Contents

4 Testing **4**
4.1 Context . . . 4
4.2 Quote of the Day . . . 4
4.3 Today's Problem . . . 4
4.4 Choosing Test Cases . . . 5
4.5 Black Box Testing . . . 6
4.6 Boundary values . . . 7
4.7 Glass Box Tests . . . 7
4.8 Duplicates . . . 9
4.9 Is Testing Useful? . . . 9
4.10 The big picture . . . 10

10 Representation Independence **12**
10.1 Context . . . 12
10.2 User-Defined Types . . . 12
10.3 Immutable Types . . . 13
10.4 Classifying Operations on Types . . . 14
10.5 Designing an Abstract Type . . . 15
10.6 Choice of Representations . . . 16
10.7 Language mechanisms . . . 19

11 Representation Invariants **21**
11.1 Context . . . 21
11.2 A Tale of Two Spaces: Rep and Abstract . . . 21
11.3 Rep Invariants and Abstraction Functions . . . 24
11.4 Another example . . . 25
11.5 Rep Invariants for Modular Reasoning . . . 25
11.6 Rep Exposure . . . 26
11.7 Summary . . . 28

12 Abstraction Functions **29**
12.1 Context . . . 29
12.2 Judging correctness . . . 29
12.3 A Failure to Represent . . . 30
12.4 A Nifty Abstraction Function . . . 31
12.5 Specification Fields . . . 33
12.6 Common Confusions . . . 34

13 Identity and Equality I **35**

13.1 Quote of the Day . . . 35
13.2 Context . . . 35
13.3 The Object Contract . . . 35
13.4 Equality and Inheritance . . . 36
13.5 Equality and Efficiency . . . 39

14 Identity and Equality II **41**
14.1 Context . . . 41
14.2 Equality and Time . . . 41
14.3 Equality and Java Collections . . . 42
14.4 Problems with Equality and Mutation . . . 43
14.5 Problems with Hashing and Mutation . . . 44
14.6 Problems with Self-Containment . . . 45
14.7 What to do? . . . 45

15 Subtyping **47**
15.1 Context . . . 47
15.2 The Lure of Subclassing . . . 47
15.3 Java Subtypes . . . 48
15.4 Examples: Hashtable and Properties . . . 49
15.5 True Subtypes and the Substitution Principle . . . 51
15.5.1 Example: Square and Rectangle . . . 52
15.6 Dangers of Inheritance . . . 54
15.7 Summary . . . 56

16 Dependencies I **57**
16.1 Context . . . 57
16.2 Quote of the Day . . . 57
16.3 Decomposition . . . 57
16.4 Parnas's Uses Relation . . . 58
16.5 Liskov's Module Dependency Diagram . . . 59
16.6 The 3-Element Model . . . 59
16.7 Example: Phone Book . . . 60
16.8 Grouping . . . 62
16.9 Assumption Annotations . . . 63
16.10Name Dependences . . . 64
16.11Summary . . . 64

17 Dependencies II **65**
17.1 Context . . . 65
17.2 Example: Timer Callbacks . . . 65
17.3 Example: the Observer Pattern . . . 68
17.4 Families and Patterns . . . 69
17.5 Coupling Due to Shared Constraints . . . 70

18 Design Patterns I **71**
18.1 Context . . . 71
18.2 Design patterns . . . 71

18.3 Familiar patterns . . . 71
18.4 Categories of Design Pattern . . . 72
18.5 Factory Methods . . . 73
18.6 Factory Methods Allow Reuse . . . 74
18.7 Singleton . . . 75
18.8 Interning . . . 76
18.9 Abstract Factory . . . 77
18.10 Flyweight . . . 78
18.11 Prototype pattern . . . 80

19 Design Patterns II **82**
19.1 Context . . . 82
19.2 The Wrapper Pattern . . . 82
19.3 Adapters . . . 82
19.4 Decorator . . . 85
19.5 Proxy . . . 86

20 Design Patterns III **88**
20.1 Context . . . 88
20.2 The Observer Pattern . . . 88
20.3 Traversing hierarchical structures . . . 91
20.3.1 Interpreter . . . 92
20.3.2 Procedural Traversal . . . 93
20.3.3 Visitor . . . 94
20.4 Model/View/Controller . . . 98
20.5 When (Not) to Use Design Patterns . . . 98

Lecture 4: Testing

4.1 Context

What you'll learn: how to accumulate evidence that a module functions correctly and fulfils its intended purpose; black-box and glass-box testing; boundary cases and duplicates.

Why you should learn this: testing allows us to build confidence that a module actually meets its specification; skills in testing will help you avoid having to exercise your skills in debugging, saving you time and hair.

What I assume you already know: basic notions of Java (objects, classes, methods), how to read and write method specifications.

4.2 Quote of the Day

> *No amount of experimentation can ever prove me right; a single experiment can prove me wrong.*
>
> Albert Einstein

4.3 Today's Problem

You are about to use a method written by a co-worker, and you'd like to be sure it actually works before using it. The method is supposed to perform region grouping on binary images. For example, given the 7×9 matrix on the left below, it should output a matrix like the one shown on the right, where the cells in each connected region of 1s are replaced with an ID number that is unique to that region.

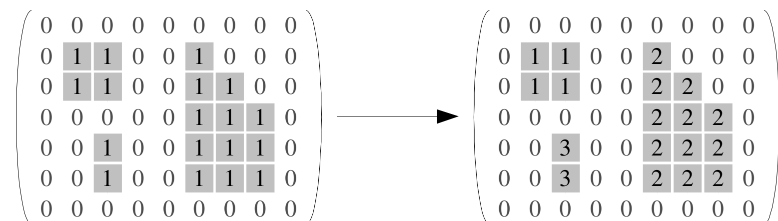


Figure 1

There are some ambiguities – for example, what exactly counts as a connected region? You and your co-worker have had misunderstandings in the past, so this time you were careful to come up with an agreed specification:

```
void labelImage(Matrix src, Matrix dest)
```

Take a matrix representing a black-and-white image, and produce another matrix where each cell of a connected region of non-zero values is replaced with an arbitrary ID unique to that region.

requires: *src* is not null; every cell in *src* contains either 0 or 1

modifies: *dest*

effects: *dest* is a matrix with the same dimensions as *src*, and:

$$\begin{array}{lcl} src(r,c)=0 & \Leftrightarrow & dest(r,c)=0 \\ src(r,c)=1 & \Leftrightarrow & dest(r,c)\neq 0 \\ & & dest(r,c)=dest(r',c') \text{ iff } \exists \text{ some sequence } (r_1,c_1),(r_2,c_2),\ldots,(r_n,c_n) \\ & & \text{with } (r_1,c_1)=(r,c) \text{ and } (r_n,c_n)=(r',c') \text{ such that:} \end{array}$$

$$\begin{array}{rcl} src(r_i,c_i) & = & 1, \quad 1\leq i\leq n \\ |r_i-r_{i+1}|+|c_i-c_{i+1}| & = & 1, \quad 1\leq i<n \end{array}$$

This captures the idea that 1s connected by a path of other 1s should have the same ID, otherwise they should be different. The last condition defines "4-connectivity": only cells immediately above, below, or the the left or right of a cell are considered connected to it (and not the diagonals).

We'd like to confirm that the co-worker's implementation actually meets this specification. One obvious approach is to just try it out and see if it works.

4.4 Choosing Test Cases

Exhaustive testing is clearly infeasible — you can't pass all possible matrices to the method and check that it returns the right result for each. Haphazard or random testing, on the other hand, is less likely to discover bugs. We want to pick a set of test cases that is small enough to run quickly, yet large enough to validate the module. To do this, we divide the input space into subdomains, each consisting of a set of inputs. The subdomains completely cover the input space, so that every input lies in at least one subdomain. Then we choose one test case from each subdomain. For example, to test `int abs(int x)`, we might divide the input space into $x < 0$ and $x \geq 0$, and choose test-cases of $x = 5$ and $x = -5$.

The idea behind subdomains is to make the best use of limited testing resources by choosing dissimilar test cases, and test parts of the input space that random testing might not reach. Ideally, the subdomains we choose should be *revealing*, which means that the module fails or succeeds on all inputs in the subdomain – it doesn't succeed on some and fail on others. If all the subdomains are revealing, then our test suite is guaranteed not to overlook any failures. For example, we are guaranteed that our test-cases of $x = 5$ and $x = -5$ are sufficient to reveal any failure in `int abs(int x)` if the implementation meets the following conditions:

- ▷ If the method fails for any $x < 0$ then it will fail for all $x < 0$
- ▷ If the method fails for any $x \geq 0$ then it will fail for all $x \geq 0$

In practice, of course, we don't have guarantees of this sort, so we fall back on heuristics to guess subdomains that are likely to be revealing. Two common heuristics are *black box* testing and *glass box* testing.

4.5 Black Box Testing

Black box tests are generated by examining only the specification of a module. Black-box tests avoid the pitfall of examining the implementation and repeating its errors or assumptions; they are essentially an independent verification of functionality. Such tests are representation-independent and can be reused even if a new implementation is substituted for the old one.

To generate black-box test cases, we examine the specification to find subdomains of the input space that could produce different behavior. For example, consider the specification for `sqrt`:

```
public double sqrt (double x)
```

throws: `IllegalArgumentException` if $x < 0$
returns: approximation to square root of x

Two subdomains immediately leap out: $x < 0$, since an exception is thrown, $x \geq 0$, since the method returns normally. We should also create a subdomain at the *boundary* of these two subdomains, since bugs often appear at discontinuities, so we would also add a test case for $x = 0$. Other subdomains for `sqrt` are more subtle:

- ▷ we might test perfect squares (where `sqrt` returns an integer value) separately from other numbers (where `sqrt` returns a non-integer);
- ▷ we might test cases where $x < sqrt(x)$ separately from cases where $x > sqrt(x)$, along with the boundary case $x = sqrt(x)$. The former subdomain is $0 < x < 1$, the latter subdomain is $x > 1$, and the boundary cases are $x = 0$ and $x = 1$.

Combining all these subdomains could give us the following black-box test cases:

- ▷ -1 (for the subdomain $x < 0$)
- ▷ 0 (a boundary case)
- ▷ 0.5 (for both $x \leq sqrt(x)$ and $sqrt(x)$ non-integer)
- ▷ 1 (another boundary case)
- ▷ 4 (for $sqrt(x)$ integer, and $x > sqrt(x)$)

Now let us try to apply this idea to the `labelImage` method. Here are some possible subdomains:

- ▷ Matrices with zero, one, or several regions.
- ▷ Matrices with one or several rows/columns.
- ▷ Square or rectangular matrices.

For each region in the matrix, we can have further subdomains:

- ▷ Regions that form a point, a line, or an area.
- ▷ Regions with or without holes.
- ▷ Regions with or without concavities in their boundaries.
- ▷ Regions that overlap with the various edges of the matrix.

This doesn't even begin to exhaust the possibilities, particularly if we start to consider the possible relationships that could exist between regions (e.g. regions that meet at a diagonal, but are not technically connected). Some example tests are shown in Figure 2. Check that you understand which of the listed subdomains are being tested, and try to develop a more complete set of tests

yourself. Notice that as for the `sqrt` example, we can often cover two or more subdomains with the same test case – smaller test suites are faster to run, so they'll be run more often; they are also easier to maintain and more likely to be correct.

A $\begin{pmatrix} 0 \end{pmatrix}$

B $\begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$

C $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$

D $\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$

Figure 2

4.6 Boundary values

There is no guarantee that the subdomains we choose for testing are actually revealing – it is possible that the module may succeed for some inputs in a subdomain and fail for others. One very common problem is that bugs can shift the borders of "real" subdomains away from what would be expected from the specification, and we don't what to miss that. So, as we saw for testing `sqrt`, we should look particularly carefully at the boundaries between one subdomain and another, since this is a likely place for bugs to arise. In effect, we treat the boundaries of subdomains as extra subdomains in their own right, and draw test cases from them.

For `int abs(int x)`, we should test 0 and perhaps 1 and -1 as boundary cases. There are also the extreme values of x, which are `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. Testing `Integer.MIN_VALUE` would uncover a problem – no Java method can work exactly as we would hope, because `Integer.MIN_VALUE` = -2147483648 while `Integer.MAX_VALUE` = 2147483647, leaving us in an awkward position when we try to compute `abs(Integer.MIN_VALUE)`. If you check the documentation for `Math.abs`, this case is dealt with specifically (do you know how?).

For the region grouping example, there are many boundaries. Obvious ones are the edges of the matrix – we would want to test for regions lying on these edges. Another important boundary condition lies at the change from one region to two regions. Yet another boundary condition is very large matrices with many regions – this will require many ID numbers, and depending on how those ID numbers are assigned and the size of the integers that can be stored in the matrices, it is possible to imagine them running out (image processing applications often use small 8-bit integers).

4.7 Glass Box Tests

Black box testing is useful, but taking the implementation of a module into consideration can suggest further cases to which we may not have otherwise devoted time. Consider a method `isPrime`, which returns `true` if its integer argument is prime. Black box testing would suggest testing this method on a prime number, say 5, and a composite number, say 6. But suppose `isPrime` is implemented with a table lookup for small integers:

```
int isPrime (int x) {
  if (x < 100) look up in a table
  else compute answer
}
```

If we tested this `isPrime` only on 5 and 6, we would only test canned answers from the table, and never actually exercise the code that computes the answer.

In glass box testing, you examine the code of the module or method to discover additional subdomains. In this case, `isPrime` should also be tested with $x < 100$, $x > 100$, and (the boundary case) $x = 100$.

A standard approach to glass box testing is to add tests until the test suite achieves adequate statement coverage: so that every statement in the module is executed by at least one test case. In the `isPrime` example, we had inadequate statement coverage from the black box tests alone, since they never caused the "compute answer" code to be executed. Statement coverage is not the only kind of coverage, and in fact is not always sufficient to evaluate a test suite. Consider the following buggy method:

```
int min (int a, int b) {
  int r = a;
  if (a <= b)
    r = a;
  return r;
}
```

For this code, a single test case with $a < b$ would achieve 100% statement coverage, but it would fail to reveal the bug in the method, which reveals itself only when the if branch is not executed. There are still stronger forms of coverage: decision coverage, condition coverage, path-complete coverage, etc. We will not examine those further here.

Returning to our region grouping example, suppose the algorithm used by your co-worker is:

- ▷ Initialize the *dest* matrix to the same size as *src*, and fill it with 0s.
- ▷ Scan the *src* matrix from left-to-right, and then from top-to-bottom, looking for 1s.
- ▷ Whenever a 1 is found in $src(i, j)$, examine $dest(i-1, j)$ and $dest(i, j-1)$ (call these values $left$ and up respectively, and treat them as 0 if they lie out of the bounds of the matrix).
  - If $left$ and up are both 0, pick an unused integer and place it in $dest(i, j)$.
  - If one of $left$ or up is non-zero, place its value in $dest(i, j)$.
  - If both $left$ and up are non-zero, place the value of up in $dest(i, j)$. If $left \neq up$, replace all instances of $left$ in $dest$ with up.

Given this information about the method's implementation, can we refine our guesses about what subdomains will be revealing for it? The accesses to $left$ and up expose an important boundary condition when cells containing 1 lie on the left or topmost edge of the matrix. We have already considered such boundary conditions in black box testing. The algorithm takes different actions based on the relative values of $left$ and up. This is *not* something we have explicitly considered during black box testing. As far as the algorithm is concerned, these variables can be in one of five relationships:

- ▷ $left$ and up are both zero.
- ▷ $left$ is zero and up is non-zero.
- ▷ $left$ is non-zero and up is zero.
- ▷ $left$ and up are non-zero and equal.
- ▷ $left$ and up are non-zero and different.

These possibilities comprise a set of subdomains we should test, by picking test cases that trigger the corresponding conditions. The following test case should in fact trigger them all:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Figure 3

As it happens, the tests already described in black box testing also hit all these possibilities, so we don't need to add any new tests.

4.8 Duplicates

Another kind of condition that it is important to consider is sharing between the inputs. For the method:

```
void labelImage(Matrix src, Matrix dest)
```

It is possible that *src* and *dest* refer to the same object. And in fact, in image processing pipelines, it is very common for this to be the case. It is very easy to neglect this possibility; A straightforward implementation of `labelImage` would begin by zeroing the *dest* matrix, which in this case would also destroy the *src* matrix, and the method would fail.

Aliasing, or multiple references to the same object, is the most common kind of problem, but other kinds of duplication can also be problematic. For example, what happens if you try to add a list that contains a set into that set itself?

4.9 Is Testing Useful?

Perhaps this all seems a little pedantic and unnecessary. With a little practice, testing in fact becomes interesting and fun. Done early, testing catches bugs before they have a chance to hide themselves in complexity. Without testing, life becomes a nightmare of late-night debugging. For example, a humanoid robot at MIT had a bug in the region growing code it used – the algorithm was just as described above, but instead of:

If $left \neq up$, replace all instances of $left$ in *dest* with *up*.

it did:

If $left = up$, replace all instances of $left$ in *dest* with *up*.

This is clearly a bug – the replacement operation does nothing, since it is called if and only if it is not needed. This bug survived in the code for a long time because the module was never tested outside of the entire robot (which took on the order of 10 minutes to start up), and the overall behavior was sufficiently complicated that the low-level grouping bug went unnoticed. In fact the bug is a little subtle, since it only has an effect around certain diagonals (it doesn't affect the case shown in Figure 1, for example). However, all those unnecessary replacements made the method quite slow, and in the end this was what prompted a reexamination of the code. When the bug was fixed, other users of the robot complained – in fact the robot code had become tuned to regions being small.

A few simple tests would have caught this bug early, rather than letting it creep in as some nagging problems that were much harder to track down. It would also have caught the bug before others grew to depend on it; this is particularly important in library code, where silly quirks often have to be maintained because existing applications depend on them (although good specifications can reduce this problem).

4.10 The big picture

Testing is only one part of a more general process called validation. The purpose of validation is to uncover problems in a program and thereby increase your confidence in the program's correctness. Validation includes:

- Formal reasoning about a module, usually called verification. Verification constructs a formal proof that a module is correct, by showing that whenever the preconditions are satisfied, the module always produces a state in which the postconditions are true. Verification is tedious to do by hand, but automated tool support for verification is a very active area of research. Small, crucial pieces of a program may be formally verified, such as the scheduler in an operating system or the bytecode interpreter in a virtual machine.
- Informal reasoning. Having somebody else carefully read your code can be a good way to uncover bugs, much like having somebody else proofread an essay you have written. In industry, this practice goes by various names (with various degrees of formality): code reviews, code inspection, walkthroughs. Pair programming is an extreme form of this idea, where two programmers work together on a single computer, with one programmer typing and the other reading and thinking about the code being typed. You can also read your own code, although it is harder to catch your own mistakes.
- Testing — running the module on carefully selected inputs and checking the results.

Validation requires having the right attitude. Your goal is not to see the module work, but to make it fail. There's a subtle difference. It is all too tempting to treat code you've just written as a precious thing, a fragile eggshell, and test it very lightly just to see it work. You have to be brutal. A good tester wields a sledgehammer and beats the module everywhere it might be vulnerable, so that those vulnerabilities can be eliminated.

