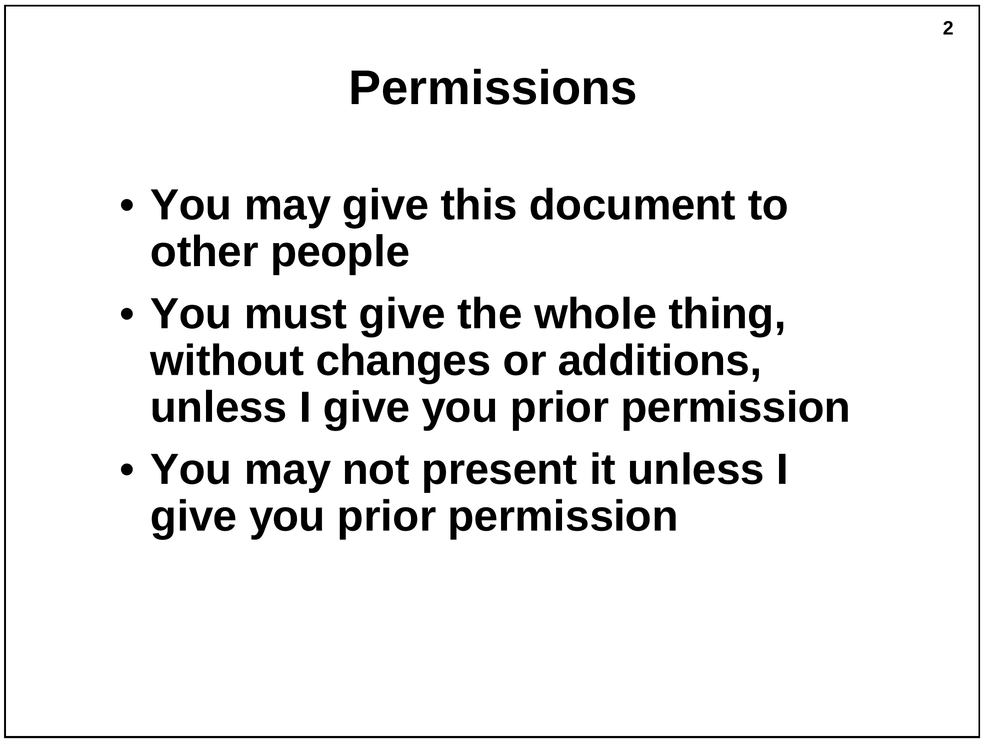
Testing for Programmers

Brian Marick
marick@testing.com
www.testing.com

TABLE OF CONTENTS



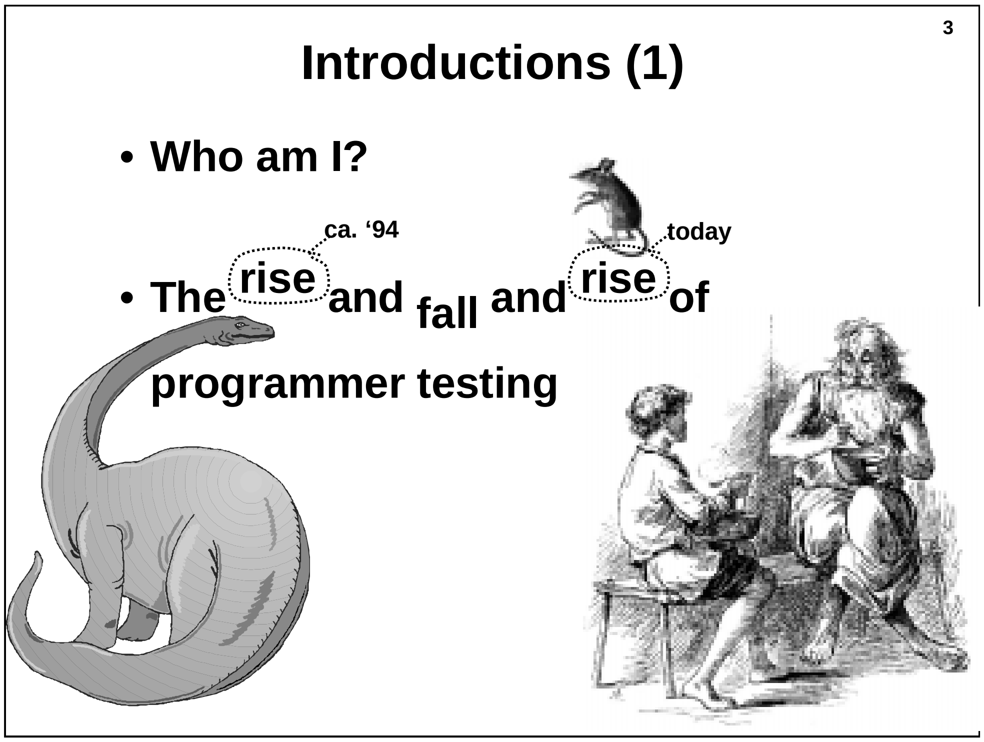
Permissions

- **You may give this document to other people**
- **You must give the whole thing, without changes or additions, unless I give you prior permission**
- **You may not present it unless I give you prior permission**

I present this material in a half-day to full-day tutorial format, but I believe these notes are suitable for self-study. So anyone may pass them on, verbatim, to anyone else.

If you're looking at a paper copy, you can find an electronic one at <http://www.testing.com/writings/half-day-programmer.pdf>.

Some of these pages refer to a "catalog". That can be found at <http://www.testing.com/writings/short-catalog.pdf>.



3

Introductions (1)

- Who am I?
- The rise and fall and rise of programmer testing

All tutorials are infected by the presenter's experience and point of view.

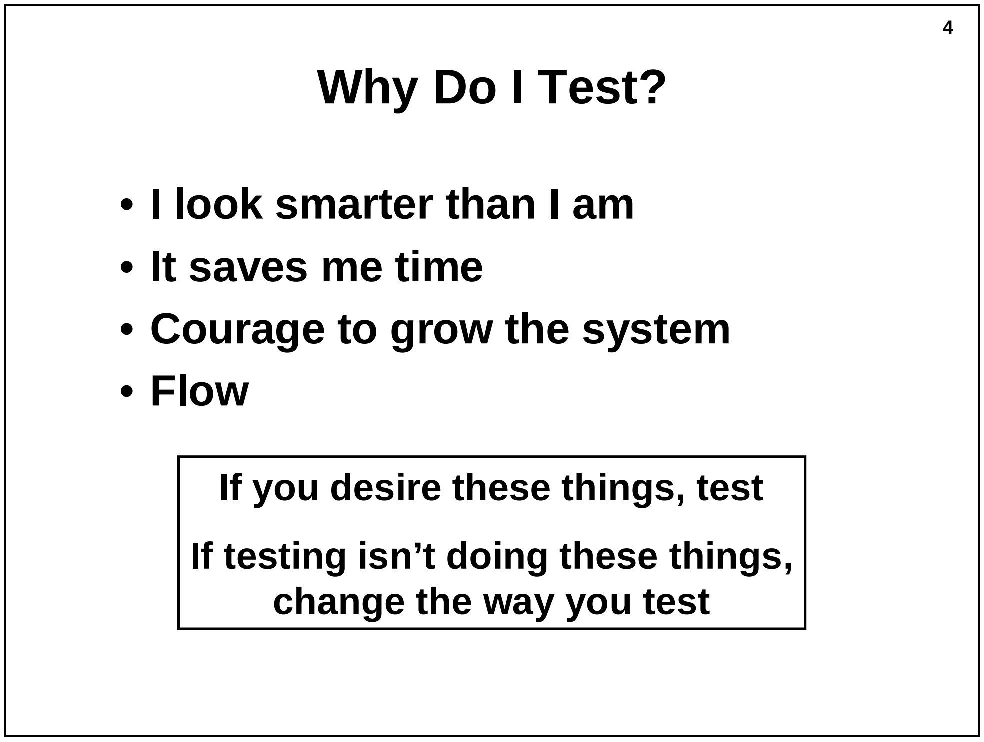
What's my experience? I graduated in 1981 from the University of Illinois. I went to work for a startup. They didn't know much about me, so they put me in testing, where "you can't do as much damage if you're no good". I was fairly good at testing. After a time, the project needed more programmers, so I moved into programming, where I spent the next seven or so years, concentrating on compiler-like things.

Importantly, I believe I was seen as a better programmer than my natural gifts would justify. That's because I tested. I made lots of bugs, but they were caught before my fellow programmers could see and be hurt by them.

I went back into testing, then went independent as a testing consultant in 1992. I specialized in teaching developers how to test. For a while, programmer testing was somewhat faddish. The reason, I think, was a common belief that the Deming-style quality efforts the Japanese had popularized for cars needed to be applied to software. Instead of "scrapping and reworking" software toward the end of the "production line", problems should be found as soon as possible.

That programmer testing movement (if I can call it that) faded away. Now, thanks to books like *Refactoring* (Fowler) and *Extreme Programming Explained* (Beck), it's back. I hope it sticks this time.

Am I a battle-scarred veteran with wisdom to impart? Or am I like many consultants, who think all that's new is just a recapitulation of their glory days? Am I a lumbering dinosaur with little to tell today's generation of nimble mammals? We'll see!



Why Do I Test?

- **I look smarter than I am**
- **It saves me time**
- **Courage to grow the system**
- **Flow**

If you desire these things, test

If testing isn't doing these things, change the way you test

These days I'm programming more. In fact, I love to program, much more than I like to test. In many ways, I'm a typical compulsive programmer. So why do I test?

As I mentioned, it makes me look smarter. That's a pretty crass reason, so I have others.

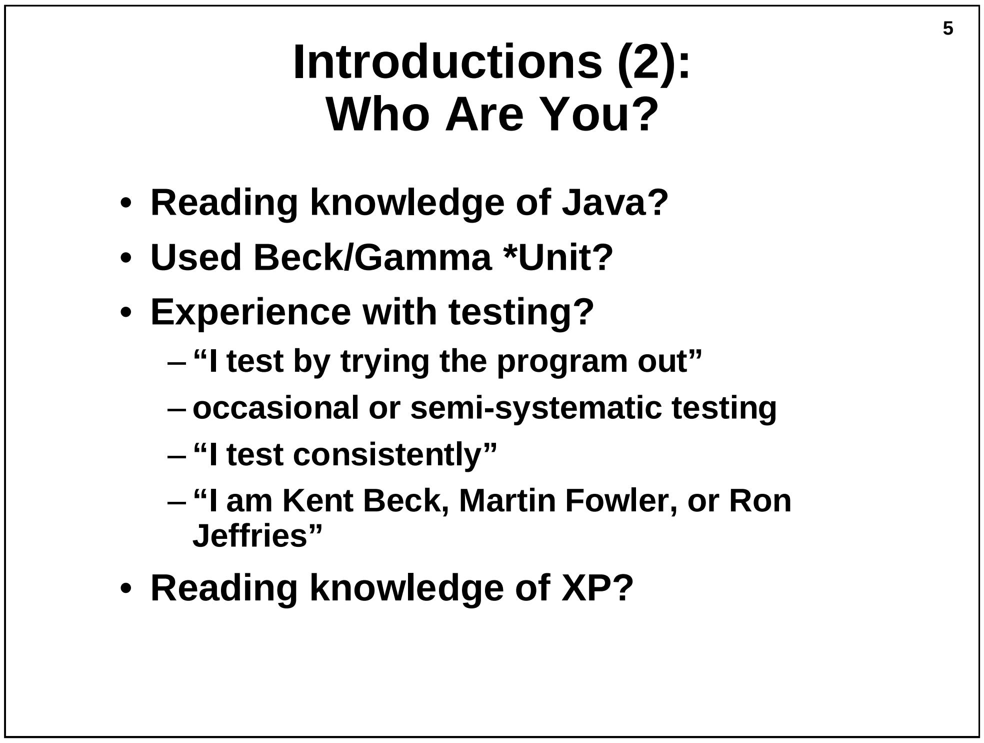
Today, most of my programming does not make me money. I don't have a lot of time. I need to get a lot done quickly. Testing makes me faster because I find bugs quicker, while they're easier to fix. But I'm only faster because the time I save through testing exceeds the time I spend doing testing. More on that later.

When I was a full-time programmer, I used to work on big systems. I used to be scared to change them. Who knew when a small change over *here* would break something way over *there*. A good set of tests gives me the courage to change the system. I'm more likely to take the risk of making it better. Testing helps keep systems useful and nimble.

Finally, when I'm programming, I want to *program*. I don't want to keep getting distracted, jumping out of the programming flow into fixing mistakes and figuring out what it is that I really want to do. Testing helps me with that - I'll explain more later.

To my mind, any testing regime that doesn't help me attain these goals should be replaced with one that does. What I describe in this tutorial works for me. It would probably also work reasonably well for you.

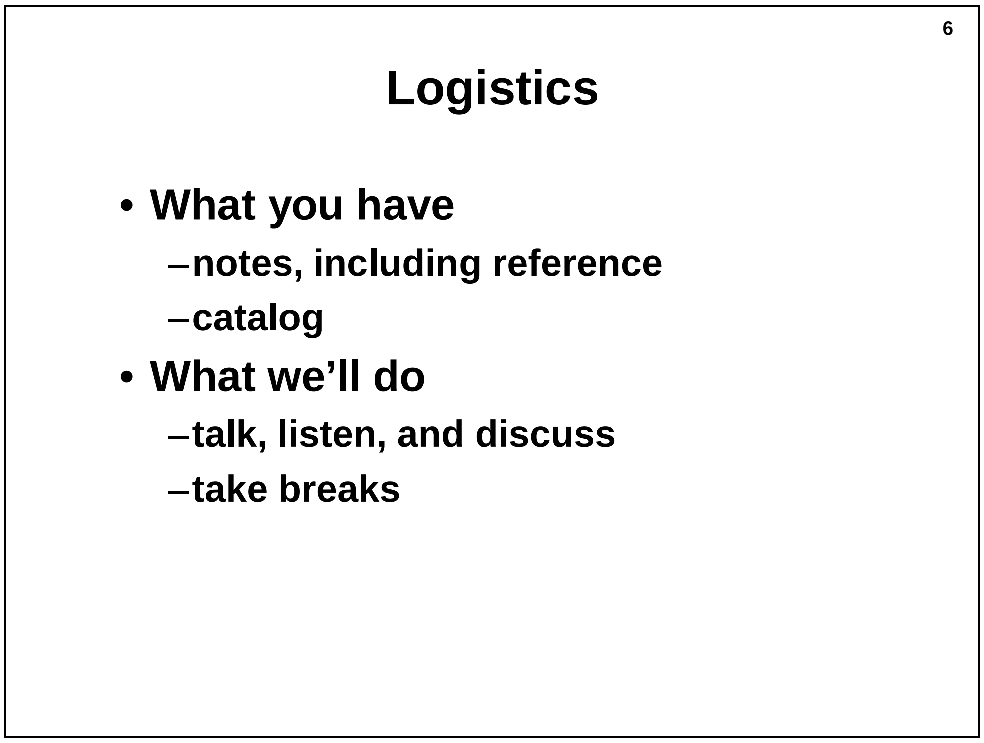
(For information about "flow", see *Flow: The Psychology of Optimal Experience*, by Mihaly Csikszentmihalyi.)



