types of the arguments. So you'd be forced to declare both methods like this:

```
Object copy ()
```

and this is a nuisance, because you'll have to downcast the result. But it's workable and sometimes the right thing to do.

There are two other ways to do copying. One is to use a static method called a 'factory' method because it creates new objects:

```
public static Point newPoint (Point p)
```

The other is to provide additional constructors, usually called 'copy constructors':

public Point (Point p)

Both of these work nicely, although they're not perfect. You can't put static methods or constructors in an interface, so they're not when you're trying to provide generic functionality. The copy constructor approach is widely used in the Java API. A nice feature of this approach is that it allows the client to choose the class of the object to be created. The argument to the copy constructor is often declared to have the type of an interface so that you can pass the constructor any type of object that implements the interface. All of Java's collection classes, for example, provide a copy constructor that takes an argument of type *Collection* or *Map*. If you want to create an array list from a linked list *l*, for example, you would just call

new ArrayList (l)

9.5 Element and Container Equality

When are two containers equal? If they are immutable, they should be equal if they contain the same elements. For example, two strings should be equal if they contain the same characters (in the same order). Otherwise, if we just kept the default *Object* equality method, a string entered at the keyboard, for example, would never match a string in a list or table, because it would be a new string object and therefore not the same object as any other. And indeed, this is exactly how *equals* is implemented in the Java *String* class, and if you want to see if two strings *s1* and *s2* contain the same character sequence, you should write

s1.equals (s2)

and not

s1 == s2

which will return false when *s1* and *s2* denote different string objects that contain the same character sequences.

9.5.1 The Problem

So much for strings – sequences of characters. Let's consider lists now, which are sequences of arbitrary objects. Should they be treated the same way, so that two lists are equal if they contain the same elements in the same order?

Suppose I'm planning a party at which my friends will sit at several different tables, and

I've written a program to help me create a seating plan. I represent each table as a list of friends, and the party as a whole as a set of these lists. The program starts by creating empty lists for the tables and inserting them into the set:

```
List t1 = new LinkedList ();
List t2 = new LinkedList ();
...
Set s = new HashSet ();
s.add (t1);
s.add (t2);
...
```

At some later point, the program will add friends to the various lists; it may also create new lists and replace existing lists in the set with them. Finally, it iterates over the contents of the set, printing out each list.

This program will fail, because the initial insertions will not have the expected effect. Even though the empty lists represent conceptually distinct table plans, they will be equal according to the *equals* method of *LinkedList*. Since *Set* uses the *equals* method on its elements to reject duplicates, all insertions but the first will have no effect, since all of the empty lists will be deemed duplicates.

How can we solve this problem? You might think that *Set* should have used == to check for duplicates instead, so that an object is regarded as a duplicate only if that very object is already in the set. But that wouldn't work for strings; it would mean that after

```
Set set = new HashSet ();
String lower = "hello";
String upper = "HELLO";
set.add (lower.toUpperCase());
...
```

the test *set.contains (upper)* would evaluate to false, since the *toUpperCase* method creates a new string.

9.5.2 The Liskov Solution

In our course text, Professor Liskov presents a systematic solution to this problem. You provide two distinct methods: *equals*, which returns true when two objects in a class are behaviourally equivalently, and *similar*, which returns true when two objects are observationally equivalent.

Here's the difference. Two objects are *behaviourally equivalent* if there is no sequence of operations that can distinguish them. On these grounds, the empty lists *t1* and *t2* from above are not equivalent, since if you insert an element into one, you can see that the other doesn't change. But two distinct strings that contain the same sequence of characters are equivalent, since you can't modify them and thus discover that they are different objects. (We're assuming you're not allowed to use == in this experiment.)

Two objects are *observationally equivalent* if you can't tell the difference between them using observer operations (and no mutations). On these grounds, the empty lists *t1* and *t2* from above are equivalent, since they have the same size, contain the same elements, etc. And two distinct strings that contain the same sequence of characters are also equivalent.

Here's how you code *equals* and *similar*. For a mutable type, you simply inherit the *equals* method from *Object*, but you write a *similar* method that performs a field-by-field comparison. For an immutable type, you override *equals* with a method that performs a field-by-field comparison, and have *similar* call *equals* so that they are the same.

This solution, when applied uniformly, is easy to understand and works well. But it's not always ideal. Suppose you want to write some code that interns objects. This means mutating a data structure so that references to objects that are structurally identical become references to the very same object. This is often done in compilers; the objects might be program variables, for example, and you want to have all references to a particular variable in the abstract syntax tree point to the same object, so that any information you store about the variable (by mutating the object) is effectively propagated to all sites in which it appears.

To do the interning, you might try to use a hash table. Every time you encounter a new object in the data structure, you look that object up in the table to see if it has a canonical representative. If it does, you replace it by the representative; otherwise you insert it as both key and value into the table.

Under the Liskov approach, this strategy would fail, because the equality test on the keys of the table would never find a match for distinct objects that are structurally equivalent, since the *equals* method of a mutable object only returns true on the very same object.

9.5.3 The Java Approach

For reasons such as this, the designer of the Java collections API did *not* follow this approach. There is no *similar* method, and *equals* is observational equivalence.

This has some convenient consequences. The interning table will work, for example. But it also has some unfortunate consequences. The seating plan program will break, because two distinct empty lists will be deemed equal.

In the Java *List* specification, two lists are equal not only if they contain the same elements in the same order, but also if they contain *equal* elements in the same order. In other words, the *equals* method is called recursively. To maintain the *Object* contract, the *hashCode* method is also called recursively on the elements. This results in a very nasty surprise. The following code, in which a list is inserted into itself, will actually fail to terminate!

```
List l = new LinkedList ();
l.add (l);
int h = l.hashCode ();
```

This is why you'll find warnings in the Java API documentation about inserting containers into themselves, such as this comment in the specification of *List*:

> *Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list.*

There are some other, even more subtle, consequences of the Java approach to do with rep exposure which are explained below.

This leaves you with two choices, both of which are acceptable in 6170:

- You can follow the Java approach, in which case you'll get the benefits of its convenience, but you'll have to deal with the complications that can arise.
- Alternatively, you can follow the Liskov approach, but in that case you'll need to figure out how to incorporate into your code the Java collection classes (such as *LinkedList* and *HashSet*).

In general, when you have to incorporate a class whose *equals* method follows a different approach from the program as a whole, you can write a wrapper around the class that replaces the *equals* method with a more suitable one. The course text gives an example of how to do this.

9.6 Rep Exposure

Let's revisit the example of rep exposure that we closed yesterday's lecture with. We imagined a variant of *LinkedList* for representing sequences without duplicates. The *add* operation has a new specification saying that the element is added only if it isn't a duplicate, and its code performs this check:

```
void add (Object o) {
    if (contains (o))
        return;
    else
        // add the element
        ...
    }
```

We record the rep invariant at the top of the file saying that the list contains no duplicates:

> *The list contains no duplicates. That is, there are no distinct entries e1 and e2 such that e1.element.equals (e2.element).*

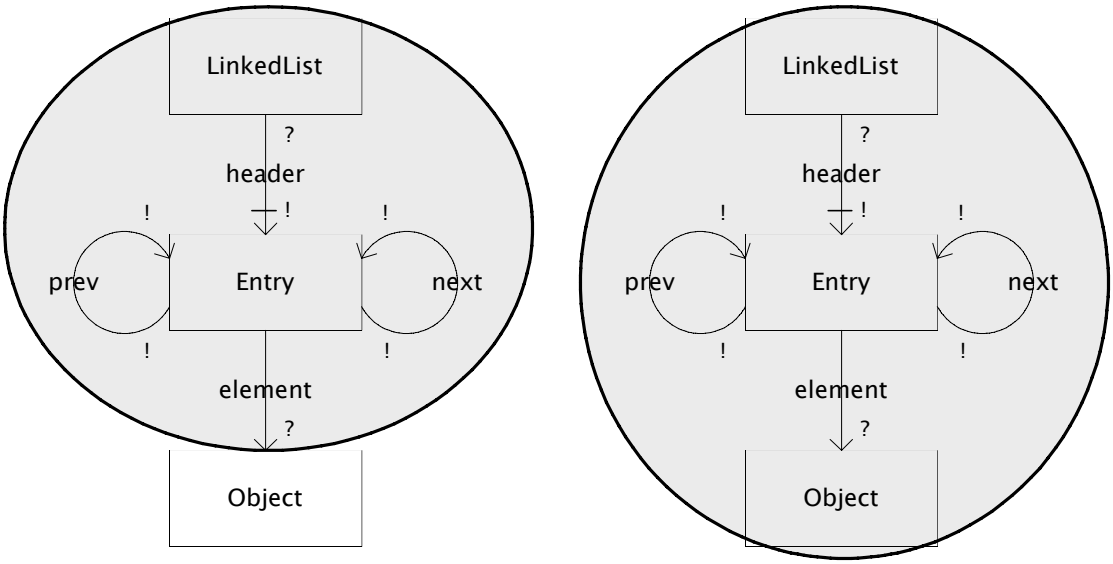
We check that it's preserved, by ensuring that every method that adds an element first performs the containment check.

Unfortunately, this isn't good enough. Watch what happens if we make a list of lists and then mutate an element list:

```
List x = new LinkedList ();
List y = new LinkedList ();
Object o = new Object ();
x.add (o);
List p = new LinkedList ();
p.add (x);
p.add (y);
x.remove (o);
```

After this code sequence, the rep invariant of p is broken. The problem is that the mutation to x makes it equal to y, since they are then both empty lists.

What's going on here? The contour that we drew around the representation actually includes the element class, since the rep invariant depends on a property of the element (see figure). Note that this problem would not have arisen if equality had been determined by the Liskov approach, since two mutable elements would be equal only if they were the very same object: the contour extends only to the reference to the element, and not to the element itself.



9.6.1 Mutating Hash Keys

A more common and insidious example of this phenomenon occurs with hash keys. If you mutate an object after it has been inserted as a key in a hash table, its hash code may change. As a result, the crucial rep invariant of the hash table – that keys are in the slots determined by their hash codes – is broken.

Here's an example. A hash set is a set implemented with a hash table: think of it as a hash table with keys and no values. If we insert an empty list into a hash set, and then add an element to the list like this:

Set s = new HashSet ();
List x = new LinkedList ();
s.add (x);
x.add (new Object ());

a subsequent call *s.contains(x)* is likely to return *false*. If you think that's acceptable, consider the fact that there may now be *no value of x* for which *s.contains(x)* returns true, even though *s.size()* will return 1!

Again, the problem is rep exposure: the contour around the hash table includes the keys.

The lesson from this is: either follow the Liskov approach, and wrap the Java list to

override its *equals* method, or make sure that you never mutate hash keys, or allow any mutation of an element of a container that might break the container's rep invariant. This is why you'll see comments such as this in the Java API specification:

> *Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.*

9.7 Views

An increasingly common idiom in object-oriented program is to have distinct objects that offer different kinds of access to the same underlying data structure. Such objects are called *views*. Usually one object is thought of as primary, and another as secondary. The primary one is called the 'underlying' or 'backing' object, and the secondary one is called the 'view'.

We are used to aliasing, when two object references point to the same object, so that a change under one name appears as a change under the other:

```
List x = new LinkedList ();
List y = x;
y.add (o); // changes y also
```

Views are tricky because they involve a subtle form of aliasing, in which the two objects have distinct types. We have seen an example of this with iterators, whose *remove* method of an iterator removes the last yielded element from the underlying collection:

```
List x = new LinkedList ();
...
Iterator i = x.iterator ();
while (i.hasNext ()) {
    Object o = i.next ();
    ...
    i.remove (); // mutates x also
    }
```

An iterator is this a view on the underlying collection. Here are two other examples of views in the Java collections API.

- Implementations of the *Map* interface are required to have a method *keySet* which returns the set of keys in the map. This set is a view; as the underlying map changes, the set will change accordingly. Unlike an iterator, this view and the underlying map can be both be modified; a deletion of a key from the set will cause the key and its value to be deleted from the map. The set does not support an *add* operation, since it would not make sense to add a key without a value. (This, by the way, is why *add* and *remove* are optional methods in the *Set* interface.)
- *List* has a method *subList* which returns a view of part of a list. It can be used to access the list with an offset, eliminating the need for explicit range operations. Any operation that expects a list can be used as a range operation by passing a *subList* view instead of a whole list. For example, the following idiom removes a range of elements from a list:

  list.subList(from, to).clear();

Ideally, a view and its backing object should both be modifiable, with effects propagated as expected between the two. Unfortunately, this is not always possible, and many views place constraints on what kinds of modification are possible. An iterator, for example, becomes invalid if the underlying collection is modified during iteration. And a sublist is invalidated by certain structural modifications to the underlying list.

Things get even trickier when there are several views on the same object. For example, if you have two iterators simultaneously on the same underlying collection, a modification through one iterator (by a call to *remove*) will invalidate the other iterator (but not the collection).

9.8 Summary

Issues of copying, views and equality show the power of object-oriented programming but also its pitfalls. You must have a systematic and uniform treatment of equality, hashing and copying in any program you write. Views are a very useful mechanism, but they must be handled carefully. Constructing an object model of your program is useful because it will remind you of where sharings occur and cause you to examine each case carefully.

Lecture 10: Dynamic Analysis, Part 1

The best way to ensure the quality of the software you build is to design it carefully from the start. The parts will fit together more cleanly, and the functionality of each part will be simpler, so you'll make fewer errors implementing it. But it's hard not to introduce some errors during coding, and an effective way to find these is to use *dynamic* techniques: that is, those that involve executing the program and observing its behaviour. In contrast, *static* techniques are ones that you use to ensure quality before you execute: by evaluating the design and by analyzing the code (either by manual review, or by using tools such as a type checker).

Some people mistakenly rely on dynamic techniques, rushing through specification and design on the assumption that they can fix things up later. There are two problems with this approach. The first is that problems in the design get enmeshed, by implementation time, with implementation problems, so they are harder to find. The second is that the cost of fixing an error in a software artifact is known to increase dramatically the later in development it is discovered. In some early studies at IBM and TRW, Barry Boehm discovered that a specification error can cost 1000 times more to fix if not discovered until implementation!

Other people mistakenly imagine that only static techniques are necessary. Although great strides have been made in technology for static analysis, we are still far from being able to catch all errors statically. Even if you have constructed a mathematical proof that your program is correct, you would be foolish not to test it.

The fundamental problem with testing is expressed in a famous aphorism of Dijkstra's:

> *Testing can reveal the presence of errors but never their absence.*

Testing, by its very nature, is incomplete. You should be very wary of making any assumptions about the reliability of a program just because it has passed a large battery of tests. In fact, the problem of determining when a piece of software is sufficiently reliable to release is one that plagues managers, and for which very little guidance exists. It is therefore best to think of testing not as a way to establish confidence that the program is right, but rather as a way to find errors. There's a subtle but vitally important difference between these viewpoints.

10.1 Defensive Programming

Defensive programming is an approach to increasing the reliability of a program by inserting redundant checks. Here's how it works. When you're writing some code, you figure out conditions that you expect to hold at certain points in the code – invariants, in other words. Then, rather than just assuming that these invariants hold, you test them explicitly. These tests are called *runtime assertions*. If an assertion fails – that is, the invariant evaluates to false – you report the error and abort the computation.

10.1.1 Guidelines

How should you use runtime assertions? 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 – that is, to check the precondition. This makes sense because a high proportion of errors are related to misunderstandings about interfaces between procedures.
- At the end of a complicated procedure, to check that the result is plausible – that is, to check the postcondition. 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 would turn an operation that executes in constant time into one that takes linear time (in the length of the array).

10.1.2 Catching Common Exceptions

Because Java is a safe language, its runtime environment – the Java Virtual Machine (JVM) – already includes runtime assertions for several important classes of error:

- Calling a method on a null object reference;
- Accessing an array out of bounds;
- Performing an invalid downcast.

These errors cause unchecked exceptions to be thrown. Moreover, the classes of the Java API themselves throw exceptions for erroneous conditions.

It is good practice to catch all these exceptions. A simple way to do this is to write a handler at the top of the program, in the main method, that terminates the program appropriately (eg, with an error message to the user, and perhaps attempting to close open files).

Note that there are some exceptions that are thrown by the JVM that you should no try to handle. Stack overflows and out-of-memory errors, for example, indicate that the program has run out of resources. In these circumstances, there's no point making things worse by trying to do more computation.

10.1.3 Checking the Rep Invariant

A very useful strategy for finding errors in an abstract type with a complex representation is to encode the rep invariant as a runtime assertion. The best way to do this is to write a method

public void checkRep ()

that throws an unchecked exception if the invariant does not hold at the point of call. This method can be inserted in the code of the data type, or called in a testbed from the outside.

Checking the rep invariant is much more powerful than checking most other kinds of invariants, because a broken rep often only results in a problem long after it was broken. With *checkRep*, you are likely to find catch the error much closer to its source. You should probably call *checkRep* at the start and end of every method, in case there is a rep exposure that causes the rep to be broken between calls. You should also remember to instrument observers, since they may mutate the rep (as a benevolent side effect).

There is an extensive literature on runtime assertions, but interestingly, there seems to be very little knowledge in industry of how to use *repCheck*.

Usually, checking the rep invariant will be too computationally intensive for the kind of runtime assertion that you would leave in production code. So you should use *checkRep* primarily in testing. For production checks, you should consider at which points the code may fail because of a broken representation invariant, and insert appropriate checks there.

10.1.4 Assert Framework

Runtime assertions can clutter up the code. It's especially bad if a reader can't easily tell which parts of the code are assertions and which are doing the actual computation. For this reason, and to make the writing of assertions more systematic and less burdensome, it's a good idea to implement a small framework for assertions.

Some programming languages, such as Eiffel, come with assertion mechanisms built-in. And requests for such a mechanism have topped all other requests for changes to Java. There are also many 3rd party tools and frameworks for adding assertions to code and for controlling them.

In practice, though, it's easy to build a small framework yourself. One approach is to implement a class, *Assert* say, with a method

public static void assert (boolean b, String location)

that throws an unchecked exception when the argument is false, containing a string indicating the location of the failed assertion. This class can encapsulate logging and error reporting. To use it, one simply writes assertions like this:

Assert.assert (x != null, "MyClass.MyMethod");

It's also possible to use Java's reflection mechanism to mitigate the need to provide location information.

10.1.5 Assertions in Subclasses

When we study subtyping, we'll see how the pre- and post-conditions of a subclass should be related to the pre- and post-conditions of its superclass in certain ways. This suggests opportunities for additional runtime checking, and also for subclasses to reuse assertion code from superclasses.

Surprisingly, most approaches to checking assertions in subclasses have been conceptually flawed. These recent papers explains why this is, and shows how to develop an assertion framework for subclasses:

- Robby Findler, Mario Latendresse, and Matthias Felleisen. Behavioral Contracts and Behavioral Subtyping. *Foundations of Software Engineering*, 2001.
- Robby Findler and Matthias Felleisen. Contract Soundness for Object-Oriented Languages. *Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

see http:
www.cs.rice.edu/~robby/publications/.

10.1.6 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).

Lecture 11: Dynamic Analysis, Part 2

In this lecture, we continue our discussion of dynamic analysis, focusing on testing. We take a brief look at some of the basic notions underlying the theory of testing, and survey the techniques most widely used in practice. At the end, we collect together some practical guidance to help you in your own testing work.

11.1 Testing

Testing is much more effective, and much less painful, if you approach it systematically. Before you start, think about:

- what properties you want to test for;
- what modules you want to test, and what order you'll test them in;
- how you'll generate test cases;
- how you'll check the results;
- when you'll know that you're done.

Deciding what properties to test for will require knowledge of the problem domain, to understand what kinds of failures will be most serious, and knowledge of the program, to understand how hard different kinds of errors will be to catch.

Choosing modules is more straightforward. You should test especially those modules that are critical, complex, or written by your team's sloppiest programmer (or the one who likes clever tricks most). Or perhaps the module that wasn't written latest at night, or just before the release…

The module dependency diagram helps determine the order. If your module depends on a module that isn't yet implemented, you'll need to write a *stub* that stands in for the module during testing. The stub provides enough behaviour for the tests at hand. It might, for example, look up answers in a table rather doing a computation.

Checking results can be hard. Some programs – such as the Foliotracker you'll be building in exercises 5 and 6 – don't even have repeatable behaviour. For others, the results are only the tip of the iceberg, and to check that things are really working, you'll need to check internal structures.

Later on we'll discuss the questions of how to generate test cases and how to know when you're done.

11.2 Regression Tests

It's very important to be able to rerun your tests when you modify your code. For this reason, it's a bad idea to do ad hoc testing that can't be repeated. It may seem like a lot of work, but in the long run, it's much less work to construct a suite of tests that can be reexecuted from a file. Such tests are called *regression tests*.

An approach to testing that goes by the name of *test first programming*, and which is part of the new development doctrine called *extreme programming*, encourages construction of regression tests even before any application code is written. JUnit, the testing framework that you've been using, was designed for this.

Regression testing of a large system is a major enterprise. It can take a week of elapsed time just to run the test scripts. So an area of current research interest is trying to determine which regression test cases can be omitted. If you know which cases test which parts of the code, you may be able to determine that a local change in one part of the code does not require that all the cases be rerun.

11.3 Criteria

To understand how tests are generated and evaluated, it helps to take a step back and think abstractly about the purpose and nature of testing.

Suppose we have a program P that is supposed to meet a specification S. We'll assume for simplicity that P is a function from inputs to outputs, and S is a function that takes an input and an output and returns a boolean. Our aim in testing is to find a test case t such that

S (t, P(t))

is false: that is, P produces a result for the input t that is not permitted by S. We will call t a *failing test case*, although of course it's really a successful one, since our aim is to find errors!

A test suite T is a set of test cases. When is a suite 'good enough'? Rather than attempting to evaluate each suite in a situation-dependent way, we can apply general *criteria*. You can think of a criterion as a function

C: Suite, Program, Spec → Boolean

that takes a test suite, a program, and a specification and returns true or false according to whether, in some systematic sense, the suite is good enough for the given program and specification.

Most criteria don't involve both the program and the specification. A criterion that involves only the program is called a *progam-based* criterion. People also use terms like 'whitebox', 'clearbox', 'glassbox', or 'structural' testing to describe testing that uses program-based criteria.

A criterion that involves only the specification is called a *specification-based* criterion. The terms 'blackbox' testing is used in association with it, to suggest that the tests are judged without being able to see inside the program. You might also hear the term 'functional' testing.

11.4 Subdomains

Practical criteria tend to have a particular structure and properties. They don't, for example, accept a test suite *T* but reject a test suite *T'* that is just like *T* but has some extra test cases in it. They also tend to be insensitive to what combinations of test cases are chosen. These aren't necessarily good properties; they just arise from the simple way in which most criteria are defined.

The input space is divided into regions usually called *subdomains*, each containing a set of inputs. The subdomains together exhaust the input space – that is, every input is in at least one subdomain. A division of the input space into subdomains defines implicitly a criterion. The criterion is that there be at least one test case from each subdomain. Subdomains are not usually disjoint, so a single test case may be in several subdomains.

The intuition behind subdomains is two-fold. First, it's an easy way (at least conceptually) to determine if a test suite is good enough. Second, we hope that by requiring a case from each subdomain, we will drive testing into regions of the input space most likely to find bugs. Intuitively, each subdomain represents a set of similar test cases; we want to maximize the benefit of our testing by picking dissimilar test cases – that is, test cases from different subdomains.

In the best case, a subdomain is *revealing*. This means that every test case in it either causes the program to fail or to succeed. The subdomain thus groups together truly equivalent cases. If all subdomains are revealing, a test suite that satisfies the criterion will be complete, since we are guaranteed that it will find any bug. In practice, it's very hard to get revealing subdomains though, but by careful choice of subdomains it's possible to have at least some subdomains whose error rate – the proportion of inputs that lead to bad outputs – is much higher than the average error rate for the input space as a whole.

11.5 Subdomain Criteria

The standard and most widely used criterion for program-based testing is *statement coverage*: that every statement in the program must be executed at least once. You can see why this is a subdomain criterion: define for each program statement the set of inputs that causes it to be executed, and pick at least one test case for each subdomain. Of course the subdomain is never explicitly constructed; it's a conceptual notion. Instead, one typically runs an instrumented version of the program that logs which statements are executed. You keep adding test cases until all statements are logged as executed.

There are more burdensome criteria than statement coverage. *Decision coverage* requires that every edge in the control flow graph of the program be executed – roughly that every branch in the program be executed both ways. It's not immediately obvious why this is more stringent that statement coverage. Consider applying these criteria to a procedure that returns the minimum of two numbers:

```
static int minimum (int a, int b) {
    if (a ≤ b)
        return a;
    else
        return b;
```

For this code, statement coverage will require inputs with a less than b and vice versa. However, for the code

```
static int minimum (int a, int b) {
    int result = b; if (b ≤ a)
        result = b;
    return result;
```

a single test case with b less than a will produce statement coverage, and the bug will be missed. Decision coverage would require a case in which the if-branch is not executed, thus exposing the bug.