There are three basic principles of testing:

- Be systematic. Haphazard testing is less likely to find bugs (unless the module is so buggy that a randomly chosen input is more likely to fail than to succeed), and it doesn't increase our confidence in module correctness. On the other hand, exhaustive testing — running the module on all possible inputs — is usually impossible. Instead, test cases must be chosen carefully and systematically. Some approaches to choosing test cases are discussed below.
- Do it early and often. Don't leave testing until the end, when you have a big pile of unvalidated code. Leaving testing until the end only makes debugging longer and more painful, because bugs may be anywhere in your code. It's far more pleasant to test your code as you develop it. One technique, testdriven development, carries this idea to its logical conclusion: you write tests before you even write any code. In other words, the development of a single module might proceed in this order:

  1. write a specification for the module;
  2. write tests that exercise the specification;
  3. write the actual code.

- Automate it. Nothing makes tests easier to run, and more likely to be run, than complete automation. A test driver should not be an interactive program that prompts you for inputs

and prints out results for you to manually check. Instead, a test driver should invoke the module itself on fixed test cases and automatically check that the results are correct. The result of the test driver should be either "all tests OK" or "these tests failed: ..." A good testing framework, like JUnit, helps you build automated test suites. You can find links to more information about JUnit on the course web page.

Lecture 10: Representation Independence

All you need in this life is ignorance and confidence, and then success is sure.

Mark Twain

10.1 Context

What you'll learn: How to (safely) keep users of a type in blissful ignorance of how it is actually represented.

Why you should learn this: If we can shield users from the representation of a type, then we are free to make changes to that representation (for example, to support optimizations) without getting cursed at. Everyone wins. And reasoning about the type becomes much simpler.

What I assume you already know: How to read and write specifications.

10.2 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 data types (ADTs): 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.

The nature of the operations that can be performed on a type have a huge impact on how easy it is to use and to reason about. We have already seen this with immutable types, which we now define more carefully.

10.3 Immutable Types

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 `ArrayList` is mutable, because you can call `add` and observe the change with the `size` operation. But `String` is immutable, because its operations create new string objects rather than changing existing ones. There is no method you can call on a given `String` that reveals the effect of any earlier method calls on that `String`. 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. Often, when you design an abstract type, it won't be immediately obvious whether the type should be mutable or immutable. You should generally err on the side of making it immutable; novices make much too much use of mutable types (and pay the price in complexity and poor performance).

Immutable types are easy to work with because they never change, ruling out whole classes of bugs. But what does it take for a type to be immutable? If there is no way to change the type's state, then clearly it is immutable. But in fact this is a stronger condition than we need. In fact we only require that there is no way to *observe* a change in the type's state. For example, suppose we have the following class representing immutable 2D points:

```
class Point {
    private double x, y;
    public Point(double x, double y) { this.x=x; this.y=y; }
    public double x() { return x; }
    public double y() { return y; }
    public double r() { return Math.sqrt(x*x+y*y); }
    public double theta() { return Math.atan2(y,x); }
}
```

Points are internally represented in Cartesian form (x, y), but can be queried in either Cartesian or polar $(r, theta)$ form (*note*, I've omitted an important check for degenerate arguments to $atan2$ for the sake of brevity). If this type is used in an application where the polar form dominates, it will be quite inefficient, since the $sqrt$ and $atan2$ operations are repeated for every access. An obvious optimization is to cache those values, so they only have to be computed the first time they are requested, and from them can simply be looked up:

```
class CachingPoint {
    // same as Point, then add the following ...
    private double r, theta;
    private boolean polarized;
    private void polarize() {
        if (!polarized) {
            r = Math.sqrt(x*x+y*y);
            theta = Math.atan2(y,x);
            polarized = true;
        }
    }
    public double r() { polarize(); return r; }
    public double theta() { polarize(); return theta; }
}
```

This should be indistinguishable to a user from the previous version of `Point` – except that, in most situations, it will be faster. But notice that the methods `r()` and `theta()` now change the internal state of the `Point`. This doesn't matter since this change is not observable externally.

So if immutable objects never change, can we do anything useful with them? We began the lecture by promising to switch from talking about a type's representation to the operations that can be performed on it. But what useful operations can we apply to an immutable object? The idea here is to use *producers*, where the output of an operation, rather than mutating an object, is placed in a fresh new object. For example, we might give `Point` a method called `translate` as follows:

```
public Point translate(double x, double y) {
    return new Point(this.x+x,this.y+y);
}
```

This method returns a new appropriately translated `Point` rather that changing the object for which it is called.

10.4 Classifying Operations on Types

We can classify the operations of an abstract type as follows:

- *Creators* create new objects of the type. A creator may take an object as an argument, but not an object of the type being created – in that case, the operation is a producer.
- *Producers* create new objects from old objects. 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 `add` method of `ArrayList`, for example, mutates the list by adding an element to its end.
- *Observers* take objects of the abstract type and return objects of a different type. The `size` method of `ArrayList`, for example, returns an integer.

We can summarize these distinctions schematically as follows, with arguments on the left, and the returned object on the right. We treat the `this` reference as an extra argument, if it is available.

creator:	$\mathtt{t}^*$	$\rightarrow$	`T`
producer:	$\mathtt{T}^+$, $\mathtt{t}^*$	$\rightarrow$	`T`
mutator:	$\mathtt{T}^+$, $\mathtt{t}^*$	$\rightarrow$	`void`
observer:	$\mathtt{T}^+$, $\mathtt{t}^*$	$\rightarrow$	`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 (e.g., `size`), and some take several.

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.

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.

Here are the appropriate classifications for the methods of the `BasicList` class of PS2:

creator	BasicList() BasicList(Collection c)
producer	*none*
mutator	void add(int index, Object element) boolean add(Object o) Object remove(int index)
observer	boolean contains(Object elem) boolean equals(Object o) Object get(int index) int hashCode() int indexOf(Object elem) boolean isEmpty() int size() Object[] toArray() Object[] toArray(Object[] a)
iterator	Iterator iterator()

Notice that there are no producers. This is common for mutable classes. Producers are used more often with immutable classes, such as `String`.

10.5 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 until we catch an exception, 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.
- A good abstract data type should be *representation independent*. This means that the use of an abstract type is independent of its representation (the actual data structure or data fields used to implement it), so that changes in representation have no effect on code outside the abstract type itself. For example, the operations offered by `BasicList` are independent of whether the list is represented as a linked list or as an array.
- Although functionality is your first concern, efficiency is important too. You need to ensure that the operations allow a client of the type to work efficiently. This impacts the choice of operations in two ways: you have to include the right operations so the client doesn't have to do something convoluted, and you must design the operations so they can be implemented efficiently.
- In practice, you won't be able to change the representation of an abstract type at all unless its operations are fully specified with preconditions (requires), postconditions (effects), and frame conditions (modifies), so that clients know what to depend on, and you know what you can safely change.

10.6 Choice of Representations

A class that implements an abstract type provides a *representation*: the actual data structure that supports the operations specified for the type. 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 same abstract type, but this is rarely done in Java).

Suppose we wish to create an abstract data type for a matrix of integers, as used in a previous lecture. Here is a specification for what we want:

> *Matrix represents a mutable matrix of integers, with 0-based, row first indices.*
> **class** Matrix
> spec **int** height;
> spec **int** width;
> spec (**int** -> **int** -> lone **int**) cells;
>
> Matrix(**int** r, **int** c)

throws MatrixIndicesOutOfBoundsException **when** $r \leq 0$ or $c \leq 0$
returns **new** result such that
result.height=r and result.width=c and no result.cells

int get(**int** r, **int** c)
throws MatrixIndicesOutOfBoundsException **when** not ($0 \leq r < height$ and $0 \leq c < width$)
returns
if some v : c.(r.cells) then v else 0

void set(**int** r, **int** c, **int** v)
modifies this.cells
throws MatrixIndicesOutOfBoundsException **when** not ($0 \leq r < height$ and $0 \leq c < width$)
effects
cells = \old(cells - (r -> c -> c.(r.cells))) + (r -> c -> v)

int rows()
returns height

int cols()
returns width

We could write this specification more informally by just borrowing the familiar mathematical notion of a matrix, but our notation works fine too. We have specified that Matrix objects are mutable. This is appropriate since we often want to make small state changes within a large Matrix (e.g. changing the value of one cell), and this can be cumbersome with an immutable object.

We've got a lot of choices for how we implement Matrix. We could use a 2D array, which is perhaps the most direct mapping of a matrix onto Java primitives:

```
public class Matrix {
    private int[][] data;
    public Matrix(int r, int c) { data = new int[r][c]; }
    public int get(int r, int c) { return data[r][c]; }
    public void set(int r, int c, int v) { data[r][c] = v; }
    public int rows() { return data.length; }
    public int cols() { return data[0].length; }
}
```

For brevity, I don't deal with the exceptions required by the specification here. We could also make an implementation that is closer to the feel of the specification, using one of Java's `Map` implementations:

```
public class Matrix {
    private Map data;
    private int height, width;

    public Matrix(int r, int c) {
        data = new HashMap();
        height = r;
```

```
        width = c;
    }

    public int get(int r, int c) {
        Object o = data.get(new Pair(r,c));
        if (o==null) return 0;
        return ((Integer)o).intValue();
    }

    public void set(int r, int c, int v) {
        data.put(new Pair(r,c),new Integer(v));
    }

    public int rows() { return height; }
    public int cols() { return width; }
}
```

(We have assumed the existence of a simple class called Pair for storing and comparing pairs of integers.) Both these implementations meet the specification (excluding exception handling), but have completely different internal representations. For relatively small matrices, the first implementation provides faster cell access times. But the second implementation is more practical for large, sparse matrices, since it doesn't waste vast areas of memory representing unused portions of the matrix. By hiding the representation, we guarantee that the two implementations can be swapped without having to rewrite code that uses the `Matrix` class.

Another possible implementation of `Matrix` is as follows:

```
public class Matrix {
    private int[] data;
    private int width;
    public Matrix(int r, int c) {
        data = new int[r*c];
        width = c;
    }
    public int get(int r, int c) { return data[r*width+c]; }
    public void set(int r, int c, int v) { data[r*width+c] = v; }
    public int rows() { return data.length/width; }
    public int cols() { return width; }
}
```

Here we map the 2D matrix on to a 1D array. Why would we want to do this? In high-speed image processing, the actual layout of an image in memory can have a big impact on performance. With a 1D array, we can allocate some extra "elbow-room" for padding, to achieve good alignment of rows in memory. Such details are arcane and processor dependent, but can really matter for performance. It is very desirable to completely hide such complex, changeable details from client code, just like modern languages insulate us from knowing (or depending on) the byte order in which integers are stored.

10.7 Language mechanisms

To prevent access to the representation, we have throughout this lecture made the fields of our abstract type `private`. If they were public, then we lose any hope of hiding the representation from the client. We also lose the constraint that all changes to the representation happen through method calls – now they can happen through external interference.

Interfaces provide another method for supporting representation independence. For example, we could provide the three implementations for matrices as three distinct classes, say `TwoDimMatrix`, `HashMatrix`, and `OneDimMatrix`, and declare that they all meet a common `Matrix` interface. The interface only declares the operations and doesn't give representations or code:

```
public interface Matrix {
    int get(int r, int c);
    void set(int r, int c, int v);
    int rows();
    int cols();
}
```

Notice that we can't declare constructors in interfaces. Now each of our classes are declared as implementations of `Matrix`:

```
public class TwoDimMatrix implements Matrix {
    private int[][] data;
    public TwoDimMatrix(int r, int c) { data = new int[r][c]; }
    public int get(int r, int c) { return data[r][c]; }
    //...
}
public class OneDimMatrix implements Matrix {
    private int[] data;
    private int width;
    //...
}
public class HashMatrix implements Matrix {
    private Map data;
    private int height, width;
    //...
}
```

Whenever possible, clients should refer only to the `Matrix` interface, so the classes containing the representations are not accessible. Here's a standard idiom for creating and manipulating objects:

```
Matrix m = new HashMatrix(50,100);
//...
m.set(5,10,1);
```

Note that the interface can't be used to *construct* the object; an interface has no constructors, and it is at the point of creation that we need to specify the implementation. But we have carefully

declared the result of the constructor call as a `Matrix` and not a `HashMatrix`. A subsequent reference to `m.data` would now be illegal, even if the field were declared public.

The dependences on the concrete classes due to constructor calls are localized as much as possible, but sometimes we would like to mitigate them further. The Factory pattern, which we will discuss later in the course, addresses this particular problem.

Lecture 11: Representation Invariants

11.1 Context

What you'll learn: How to find representation invariants and avoid representation exposure.

Why you should learn this: An understanding of the theory of abstract types helps you avoid whole classes of nasty, subtle bugs – or at minimum alerts you to their existence.

What I assume you already know: How to write a simple ADT.

In the next two lectures, 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.

11.2 A Tale of Two Spaces: Rep and Abstract

We now take a deeper look at the theory underlying abstract data types. This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types. If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps.

In thinking about an abstract type, it helps to consider the relationship between two spaces of values. The space of *rep* or *representation* values consists of the values of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated. For now, though, it will suffice to view it simply as a mathematical value.

The space of *abstract* values consists of the values that the type is designed to support. These are a figment of our imagination. They're platonic entities that don't exist as described, but they are the way we want to view the elements of the abstract type, as clients of the type. For example, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; the fact that it might be implemented as an array of primitive (bounded) integers, say, is not relevant to the user of the type. But of course the implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.

Suppose we wish to create an abstract type to represent a set of characters, with the following specification:

CharSet represents a set of characters
class CharSet
spec (set **char**) cs;

CharSet()
  returns new result such that no result.cs

void add(**char** ch)
  modifies this.cs
  effects cs = \old(cs) + ch

void remove(**char** ch)
  modifies this.cs
  effects cs = \old(cs) - ch

boolean member(**char** ch)
  returns ch in cs

This specification uses a spec field `cs` with is a (mathematical) set of `char`s. Suppose we chose to implement this specification using a `StringBuffer` as follows:

```
public class CharSet {
    private StringBuffer s;
    public CharSet() {
        s = new StringBuffer();
    }
    public void add(char ch) {
        if (!member(ch)) s.append(ch);
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        if (index>=0) {
            s.deleteCharAt(index);
        }
    }
    public boolean member(char ch) {
        return s.indexOf(String.valueOf(ch))!=-1;
    }
}
```

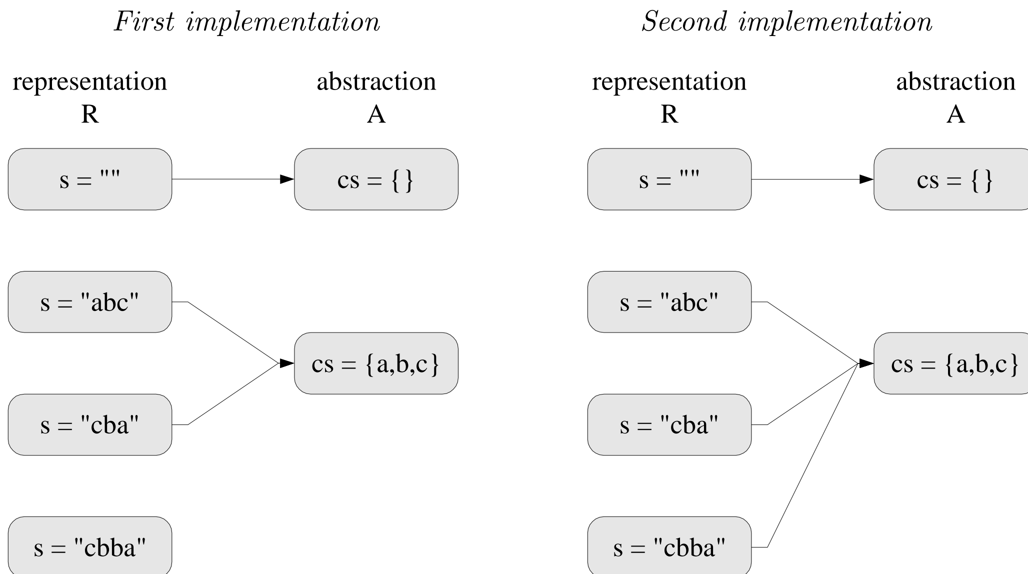
This works just fine. What are the rep values and the abstract values of this type? The abstract values are easy, and are given purely by the specification – they can be described by the possible values of the spec field `cs`: any set of characters, such as {} or {a,b,c}. The rep values are given by examining the implementation. They can be described by the possible values of the rep field `s` in the implementation. If you examine the code, you will discover that there are constraints on the possible values of `s`. The logic of the `add()` method is such that only one instance of each character can appear in `s`; "adb" is possible but not "adbd", for example. This is a useful fact, and vital to the correct functioning of the `remove()` method.

Contrast the following alternative implementation of `CharSet`:

```
public class CharSet {
    // everything else the same, except the following methods...
    public void add(char ch) {
        s.append(ch);
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        while (index>=0) {
            s.deleteCharAt(index);
            index = s.indexOf(String.valueOf(ch));
        }
    }
}
```

This works just fine too. But now the `add` method doesn't check for duplicates, so there is no constraint on `s`. The `remove` method needs to be changed to dovetail with the new `add` behavior.

We can look at the rep (R) and abstract (A) value spaces for the two implementations graphically, drawing arcs from each rep value to the abstract value it represents:



The graphs here are obviously not complete, and just show a few samples from infinite sets of values. There are several things to note about the graphs:

1. Every abstract value is mapped to from something. The purpose of implementing the abstract type is to support operations on abstract values. So we will need to be able to create and manipulate representations for all the possible abstract values we care about.

2. Some abstract values are mapped to by more than one rep value. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.

3. Not all rep values are mapped. In this case, the string "cbba" is not mapped for the first implementation. If the type of the rep field(s) are nontrivial, it will not make sense to give

an interpretation for all rep values. A doubly-linked list representation, for example, can be twisted into all kinds of pretzel configurations that won't correspond to simple sequences, and for which we won't want to write special cases in the code. Or sometimes we will want to impose certain properties on the rep to make the code of the operations more efficient or easier to write. In the first implementation, the implementer decided that the string `s` should not contain duplicates. This made it possible to terminate the `remove` method when it hits the first instance of a particular character, since we know there can be at most one. This optimization wasn't possible in the second implementation.

11.3 Rep Invariants and Abstraction Functions

In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite. So we describe it by giving two things:

- An abstraction function that maps rep values to the abstract values they represent:

  ```
  AF : R -> A
  ```

  The arcs in the previous diagram denote the abstraction function. In the terminology of functions, the properties 1-3 we discussed above can be expressed by saying that the function is onto, not necessarily one-to-one, and often partial.

- A rep invariant that maps rep values to boolean:

  ```
  RI : R -> boolean
  ```

  For a rep value `r`, `RI(r)` is true if and only if `r` is mapped by `AF`. In other words, `RI` tells us whether a given rep value is well-formed. Alternatively, you can think of `RI` as a set: it's the subset of rep values on which `AF` is defined.

We will look at abstraction functions next day. Today, let's look at the rep invariant, `RI`. For our first implementation of `CharSet`, it was:

```
RI(r) = r.s != null &&
        r.s contains no duplicates
```

Like specifications, rep invariants can be expressed in various levels of (in)formality. For example, `r.s contains no duplicates` could be written as something like `i!=j => r.s[i]!=r.s[j]`, but it is probably clearer as is.