Introductions (2): Who Are You?

- Reading knowledge of Java?
- Used Beck/Gamma *Unit?
- Experience with testing?
  - "I test by trying the program out"
  - occasional or semi-systematic testing
  - "I test consistently"
  - "I am Kent Beck, Martin Fowler, or Ron Jeffries"
- Reading knowledge of XP?

(XP is the common abbreviation for Extreme Programming.)



Logistics

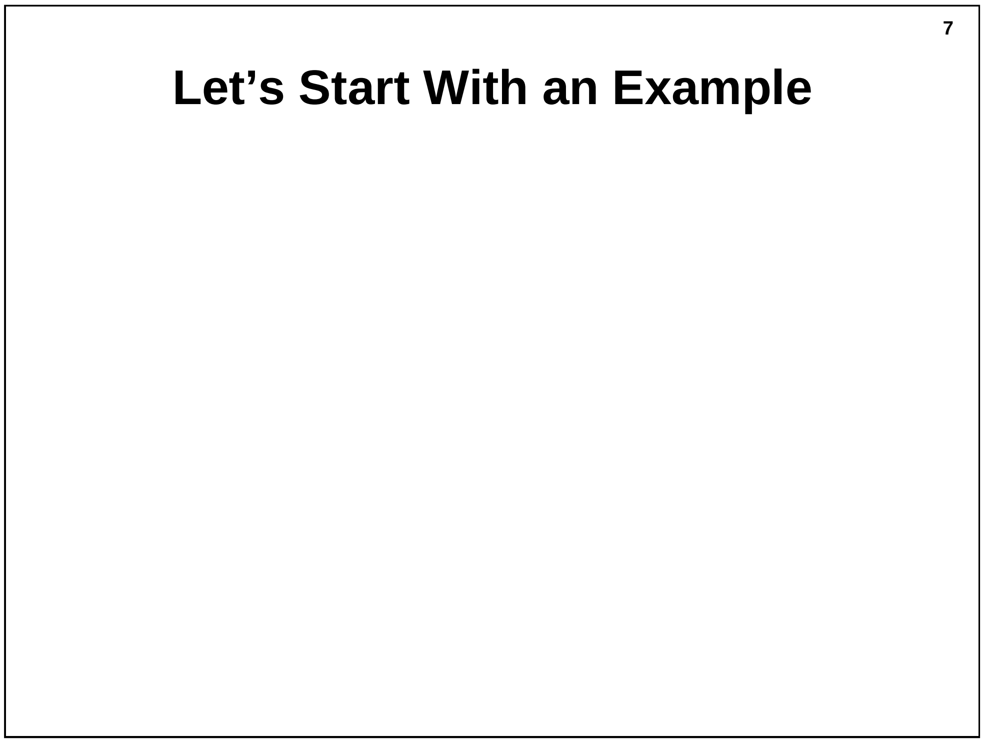
- **What you have**
  - **notes, including reference**
  - **catalog**
- **What we’ll do**
  - **talk, listen, and discuss**
  - **take breaks**

You have these slides and a one page catalog to use in testing. It’s the last page in this handout.

I wish there were labs in this tutorial, but time doesn’t allow.

Labs have value for teaching, of course, but they’re especially nice because they break up the pace and keep people alert.

To break up the tedium, I’ll pause for class discussion. Please feel free to talk throughout the half-day.

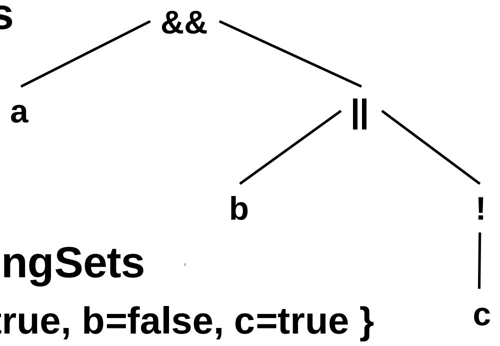


Let’s Start With an Example

At this point in the tutorial, I switch to a different slide show. It shows how I used developer testing in some code I was working on. I’ve written that story up separately. You can find it here: <http://www.testingcraft.com/trace-testing/test-then-code.html>

Let’s Design Some Tests (Background)

- Evaluate boolean expressions
- Exprs



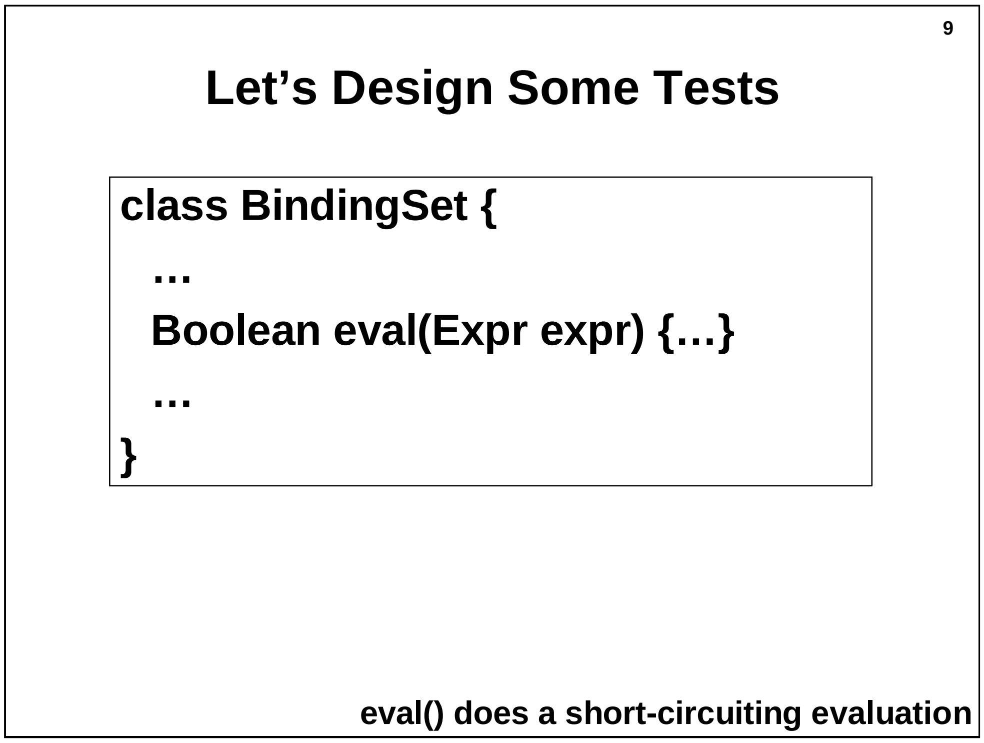
- BindingSets

  { a=true, b=false, c=true }

Let’s think about some tests for some code I wrote recently. It’s pretty mundane code.

There are trees, **Expr**s, that represent boolean expressions. They can contain terminals, && for the AND operator, || for the OR operator, and ! for the NOT operator.

BindingSets represent bindings (or assignments) of boolean values to terminals.



9

Let’s Design Some Tests

```
class BindingSet {
  …
  Boolean eval(Expr expr) {…}
  …
}
```

eval() does a short-circuiting evaluation

Class BindingSet has a particular method, called **eval**. It is given a boolean expression to evaluate. Suppose it’s given the Expr form of this string:

a && (b || !c)

Suppose the BindingSet contains these bindings:

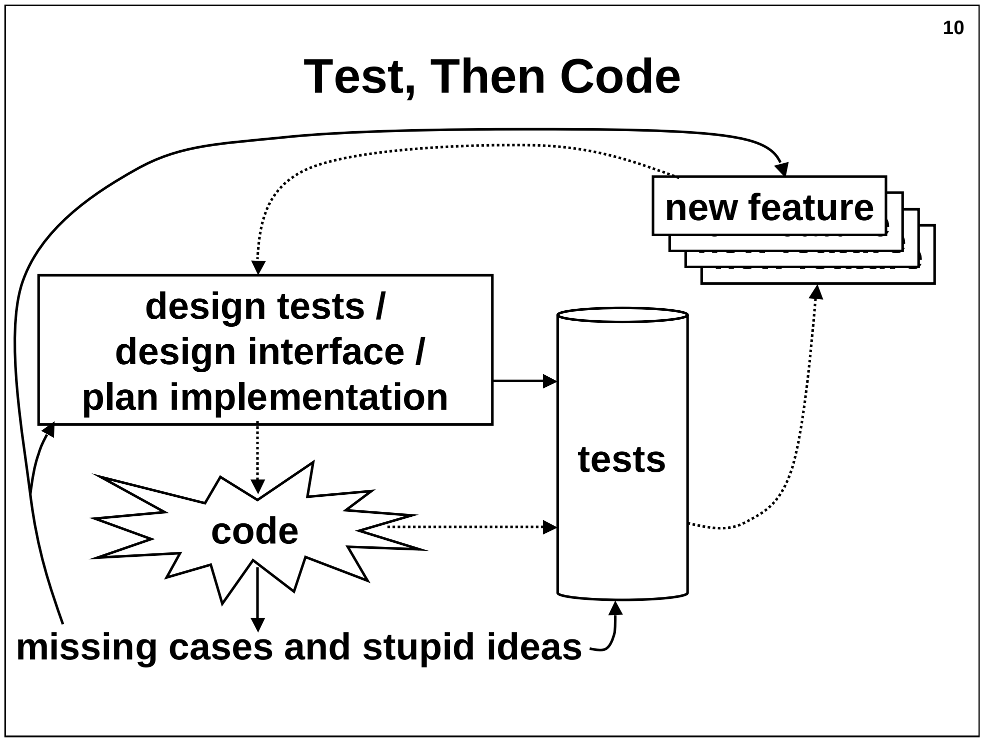
{ a = true, b = false, c = true }

The result should be Boolean.FALSE.

The above is a description of a test case. It first describes some inputs. I use the term “input” to refer to any datum that could affect the computation. In the example, the inputs are both the argument to eval() and also the state of the BindingSet instance.

A test case also describes an expected result. If the actual result from the method matches the expected result, the method passes the test. Otherwise, it fails.

One other thing about eval(): it follows the usual short-circuiting evaluation used in programming languages. That is, if the expression is A&&B and A is false, B is not evaluated.


10

Test, Then Code

Notice what we just did. We designed some tests before looking at the code. We might have even been designing the tests before the code was written. In fact, that's the right order: Test, then code.

To be more explicit:

First, you're given one or more features to implement. To do that, you need to decide on interfaces and behaviors for your code. You also do some amount of thinking about what the code will look like.

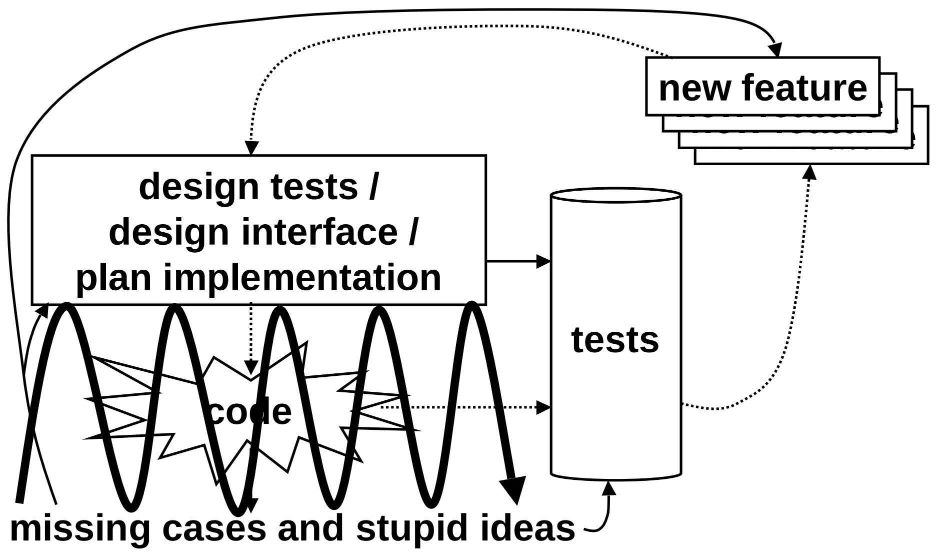
As you think about interfaces and behavior, you document that thinking in the form of tests for those interfaces. For example, you might have a set of tests for any significant class that you write.

Then you write the code. The code is not done until it passes the tests. Until it does, you keep working. You don't start on any new features.

Now, unless you're superhuman, the process of writing code will reveal things you overlooked. Some of your ideas about behavior or interfaces will now look dumb. That's OK. You go back and change your design. You may even change the definition of the problem you're trying to solve (the feature you're trying to implement).

Importantly, the discoveries you make during coding lead to new tests. If you notice you need a new behavior, you first add a test that checks whether that behavior is implemented correctly.

Testing Early: Saves Time, Smoother Flow



Why is it so important to write the tests before the code?

• One reason has already been mentioned: finding mistakes before coding promotes the very efficient flow experience during coding. You don't spend so much time jarringly bouncing back and forth between "what should this do?" and "how should it do it?"

• Writing tests first prevents bugs. You know *specifically* what the code has to do, including some things you otherwise wouldn't have thought of. Unsurprisingly, when you code you mostly get those things right. If you write the tests after the code, you'll find you got more things wrong.

An Implemented Test

```
public void testDemo() {
   BindingSet env = new BindingSet();
   env.set("a", Boolean.TRUE); // Note B is not set.
   Expr expr = Parser.parse("a || b");
   assert(Boolean.TRUE == env.eval(expr));
}
```

Here is what an implemented test looks like. I'm using Beck and Gamma's jUnit test framework, which you can find at <http://www.junit.org>.