There are many forms of *condition coverage* that require, in various ways, that the boolean expressions tested in a conditional evaluate both to true and false. One particular form of condition coverage, known as MCDC, is required by a DoD standard for safety critical software, such as avionics. This standard, DO-178B, classifies failures into three levels, and demands a different level of coverage for each:

Level C: failure reduces the safety margin
Example: radio data link

Requires: statement coverage

Level B: failure reduces the capability of the aircraft or crew
Example: GPS
Requires: decision coverage

Level A: failure causes loss of aircraft
Example: flight management system
Requires: MCDC coverage

Another common form of program-based subdomain criterion is used in *boundary testing*. This requires that the boundary cases for every conditional be evaluated. For example, if your program tests $x < n$, you would require test cases that produce $x = n$, x=n-1, and x=n+1.

Specification-based criteria are also usually cast in terms of subdomains. Because specifications are usually informal – that is, not written in any precise notation – the criteria tend to be much vaguer. The most common approach is to define subdomains according to the structure of the specification and the values of the underlying data types. For example, the subdomains for a method that inserts an element into a set might include:

- the set is empty
- the set is non-empty and the element is not in the set
- the set is non-empty and the element is in the set

You can also use any conditional structure in the specification to guide the division into subdomains. Moreover, in practice, testers make use of their knowledge of the kinds of errors that often arise in code. For example, if you're testing a procedure that finds an element in an array, you would likely put the element at the start, in the middle and at the end, simply because these are likely to be handled differently in the code.

11.6 Feasibility

Full coverage is rarely possible. In fact, even achieving 100% statement coverage is usually impossible, at the very least because of defensive code which should never be executed. Operations of an abstract data type that have no clients will not be exercised by any system-level test case, although they can be exercised by unit tests.

A criterion is said to be *feasible* if it is possible to satisfy it. In practice, criteria are not usually feasible. In subdomain terms, they contain empty subdomains. The practical

question is to determine whether a particular subdomain is empty or not; if empty, there is no point trying to find a test case to satisfy it.

Generally speaking, the more elaborate the criterion, the harder this determination becomes. For example, *path coverage* requires that every path in the program be executed. Suppose the program looks like this:

if C1 then S1;
if C2 then S2;

Then to determine whether the path $S1;S2$ is feasible, we need to figure out whether the conditions $C1$ and $C2$ can both evaluate to true. For a complex program, this is a non-trivial task, and, in the worst case, is no easier than determining the correctness of the program by reasoning!

Despite these problems, the idea of coverage is a very important one in practice. If there are significant parts of your program that have *never* been executed, you shouldn't have much confidence in their correctness!

11.7 Practical Guidance

It should be clear why neither program-based nor specification-based criteria are alone good enough. If you only look at the program, you'll miss errors of omission. If you only look at the specification, you'll miss errors that arise from implementation concerns, such as when a resource boundary is hit and some special compensating behaviour is needed. In the implementation of the Java *ArrayList*, for example, the array in the representation is replaced when it's full. To test this behaviour, you'll need to insert enough elements into the *ArrayList* to fill the array.

Experience suggests that the best way to develop a good test suite is to use specification-based criteria to guide the development of the suite, and program-based criteria to evaluate it. So you examine the specification, and define input subdomains. Based on these, you write test cases. You execute the test cases, and measure the code coverage. If the coverage is inadequate, you add new test cases.

In an industrial setting, you would use a special coverage tool to measure code coverage. In 6170, we don't expect you to learn to use another tool. Instead, you should just consider your test cases carefully enough that you make an argument that you have achieved reasonable coverage.

Runtime assertions, especially representation invariant checks, will dramatically amplify the power of your testing. You'll find more bugs with fewer cases, and you'll

track them down more easily.

Design Patterns

6.170 Lectures 12–14

October 2, 3, and 10, 2001

Contents

Reading: Chapter 15 of *Program Development in Java* by Barbara Liskov

1 Design patterns

A design pattern is:

- a standard solution to a common programming problem
- a technique for making code more flexible by making it meet certain criteria
- a design or implementation structure that achieves a particular purpose
- a high-level programming idiom
- shorthand for describing certain aspects of program organization
- connections among program components
- the shape of an object diagram or object model

1.1 Examples

Here are some examples of design patterns which you have already seen. For each design pattern, this list notes the problem it is trying to solve, the solution that the design pattern supplies, and any disadvantages associated with the design pattern. A software designer must trade off the advantages against the disadvantages when deciding whether to use a design pattern. Tradeoffs between flexibility and performance are common, as you will often discover in computer science (and other fields).

Encapsulation (data hiding)

Problem: Exposed fields can be directly manipulated from outside, leading to violations of the representation invariant or undesirable dependences that prevent changing the implementation.
Solution: Hide some components, permitting only stylized access to the object.
Disadvantages: The interface may not (efficiently) provide all desired operations. Indirection may reduce performance.

Subclassing (inheritance)

Problem: Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.
Solution: Inherit default members from a superclass; select the correct implementation via run-time dispatching.
Disadvantages: Code for a class is spread out, potentially reducing understandability. Run-time dispatching introduces overhead.

Iteration

Problem: Clients that wish to access all members of a collection must perform a specialized traversal for each data structure. This introduces undesirable dependences and does not extend to other collections.
Solution: Implementations, which have knowledge of the representation, perform traversals and do bookkeeping. The results are communicated to clients via a standard interface.
Disadvantages: Iteration order is fixed by the implementation and not under the control of the client.

Exceptions

Problem: Errors occurring in one part of the code should often be handled elsewhere. Code should not be cluttered with error-handling code, nor return values preempted by error codes.

Solution: Introduce language structures for throwing and catching exceptions.

Disadvantages: Code may still be cluttered. It can be hard to know where an exception will be handled. Programmers may be tempted to use exceptions for normal control flow, which is confusing and usually inefficient.

These particular design patterns are so important that they are built into Java. Other design patterns are so important that they are built into other languages. Some design patterns may never be built into languages, but are still useful in their place.

1.2 When (not) to use design patterns

The first rule of design patterns is the same as the first rule of optimization: delay. Just as you shouldn't optimize prematurely, don't use design patterns prematurely. It may be best to first implement something and ensure that it works, then use the design pattern to improve weaknesses; this is especially true if you do not yet grasp all the details of the design. (If you fully understand the domain and problem, it may make sense to use design patterns from the start, just as it makes sense to use a more efficient rather than a less efficient algorithm from the very beginning in some applications.)

Design patterns may increase or decrease the understandability of a design or implementation. They can decrease understandability by adding indirection or increasing the amount of code. They can increase understandability by improving modularity, better separating concerns, and easing description. Once you learn the vocabulary of design patterns, you will be able to communicate more precisely and rapidly with other people who know the vocabulary. It's much better to say, "This is an instance of the visitor pattern" than "This is some code that traverses a structure and makes callbacks, and some certain methods must be present, and they are called in this particular way and in this particular order."

Most people use design patterns when they notice a problem with their design — something that ought to be easy isn't — or their implementation — such as performance. Examine the offending design or code. What are its problems, and what compromises does it make? What would you like to do that is presently too hard? Then, check a design pattern reference. Look for patterns that address the issues you are concerned with.

The canonical design pattern reference is the "gang of four" book, *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995. Design patterns are popular now, so new books continue to appear.

1.3 Why should you care?

If you are a talented designer and programmer, or you have a lot of time to gain experience, you may encounter or invent many design patterns yourself. However, this is not an effective use of your time. A design pattern represents work by someone else who also encountered the problem, tried many possible solutions, and selected and described one of the best. You should take advantage of that.

Design patterns may seem abstract at first, or you may not be convinced that they address a significant problem. You will come to appreciate them as you build and modify larger systems — perhaps during your work on the Gizmoball final project.

2 Creational patterns

2.1 Factories

Suppose you are writing a class to represent a bicycle race. A race consists of many bicycles (among other objects, perhaps).

```
class Race {

  Race createRace() {
    Frame frame1 = new Frame();
    Wheel frontWheel1 = new Wheel();
    Wheel rearWheel1 = new Wheel();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new Frame();
    Wheel frontWheel2 = new Wheel();
    Wheel rearWheel2 = new Wheel();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
  }

}
```

You can specialize `Race` for other bicycle races:

```
// French race
class TourDeFrance extends Race {

  Race createRace() {
    Frame frame1 = new RacingFrame();
    Wheel frontWheel1 = new Wheel700c();
    Wheel rearWheel1 = new Wheel700c();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new RacingFrame();
    Wheel frontWheel2 = new Wheel700c();
    Wheel rearWheel2 = new Wheel700c();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
  }

  ...
}

// all-terrain bicycle race
class Cyclocross extends Race {

  Race createRace() {
    Frame frame1 = new MountainFrame();
    Wheel frontWheel1 = new Wheel27in();
```

```
    Wheel rearWheel1 = new Wheel27in();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new MountainFrame();
    Wheel frontWheel2 = new Wheel27in();
    Wheel rearWheel2 = new Wheel27in();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
  }

  ...
}
```

In the subclasses, `createRace` returns a `Race` because the Java compiler enforces that overridden methods have identical return types.

For brevity, the code fragments above omit many other methods relating to bicycle races, some of which appear in each class and others of which appear only in certain classes.

The repeated code is tedious, and in particular, we weren't able to reuse method `Race.createRace` at all. (There is a separate issue of abstracting out the creation of a single bicycle to a function; we will use that without further discussion, as it is obvious, at least after 6.001.) There must be a better way. The Factory design patterns provide an answer.

2.1.1 Factory method

A factory method is a method that manufactures objects of a particular type.

We can add factory methods to `Race`:

```
class Race {

  Frame createFrame() { return new Frame(); }
  Wheel createWheel() { return new Wheel(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new Bicycle(frame, front, rear);
  }
  // return a complete bicycle without needing any arguments
  Bicycle completeBicycle() {
    Frame frame = createFrame();
    Wheel frontWheel = createWheel();
    Wheel rearWheel = createWheel();
    return createBicycle(frame, frontWheel, rearWheel);
  }

  Race createRace() {
    Bicycle bike1 = completeBicycle();
    Bicycle bike2 = completeBicycle();
    ...
  }

}
```

Now subclasses can reuse `createRace` and even `completeBicycle` without change:

```
// French race
class TourDeFrance extends Race {

  Frame createFrame() { return new RacingFrame(); }
  Wheel createWheel() { return new Wheel700c(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }

}

class Cyclocross extends Race {

  Frame createFrame() { return new MountainFrame(); }
  Wheel createWheel() { return new Wheel26inch(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }

}
```

The `create...` methods are called *factory methods*.

2.1.2 Factory object

If there are many objects to construct, including the factory methods in each class can bloat the code and make it hard to change. Sibling subclasses cannot easily share the same factory method.

A *factory object* is an object that encapsulates factory methods.

```
class BicycleFactory {
  Frame createFrame() { return new Frame(); }
  Wheel createWheel() { return new Wheel(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new Bicycle(frame, front, rear);
  }

  // return a complete bicycle without needing any arguments
  Bicycle completeBicycle() {
    Frame frame = createFrame();
    Wheel frontWheel = createWheel();
    Wheel rearWheel = createWheel();
    return createBicycle(frame, frontWheel, rearWheel);
  }
}

class RacingBicycleFactory {
  Frame createFrame() { return new RacingFrame(); }
```

```
  Wheel createWheel() { return new Wheel700c(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }
}

class MountainBicycleFactory {
  Frame createFrame() { return new MountainFrame(); }
  Wheel createWheel() { return new Wheel26inch(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }
}
```

The `Race` methods use the factory objects.

```
class Race {

  BicycleFactory bfactory;

  // constructor
  Race() {
    bfactory = new BicycleFactory();
  }

  Race createRace() {
    Bicycle bike1 = bfactory.completeBicycle();
    Bicycle bike2 = bfactory.completeBicycle();
    ...
  }

}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance() {
    bfactory = new RacingBicycleFactory();
  }
}

class Cyclocross extends Race {
  // constructor
  Cyclocross() {
    bfactory = new MountainBicycleFactory();
  }
}
```

In this version of the code, the type of bicycle is still hard-coded into each variety of race. There is a more flexible method which requires a change to the way that clients call the constructor.

```
class Race {

  BicycleFactory bfactory;

  // constructor
  Race(BicycleFactory bfactory) {
    this.bfactory = bfactory;
  }

  Race createRace() {
    Bicycle bike1 = bfactory.completeBicycle();
    Bicycle bike2 = bfactory.completeBicycle();
    ...
  }

}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance(BicycleFactory bfactory) {
    this.bfactory = bfactory;
  }
}

class Cyclocross extends Race {
  // constructor
  Cyclocross(BicycleFactory bfactory) {
    this.bfactory = bfactory;
  }
}
```

This is the most flexible mechanism of all. Now a client can control both the variety of race and the variety of bicycle used in the race, for instance via a call like

```
new TourDeFrance(new TricycleFactory())
```

One reason that factory methods are required is *the first weakness of Java constructors*: Java constructors always return an object of the specified type. They can never return an object of a subtype, even though that would be type-correct (both according to Java subtyping and according to true behavior subtyping as will be described in Lecture 15).

In fact, `createRace` is itself a factory method.

2.1.3 Prototype

The prototype pattern provides another way to construct objects of arbitrary types. Rather than passing in a `BicycleFactory` object, a `Bicycle` object is passed in. Its `clone` method is invoked to create new bicycles; we are making copies of the given object.

```
class Bicycle {
```

```
  Object clone() { ... }
}

class Frame {
  Object clone() { ... }
}

class Wheel {
  Object clone() { ... }
}

class RacingBicycle {
  Object clone() { ... }
}

class RacingFrame {
  Object clone() { ... }
}

class Wheel700c {
  Object clone() { ... }
}

class MountainBicycle {
  Object clone() { ... }
}

class MountainFrame {
  Object clone() { ... }
}

class Wheel26inch {
  Object clone() { ... }
}

class Race {

  Bicycle bproto;

  // constructor
  Race(Bicycle bproto) {
    this.bproto = bproto;
  }

  Race createRace() {
    Bicycle bike1 = (Bicycle) bproto.clone();
    Bicycle bike2 = (Bicycle) bproto.clone();
    ...
```

```
  }

}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance(Bicycle bproto) {
    this.bproto = bproto;
  }
}

class Cyclocross extends Race {
  // constructor
  Cyclocross(Bicycle bproto) {
    this.bproto = bproto;
  }
}
```

Effectively, each object is itself a factory specialized to making objects just like itself. Prototypes are frequently used in dynamically typed languages such as Smalltalk, less frequently used in statically typed languages such as C++ and Java.

There is no free lunch: the code to create objects of particular classes must go somewhere. Factory methods put the code in methods in the client; factory objects put the code in methods in a factory object; and prototypes put the code in `clone` methods.

2.2 Sharing

Several other design patterns are related to object creation in that they affect constructors (and require the use of factories) and are related to structure in that they specify patterns of sharing among various objects.

2.2.1 Singleton

The singleton pattern guarantees that only one object of a particular class ever exists. You might want to use this for `Gym` in your Gym Manager from recitation, because its methods (such as waiting lists for particular machines) are best suited to management of a single location, not oversight from the national office of a chain. A program that instantiates multiple copies probably has an error, but use of the singleton pattern renders such an error harmless.

```
class Gym {
  private static Gym theGym;
  // constructor
  private Gym() { ... }
  // factory method
  public static getGym() {
    if (theGym == null) {
      theGym = new Gym();
    }
    return theGym;
```

```
    }
    ...
}
```

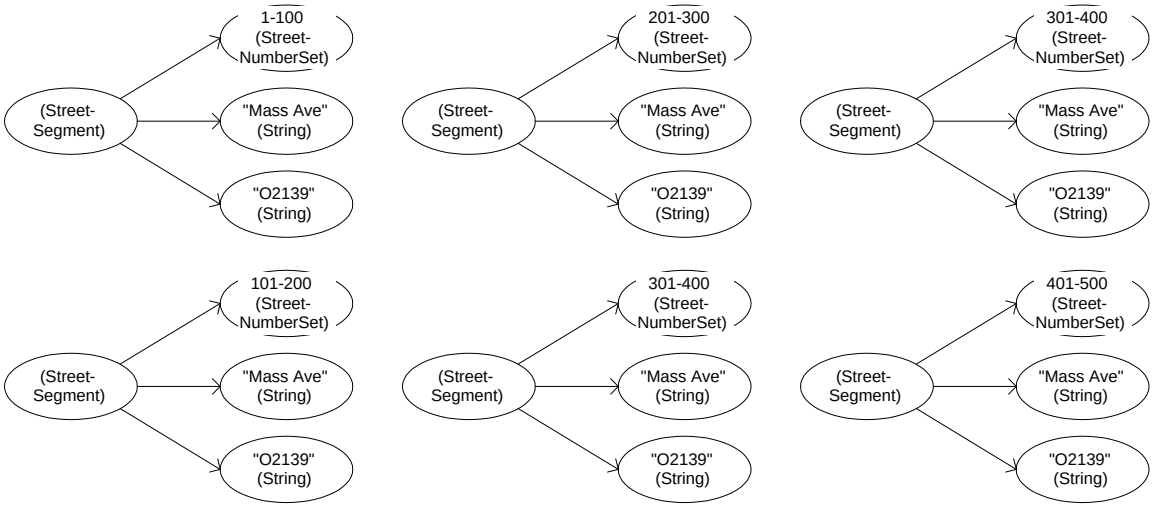
The singleton pattern is also useful for large, expensive objects that should not be multiply instantiated.

The reason that a factory method, rather than a constructor, must be used is *the second weakness of Java constructors*: Java constructors always return a new object, never a pre-existing object.

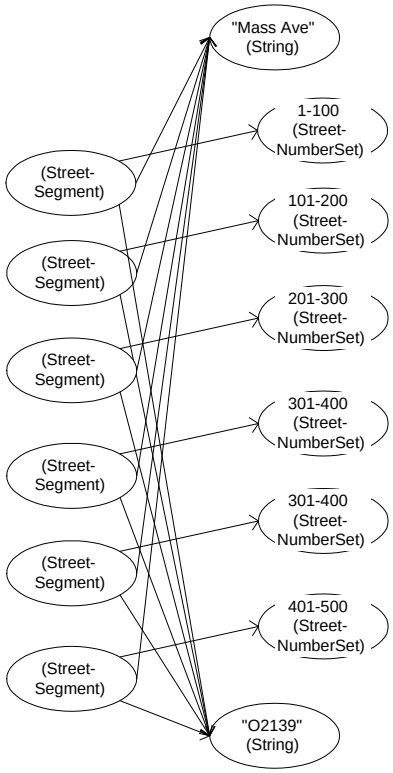
2.2.2 Interning

The interning design pattern reuses existing objects rather than creating new ones. If a client requests an object which is equal to one that already exists, then the pre-existing one is returned instead. This is correct only for immutable objects.

As an example, MapQuick may represent a particular street by many `StreetSegment`s. Those `StreetSegment` objects will have the same street name and zipcode. Here is one possible object diagram (snapshot) for a part of the street.



This representation is correct (for instance, all pairs of street names return true when compared with `equals`), but it is unnecessarily wasteful of space. This would be a better runtime configuration of the system:



The difference in space is substantial — enough that it is very unlikely that you could read even a modest database into MapQuick in the absence of this sharing. Therefore, the implementation of `StreetSegReader` which you were provided performs this operation.

Interning arranges for objects that are immutable to be reused — rather than create a new object, a canonical representation is reused. Interning requires a table of all the objects that have ever been created; if it contains any objects that would be equal to the desired object, that version is returned instead. For performance reasons, a hashtable from contents to objects is usually returned (since equality depends only on the contents).

Here is a code fragment that canonicalizes (interns) strings that name segments.

```
HashMap segnames = new HashMap();

canonicalName(String n) {
  if (segnames.containsKey(n)) {
    return segnames.get(n);
  } else {
    segnames.put(n, n);
    return n;
  }
}
```

Strings are a special case, since the best representation for the sequence of characters (the content) is itself a string; we end up with a table from strings to strings. This strategy is correct in general: the code constructs a non-canonical representation, maps it to the canonical representation, and returns the canonical one. However, depending on how much work the constructor is, it may be more efficient not to construct the non-canonical representation if not necessary, in which case

the table might map from contents to canonical representations. In that case, if we were interning `GeoPoint`s, we would index the table by the latitude and longitude.

This code uses a map from strings to themselves, but it cannot use a `Set` rather than a `Map`. The reason is that `Set`s do not have a `get` operation, only a `contains` operation. The `contains` operation uses `equals` for its comparison. Thus, even if `myset.contains(mystring)`, that doesn't mean that `mystring` is identically a member of `myset`, and there is no convenient way to access the element of `myset` that `equals` `mystring`.

The notion of having only one version of a given string is such an important one that it is built into Java; `String.intern` returns the canonical version of a string.

The Liskov text discusses interning in section 15.2.1, but calls the pattern "flyweight," which is different than the standard terminology in the field.

2.2.3 Flyweight

Flyweight is a generalization of interning. (Liskov text section 15.2.1, titled "Flyweight," discusses interning, which is a special case of flyweight.) Interning is applicable only when an object is completely immutable. The more general form of flyweight can be used when most (but not necessarily all) of an object is immutable.

Consider the case of bicycle spokes.

```
class Wheel {
  ...
  FullSpoke[] spokes;
  ...
}

// We'll name a trimmed-down version "Spoke", so call this "FullSpoke"
class FullSpoke {
  int length;
  int diameter;
  boolean tapered;
  Metal material;
  float weight;
  float threading;
  boolean crimped;
  int location;     // which rim and hub holes this is installed in
}
```

There are typically 32 or 36 spokes per wheel (up to 48 per wheel for a `TandemBicycle`). However, there are usually only three different varieties of spoke per bicycle: one for the front wheel and two for the rear wheel (because the rear hub is off-center, so different lengths are required). We would prefer to allocate just three different `Spoke` (or `FullSpoke`) objects rather than one per spoke of the bicycle. It's not acceptable to have a single `Spoke` object in `Wheel` rather than an array not just because of the asymmetry of the rear wheel but also because I might replace a spoke (say, after one breaks) with another that has the same length but differs in other characteristics. Interning is not an option because the objects are not identical: they differ in their `location` field. In a bicycle rally of 10,000 bicycles, there might only be a few hundred different varieties of spoke but a million instances of them; it would be disastrous to allocate millions of `Spoke` objects. Spoke

objects could be shared between different bicycles (two friends with identical bicycles could share the same spoke #22 on the front wheel), but that still leaves a factor of 32 or 36 too little sharing, and in any event it is more likely that there would be similar spokes within a bicycle than across bicycles.

The first step for using the flyweight pattern is to separate the *intrinsic* state from the *extrinsic* state. The intrinsic state is kept in the object; the extrinsic state is kept outside the object. In order to permit interning, the intrinsic state should be both immutable and similar across objects.

Create a `location`-less `Spoke` class for the intrinsic state:

```
class Spoke {
  int length;
  int diameter;
  boolean tapered;
  Metal material;
  float weight;
  float threading;
  boolean crimped;
}
```

To add the extrinsic state, it does not work to do

```
class InstalledSpokeFull extends Spoke {
  int location;
}
```

because that is just shorthand for `FullSpoke`; `InstalledSpokeFull` takes the same amount of memory as `FullSpoke` because it has the same fields.

Another possibility is

```
class InstalledSpokeWrapper {
  Spoke s;
  int location;
}
```

This is an example of a wrapper (which we will learn more about very soon), and it saves quite a bit of space, because `Spoke` objects can be shared among `InstalledSpokeWrapper` objects. However, there is a solution which uses even less space.

Notice that the location is apparent from the index of the `Spoke` object in the `Wheel.spokes` array:

```
class Wheel {
  ...
  Spoke[] spokes;
  ...
}
```

There is no need to store that (extrinsic) information at all. Some client code (in `Wheel`) must change, because `FullSpoke` methods which used the `location` field must be given access to that information.

Given this version using `FullSpoke`:

```
class FullSpoke {
  // tension the spoke by turning the nipple the specified number of turns
  void tighten(int turns) {
    ... location ...
  }
}

class Wheel {
  FullSpoke[] spokes;

  // The method should be named "true", but that identifier name is a poor choice
  void align() {
    while (wheel is misaligned) {
      ... spokes[i].tighten(numturns) ...
    }
  }
}
```

The corresponding new version using a flyweight spoke is:

```
class Spoke {
  void tighten(int turns, int location) {
    ... location ...
  }
}

class Wheel {
  FullSpoke[] spokes;

  void align() {
    while (wheel is misaligned) {
      ... spokes[i].tighten(numturns, i) ...
    }
  }
}
```

The reference to an interned `Spoke` is so lightweight by comparison with a reference to a non-interned `FullSpoke` that the former is called a flyweight; that could equally apply to `InstalledSpokeWrapper`, though its space overhead is a minimum of three times as great (and possibly more).