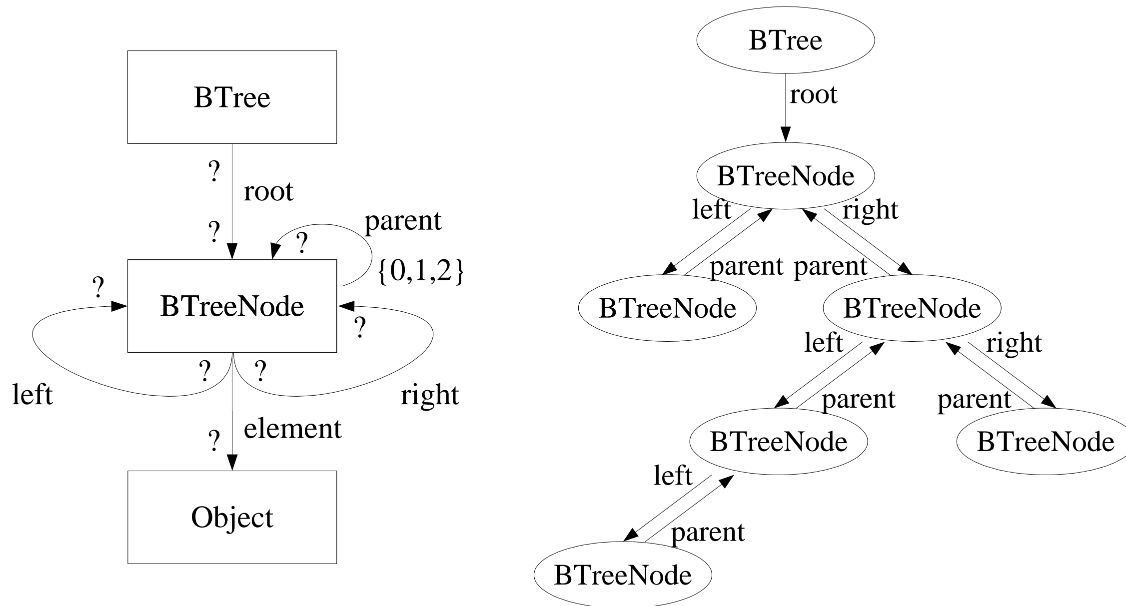
For our second implementation of `CharSet`, the rep invariant was weaker:

```
RI(r) = r.s != null
```

The rep invariant acts as a kind of *convention* for using the rep space. For example, once we decide that in our implementation of `CharSet` we will not allow duplicates in `r.s`, then we can write the `add()` and `remove()` method more independently. The abstraction function also helps with this, as we will see next day.

11.4 Another example

Consider a binary tree ADT, whose implementation has an object model as follows:



(object model on left, snapshot of a typical `BTree` on the right, omitting the `element` relation for clarity). We would like to write a rep invariante RI that is always true for our chosen representation:

```
all tree: BTree | RI(tree)
```

We can write RI recursively as:

```
RI(tree) = (tree.root!=null) =>
               (tree.root.parent = null && RI(tree,tree.root))

RI(tree,node) = ((node.left!=null) => (node.left!=node.right)) &&
                ((node.left!=null) =>
                    (node.left.parent = node && RI(tree,node.left))) &&
                ((node.right!=null) =>
                    (node.right.parent = node && RI(tree,node.right)))
```

This makes sure that the `parent`, `left` and `right` fields are consistent throughout the tree. Written this way, it is easy to imagine converting the `RI` to test code. You could also write the RI in many other ways; for example, the recursive construction is not necessary. The constraints in a tree are very similar to the file system example in a previous lecture. For example, to ensure that all nodes trace back to the root through their parent fields, we could write:

```
RI(tree) = all node: tree.root.*(left+right) | tree.root in n.*parent
           // ... other conditions, this is not complete ...
```

11.5 Rep Invariants for Modular Reasoning

The rep invariant makes *modular reasoning* possible. To check whether an operation is implemented in a way that is consistent with the rep invariant, 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.

Rep invariants can often be translated to code, and used to periodically check whether an object is in a valid state. A valid state is a rep value for which there is some corresponding abstract value – in other words, those rep values for which the rep invariant holds. If we detect that the rep invariant is false, then we know that something has gone wrong – the object could not possibly represent an abstract value. If the rep invariant is true, we know that the object does indeed represent an abstract value (though not necessarily the right one; this is not a panacea for detecting all bugs). Here is what we do:

- Create a method `checkRep()` which checks that the rep invariant holds (and fails immediately if it does not).
- Place calls to `checkRep()` at the end of every constructor, to confirm that the rep invariant holds when an object is first created.
- Place calls to `checkRep()` at the beginning and end of every public method: mutators, producers, and observers too (they may change the rep through *benevolent side-effects*, and anyway, do you really trust them?)

If we do all this, we can apply the following *structural induction*: if an invariant of an abstract data type is (1) established by creators; (2) preserved by producers, mutators, and observers; and (3) the representation of the type is never exposed, then the invariant is true of all instances of the abstract data type.

Notice the caveat about the representation being exposed (called *rep exposure*). How can this happen?

11.6 Rep Exposure

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

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

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

The simplest form of rep exposure involves allowing client code to manipulate the representation directly. We saw that this is easy to rule out by making all fields private, so we don't usually even regard it as a kind of exposure. Instead, the rep exposure we'll be concerned with arises because a mutable object inside the representation is accessible from the outside, through a different path. Two common ways in which this happens are:

- when a reference to a mutable object is passed in and made part of the rep, despite being accessible as an existing reference from the outside.
- when a reference to a mutable object in the rep is returned as the result of an operation.

Consider the following class representing an interval of time between two `Date`s:

```
public class Interval {
    private final Date start, stop;
    private final long duration;
    public Interval(Date start, Date stop) {
        this.start = start;
        this.stop = stop;
        duration = stop.getTime()-start.getTime();
    }
    public Date getStart() { return start; }
    public Date getStop() { return stop; }
    public long getDuration() { return duration; }
}
```

The rep invariant will capture the need for consistency between the duration, start, and stop fields. But unfortunately this implementation leaks like a sieve because the Java `Date` class is mutable, and it is easy for a client to end up having a reference to the `Date` objects stored in `start` and `stop` – and hence being free to change these without updating duration.

To avoid this rep exposure, we need to make *defensive copies*, using copy constructors, clone methods, or whatever is available:

```
public class Interval {
    //...
    public Interval(Date start, Date stop) {
        this.start = new Date(start.getTime());
        this.stop = new Date(stop.getTime());
        duration = stop.getTime() - start.getTime();
    }
    public Date getStart() { return new Date(start.getTime()); }
    public Date getStop() { return new Date(stop.getTime()); }
}
```

Another common example of rep exposure can occur when a method returns a collection. When the representation already contains a collection object of the appropriate type, it is quite tempting to return it directly. For example, in Java `List`s must have a method `toArray()` that returns an array of elements corresponding to the elements of the list. Suppose we implement a class `VeryExclusiveList` which only accepts `HighSociety` objects. If in our implementation we use an array, we might be tempted to just return this array when implementing the `toArray()` method. But this leaves us wide open:

```
List posh = new VeryExclusiveList();
posh.add(new HighSociety("Lord Vandersnoot"));
posh.add(new HighSociety("Lady Bassington-Bassington"));
//...
Object[] whosWho = posh.toArray();
whosWho[0] = new Ruffian();  // ouch! who left the door open...
```

A more subtle variant of this problem arises with iterators. Many Java classes have a method that returns an iterator. It is often a good design decision to provide a method that returns an iterator over a collection, rather than the collection itself, to save the cost of making defensive copies. But building an iterator is a lot of work for the implementor, so we might be tempted to use one that's already provided by the Java library. Suppose our representation includes an `ArrayList` field called `list` that holds a collection of elements, and we want to implement a method:

```
public Iterator elements()
```

that returns an iterator over those elements. Noticing that the `ArrayList` interface provides its own method that returns an iterator, we implement our method like this:

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

Unfortunately, this too is a rep exposure! The `Iterator` interface in Java includes an optional `remove` operation that allows the client to remove elements from the underlying collection. `ArrayList` implements this operation, so the result of this method is an iterator object that can actually affect the list collection, outside the abstract type.

Fortunately there's a solution to this problem:

```
public Iterator elements() {
    return Collections.unmodifiableList(list).iterator();
}
```

The `unmodifiableList` method in `Collections` creates a wrapper around the list you give it, which forbids all mutations through the wrapper.

11.7 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.

Lecture 12: Abstraction Functions

12.1 Context

What you'll learn: How to read, write, and use abstraction functions to describe the relationship between the abstract values of a type and its representation.

Why you should learn this: Together with the representation invariant, the abstraction function allows us to reason in a modular fashion about the correctness of operations.

What I assume you already know: Representation invariants, ADTs, object models.

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

12.2 Judging correctness

Recall the `CharSet` class from last lecture, which represented a set of characters as a string with no duplicates:

```
public class CharSet {
    private StringBuffer s;
    public CharSet() {
        s = new StringBuffer();
    }
    public void add(char ch) {
        if (!member(ch)) s.append(ch);
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        if (index>0) {
            s.deleteCharAt(index);
        }
    }
    public boolean member(char ch) {
        return s.indexOf(String.valueOf(ch))!=-1;
    }
}
```

Well, this is *almost* the same code as last lecture. But there is a bug. Can you see it? If we work through each method, it is clear that the rep invariant is never broken – `s` will never contain duplicates. So `CharSet` objects are always valid. But they may not be *correct*. Look at the `remove()` method. If the character to be removed is at the beginning of the string (`index==0`) then it is incorrectly ignored.

```
CharSet chars = new CharSet();
chars.add('a');   // chars is { a }
chars.add('b');   // chars is { a, b }
chars.remove('a');   // chars is { a, b } (wrong!)
chars.remove('b');   // chars is { a }
```

Because of the rep invariant, when we start debugging `remove()` we can be confident that it is at least starting from a valid representation. We don't have to consider the possibility that duplicate characters may have crept in during previous operations (which could indeed cause our `remove()` method to fail). For a simple class like `CharSet`, this may not seem like a big deal – but imagine debugging a `remove()` method for tree nodes, for example, and discovering after tearing out most of your hair that the method itself was working just fine, but a previous method call had left your object in a subtly inconsistent state that confused `remove()`. The rep invariant is a great aid to your sanity.

To judge whether the operations on `CharSet` are working correctly, we need to be able to interpret the representation (a `StringBuffer`) as an abstract value (a set of characters). We can formalize this with an *abstraction function*, mapping from representation space to abstract space `AF: R->A`:

```
AF(r) = a such that
   a.cs = { r.s[i] | 0 <= i < r.s.size }
```

where `cs` is the spec field used in the specification of `CharSet` (see previous lecture). We can write this more consisely as:

```
cs = { s[i] | 0 <= i < s.size }
```

You might get the impression that abstraction functions state the obvious: that just looking at the rep, you could guess how it should be interpreted. Much of the time, this is actually true, and for this reason abstraction functions are less important than rep invariants.

This assumes, by the way, that a human being is interpreting the rep. If you want to build a tool that analyzes code automatically for compliance with its spec – even just that modifications are within the scope permitted by a modifies clauses – you will need to provide the tool with an abstraction function.

Sometimes, however, a clever representation may have an interpretation that is far from obvious. In this case, an abstraction function is a very useful bit of documentation, both for yourself and others. Writing the abstraction function for a class like this is a useful exercise to make sure you really have a self-consistent idea of how your representation will work.

12.3 A Failure to Represent

This section is a case study in how writing an abstraction function can turn up subtle problems in a representation. In our discussion of rep exposure in the previous lecture, we saw that immutable

objects are easier to reason about, and can be shared freely, unlike mutable objects which must be guarded jealously. Suppose we would like to create an *immutable list* class, whose membership never changes over time. We know how to implement an immutable list in Scheme; the `cons` operation creates a new list whose tail is the old list, and the `car` and `cdr` operations do the opposite, breaking the list into the first element and the tail. Let's try writing this in Java:

```
public class Cons {
    private final Object car;
    private final Cons cdr;
    public Cons(Object head, Cons tail) {
        car = head;
        cdr = tail;
    }
    public Object car() { return car; }
    public Cons cdr() { return cdr; }
}
```

This looks workable. We can write code quite reminiscent of Scheme:

```
Cons list = new Cons("Elvis", new Cons("lives", null));
System.out.println(list.car()); // prints "Elvis"
System.out.println(list.cdr().car()); // prints "lives"
```

Now let's try to write an abstraction function which maps from a `Cons` object to a sequence:

```
AF(r) = [ r.car ] if r.cdr=null,
        [ r.car ] ^ AF(r.cdr) if r.cdr!=null
```

where `[x]` is a sequence containing the single element `x`, and `^` is sequence concatenation. This recursive abstraction function works quite well, giving on interpretation of `["Elvis","Lives"]` for the example above. It has one glaring flaw, though. There is no representation for the empty list! The role of empty list can be partially played by `null`; for example `new Cons("lives",null)` could be interpreted as prepending `"lives"` to the empty list. But `null` is not an object, and we cannot call methods on it; all clients of `Cons` would need to test for this special case, or else we would need to put `Cons` in a wrapper class that deals with this. So we find a mismatch between our intuitive idea of how `Cons` should work and the true logic of the obvious implementation. Writing the abstraction function for a class is a very good way to spot conceptual errors like this at an early stage.

12.4 A Nifty Abstraction Function

Suppose we were even more ambitious and decided to try to build an immutable `Queue` datatype. It's not obvious how to implement this efficiently. We know how to implement an immutable `List` by creating and breaking up `cons` structures. The snag with a queue is that the elements go on one end and come off the other. This does not come naturally with `cons` structures.

A very clever solution (well-known in the functional programming community) is to employ a *pair* of immutable lists, called say `forward` and `backward`. Here is an implementation:

```java
public class ImmutableQueue {
    private ImmutableList forward = new ImmutableList();
    private ImmutableList backward = new ImmutableList();

    public ImmutableQueue() {}

    public void enqueue(Object o) {
        backward = backward.cons(o);
    }

    public Object dequeue() {
        if (isEmpty()) { throw new EmptyQueueException(); }
        if (forward.isEmpty()) {
            while (!backward.isEmpty()) {
                forward = forward.cons(backward.car());
                backward = backward.cdr();
            }
        }
        if (!forward.isEmpty()) {
            Object o = forward.car();
            forward = forward.cdr();
            return o;
        }
        return null;
    }

    public boolean isEmpty() {
        return forward.isEmpty() && backward.isEmpty();
    }
}
```

`ImmutableList` is a wrapper around `Cons` that can deal with empty lists (you could try to write this class as an exercise).

How does `ImmutableQueue` work? Squinting at it for a while reveals that when objects are added to the queue, they are added to `backward`, and when they are removed from the queue they are taken from `forward`. And in between there is some kind of shuffling going on in the `dequeue` method. This is an example of an ADT that is much easier to understand using its abstraction function:

```
AF(r) = AF(r.forward) ^  rev(AF(r.backward))
```

We appeal to a function `rev` that reverses a list, `^` is the concatenation operator, and we assume that `ImmutableList` has an abstraction function of its own. We think of the elements as ordered so that the first element to be removed will be at the beginning of the list. The queue is broken into two parts that are concatenated together. Elements at the front of the queue appear in the list `forward` in natural order. Elements at the back of the queue appear in the list `backward`, in *reversed order*.

Here's how the rep is manipulated. To enqueue an element, we simply `cons` it to the `backward` list. To dequeue an element, we take it off the `forward` list using `car` and `cdr` if the list is non-empty. If it's empty, we reverse the `backward` list, and make it the `forward` list, and replace the `backward` list by the empty list. Here's an example:

Assume the queue is initially empty with the `backward` list and the `forward` list being empty. If A, B and C are enqueued, the `backward` list becomes [C,B,A] and the `forward` list is still empty. If we wish to dequeue, then the `backward` list is reversed to create an [A,B,C] `forward` list, the `backward` list is set to empty, and A is dequeued by taking the `car` of the `forward` list. At this point enqeueing D and E results in a `backward` list of [E,D] and a `forward` list of [B,C]. `AF(r)` will interpret this concrete configuration as the abstract queue [B,C,D,E].

Reversing the list takes time proportional to its length. So, if a lot of `enqueue` operations have been performed without an intervening `dequeue`, a single `dequeue` operation may take some time. But note that each element can only participate in one reversal. The total cost of the reversals over the life of the queue is proportional to the number of elements dequeued. So the cost of each operation is, averaged over all the operations, constant time. This kind of analysis is called an `amortized` analysis, because the cost of a single operation is 'amortized' over all operations.

12.5 Specification Fields

The abstract values of many abstract data types have a tuple structure at the top-level. For example, a line is a pair of points; a mailing address is a number, a street, a city and a zipcode; a URL is a protocol, a host name, and a resource name. In these cases, one can specify a single function that maps representation objects to tuples. This is the approach followed by the Liskov and Guttag book. But it's convenient, and perhaps more natural, to break the function into several separate functions, each viewed as *defining* a specification field.

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

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

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

This scheme is so convenient that we use it even when there is only one specification field. For example, when we refer to the ith element of a list `l` as `l.elts[i]`, this used a specification field `elts` whose value is a mathematical sequence. It allowed us to talk about the elements of the vector without mentioning the representation. Without the specification field, we would have to write `AF(l)` to denote the list's element sequence, to distinguish it from the value of `l` itself – a Java object reference. Our abstraction function for our immutable queue now becomes:

```
q.elts = q.forward.elts ^ rev(q.backward.elts)
```

Note that there are two different `elts` fields here: the one on the left corresponding to the abstraction function of queues, and the two occurrences on the right corresponding to the abstraction function of lists. For annotating code, specification fields are especially convenient. Using the convention that we can omit explicit mention of `this`, we might write, for example:

```
// abstraction function
// elts: sequence of elements in queue,
//       in order from first in to last out
// elts = front.elts ^ rev(back.elts)
```

as a comment in the `ImmutableQueue` class.

12.6 Common Confusions

A common confusion students have about abstraction functions and rep invariants is that they imagine that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone. If this were the case, they would be of little use, since they would be saying something redundant that's already available elsewhere.

It's easy to see why the abstract value space alone doesn't determine AF or RI: there can be several representations for the same abstract type. A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character. Clearly we need to separate functions to map these two different rep value spaces.

It's less obvious why the choice of both spaces doesn't determine AF and RI. The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values will be deemed to be legal, and of those that are legal, how they will be interpreted. Rather than deciding, as we did earlier, that the strings have no duplicates, we could instead allow duplicates, but at the same time require that the characters be sorted, appearing in nondecreasing order. This would allow us to perform a binary search on the string and thus check membership in logarithmic rather than linear time. Same rep value space – different rep invariant.

Even with the same type for the rep value space and the same rep invariant RI, we might still have different interpretations AF. Suppose RI admits any string of characters. Then we could define AF, as we did, to interpret the array's elements as the elements of the set. But there's no *a priori* reason to let the rep decide the interpretation. Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string "acgg" represents the set {`a,b,c,g`}.

The essential point is that designing an abstract type means not only choosing the two spaces – the abstract value space for the specification and the rep value space for the implementation – but also deciding what rep values to use and how to interpret them.

Lecture 13: Identity and Equality I

13.1 Quote of the Day

I like pigs. Dogs look up to us. Cats look down on us. Pigs treat us as equals.

Winston Churchill

13.2 Context

What you'll learn: Object equality and hash codes; how to meet the object contract.

Why you should learn this: As your objects are shuffled around in your program, into and out of collections for example, they will from time to time be tested for equality; if you haven't thought carefully about what this means for your classes, the result will be bugs whose effects are very non-local and hard to trace.