This test checks that short-circuiting is handled correctly. That is, "A||B" should have the value TRUE when A is TRUE, even when B has no value.

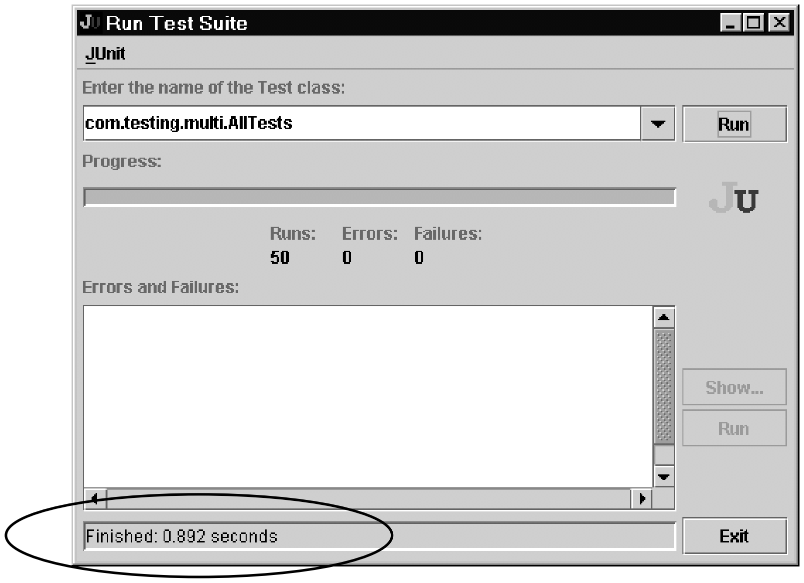
It first creates a binding set and puts a=TRUE into it. It then creates an Expr from the string "a||b".

Then the test **assert**s that eval(), applied to the expression, is identically TRUE. Assert() is a method provided by jUnit. It records an error if its boolean argument is FALSE and proceeds to the next test.

Note: You know how people argue about where to place curly braces in code or about whether instance variables should begin with "_"? Test drivers provoke the same passion about issues like how many asserts should be in a test method, how much setup should be inside the test method vs. in a separate setup method, and what the test hierarchy should look like.

Those decisions have to be made. However, I don't think it's critical how you make them, so I won't discuss them in this tutorial. I have too much material already. You're welcome to add them to the Hot Topics list.

The Result



```
com.testing.multi.AllTests
Runs:  Errors:  Failures:
50     0        0
Errors and Failures:
Finished: 0.892 seconds
```

test often

This page shows you what you want to see: all the tests have passed.

(You may notice that I have only 50 tests. I actually have many times that number of asserts, since I tend to put multiple asserts per test. Specifically, I have 376 asserts in my test suite.)

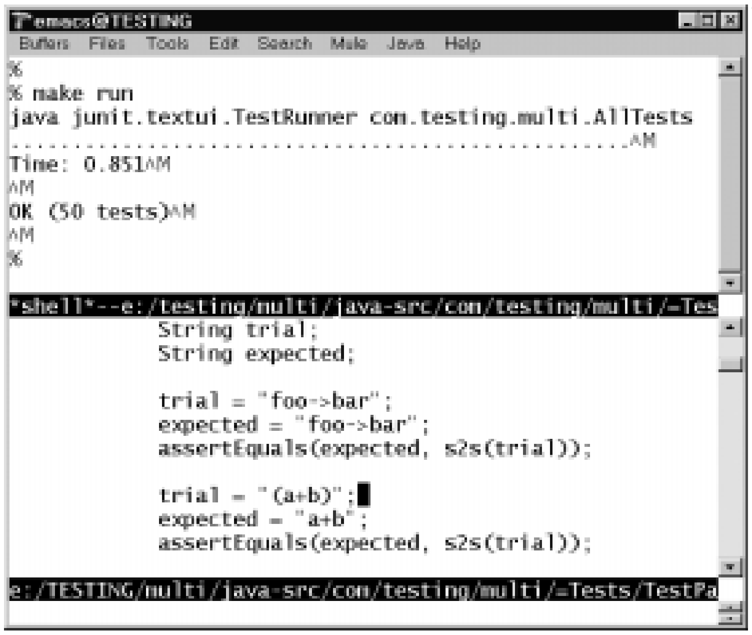
I've circled something quite important: all my tests for a chunk of code that today is 1681 lines run in less than a second. Because they run so fast, I'm encouraged to run them often. I make a change, run a test, make a change, run a test. I'm never in doubt about where my program stands.

This **quick cycling** between coding and testing **is essential.** With it, you find bugs immediately. Fixing them is usually simple and doesn't disrupt the flow of what you're doing. Quick cycling is what makes programmers true believers in testing.

Note: that's it for this course on the mechanics of using a test automation tool. They - and especially jUnit - are easy to use, and I refer you to their tutorials.

A side note on my test suite: I actually have 2309 lines of test code right now, even more than product code. That's misleading. I spent much less time on test code than on product code. There's much more cutting and pasting, more boilerplate code.

(What I Really Do)



```
%
% make run
java junit.textui.TestRunner com.testing.multi.AllTests
..............................................^M
Time: 0.851^M
^M
OK (50 tests)^M
^M
%
```

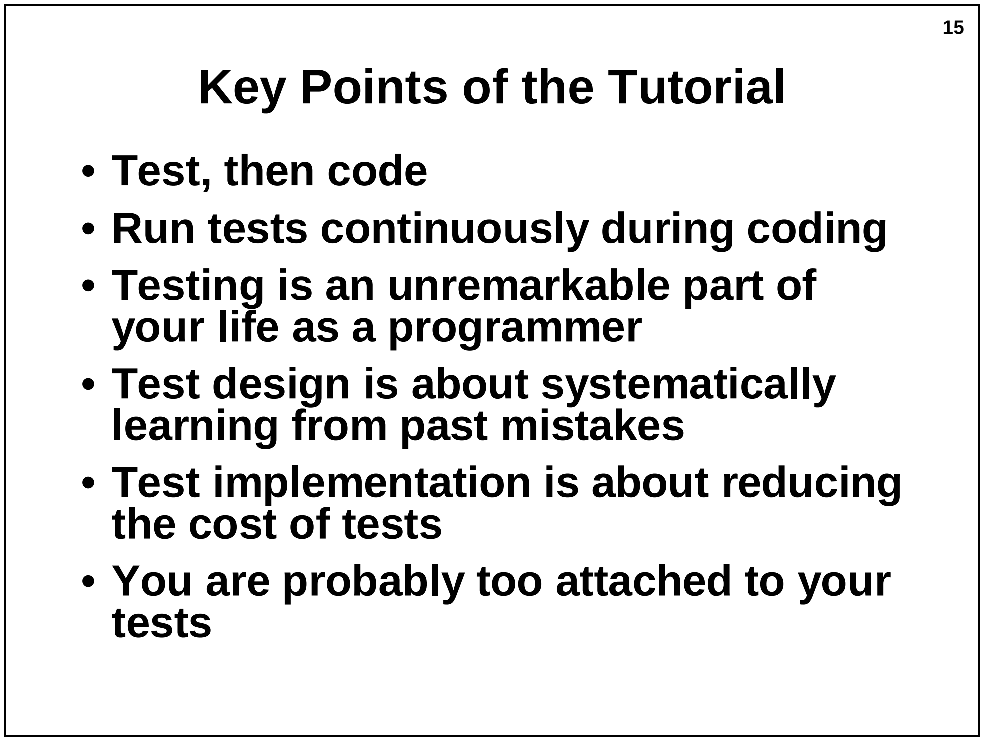
shell--e:/testing/multi/java-src/com/testing/multi/=Tes

```
            String trial;
            String expected;

            trial = "foo->bar";
            expected = "foo->bar";
            assertEquals(expected, s2s(trial));

            trial = "(a+b)";
            expected = "a+b";
            assertEquals(expected, s2s(trial));
```

e:/TESTING/multi/java-src/com/testing/multi/=Tests/TestPa

Actually, I don’t usually use the GUI. I run the command-line interface through an Emacs shell window, as shown in the slide.

Maybe I *am* a dinosaur: Emacs is still my favorite programmer’s editor. Plus, using the command line saves me 0.041 seconds per run…



15

Key Points of the Tutorial

- **Test, then code**
- **Run tests continuously during coding**
- **Testing is an unremarkable part of your life as a programmer**
- **Test design is about systematically learning from past mistakes**
- **Test implementation is about reducing the cost of tests**
- **You are probably too attached to your tests**

Having gotten the basics out of the way, let me summarize where we'll be going. I would like to begin to convince you of the following points:

• You should create a test before you write the code it is to test.

• You should run tests continually during coding. I am completely unsurprised when I find myself running my test suite many times an hour.

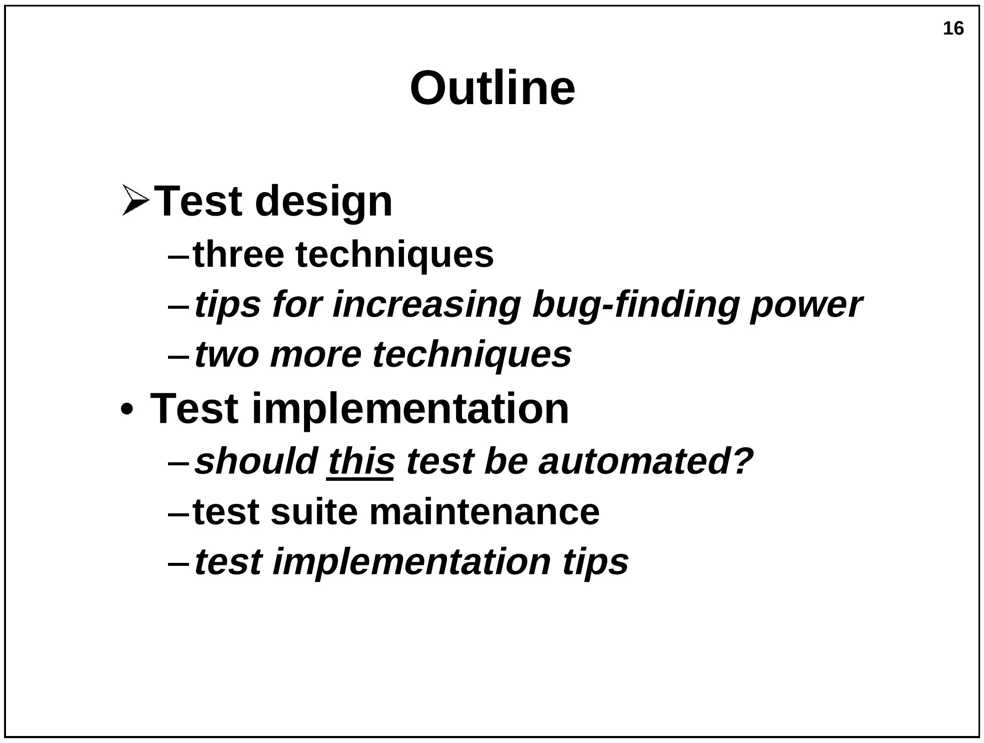
• Testing is not a **separate** activity from programming. It's as much a part of programming as compiling, browsing code, or any of the other things that are part of the process of getting correct program text into the machine.

Those points have, I hope, been covered already. Some others are coming up:

• When you design tests, you're only incidentally focusing on demonstrating that the code works. Oh, you want to show that, all right, but that's an easy consequence of trying to show that the code is wrong. You're looking to see if you've made certain likely mistakes. You know they're likely because you've made them before.

• If test design is about picking test inputs that have power at finding mistakes, test implementation is about running those inputs as cheaply as possible. The more bang for the buck, the more likely you are to continue testing.

• Most programmers are too attached to their tests. They automate too many of them. They keep them running after they've outlived their usefulness.



Outline

- ➢**Test design**
  - –**three techniques**
  - –***tips for increasing bug-finding power***
  - –***two more techniques***
- **Test implementation**
  - –***should <u>this</u> test be automated?***
  - –**test suite maintenance**
  - –***test implementation tips***

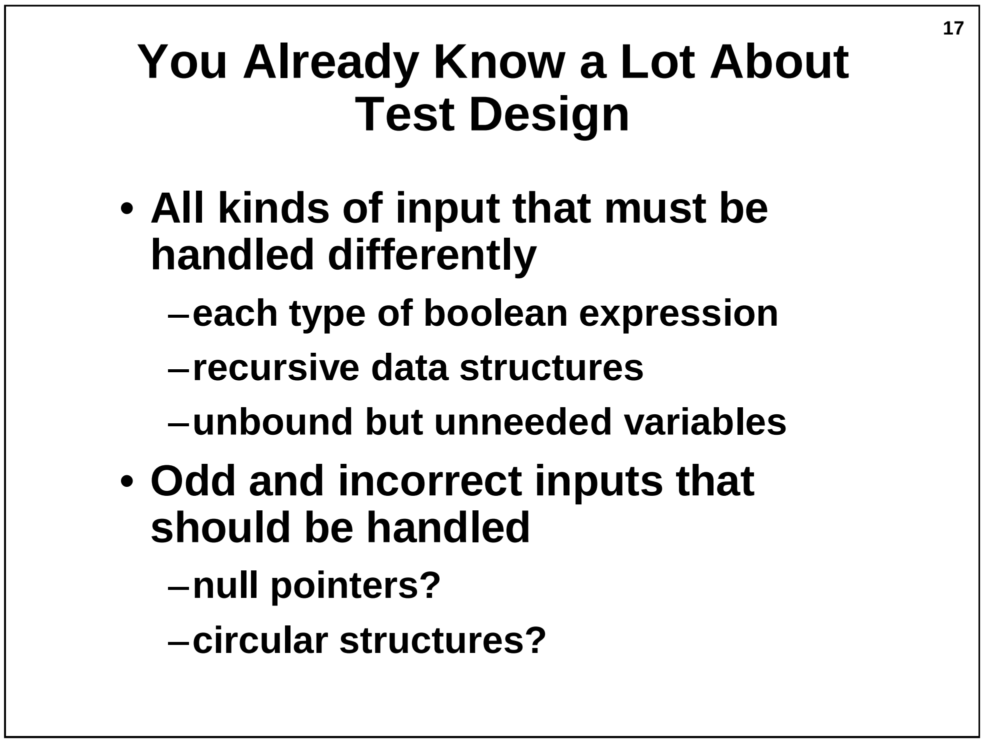
There are two broad topics. The first is test design. What is it that makes a good test? How much testing should you do?