The same trick works if `FullSpoke` contains a `wheel` field which refers to the wheel on which it is installed; `Wheel` methods can simply pass `this` to the `Spoke` method.

If `FullSpoke` also contains a boolean field `broken`, how can that be represented? It is also extrinsic information, but it does not appear in the program implicitly as the location and wheel do. It must be stored explicitly in `Wheel`, probably as a `boolean[]` which parallels the `spokes` array. This is slightly unfortunate — the code is starting to get ugly — but acceptable if space savings are critical. If there are many such fields, however, the design should be reconsidered.

Remember that flyweight should only be used at all after profiling has determined that space usage is a critical bottleneck. Introducing such constructs into programs complicates them and presents many opportunities for error. It should be undertaken only in very limited circumstances.

3 Behavioral patterns

3.1 Multi-way communication

It is easy enough for a single client to use a single abstraction. (We have seen patterns for easing the task of changing the abstraction being used, which is a common task.) However, occasionally a client may need to use multiple abstractions; furthermore, the client may not know ahead of time how many or even which abstractions will be used. The observer, blackboard, and mediator patterns permit such communication.

3.1.1 Observer

Suppose that there is a database of all MIT student grades, and the 6.170 staff wishes to view the grades of 6.170 students. They could write a `SpreadsheetView` class that displays information from the database. (We will assume that the viewer caches information about 6.170 students — it needs this information in order to redraw, for example — but whether it does so is not an important part of this discussion.) The display might look something like this:

	PS1	PS2	PS3
B. Bitdiddle	45	85	80
A. Hacker	95	90	85
A. Turing	90	100	95

Suppose the code to communicate between the grade database and the view of the database uses the following interface:

```
interface GradeDBViewer {
  void update(String course, String name, String assignment, int grade);
}
```

When new grade information is available (say, a new assignment is graded and entered, or an assignment is regraded and the old grade corrected), the grade database must communicate that information to the view. Let's suppose that Ben Bitdiddle has demanded a regrade on problem set 1, and that regrade did reveal grading errors: Ben's score should have been 30. The database code must somewhere make calls to `SpreadsheetView.update`. Suppose that it does so in the following way:

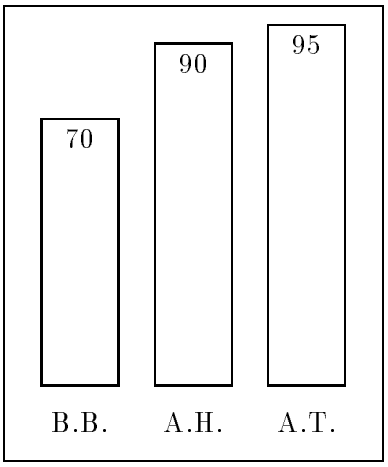
```
SpreadsheetView ssv = new SpreadsheetView();
...
ssv.update("6.170", "B. Bitdiddle", "PS1", 30);
```

(For brevity, this code shows literal values rather than variables for the `update` arguments.)

Then the spreadsheet view would redisplay itself in the following way:

	PS1	PS2	PS3
B. Bitdiddle	30	85	80
A. Hacker	95	90	85
A. Turing	90	100	95

The staff might later decide that they would like to also view grade averages as a bargraph, and implement such a viewer:



Maintaining such a view in addition to the spreadsheet view requires modifying the database code:

```
SpreadsheetView ssv = new SpreadsheetView();
BargraphView bgv = new BargraphView();
...
ssv.update("6.170", "B. Bitdiddle", "PS1", 30);
bgv.update("6.170", "B. Bitdiddle", "PS1", 30);
```

Likewise, adding a pie chart view, or removing some view, would require yet more modifications to the database code. Object-oriented programming (not to mention good programming practice) is supposed to provide relief from such hard-coded modifications: code should be reusable without editing and recompiling either the client or the implementation.

The observer pattern achieves the goal in this case. Rather than hard-coding which views to update, the database can maintain a list of observers which should be notified when its state changes.

```
Vector observers = new Vector();
...
for (int i=0; i<observers.size(); i++) {
  GradeDBViewer v = (GradeDBViewer) observers[i];
  v.update("6.170", "B. Bitdiddle", "PS1", 30);
}
```

In order to initialize the vector of observers, the database will provide two additional methods, `register` to add an observer and `remove` to remove an observer.

```
void register(GradeDBViewer observer) {
  observers.add(observer);
}

boolean remove(GradeDBViewer observer) {
  return observers.remove(observer);
}
```

The observer pattern permits client code (which manages the database and the viewers) to select which observers are active, and observers can even be added and removed at run-time.

This discussion has glossed over a number of details. For instance, the client might store all the information of interest to it (which might be all the 6.170 grades, or just the grades for some students, or just the number of updates to the database for a `DatabaseActivityViewer`), duplicating parts of the database, or the client might read the database when needed. A related design decision is whether the database sends all potentially relevant information to the client when an update occurs (this is the *push* structure), or the database simply informs the client, "an update has occurred" (this is the *pull* structure). The pull structure forces the client to request information, which may result in more messages, but overall a smaller amount of data transferred.

3.1.2 Blackboard

The blackboard pattern generalizes the observer pattern to permit multiple data sources as well as multiple viewers. It also has the effect of completely decoupling producers and consumers of information.

A blackboard is a repository of messages which is readable and writable by all processes. Whenever an event occurs that might be of interest to another party, the process responsible for or knowledgeable about the event adds to the blackboard an announcement of the event. Other processes can read the blackboard. In the typical case, they will ignore most of its contents, which do not concern them, but they may take action on other events. A process which posts an announcement to the blackboard has no idea whether zero, one, or many other processes are paying attention to its announcements.

Blackboards generally do not enforce a particular structure on their announcements, but a well-understood message format is required so that processes can interoperate. Some blackboards provide filtering services so that clients do not see all announcements, just those of a particular type; other blackboards automatically send announcements to clients which have registered interest (this is a pull structure).

An ordinary bulletin board (either the physical or the electronic kind) is an example of a blackboard system. Another example of a blackboard at MIT is the zephyr messaging service.

The Liskov text calls this pattern "white board" rather than "blackboard." The former name may be more modern-seeming, but the latter is standard computer science terminology which has been in use for decades and will be more quickly recognized outside 6.170. The first major blackboard system was the Hearsay-II speech recognition system, implemented between 1971 and 1976.

3.1.3 Mediator

The mediator pattern is intermediate between observer and blackboard. It decouples information producers and consumers but does not decouple control. Whereas blackboard communication is asynchronous, mediators are synchronous: they do not return control to the producer before passing the information to all consumers.

3.2 Traversing composites

Refer to Section 4.2 for composite patterns.

This section discusses traversing composites and/or performing other operations on all the subparts of a composite. Our goal is to support many different operations, and to be able to perform them on many different subparts of a composite. Since both the operation to be performed and the type of composite object to be operated upon affect the implementation, deciding how to break down the problem can be difficult.

Consider the example of an *abstract syntax tree*, or AST, which is a representation of (the syntax of) a computer program. For instance, the binary addition operator `+` might be represented by `PlusOp` objects:

```
class PlusOp extends Expression {
  Expression leftExp;
  Expression rightExp;
  ...
}
```

Variable references, assignment operations (`a=b`), and conditional expressions (`a?b:c`) are other types of expression:

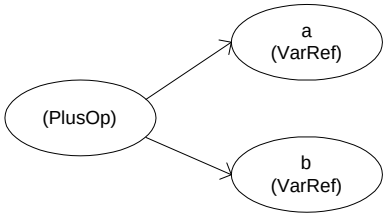
```
class VarRef extends Expression {
  String varname;
  ...
}

class AssignOp extends Expression {
  VarRef lvalue;                // left-hand side; "a" in "a=b"
  Expression rvalue;            // right-hand side; "b" in "a=b"
  ...
}

class CondExpr extends Expression {
  Expression condition;
  Expression thenExpr;
  Expression elseExpr;
  ...
}
```

A complete representation would have many other AST node types as well, such as `AssignOp` for assignments, for expressions, and so forth.

A particular use of `+`, such as `a + b`, would be represented at runtime by



A compiler or other program analysis tool creates an AST by parsing the target program; after parsing, the tool performs operations, such as typechecking, pretty-printing, optimizing, or generating code, on the AST. Each operation is different than the others, but each AST node is also unlike the others.

Each box of this table will be filled in with a different piece of code:

		Objects	
		`CondExpr`	`AssignOp`
Operations	typecheck		
	pretty-print		

The question is whether to organize the code so as to group all the typechecking code together (and necessarily spread code dealing with `CondExpr`s across the implementation) or to group all the code dealing with a particular type of expression, but split up code dealing with a particular operation.

(A related issue is how to select and execute the proper block of code, regardless of where it may be located. Java's method dispatch mechanism selects which version of an overloaded method to call based on the run-time type of the receiver. This makes it possible to dispatch based on either operations or objects, but not both at the same time.)

The interpreter and procedural patterns (and visitor, a refinement of procedural) permit expression of operations over composite objects such as ASTs. Interpreter collects together similar objects and spreads apart similar operations. Procedural collects similar operations and spreads apart similar objects. That means that

> Interpreter makes it easy to add objects, hard to add operations.
> Procedural makes it easy to add operations, hard to add objects.

"Easy" and "hard" refer to how many different classes need to be modified. When the interpreter class is used, adding a new object requires writing a single new class, but adding a new operation requires modifying every existing class. The reverse is true for procedural. Both patterns have classes for all objects that capture those objects' idiosyncrasies, as illustrated in the code examples for `CondExpr` and `AssignOp` above; the question is where to place the implementations of operations that exist for all objects. The examples below should clarify this notion.

Which approach should you take when designing a software system depends on two factors. First, do you view the system as operation-centric or operand-centric? Are the algorithms central, or are the objects? (In an object-oriented system, often the objects are.) Second, what aspects of the system are most likely to change? (A programming language's syntax rarely changes to add new types of expression, but a program analyzer such as a compiler is often extended with new functionality.) Those changes should be eased by your choice of design pattern.

3.2.1 Interpreter

The interpreter pattern groups together all the operations for a particular variety of object. It uses the pre-existing classes for objects and adds to each class a method for each supported operation. For example,

```
class Expression {
  ...
  Type typecheck();
  String prettyPrint();
}

...

class AssignOp extends Expression {
  ...
  Type typecheck() { ... }
  String prettyPrint() { ... }
}
```

```
class CondExpr extends Expression {
  ...
  Type typecheck() { ... }
  String prettyPrint() { ... }
}
```

3.2.2 Procedural

The procedural pattern groups together all the code that implements a particular operation. It creates a class for each operation; the class has a separate method for each type of operand. For example, typechecking code might look like this:

```
class Typecheck {
  ...
  // typecheck "a?b:c"
  Type tcCondExpr(CondExpr e) {
    Type codeType = tcExpression(e.condition); // type of "a"
    Type thenType = tcExpression(e.thenExpr);  // type of "b"
    Type elseType = tcExpression(e.elseExpr);  // type of "c"
    // BoolType is defined elsewhere
    if ((condType == BoolType) && (thenType == elseType)) {
      // This expression is well-typed, because the condition is of boolean
      // type and the then and else branches have the same type.
      // The type of the whole expression is the type of the branches.
      return thenType;
    } else {
      return ErrorType;         // ErrorType is defined elsewhere
    }
  }

  // typecheck "a=b"
  Type tcAssignOp(AssignOp e) {
    ...
  }

}
```

The procedural pattern works well enough, but there is one ugly aspect: the definition of `tcExpression`. It needs to call `tcCondExpr` or `tcAssignOp` or `tcVarRef` or some other function, depending on the run-time type of the subcomponents of an expression.

```
class Typecheck {
  ...
  Type tcExpression(Expression e) {
    if (e instanceof PlusOp) {
      return tcPlusOp((PlusOp)e);
    } else if (e instanceof VarRef) {
      return tcVarRef((VarRef)e);
    } else if (e instanceof AssignOp) {
```

```
      return tcAssignOp((AssignOp)e);
    } else if (e instanceof CondExpr) {
      return tcCondExpr((CondExpr)e);
    } else ...
    ...
  }

}
```

Maintaining this code is tedious and error-prone, and the long cascaded `if` tests are likely to run slowly. Furthermore, even though this code would be undesirable even if it occurred only once, in fact it occurs again in the `PrettyPrint` class and in every other operation class. Systematic repetition in code is usually a sign for a need to redesign, possibly using a design pattern.

We already know of a Java construct that automatically chooses which code to execute based on a type test: method dispatching. It does the same kind of comparison and selection as the cascaded `if` tests, but does not clutter the code and is likely to be considerably more efficient. The visitor pattern takes advantage of this.

3.2.3 Visitor

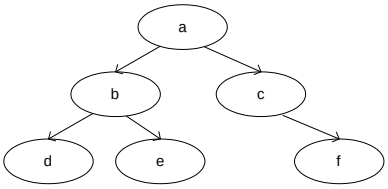
The visitor pattern encodes a depth-first traversal (or alternately, some other variety of traversal) over a hierarchical data structure such as one resulting from the composite pattern. The visitor pattern depends on two operations: nodes (objects) accept visitors, and visitors visit nodes (objects). Conceptually, the code structure is as follows:

```
class Node {
  ...
  void accept(Visitor v) {
    for each child of this node {
      child.accept(v);
    }
    v.visit(this);
  }
}

class Visitor {
  ...
  void visit(Node n) {
    perform work on n
  }
}
```

The `accept` and `visit` methods work together so that `n.accept(v)` performs a depth-first traversal of the structure rooted at `n`, with the operation represented by `v` performed on each component of the structure in turn.

Consider a composite with the following structure:



The sequence of calls resulting from `a.accept(v)` for some visitor `v` is:

```
a.accept(v)
  b.accept(v)
    d.accept(v)
      v.visit(d)
    e.accept(v)
      v.visit(e)
    v.visit(b)
  c.accept(v)
    f.accept(v)
      v.visit(f)
    v.visit(c)
  v.visit(a)
```

The sequence of calls to `visit`, which performs the actual work, is `d`, `e`, `b`, `f`, `c`, `a`; this is a depth-first search. The `visit` method might count the number of nodes, or perform type-checking, or some other operation.

The visitor pattern requires the addition of `visit` and `accept` methods; see the Liskov book for an example. As with the procedural pattern, visitor makes it easy to add operations (visitors) but hard to add nodes (which requires modifying each existing visitor).

A visitor is very much like an iterator: essentially, each element of a data structure is presented in turn to the `visit` method. It gives the opportunity for more, however: a visitor can accumulate state that would be impossible to determine from the sequence of nodes alone. Unfortunately, the implementation structure described above does not provide any way for one call of `visit` to communicate with another.

Here are two possible solutions to this problem. The book proposes saving information in a separate data structure (for example, a stack) which can be read and written. This keeps the visitors and acceptors clean, but it can be hard to see how data flows between calls.

An alternate solution is to move some of the work into the visitor itself:

```
class Node {
  ...
  void accept(Visitor v) {
    v.visit(this);
  }
}

class Visitor {
  ...
  void visit(Node n) {
    for each child of this node {
      child.accept(v);
```

```
    }
    perform work on n
  }
}
```

This solution has several problems. For one thing, there are many visitors, so the traversal code is repeated many times rather than just appearing once (since there is only one acceptor). Second, the acceptor is not really doing anything any more. The visitor is essentially doing a depth-first search of its own. This solution does have the merit of making the information flow clearer, in the common case that a visitor for a node depends on the results of from visiting children.

3.3 State

We will not discuss the state pattern in detail, but you might want to consider it for implementing `StreetNumberSet`.

4 Structural patterns

4.1 Wrappers

Wrappers modify the behavior of another class; they are usually a thin veneer over the encapsulated class, which does the real work. The wrapper may modify the interface, extend the behavior, or restrict access. The wrapper intermediates between two incompatible interfaces, translating calls between the interfaces. This permits two pieces of code that were not designed or written together, and thus are slightly incompatible, to be used together anyway.

Three varieties of wrappers are adapters, decorators, and proxies:

Pattern	Functionality	Interface
Adapter	same	different
Decorator	different	same
Proxy	same	same

The functionality and interfaces compared are those at the inside and outside of the wrapper; that is, a client's view of the wrapped object is compared to a client's view of the wrapper.

The remainder of this section discusses the three varieties of wrapper, then examine tradeoffs between two implementation strategies, subclassing and delegation.

4.1.1 Adapter

Adapters change the interface of a class without changing its basic functionality. For instance, they might permit interoperability between a geometry package that requires angles to be specified in radians and a client that expects to pass angles in degrees. Here are two other examples:

Example: Rectangle Suppose that you have written code that works on `Rectangle` objects and calls their `scale` method.

```
interface Rectangle {
  // grow or shrink this by the given factor
  void scale(float factor);

  // other operations
  float area();
  float circumference();
  ...
}

class myClass {

  void myMethod(Rectangle r) {
    ...
    r.scale(2);
    ...
  }

}
```

Suppose there is another class `NonScaleableRectangle` which lacks the `scale` method but does have the other methods of `Rectangle`, as well as additional `setWidth` and `setHeight` methods.

```
class NonScaleableRectangle {
  void setWidth(float width) { ... }
  void setHeight(float height) { ... }
  ...
}
```

You may wish to switch to (or at least permit use of) this variety of rectangle, perhaps because it has desirable features, such as better performance, or perhaps because it is used elsewhere, in a system with which you need to interoperate.

You cannot use `NonScaleableRectangle` directly because of the incompatible interface. However, you can write an adapter which permits its use. There are two ways to do this: subclassing and delegation. The subclassing solution will be familiar:

```
class ScaleableRectangle1 extends NonScaleableRectangle implements Rectangle {
  void scale(float factor) {
    setWidth(factor * getWidth());
    setHeight(factor * getHeight());
  }
}
```

Delegation is a technique for "passing the buck", forwarding a request so that a different object does the requested work.

```
class ScaleableRectangle2 implements Rectangle {
  NonScaleableRectangle r;
  ScaleableRectangle2(NonScaleableRectangle r) {
    this.r = r;
  }

  void scale(float factor) {
    setWidth(factor * getWidth());
    setHeight(factor * getHeight());
  }

  float area() { return r.area(); }
  float circumference() { return r.circumference(); }
  ...
}
```

Example: Palette Suppose that Professor Jackson calls Professor Ernst late at night because someone has discovered a problem with the problem set: it needs to support bicycles that can be repainted (to change their color). The professors split up the work: Professor Jackson will write a `ColorPalette` class with a method that, given a name like "red" or "blue" or "taupe", returns an array of three RGB values, and Professor Devadas will write code that uses this class. The professors do so, test their work, and go away for the weekend, leaving the the `.class` files for the TAs to integrate. They find that Professor Devadas has written code that depends on

```
interface ColorPalette {
  // returns RGB values
  int[] getColor(String name);
}
```

but Professor Jackson has implemented a class that adheres to

```
interface ColourPalette {
  // returns RGB values
  int[] getColour(String name);
}
```

What are the TAs to do? They do not have access to the source, and they do not have time to reimplement and retest. Their solution is to write a adapter for `ColourPalette` that changes the operation name. They can implement the adapter either by subclassing or by delegation.

4.1.2 Decorator

Whereas an adapter changes an interface without adding new functionality, a decorator extends functionality while maintaining the same interface. Typically, a decorator does not change existing functionality, only adds to it, so that objects of the resulting class behave exactly like the original ones, but also do something extra.

This sounds like subclassing, but not every instance of subclassing is a decoration. First, the implementation of an operation may be completely different or reimplemented in a subclass; that is not usually the case for a decorator, which contains relatively less functionality and reuses the superclass code. Second, subclasses can introduce new operations; wrappers (including decorators) generally do not.

An example of decoration is a `Window` interface (for a window manager) and a `BorderedWindow` interface. The `BorderedWindow` behaves exactly like the `Window`, except that it also draws a border around the outside.

Suppose that `Window` is implemented like this:

```
interface Window {
  // rectangle bounding the window
  Rectangle bounds();
  // draw this on the specified screen
  void draw(Screen s);
  ...
}

class WindowImpl implements Window {
  ...
}
```

The subclassing implementation would look like this:

```
class BorderedWindow1 extends WindowImpl {
  void draw(Screen s) {
    super.draw(s);
    bounds().draw(s);
```

```
  }
}
```

The delegation implementation would look like this:

```
class BorderedWindow2 implements Window {
  Window innerWindow;

  BorderedWindow2(Window innerWindow) {
    this.innerWindow = innerWindow;
  }

  void draw(Screen s) {
    innerWindow.draw(s);
    innerWindow.bounds().draw(s);
  }
}
```

4.1.3 Proxy

A proxy is a wrapper that has the same interface and the same functionality as the class it wraps. This does not sound very useful on the face of it. However, proxies serve an important purpose in controlling access to other objects. This is particularly valuable if those objects must be accessed in a stylized or complicated way.

For example, if an object is on a remote machine, then accessing it requires use of various network facilities. It is easier to create a local proxy that understands the network and performs the necessary operations, then returns the result. This simplifies the client by localizing network-specific code in another location.

As another example, an object may require locking if it can be accessed by multiple clients. The lock represents the right to read and/or update the object; without the lock, concurrent updates could leave the object in an inconsistent state, or reads in the middle of a sequence of updates could observe an inconsistent state. A proxy could take care of locking an object before an operation or sequence of operations, then unlocking it afterward. This is less error-prone than requiring clients to correctly implement the locking protocol.

Another variety of proxy is a security proxy. It might operate correctly if the caller has the correct credentials (such as a valid Kerberos certificate), but throw an error if an unauthorized user attempts to perform operations.

A final example is a proxy for an object that may not yet exist. If creating an object is expensive (because of computation or network latency), then it can be represented by a proxy instead. That proxy could immediately start to create the object in a background task in the hope that it is ready by the time the first operation is invoked, or it could delay creating the object until an operation is invoked. In the former case, the rest of the system can proceed without waiting; in the latter case, the work of creating the object need never be performed if it is never used. In either case, operations are delayed until the object is ready.

An example of a proxy for a non-existent object is Emacs's autoload functionality. For instance, I have a file `util-mde.el` which defines of a number of useful functions. However, I don't want to slow down Emacs by loading it every time I start Emacs. Instead, my `.emacs` file contains code like this:

```
(autoload 'looking-back-at "util-mde")
(autoload 'in-buffer "util-mde")
(autoload 'in-window "util-mde")
```

The form `(autoload 'function "file")` is essentially equivalent to (in Scheme syntax; Emacs Lisp uses `defun`)

```
(define function ()
  (load "file")  ;; redefine function
  (function)     ;; call the new version
)
```

Emacs autoloads most of its own functionality, from the mail and news readers to the Java editing mode. People who complain that Emacs starts up too slowly often have put indiscriminate `load` forms in their `.emacs` files; that's like using an inefficient implementation, then complaining that the compiler is poor because the resulting program runs slowly.

Proxy capabilities are particularly useful when clients have no knowledge of whether the object they are manipulating has special properties (such as being located on a remote machine, requiring locking or security, or not being loaded). It is best to insulate the client from such concerns and localize them in a proxy wrapper.

4.1.4 Subclassing vs. delegation

Subclassing and delegation are two implementation strategies for implementing wrappers. You are already familiar with subclassing; delegation stores an object in a field, then passes messages to the object.

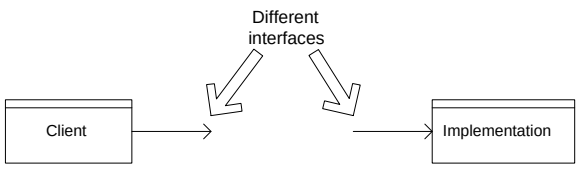
Subclassing automatically gives clients access to all the methods of the superclass. Delegation forces the creation of many small methods like

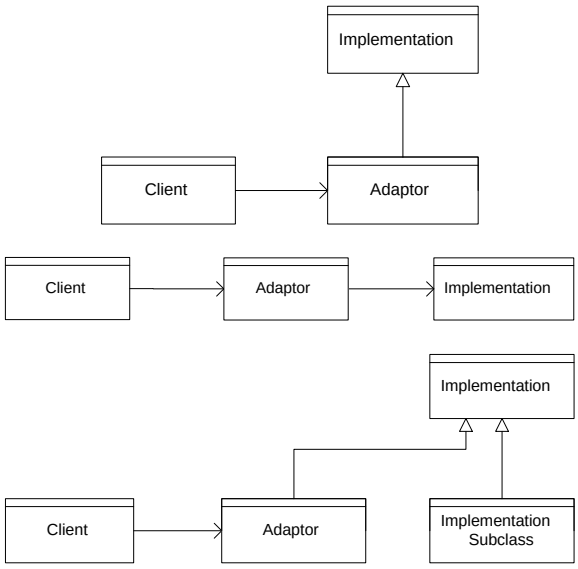
```
void area() { return r.area(); }
```

On the other hand, in order to prevent access to certain methods of the parent, the subclass must override them to throw an error; it would be cleaner not to have them in the interface, which delegation easily permits.