What I assume you already know: Proficient in reading specifications, somewhat familiar with the Java collection classes and the use of hash maps.

13.3 The Object Contract

In Java, 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, and the method for generating a hash code.

```
public boolean equals(Object o);
public int hashCode();
```

Like any other methods of a superclass, these methods can be overridden. We will see in lecture next week 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. So a class that overrides `equals` and `hashCode` should scrupulously obey their specification.

The specification of the `Object` class given in its documentation is rather abstract and may seem abstruse. But failing to obey it has dire consequences, and tends to result in horribly obscure bugs. Worse, if you do not 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 – i.e. be 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; and
- ▷ *hashCode* must produce the same result for two objects that are deemed equal by the *equals* method.

13.4 Equality and Inheritance

For the moment, let us ignore the `hashCode` method and 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 is hard to imagine how the equals method would be used: you would have to worry about whether to write `a.equals(b)` or `b.equals(a)`, for example, if it were not symmetric.

What is much less obvious, however, is how easy it is to break these properties inadvertently. The following example shows how symmetry and transitivity can be broken in the presence of inheritance. Consider a simple class that stores a duration in time, with a field for the number of days and the number of seconds:

```
public class Duration {
    private final int day;
    private final int sec;
    public Duration(int day, int sec) {
        this.day = day;  this.sec = sec;
    }
    public boolean equals(Object o) {
        if (!(o instanceof Duration))
            return false;
        Duration d = (Duration) o;
        return d.day == day && d.sec == sec;
    }
}
```

The `equals` method takes an `Object` as its argument; this is mandated by the object contract. The method returns `true` if the object passed is a `Duration` and stores the same interval (assume for now that the `day` and `sec` fields need to be exactly the same for two `Duration` objects to be considered equal). Now suppose find that we sometimes have a need for more precision, and we add a field for nanoseconds in a derived class:

```
public class NanoDuration extends Duration {
    private final int nano;
    public NanoDuration(int day, int sec, int nano) {
        super(day,sec);
        this.nano = nano;
    }
}
```

What should the `equals` method of `NanoDuration` look like? We could just inherit `equals` from `Duration`, but then two `NanoDuration`s will be deemed equal even if they do indeed differ by nanoseconds. We could override it like this:

```
public boolean equals(Object o) {
    if (!(o instanceof NanoDuration))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

This seemingly inoffensive method actually violates the requirement of symmetry. To see why, consider a `Duration` and a `NanoDuration`:

```
Duration d = new Duration(1,12);
NanoDuration nd = new NanoDuration(1,12,123);
System.out.println(d.equals(nd)); // true
System.out.println(nd.equals(d)); // false! – not symmetric
```

Notice that `d.equals(nd)` returns true, but `nd.equals(d)` returns false! The problem is that these two expressions use different `equals` methods: the first uses the method from `Duration`, which ignores nanoseconds, and the second uses the method from `NanoDuration`.

We could try to fix this by having the `equals` method of `NanoDuration` ignore nanoseconds when comparing against a normal `Duration`:

```
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    // if o is a normal Duration, compare without nano field
    if (!(o instanceof NanoDuration))
        return super.equals(o);
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

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

```
NanoDuration d1 = new NanoDuration(1,12,123);
Duration d2 = new Duration(1,12);
NanoDuration d3 = new NanoDuration(1,12,999);
System.out.println(d1.equals(d2)); // true
System.out.println(d2.equals(d3)); // true
System.out.println(d1.equals(d3)); // false! – not transitive
```

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

It turns out there is no solution to this problem: it is a fundamental flaw in inheritance. You cannot write a good `equals` method for `NanoDuration` if it inherits from `Duration` as written. However, there are two workarounds. The first is to change `Duration`'s `equals` method so that it rejects equality with any of its subclasses:

```
public class Duration {
    //...
    public boolean equals(Object o) {
        if (o == null || !o.getClass().equals(getClass()))
            return false;
        Duration d = (Duration) o;
        return d.day == day && d.sec == sec;
    }
    //...
}
```

Explicit comparison of classes is a stronger test than `instanceof`. A `NanoDuration` is an `instanceof` `Duration`, but `NanoDuration.getClass() != Duration.getClass()`. The drawback of this approach is that you lose the ability for harmless `Duration` subclasses to compare for equality to a `Duration`. For example, you might write a subclass `ArithmeticDuration` that doesn't add any new attributes to `Duration`, but merely offers some new methods for adding and subtracting durations. With the `getClass` workaround, a `ArithmeticDuration` can never equal a `Duration`, even though it has the same `(day,sec)` value internally. The second workaround is to implement `NanoDuration` using `Duration` in its representation, rather than inheriting it:

```
public class NanoDuration {
    final Duration d;
    final int nano;
    //...
}
```

Since `NanoDuration` no longer extends `Duration`, the problem goes away. This strategy is called *composition*; we will see more of it in a later lecture. Bloch's book 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(Duration d)
```

where you've substituted another type for `Object` in the declaration of equals?

The problems we've discussed here are based on two Java classes, `Date` and `Timestamp`. `Timestamp` extends `Date` by adding a nanosecond field to it, just like our example. This causes so many problems when objects of the two classes are compared that more recent versions of the specification for `Timestamp` are full of caveats and complexities:

> Note: This type is a composite of a java.util.Date and a separate nanoseconds value. Only integral seconds are stored in the java.util.Date component. The fractional seconds - the nanos - are separate. The Timestamp.equals(Object) method never returns true

> when passed a value of type java.util.Date because the nanos component of a date is unknown. As a result, the Timestamp.equals(Object) method is not symmetric with respect to the java.util.Date.equals(Object) method. Also, the hashcode method uses the underlying java.util.Data implementation and therefore does not include nanos in its computation.
>
> Due to the differences between the Timestamp class and the java.util.Date class mentioned above, it is recommended that code not view Timestamp values generically as an instance of java.util.Date. The inheritance relationship between Timestamp and java.util.Date really denotes implementation inheritance, and not type inheritance.

In this last paragraph, the designers are basically throwing up their hands and saying that the only way to solve this problem is to deny `Timestamp`'s parentage, and treat it as an entirely separate class to `Date`. In retrospect, this could have been done by using `Date` in the *implementation* of `Timestamp`, rather than using it in the specification. We will return to this topic in a later lecture.

13.5 Equality and Efficiency

To understand the part of the Object contract relating to the `hashCode` method, you'll need to have some idea of how hash tables work. Hash tables are a fantastic invention – one of the best ideas of computer science. A hash table is a representation for a mapping: an abstract data type that maps keys to values. Hash tables 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 hash table 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 (e.g., by a modulo division). The value is then inserted at that index.

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 hash table 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 Joshua Bloch's book *Effective Java* 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.)

Some years ago, so the legend goes, a 6.170 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...

What hash code would work for our `Duration` class, where `a.equals(b)` if they both have the same `day` and `sec` fields? Here are some possibilities that would work, but lead to greater or lesser efficiency in hash tables:

```
// always safe, but makes hash tables completely inefficient
public int hashCode() {
    return 1;
}
 // safe; but collisions for Durations that differ in sec only
public int hashCode() {
    return day;
}
 // safe; collisions possible but don't seem likely to happen systematically
public int hashCode() {
    return day+sec;
}
```

It is important to realize that the default `hashCode` implementation inherited from `Object` is *not* appropriate. The default implementation is generally based on some part of the address of an object in memory. This works just fine as a hash code if the only thing an object is equal to is itself. But once we can create two objects which we wish to treat as equal (for example, two `Duration` objects with the same `day` and `sec` fields), then this hash code is inappropriate, since it may give these equal objects different hash codes.

Suppose we now changed our idea of equality for `Duration` to the following:

```
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    Duration d = (Duration) o;
    return 24*60*60*day+sec == 24*60*60*d.day+d.sec;
}
```

Now the `day` and `sec` fields do not have to be exactly the same for two `Duration`s to be equal, as long as the total number of seconds represented is the same. Hash codes of `day` or `day+sec` as we had above now no longer work, since two equal `Duration`s might be given different hash codes. Here is one that works:

```
public int hashCode() {
    return 24*60*60*day + sec;
}
```

So we see that whenever the behavior of the `equals` method changes, we must reconsider whether the `hashCode` method is still safe.

Lecture 14: Identity and Equality II

14.1 Context

What you'll learn: Behavioral and observational equivalence, common problems with equality and collections.

Why you should learn this: Intuitions about equality can be inconsistent; if you don't think through your approach to object identity it can be very hard to fix problems that arise later.

What I assume you already know: The basic object contract covered in the previous lecture.

14.2 Equality and Time

If two objects are equal now, will they always be equal in future? For mathematical objects, the answer is "yes"; for Java objects the answer is "you choose" – the Object Contract doesn't specify. For immutable objects, it is natural for equality to be eternal, but for mutable objects we face a choice. For example, consider Java's mutable `StringBuffer` class. Its designers chose to implement its `equals` method so that each `StringBuffer` is equal to itself, and no other object – not even another `StringBuffer` containing the same characters. We call this *referential equality*, and it what the default `equals` method in `Object` provides (and what the `==` operator tests).

```
StringBuffer s1 = new StringBuffer("hello");
StringBuffer s2 = new StringBuffer("hello");
System.out.println(s1.equals(s2));  // false
```

Of course, the designers could have chosen differently, and made `equals` return `true` in such situations. But then equality would not be eternal – two `StringBuffer`s with the same content now may not have the same content in future. We see this other transient kind of equality implemented with Java's `Date` class, where equality is based on a field-by-field comparison of the objects' contents:

```
Date d1 = new Date(0);   // January 1, 1970 00:00:00 GMT
Date d2 = new Date(0);
System.out.println(d1.equals(d2));  // true
d2.setTime(1); // a millisecond later
System.out.println(d1.equals(d2));  // false
```

In the recommended text for this course, Professor Liskov presents a systematic approach to thinking about object equality. The first step is to distinguish *behavioral* from *observational* equivalence.

We say that two objects are *behaviorally equivalent* if there is no sequence of operations that can distinguish them. For example, two `String` objects with the same content are behaviorally equivalent since there are no methods we can call on either that will distinguish them. The `Date` objects *d1* and *d2* from above are *not* behaviorally equivalent, since if we mutate one of them with `setTime()` we can distinguish the objects by their different responses to `getTime()` (for example). Behavioral equivalence ensures that if two objects are equal or unequal at one point, they continue to be equal or unequal forever after; there is nothing you can do to two unequal objects that will make them become equal.

Two objects are *observationally equivalent* if there is no sequence of *observer* operations that can distinguish them – that is, we exclude mutators from consideration. On these grounds, the two `StringBuffer` objects `s1` and `s2` are observationally equivalent. The two `Date` objects `d1` and `d1` are initially observationally equivalent, until `d1` is mutated. We exclude the `==` operator from the set of operations we can use to distinguish objects. If it is included, then no objects are equivalent.

Following what we will call the **Liskov approach** to equality, you provide two distinct methods for your classes: `equals`, which returns true when two objects in a class are behaviorally equivalent, and `similar`, which returns true when two objects are observationally 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.

For example the `StringBuffer` class has an `equals` method that is compatible with the Liskov approach. A mutable object is only behaviorally equivalent to itself. If we augmented that class with a `similar` method, here is how it would work:

```
StringBuffer s1 = new StringBuffer("hello");
StringBuffer s2 = new StringBuffer("hello");
System.out.println(s1.equals(s2));  // false
System.out.println(s1.similar(s2));  // true
```

The `String` class also has an `equals` method that is compatible with the Liskov approach. An implementation of `similar` for `String` would simply call `equals`, since behavioral and observational equivalence are the same thing for an immutable object.

The Liskov approach, when applied uniformly, is easy to understand and works well. But it's not always ideal. Sometimes it is convenient (though risky!) to construct objects by calls to mutators during an initialization phase before treating them as immutable from then on, with equality being based on a field-by-field comparison. The Liskov approach discourages this (which is quite reasonable, since it is error-prone), although it is still possible by wrapping the object as an immutable "snapshot" after initialization. A more important pragmatic concern is that Java itself does not follow the Liskov approach, particularly in Java Collections.

14.3 Equality and Java Collections

The designers of the Java collections API did *not* follow the Liskov approach. There is no `similar` method, and `equals` is observational equivalence. What are the consequences of this choice?

Consider the following scenario. 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 hash 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.

In words, this program sounds okay. But it will actually 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.

You might think that one solution to this problem would be for `Set` to 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.

If `Set` and `LinkedList` used behavioral equivalence, then both code fragments in this section would work the way we want them to. But this is not what Java does, and we have to live with that.

14.4 Problems with Equality and Mutation

There are some subtle consequences of the Java approach, including the risk of representation exposure. Consider a `Set`, which must not contain two equal elements (expressed as a rep invariant: for all $i \neq j$ `elt[i].equals(elt[j])` is false). If equality of elements is defined as behavioral equivalence, then this can be implemented by simply refusing to add an element which is equal to one that is already present. But if any elements instead use observational equivalence, then this can run into trouble. A client can now mutate an object in the set to make it equal to another object in the set, thereby breaking `Set`'s rep invariant.

```
Set s = ...;
Date d1 = new Date(0);
Date d2 = new Date(1000);
```

```
s.add(d1);
s.add(d2);
System.out.println(s.size()); // 2 elements
d2.setTime(0);
System.out.println(s.size()); // still 2 elements
```

A consistent application of the Liskov approach doesn't suffer from this problem. A sensible implementation of `Set` needs to be able to test behavioral equivalence, but with the Java approach there is no guarantee (or requirement) that `equals` does that. The best Java can do is but caveats in its specification of `Set`:

> 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.

The same problem applies to the keys of `Map`s, which should form a set.

14.5 Problems with Hashing and Mutation

A separate problem arises for `Sets` (and `Maps`) implemented using hash tables, when mutable elements are used. Consider the following code:

```
List friends =
    new LinkedList(Arrays.asList(new String[] {"yoda","zaphod"}));
List enemies =
    new LinkedList(Arrays.asList(new String[] {"mr big","dr evil"}));
Set h = new HashSet();
h.add(friends);
h.add(enemies);
System.out.println(h.contains(friends)); // true
friends.add("yoda");
System.out.println(h.contains(friends)); // false
```

Here we have a set with two mutable elements. After they are inserted in the set, one of them is mutated. The two elements remain unequal, so the problem discussed in the previous section does not apply. But a new problem arises. The mutated element will probably now have a different hash code than it had at the time of insertion. So if we test for the presence of that element in the set, the `Set` is very likely to report that the element is not present (unless the hash code by coincidence remains the same). The `Set` will be looking for our element in our hash bucket. Vexingly, if we iterate through the `Set`, we *will* find our element. So the `Set` is now in an inconsistent state.

Again, this problem would not arise if `Set` used behavioral equivalence. The practical remedy for this problem is the same as in the previous section – use immutable elements, or never mutate elements once they are in a `Set`. The same applies to keys in a `Map`.

14.6 Problems with Self-Containment

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.
The algorithm used to calculate `hashCode` for `List` is actually specified in detail as:

```
hashCode = 1;
Iterator i = list.iterator();
while (i.hasNext()) {
    Object obj = i.next();
    hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
}
```

This is verbatim from the Java specification. What gives? Why is code showing up in a specification? It is a thorny question. Giving the code commits Java to that code; better hashing techniques cannot be added without causing pain, and if the implementation is bad the consequences are amplified (this actually happened with the `hashCode` for `String` in an early Java version). But if it is omitted, clients cannot be sure whether their use of hashes will be efficient. Java classes generally give the details of their `hashCode` implementation, and are worth looking at as examples of how such methods are written in practice (see e.g. the `hashCode` specification for `Set`).

The recursion in `hashCode` results in a very nasty surprise. The following code, in which a list is inserted into itself, will actually fail to terminate!

```
List lst = new LinkedList();
lst.add(lst); // okay so far...
int h = lst.hashCode(); // infinite loop!
```

It is also possible for `equals` to fail to terminate (see if you kind find a way to do this). 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.

Although behavioral and observational equivalence are perfectly well defined concepts for structures that contain references to themselves, or other kinds of twisted recursion (can two such structures be distinguished by method calls?), implementing tests for equivalence that work in these cases gets tricky.

14.7 What to do?

You may be wondering what we require you to do for your classes, and whether follow the Liskov approach or the Java approach. Both are in fact 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.

Lecture 15: Subtyping

15.1 Context

What you'll learn: when one type can safely be substituted for another; when inheritance helps and when it hurts; the distinction between subclassing and subtyping.

Why you should learn this: Correct use of subtyping is key to achieving many of the benefits of specifications discussed in previous lectures, such as decoupling and interchangeable families of components.

What I assume you already know: At this point you should have some practical experience with extending classes in Java.

15.2 The Lure of Subclassing

Frequently, as programmers, we find ourselves wanting to create a new class that is very like one we already have, but with a few additions. For example, perhaps we run a web store with a class for `Products`:

```
class Product {
    private String title, description;
    private float price; // floats are actually a bad way to store prices (why?)
    public float getPrice() { return price; }
    public float getTax() { return getPrice() * 0.05f; }
    //...
}
```

Suppose some of our products don't sell so well, so we want to offer discounts on them. Hopefully you would not even consider implementing this by cutting and pasting the `Product` code and tinkering with it to make the new class, such as this one:

```
class SaleProduct {
    private String title, description;
    private float price;
    private float factor;
    public float getPrice() { return price*factor; }
    public float getTax() { return getPrice() * 0.05f; }
    //...
}
```

Cutting and pasting leads to nightmarish maintenance problems, since if something changes (such as the sales tax rate) you have to update it in multiple locations, being careful about the different customizations that have been made. Your instinct in a situation like this may instead be to use inheritance:

```
class SaleProduct extends Product {
    private float factor;
    public float getPrice() { return super.getPrice()*factor; }
    //...
}
```

`SaleProduct` need not implement methods and fields that appear in its superclass `Product`; the `Product` versions are automatically used by Java when they are not overridden in the subclass. This approach avoids copying all the common code, which is tiresome and error-prone (common errors are failure to copy correctly or failure to make a required change). Additionally, if a bug is found in one version, there is no longer a risk of forgetting to propagate the fix to all versions of the code. Finally, it is much easier to comprehend the distinction between two classes when written this way (where just the changes are highlighted), as opposed to the cut-and-paste case where you need to search for differences hidden in a mass of similarities.

The mechanism we have used here, called *subclassing*, is a very useful tool for implementation reuse, and is supported by many programming languages. Implementations of subclasses need not repeat unchanged fields and methods, but can reuse those of the superclass.

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

15.3 Java Subtypes

The `getPrice` method of `SaleProduct` 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, `SaleProduct` can reuse `Product`'s `getTax` method unchanged. When `getTax` is invoked on a `SalesProduct`, the inherited `Product` version is used. Then, the call to `getPrice` inside `getTax` invokes the version of that method appropriate for the *runtime type* of the object (`SaleProduct`), so the `SaleProduct` 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. For example, in the following code:

```
Product p = new SaleProduct();
//...
System.out.println(p.getPrice());
```

the `getPrice` method called is `SaleProduct`'s, not `Product`'s. 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. We will return to this issue towards the end of the lecture.

Java types are classes, interfaces, or primitives. 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:

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. For each method in the subtype that corresponds to one in the supertype
   - ▷ The arguments must have the same types.
   - ▷ The result must have the same type.
   - ▷ There must be no additional declared exceptions.

Potentional advantages of Java subtyping are:

- 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; as we saw above, 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.

In fact, Java's notion of subtyping is too weak to really guarantee that clients need not change their code when passed a subtype. We have already seen this in the lectures on equality, with Java's `Date` and `Timestamp` classes, where `Timestamp` extends `Date` but the client is urged never to actually treat a `Timestamp` as a `Date`. Let's look at another example of the weakness in Java's subtypes, and then introduce a much stronger, more useful idea of *true subtypes*, based on what is really required for substitution of types to be possible.

15.4 Examples: Hashtable and Properties

The designers of the Java library made an early mistake related to subclassing and subtyping that they probably wish they could take back. Java 1.0 had a `Hashtable` class for storing arbitrary associations between keys and values (this class is very similar to `HashMap`, except for synchronization and a few other issues)

```
public class Hashtable {
    public Object put(Object key, Object value);
    public Object get(Object key);
    //...
}
```

Java 1.0 also had a class `Properties` for storing a property list – a set of name-value pairs where both are strings. It makes sense to reuse the implementation of `Hashtable` in the implementation of `Properties`, since the two classes are very similar, and in fact share some common methods

(like `isEmpty`). Unfortunately, the implementation was reused by making `Properties` a subclass of `Hashtable`:

```
public class Properties extends Hashtable {
    public Object setProperty(String key, String value);
    public String getProperty(String key);
    //...
}
```

What's wrong with this? Plenty. The specification for the `Properties` class states that it should only be used for storing string names and values. In fact, its methods for saving and loading the property list from a file break if any non-string objects were stored in the table. Unfortunately, by making `Properties` a subclass of `Hashtable`, the designers implicitly permitted any `Properties` object to be used as if it were a `Hashtable`. Arbitrary objects can be added by calling the `put` method inherited from `Hashtable`, and accessed using `get`. A further complication is that the `Properties` class has a mechanism for introducing default values for keys. This makes the meaning of inherited methods such as `contains`, `size`, or `remove` somewhat ambiguous (do they take the defaults into consideration or not?). For example, it is quite possible that `getProperty("foo")` would return a value, while `get("foo")` fails and `contains("foo")` returns `false`. This is potentially confusing to say the least. By the time these problems was discovered, it was too late to correct them because clients depended on the use of nonstring keys and values. For backwards compatibility reasons, this design mistake is still found in the Java library years later, with the following caveat in the specification for `Properties`:

> Because `Properties` inherits from `Hashtable`, the `put` and `putAll` methods can be applied to a `Properties` object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not `Strings`. The `setProperty` method should be used instead. If the `store` or `save` method is called on a "compromised" `Properties` object that contains a non-`String` key or value, the call will fail.

There is another way to reuse implementation which would have been far better than subclassing in this case: *composition*. In other words, make the hash table part of the representation of `Properties`, and delegate calls to it instead of inheriting methods from it:

```
public class Properties /* no longer extends Hashtable */ {
    private Hashtable map;
    //...
    public Object setProperty(String key, String value) {
        return map.put(key,value);
    }

    public String getProperty(String key) {
        //...
    }
    //...
}
```

15.5 True Subtypes and the Substitution Principle

How can we avoid running into problems like `Properties` or `Timestamp`? By using a much stronger and well-founded notion of subtyping, rather than whatever notion our chosen implementation language (Java) happens to support.

We will say that type A is a *true 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. Type theorists often write the subtype relation using a squared-off subset symbol:

$$A \sqsubseteq B$$

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

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 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 no more "requires" clauses, and each one is no more strict than the one in the supertype method.
     - the argument types may be supertypes of the ones in the supertype. In other words, for a parameter type P_{super} in the supertype, then parameter type P_{sub} is permitted in the subtype if $P_{super} \sqsubseteq P_{sub}$. 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. Any exception type E_{sub} thrown in the subtype must be a subtype of an exception for the supertype E_{super}.
     - there are no more 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. In other words, for a result type R_{super} in the supertype, then result type R_{sub} is permitted in

the subtype if $R_{sub} \sqsubseteq R_{super}$. 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:

```
Product B.f(Product arg);
SaleProduct A.f(Object arg);
```

Method `B.f` takes a `Product` as its argument, but `A.f` can accept any `Object` (which includes all `Products`). Method `B.f` returns a `Product` as its result, but `A.f` returns a `SaleProduct` (which is indeed a `Product`).

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.)

For example, suppose we have a class `Interval` which is specified to represent a non-negative interval. We then decide it would be useful to have negative `Intervals` (so we can do proper arithmetic) and create a subclass `SignedInterval` with a new `negate` method.

`SignedInterval` is *not* a true subtype of `Interval`. Even though there is no problem with any method of `SignedInterval` (`negate` is a new method, so the rules about corresponding methods do not apply) this method violates a promise made by `Interval`.

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.

These requirements have some similarity to those of Java subtypes, but are by no means the same.

- Java requires type equality for the arguments and result of a method, which is stronger than strictly necessary to guarantee type-safety, as we've seen above. This prohibits some code we might like to write. However, it simplifies the Java language syntax and semantics.
- Java has no notion of a behavioral specification, so it can perform no checks on *properties* and can make no guarantees about behavior.

15.5.1 Example: Square and Rectangle

We know from elementary school that every square is a rectangle. Suppose we are writing a graphics program, and decide we want to make `Square` a subtype of `Rectangle`, which included a `setSize` method:

```
public class Rectangle {
    //...
    public void setSize(int w, int h);
}
```

```
public class Square extends Rectangle {
    // what should go here for setSize?
}
```

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

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

public void setSize(int edgeLength);

// throws BadSizeException if w != h
public 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 `Squares` can't be substituted for `Rectangles`. 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 subtypes is weaker than the 6.170 notion of true subtypes.

There isn't a good 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

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

method that shrinks or grows a shape. Other solutions are also possible. But when you meet problems like this, it is worth reconsidering whether you are using inheritance appropriately.

Java assumes that subclasses can be substituted, and provides language support to make that easy. If your subclasses are not actually true subtypes, then you will need discipline to avoid running into problems. As a general rule, and particularly for classes you expose to clients:

> If it isn't a true subtype, don't make it a subclass.

But even that isn't the full story. Further perils lurk in subclassing even if you have been careful to use it only for true subtypes.

15.6 Dangers of Inheritance

Inheritance (subclassing) is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software. It is generally safe to use inheritance within a package, where the subclass and superclass implementation are under the control of the same programmer; beyond this great care is required.

Unlike method invocation, inheritance breaks encapsulation. A subclass depends on the implementation details of its superclass for its proper function. The superclass' implementation may change from release to release, and if it does the subclass may break, even though its code has not been touched. The subclass must evolve in tandem with its superclass, unless the superclass' authors have designed and documented it specifically for the purpose of being extended.

Let's suppose that we have a program that uses a `HashSet`. To tune the performance of our program, we need to query the `HashSet` as to how many elements have been added since it was created. This is not to be confused with its current size, which goes down when an element is removed. To provide this functionality, we write a `HashSet` variant that keeps count of the number of attempted element insertions and exports an accessor for this count. The `HashSet` class contains two methods capable of adding elements, `add` and `addAll`, so we override both of these methods:

```
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet() { }
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

This class looks reasonable, but it doesn't work. Suppose we create an instance and add three elements using the `addAll` method:

```
InstrumentedHashSet s = new InstrumentedHashSet();
System.out.println(s.getAddCount()); // 0
s.addAll(Arrays.asList(new String[] {"S", "Cr", "P"}));
System.out.println(s.getAddCount()); // ?
```

We would expect the `getAddCount` method to return three at this point, but it returns six. What went wrong? Internally, `HashSet`'s `addAll` method is implemented on top of its `add` method,

although `HashSet` does not document this implementation detail, and it shouldn't have to. The `addAll` method in `InstrumentedHashSet` added three to `addCount` and then invoked `HashSet`'s `addAll` implementation using `super.addAll`. This in turn invoked the `add` method, as overridden in `InstrumentedHashSet`, once for each element. Each of these three invocations added one more to `addCount`, or a total increase of six: Each element added with the `addAll` method is double-counted!

We could "fix" the subclass by eliminating its override of the `addAll` method. While the resulting class would work, it would depend for its proper function on the fact that `HashSet`'s `addAll` method is implemented on top of its `add` method. This "self-use" is an implementation detail, not guaranteed to hold in all implementations of `HashSet` and subject to change from release to release. Therefore, the resulting `InstrumentedHashSet` class would be fragile.

A slightly better solution is to override the `addAll` method to iterate over the specified collection, calling the `add` method once for each element. This amounts to reimplementing superclass methods that may or may not result in self-use, which is difficult and error-prone. Additionally, it is not always possible to do this if the method requires access to private fields inaccessible to the subclass.

The above problem stems from overriding methods. We've already seen a way to get around undesirable problems introduced through inheritance, and it works again here – *composition.* Instead of extending an existing class, give your new class a private field that references an instance of the existing class. This design is called composition because the existing class becomes a component of the new one. Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results. This is known as forwarding. The resulting class will be solid, with no dependencies on the implementation details of the existing class. Here is how it works:

```
public class InstrumentedHashSet {
    private final HashSet s;
    private int addCount = 0;
    public InstrumentedHashSet(HashSet s) {
        this.s = s;
    }
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

Now, in order to get a `InstrumentedHashSet` we first need to create a new `HashSet` and pass it to the constructor for `InstrumentedHashSet`. The `InstrumentedHashSet` class is known as a *wrapper* class because each `InstrumentedHashSet` instance wraps another `HashSet` instance.

15.7 Summary

We have defined true subtyping using the substitution principle: A is a true subtype of B when any object that satisfies A's specification also satisfies B's specification (and hence an object of type A can be used anywhere a B is expected). Subtypes in Java do not meet this definition, and consequently give no guarantee of substitutability. Subclassing is appropriate only in circumstances where the subclass really is a *true subtype* of the superclass. In other words, a class B should extend a class A if B is a subtype of A. If you are tempted to have a class B extend a class A, ask yourself the question: "Is every B really an A?" If you cannot answer yes to this question, B should not extend A. If the answer is no, it is often the case that B should contain a private instance of A and expose a smaller and simpler API: A is not an essential part of B, merely a detail of its implementation.

Even if you have a true subtype, you should carefully consider the use of `extends`, because as can be seen by the example of `InstrumentedHashSet` – which is almost a true subtype of `HashSet`, if we discount the object contract (do you see the problem we would encounter writing an appropriate `equals` method?) – the implementation of the subclass may not work due to unspecified behavior of the superclass. What happens in that example is that the subclass methods break because the methods of the superclass have an implicit dependence between them which is not in the superclass specification. You should be able to convince yourself that dependences amongst the superclass methods will not impact subclass behavior except through the use of `extends`.

As we've seen, there are a number of violations of this strategy in the Java platform libraries. A property list is not a hash table, so `Properties` should not extend `Hashtable`. A stack is not a vector, so `Stack` should not extend `Vector`. In both cases, composition would be appropriate. Using inheritance where composition is appropriate can lead to confusing semantics, as we saw with `Properties` and `Hashtable`s. All of these led to problems and confusion, which hopefully you can avoid in your own projects!

Lecture 16: Dependencies I

16.1 Context

What you'll learn: How to create module dependency diagrams (MDDs) to capture and communicate the dependences between cooperating parts of a program.

Why you should learn this: MDDs let you see at a glance crucial properties of your program. Drawing MDDs is useful as part of your design process before you start writing code, and you will often find it helpful to draw an MDD for a program that you're maintaining if one doesn't already exist.

What I assume you already know: Familiar with object model notation; comfortable with the object contract, Java collections, specifications and testing.

16.2 Quote of the Day

> *Coupling is the path to the dark side. Coupling leads to complexity. Complexity leads to confusion. Confusion leads to suffering. Once you start down the dark path, forever will it dominate your destiny, consume you it will.* Yoda (slightly paraphrased)

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.

16.3 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*. What benefits come from dividing a program into smaller parts? Here are some:

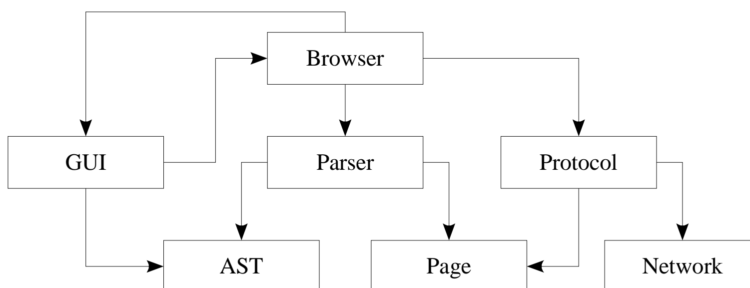
- ▷ *Division of labor.* A program doesn't just appear out of thin air: it has to be built gradually. If you divide it into parts, you can get it built more quickly by having different people work on different parts.
- ▷ *Reuse.* Sometimes it's possible to factor out parts that different programs have 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 building it in small parts. Each time a part is complete, it can be analyzed for correctness
- ▷ *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.

16.4 Parnas's Uses Relation

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

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

Suppose, for example, we are designing a web browser. Here is a uses diagram showing a possible design:



The `Main` module uses the `Protocol` module to engage in the HTTP protocol, the `Parse` module to parse the HTML page received, and the `GUI` module to display it on the screen. These modules in turn use other modules. `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 `AST` module 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.

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

- **Reusable subsets**. Which groups of modules can we take and reuse in another program? Suppose we want the functionality that `Parser` provides. Then since `Parser` uses `AST` and `Page` we will need to take those also. So the set of modules {`Parser,AST,Page`} forms what Parnas calls a 'reusable subset'. If we want `Browser`, we need everything; that is, there is no reusable subset that contains `Browser` that isn't the whole program.

- **Testing and order of construction**. What order should we build the program in, and where will we need testing stubs? To test `Network`, no other modules are needed. To test `Protocol` we need `Network` and `Page`. This suggests an ordering: we construct the leaves first, and then the modules on which they depend, and so on. If the program is layered, this strategy will allow us to avoid writing stubs entirely. Or in the case of a program like ours, it will tell us which modules must be implemented together or which stubs we must write.

- **Division of labor**. The uses diagram helps figure out how to allocate the task of implementing modules to different teams. Suppose we have two teams. It's clear that dividing into

{`Browser`,`GUI`,`Parser`,`AST`} and {`Protocol`,`Page`,`Network`} is a better choice than dividing into {`GUI`,`Parser`,`Protocol`} and {`Browser`,`AST`,`Page`,`Network`}, for example, because it allows subprograms to be put together that can be tested independently with fewer stubs. Perhaps it suggests that `Page` would be a strategic module to do early, to make independent work on `Parser` and `Protocol` easy.

But the uses diagram is lacking in some key respects:

- ▷ **Transitivity**. By definition, it's transitive. In other words, if `Browser` uses `Protocol` and `Protocol` uses `Network`, then `Browser` uses `Network` too. But perhaps the whole reason for inserting `Protocol` between `Browser` and `Network` was to decouple them from one another. The problem here is that the uses relation doesn't say how one module uses another, so in this scenario, it's not possible to distinguish the modifications to `Network` that will be propagated through to `Protocol` (and further through `Protocol` to `Browser`) from those that won't. This makes the uses diagram unsuited for reasoning about change propagation, and also for answering questions about which modules should be examined to trace a bug in the behavior of one module (or to verify that the module operates correctly).
- ▷ **Modern programming features**. It's not obvious how to draw a uses diagram for a Java program, in which there are modules that are actually specifications and not executable code (ie, interfaces), and in which relationships between modules can arise on the fly at runtime (because passing an object essentially passes a handle to the code of its class).

16.5 Liskov's Module Dependency Diagram

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

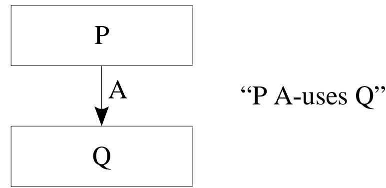
So now it's clear that some kinds of changes don't propagate. If we want to change B, but only the implementation and not the specification, then A need not change. If we want to change the specification of B, then A is likely to need changing. Oddly, this may not require a change to the implementation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

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

16.6 The 3-Element Model

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

assumption is discharged by Q. There's a lot of new terms here so let's look at this statement piece by piece.



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

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

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

Why this business about how the program is configured? The point is that a module may be designed with an assumption in mind, but may be written in such a way that which module discharges that assumption is postponed until runtime. The Java `HashMap` class, for example, assumes that the implementation of its keys obeys the `Object` contract. So in fact we will show a dependence of `HashMap` on every class that might be a key for it in the particular program under consideration - and there may be several such classes.

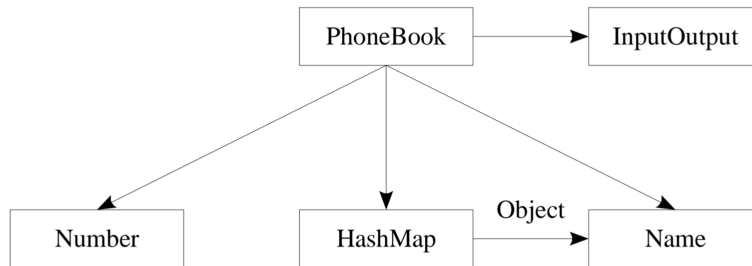
16.7 Example: Phone Book

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

- `PhoneBook`, the main class.
- `Name`, a class implementing an abstract data type for names. It will provide methods to parse a name into a first name and surname, a `toString` method to format a name for output, an `equals` method to check whether two names are the same, and perhaps other observers that implement looser matches.

▷ Number, a class implementing an abstract data type for phone numbers. It will provide methods to parse a number into area code, number, extension, etc, and a toString method to format a number for output.
▷ InputOutput, a class which encapsulates all the input-output facilities we'll need.
▷ HashMap, the standard Java hashtable-based implementation of maps that we'll use to store the name/number association.

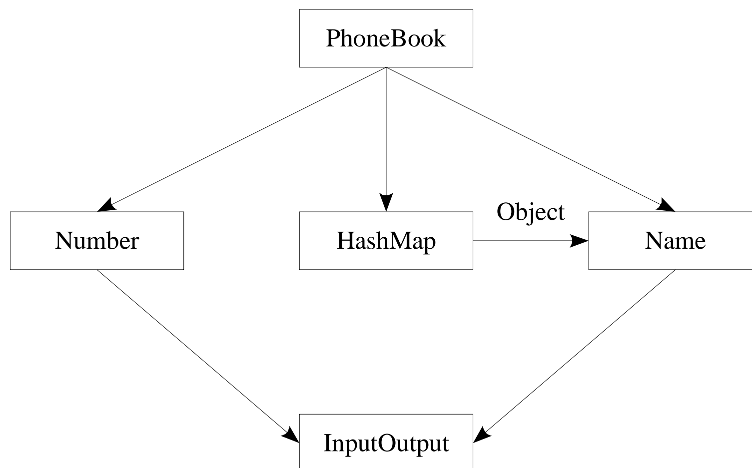
Here is an MDD of a reasonable design:



The main class PhoneBook uses all the other classes, not surprisingly. Name and Number are not dependent on anything; they're standalone data abstractions that can be reused. HashMap has a dependence on Name through the Object contract, because if Name's equals and hashCode methods don't obey the specification given in Object, the hashing mechanism may fail. Note that HashMap is in fact standalone, even though it depends on Name, and it certainly wasn't designed with Name in mind! The assumption labelled Object tells us it can be reused in any program that provides a class satisfying the Object contract - not a very arduous demand.

There is no dependence of HashMap on Number, because the HashMap calls methods only on its keys and not its values. Note that only the dependence of HashMap on Name is labelled. All the other dependences have assumptions that are the specifications of their target classes. PhoneBook's dependence on Name, for example, can assume all the properties of the Name class.

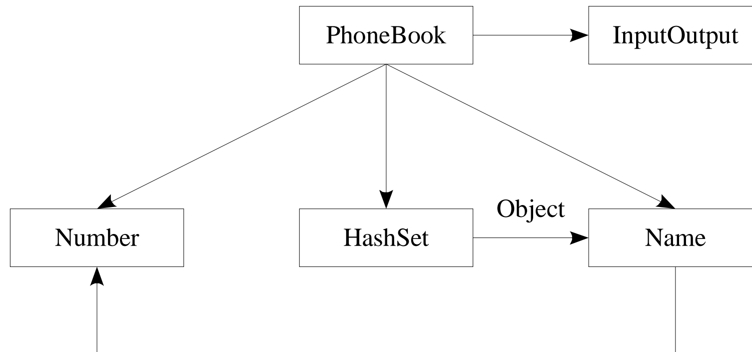
Here are examples of variant designs that are less attractive. The first embeds InputOutput functionality inside the Name and Number types. Name, for example, may have a method called something like getNameFromConsole that reads characters at the console and returns an abstract Name object:



This is a bad design, because Name and Number can no longer be reused independently of InputOutput. It is also likely to create a different kind of dependency which our MDD does not represent, called a sharing constraint. We are likely to want Name and Number to prompt for input

and handle input errors in a consistent manner. If we change how it's done in one class, we'll need to change the other. This isn't a dependence of the sort we've been describing, since neither class uses the other, but it is a dependence nonetheless. In the original design, decisions about how to prompt for input and handle input errors are localized within a single module `PhoneBook`.

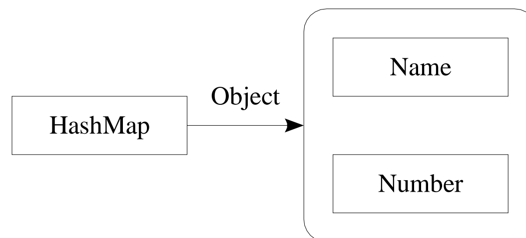
The second bad design uses a `HashSet` instead of a `HashMap` to allow `Name` objects to be looked up, and associates `Name`s and `Number`s directly by putting a number field inside the `Name` class:



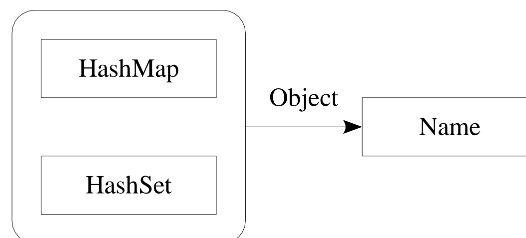
This is bad because it hardwires the relationship between the `Name` and `Number` types inside `Name`. We can't reuse one without the other. And suppose we want to implement different relationships: to allow several numbers to be associated with a single name, or to allow reverse lookup from numbers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

16.8 Grouping

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



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



You can group modules with contours in any way you like, even partially overlapping them (although it can get incomprehensible if you have too many overlappings). And you can draw an arc from a contour to a contour. The meaning of the diagram is always clear: you just replace each arc by a

set of arcs, so that P gets an A-use to Q if P is within a contour c and Q is within a contour c', and there's an arc marked A from c to c'.

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

16.9 Assumption Annotations

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

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

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

This means that when you draw an MDD you have the freedom to omit certain bits of information, increasing the set of dependences described:

- ▷ You can omit the assumption label on a dependency edge. In this case, the dependence is assumed to be on the full specification of the used class.
- ▷ You can use a class or interface name without specifying which methods are relevant; this implies that potentially all methods may be used.

▷ You can draw a dependence arc to or from a contour when in fact it should go to or from a particular subset of the boxes within the contour, thus adding dependences between modules that may not actually have dependences.

16.10 Name Dependences

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

16.11 Summary

So far, we've built up the basic ground rules of MDDs. Next, we will put our new tools to use in finding and reducing coupling in common scenarios.

Lecture 17: Dependencies II

17.1 Context

What you'll learn: How to draw module dependency diagrams (MDDs) for a program; how to draw MDDs for families of programs.

Why you should learn this: MDDs let you see at a glance crucial properties of a program, and spot where dependencies can be removed, weakened, or reorganized based on which parts of your system are most likely to change.

What I assume you already know: Familiar with the basic MDD notation from the previous lecture.

17.2 Example: Timer Callbacks

Last day we saw the basic notation for module dependency diagrams (MDDs). These diagrams are very useful during design to clarify your thoughts. But it may be difficult for you to see the relationship between these abstract diagrams and concrete programs until you have a little practice. So in this section, we will look at an example program and an MDD for it side by side, and see how changes in one are reflected in the other.

Suppose we want to write a program which offers the hard-working user occasional warnings about the perils of Repetitive Strain Injury, and encourages them to take a break. We might start with a class for displaying such messages and offering suggestions for useful exercises:

```
public class TimeToStretch {
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }

    public void suggestExercise() {
        //...
    }
}
```

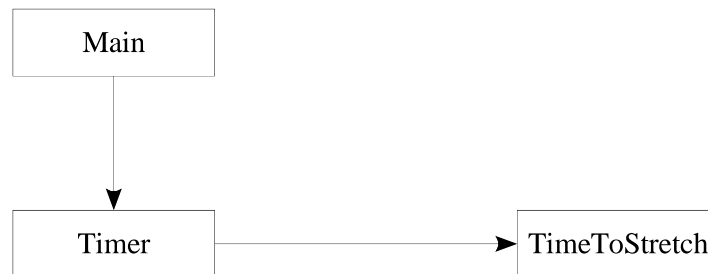
We would like the `run` method of `TimeToStretch` to execute from time to time. Here is a simple `Timer` class to do that, using a never-ending "busy-loop" that repeats and calls `run` whenever the time is right (this is a very inefficient use of CPU cycles, but lets us postpone a discussion of multi-threaded operation for another day).

```
public class Timer {
    private final TimeToStretch tts = new TimeToStretch();
    public void start() {
        while (true) {
            //...
            if (enoughTimeHasPassed) {
                tts.run();
            }
            //...
        }
    }
}
```

Finally, we create a `Timer` instance and start it running from some `Main` class:

```
Timer t = new Timer();
t.start();
```

Looking at the dependencies between our classes as we have written them, the `Main` class will depend on `Timer`, and `Timer` in turn depends on `TimeToStretch`. Here is a simple MDD of this:



This organization is not very satisfying. The `Timer` class is a prime candidate for reuse – there are lots of applications for making an event occur from time to time – but as we have designed it, `Timer` depends on the very application-specific `TimeToStretch` class. Ideally we would like to minimize this dependence. And in fact, the `Timer` doesn't really need to know much about `TimeToStretch`, other than the fact that it has a `run` method. We could abstract out this functionality into an interface or an abstract class:

```
public abstract class TimerTask {
    public abstract void run();
}
```

and then change `TimeToStretch` to be a subclass of `TimerTask`:

```
public class TimeToStretch extends TimerTask {
    //...
}
```

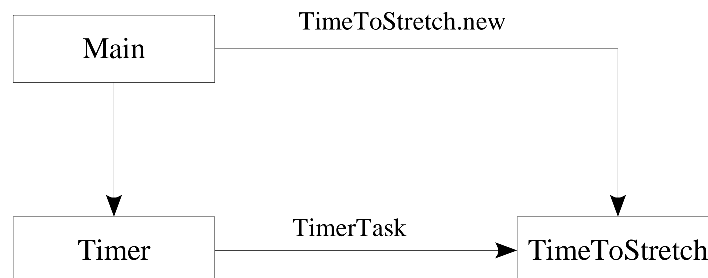
With those changes, we can rewrite the `Timer` class so that it only depends on `TimeToStretch` through `TimerTask`:

```java
public class Timer {
    private final TimerTask task;
    public Timer(TimerTask task) {
        this.task = task;
    }
    public void start() {
        //...
        task.run();
        //...
    }
}
```

I've chosen to move responsibility for creating a `TimeToStretch` instance outside of the `Timer` class:

```java
Timer t = new Timer(new TimeToStretch());
t.start();
```

With these changes, our MDD becomes:



Now `Timer` is reusable. It can be used in any situation where some other module is willing to take on the `TimerTask` role, which is not very onerous.

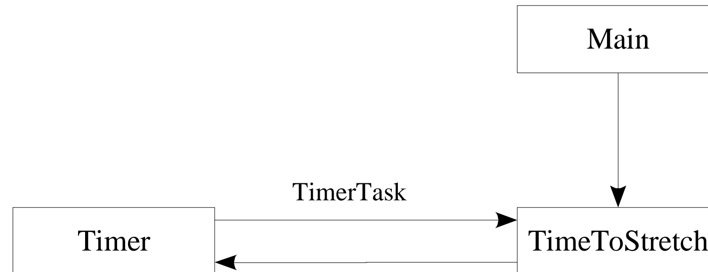
A further change we can make is to realize that `Main` doesn't really need to know about the `Timer` class. We can let `TimeToStretch` be responsible for its own `Timer`, which would make a lot of sense in a more sophisticated implementation; `TimeToStretch` might want to control the frequency at which `run` is called, start/stop the timer, etc.

```java
public class TimeToStretch extends TimerTask {
    private final Timer timer;
    public TimeToStretch() {
        timer = new Timer(this);
    }
    public void start() {
        timer.start();
    }
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
    //...
}
```

Now we start things going from our `Main` class with:

```
TimeToStretch tts = new TimeToStretch();
tts.start();
```

And now our MDD becomes:

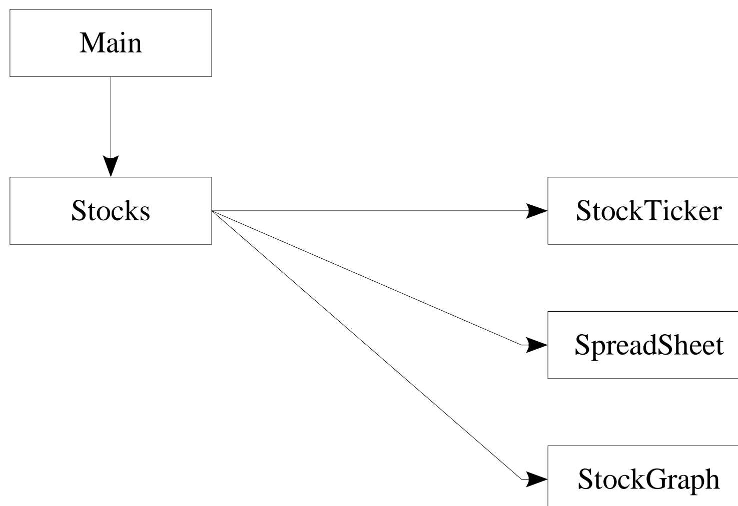


Notice that we have reversed the direction of the dependence between `Timer` and `TimeToStretch`, with only a minimal *callback* dependence in the original direction. This is valuable because it allows us to reuse the `Timer` class.

As a side note, Java does in fact have `Timer/TimerTask` classes which are very similar to what we have developed here. The implementation is of course quite different and vastly more efficient, but the principle is basically the same.

17.3 Example: the Observer Pattern

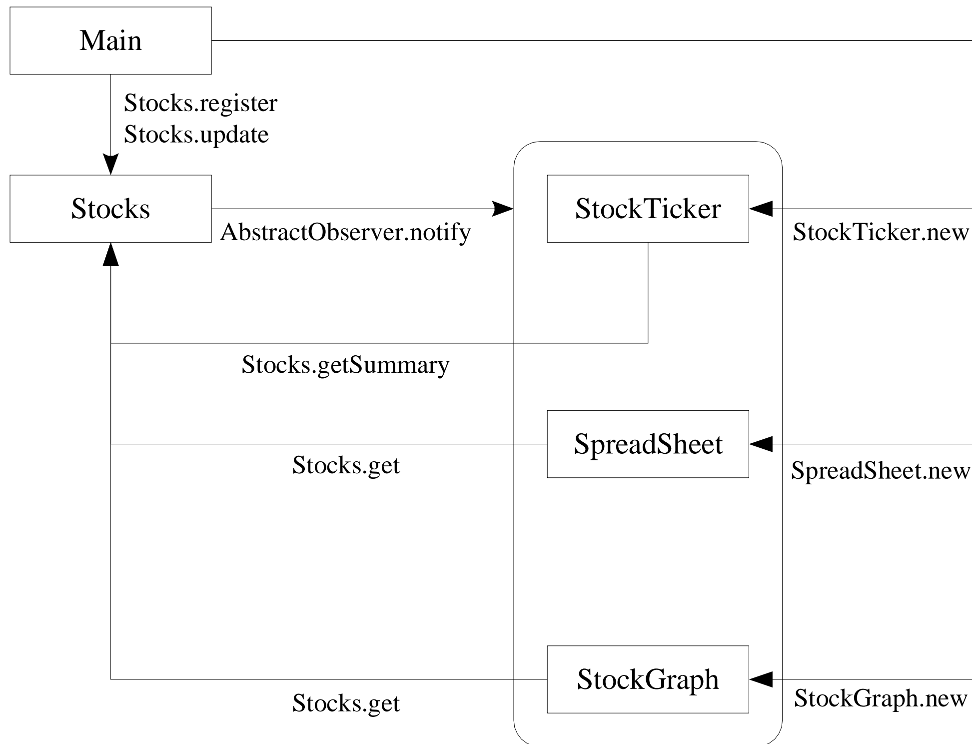
Suppose we have an application which maintains a database of information about stocks in a class called `Stocks`. Users can view this information within the application in many ways, such as with graphs, spreadsheets, stock tickers, etc. As stocks change, all these views need to be updated. In a simple implementation, our `Stocks` class would be responsible for making those updates, since it is the only module that knows about the changes:



This is a dissatisfying design, since every time we add or change the design of a viewer, we need to fix `Stocks`. We would like to insulate that class from the vagaries of the viewers. We can do that just like we did with `Timer`. We create an interface or base class for all our viewers (let's call it `AbstractObserver`) which just has a `notify` method. This is all `Stocks` need assume the viewers

support. When a stock changes, `Stocks` calls `notify` on all the viewers – and *they* are responsible for pulling the information they need from `Stocks`.

Now rather than `Stocks` depending on the viewers, the viewers depend on `Stocks` – this means that we can add or change viewers without touching `Stocks`. Here is the MDD:



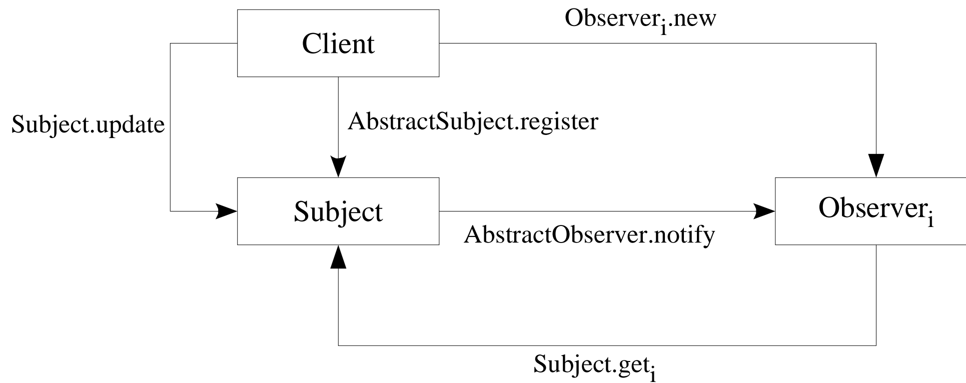
In the MDD, just as in the `Timer` example, we have addressed the issue of who creates the viewers, since `Stocks` no longer can. We give `Main` this responsibility. `Main` lets `Stocks` know about viewers through a `register` method, and changes stocks using an `update` method. In the diagram, we illustrate that the viewers might use different observers (`get`, `getSummary`) to get different information from the `Stocks` class – for example, a `StockTicker` might be watching a particular stock rather than monitoring all changes like a `SpreadSheet` would.

The reorganization we just did has a name; it is called the *observer* pattern. We will talk about it again next week. For now, let's look at how we could draw a *general* diagram for the pattern.

17.4 Families and Patterns

Sometimes we will want to draw an MDD that describes not one particular program, but a family of programs. This will let us develop a toolbox of good ideas that we can carry with us from project to project. The lectures next week will be devoted to a large number of these good ideas, which we call *design patterns.* MDDs will be very useful for describing and applying these patterns, but we need to generalize our notation a little first.

As a convenient shorthand, we'll use subscripts on module names and assumptions to indicate templates that can be replicated; each separate subscript can replicated independently. With this shorthand, an MDD for the observer pattern looks like this:



The subscript i indicates that there may be several distinct concrete `Observer` classes. The `Client` class has specific constructor dependences on these. And each of these `Observer` classes may call a different `get` method of `Subject`.

`Subject` plays the role that `Stocks` played in the previous section – it encapsulates the data that is being observed, and is responsible for notifying the `Observer`s when the data changes. We have abstracted out to an `AbstractSubject` assumption for the `register` method, just as we did for the `notify` method of `AbstractObserver`.

17.5 Coupling Due to Shared Constraints

There's a different kind of coupling which isn't shown in a module dependency diagram. Two modules 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 modules, *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 modules agree on the file format. If the file format is changed, both modules will need to change.

To avoid this kind of coupling, you have to try to localize functionality associated with any constraint in a single module. 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 modules. You start by listing the key design decisions (such as choice of file format), and then assign each to a module 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.

Lecture 18: Design Patterns I

18.1 Context

What you'll learn: This week we will study a collection of *design patterns* which are standard solutions to common problems. Today we look at *creational* design patterns used to minimize coupling introduced by creating objects.

Why you should learn this: Design patterns are distilled nuggets of programmer wisdom, debugged and documented sometimes over decades. Familiarity with these patterns will save you time.

What I assume you already know: How to read and write module dependency diagrams.

18.2 Design patterns

When we first design a system, it will often have some serious problems that we didn't anticipate. After careful, painstaking redesign, we come up with something better. Design patterns describe those "better" solutions that have been rediscovered time and time again over the years. Studying design patterns helps you see a few steps ahead in the game of design, foresee non-obvious problems, and make wiser choices. 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 model or module dependency diagram

To help clarify these diverse ideas, let us look at some patterns you have already seen.

18.3 Familiar patterns

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.

18.4 Categories of Design Pattern

Perhaps the best way to understand design patterns is as strategies for anticipating redesign. Our first pass at designing a system probably won't achieve everything we would like. Design patterns can help us anticipate and avoid common causes of redesign, such as:

- **Creating an object of a specific class**. This commits us to a specific *implementation* of the object; we would rather just commit to an *interface* the object should meet. We would

often like to preserve the ability to substitute objects of other classes that meet the required interface. We therefore need to create objects *indirectly* without naming a specific class.

- **Executing a specific operation**. It is often useful to place indirection between a request for an operation and the performance of that operation, so that we can later change how that request is satisfied. (Polymorphism is an example of this!)
- **Using platform-specific interfaces**. Different hardware and software platforms offer somewhat different functionality through somewhat different interfaces. Even a single platform can change from version to version. It is useful to limit your systems dependencies on such vagaries.
- **Relying on object representation or implementation**. Hiding such information insulates users of an object from changes in the representation and implementation, and prevents a cascade of changes.

Design patterns seek to reduce coupling and retain flexibility in our design. *Creational* patterns seek to minimize inflexibility introduced by object creation. *Structural* patterns minimize coupling introduced when classes or objects are assembled to form larger structures *Behavioral* patterns minimize coupling introduced when classes or objects cooperate to perform operations and distributed tasks. Today we will look at a selection of helpful creational patterns.

18.5 Factory Methods

Sometimes we would prefer to have more flexibility in the objects we create than constructors allow. In particular, constructors in Java have two major limitations:

▷ A constructor can only return an instance of its associated class, never a subtype – even though a subtype would be acceptable, by definition.
▷ A constructor must always return a new object, never a previously allocated object. This can be annoying – for example, there is really no need to have multiple instances of equal immutable objects.

Factory methods let us skirt around these limitations. A factory method is simply a method we call to create an object for us. For example, the Java class `DateFormat` has several factory methods, such as `getDateInstance`:

```
DateFormat df = DateFormat.getDateInstance();
Date today = new Date();
System.out.println(df.format(today)); // "Oct 25, 2004"
```

Here the call to `getDateInstance` returns an object we can use to format dates into human-readable form, based on where in the world the application is running. One immediate benefit of factory methods is that they can be more readable than constructors. For example, `DateFormat` objects can be created to format dates, times, or both dates and times. With constructors, we would need to pass in a flag, whereas with factory methods, we can do the same, or simply give a sensible name to the method:

```
DateFormat tf = DateFormat.getTimeInstance();
System.out.println(tf.format(today)); // "12:16:14AM"
```

This is already very useful (think about constuctors and factory methods for a `Point` class that can be constructed using two `floats` representing either a cartesian or polar coordinate). Another benefit is that a factory method is free to return a *subtype* of the requested type. And in fact, in the examples above, the factory methods will return an instance of a class `SimpleDateFormat` which extends `DateFormat` (which is actually an abstract class). We could have created `SimpleDateFormats` directly using its constructor, but if you read the specification for this class, all its constructors have the following note:

> Note: This constructor may not support all locales. For full coverage, use the factory methods in the `DateFormat` class.

In other words, the designers want the freedom of introducing different subtypes of `DateFormat` for countries with particularly odd date/time formats. From the client's point of view, this is nice. For example, consider client code that requests versions of `DateFormat` specialized for different locations:

```
DateFormat df1 = DateFormat.getDateInstance(DateFormat.FULL,
                 Locale.FRANCE);
DateFormat df2 = DateFormat.getDateInstance(DateFormat.SHORT,
                 Locale.JAPAN);
System.out.println(df1.format(today)); // "lundi 25 octobre 2004"
System.out.println(df2.format(today)); // "04/10/25"
```

The client doesn't need to know what subclasses are being used to manage the different counties. At the time of writing, `SimpleDateFormat` is used for both, but it is good for the designer and good for us that we don't have to depend on that.

18.6 Factory Methods Allow Reuse

Factory methods can also overcome the *second weakness of Java constructors*: Java constructors always return a new object, and can never reuse existing objects. Factory methods don't have that limitation. Consider the simple immutable class `Boolean`, whose implementation might look like this:

```
public class Boolean {
    private final boolean value;
    public Boolean(boolean b) {
        value = b;
    }
    public static final Boolean FALSE = new Boolean(false);
    public static final Boolean TRUE = new Boolean(true);
    // Convert primitive boolean to Boolean
    public static Boolean valueOf(boolean b) {
        return (b ? Boolean.TRUE : Boolean.FALSE);
    }
    //...
}
```

The `valueOf` method is a factory method. It is specified to return either `Boolean.TRUE` or `Boolean.FALSE`, two constants provided by the class. Why is such a method useful? As of Java 1.4, the constructor for `Boolean` has the following text in bold:

> Note: It is rarely appropriate to use this constructor. Unless a *new* instance is required, the static factory `valueOf(boolean)` is generally a better choice. It is likely to yield significantly better space and time performance.

What does this mean? Calling the constructor to make a `Boolean` makes a fresh object; calling `valueOf` reuses an existing object. Since immutable objects can be shared freely, the latter is preferable.

One common approach in such situations is to make the constructor private. In that case, `Boolean.TRUE` and `Boolean.FALSE` would be the *only* `Boolean`s that could ever be created. This idea is a pattern in itself, called *type-safe enumeration*, where we make a class with a finite set of instances, with no way other instances could be created. If you declared a method with a parameter of type `Boolean`, and if `Boolean`'s constructor were private, then you would be guaranteed that anything passed to your method would be either the `Boolean.TRUE` or `Boolean.FALSE` object. Of course, `Boolean`'s constructor is not private – here is a fragment of an interview with Joshua Bloch on this subject:

> For example, the Boolean type, which is a boxed primitive boolean, simply should not have had public constructors. It is basically a type-safe enum, and it should have been one. There should only be two Boolean objects in a VM at any time, one true and one false. There's really no great advantage to allow multiple trues or multiple falses, and I've seen programs that produce millions of trues and millions of falses, creating needless work for the garbage collector. So, in the case of immutables, I think factory methods are great.
>
> (*Josh Bloch on Design*, JavaWorld, January 4, 2004)

18.7 Singleton

The singleton pattern is a special case of a factory method, and is used to guarantee that only *one* instance of a particular class ever exists. This is useful for classes representing unique physical resources, such as the display. For example, to ensure that there can only ever be one `Hero`, you could use a private constructor with a static factory:

```
public class Hero {
    private static final Hero INSTANCE = new Hero();
    private Hero() {
        // we make this private so nobody outside Hero can create an instance
    }
    public static Hero getInstance() {
        return INSTANCE;
    }
    //...
}
```

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. The reason that we make our singleton instance private is so that, if we later find we need to, we could create multiple instances – for example, one per thread.

18.8 Interning

The interning design pattern reuses existing objects rather than creating new ones. If a client requests an object that is equal to one that already exists, then the pre-existing one is returned instead. Interning is generally useful only for immutable objects. For example, an image might be represented as an array of pixels, each of which is a `Color`. A large image may have many pixels in it, but only a few distinct `Colors`. Representing every pixel with a unique `Color` object is unnecessarily wasteful of space. A better way would be to share the `Colors`.

Interning arranges for objects that are immutable to be reused. Interning requires a table of all the objects (of the given type) that have ever been created. If the table contains an object that is equivalent to the desired object, the table's object (the interned object) is returned instead. Otherwise, the desired object is created and stored in the table for future reference.

Here is a method that returns an interned `Color`, given its red, green, and blue values:

```
class ColorIntern {
    private static Map interned_colors = new HashMap();
    public static Color fromRGB(int r, int g, int b) {
        Color color = new Color(r,g,b);
        if (interned_colors.containsKey(color)) {
            return (Color) interned_colors.get(color);
        } else {
            interned_colors.put(color,color);
            return color;
        }
    }
}
```

Now if we use `fromRGB` as a factory method to make our `Colors`, we are guaranteed that equal colors will be the same identical object, with no duplicates:

```
Color red1 = ColorIntern.fromRGB(255,0,0);
Color red2 = ColorIntern.fromRGB(255,0,0);
Color green1 = ColorIntern.fromRGB(0,255,0);
Color green2 = ColorIntern.fromRGB(0,255,0);
System.out.println(red1==red2);     // true
System.out.println(green1==green2); // true
System.out.println(red1==green1);   // false
```

This method creates a trial `Color` object before checking whether the color has already been interned. When the interned objects are expensive to create, however, it may make more sense for the interning table to map the content of the object (in this case, its RGB value) to the interned object, in order to save the cost of construction when the object already exists. Note that this code uses a `Map` rather than a `Set`, even though all we're really doing is storing a set of interned Colors. The reason is that sets do not have a `get` operation, only `contains`. In other words, we'd

be able to test whether the set of interned objects contains the color we want, but we'd be unable to actually obtain a reference to the interned object.

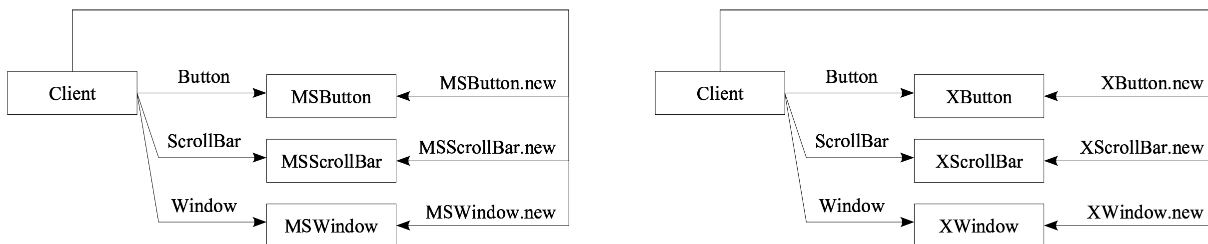
Interning for strings is so important that it is built into Java. `String.intern` returns an interned version of a string, and is called automatically on all literal `String`s to reduce memory requirements.

```
String s1 = new String("foo");
String s2 = new String("foo");
System.out.println(s1==s2); // false
System.out.println(s1.intern()==s2.intern()); // true
```

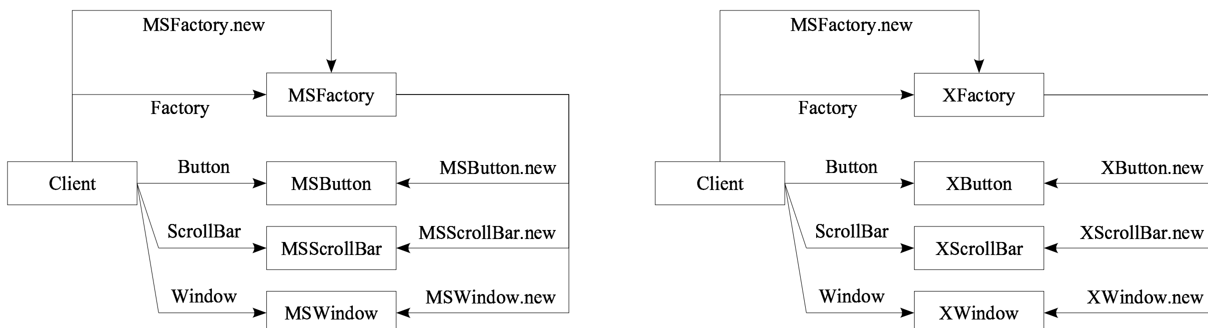
18.9 Abstract Factory

Sometimes we have a whole family of classes we wish to be able to interchange easily. Perhaps a client uses a set of graphical widgets which need to be changed to be appropriate for the platform the application runs on. For example, the client may depend on one set of widgets in Microsoft Windows, and another under X:

If the client is dependent on platform-specific characteristics of the widgets, then moving from one platform to another will be painful. We can define generic interfaces for the widgets, such as `Button` and `ScrollBar`. Now the client no longer needs to know the concrete classes of the widgets – except when creating them:



This is better, but it is unsatisiying that widget creation (which could be scattered throughout client code) still requires the client to refer to platform-specific classes. So we go one step further, and give responsibility for creating the right kinds of widgets to a *factory class*:



Now the only platform-dependent code in the client is the initial creation of the right kind of factory. And we could even remove that if we wished.

```
XWindow window = new XWindow(100,100,200,200);
window.add(new XButton("OK"));
window.add(new XScrollBar());
```

Gets replaced with:

```
WidgetFactory factory = WidgetFactory.getFactory();
Window window = factory.createWindow(100,100,200,200);
window.add(factory.createButton("OK"));
window.add(factory.createScrollBar());
```

18.10 Flyweight

The flyweight pattern is a generalization of interning. Interning is applicable only when an object is completely immutable and all of its state can be shared. The more general flyweight pattern can be used when most (but not necessarily all) of an object is immutable, or when instances of the object share most but not all of their state. Suppose you are building a viewer for matrices. You wish to be able to view the content of a matrix, and apply formatting to rows, columns, or individual cells. Let's do this by creating a `CellWindow` class which knows how to draw an individual cell:

```
class CellWindow {
    // each CellWindow corresponds to one cell in a Matrix
    Matrix matrix;
    int row, col;
    // each CellWindow can have its own style
    Font font;
    Color color;
    Alignment alignment;
    // each CellWindow corresponds to a region within an enclosing window
    Rectangle boundary;
    // each CellWindow knows how to draw itself
    void draw(Graphics g) {
        g.setFont(font); // etc...
        g.drawString(matrix.get(row,col).toString(),
                     boundary.x, boundary.y);
    }
}
```

Generally, large numbers of cells will have a great deal in common: the matrix they belong to, and their formatting (whole columns or rows are likely to be generally the same). Ideally, we would like to use interning to share this state. But we can't intern `CellWindows` as they are, because they also contain state information that is less likely to be common – the indices of the cell they show, and the rectangular region of the screen allocated for them. We can improve the situation by separating out the state that can be shared so we can intern that part:

```java
// CellShare represents properties that are often shared by many CellWindows
class CellShare {
    Matrix matrix;
    Font font;
    Color color;
    Alignment alignment;
}
```

And now our `CellWindow` is made up of the sharable part `CellShare` along with the remaining state which it doesn't make sense to try and share:

```java
class CellWindow {
    CellShare share; // intern state that can be shared
    int row, col;
    Rectangle boundary;
    void draw(Graphics g) {
        g.setFont(share.font); // etc...
        g.drawString(share.matrix.get(row,col).toString(),
                     boundary.x, boundary.y);
    }
}
```

Sometimes we can even remove this remaining unshared state. This is possible if we can recover the state from the object's context when needed. This is called *extrinsic* state. For example, the `boundary` and the `row` and `col` could be be passed in as arguments to `draw`:

```java
class CellWindow {
    Font font;
    Color color;
    Alignment alignment;
    void draw(Graphics g, Matrix matrix, int row, int col,
              Rectangle boundary) {
        g.setFont(font);  // etc...
        g.drawString(matrix.get(row,col).toString(),
                     boundary.x, boundary.y);
    }
}
```

This places more work on the class drawing the matrix. Instead of deciding the boundary and associating a `CellWindow` with a particular matrix cell at the time of its construction, this must be recomputed when drawing. In practice, this could be trivial if the `CellWindow`s are stored in an array in a class responsible for calling them to be redrawn when needed.

The main disadvantage of the flyweight pattern is that a reference to a `CellWindow` is no longer sufficient to do everything you might want to do with it, such as draw it. You have to know its context as well. Often, however, flyweight can be used "under the covers" to improve the performance of a system, while from a client's point of view objects are still heavyweight.

Flyweight should only be considered after profiling has determined that space usage is a critical bottleneck in the program. Introducing such constructs into programs complicates them and presents many opportunities for error. It should be undertaken only in limited circumstances.

18.11 Prototype pattern

Suppose we are designing a toolbar for a drawing tool, and we wish to have buttons for creating various kinds of shapes: rectangles, squares, rectangles with rounded corners, squares with rounded corners, ovals, circles, and either filled or unfilled versions of all of these. We could create a basic shape type:

```
public abstract class Shape {
    public abstract void draw(Graphics g);
}
```

Then we could subclass this for each of the many kinds of shapes we would like to make, and then create a set of buttons that construct instances of those classes. This would work, but is very tedious. An alternative is the *prototype* pattern, where we create instances of the objects we want, and then request them to copy themselves when a client needs an instance. We add a method for making this copy to our interface or base class:

```
public abstract class Shape {
    public abstract void draw(Graphics g);
    public abstract Shape copyShape();
}
```

For example we might class for drawing various kinds of rectangles class like this:

```
class RectShape extends Shape {
    private final boolean filled, square, rounded;
    public RectShape(boolean filled,
                     boolean square,
                     boolean rounded) {
        this.filled = filled;
        this.square = square;
        this.rounded = rounded;
    }
    public void draw(Graphics g) {
        // draw an appropriate rectangle
    }
    public Shape copyShape() {
        return new RectShape(filled,square,rounded);
    }
    //...
}
```

Then we can set up our toolbar by simply adding different prototype instances to it, rather than having to make many different subclasses:

```
toolbar.add(new RectShape(false,false,false));
toolbar.add(new RectShape(false,false,true));
toolbar.add(new RectShape(false,true,false));
toolbar.add(new RectShape(false,true,true));
```

```
toolbar.add(new RectShape(true,false,false));
toolbar.add(new RectShape(true,false,true));
toolbar.add(new RectShape(true,true,false));
toolbar.add(new RectShape(true,true,true));
```

In passing, the arguments to `RectShape`'s constructor are quite confusing – it would be easy to mix up the order of filled versus square versus rounded. This is a prime candidate for using type-safe enumerations, which are safer and easier to read.

Java has a `Cloneable` interface that specifies a `clone` method which many use for making object copies; if you plan to do that, beware – there are many subtleties that can bite you. See the Bloch book for details.

Lecture 19: Design Patterns II

19.1 Context

What you'll learn: Today we look at structural patterns, which seek to minimize coupling when classes or objects are assembled to form larger structures.

Why you should learn this: The patterns you will see today are very widely used because they are simple yet very powerful; they should become second nature to you.

What I assume you already know: The general goals and intentions behind design patterns, as discussed in the previous lecture.

19.2 The Wrapper Pattern

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:

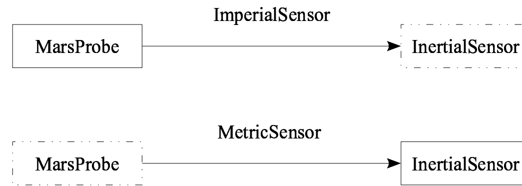
Pattern	Functionality	Interface
Adapter	same	different
Decorator	different	same
Proxy	same	same

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. It is easiest to see the difference between these varieties of wrappers by looking at concrete examples.

19.3 Adapters

Suppose we are working as part of a team to build a Mars probe that is equipped with a sensor to track its position and speed. The team is split into two parts; one group builds the probe (and associated software), while another team builds the sensor (and its software).

Through an unfortunate misunderstanding, the group building the sensor designs it to report metric units, while the group building the probe expects it to report imperial units:



Naturally, NASA catches the problem early in testing. What can they do? In situations like this, adapters are often a very practical solution to make a class "fit" somewhere when it has the right basic functionality but doesn't exactly have the interface we need. Here is the interface the probe team coded to:

```
public interface ImperialSensor {
    // returns: distance traveled in miles
    public double getDistance();
    // returns: speed in miles per hour
    public double getSpeed();
}
```

But what they've actually got is this:

```
public class InertialSensor {
    // returns: distance traveled in meters
    public double getDistance() { ... }
    // returns: speed in meters per second
    public double getSpeed() { ... }
    //...
}
```

The probe team could try to *adapt* this class by extending it:

```
public class AdaptedSensor extends InertialSensor {
    // returns: distance traveled in miles
    public double getDistance() {
        return super.getDistance() * MILES_PER_METER;
    }
    // returns: speed in miles per hour
    public double getSpeed() {
        return super.getSpeed() * MILES_PER_METER * HOURS_PER_SECOND;
    }
}
```

This approach is generally fine in many cases, for example to add some trivial observers. In this case, though, it is dangerous, since `AdaptedSensor` is *not a true subtype* of `InertialSensor`: it doesn't meet the specification for that class since it has changed units. The probe team would have to be careful not to use any code (say from a third party) that manipulates `InertialSensors`, since as far as Java is concerned `AdaptedSensors` can be freely passed to methods expecting `InertialSensors`, but the results may be nonsense.

A better alternative is to use composition and delegation, as we have seen several times before in other lectures:

```
public class AdaptedSensor implements ImperialSensor {
    public final InertialSensor is;
    public AdaptedSensor(InertialSensor is) {
        this.is = is;
    }
    // returns: distance traveled in miles
    public double getDistance() {
        return is.getDistance() * MILES_PER_METER;
    }
    // returns: speed in miles per hour
    public double getSpeed() {
        return is.getSpeed() * MILES_PER_METER * HOURS_PER_SECOND;
    }
}
```

Now we don't have any assumption that our `AdaptedSensor` is a subtype of `InertialSensor`. The MDD for this scenario is very typical, with the wrapper sitting between a client and a used class, using that class's functionality but transforming it somewhat:

Adapters come in very handy when we try to integrate our own code with externally developed libraries. For example, suppose we have been building a system for drawing shapes based on the following interface:

```
public interface ScaleShape {
    void draw(Graphics g);
    void scale(float factor);
    Rectangle bounds();
    //...
}
```

We find that we wish to use classes from a library that does something close, but not exactly matching, what we do:

```
public interface SizeShape {
    void draw(Graphics g);
    void setWidth(float width);
    void setHeight(float height);
    int getWidth();
    int getHeight();
    Point getTopLeft();
    Point getBottomRight();
    //...
}
```

Shapes in the library have a different interface; they can have their width and height independently varied, for example, while we only really need control of overall scale. We can write an adapter using composition to take instances of classes from the library and wrap them to look more familiar, with a `scale` method:

```
class ShapeAdapter implements ScaleShape {
    final SizeShape sizeShape;
    public ShapeAdapter(SizeShape shape) {
        sizeShape = shape;
    }
    public void scale(float factor) {
        sizeShape.setWidth(sizeShape.getWidth()*factor);
        sizeShape.setHeight(sizeShape.getHeight()*factor);
    }
    public void draw(Graphics g) {
        sizeShape.draw(g);
    }
    public Rectangle bounds() {
        Point topLeft = sizeShape.getTopLeft();
        Point bottomRight = sizeShape.getBottomRight();
        return new Rectangle(topLeft.x, topLeft.y,
                             bottomRight.x, bottomRight.y);
    }
    //...
}
```

It would be perfectly fine to do this adaptation using subclassing, but we would have to subclass each class of interest separately – the composition approach gives us a wrapper that will work on *any* class that meets the `SizeShape` interface.

19.4 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 can "fit in" to existing uses of the undecorated object without modification, 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. A very direct example of decoration is adding borders to the appearance of existing shapes:

```
public class BorderDecorator implements ScaleShape {
    final private ScaleShape shape;
    public BorderDecorator(ScaleShape shape) {
        this.shape = shape;
    }
    public void draw(Graphics g) {
        shape.draw(g);
        g.drawRectangle(shape.bounds());
    }
    // remember to delegate all other methods to shape
}
```

A “decorated” shape can be used anywhere an undecorated shape normally can, but its visual appearance has now been extended. It is quite natural to implement decoration using subclassing, but we have chosen composition here because it allows us to decorate arbitrary shape objects, not just a particular class we might choose to extend.

19.5 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.

For example, suppose we are building a word processor which can work with very large documents containing many images, charts, etc. Every image is represented as an instance of a class `EmbeddedImage`:

```
public class EmbeddedImage implements Drawable {
    public EmbeddedImage(String filename) {
        // loads image from file
    }
    public void draw(Graphics g) {
        // renders image
    }
    //...
}
```

As a page is loading, `EmbeddedImage` instances are created and told where to load their image from. This design may work well with small documents, but for large documents it has a problem – the

user has to wait for all the images to load before viewing the document, even if most of the images won't even be visible initially on the page they see.

Rather than trying to change how page loading happens, we can introduce a *proxy* between the document and each image it contains. This proxy will delay loading of the image until it is actually needed:

```
class EmbeddedImageProxy implements Drawable {
    private EmbeddedImage image;
    private final String filename;
    public EmbeddedImageProxy(String filename)  {
        this.filename = filename;
        image = null;
    }
    private void load() {
        if (image==null)  {
            image = new EmbeddedImage(filename);
        }
    }
    public void draw(Graphics g) {
        load();
        image.draw(g);
    }
    //...
}
```

Once you start playing this kind of game, all sorts of things become possible. For example, our solution now is better, but it might be better still to have images load in the background while the user is reading the document, for faster response. We could easily add this feature in the proxy without complicating the document loading code at all.

Lecture 20: Design Patterns III

20.1 Context

What you'll learn: Today we finish up our discussion of design patterns by looking at *behavioral* patterns, which seek to minimize coupling introduced when objects cooperate to perform operations and distributed tasks.

Why you should learn this: The flow of communication between objects at run-time can be complex to design and understand. Behavioral patterns embody a kernel of simplicity that we can focus on to regulate the confusion.

What I assume you already know: The general goals and intentions behind design patterns, and the creational and structural patterns discussed in previous lectures.

20.2 The Observer Pattern

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
G. Bates	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:

```
public 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 Gill Bates has demanded a regrade on problem set 1, and that regrade did reveal grading errors: Gill'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:

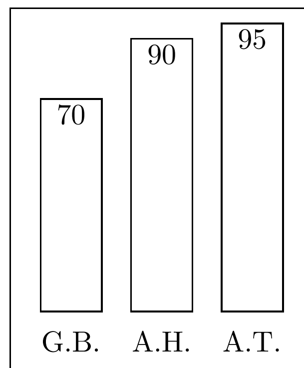
```
SpreadsheetView ssv = new SpreadsheetView();
//...
ssv.update("6.170", "G. Bates", "PS1", 30);
```

(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
G. Bates	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", "G. Bates", "PS1", 30);
bgv.update("6.170", "G. Bates", "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.

```
ArrayList observers = new ArrayList();
//...
for (int i=0; i<observers.size(); i++) {
    GradeDBViewer v = (GradeDBViewer) observers.get(i);
    v.update("6.170", "G. Bates", "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.

The Java library has some basic support for observers, using the abstract class `Observer` and the interface `Observable`. Suppose we have a sign-up sheet for 6.170 students:

```
public class SignupSheet extends Observable {
    private List students = new ArrayList();
    public void addStudent(String student) {
        students.add(student);
        setChanged();
        notifyObservers();
    }
    public int size() {
        return students.size();
    }
}
```

We can create an observer for that sign-up sheet by implementing an `update` method:

```
public class SignupObserver implements Observer {
    public void update(Observable o, Object arg) {
        System.out.println("Signup count: " + ((SignupSheet)o).size());
    }
}
```

Now we can attach an observer (or many observers) to the sign-up sheet without having to change any of its code:

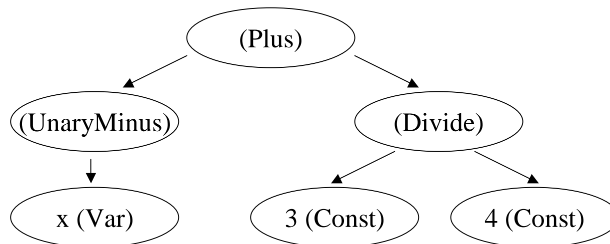
```
SignupSheet s = new SignupSheet();
s.addStudent("torvalds");
s.addObserver(new SignupObserver());
s.addStudent("gatesb");
// outputs: "Signup count: 2"
```

20.3 Traversing hierarchical structures

Iterators are a familiar design pattern that facilitate traversal through a linear collection. What about hierarchical collections? Object oriented software systems are full of hierarchies:

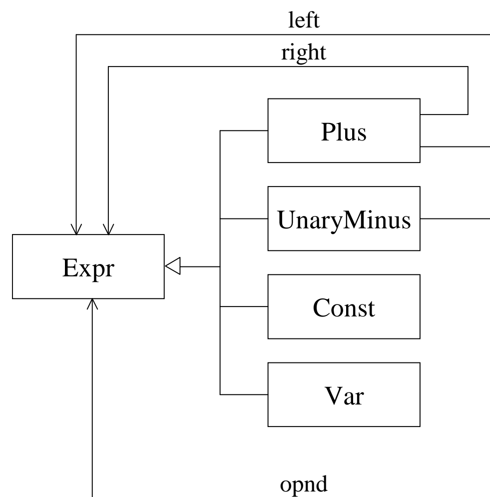
- ▷ a user interface is a hierarchical collection of UI objects: windows, panes, buttons, textboxes;
- ▷ a filesystem is a hierarchy of directories and files;
- ▷ an email client contains a hierarchy of mail accounts, folders, messages, and attachments.

Before we look at how to traverse these kinds of hierarchies, we should first consider how the hierarchy might be represented. Recall that the composite 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 a higher-level object with structure as opposed to a basic object. The same operations work on both, because both extend the same type. Consider the problem of representing expressions in a programming language. Syntactically, the expression $-x + 3/4$ can be represented by a hierarchy of objects called an abstract syntax tree:



The nodes at the bottom of the tree, instances of `Const` and `Var`, are the atomic units, or leaves. The internal nodes of the tree, corresponding to operators like `Plus` and `Divide`, are composites.

The key idea of the composite pattern is that composites and leaves all share a common interface, which in this case we'll call `Expr`. Here's part of the object model showing the relevant classes:



The shape of this object model tells you it's a composite pattern: a base class (`Expr`) with several subclasses, some of which are terminal (the leaves `Const` and `Var`), and some of which have references to subexpressions (the composites `Plus` and `UnaryMinus`). The code for these classes might look like this, if we only pay attention to the fields and ignore the methods:

```
interface Expr {
    // we will talk about what might be here later
}
class Plus implements Expr {
    Expr left;
    Expr right;
    //...
}
class UnaryMinus implements Expr {
    Expr opnd;
    //...
}
class Const implements Expr {
    double value;
    //...
}
class Var implements Expr {
    String name;
    //...
}
```

A complete representation would need other classes as well, to represent *Multiply*, *Divide*, *Subtract*, etc.

20.3.1 Interpreter

Now we're ready to talk about patterns for traversing composite structures, using `Expr` as an example. There are many operations we may want to perform on an expression. Suppose we want to evaluate the expression, given some execution context that assigns values to variables. Evaluation might be provided as a method of `Expr`:

```
interface Expr {
    double eval (Context context);
    //...
}
```

Each type of expression then implements `eval` in the appropriate manner. In particular, composites like `Plus` have to recursively evaluate their subexpressions:

```
class Plus implements Expr {
    Expr left;
    Expr right;
    double eval (Context context) {
        return left.eval (context) + right.eval (context);
    }
    //...
}
```

```
class Const implements Expr {
    double value;
    double eval (Context context) {
        return value;
    }
    //...
}

class Var implements Expr {
    String name;
    double eval (Context context) {
        return context.getVarValue (name);
    }
    //...
}
```

This technique is called the interpreter pattern. An operation that needs to traverse a composite hierarchy is declared as a method on the composite base class, in this case `Expr`. Composite nodes recursively call the operation on their parts, and the leaves terminate the recursion. In the interpreter pattern, both the operation and the means of traversal are bound up in the composite classes themselves.

20.3.2 Procedural Traversal

A number of operations might be reasonably implemented as interpreter methods on `Expr`: pretty-printing, type checking, and optimization come to mind. But a problem arises when a client wants to define a new operation. For example, suppose I want to know all the variable names that are used by an expression, perhaps so I can introduce a new variable and avoid conflicts with existing names. With the interpreter pattern, I would have to add a new method to `Expr` for my new operation. Then I'd have to implement the new method in all the subclasses of the composite. I'd need to change many classes to implement just one new operation. Furthermore, I'd need access to the source code of the expression classes, which isn't always possible. A first step towards fixing this problem is to represent the new operation as a class itself. (This is what the Liskov text calls the procedural approach to hierarchy traversal, although here we're using a class rather than a collection of static procedures.) The class has a separate method for each type of expression. Methods corresponding to composites take care of traversing the composite's children. Here's the code:

```
class FindVariables {
    Set vars = new HashSet ();
    void forExpr (Expr e) {
        // we will fill in this method's body shortly
    }
    void forPlus (Plus e) {
        forExpr (e.left ());
        forExpr (e.right ());
    }
    void forConst (Const e) { }
    void forVar (Var e) {
```

```
            vars.add (e.name ());
        }
        //...
    }
```

This class traverses the expression tree, and every time a `Var` node is encountered, its variable name is added to a set. For convenience of other clients, it's probably best to hide this operation class behind a simple method interface that instantiates it and invokes it correctly:

```
    static Set variablesUsed (Expr e) {
        FindVariables op = new FindVariables ();
        op.forExpr (e);
        return op.vars;
    }
```

This works well enough – at least we didn't need to change the expression classes – but it has one ugly aspect that was omitted from the code above. The method for a composite like `forPlus` doesn't know what method it should call for its subexpressions, since they could be any type of node. So it calls `forExpr`, which tests for and dispatches on all the possible expression types:

```
    void forExpr (Expr e) {
        if (e instanceof Plus) forPlus ((Plus) e);
        else if (e instanceof Const) forConst ((Const) e);
        else if (e instanceof Var) forVar ((Var) e);
        //...
        else assert (false); // don't know e's type
    }
```

Maintaining this code is tedious and error-prone. The long list of cascaded if tests are likely to run slowly. Furthermore, even though this code is bad enough appearing once, in fact it must occur repeatedly in every operation class we define. Systematic repetition in code is usually a sign of a need to redesign, possibly using a pattern.

We already know a Java construct that automatically chooses code to execute based on the type of an object: method dispatch. Method dispatch does the same kind of comparison and selection as the cascaded if tests, but does not clutter the code, is likely to be more efficient, and gives us compile-time checking if we forget a case. The visitor pattern takes advantage of this.

20.3.3 Visitor

The visitor pattern encodes a traversal over a composite hierarchy. As in the procedural approach, an operation is represented by a class, called a visitor, with a method for each type of node in the composite. However, instead of putting type-testing code in the visitor to decide which method needs to be called, that responsibility is shifted to the composite classes. Furthermore, the composite classes will also assume responsibility for the traversal. Here's the basic scheme:

1. A composite node is asked to accept a visitor by a call to its accept method:

   ```
   composite.accept (visitor)
   ```

2. The composite recursively passes the visitor on to its parts:

```
for each part in composite,
-   part.accept (visitor)
```

3. Depending on its type, the composite calls the appropriate method of the visitor, passing itself as an argument.

```
visitor.forNodeType (composite)
```

The accept and visit methods work together in such a way that composite.accept(visitor) performs a depth-first traversal of the structure rooted at composite. In this case, the traversal is called post-order because a node is visited after all its children have been visited. Other possibilities are pre-order, which visits the node first, and in-order, which visits the node between its two children and makes sense only for binary trees.

For our expression example, the code looks like this:

```
interface Visitor {
    void forPlus (Plus e);
    void forConst (Const e);
    void forVar (Var e);
    //...
}
interface Expr {
    void accept (Visitor v);
    //...
}
class Plus implements Expr {
    Expr left;
    Expr right;
    void accept (Visitor v) {
        left.accept (v);
        right.accept (v);
        v.forPlus (this);
    }
    //...
}
class Const implements Expr {
    String name;
    void accept (Visitor v) {
        v.forConst (this);
    }
    //...
}
class Var implements Expr {
    String name;
    void accept (Visitor v) {
        v.forVar (this);
    }
    //...
}
```

Using the visitor pattern, the `FindVariables` operation could be implemented much more simply. There is no need for `instanceof` tests or downcasts, or even any traversal code. In fact, only the `forVar` method has a nontrivial body:

```
class FindVariables implements Visitor {
    Set vars = new HashSet ();
    void forPlus (Plus e) { }
    void forConst (Const e) { }
    void forVar (Var e) { vars.add (e.name ()); }
    //...
}
```

However, the method that uses `FindVariables` must reverse the way it invokes the operation. Instead of passing the expression to the operation, it must pass the operation to the expression:

```
static Set variablesUsed (Expr e) {
    FindVariables op = new FindVariables ();
    e.accept (op);
    return op.vars;
}
```

This validates the decision we made to hide `FindVariables` behind a method interface. Otherwise, changing it to a visitor would have entailed changing every place in the code where it was invoked.

A visitor is very much like an iterator. Each element of the hierarchy is presented to the visitor in turn, with the additional benefit (not provided by the iterator pattern) of a dispatch based on the type of the element. Further more, a visitor can accumulate state over the course of its traversal, e.g., the set of variable names encountered. Unfortunately, the visitor pattern shown above does not give the visitor any control over the traversal. What if we wanted to implement `eval` as a visitor, rather than an interpreter? This would be hard as it stands. To implement `Plus`, for example, the visitor would need the results of evaluating its two subexpressions, but the design above only makes it easy to receive the result of the last node that was visited.

The Liskov text proposes one solution to this problem: saving results on a stack and popping them off at the appropriate times. This keeps the visitors and acceptors clean, but it can be hard to see how data flows between visitor calls.

Another solution is to move responsibility for the traversal back into the visitor. We'll call it a `SelfGuidedVisitor` to distinguish it from the more passive kind of visitor, but its methods are superficially the same:

```
interface SelfGuidedVisitor {
    void forPlus (Plus e);
    void forConst (Const e);
    void forVar (Var e);
    //...
}
```

When given a self-guided visitor, the `accept` methods do nothing but type-dispatching:

```
interface Expr {
    void accept (SelfGuidedVisitor v);
    //...
}
class Plus implements Expr {
    Expr left;
    Expr right;
    void accept (SelfGuidedVisitor v) {
        v.forPlus (this);
    }
    //...
}
class Var implements Expr {
    String name;
    void accept (SelfGuidedVisitor v) {
        v.forVar (this);
    }
    //...
}
```

The self-guided visitor must take over its own traversal, by asking the children of each visited node to accept it. Because it controls when children are visited, the visitor can also keep track of the results returned by those children:

```
class Evaluator implements SelfGuidedVisitor {
    Context context; // maps variables > values
    double result; // value of last subexpr evaluated
    void forPlus (Plus e) {
        // evaluate left subexpression
        e.left ().accept (this);
        // hang onto its result
        double left = result;
        // evaluate right subexpression
        e.right ().accept (this);
        double right = result;
        // combine the two results
        result = left + right;
    }
    void forConst (Const e) {
        result = e.value ();
    }
    void forVar (Var e) {
        result = context.getVarValue (e.name ());
    }
    //...
}
```

Both variants of the visitor pattern are useful members of your toolbox, and a well designed composite hierarchy may need to support both kinds.

20.4 Model/View/Controller

The model/view/controller (MVC) pattern is a *separation of concerns* between:

- The *model*, which manages application data – for example, the text contained in a document, or the values in a spreadsheet.
- *Views*, which manage how different parts of the model are presented to the user – for example, charts, graphs, etc.
- *Controllers*, which manage how input from the user via a keyboard, mouse, etc. changes the model.

For example, a graphical text input box might have a mutable string as its model, a view to render that string within a confined image region, and a controller to accept and respond to keystrokes by inserting characters into the string.

This pattern has a lot in common with the `Observer` pattern; the model knows very little about the views and controllers, other than the fact that it can notify them about changes. It also makes use of the `Composite` pattern, since views are commonly nested.

In practice, it is quite hard to separate views from controllers, particularly when continuous visual feedback is to be given to the user. So the MVC pattern often becomes Model-View, where the controller aspect is folded in with the view. But the term is still used very frequently and you should be familiar with it.

20.5 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.