I'll cover three test design techniques in the presentation. I'll leave two others for reference. I also have a variety of tips for improving the bug-finding power of tests. I'll leave those for reference as well. (There's not enough time in half a day to cover everything.)

The second topic is implementation. What should tests look like? How should they change over time? How can they be made less expensive?

(Cost, in this context, is mainly time. Other costs, like tools, are in the noise.)

Again, some of the material is reference, for later reading. We *will* cover the important topic of test suite maintenance. We'll at least touch on how you decide whether a test is worth automating, and we might cover it in depth. We're unlikely to cover the test implementation tips - those are well suited for later reference.


17

You Already Know a Lot About Test Design

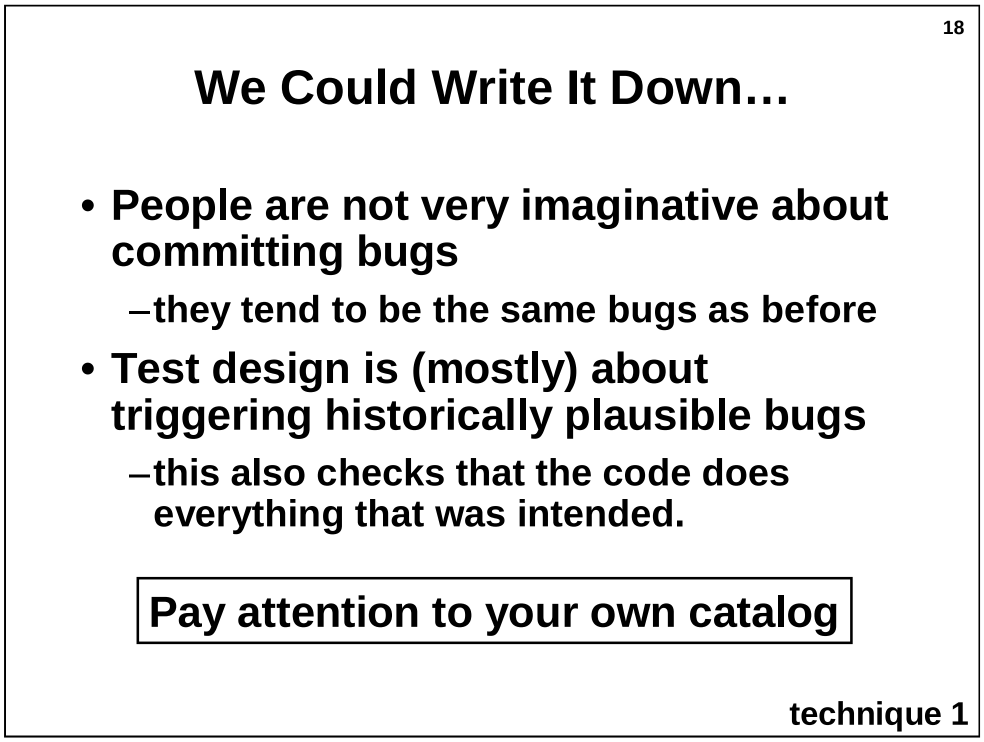
- **All kinds of input that must be handled differently**
  - **each type of boolean expression**
  - **recursive data structures**
  - **unbound but unneeded variables**
- **Odd and incorrect inputs that should be handled**
  - **null pointers?**
  - **circular structures?**

Let's get started with test design.

I hope that the eval() example has demonstrated that the *collective* experience of the audience will come up with a good set of tests. (Though it is likely that not everyone will arrive at all the test cases - perhaps only a few people will.)

I expect we'll have tests that exercise the different kinds of trees (ones rooted with &&, ones rooted with ||, and ones rooted with !). We'll probably have trees that are deeper than one level (forcing recursion). We'll have tests of short-circuiting and probably tests for what happens when the expression can't be evaluated.

I bet we'll also have tests checking what happens when eval() is handed a null pointer instead of an Expr. Probably someone will have suggested a test that checks what happens when the tree isn't a tree at all, but rather some circular structure. I'll choose not to write that test - if someone passes in a circular structure, the results are undefined.



18

We Could Write It Down…

- **People are not very imaginative about committing bugs**
  - **–they tend to be the same bugs as before**
- **Test design is (mostly) about triggering historically plausible bugs**
  - **–this also checks that the code does everything that was intended.**

$\boxed{\textbf{Pay attention to your own catalog}}$

technique 1

Where did those test ideas come from? I claim they are primarily - though perhaps subconsciously - based on **the memory of error.** We've all made mistakes.

• Programs do a case analysis - dividing the space of inputs into subspaces, each of which is handled differently. (That's what IF statements, SWITCH statements, and inheritance do.) We've all had the experience of mishandling one of the cases, so we're inclined to check all of them.

• We've been surprised by getting a null pointer instead of an object, so we're inclined to see if *this* piece of code blows up in the way others have in the past.

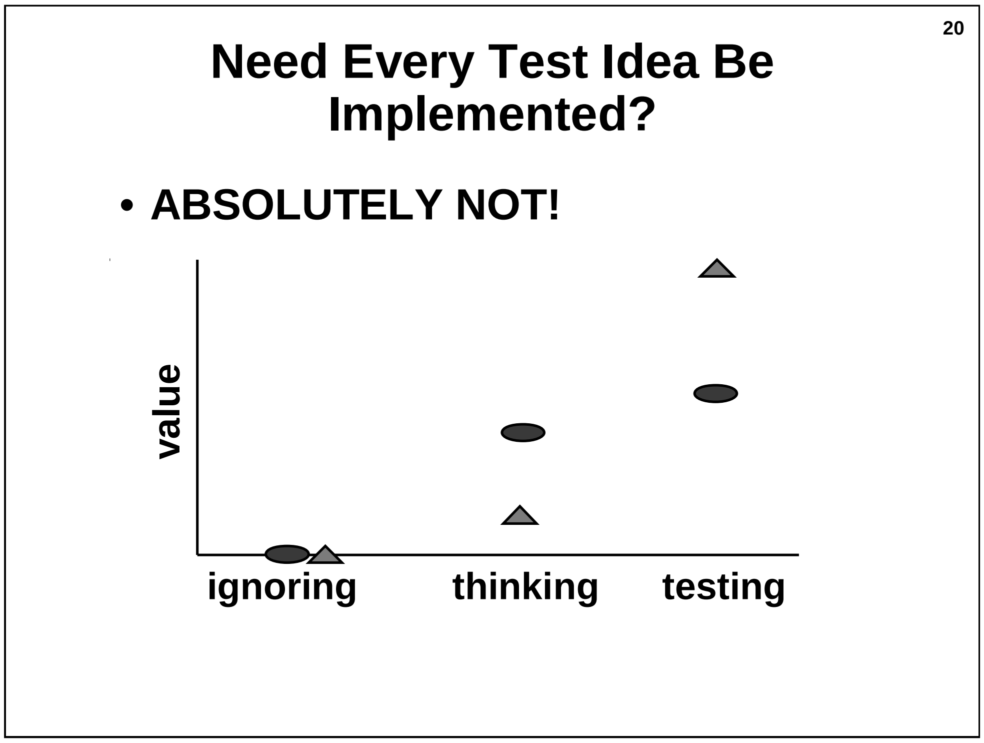
And so on. People make new and innovative mistakes much less often than they repeat the same old typos, thinkos, and other mistakes. Therefore, testing - especially programmer testing - can get a lot of mileage by checking for those common mistakes.

Programming is a pack-rat activity. You need to know lots of different small things to succeed: different idioms, patterns, tricks of the trade, and so forth. Good programmers are always on the lookout for those things, adding them to their toolbox.

Another compartment in that toolbox should be a catalog of good failure-provoking ideas for tests. When you fix a bug that your tests missed, ask what kind of inputs would have uncovered it. Can you try those inputs in similar situations?

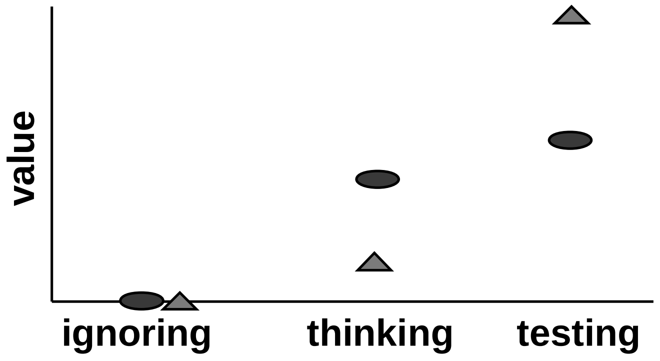
A brief example of such a catalog is given as the last page of this handout. It's based on a much longer catalog that can be found in my book *The Craft of Software Testing* and on my web site at http://www.testing.com/writings/catalog.pdf.

Class Discussion:
Applying the Catalog



Need Every Test Idea Be Implemented?

- **ABSOLUTELY NOT!**



The catalog provides what I'm going to call "test ideas". They're just ideas, not commandments from on high. Sometimes they're obviously inapplicable. Sometimes they're applicable, but you don't think they're worth testing.

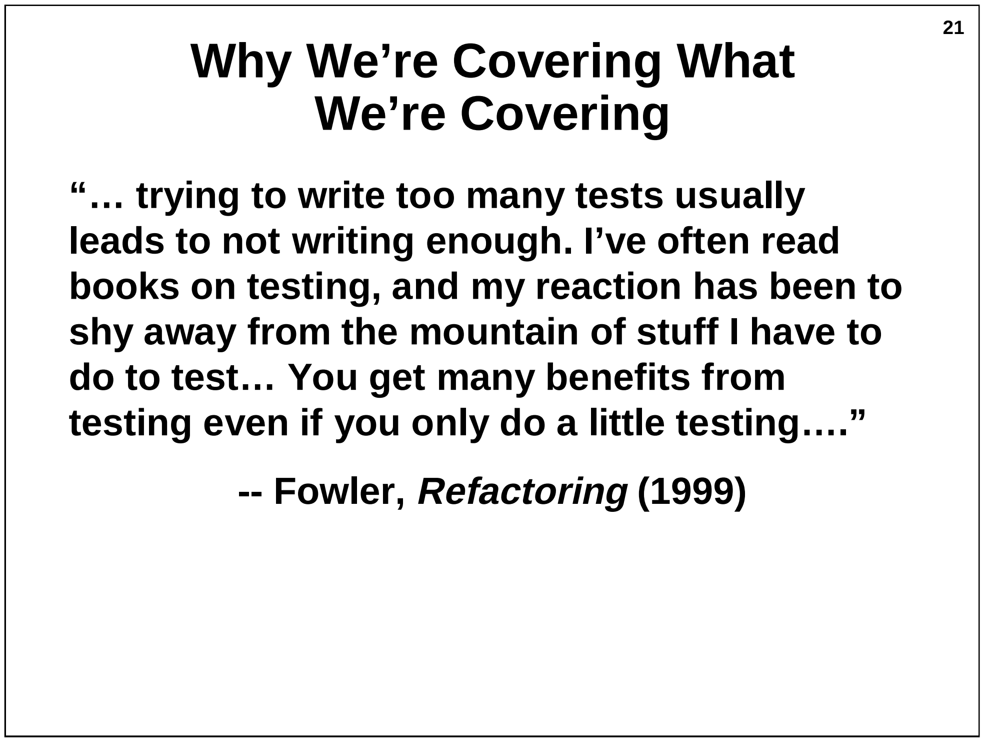
Suppose you were writing OS disk drivers in 1990. You might have looked at the catalog's "one more than maximum size" entry and asked yourself "so what if disks exceed two gig in size?" You couldn't test that - there weren't any such disks (at least, *I* never had one). But you could think about how your code would have to be changed if that day ever came. The code would be better for it.

This especially applies to error handling code. Some errors are extremely difficult to cause. (How do you make that memory allocation on line 345 fail?)

I think it's always the case that thinking+testing has more value than just thinking. No matter how hard you think, it's easy to believe that the code does what you want, not what it does.

But sometimes testing's incremental value is huge (as in the triangles on the slide), and sometimes it's small. Implement tests when they provide value. (We'll see more about that later.)

Let me emphasize this: the mere act of thinking of a test idea will sometimes lead you to smack your forehead and say, "Of course!" That's great return on investment.



Why We're Covering What We're Covering

"… trying to write too many tests usually leads to not writing enough. I've often read books on testing, and my reaction has been to shy away from the mountain of stuff I have to do to test… You get many benefits from testing even if you only do a little testing…."

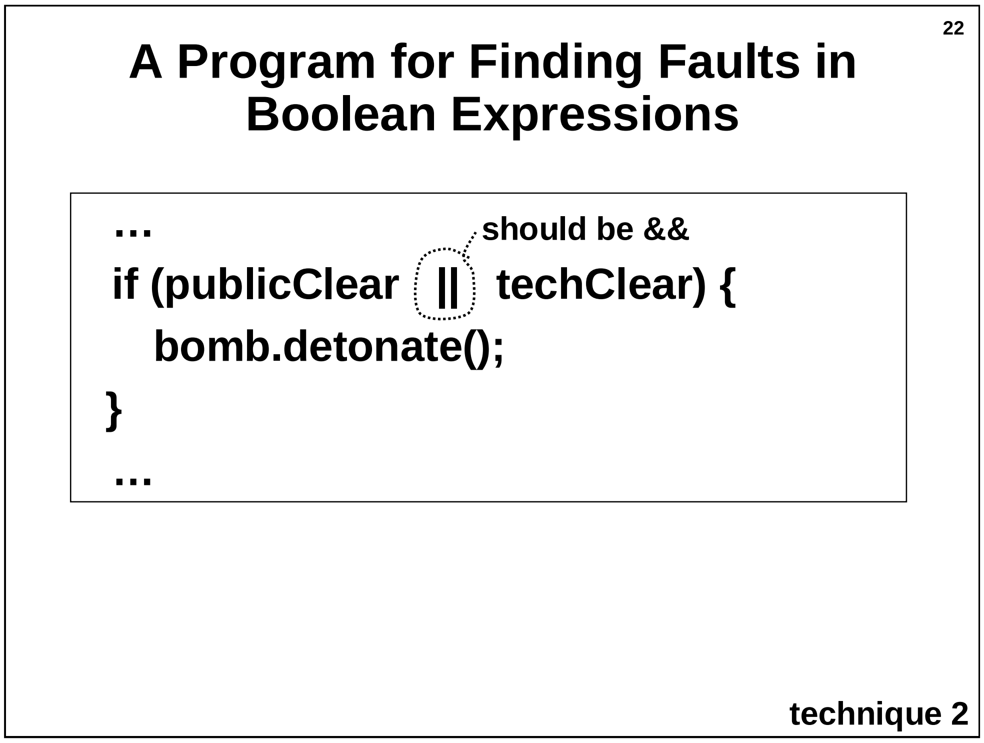
-- Fowler, *Refactoring* (1999)

The quote from Fowler is quite important. It's shaped the way I've designed this tutorial.