Another potential advantage of subclassing is that it is built into the language; it is likely to be fairly easy to understand and the implementation of subclassing is likely to be fairly efficient.

Delegation is usually the preferred technique for wrappers (and for many design patterns). Wrappers can be added and removed dynamically. For instance, a window can be bordered when active and unbordered otherwise. Another advantage is that objects of arbitrary concrete classes can be wrapped. Creating a subclass specifies the exact type of the object being wrapped. By contrast, the wrapper can wrap an object of any subclass of the declared type of the contained object. Another strength of delegation (related to the previous one) is the ability to use multiple adapters. (For instance, consider how you would create a doubly-bordered window.)





Implementations of the three varieties of wrappers have the same gross structure; without looking at the method bodies to see what work is being done, it is not obvious whether a wrapper is a decorator or a proxy. (An adapter has a different interface than the class it interfaces to.) Some wrappers can even have aspects of more than one variety, though in that case it is clearest for the documentation to say (for instance), "This is both an adapter and a decorator."

4.2 Composite

The composite design pattern permits a client to manipulate either an atomic unit or a collection of units in exactly the same way. The client need not create special code for the case of being given a higher-level object with structure as opposed to being given a basic object; the same operations work on both.

Composite is good for object with part-whole relationships, and the client should not have to worry about whether its argument is atomic or composed of parts.

For example, a bicycle can be decomposed in the following way:

```
Bicycle
  Wheel
    skewer
    hub
    spokes
    nipples
    rim
    tube
    tire
  Frame
  Drivetrain
  ...
```

Given a bicycle component, I might want to determine its weight or cost regardless of whether it is itself composed of subcomponents. A client which is given a bicycle component shouldn't have to treat it differently if it is a wheel as opposed to a reflector or a saddle.

The solution to this problem is for all bicycle components to satisfy a common interface:

```
class BicycleComponent {
  int weight();
  float cost();
}
```

The implementation of `Wheel.weight` might itself call `weight` on its subparts, but that is of no import to the client (and the client shouldn't have to worry about that).

An alternative to using a common interface is to have a common superclass; in either way, all bicycle components at all levels provide the same methods and can be used interchangeably.

As another example, a lending library's holdings might organized into levels as follows:

Library
 Section (for a given genre)
  Shelf
   Volume
    Page
     Column
      Word
       Letter

If all of these satisfy the interface

```
interface Text {
  String getText();
}
```

then a client can (say) count the number of words, or perform other operations, on as large or as small a part of the library's holdings as desired.

The book gives another example, the syntax of computer programs. Notice that there are two completely unrelated tree structures, illustrated in figures 15.12 and 15.13 on page 392. One is the abstract syntax tree, which breaks down the syntax of a particular utterance in the language, such as a particular block of code. The other is the class hierarchy, which expresses subtyping and inheritance. (In Java, whenever there is inheritance, there is always subtyping, because every subclass is a subtype.) The latter organization permits `Node` to have methods like `typeCheck` or `prettyPrint` which can be called regardless of which concrete `Node` is being operated on. Methods, classes, and packages might also be `Node`s in this representation.

Subtypes and Subclasses

6.170 Lecture 15

October 15, 2001

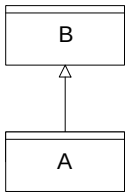
Contents

Reading: Chapter 7 of *Program Development in Java* by Barbara Liskov

(See your Java book for Java details such as abstract types (not all methods are implemented, and no objects can be instantiated) details of interfaces, and access modifiers (public, private, protected, default). They won't be covered in this lecture.

1 Subtypes

We say that that *A is a B* if every A object is also a B object. For instance, every automobile is a vehicle, and every bicycle is a vehicle, and every pogo stick is a vehicle; every vehicle is a mode of transport, as is every pack animal. We annotate this subset relationship in a module dependence diagram:



This subset relationship is a necessary but not sufficient condition for a *subtyping* relationship. Type A is a subtype of type B when A's specification implies B's specification. That is, any object (or class) that satisfies A's specification also satisfies B's specification, because B's specification is weaker.

Another way to put this is that anywhere in the code, if you expect a B object, an A object is acceptable. Code written to work with B objects (and to depend on their properties) is guaranteed to continue to work if it is supplied A objects instead; furthermore, the behavior will be the same, if we only consider the aspects of A's behavior that is also included in B's behavior. (A may introduce new behaviors that B does not have, but it may only change existing B behaviors in certain ways; see below.)

2 Example: Bicycles

Suppose we have a class for representing bicycles. Here is a partial implementation:

```
class Bicycle {
  private int framesize;
  private int chainringGears;
  private int freewheelGears;
  ...

  // returns the number of gears on this bicycle
  public int gears() { return chainringGears * freewheelGears; }
  // returns the cost of this bicycle
  public float cost() { ... }
  // returns the sales tax owed on this bicycle
  public float salesTax() { return cost() * .0825; }
  // effects: transports the rider from work to home
  public void goHome() { ... }
  ...
}
```

A new class representing bicycles with headlamps can accommodate late nights (or early mornings).

```
class LightedBicycle {
  private int framesize;
  private int chainringGears;
  private int freewheelGears;
  private BatteryType battery;
  ...

  // returns the number of gears on this bicycle
  public int gears() { return chainringGears * freewheelGears; }
  // returns the cost of this bicycle
  float cost() { ... }
  // returns the sales tax owed on this bicycle
  public float salesTax() { return cost() * .0825; }
  // effects: transports the rider from work to home
  public void goHome() { ... }
  // effects: replaces the existing battery with the argument b
  public void changeBattery(BatteryType b);
  ...
}
```

Copying all the code is tiresome and error-prone. (The error might be failure to copy correctly or failure to make a required change.) Additionally, if a bug is found in one version, it is easy to forget to propagate the fix to all versions of the code. Finally, it is very hard to comprehend the distinction the two classes by looking for differences themselves in a mass of similarities.

Java and other programming languages use subclassing to overcome these difficulties. Subclassing permits reuse of implementations and overriding of methods.

A better implementation of `LightedBicycle` is

```
class LightedBicycle extends Bicycle {
  private BatteryType battery;
  ...

  // returns the cost of this bicycle
  float cost() { return super.cost() + battery.cost(); }
  // effects: transports the rider from work to home
  public void goHome() { ... }
  // effects: replaces the existing battery with the argument b
  public void changeBattery(BatteryType b);
  ...
}
```

`LightedBicycle` need not implement methods and fields that appear in its superclass `Bicycle`; the `Bicycle` versions are automatically used by Java when they are not overridden in the subclass.

Consider the following implementations of the `goHome` method (along with more complete specifications). If these are the only changes, are `LightedBicycle` and `RacingBicycle` subtypes of `Bicycle`? (For the time being we will talk about subtyping; we'll return to the differences between Java subclassing, Java subtyping, and true subtyping later.)

```
class Bicycle {
  ...
  // requires: windspeed < 20mph && daylight
  // effects: transports the rider from work to home
  void goHome();
}

class LightedBicycle {
  ...
  // requires: windspeed < 20mph
  // effects: transports the rider from work to home
  void goHome();
}

class RacingBicycle {
  ...
  // requires: windspeed < 20mph && daylight
  // effects: transports the rider from work to home
  //          in an elapsed time of < 10 minutes
  //          && gets the rider sweaty
  void goHome();
}
```

To answer that question, recall the definition of subtyping: can an object of the subtype be substituted anywhere that code expects an object of the supertype? If so, the subtyping relationship is valid.

In this case, both `LightedBicycle` and `RacingBicycle` are subtypes of `Bicycle`. In the first case, the requirement is relaxed; in the second case, the effects are strengthened in a way that still satisfies the superclass's effects.

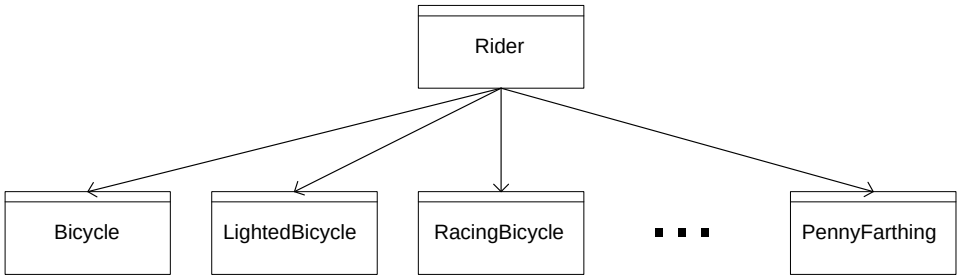
The `cost` method of `LightedBicycle` shows another capability of subclassing in Java. Methods can be overridden to provide a new implementation in a subclass. This enables more code reuse; in particular, `LightedBicycle` can reuse `Bicycle`'s `salesTax` method. When `salesTax` is invoked on a `LightedBicycle`, the `Bicycle` version is used instead. Then, the call to `cost` inside `salesTax` invokes the version based on the runtime type of the object (`LightedBicycle`), so the `LightedBicycle` version is used. Regardless of the declared type of an object, an implementation of a method with multiple implementations (of the same signature) is always selected based on the run-time type.

In fact, there is no way for an external client to invoke the version of a method specified by the declared type or any other type that is not the run-time type. This is an important and very desirable property of Java (and other object-oriented languages). Suppose that the subclass maintains some extra fields which are kept in sync with fields of the superclass. If superclass methods could be invoked directly, possibly modifying superclass fields without also updating subclass fields, then the representation invariant of the subclass would be broken.

A subclass may invoke methods of its parent via use of `super`, however. Sometimes this is useful when the subclass method needs to do just a little bit more work; recall the `LightedBicycle` implementation of `cost`:

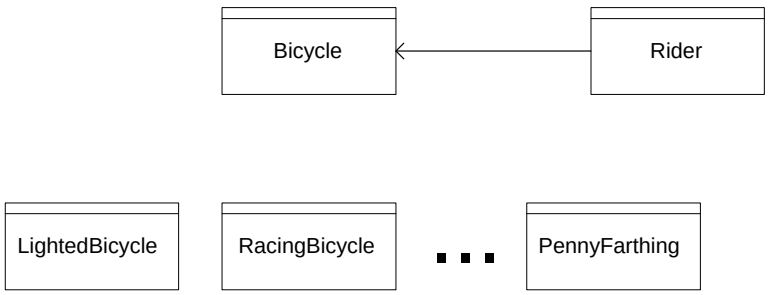
```
class LightedBicycle extends Bicycle {
  // returns the cost of this bicycle
  float cost() { return super.cost() + battery.cost(); }
}
```

Suppose the `Rider` class models people who ride bicycles. In the absence of subclassing and subtypes, the module dependence diagram would look something like this:

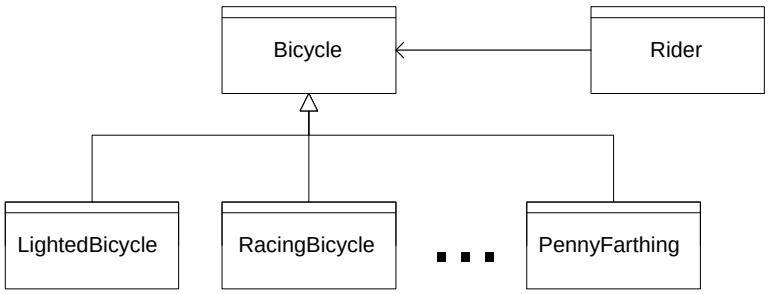


The code for `Rider` would also need to test which type of object it had been passed, which would be ugly, verbose, and error-prone.

With subtyping, the MDD dependences look like this:



The many dependences have been reduced to a single one.
When subtype arrows are added, the diagram is only a bit more complicated:



Even though there are just as many arrows, this diagram is much simpler than the original one: dependence edges complicate designs and implementations more than other types of edge.

3 Example: Square and rectangle

We know from elementary school that every square is a rectangle. Suppose we wanted to make `Square` a subtype of `Rectangle`, which included a `setSize` method:

```
class Rectangle {
  ...

  // effects: sets width and height to the specified values
  //          (that is, this.width' = w && this.height' = h)
  void setSize(int w, int h);
}

class Square extends Rectangle {
  ...
}
```

Which of the following methods is right for `Square`?

```
  // requires: w = h
  void setSize(int w, int h);

  void setSize(int edgeLength);

  // throws BadSizeException if w != h
  void setSize(int w, int h) throws BadSizeException;
```

The first one isn't right because the subclass method requires more than the superclass method. Thus, subclass objects can't be substituted for superclass objects, as there might be code that called `setSize` with non-equal arguments.

The second one isn't right (all by itself) because the subclass still must specify a behavior for `setSize(int, int)`; that definition is of a different method (whose name is the same but whose signature differs).

The third one isn't right because it throws an exception that the superclass doesn't mention. Thus, it again has different behavior and so `Square`s can't be substituted for `Rectangle`s. (If `BadSizeException` is an unchecked exception, then Java will permit the third method to compile; but then again, it will also permit the first method to compile. Java's notion of subtype is weaker than the 6.170 notion of subtype. With no hubris whatsoever, we will call the latter "true subtypes" to distinguish them from Java subtypes.)

There isn't a way out of this quandary without modifying the supertype. Sometimes subtypes do not accord with our intuition! Or, our intuition about what is a good supertype is wrong.

One plausible solution would be to change `Rectangle.setSize` to specify that it throws the exception; of course, in practice only `Square.setSize` would do so. Another solution would be to eliminate `setSize` and instead have a

```
void scale(double scaleFactor);
```

method that shrinks or grows a shape. Other solutions are also possible.

4 Substitution principle

The *substitution principle* is the theoretical underpinning of subtypes; it provides a precise definition of when two types are subtypes. Informally, it states that subtypes must be substitutable for supertypes. This guarantees that if code depends on (any aspect of) a supertype, but an object of a subtype is substituted, system behavior will not be affected. (The Java compiler also requires that the `extends` or `implements` clause names the parent in order for subtypes to be used in place of supertypes.)

The methods of a subtype must hold certain relationships to the methods of the supertype, and the subtype must guarantee that any properties of the supertype (such as representation invariants or specification constraints) are not violated by the subtype.

methods There are two necessary properties:

1. For each method in the supertype, the subtype must have a corresponding method. (The subtype is allowed to introduce additional, new methods that do not appear in the supertype.)
2. Each method in subtype that corresponds to one in the supertype:
   - requires less (has a weaker precondition)
     - there are fewer "requires" clauses, and each one is less strict than the one in the supertype method.
     - the argument types may be supertypes of the ones in the supertype. This is called contravariance, and it feels somewhat backward, because the arguments to the subtype method are supertypes of the arguments to the supertype method. However, it makes sense, because any arguments passed to the supertype method are sure to be legal arguments to the subtype method.
   - guarantees more (has a stronger postcondition)
     - there are no more exceptions
     - there are fewer modified variables
     - in the description of the result and/or result state, there are more clauses, and they describe stronger properties

– the result type may be a subtype of that of the supertype. This is called covariance: the return type of the subtype method is a subtype of the return type of the supertype method.

(The above descriptions should all permit equality; for instance, "requires less" should be "requires no more", and "less strict" should be "no more strict". They are stated in this form for ease of reading.)

The subtype method should not promise to have more or different results; it merely promises to do what the supertype method did, but possibly to ensure additional properties as well. For instance, if a supertype method returns a number larger than its argument, a subtype method could return a prime number larger than its argument.

As an example of the type constraints, if `A` is a subtype of `B`, then the following would be a legal overriding:

```
Bicycle B.f(Bicycle arg);
RacingBicycle A.f(Vehicle arg);
```

Method `B.f` takes a `Bicycle` as its argument, but `A.f` can accept any `Vehicle` (which includes all `Bicycles`). Method `B.f` returns a `Bicycle` as its result, but `A.f` returns a `RacingBicycle` (which is itself a `Bicycle`).

properties Any properties guaranteed by a supertype, such as constraints over the values that may appear in specification fields, must be guaranteed by the subtype as well. (The subtype is permitted to strengthen these constraints.)

As a simple example from the book, consider `FatSet`, which is always nonempty.

```
class FatSet {
  // Specification constraints: this always contains at least one element

  ...

  // effects: if this contains x and this.size > 1, removes x from this
  void remove(int x);
}
```

Type `SuperFatSet` with additional method

```
// effects: removes x from this
void reallyRemove(int x)
```

is not a subtype of `FatSet`. Even though there is no problem with any method of `FatSet` — `reallyRemove` is a new method, so the rules about corresponding methods do not apply — this method violates the constraint.

If the subtype object is considered purely as a supertype object (that is, only the supertype methods and fields are queried), then the result should be the same as if an object of the supertype had been manipulated all along instead.

In Section 7.9, the book describes the substitution principle as placing constraints on

- signatures: this is essentially the contravariant and covariant rules above. (A procedure's signature is its name, argument types, return types, and exceptions.)

- methods: this is constraints on the behavior, or all aspects of a specification that cannot be expressed in a signature
- properties: as above

5 Java subclasses and subtypes

Java types are classes, interfaces, or primitives. Java has its own notion of subtype (which involves only classes and interfaces). This is a weaker notion than the true subtyping described above; Java subtypes do not necessarily satisfy the substitution principle. Further, a subtype definition that satisfies the substitution principle may not be allowed in Java, and will not compile.

In order for a type to be a Java subtype of another type, the relationship must be declared (via Java's `extends` or `implements` syntax), and the methods must satisfy two properties similar to, but weaker than, those for true subtyping:

1. for each method in the supertype, the subtype must have a corresponding method. (The subtype is allowed to introduce additional, new methods that do not appear in the supertype.)
2. each method in subtype that corresponds to one in the supertype
    - the arguments must have the same types
    - the result must have the same type
    - there are no more declared exceptions

Java has no notion of a behavioral specification, so it performs no such checks and can make no guarantees about behavior. The requirement of type equality for arguments and result is stronger than strictly necessary to guarantee type-safety. This prohibits some code we might like to write.. However, it simplifies the Java language syntax and semantics.

Subclassing has a number of advantages, all of which stem from reuse:

- Implementations of subclasses need not repeat unchanged fields and methods, but can reuse those of the superclass
- Clients (callers) need not change code when new subtypes are added, but can reuse the existing code (which doesn't mention the subtypes at all, just the supertype
- The resulting design has better modularity and reduced complexity, because designers, implementers, and users only have to understand the supertype, not every subtype; this is specification reuse.

A key mechanism that enables these benefits is overriding, which specializes behavior for some methods. In the absence of overriding, any change to behavior (even a compatible one) could force a complete reimplementation. Overriding permits part of an implementation to be changed without changing other parts that depend on it. This permits more code and specification reuse, both by the implementation and the client.

A potential disadvantage of subclassing is the opportunities it presents for inappropriate reuse. Subclasses and superclasses may depend on one another (explicitly by type name or implicitly by knowledge of the implementation), particularly since subclasses have access to the protected parts of the superclass implementation. These extra dependences complicate the MDD, the design, and the implementation, making it harder to code, understand, and modify.

6 Java interfaces

Sometimes you want guarantees about behavior without sharing code. For instance, you may want to require that elements of a specific container be ordered or support a particular operation, without providing a default implementation (because every ordering relationship has a different implementation). Java provides interfaces to fill this need. Interfaces guarantee no code reuse. Another advantage of interfaces is that a class may implement multiple interfaces and an interface may extend multiple interfaces; by contrast, a class can only extend one class. It turns out that in practice, implementing multiple interfaces and extending a single superclass provides most of the benefits of arbitrary inheritance, but with a simpler implementation and language semantics. A disadvantage of interfaces is that they provide no way to specify the signature (or behavior) of a constructor.

Lecture 16: Case Study: The Java Collections API

You can't be a competent Java programmer without understanding the crucial parts of the Java library. The basic types are all in *java.lang*, and are part of the language proper. The package *java.util* provides collections – sets, lists and maps – and you should know it well. The package *java.io* is also important, but you can get away with only a rough familiarity with what's in it, delving in as needed.

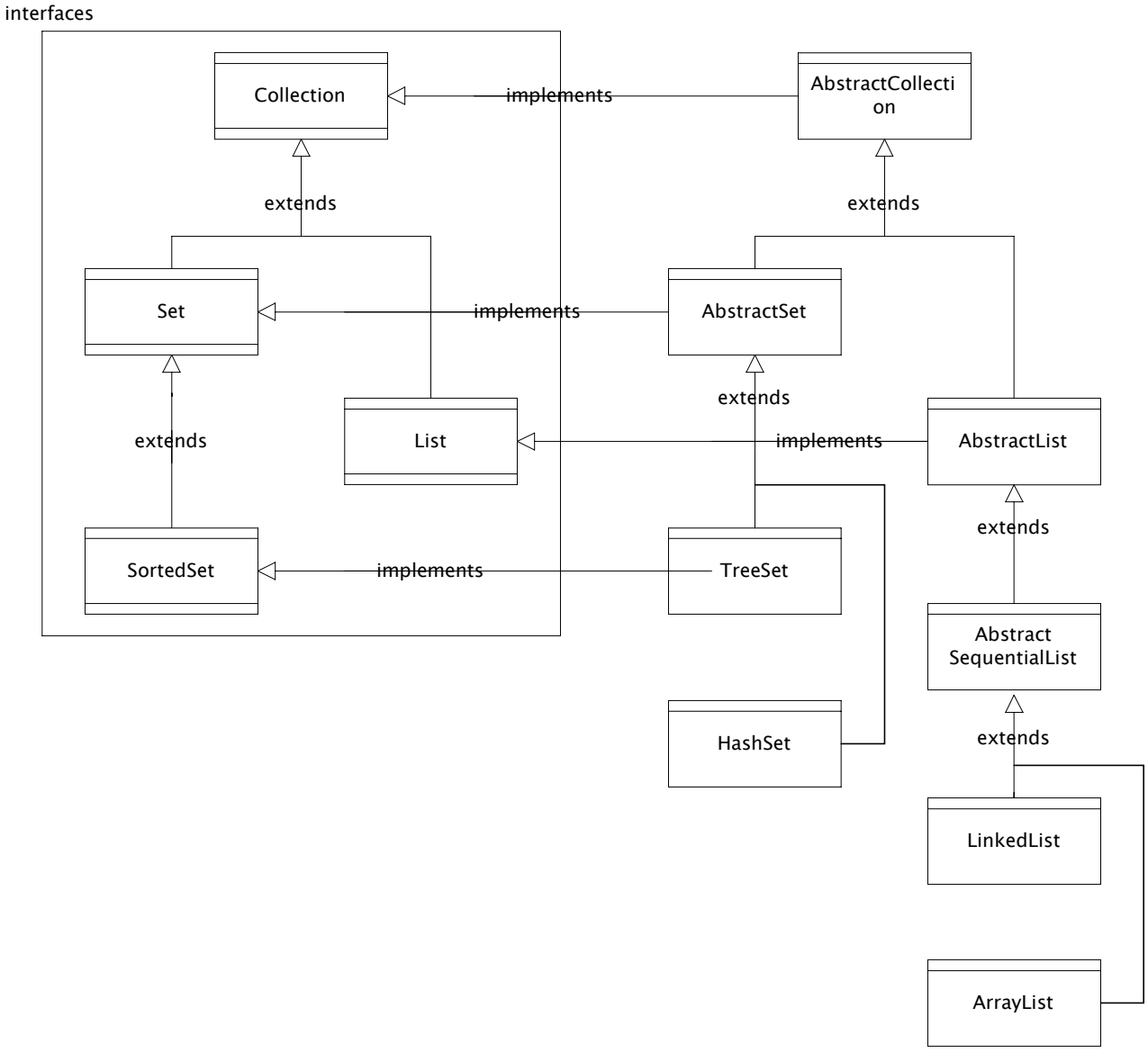
In this lecture, we'll look at the design of java.util, often called the Java 'collections API'. It's worth understanding not only because the collection classes are extremely useful, but also because the API is a nice example of well-engineered code. It's fairly easy to understand, and is very well documented. It was designed and written by Joshua Bloch, who wrote the *Effective Java* book that we recommended at the start of term.

At the same time, though, almost all the complexities of object-oriented programming appear somewhere in it, so if you study the API carefully, you'll get a broad understanding of programming issues that you probably haven't yet considered in your own code. In fact, it wouldn't be an exaggeration to say that if you figure out how just one of the classes (eg, *ArrayList*) works in its entirety, then you will have mastered all the concepts of Java. We won't have time to look at all the issues today, but we'll touch on many of them. Some of them, such as serialization and synchronization, are beyond the scope of the course.

16.1 Type Hierarchy

Roughly, the API offers three kinds of collection: sets, lists and maps. A set is a collection of elements that does not maintain their order or their count – each element is either in the set or not. A list is a sequence of elements, and thus maintains both order and count. A map is an association between keys and values: it holds a set of keys, and maps each key to a single value.

The API organizes its classes with a hierarchy of interfaces – the specifications of the various types – and a separate hierarchy of implemention classes. The diagram shows some select classes and interfaces to illustrate this. The interface *Collection* captures the common properties of lists and sets, but not maps, but we'll use the informal term



'collections' to refer to maps anyway. *SortedMap* and *SortedSet* are used for maps and sets which provide additional operations to retrieve the elements in some order.