I've condensed my catalog down to one page because that's handy enough that people will actually use it. It contains the essential categories.

The catalog is simple to use. The next two techniques are also simple - maybe even simplistic. It seems like you should expect more.

If you're new to testing, I'd like you to just start with these first three techniques as described. As you start to see benefits from testing, add the techniques and enhancements given in the reference sections.



22

A Program for Finding Faults in Boolean Expressions

should be &&

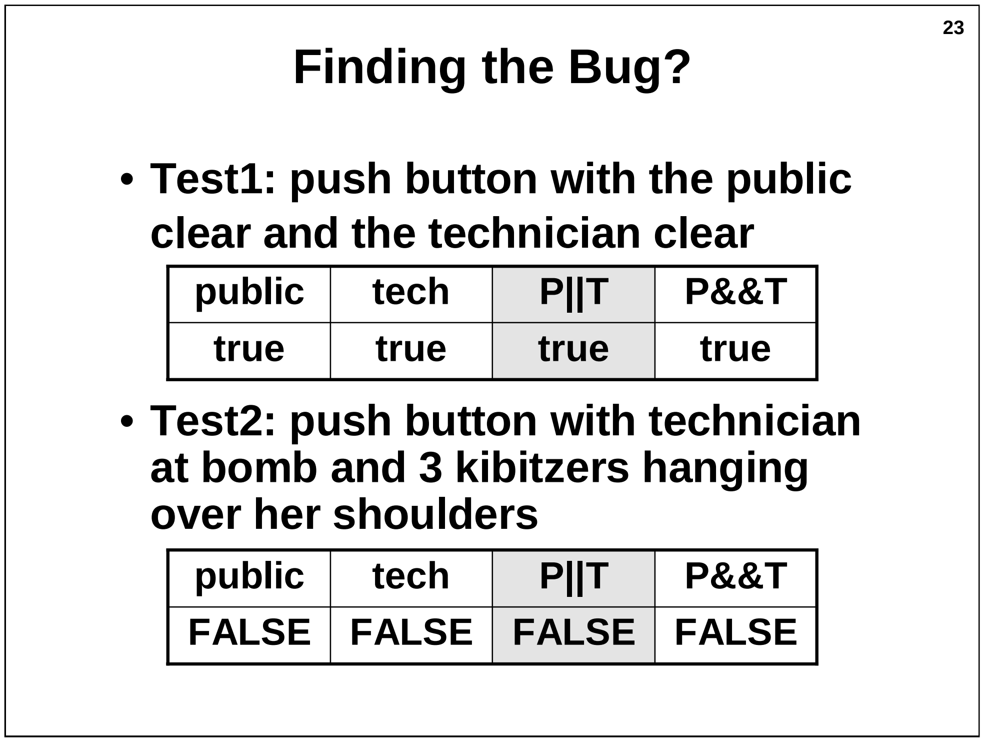
```
…
if (publicClear || techClear) {
   bomb.detonate();
}
…
```

technique 2

The next two techniques give the background you'll need to use a program that provides more test ideas. It gives you test ideas that find certain kinds of historically likely bugs.

People often make mistakes writing boolean expressions. In this case, someone wrote an || when she should have written an &&.

In this code, the result is bad. Instead of detonating a bomb when the public is clear *and* the bomb technician is clear, it detonates it when *either* of those two things is true. Pity the poor bomb technician who shoos the public away and sets to work…



Finding the Bug?

- **Test1: push button with the public clear and the technician clear**

public	tech	P||T	P&&T
true	true	true	true

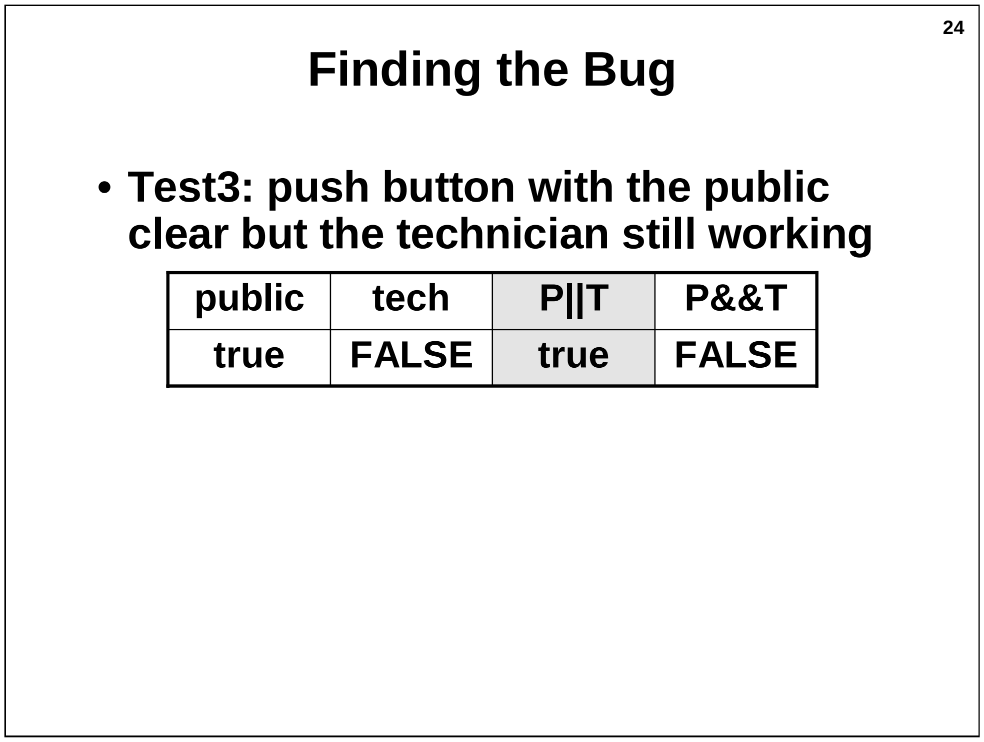
- **Test2: push button with technician at bomb and 3 kibitzers hanging over her shoulders**

public	tech	P||T	P&&T
FALSE	FALSE	FALSE	FALSE

So, the question is: what test would find the bug? Let me go through this laboriously.

If (in test 1) you wrote a test that checked what happens when both the public and the technician are clear, you'd find that the code does the correct thing (blows up the bomb). That's because the *code as written* evaluates to the same value as the *code that should have been written*. So the program takes the right path for the wrong reason: no failure.

For the same reason, the program behaves correctly when both the technician and the public are next to the bomb: the bomb is not detonated.

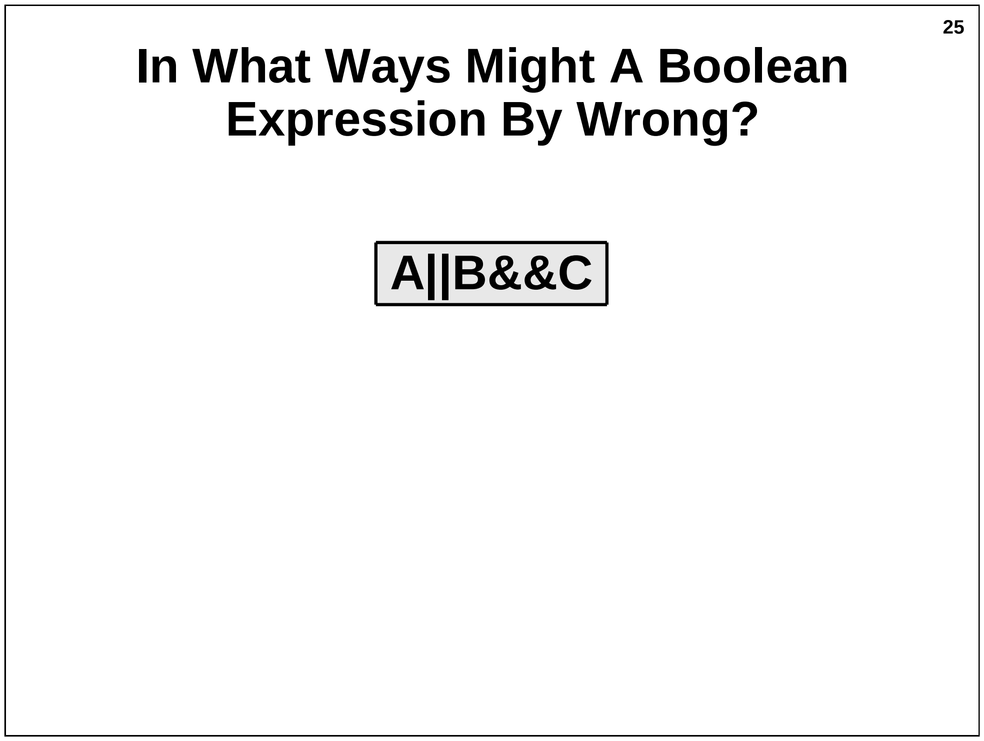


Finding the Bug

- **Test3: push button with the public clear but the technician still working**

public	tech	P||T	P&&T
true	FALSE	true	FALSE

To find the bug, you have to have a case in which the code as written evaluates differently than the code that should have been written. Test 3 shows such a case.


25

In What Ways Might A Boolean Expression By Wrong?

A||B&&C

Any given boolean expression may be wrong in a number of ways. The slide shows A||B&&C. In class, we’ll develop a set of likely faults. Those are the boolean expressions that perhaps *should have been* written instead of the expression that was written.

Here are the ones the program will worry about:

(A||B)&&C	**A&&B&&C**	**!A||B&&C**	**A||!(B&&C)**
A||B||C	**A||!B&&C**	**A||B&&!C**	
A	**B&&C**	**A||B**	**A||C**

• Perhaps the expression-as-written was misparenthesized.

• Perhaps an || should have been an &&, or vice-versa.

• Perhaps a ! was omitted, or should have been added.

• Perhaps the expression was made too complicated, has too many terms. These kinds of faults aren’t all that likely, but cost little to check for.

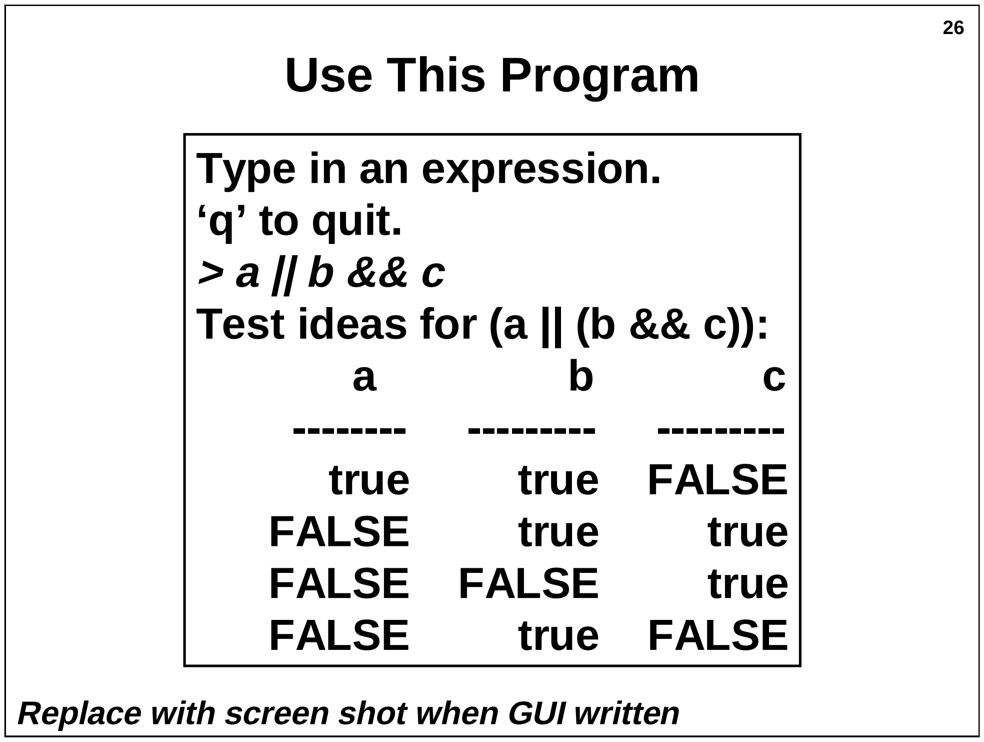
These do not exhaust the likely faults. For example, perhaps the expression should have been:

a||b&&c**&&d**

The technique we’re talking about in these slides won’t necessarily find that fault. It also won’t necessarily find a fault where the wrong variable was used:

a||**d**&&c

That’s OK. No one expects perfection, I hope.



26

Use This Program

```
Type in an expression.
'q' to quit.
> a || b && c
Test ideas for (a || (b && c)):
          a          b          c
   --------  ---------  ---------
       true       true      FALSE
      FALSE       true       true
      FALSE      FALSE       true
      FALSE       true      FALSE
```

Replace with screen shot when GUI written

The program finds a small set of inputs (test ideas) that find likely faults in boolean expressions. As I write (December, 2000), its GUI isn't quite done. When it is, it will be up on my web page as both an applet and a download. (See http://www.testing.com/tools.html). Contact me if you want the command line version (marick@testing.com).