The concrete implementation classes, such as *LinkedList*, are built on top of skeletal implementations, such as *AbstractList*, from which they inherit. This parallel structure of interfaces and classes is an important idiom that is worth studying. Many novice

programmers are tempted to use abstract classes when they should be using interfaces. But in general, you should prefer interfaces to abstract classes. You can't easily retrofit an existing class to extend an abstract class (because a class can have at most one superclass), but it's usually easy to make it implement a new interface.

Bloch shows (in Item 16 of his book: 'Prefer interfaces to abstract classes') how to combine the advantages of both, using skeletal implementation classes, as he does here in the Collections API. You get the advantage of interfaces for specification-based decoupling, and the advantage of abstract classes to factor out shared code amongst related implementations.

Each Java interface comes with an informal specification in the Java API documentation. This is important because it tells a user of a class that implements an interface what to expect. If you implement a class and claim that it meets the specification *List*, for example, you have to ensure that it meets the informal specification too, otherwise it will fail to behave according to programmers' expectations.

These specifications are intentionally incomplete (as many specifications often are). The concrete classes also have specifications, and these fill in some of the details of the interface specifications. The *List* interface, for example, doesn't say whether null elements can be stored, but *ArrayList* and *LinkedList* say explicitly that nulls are allowed. *HashMap* allows both null keys and null values, unlike *Hashtable*, which allows neither.

When you write code that uses collection classes, you should refer to an object by the most general interface or class possible. For example,

```
List p = new LinkedList ();
```

is better style than

```
LinkedList p = new LinkedList ();
```

If your code compiles with the former, then you can easily change to a different list implementation later:

```
List p = new ArrayList ();
```

since all the following code relied only on *p* being a *List*. With the latter, however, you may find that you can't make the change, because some other part of your program performs an operation on *x* that only *LinkedList* provides – an operation that in fact might not be needed. This is explained in more detail in Item 34 of Bloch's book ('Refer to objects by their interfaces').

We'll see a more sophisticated example of this in the Tagger case study next week, where part of the code requires access to the keys of a *HashMap*. Rather than passing the whole map, we pass a view of the map of type *Set*:

```
Set keys = map.keySet ();
```

Now the code that uses *keys* does not even know that this set is the set of keys of a map.

16.2 Optional Methods

The collections API allows a class to claim to implement a collections interface without implementing all of its methods. For example, all the mutators of *List* are specified as *optional*. This means you can implement a class that satisfies the *List* specification, but which throws an *UnsupportedOperationException* whenever you call a mutator, such as *add*.

This intentional weakening of the *List* specification is problematic, because it means that if you're writing some code that receives a list, you don't know, in the absence of additional information about the list, whether it will support *add*.

But without this notion of optional operations, you'd have to declare a separate interface *ImmutableList*. These interfaces would proliferate. Sometimes, we want to require some mutators but not others. For example, the *keySet* method of *HashMap* returns a *Set* containing the keys of the map. This set is a view: deleting a key from the set causes a key and its associated value to be deleted from the map. So *remove* is supported. But *add* is unsupported, since you can't add a key to a map without an associated value.

So the use of optional operations is a reasonable engineering judgment. It means less compile-time checking, but it reduces the number of interfaces.

16.3 Polymorphism

All these containers – sets, lists and maps – take elements of type *Object*. They are said to be *polymorphic*, meaning 'many shaped', because they allow you to make different kinds of containers: lists of Integers, lists of URLs, lists of lists, and so on.

This kind of polymorphism is called *subtype polymorphism*, because it relies on the type hierarchy. A different form of polymorphism, called *parametric polymorphism*, allows you to define containers with type parameters, so that a client can indicate what type of element a particular containers will contain:

```
List[URL] bookmarks; // not legal Java
```

Java doesn't have this kind of polymorphism, although there have been many proposals to add it. Parametric polymorphism has the big advantage that the programmer can tell the compiler what type the elements have. The compiler can then catch errors in which an element of the wrong type is inserted, or an element that is extracted is treated as having a different type.

With subtype polymorphism, you have to explicitly *cast* the elements on extraction. Consider this code:

```
List bookmarks = new LinkedList ();
URL u = ...;
bookmarks.add (u);
...
URL x = bookmarks.get (0); // compiler will reject this
```

The statement that adds *u* is fine, since the *add* method expects an *Object*, and *URL* is a subclass of *Object*. The statement that gets *x*, however, is broken, since the type of the expression on the RHS is *Object*, and you can't assign an *Object* to a variable of type *URL*, since then you couldn't rely on that variable holding a *URL*. So a downcast is needed, and we have to write instead:

```
URL x = (URL) bookmarks.get (0);
```

The effect of the downcast is to perform a runtime check. If it succeeds, and the result of the method call is of type *URL*, execution continues normally. If it fails, because the result is not of the correct type, a *ClassCastException* is thrown, and the assignment is not performed. Make sure you understand this, and don't get confused into thinking (as students often do) that the cast somehow mutates the object returned by the method invocation. Objects carry their type at runtime, and if an object was created with a constructor from the class *URL*, it will have that type, and there is no need to somehow 'change it' to give it that type.

These downcasts can be a nuisance and occasionally it's worth writing a wrapper class just to factor them out. In a browser, you'd probably want a special abstract data type for a list of bookmarks anyway (to support other functionality). If you did this, you would perform the cast within the abstract type, and clients would see methods such as

```
URL getURL (int i);
```

which would not require the cast in their calling contexts, thus limiting the scope in which cast errors can occur.

Subtype polymorphism does give some flexibility that parametric polymorphism does not. You can form *heterogeneous* containers that contain different kinds of elements. And you can put containers inside themselves – try and figure out how to express this as a polymorphic type – although this is not usually a wise thing to do. In fact, as we mentioned in our earlier lecture on equality, the Java API classes will break if you do this.

Writing down what type of an element a container has is often the most important part of an abstract type's rep invariant. You should get into the habit of writing a comment whenever you declare a container, either using a pseudo-parametric type declaration:

List bookmarks; // List [URL]

or as part of the rep invariant proper:

RI: bookmarks.elems in URL

16.4 Skeletal Implementations

The concrete implementations of the collections build on skeletal implementations. These use the *Template Method* design pattern (see Gamma et al, pages 325–330). An abstract class has no instance variables of its own, but defines 'template methods' that call other 'hook methods' that are declared to be abstract and have no code. In the inheriting subclass, the hook methods are overridden, and the template methods are inherited unchanged.

AbstractList, for example, makes *iterator* a template method that returns an iterator implemented using the *get* method as a hook. The *equals* method is implemented as another template in terms of *iterator*. A subclass, such as *ArrayList*, then provides a representation (such as an array of elements) and an implemenation for *get* (such as getting the *i*th element of the array), and can inherit *iterator* and *equals*.

Some concrete classes replace the abstract implementations. *LinkedList*, for example, replaces the iterator functionality, since, with using the representation of list entries directly, it's possible to write a much more efficient traversal than using the hook method *get*, which does a sequential search for each call!

16.5 Capacity, Allocation & GC

An implementation that uses an array for its representation – such as *ArrayList* or *HashMap* – must select a size for the array when it is allocated. Choosing a good size can be important for performance. If it's too small, the array will have to be replaced

by a new array, incurring the cost of allocating the new one and garbage collecting the old one. If it's too large, space will be wasted, which will be a problem especially when there are many instances of the collection type.

Such implementations therefore provide constructors in which the client can set an initial capacity, from which the allocation size can be determined. *ArrayList*, for example, has the constructor

public ArrayList(int initialCapacity)
Constructs an empty list with the specified initial capacity.
Parameters:
initialCapacity - the initial capacity of the list.
Throws:
IllegalArgumentException - if the specified initial capacity is negative

There are also methods that adjust the allocation: *trimToSize*, which sets the capacity so that the container is just large enough for the elements currently stored, and *ensureCapacity*, which increases capacity to some given amount.

Using the capacity features is tricky. If you don't have precise knowledge of how big your collections are for the particular application, you can run a profiler to find out.

Note that this notion of capacity translates a behavioural problem into a performance problem – a very desirable tradeoff. In many old programs, there are fixed resource limits, and when they are reached, the program just fails. With the capacity approach, the program just slows down. It's a good idea to design a program so that it works efficiently almost all the time, even if there's a performance hit occasionally.

If you study the implementation of the *remove* method in *ArrayList*, you'll see this code:

```
public Object remove(int index) {
    ...
    elementData[—size] = null; // Let gc do its work
    ...
```

What's going on? Isn't garbage collection automatic? Herein lies a common mistake of many novice programmers. If you have an array in your representation, with a distinct instance variable holding an index to indicate which elements of the array are to be considered part of the abstract collection, it's tempting to think that to remove elements all you need to do is decrement this index. An analysis in terms of the abstraction function will support this confusion: elements that fall above the index are, after

all, not considered part of the abstract collection, and their values are irrelevant.

There's a snag, however. If you fail to assign null to the unused slots, the elements whose references sit in those slots will not be garbage collected, even if there are no other references to those elements elsewhere in the program. The garbage collector can't read the abstraction function, so it doesn't know that those elements are not really reachable from the collection, even though they are reachable in the representation. If you forget to null out these slots, the performance of your program may suffer badly.

16.6 Copies, Conversions, Wrappers, etc

All the concrete collection classes provide constructors that take collections as arguments. These allow you to copy collections, and to convert one collection type to another. For example, *LinkedList* has

public LinkedList(Collection c)
Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
Parameters:
c – the collection whose elements are to be placed into this list.

which can be used for copying:

```
List p = new LinkedList ()
...
List pCopy = new LinkedList (p)
```

of for creating a linked list from some other collection type:

```
Set s = new HashSet ()
...
List p = new LinkedList (s)
```

Since constructors cannot be declared in interfaces, the specification *List* doesn't say that all of its implementations should have such constructors, although they do.

There is a special class *java.util.Collections* that contains a bunch of static methods that operate on, or return collections. Some of these are generic algorithms (eg, for sorting), and some are wrappers. For example, the method *unmodifiableList* takes a list and returns a list with the same elements, but which is immutable:

public static List unmodifiableList(List list)
Returns an unmodifiable view of the specified list. This method allows modules

to provide users with "read-only" access to internal lists. Query operations on the returned list "read through" to the specified list, and attempts to modify the returned list, whether direct or via its iterator, result in an UnsupportedOperationException.

The returned list will be serializable if the specified list is serializable.

Parameters:

list - the list for which an unmodifiable view is to be returned.

Returns:

an unmodifiable view of the specified list.

The list returned isn't exactly immutable, since it's value can change because of modifications to the underlying list (see Section 16.8below), but it can't be modified directly. There are similar methods that take collections and return wrapped versions that are synchronized.

16.7 Sorted Collections

A sorted collection must have some way to compare elements to determine their order. The Collections API offers two approaches to this. You can use the 'natural ordering', which is determined by using the *compareTo* method on the element type from the interface *java.lang.Comparable*:

public int compareTo(Object o)

which returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the given object *o*. When you add an element to a sorted collection that is using the natural ordering, the element must have been constructed in a class that implements the *Comparable* interface. The *add* method downcasts the element to *Comparable* in order to compare it with the existing elements in the collection, so if this was not the case, it will throw a class cast exception.

Alternatively, you can use an ordering given independently of the elements, as an object that implements the interface *java.util.Comparator*, which has the method

public int compare(Object o1, Object o2)

which is just like *compareTo*, but takes both elements to be compared as arguments. This is an instance of the *Strategy* pattern, in which an algorithm is decoupled from the code that uses it (see Gamma, pp. 315–323).

Which approach is used depends on which constructor you use to create the collection

object. If you use the constructor that takes a *Comparator* as an argument, that will be used to determine the ordering; if you use the no-argument constructor, the natural ordering will be used.

Comparison suffers from the same problems as equality, which we discussed in detail in Lecture 9. A sorted collection has a rep invariant that the elements of the representation are sorted. If the ordering of two elements can be changed by mutating one of them through a public method call, a rep exposure occurs.

16.8 Views

We introduced the notion of views in Lecture 9. Views are a sophisticated mechanism, very useful now and then, but dangerous. They break many of our basic conceptions about what kinds of behaviour can occur in a well-formed object-oriented program.

Three kinds of views can be identified, according to their purpose:

- *Functionality extension*. Some views are provided to extend the functionality of an object without adding new methods to its class. Iterators fall in this category. One could instead put the methods *next* and *hasNext* in the collection class itself. But this would complicate the API of the class itself. It would also be hard to support multiple iterations over the same collection. We could add a *reset* method to the class which is called to restart an iteration, but this would only allow one iteration at a time. Such a method would also lead to errors in which the programmer forgets to reset.
- *Decoupling*. Some views provide a subset of the functionality of the underlying collection. The *keySet* method on *Map*, for example, returns a set that consists of the keys of the map. It therefore allows part of the code that is only concerned with the keys, and not with the values, to be decoupled from the rest of the specification of *Map*.
- *Coordinate Transformation*. The view provided by the *subList* method of *List* gives a kind of coordinate transformation. Mutations on the view produce mutations on the underlying list, but allow access to the list by an indexing that is offset by the parameter passed to the *subList* method.

Views are dangerous for two reasons. First, things change underneath you: call *remove* on an iterator and its underlying collection changes; call *remove* on a map and its key set view changes (and vice versa). This is a form of abstract aliasing in which a mutation to one object causes another object, of a different type to change. The two objects need not even within the same lexical scope. Note that the meaning of our modifies clause in specifications must be refined: if you say *modifies c* and *c* has a view *v*, does that mean that *v* can change also?

Second, the specification of a method that returns a view often limits the kinds of mutation that are allowed. To make sure that your code works, you'll need to understand this specification. And not surprisingly, these specifications are often obscure. The post-requires clause of the Liskov text is one way to extend our specification notion to handle some of the complications.

Some views allow only the underlying collection to be mutated. Others allow only the view to be mutated – iterators, for example. Some allow mutations to both the view and the underlying collection, but place complex stipulations on the mutations. The Collections API, for example, says that when a sublist view has been taken on a list, this underlying list must not suffer any 'structural modifications'; it explains this term rather obliquely as follows:

> *Structural modifications are those that change the size of this list, or otherwise perturb it in such a fashion that iterations in progress may yield incorrect results.*

It's not clear exactly what this means. My inclination would be to avoid any modifications of the underling list.

The situation is complicated further by the possibility of multiple views on the same underlying collection. You can have multiple iterators on the same list, for example. In this case, you have to also consider interactions *between* views. If you modify the list through one of its iterators, the other iterators will be invalidated, and must not be used subsequently.

There are some useful strategies that mitigate the complexity of views. If you are using a view, you should think carefully about whether these will help:

- You can limit the scope in which the view is accessible. For example, by using a for-loop rather than a while-loop for an iterator, you can limit the scope of the iterator's declaration to the loop itself. This makes it much easier to ensure that there aren't any unintended interactions during iteration. This isn't always possible; the Tagger program that we'll discuss next week mutates an iterator several method calls away, and in a different class, from its creation site!
- You can prevent mutation of a view or underlying object by wrapping it using a method of the *Collections* class. For example, if you take a *keySet* view on a map, and don't intend to modify it, you could make the set immutable:

```
Set s = map.keySey ();
Set safe_s = Collections.unmodifiableSet (s);
```

Lecture 17: Case Study: JUnit

The JUnit testing framework which you've been using to test your own code in 6.170 is worth studying in its own right. It was developed by Kent Beck and Erich Gamma. Beck is an exponent of patterns and Extreme Programming (XP); Gamma is one of the authors of the celebrated design patterns book. JUnit is open source, so you can study the source code yourself. There's also a nice explanatory article in the JUnit distribution, entitled 'A Cook's Tour', which explains the design of JUnit in terms of design patterns, and from which much of the material in this lecture is drawn.

JUnit has been a great success. Martin Fowler, an insightful and pragmatic proponent of patterns and XP (and also author of a wonderful book on object models called *Analysis Patterns*), says about JUnit:

> *Never in the field of software development was so much owed by so many to so few lines of code.*

JUnit's ease of use is no doubt in large part responsible for its popularity. You might think that. since it doesn't do very much – it just runs a bunch of tests and reports their results – JUnit should be very simple. In fact, the code is rather complicated. The main reason for this is that it has been designed as a framework, to be extended in many unanticipated ways, and so it's full of rather complex patterns and indirections designed to allow an implementer to override some parts of the framework while preserving other parts.

Another complicating influence is a desire to make tests easy to write. There's a clever hack involving reflection that turns methods of a class into individual instances of the type *Test*. Here's another example of a hack that seems unconscionable at first. The abstract class *TestCase* inherits from the class *Assert*, which contains a bunch of static assertion methods, simply to allow a call to the static *assert* method to be written as just *assert (...)*, rather than *Assert.assert (...)*. In no sense is *TestCase* a subtype of *Assert*, of course, so this really makes no sense. But it does allow code within *TestCase* to be written more succinctly. And since all the test cases the user writes are methods of the *TestCase* class, this is actually pretty significant.

The use of patterns is skillful and well motivated. The key patterns we'll look at are: *Template Method*, the key pattern of framework programming; *Command*, *Composite*, and *Observer*. All these patterns are explained at length in Gamma et al, and, with the

exception of *Command*, have been covered already in this course.

My personal opinion is that JUnit, the jewel in the crown of XP, itself belies the fundamental message of the movement – that code alone is enough. It's a perfect example of a program that is almost incomprehensible without some abstract, global representations of the design explaining how the parts fit together. It doesn't help that the code is pretty lean on comments – and where are there comments they tend to dwell on which Swiss mountain the developer was sitting on when the code was written. Perhaps high altitude and thin air explains the coding style.The 'Cook's Tour' is essential; without it, it would take hours to grasp the subtleties of what's going on. And it would be helpful to have even more design representations. The 'Cook's Tour' presents a simplified view, and I had to construct for myself an object model explaining, for example, how the listeners work.

If you're one of those students who's skeptical about design representations, and who still thinks that code is all that matters, you should stop reading here, and curl up in a chair to spend an evening with JUnit's source code. Who knows, it may change your mind…

You can download the source code and documentation for JUnit from

http://www.junit.org/.

There's an open source repository at

http://sourceforge.net/projects/junit/

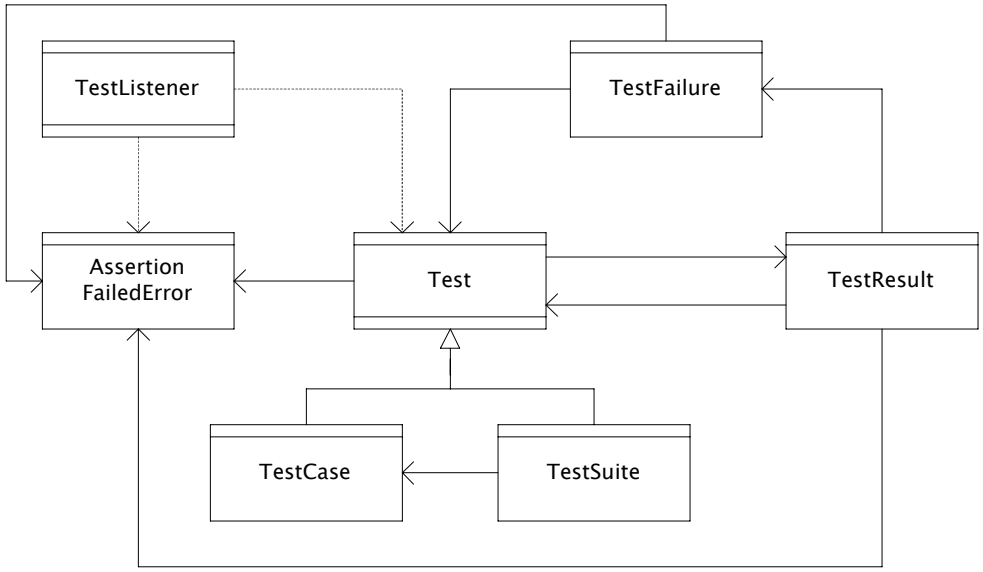
where can view (and contribute) bug reports.

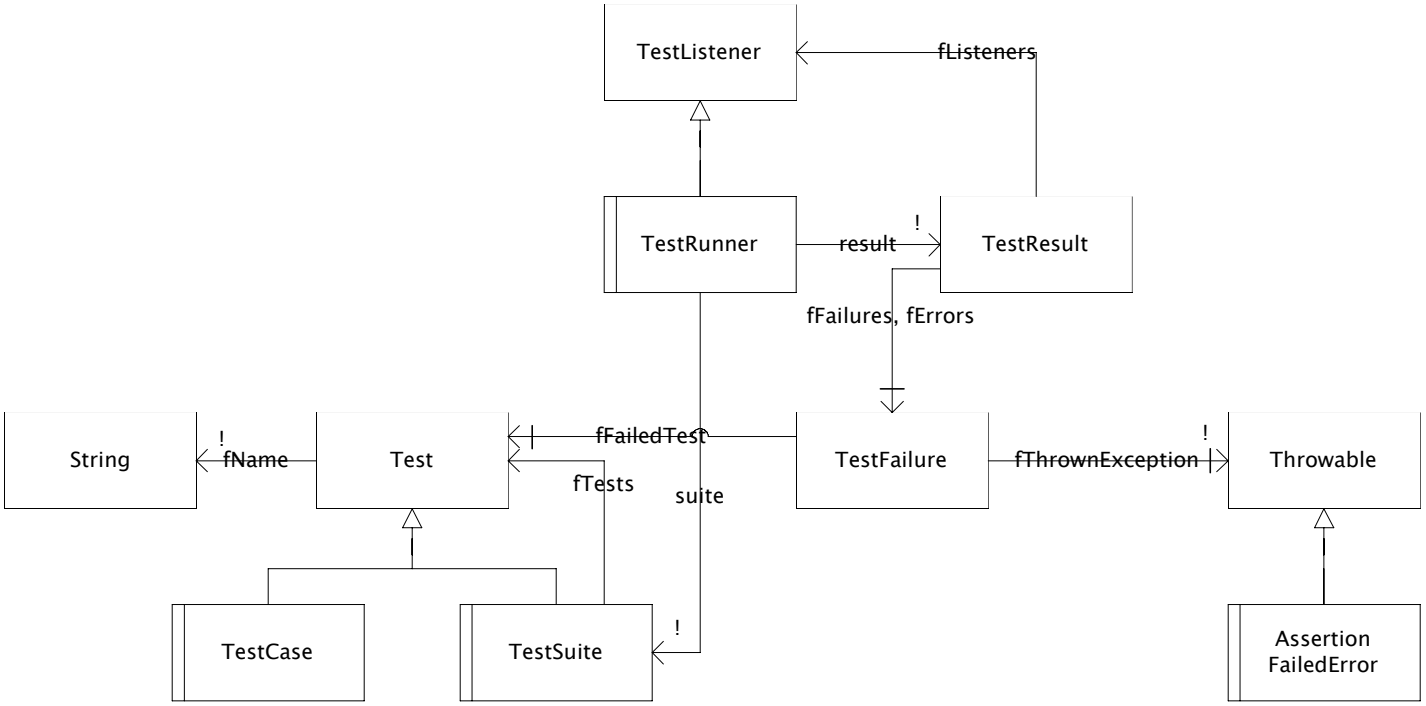
17.1 Overview

JUnit has several packages: *framework* for the basic framework, *runner* for some abstract classes for running tests, *textui* and *swingui* for user interfaces, and *extensions* for some useful additions to the framework. We'll focus on the *framework* package.

The diagrams show the object model and module dependences. You may want to follow along with these diagrams as you read our discussion. Both of these include only the framework modules, although I've included *TestRunner* in the object model to show how the listeners are connected; it's relations, *suite* and *result* are local variables of its *doRun* method.

Note that the module dependency diagram is almost fully connected. This is not surprising for a framework; the modules are not intended to be used independently.


TestListener
TestFailure
Assertion
FailedError
Test
TestResult
TestCase
TestSuite


TestListener
fListeners
TestRunner
result
!
TestResult
fFailures, fErrors
String
!
fName
Test
fFailedTest
fTests
suite
TestFailure
fThrownException
!
Throwable
TestCase
TestSuite
!
Assertion
FailedError

17.2 Command

The command pattern encapsulates a function as an object. It's how you implement a closure – remember that from 6.001? – in an object-oriented language. The command class typically has a single method with a name like *do*, *run* or *perform*. An instance of a subclass is created that overrides this method, and usually also encapsulates some state (in 6.001 lingo, the environment of the closure). The command can then be passed around as an object, and 'executed' by calling the method.