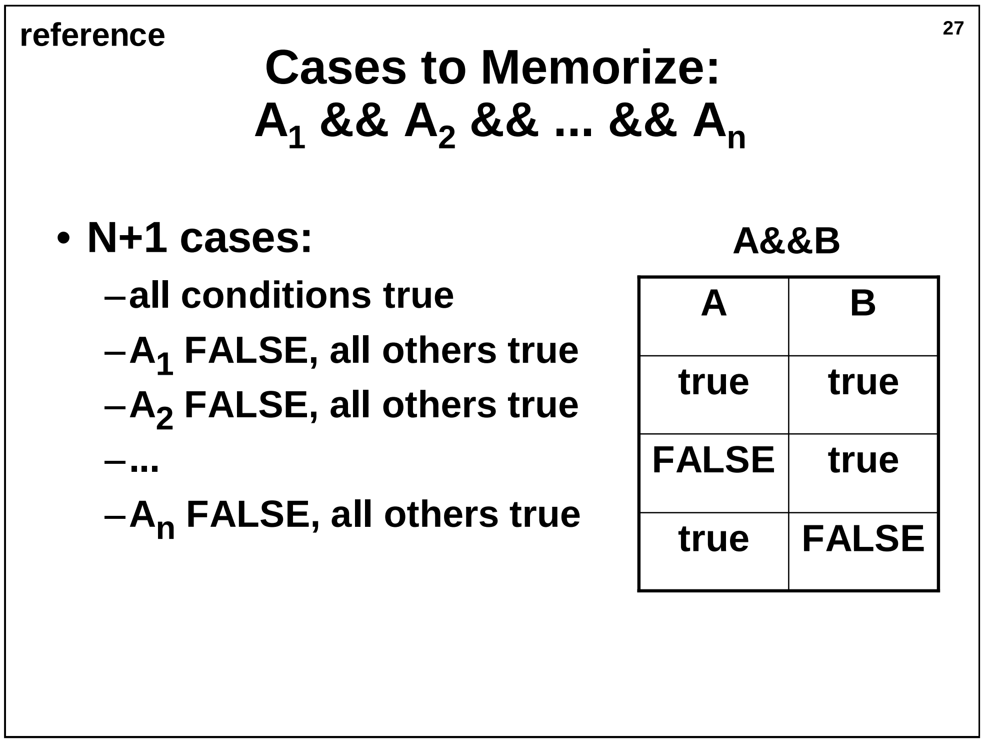
The example on the slide shows the four test cases that suffice to distinguish the expression as given from all the hypothetically more-correct ones on the previous page.

Note that there are only 4 cases, instead of the 2^3 possibilities. I haven't proved that the program produces minimal sets, but I suspect it does.

This code has been checked on all 2,752,512 expressions that have five or fewer binary boolean operators (ands and ors). For example, this:

```
!(!g316559 || !(g329998 || !(!(!g330426 || !(!g330486 || !g330520)) ||
!g330562)))
```

I'm pretty sure it gives the right answers for those expressions.



reference

27

Cases to Memorize: A_1 && A_2 && ... && A_n

- **N+1 cases:**
  - –all conditions true
  - –A_1 FALSE, all others true
  - –A_2 FALSE, all others true
  - –...
  - –A_n FALSE, all others true

A&&B

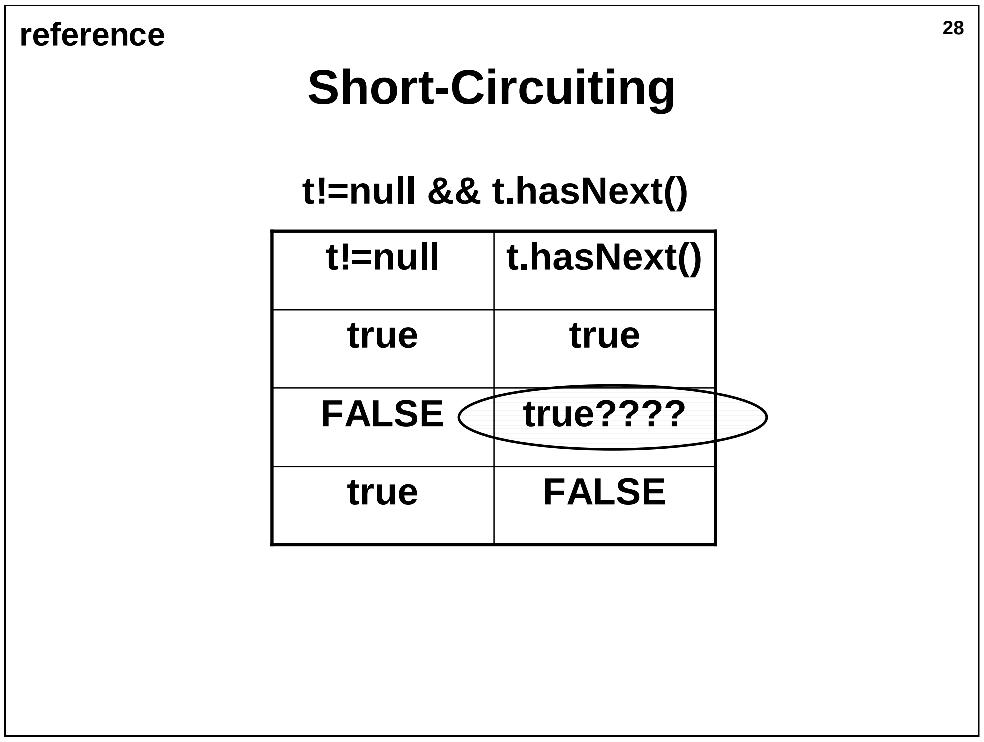
A	B
true	true
FALSE	true
true	FALSE

Relatively few boolean expressions in programs are even as complicated as the one on the previous page. Rather than always running to a program for the answer, there are a few cases worth memorizing.

When the expression contains all &&s and no ||s, there are N+1 cases. The first is the case where all the boolean terms are TRUE, and the whole expression evaluates TRUE.

The remaining cases are each one where exactly one of the terms is FALSE, so the whole expression is FALSE.

A&&B is the simplest example of such an expression. Its three required tests are given on the right.


reference

Short-Circuiting

t!=null && t.hasNext()

t!=null	t.hasNext()
true	true
FALSE	true????
true	FALSE

In C, Java, C++, and similar languages, when the first term of an && evaluates FALSE, the second term is not evaluated. Why, then, should you make that second term be TRUE?

Even worse, it may be that the second term cannot be made TRUE when the first is FALSE. (The slide gives an example.)

Well, if you can't make it TRUE, you shouldn't.

But if you can, you should. The reason is that making a particular term TRUE isn't the point. The point is to distinguish the expression as written from the expression that should have been written. There are some variants of x&&y that the bindings (x=false, y=false) will not detect but (x=false, y=true) will.

That is, if x&&y is correct, it doesn't matter what value you choose for y, given x=false. But if we knew for sure it was correct, we wouldn't be testing in the first place. If it's incorrect, the choice might matter.

reference

Cases to Memorize: A_1 || A_2 || ... || A_n

- **N+1 cases:**
  - –all conditions FALSE
  - –A_1 true, all others FALSE
  - –A_2 true, all others FALSE
  - –...
  - –A_n true, all others FALSE

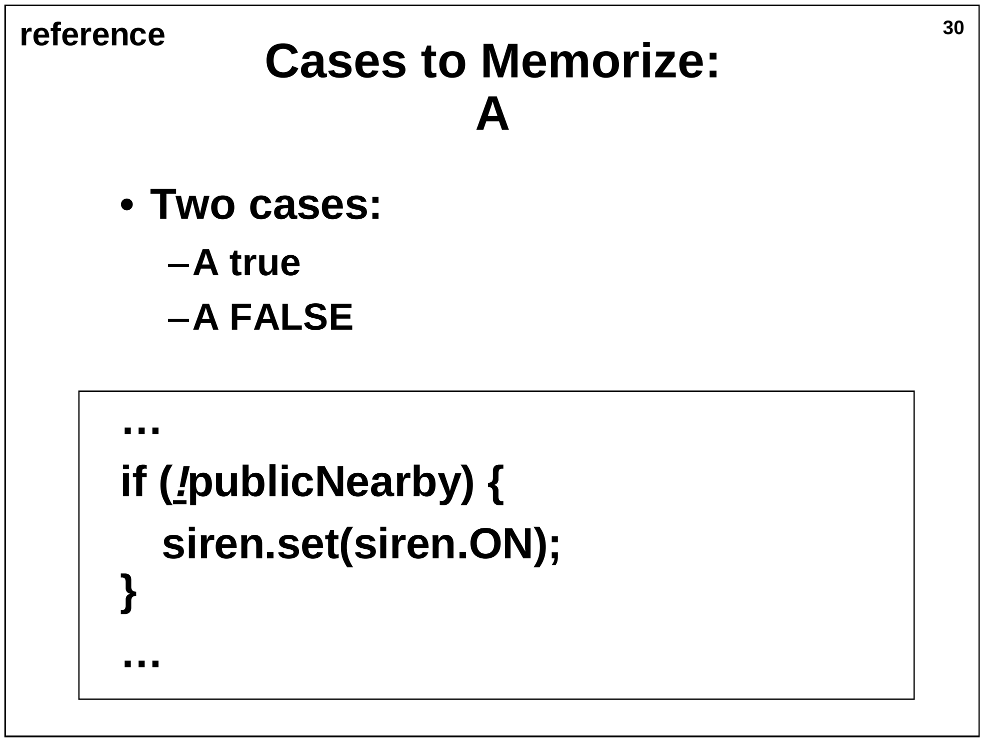
A||B

A	B
FALSE	FALSE
true	FALSE
FALSE	true

As you might expect, an expression with all ||s has a solution quite similar to one with all &&s.

You want the single expression that evaluates to FALSE (because all terms are FALSE).

You want each of the expressions that evaluates to TRUE because exactly one of the terms is TRUE.


reference

Cases to Memorize: A

- **Two cases:**
  - A true
  - A FALSE

```
…
if (!publicNearby) {
   siren.set(siren.ON);
}
…
```

And, whenever you have a boolean value by itself, try both cases.

The slide shows the kind of bug you might find. The “!” is incorrect.

The kind of bug this rule catches is the one where the boolean expression is the negation of what it should be. Either of the cases catches this bug. Why, then, should you try both?

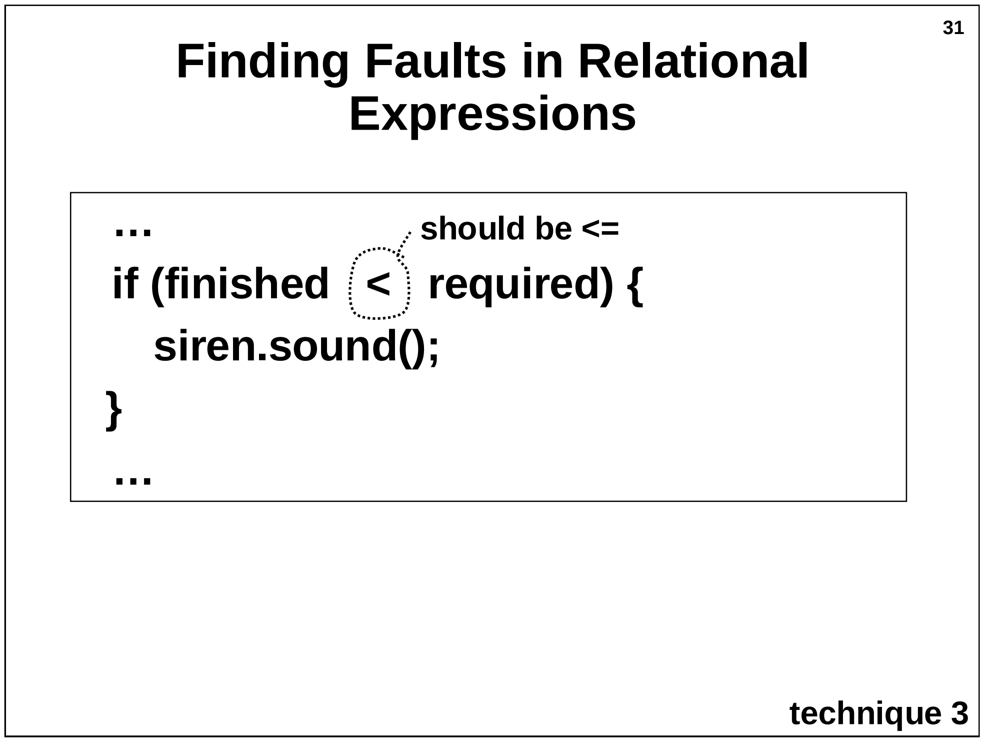
The reason is that the slide contains two bugs. In the IF statement, the siren is turned on, but never turned off. It should be.

This is a fairly common bug: one where the code just plain never works. In C, the canonical example of such a bug is this:

```
fprintf(“Error %d”, code)
```

The first argument has been omitted, so a string is treated as a destination stream.

Trying both values ensures that all lines of code in the program are executed, which triggers all such bugs. (Exception: this doesn’t force catch blocks to be exercised - we’ll talk about those later.)



Finding Faults in Relational Expressions

should be <=

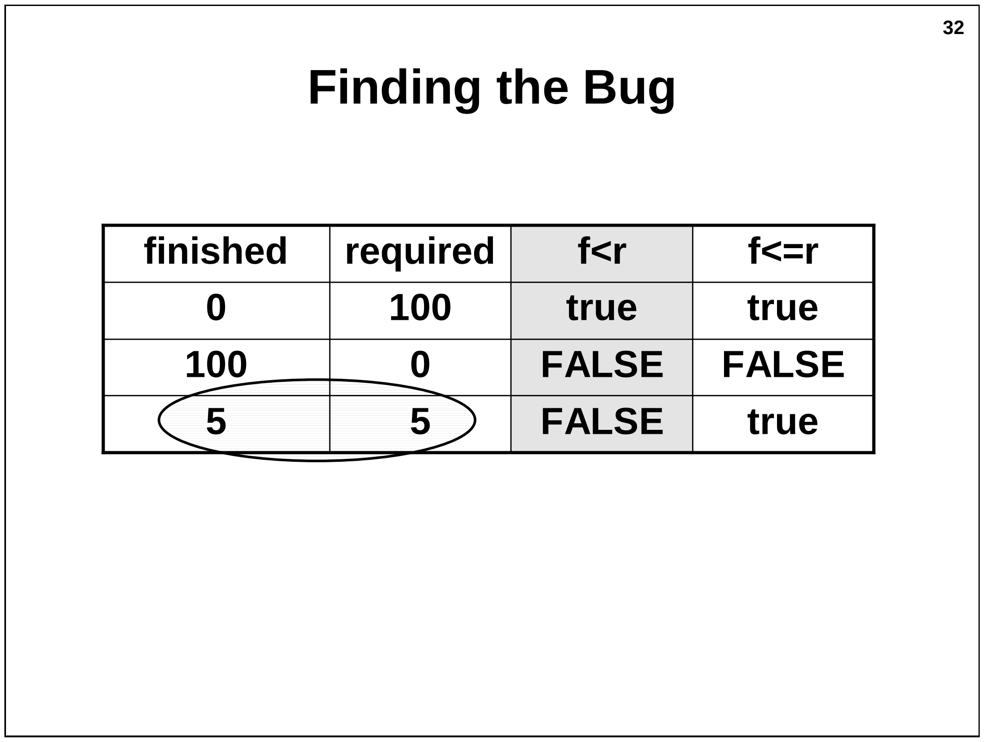
```
…
if (finished < required) {
   siren.sound();
}
…
```

technique 3

Let’s briefly move on to a related fault that the program helps with.