In JUnit, test cases are represented as command objects that implement the interface *Test*:

```
public interface Test {
    public void run();
    }
```

Actual test cases are instances of a subclass of a concrete class *TestCase*:

```
public abstract class TestCase implements Test {
    private String fName;
    public TestCase(String name) {
        fName= name;
    }

    public void run() {
        ...
        }
}
```

In fact, the actual code isn't quite like this, but starting from this simplified version will allow us to explain the basic patterns more easily. Note that the constructor associates a name with the test case, which will be useful when reporting results. In fact, all the classes that implement *Test* have this property, so it might have been good to add a method

```
public String getName ()
```

to the *Test* interface. Note also that the authors of JUnit use the convention that identifiers that begin with a lowercase *f* are fields of a class (that is, instance variables).

We'll see a more elaborate example of the command pattern when we study the *Tagger* program next week.

17.3 Template Method

One might make *run* an abstract method, thus requiring all subclasses to override it. But most test cases have three phases: setting up the context, performing the test, then tearing down the context. We can factor out this common structure by making *run* a *template method*:

```
public void run() {
    setUp();
    runTest();
    tearDown();
}
```

The default implementations of the hook methods do nothing:

```
protected void runTest() { }
protected void setUp() { }
protected void tearDown() { }
```

They are declared as protected so that they are accessible from subclasses (and can thus be overridden) but not accessible from outside the package. It would be nice to be able to prevent access except from subclasses, but Java doesn't offer such a mode. A subclass can selectively override these methods; if it overrides only *runTest*, for example, there will be no special *setUp* or *tearDown* behaviour.

We saw this same pattern in the last lecture in the skeletal implementations of the Java collections API. It is sometimes referred to in a rather corny way as the *Hollywood Principle*. A traditional API provides methods that get called by the client; a framework, in contrast, makes calls to the methods of its client: 'don't call us, we'll call you'. Pervasive use of templates is the essence of framework programming. It's very powerful, but also easy to write programs that are completely incomprehensible, since method implementations make calls at multiple levels in the inheritance hierarchy.

It can be difficult to know what's expected of a subclass in a framework. An analog of pre- and post-conditions hasn't been developed, and the state of the art is rather crude. You usually have to read the source code of the framework to use it effectively. The Java collections API does better than most frameworks, by including in the specifications of template methods some careful descriptions of how they are implemented. This is of course anathema to the idea of abstract specification, but it's unavoidable in the context of a framework.

17.4 Composite

As we discussed in Lecture 11, test cases are grouped into test suites. But what you do with a test suite is essentially the same as what you do with a test: you run it, and you report the result. This suggests using the *Composite* pattern, in which a composite object shares an interface with its elementary components.

Here, the interface is *Test*, the composite is *TestSuite*, and the elementary components are members of *TestCase*. *TestSuite* is a concrete class that implements *Test*, but whose *run* method, unlike the *run* method of *TestCase*, calls the *run* method of each test case that the suite contains. Instances of *TestCase* are added to a *TestSuite* instance with the method *addTest*; there's also a constructor that creates a *TestSuite* with a whole bunch of test cases, as we'll see later.

The example of *Composite* in the Gamma book has the interface include all the operations of the composite. Following this approach, *Test* should include methods like *addTest*, which apply only to *TestSuite* objects. The implementation section of the pattern description explains that there is a tradeoff between transparency – making the composite and leaf objects look the same – and safety – preventing inappropriate operations from being called. In terms of our discussion in the subtyping lecture, the question is whether the interface should be a true supertype. In my opinion it should be, since the benefits of safety outweigh those of transparency, and, moreover, the inclusion of composite operations in the interface is confusing. JUnit follows this approach, and does not include *addTest* in the interface *Test*.

17.5 Collecting Parameter

The *run* method of *Test* actually has this signature:

public void run(TestResult result);

It takes a single argument that is mutated to record the result of running the test. Beck calls this a 'collecting parameter' and views it as a design pattern in its own right.

There are two ways in which a test can fail. Either it produces the wrong result (which may include not throwing an expected exception), or it throws an unexpected exception (such as *IndexOutOfBoundsException*). JUnit calls the former 'failures' and the latter 'errors'. An instance of *TestResult* contains a sequence of failures and a sequence of errors, each failure or error being represented as an instance of the class *TestFailure*, which contains a reference to a *Test* and a reference to the exception object generated by the failure or error. (Failures always produce exceptions, since even when an unexpected result is produced without an exception, the *assert* method used in the test con-

verts the mismatch into an exception).

The *run* method in *TestSuite* is essentially unchanged; it just passes the *TestResult* when invoking the *run* method of each of its tets. The *run* method in *TestCase* looks something like this:

```
public void run (TestResult result) {
    setUp ();
    try {
        runTest ();
        }
    catch
        (AssertionFailedError e) {
            result.addFailure (test, e);
            }
        (Throwable e) {
            result.addError (test, e);
            }
    tearDown ();
    }
```

In fact, the control flow of the template method *run* is more complicated than we have suggested. Here are some pseudocode fragments showing what happens. It ignores the *setUp* and *tearDown* activities, and considers a use of *TestSuite* within a textual user interface:

```
junit.textui.TestRunner.doRun (TestSuite suite) {
    result = new TestResult ();
    result.addListener (this);
    suite.run (result);
    print (result);
    }

junit.framework.TestSuite.run (TestResult result) {
    forall test: suite.tests
        test.run (result);
    }

junit.framework.TestCase.run (TestResult result) {
    result.run (this);
    }
```

```
junit.framework.TestResult.run (Test test) {
    try {
        test.runBare ();
        }
    catch (AssertionFailedError e) {
        addFailure (test, e);
        }
    catch (Throwable e) {
        addError (test, e);
        }
    }

junit.framework.TestCase.runBare (TestResult result) {
    setUp();
    try {
        runTest();
        }
    finally {
        tearDown();
        }
    }
```

TestRunner is a user interface class that calls the framework and displays the results. There's a GUI version *junit.swingui* and a simple console version *junit.textui*, which we've shown an excerpt from here. We'll come to the listener later; ignore it for now.

Here's how it works. The *TestRunner* object creates a new *TestResult* to hold the results of the test; it runs the suite, and prints the results. The *run* method of *TestSuite* calls the *run* method of each of its constituent tests; these may themselves be *TestSuite* objects, so the method may be called recursively. This is a nice illustration of the simplicity that *Composite* brings. Eventually, since there is an invariant that a *TestSuite* cannot contain itself – not actually specified, and not enforced by the code of *TestSuite* either – the method will bottom out by calling the *run* methods of objects of type *TestCase*.

The *run* method of *TestCase* now has the receiver *TestCase* object swap places with the *TestResult* object, and calls the *run* method of *TestResult* with the *TestCase* as an argument. (Why?). The *run* method of *TestResult* then calls the *runBare* method of *TestCase*, which is the actual template method that executes the test. If the test fails, it

throws an exception, which is caught by the *run* method in *TestResult*, which then packages the test and exception as a failure or error of the *TestResult*.

17.6 Observer

For an interactive user interface, we'd like to show the results of the test as it happens incrementally. To achieve this, JUnit uses the *Observer* pattern.

The *TestRunner* class implements an interface *TestListener* which has methods *addFailure* and *addError* of its own. It plays the role of *Observer*. The class *TestResult* plays the role of *Subject*; it provides a method

```
public void addListener(TestListener listener)
```

which adds an observer. When the *addFailure* method of *TestResult* is called, in addition to updating its list of failures, it calls the *addFailure* method on each of its observers:

```
public synchronized void addFailure(Test test, AssertionFailedError e) {
    fFailures.addElement(new TestFailure(test, e));
        for (Enumeration e= cloneListeners().elements(); e.hasMoreElements(); ) {
            ((TestListener)e.nextElement()).addFailure(test, e);
            }
}
```

In the textual user interface, the *addFailure* method of *TestRunner* simply prints a character *F* to the screen. In the graphical user interface, it adds the failure to a list display and changes the colour of the progress bar to red.

17.7 Reflection Hacks

Recall that a test case is an instance of the class *TestCase*. To create a test suite in plain old Java, a user would have to create a fresh subclass of *TestCase* for each test case, and instantiate it. An elegant way to do this is to use anonymous inner classes, creating the test case as an instance of a subclass that has no name. But it's still tedious, so JUnit provides a clever hack.

The user provides a class for each test suite – called *MySuite* say – that is a subclass of *TestCase*, and which contains many test methods, each having a name beginning with the string 'test'. These are taken to be individual test cases.

```
public class MySuite extends TestCase {
```

```
void testFoo () {
   int x = MyClass.add (1, 2);
   assertEquals (x, 3);
   }
void testBar () {
   ...
   }
}
```

The class object *MySuite* itself is passed to the *TestSuite* constructor. Using reflection, the code in *TestSuite* instantiates *MySuite* for each of its methods beginning with 'test', passing the name of the method as an argument to the constructor. As a result, for each test method, a fresh *TestCase* object is created, with its name bound to the name of the test method. The *runTest* method of *TestCase* calls, again using reflection, the method whose name matches the name of the *TestCase* object itself, roughly like this:

```
void runTest () {
   Method m = getMethod (fName);
   m.invoke ();
   }
```

This scheme is obscure, and dangerous, and not the kind of thing you should emulate in your code. Here it's justifiable, because it's limited to a small part of the JUnit code, and it brings a huge advantage to the user of JUnit.

17.8 Questions for Self-Study

These questions arose when I constructed the object model for JUnit. They don't all have clear answers.

- Why are listeners attached to *TestResult*? Isn't *TestResult* already a kind of listener itself?
- Can a *TestSuite* contain no tests? Can it contain itself?
- Are *Test* names unique?
- Does the *fFailedTest* field of *TestFailure* always point to a *TestCase*?

Lecture 18: Case Study: Tagger

18.1 Overview

In this lecture, I'll explain the design of *Tagger*, a small program that I wrote last summer. I've used Tagger for all my writing and publishing for the last few months. It was used for this document (6170 lecture notes), and for all papers I've written since June 2001 (see http://sdg.lcs.mit.edu/~dnj/publications).

I've chosen Tagger as our third case study for several reasons. First, it's a program I wrote myself, so I understand it better than others. Second, it provides demonstrations of several of the patterns and idioms that you have been studying; it has a nice use of views from the Java Collections API (our first case study). Third, unlike the previous two case studies, it's a less polished piece of work, and therefore more like what you should expect to produce in your final project. I spent a few days designing it, then a week building it.

A slide presentation is available that summarizes these notes.

18.2 Purpose

Tagger is a tiny text-processing application to aid in the production of technical papers and books. It is used as a front-end for WYSIWYG layout programs, such as QuarkXpress and Adobe Indesign, combining their benefits with some of the benefits of compilation-based tools such as TeX.

Tools like TeX are good because they allow you to edit the document in a powerful text editor, and exchange documents easily by email. Because formatting is indicated by textual tags, you can alter the formatting of a document using the same mechanisms – such as search and replace – that you use to alter the text itself. Mathematical symbols can be referred to symbolically (\alpha to get an α, for example), which generally speeds up typing, there being no need to select special characters form a palette, and decouples the document from the choice of mathematical font. Cross referencing is easily expressed by explicitly assigning symbolic names to paragraphs, and then using those symbolic names in citations.

On the other hand, tools like TeX have serious problems. They don't accommodate the

wide range of postscript fonts now available without considerable customization by the user. Adjusting the layout is rarely easy; a simple alteration, such as changing a margin or changing the spacing of headings, usually requires considerable expertise. And the typographic quality of the documents they produce is inferior to that of modern layout tools. Quark and Indesign, for example, both allow you to set a 'baseline grid' that lines snap to, so that the lines of text in facing columns on a page are lined up. Their hyphenation algorithms seem to perform better. Indesign gives access to all the features of OpenType fonts, and offers optical alignment.

Tagger's approach is very simple. The user writes a document in a simple markup language. The markup language offers almost no direct control over formatting, beyond commands to put text in bold, italic, etc. Instead, paragraphs are labelled with the names of *paragraph styles*. Tagger converts the document into a file in the import format of a layout program such as Quark. Within Quark, the user sets up a *stylesheet* that assigns the typographic features to each paragraph style. As the paragraphs are imported, they are typeset according to the appropriate style in the stylesheet.

Of course, one could simply write the import file for the layout program instead. But each layout program has a different import format. Although Tagger currently only generates input to Quark, it would be easy to add support for Indesign and other similar programs. Also, the import formats tend to be low-level, and they are much more cumbersome to write than our markup language. Strangely, the import format for Indesign cannot even be prepared in a text editor, since it relies on distinguishing linefeeds from carriage returns. Tagger also translates symbolic names for mathematical characters into font and index information; in the import format, instead of writing something like \alpha to display α, you'd need to give the name of the mathematical font and the index at which the character occurs.

18.3 Features

Tagger offers the following features:

- Tagging of paragraphs with style names. A separate stylesheet may specify for each style a default style that follows it, so that in many cases a paragraph need not be explicitly tagged.
- Automatic numbering of paragraphs. The stylesheet specifies which styles should be numbered, what the numbering hierarchy should be (eg, *section* above *subsection*), what style numbers should be generated in (alphabetic, arabic, roman, etc), and how numbering strings should be composed with appropriate leaders, trailers and separators.

- Symbolic naming of special characters. Mapping files translate symbolic names to font name/index pairs. Tagger comes with some standard mapping files, and it is straightforward to write new files to make characters from other fonts accessible.
- Math mode. Text written between dollar signs is treated as mathematical text. Alphabetic characters are italicized, but numbers, punctuation and special symbols are unchanged.
- Cross references. A paragraph can be marked with a label; a citation elsewhere using this label then generates a string referring to the labelled paragraph. By default, the string generated is the numbering string created by the automatic numbering facility, but the user may specify a string explicitly instead. This feature, in combination with automatic numbering, makes it easy to handle bibliographic references.
- Basic character formatting. Text can be put in italics, made a subscript, etc.
- Whitespace. Whitespace is mostly preserved, so it's easy to indent text with tabs or spaces.
- Shorthands. Several common shorthands are provided: three dots is automatically converted to an ellipsis, for example, two hyphens to an en-dash, and so on. Text between underscores is italicized. Inverted commas are resolved into the appropriate quote marks according to context: "It's good to be a 'software engineer' in '01".

18.4 Design Overview

The basic organization is very simple. The main class, *Tagger*, uses the *SourceParser* class to parse the input text into a stream of *Token* objects. Each *Token* object has a *TokenType*. The *Token* is passed to an *Engine* object, which maps the *TokenType* to a list of *Action* objects. Each of these *Action* objects is then executed. A typical effect of an *Action* is to generate textual output; this is done via an interface *Generator* that hides from the *Action* the actual choice of import formatting language. Currently, there is only one class that implements *Generator*, *QuarkGenerator*, which produces text for import into QuarkXpress.

Some *Action* objects cause files to be read; for example, the source text *\loadstyles{foo.txt}* causes the file *foo.txt* to be parsed as a style file. Style files and character maps have the same syntax: a sequence of lines, each consisting of a list of properties, each property being a pair consisting of a property name and a value. The contents of both kinds of file are represented as *PropertyMap* objects. Each such object contains a mapping from property names to property lists, being lists of *Property* objects, each consisting of a property name and a value. The class *PropertyParser* is a parser for property files.

The *Numbering* class creates numbering strings. An instance of the class is generated for each style file, since the style file contains numbering directives.

The module dependency diagram shows the modules of the Tagger program and their dependences on one another. The dotted contour groups together modules that share dependences: this allows us to avoid drawing a dependence arrow from the *Tagger* class to almost every other class. The numbers labelling dependence edges refer to comments in the list of unexpected dependences which explain why a dependence that one would not have expect to be present is in fact present. For example, the note on the dependence from *StandardEngine* to *Property* reminds you that the *StandardEngine* code (actually its anonymous inner classes that subtype *Action*) generates an index file of cross-reference information. For this, it needs access to *Property*. All other uses of *Property*, in the reading and processing of style files and character maps, are handled by inferior classes such as *PropertyMap*.

18.5 Design Features

Here are some notable features of the design. Some are illustrated with object models, sometimes of the code, and sometimes of the underlying conceptual structures.

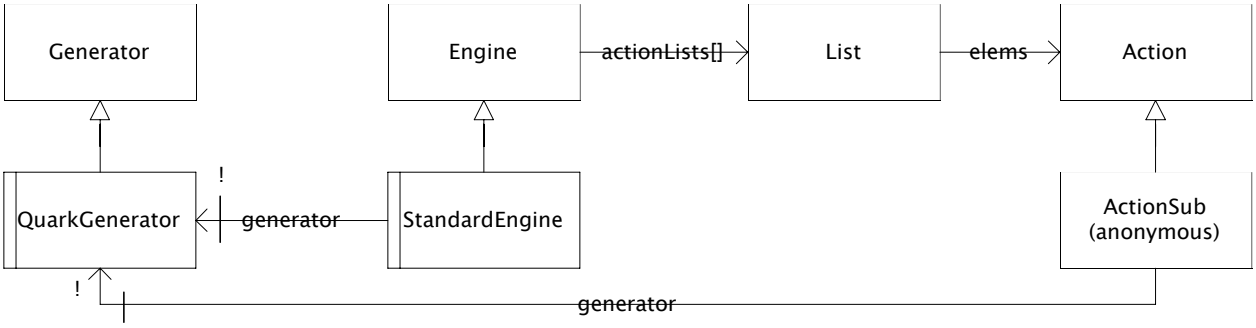
18.5.1 Generator Interface

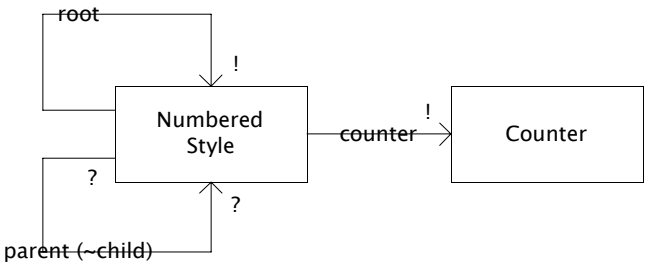
Actions interact with the backend generator through the *Generator* interface. This ensures that they are not dependent on the features of any particular backend generator. So long as the shared properties of different backends can be captured in the interface, it should be easy to adapt the application to generate output for a variety of different layout tools (such as InDesign and PageMaker in addition to Quark). A plaintext generator that produces simple ASCII text suitable for an email message would be easily written.

The object model shows the relationship amongst actions, concrete generators and the *Generator* interface.

18.5.2 Numbering

Each style is either numbered or not. If numbered, it belongs to a *series*. The series has a root style, and there is a chain of styles from the root; we'll say that if the chain goes from style *s* to style *t*, then *s* is the parent of *t* and *t* is the child of *s*. If there is only one style in a series, it is the root, and it has no children. This structure is specified in the style file simply by indicating what each style's parent is, if any, and giving it a counter





property if it is to be numbered. The numbering algorithm works as follows. A counter is associated with each numbered style, as is initialized with a value one less than the first value to be generated. When a paragraph of a given numbered style is encountered, its counter is incremented, and the counters of all its descendants are reset. A numbering string is constructed by concatenating its leader, the current counter values of each of its ancestors from the root to it, separated by separators, and its trailer. The leader, separator and trailer of each style are given in the style file.

The object model shows the relationships maintained by the *Numbering* object. The following invariants apply:

- If *s* is the parent of *t*, then *t* is the child of *s*, and vice versa.
- Every style points to a root style, which is either itself, it it has no parent, or the first ancestor without a parent.
- Series are disjoint: no style can belong to two series. So no two distinct styles can share a parent or child.
- No style is its own parent or child.

- If a style is numbered, its ancestors must be also.

18.5.3 Index Files

The problem of handling forward references is dealt with as in LaTeX. An index file that associates citation tags with the citation strings to be inserted in their place is generated during a run. Forward references are not resolved, but are picked up by running the tool again. New associations generated during the run are not checked against old ones from the index file, so it is possible for a run to produce bad cross-references after extensive editing. However, running the tool twice in a row will always produce correct results after the second run.

This scheme should be easily extendable to references across multiple source files.

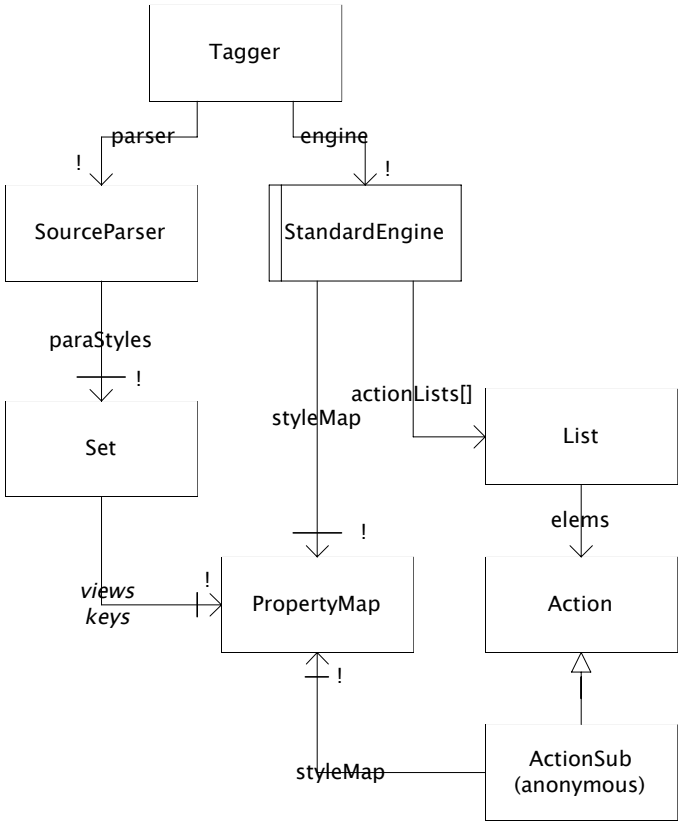
The problem object model shows the conceptual relationships involved in cross reference generation. Each paragraph may have a label and may have a numbering string; one of these is used for displaying citations to the paragraph, with the label overriding the numbering string. A paragraph may have a tag for citation by other paragraphs, and may cite the tags of other paragraphs. A paragraph p references another paragraph q if p cites t and t is the tag of q. Note that tags cannot be shared by more than one paragraph; they are unique identifiers.

18.5.4 Property Maps

Style sheets and character maps have the same syntax, and are represented internally by the same abstract data type, *PropertyMap*. This allows the same parser to be used for both. The index files generated by the cross-referencing mechanism also use the same syntax and representation. The *Numbering* class actually augments the internal representation of the style sheet by adding new, redundant properties to make it easier to generate numbering strings. This is a bit of a hack.

18.6 Style Set View

The *SourceParser* must be able to distinguish paragraph style names from other commands, since the syntax does not require that they be specially marked in any way. It is therefore constructed with a reference to a *Set* of style names. At the start, however, the style names are not known. When a style sheet is loaded, the *Action* that reads the file produces the *PropertyMap*. So here's what happens: at the start the *Engine* is passed an empty *PropertyMap*, and the *Set* passed to the *SourceParser* is a *view* on this map. When the style sheet is loaded, the *PropertyMap* is mutated by the addition of new

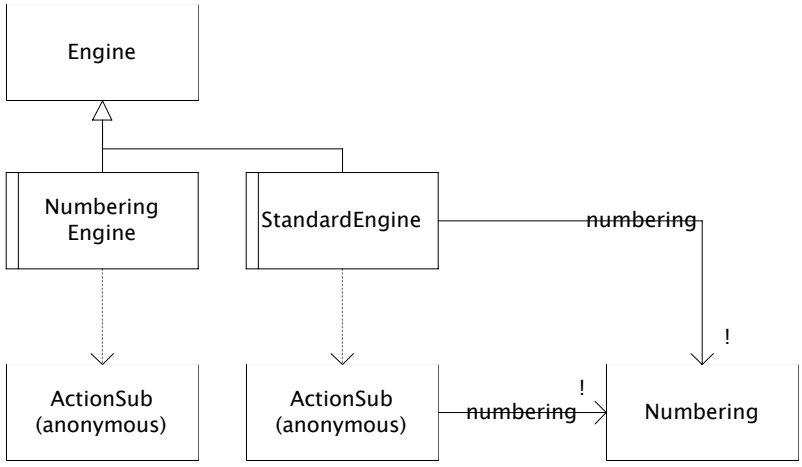


styles, and the view changes in concert with it. This mechanism is a bit tricky, but it does allow us to decouple *SourceParser* from *PropertyMap*.

The object model shows how the view connects the *SourceParser* and the *StandardEngine* in the code.

18.6.1 Multiple Engines

One might expect the *Engine* class to be a singleton, but it isn't. There is one *Engine* object for handling the source text, and a second *Engine*, with fewer actions, for processing numbering strings. In giving numbering directives in the style file, you can use the markup language. For example, the centered dot at the start of the bulleted paragraphs above are generated because the paragraphs were labelled with style *point*, and the numbering directive in the style file says that even though *point* isn't numbered, the

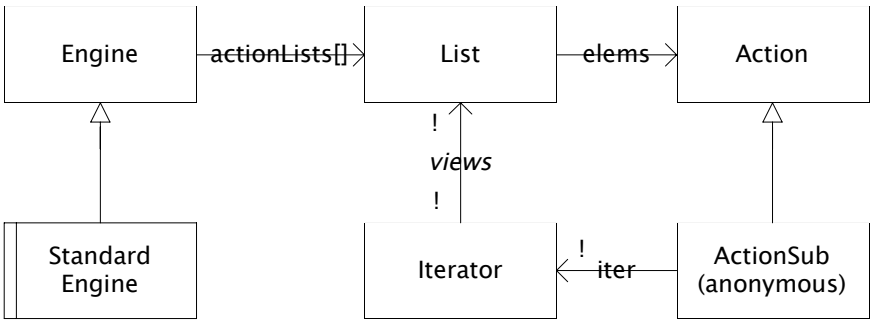


numbering string should still include a leading centered dot and a tab. The centered dot is referred to by the symbolic name \periodcentered. The *Numbering* object generates a string containing this symbolic name as a substring, which must be parsed and acted upon just like the tokens in the source text itself.

The object model shows that some actions have a reference to a *Numbering* object. The resulting string, not shown, is passed to a different *NumberingEngine* with its own actions. The dotted arrow depicts, informally, the relationship between an engine and its registered actions.

18.6.2 Comodification

The implementation of *Engine* uses an iterator to traverse the list of *Action* objects associated with the *TokenType* at hand. The *perform* method of each of these *Action* objects is executed. This perform method, as mentioned above, can itself register and deregister *Action* objects. If it were to do this for the *TokenType* at hand, the list being iterated over would be modified, thus violating Java's ban on concurrently modifying a collection while its iterator is active. The only case in which it seems to be necessary to do this is for 'one shot' actions which deregister themselves as soon as they occur. To handle these, *Engine* passes the iterator as an argument to the *perform* method of *Action*, which then invokes the *remove* method of the iterator to remove the *Action* object from the underlying list. This is an unusual and rather obscure use of the *remove* method, since the site of the call to *remove* is far from the site of the loop.



The object model shows how an action belonging to an action list of an engine has access to that list indirectly via an iterator.

18.6.3 Dynamic Registration

An *Action* may cause the registration or deregistration of other *Action* objects. For example, when a dollar is encountered for the first time, an *Action* is executed that registers an italicization action against the *TokenType* for alphabetic character sequences. When the next dollar sign is processed, this action is deregistered. As a result, alphabetic characters are italicized when placed between dollar signs.

18.6.4 Anonymous Inner Classes

Most of the *Action* objects are implemented using anonymous inner classes. To understand these, it's important to notice that the methods of the inner classes have access to the variables in the enclosing scope. Assignments to these variables are not permitted by Java, since the enclosing environment isn't a proper closure. This is why the variables that represent state in the processing of a paragraph are encapsulated in an object of class *ParaSettings*. Note that there are very few such variables: this is one of the key benefits of the action-based organization.

18.6.5 Type-Safe Enumerations

Various enumerations, such as *Format* and *TokenType* are implemented in a type-safe manner according to the idiom described in Item 21 of Bloch's *Effective Java*. Unlike the common practice of representing enumerations with static variables bound to integers, this ensures type safety. Unlike the algebraic datatypes of languages such as ML,

however, it does not allow the compiler to check that a case statement handles every value of the enumeration.

18.7 Design Alternatives

Here are some different designs that I might have used but chose not to:

- Use a line-based scripting language like Perl, sed or awk. These are not well suited to applications like Tagger in which lines have no significance, and context (such as whether italic mode is on) must carried across line breaks. I also didn't have the patience to debug a complex Perl script, and preferred to use a typesafe language.
- A classic object-oriented design in which each token type is represented as its own subclass of a class *Token*, and the actions are methods of these subclasses, relying on dynamic dispatch to select actions for tokens. This approach is simple, but it creates a huge number of classes. Worse, it splits functionality across many classes; math mode, for example, would not be coded in one place, but in all the tokens involved. This problem is what motivated the *Visitor* pattern. This approach would not allow behaviours to be changed dynamically the way my design does.
- A design in which the choice of behaviour according to token type is determined by a big case statement, or by methods of a *Visitor* pattern. This would have created a mass of global variables, so that functions such as math mode would pollute the entire case statement. In the action-based design, in contrast, these functions are mostly encapsulated.
- A standard compiler organization using an intermediate abstract syntax tree rather than a token stream. This would be far more flexible, and allows much better error reporting, but it's much more work to implement.

18.8 Design Defects

Here are some of the known defects of Tagger:

- Because Tagger doesn't see the final layout onto pages, it can't handle page-based issues, such as footnotes and running headers. With a more expressive import language (like some of the third-party formats marketed for Quark), it may be possible to handle these. Insertion of graphics suffers from the same problem: currently, they must be inserted by hand in the layout program.
- Tagger does not provide facilities for laying out tables.
- Tagger doesn't offer the features of TeX for mathematical formulas: it can't do multiline formulas that involve summations and integrals, for example.
- I had originally hoped to include backends for LaTeX and HTML, but too much of the functionality is incorporated in the actions themselves for these to implement-

ed as simply as backends for other layout programs, such as InDesign and Pagemaker.

- Style files are currently not properly checked. Errors in the numbering relations (for example, indicating that a style is a parent of itself) can cause Tagger to malfunction or crash without appropriate warnings.
- Error reporting is not always helpful. Errors found while parsing property files, for example, do not report line numbers.
- Property file syntax is represented in two different places in the code: in the *dump* method of *PropertyMap*, and in the parsing methods of *PropertyParser*. This is an undesirable coupling.
- Character maps and style sheets can override each other. No warning is given when this happens. If two character maps define the same symbolic character name, the last definition is used. A style name can shadow a character name. For example, the style name "section" shadows a character of that name, and prevents the display of the section symbol.
- Progress reporting is an ugly hack. As numbering strings are generated, they are passed to a special engine for display purposes that strips out characters that would not appear on a console. This mechanism assumes that the displaying of paragraph numbers is a reasonable way to show progress. Often it is, but it may not be when numbering is used for small items (eg, bibliographic references and lines of code).
- Although many of the actions can be understood independently, there are subtle interactins amongst some them. The behaviour associated with starting a new paragraph, for example, involves several actions, dynamic registrations and deregistrations, and state that persists across actions, encapsulated in the *ParaSettings* object. This reflects the context-sensitive nature of the problem: in this case that the beginning of a paragraph should cause a style directive to be generated by default, unless there is an explicit paragraph style command.
- Some characters (such as <) may not be used in the source text because they are interpreted as control characters by Quark. Tagger does not recognize these and wrap them appropriately, so the user must refer to them by symbolic names (such as \less).
- Character styles are not currently supported, but will be added soon.
- Quote mark disambiguation does not handle all cases correctly.

18.9 Development Process

Tagger was first written as a Perl script, to experiment with the idea of generating input for Adobe Indesign. I had been attempting to write text in Indesign's input format, but found it burdensome, especially since it distinguishes between linefeeds and carriage

returns, and therefore cannot be prepared in a text editor. This experiment was successful, and led me to become more ambitious, and add features such as automatic numbering. The Perl script was brittle and hard to maintain, so I decided to write a Java version instead.

I spent a few days designing the source language. This language was refined as I developed the implementation, and discovered what was easy to parse, and what was easy to write. I started by implementing the source parser, since I hate writing parsers, and wanted to get it done. The initial design was represented as a series of object models and module dependency diagrams.

I did not perform any unit testing, because most of the complexity was in classes (such as *StandardEngine*) that cannot be easily tested in isolation. I probably should have written unit tests for small data types such as *Counter*. Testing of the program as a whole was done manually, by eyeballing the output, and seeing how Quark processed it.

The program was improved incrementally over a few months as I used it in during my writing. So far, I have found 4 bugs in the code (3kloc, including comments) so far. I suspect there are many more bugs that do not show up even in extended usage of the tool, because pathological cases arise so rarely. Given that this program is for personal use, I'm content to let my daily use be its testing; of course, were this to be distributed, proper testing would be necessary. I have made about 20 fixes to the code in response to improvements in the source language.

In preparing the program for wider distribution, I added specifications for public methods. Since I was the only programmer, I had relied previously on writing specs only for tricky procedures. I also refactored the code in places, for example, introducing the typesafe enum pattern. To reduce the risk of introducing bugs, I wrote a crude regression test framework (as the *runTest* method the main *Tagger* class), which compares the generated file to a file previously generated.

18.10 User Guide

A very rough and unfinished user guide follows.

18.10.1 Command-line Arguments

The Tagger application is invoked with one argument, the name of the file to be processed, and an optional second argument, giving a pathname from which files referred to in the source file should be interpreted. The name of the source file is given

without an extension; the extension is assumed to be .txt. The generated file is given the suffix .tag.txt. By default, Tagger tries to open files mentioned in the source file at their specified location, and only if that fails does it prepend the optional pathname.

18.10.2 Overall Structure

Before a character symbol is used, a file defining the symbol must be loaded with the \loadchars command. Before a style name is used, a style sheet defining it must be loaded with the \loadstyles command. It is convenient to load character maps and style sheets in a preamble at the top of the file.

18.10.3 Lexical

The source text is parsed into paragraphs. The first printing text of the file starts a paragraph. Paragraphs are separated by a blank line (that is, either empty or containing whitespace – tabs or spaces) or by the special symbol \p, used for short paragraphs (such as lines of code to be numbered) that the user does not want to separate with blank lines.

Commands are prefixed with a backslash. Two forward slashes indicate a manual linebreak.

Commands are classified into *printing* commands, such as \alpha, which causes α to be generated, and *non-printing* commands, such as a paragraph style command like \section, which cause formatting changes but do not generate output that appears as text in the final document.

Whitespace is generally preserved, except the whitespace that follows non-printing commands. This allows the user to mark a paragraph with a style name on a previous line, with the linebreak being consumed. Tabs produce whatever indentation is specified in the style sheet of the layout program.

Since a command cannot be followed immediately by text, but requires whitespace following it, there is a special command \eat that consumes its following whitespace.

Hyphens and dots are translated to dashes and ellipses respectively. For example, a single hyphen is treated as a hyphen, two as an en-dash, and three as an em-dash. Quote marks are disambiguated according to context.

To insert a character that has special meaning as a simpel character, precede it with a backslash. for example, the string \eat is generated by typing \\eat.

18.10.4 Commands

- Paragraph style. If *style* is the name of a style defined in a style sheet previously loaded, the command *style* following a paragraph break or at the start of the file indicates that the paragraph it begins is to be set in the style *style*. Appending an asterisk at the end (*style**) causes numbering to be suppressed: no numbering string is generated, and counters are not incremented. The default paragraph style *body* is used for paragraphs that are not marked with an explicit style, and which do not acquire a style by virtue of following a style with a next style defined in the style sheet.
- Character symbol. If *char* is the name of a character defined in a character map previously loaded, the command *char* causes that character to be inserted.
- Italic mode. Text between underscores is italicized.
- Math mode. text between dollar symbols is put in math mode: all alphabetic and numeric strings are italicized, but other characters (such as punctuation and mathematical symbols) are unaltered.
- New command. The commands *new{column}* and *new{line}* start a new column and line respectively. *new{line}* is equivalent to //.
- Format commands. The string *format*<*text*> puts the text in *text* in the format specified by the format command *format*. Allowable format commands are *sub* and *super* for sub- and super-scripts, *bold*, *roman* and *italic*.
- Cross references. The command *tag{t}* tags a paragraph with the name *t* that can be used to refer to the paragraph. The command *label{l}* associates the label string *l* with the paragraph. The command *cite{t}* generates a cross-reference to the paragraph with the tag *t*. If the paragraph was labelled explicitly by a *label* command, the label is used as the cross-reference, otherwise the numbering string generated for the paragraph is used.

18.10.5 Style Sheet Format

A style sheet consists of a sequence of lines, each specifying the properties of a style. The first property names the style itself. Subsequent properties may include any of the following:

- *next*. Indicates which style follows this paragraph style by default.
- *counter*. If present, indicates that paragraphs of the style are to be autonumbered. The property value indicates both the initial value of the counter, and the style of counting: 0, 1, 2, etc for arabic counting; a, b, or A, B, etc for alphabetic counting. For example <*counter*:*B*> says that the counter should b upper-case letters, starting B, C, ...

- *trailer*. Source text to be inserted after the numbering string and before the paragraph text. May include commands such as special characters, new column, etc.
- *leader*. Source text to be inserted before the numbering string and before the paragraph text. May include commands such as special characters, new column, etc.
- *separator*. Source text to be inserted following the counter of this style, in numbering strings of paragraphs of the child style. This is used to insert dots between counters, for example.
- *parent*. For autonumbering, styles must be organized into a hierarchy. Styles are grouped into numbering series; each series has a root style, and a chain of child styles. The numbering series is indicated solely by giving a parent for each numbered style, except for a root, which has no parent. For example, to number chapters, sections and subsections in the standard way, one would make *chapter* the parent of *section* (by giving it this parent property in its property list), and *section* the parent of *subsection*.

Here is an example of a complete style file for that numbers sections 1, 2, etc; numbers subsections 1.1, 1.2, etc; separates numbers from their paragraphs by a tab; and places a centered dot before each paragraph of style point:

<style:section><next:noindent><counter:1><separator:.><trailer: >
<style:subsection><next:noindent><parent:section><counter:1><separator:.><trailer: >
<style:point><next:body><leader:\periodcentered >

18.10.6 Character Map Format

Each line of a character map file has the form

<char:myname><font:myfont><index:myindex>

where the character symbol *myname* appears in *myfont* at index *myindex*. The font property may be omitted if the character appears in the standard font.

Lecture 19: Conceptual Models

The same object model notation that we've used to describe the structure of the heap in an executing program – what objects there are and how they are related by fields – can be used more abstractly, to describe the state space of a system or of the environment in which a system operates. I call these 'conceptual models'; in the course text, they're called 'data models'. You've already actually built some of these in Exercise 4, in the warmup examples, and when you used the object modelling notation to describe the structure of the Boston subway system.

The notation itself is very simple indeed, and models are easy to interpret once you loosen yourself from an implementation-oriented view, replacing Java objects by entities in the real world, fields by relations, and so on. After this lecture, you should have no trouble *reading* conceptual models.

Writing them, on the other hand, takes more practice. It involves making appropriate abstractions – just as you have to do when you design the interface of an abstract data type. Doing this well is hard, but the obstacle is nothing to do with object models in particular. It always difficult to get to the essence of a problem and articulate it succinctly.

Once you've overcome this obstacle, and constructed a conceptual model, you're half way to a solution of your problem. It's often been said that if you can say exactly what your problem is, then you've made progress towards solving it. In software development, you're more than half way there.

So don't expect to be able to build conceptual models without some practice. It's a lot of fun, though, and as you hone your modelling skills, you'll find that you become a better designer. As your conceptual structures gain clarity, the structures in your code will become simpler and cleaner too, and coding will be more productive.

In the lecture itself, I'll try and give some sense of how models are constructed incrementally. In these notes, the models are shown in their final form.

19.1 Atoms, Sets and Relations

The structures of our models will be built from sets, relations and atoms. An atom is a primitive entity that is

- *indivisible*: it can't be broken down into smaller parts;
- *immutable*: its properties don't change over time; and
- *uninterpreted*: it doesn't have any built-in properties, the way numbers do, for example.

Elementary particles aside, very few things in the real world are atomic. But that won't stop us from modelling them as atomic. In fact, our modelling approach has no built-in notion of composites at all. To model a part *x* that consists of parts *y* and *z*, we'll treat *x*, along with *y* and *z*, as atomic, and represent the containment by an explicit relation between them.

A set is just a collection of atoms, with no notion of repetition count or order. A relation is a structure that relates atoms. Mathematically, it's a set of pairs, each pair consisting of two atoms, in a specified order. You can think of a relation as a table with two columns, in which each entry is an atom. The order in which the columns appear is important, but the order of the rows is irrelevant. Each row must have an entry in every column.

It'll be convenient to define some operators on sets and relations. We'll use these in explaining our graphical models, but they can also be used to write more expressive constraints.

Given two sets, *s* and *t*, you can take their union *s+t*, their intersection *s&t*, or their difference *s-t*. We'll write *no s* so say that an expression *s* denotes an empty set, and *some s* to say that it denotes a non-empty set. To say that every member of *s* is also a member of *t*, we'll write *s in t*. We'll write *s = t* when every element of *s* is an element of *t* and vice versa.

Given a set *s* and a relation *r*, we'll write *s.r* for the *image* of *s* under the relation *r* – the set of elements that *r* maps the elements of *s* to. We can define it formally like this:

s.r = {y | some x: s | (x,y) in r}

Given a relation *r*, we'll write *~r* for the *transpose* of *r* – the mirror image relation, defined like this:

~r = {(y,x) | (x,y) in r}

Finally, we'll write *+r* for the *transitive closure* of *r*: it's the relation that associates *x* to *y*, if there's some finite sequence of atoms *z*1, *z*2, …, *zn* such that

(x,z1) in r
(z1,z2) in r
(z2,z3) in r

...
(zn,y) in r

and **r* for the *reflexive transitive closure* of *r*: it's just like the transitive closure, but in addition it relates each atom to itself. You can think of the transitive closure as taking the image under one, two, or three, etc applications of the relation; the reflexive transitive closure includes zero applications.

Let's look at some examples. Suppose we have a set *Person* of people who exist or existed at some time; sets *Man* and *Woman* of male and female persons; a relation *parents* that associates a person with his or her parents; and a relation *spouse* that associates a person with his or her spouse.

Can you interpret each of these statements? Which is true for the real world?

no (Man & Woman)
Man + Woman = Person
all p: Person | some p.spouse => p.spouse.spouse = p
all p: Person | some p.spouse
all p: Person | some p.parents
no p: Person | p.spouse = p
all p: Person | p.sisters = {q: Woman | p.parents = q.parents}
all p: Person | p.siblings = p.parents.~parents

So far, these reflect basic observations about the world and definitions of terms. Here are some that get us into more controversial territory:

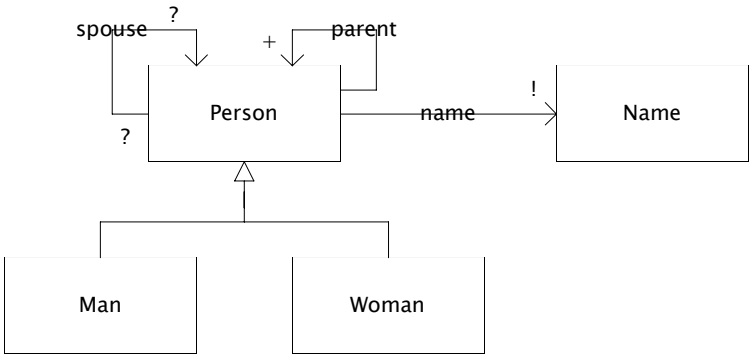
no p: Person | some (p.parents & p.spouse.parents)
Man.spouse in Woman
*some adam: Person | all p: Person | adam in p.*parents*
all p: Man | no q, r: p.spouse | q != r

I'm assuming you understand basic logical notation. I also slipped in a set comprehension in the definition of *sister*.

How would you write these statements in our notation?

Every person has a mother
Nobody has two mothers
A cousin is someone who shares a grandparent

The first statement illustrates something interesting and important. It's very easy to assume that the meaning of a term is obvious. In software development, it's very dan-



gerous! Ambiguity and vagueness in the meaning of terms causes endless problems. Developers understand requirements differently, and end up implementing modules that don't fit together, or don't solve the client's problem.

So we need to say carefully what each set and relation means. In this case, we have to say what *mother* means. Is it biological mother, or legal mother? Or maybe something else? When you construct a conceptual model that uses any terms that are not already define in the context in which you are working, you must provide a glossary. So here, we might write in our glossary:

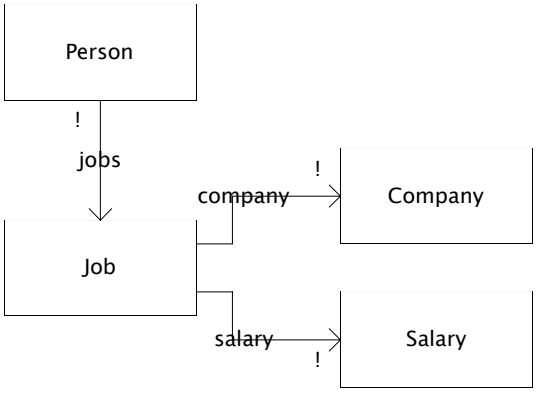
> *mother: (p,q) in mother means that person q is the biological mother of person p.*

19.2 Graphical Notation

There's no need to go through all the details of the graphical notation again; you've seen it before in the lecture on object models. All we need to do here is reinterpret the notation more abstractly.

Take a look at the object model for the family tree. Each box denotes a set of atoms – not a set of objects in a Java program, or a class! Each (open headed) arrow denotes a relation from one set to another. It denotes an abstract association, not a field or an instance variable.

The direction of the arrow has semantic consequence of course: it makes a big difference whether p is the parent of q or vice versa. But for any relation, we could equal well use a different relation that is the transpose; *children* instead of *parents* for example. There's no notion of navigability, or a relation belonging to a set in the way an instance variable belongs to a class.



The fat, closed-headed arrow denotes subset. Two sets that share an arrow are disjoint. We can fill in the arrow head to say that the subsets are also exhaustive: that every member of the superset is a member of at least one of the subsets. In this example, we've said that every *Person* is a *Man* or a *Woman*.

The sets that have no supersets are called *domains*. They're assumed to be disjoint. No atom is both a person and a name, for example.

We won't review the multiplicity and mutability markings here; they're explained nicely in the course text.

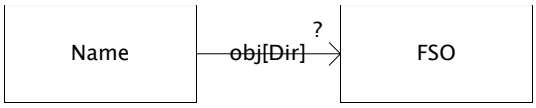
19.3 Ternary Relations

Sometimes we want to describe relationships that involve three, not two, sets. For example, we might want to record the fact that a person earns a salary working for a company. If persons can work for several companies, and earn a different salary at each, we can't just associate the salary with the person.

Often, the easiest way out of this is to create a new domain. Here, we could introduce *Job*, and draw an object model showing a relation *jobs* from *Person* to *Job*, a relation *salary* from *Job* to *Salary*, and a relation *company* from *Job* to *Company*.

This approach works well when the domain you introduce already corresponds to some natural set of atoms – it's a notion already understood in the problem domain.

Alternatively, you can introduce an *indexed relation*. If you mark the arrow from *A* to *B* with the label *r[Index]*, this means that for each atom *i* in the set *Index*, there is a relation *r[i]* from *A* to *B*. For example, to model naming in a file system, we might have an



indexed relation *obj[Dir]* from *Name* to *FileSystemObject*, since there is conceptually a separate naming relation for each directory of the file system.

Finally, you can draw the object model and say that it's a projection: that it shows the relationships for a particular atom in some domain. For example, in designing a word processor, there might be a ternary relationship *format* that associates a *StyleName* with a *Format* in a given *Stylesheet*. We may want to draw a model that considers only a single stylesheet, so that the relation becomes binary.

19.4 Three Examples

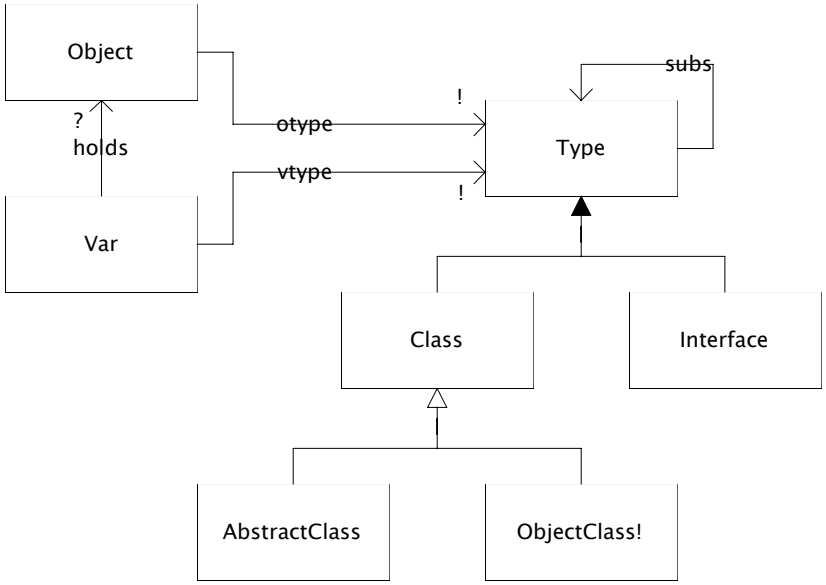
Let's look at three examples of conceptual models. They are all very simple, but they are not trivial. They demonstrate, I hope, that constructing even very small models is useful. As you work on your final project, and in any subsequent developments you do, you should construct conceptual models as you need them. Don't feel a need to have a single, all inclusive model; sketch a variety of smaller ones, and then consider which of them may need to be integrated. Until you have some experience with conceptual modelling, you'll probably thin something conceptual notion is obvious, and won't discover that it isn't until your deep into the code. So try and play around with more models that yo think you need at first. And if you strike complexity while you're coding, back off, and sketch some models.