It’s common for programmers to misuse relational operators. In this example, the programmer typed “<“ when “<=“ would have been correct.

(Thankfully, iterators have made such faults much less common, but “off by one” errors have hardly disappeared.)



Finding the Bug

finished	required	f<r	f<=r
0	100	true	true
100	0	FALSE	FALSE
5	5	FALSE	true

The bug on the previous side can only be found when the left-hand side of the operator is equal to the right-hand side. For all other values, the code will do the right thing for the wrong reason.

reference

Testing Relational Operators

A<B, A>=B

$A = B - \varepsilon$

A = B

A>B, A<=B

A = B

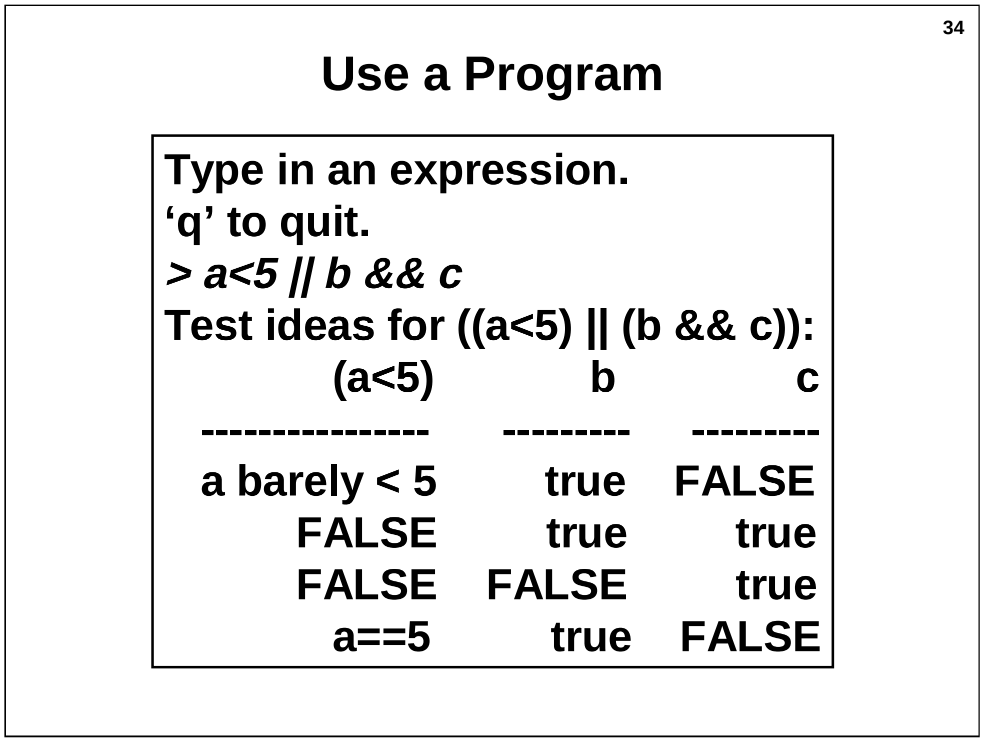
$A = B + \varepsilon$

Here are the ways you should test relational operators. ε is a very small value. If both sides of the expression are integers, ε should be 1. If one of the values is a floating point number, it should be a number quite close to zero. (It's probably not important that it be the absolutely smallest such number, but it should be much smaller than 1.)

So, for the expression

finished < required

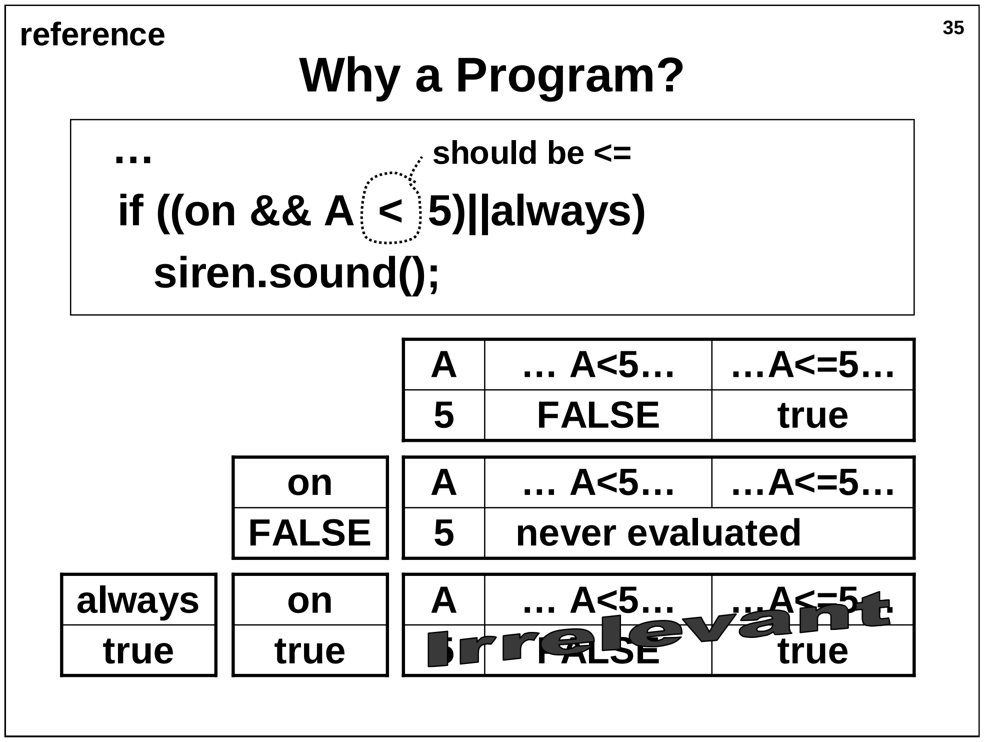
you should have a test in which FINISHED is a little bit less than REQUIRED (evaluates to true) and one in which FINISHED exactly equals REQUIRED (the false case).



Use a Program

```
Type in an expression.
‘q’ to quit.
> a<5 || b && c
Test ideas for ((a<5) || (b && c)):
           (a<5)          b          c
   ------------     ---------  ---------
   a barely < 5        true      FALSE
          FALSE        true       true
          FALSE       FALSE       true
           a==5        true      FALSE
```

My same program that generates cases for boolean expressions will also take mixed boolean-relational expressions.


reference 35

Why a Program?

```
 …                         should be <=
if ((on && A < 5)||always)
   siren.sound();
```

A	… A<5…	…A<=5…
5	FALSE	true

on
FALSE

A	… A<5…	…A<=5…
5	never evaluated	

always	on
true	true

A	… A<5…	…A<=5…
5	FALSE	true

Irrelevant

The program is useful because getting mixed boolean-relational operators right is a little tricky, as shown by the example on the slide.

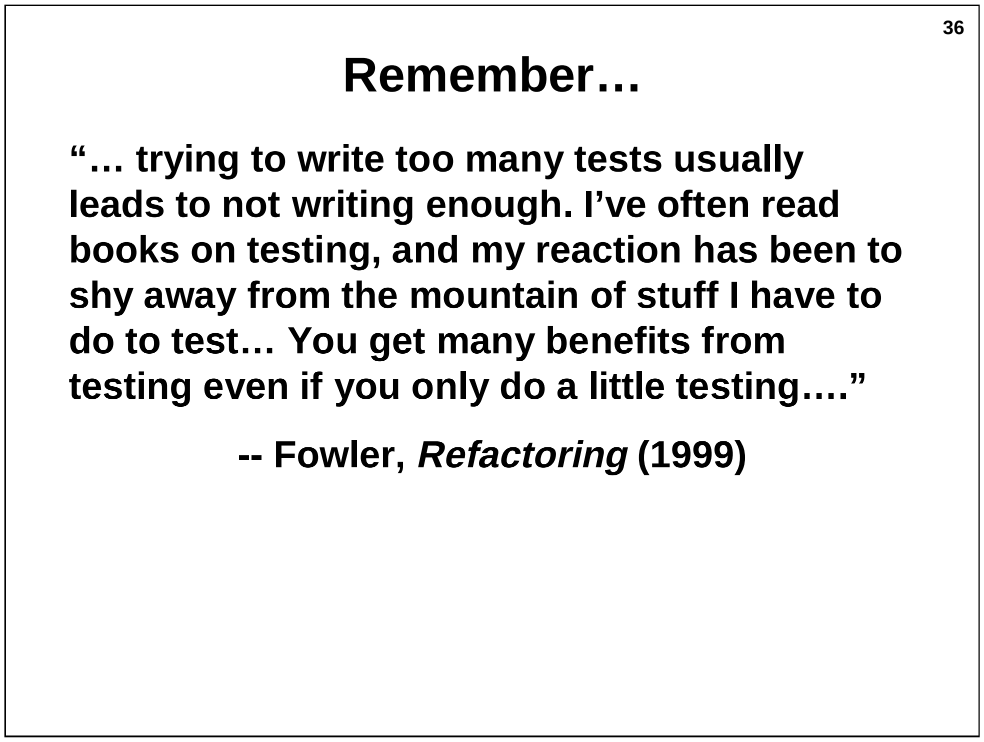
You might decide you want to write a test with A=5. But if ON happens to be FALSE, the relational expression will never be executed. So, if the code should read A<=5, it will never be presented with the value that would detect that. Bug not found.

Suppose, though, that ON is true. The expression will be evaluated. If the bug exists, it will yield the wrong value. It will produce FALSE when the correct code would produce TRUE. So far so good!

But what if ALWAYS is true? Then it doesn't matter what the first term in the || is - the whole expression will always be TRUE. The failure will be "damped out".

My program gives the right test ideas in such situations (unless it has a bug).

That is, the case in which A is 5 will always have ON be true and have ALWAYS be false. Even better, the expression will have the same number of test ideas (4) as an expression without the relational operator.



36

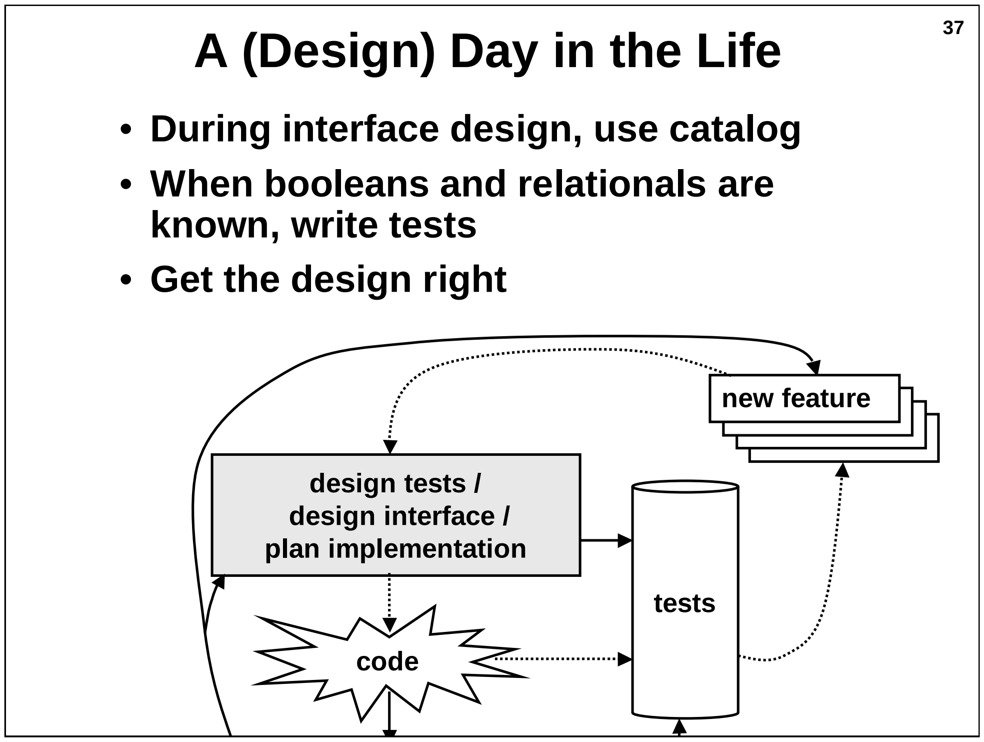
Remember…

“… trying to write too many tests usually leads to not writing enough. I’ve often read books on testing, and my reaction has been to shy away from the mountain of stuff I have to do to test… You get many benefits from testing even if you only do a little testing….”

-- Fowler, *Refactoring* (1999)

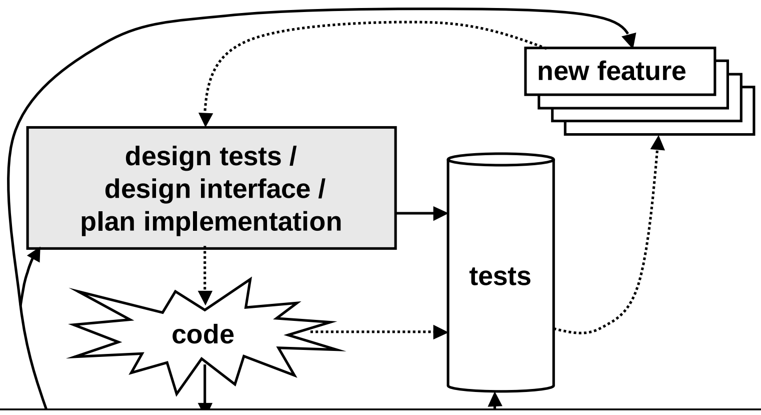
I’m worried that it won’t seem I’ve given you much: a one-page sheet of paper and a program to run to get a few additional test cases.

But I want to err on the side of not overwhelming you with work. Once you have a visceral understanding of the value of testing, you can add additional techniques.