19.4.1 Java Types

Our first model shows the relationships between objects and variables and their types in Java. Understanding this model is crucial to understanding dynamic dispatch and type casts.

There are three domains:

- *Object*: the set of instance objects that exist in the heap at runtime.
- *Var*: the set of variables that hold objects as their value. These include instance variables, method arguments, static variables and local variables.
- *Type*: the set of object types defined by classes and interfaces.



We'll ignore null references and primitive types such as *int*.

The domain *Type* is classified into classes, abstract classes, and interfaces. The set *ObjectClass* is a singleton – it's only member is the class called *Object*.

There are four relations:

- *holds*: maps a variable to the object it holds a reference to;
- *otype*: maps an object to its type – the type it acquired by virtue of being constructed by the constructor of some class;
- *vtype*: maps a variable to its declared type;
- *subs*: maps a type to its immediate subtypes. The subtypes of a class are the classes that extend it; the subtypes of an interface are the interfaces that extend it and the classes that implement it.

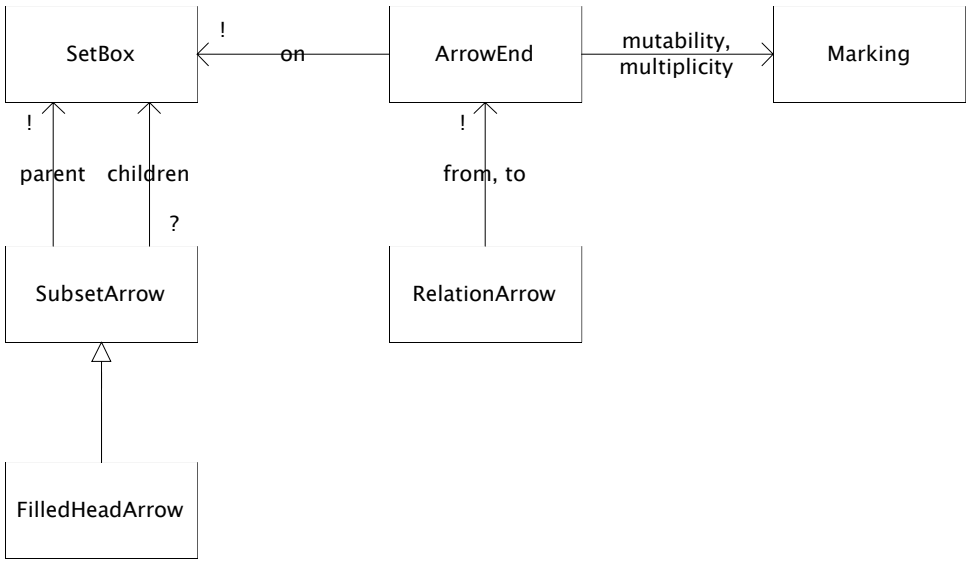
Here are some constraints that can't be expressed graphically:

- First, the essential type safety property – that the type of an object held in a variable is in the set of direct or indirect subtypes of the variable's type:

  *all v: Var | v.holds.otype in v.vtype.*sub*

- Some properties of the type hierarchy in Java: that every type is a direct or indirect subtype of the class *Object*; that a class can be the subtype of at most one other class; and that no type can directly or indirectly subtype itself;

  *Type in ObjectClass.*sub*



all c: Class | no c1, c2: Class | c1 != c2 && c in c1.sub && c in c2.sub
no t: Type | t in t.+sub

- That interfaces and abstract classes can't be instantiated:

  no o: Object | o.otype in (AbstractClass + Interface)

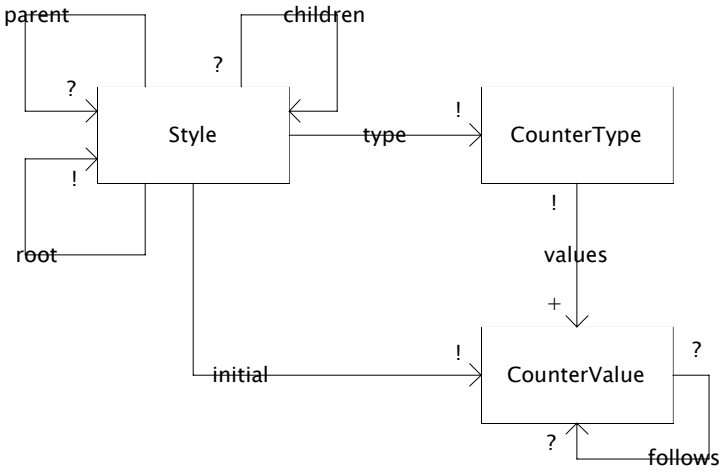
19.4.2 Meta Model

Our next model is a meta model of the graphical object modelling notation itself. It should be self-explanatory. A constraint:

- A set box cannot have a subset arrow to itself:

  no a: SubsetArrow | a.parent in a.children

There are very few constraints aside from those that require the subset hierarchy to be a tree; this is what makes the notation flexible. Often constraints are useful to *define* new sets and relations. For example, suppose we want to classify those relation arcs that represent *homogeneous* relations: relations that relate objects in a single domain. Can you define this notion as a new set? (Hint: it's probably easiest to start by defining a relation *super* from *SetBox* to *SetBox*; then a set *DomainBox*; and then defining the set *HomoArrow* in terms of these.)



19.4.3 Numbering

Our third model describes part of the Tagger application that was the subject of yesterday's lecture. It shows what information is stored in the stylesheet for numbering paragraphs, but does not show how the assignment of numbering strings to paragraphs themselves. That can be added too, but it's a bit trickier.

The domains are:

- *Style*: the set of paragraph style names;
- *CounterType*: the set of types of counters (eg, arabic, alphabetic, roman)
- *CounterValue*: the set of values a counter can take on (such as 1, 2, 3, or a, b, c)

The relations are:

- *type*: associates a style name with its declared counter type;
- *initial*: associates a style name with its declared initial counter value;
- *values*: associates a style name with its declared initial counter value;
- *follows*: associates a counter value with the value that follows it;
- *parent*: associates a style name with its parent style – *section*, for example, might be the parent of *subsection*.

Note that a style cannot have more than one parent. We do allow two styles to share a parent; this allows independent numbering, of, for example, figures and subsections within a section.

Here are some constraints:

- The initial value of a style's counter must be in the set given by its countertype. In tagger, this is enforced by the syntax: the declaration (eg, <counter:a> determines

both at once).

all s: Style | s.initial in s.type.values

- A style cannot be its own parent.

  no s: Style | s = s.parent

And here are some definitions:

- A style's children are those styles for which it is a parent:

  all s: Style | s.children = s.~parent

- The root of a style is the ancestor that has no parent.

  *all s: Style | s.root = {r: s.*parent | no r.parent}*

19.5 Conclusion

The graphical notation is described in more detail in the course text. You may also find helpful last year's 6170 lecture notes; there's a PDF file with a contents list as bookmarks on the web at:

http://sdg.lcs.mit.edu/~dnj/publications.html#fall00-lectures

You'll find a lecture there on conceptual modelling which includes a more substantial case study than the small ones given here.

The textual notation is called Alloy and was designed in the Software Design Group here at MIT. We've built an automatic analyzer for Alloy that can do simulations and checking. If you'd like to know more about it, look on the same publications page, and you'll find papers that describe the language and illustrate it with case studies.

Lecture 20: 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.

20.1 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.

20.2 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.

20.3Design 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.

20.4 Overview of Strategy

How are these desirable properties obtained?

20.4.1 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.

20.4.2 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.

20.4.3Efficiency

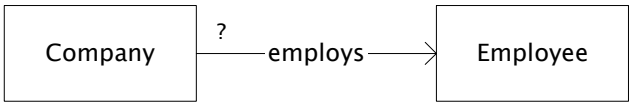
- *Object Model.* Your choice of code object model is crucial, because it's hard to change. So you should consider critical performance targets early on in the design. We'll look later in the lecture at some sample transformations that you can apply to the object model to improve efficiency.
- *Avoid 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.

20.5 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.

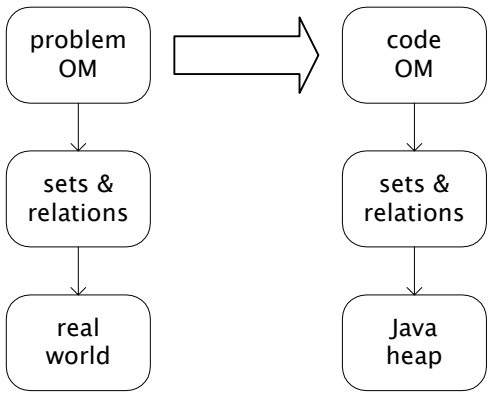
For example, suppose we have an object model with a relation *employs* from *Company* to *Employee*.



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.



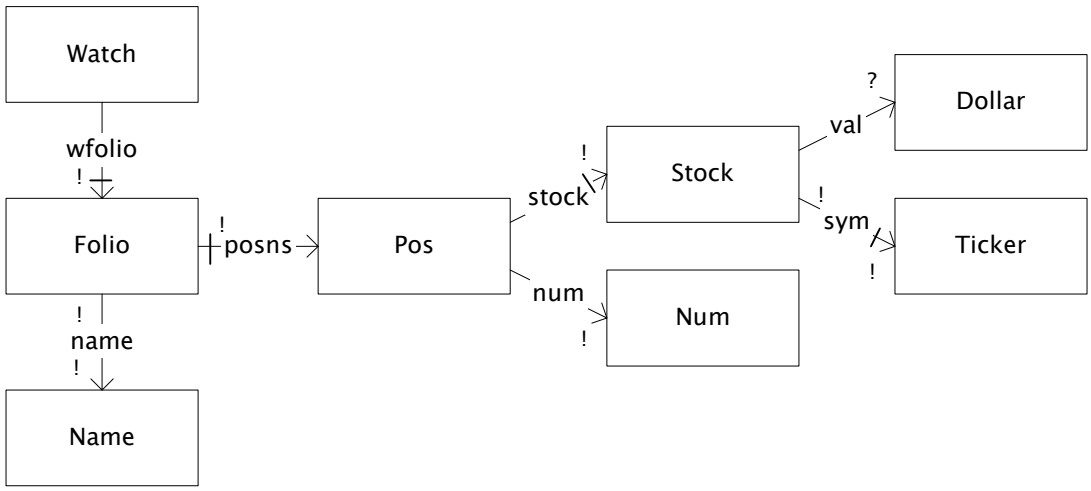
How is this transformation accomplished? One way is to brainstorm and play with different code model fragments until they coalesce. This can be a reasonable way to work. You need to check that the code object model is faithful to the problem object model. It must be capable of representing at

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.

20.6 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.

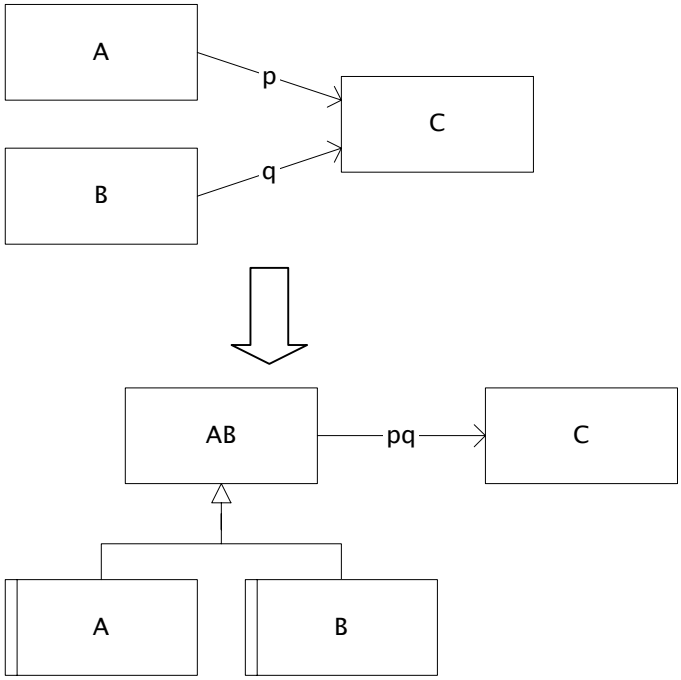


20.7 Catalog of Transformations

20.7.1 Introducing a Generalization

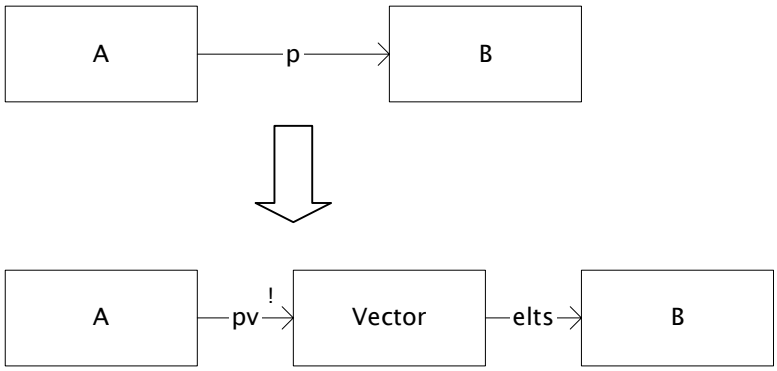
If *A* and *B* are sets with relations *p* and *q*, of the same multiplicity and mutability, to set *C*, we can introduce a generalization *AB* and replace *p* and *q* by a single relation *pq* from *AB* to *C*. The rela-

tion *pq* may not have the same source multiplicity as *p* and *q*.

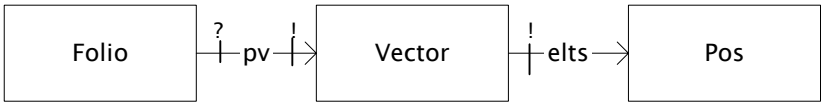


20.7.2Inserting a Collection

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.

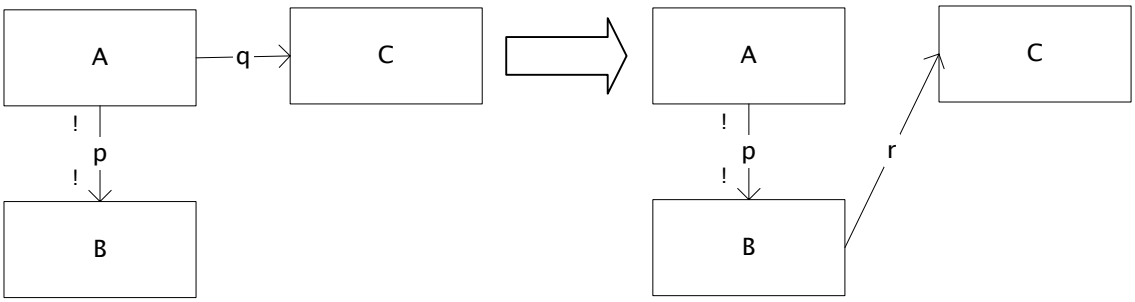


20.7.3Reversing a relation

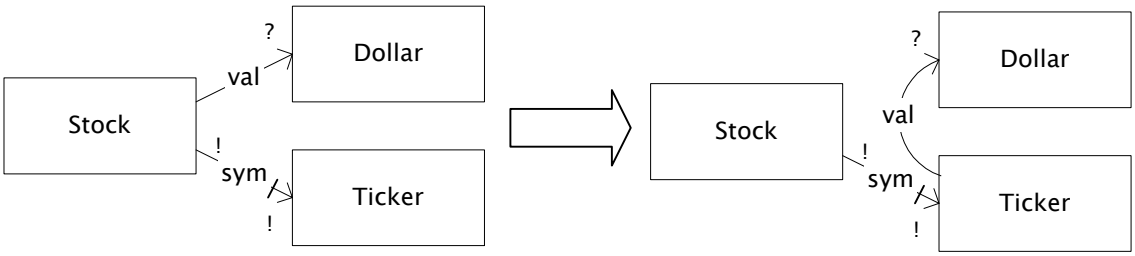
Since the direction of a relation doesn't imply the ability to navigate it in that direction, it is always permissible to reverse it. Eventually, of course, we will interpret relations as fields, so it is common to reverse relations so that they are oriented in the direction of expected navigation. In 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.

20.7.4Moving 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.



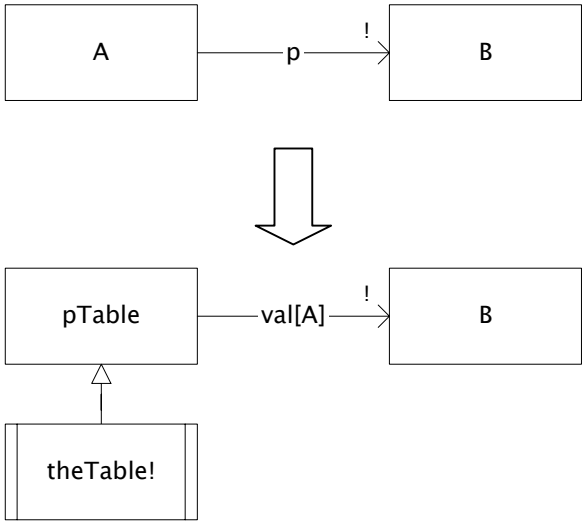
In our example, we can replace the *val* relation between *Stock* and *Dollar* by a relation between *Ticker* and *Dollar*. It's convenient to use the same name for the new relation, although technically it will be a different relation.



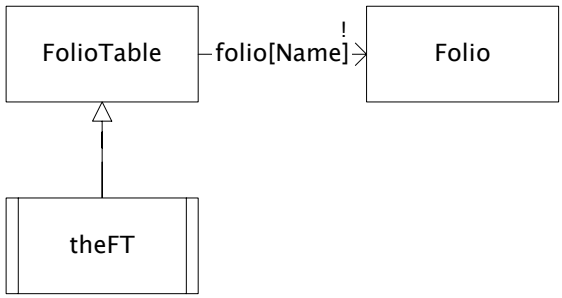
20.7.5Relation to Table

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

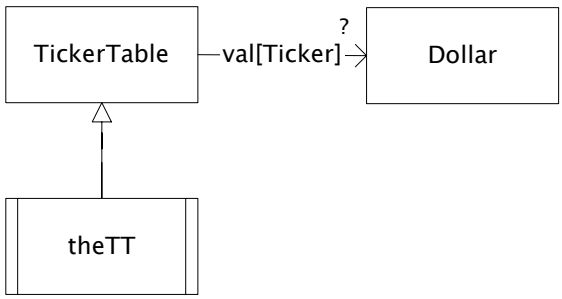
to support mappings to null values.



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.



20.7.6 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*.

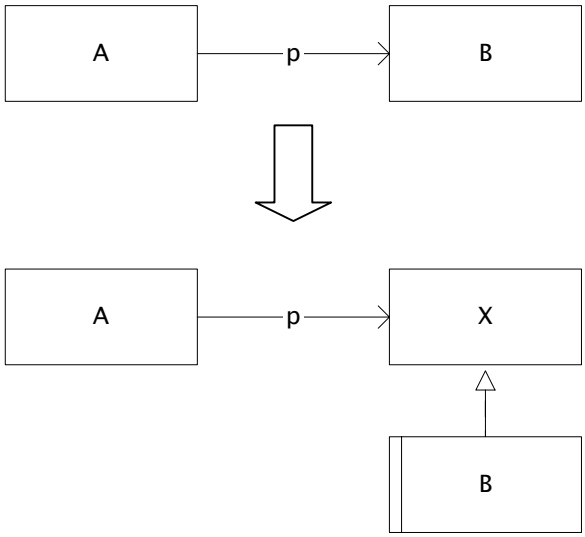
20.7.7Factoring out Mutable Relations

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.

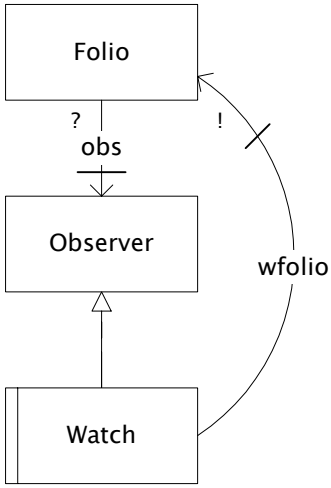
20.7.8Interpolating an interface

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*.



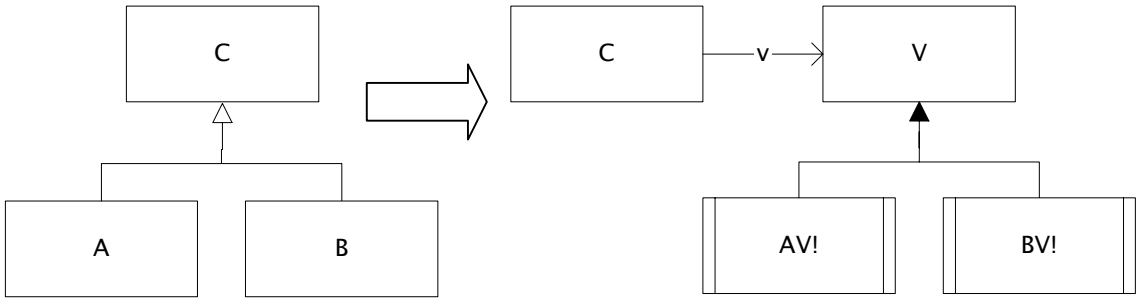
The Observer design pattern is an example of the result of this transformation. In our example, we

might make the *Watch* objects observers of the *Folio* objects:



20.7.9Eliminating 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



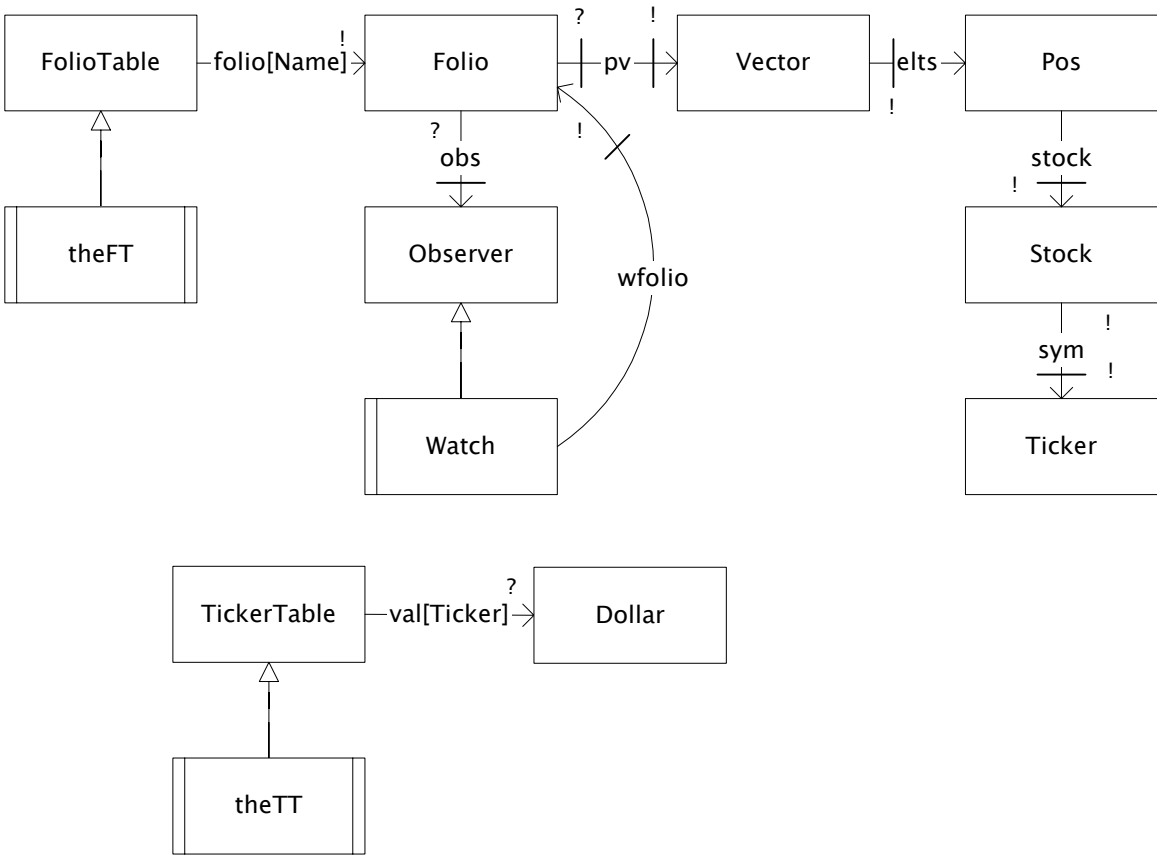
Where there is only one or two dynamic subsets, the classifier values can be primitive boolean values.

The classification can also be transformed to several singleton sets, one for each subset.

20.8Final 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.



20.9UML 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).