A (Design) Day in the Life

- During interface design, use catalog
- When booleans and relationals are known, write tests
- Get the design right



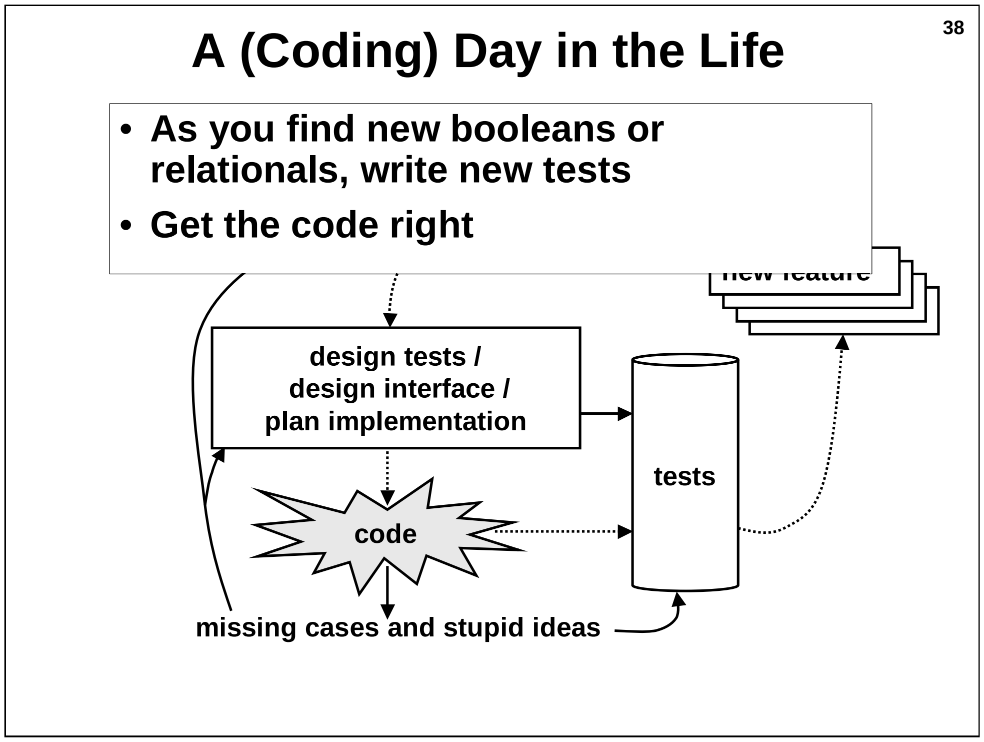
Now that we've seen some techniques, let me describe how I use test design in my own programming.

I don't claim that I always do exactly what I describe here. Sometimes I do things somewhat differently, sometimes for good reasons, sometimes for bad. It's accurate to say that my standard practice orbits around this.

I think about design and interfaces before coding.

I use the catalog (which I have memorized) to remind me of special cases that have to be handled in my design. If a method has to do something special when condition A and condition B are both true, I know three tests to write.

My goal during design is to use test ideas to prepare the way for coding.



38

A (Coding) Day in the Life

- **As you find new booleans or relationals, write new tests**
- **Get the code right**

new feature
design tests / design interface / plan implementation
tests
code
missing cases and stupid ideas

When writing code, I discover new booleans and relational operators. As a hoary example that's never actually come up, I might need to sort an array. I decide to use a quicksort if the array is big, an insertion sort if the array is small. That's not something that I considered during design, but I now have this boolean:

if (a.length < 15)…

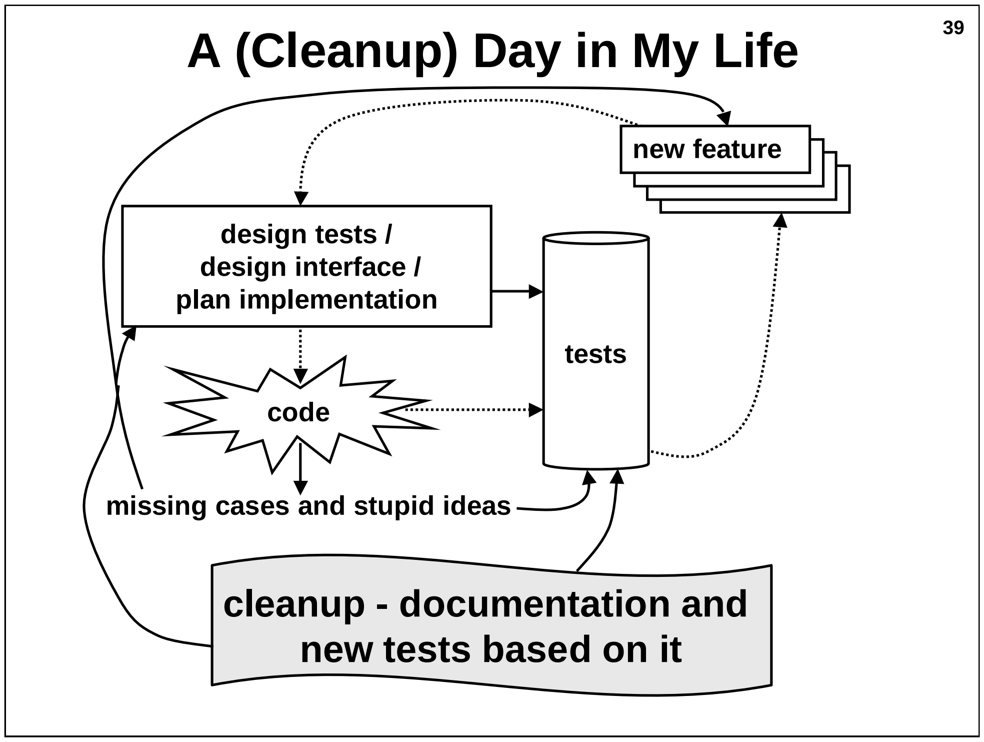
and two test cases:

array length 14

array length is 15

(The exact boundary doesn't matter in this case - there'd be no practical difference if I had intended to use <= and made a typo - but I might as well use it.)

Thinking about test cases helps me get my code right as I write it and discover new things.



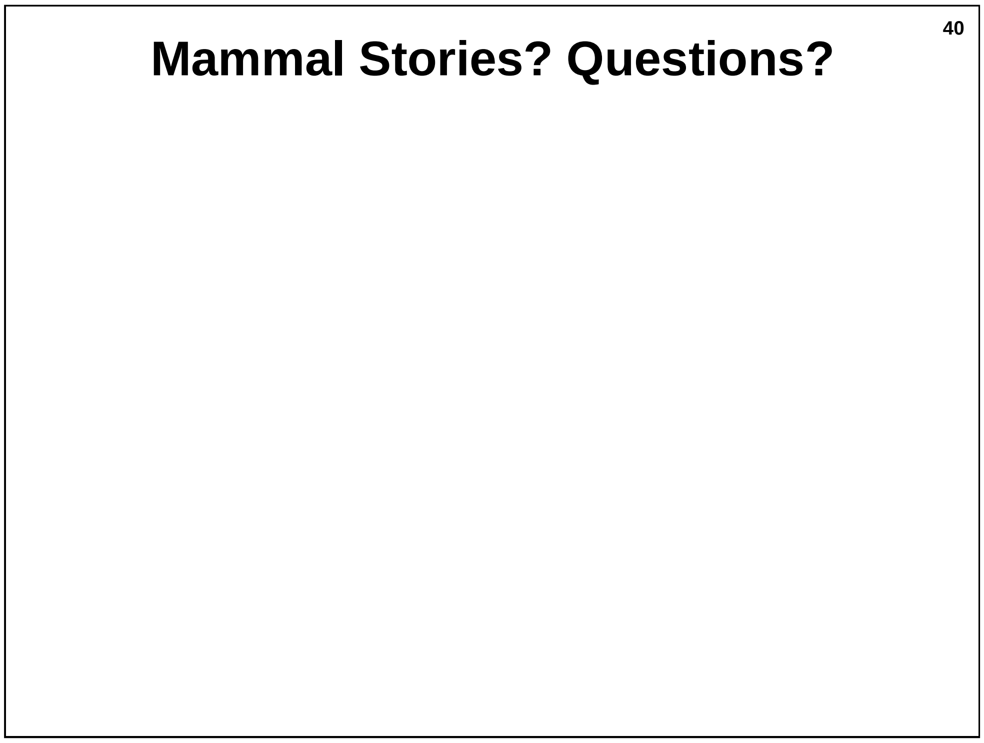
A (Cleanup) Day in My Life

39
new feature
design tests / design interface / plan implementation
tests
code
missing cases and stupid ideas
cleanup - documentation and new tests based on it

My brain is funny. I can't write decent comments while I code. "Coding mode" is just different than "explaining things to a reader mode".

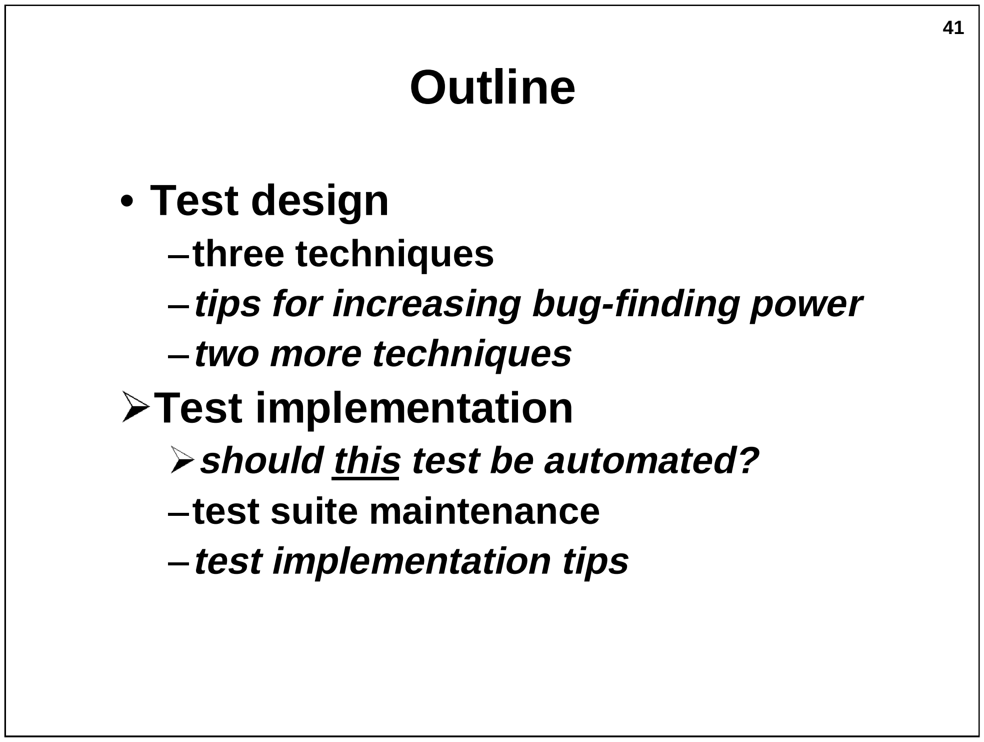
So, after coding, and after a pause to try to forget some of what I know is "obvious" about the code, I go through and do a cleanup pass. This pass also includes things like variable renamings and other refactorings. I'm not surprised when I discover new missing cases or other code I should add to improve things.

A part of adding code is adding tests for that code.



Mammal Stories? Questions?

“Mammal stories” is refers back to the dinosaur vs. mammal distinction from slide 2.

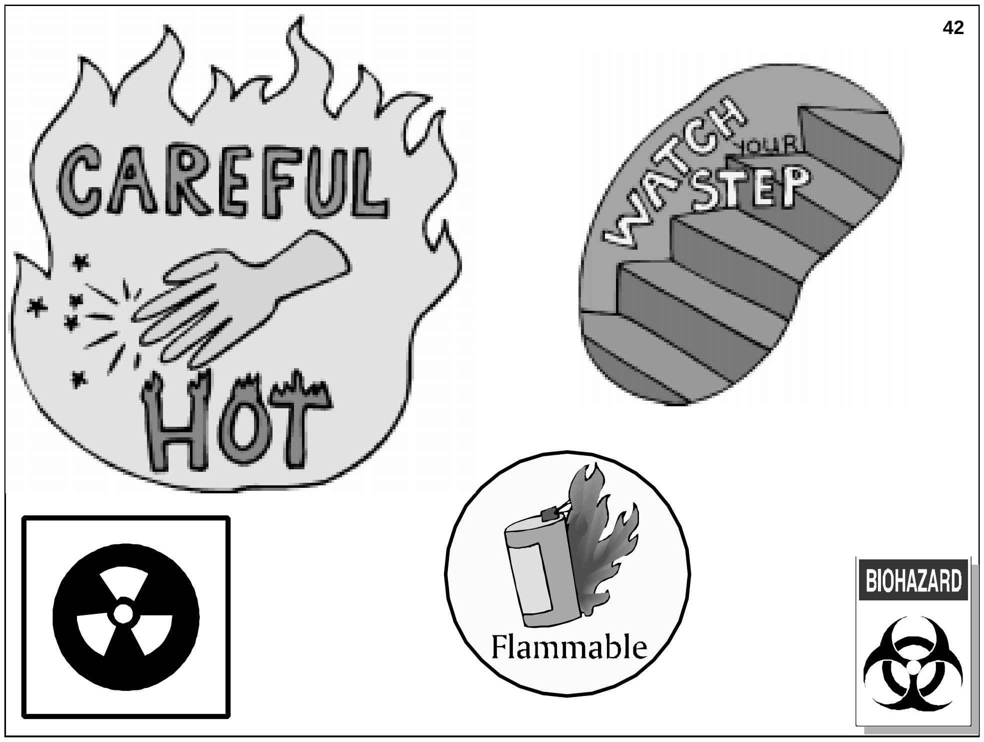


41

Outline

- **Test design**
  - –**three techniques**
  - –***tips for increasing bug-finding power***
  - –***two more techniques***
- ➢**Test implementation**
  - ➢***should this test be automated?***
  - –**test suite maintenance**
  - –***test implementation tips***

We'll now switch gears and talk about what to do with tests after they're designed. The first topic is how to decide whether a test should be manual or automated.



Most of the time, I'm depressingly uncontroversial. Even when I want to stir up a fuss, people don't get excited.

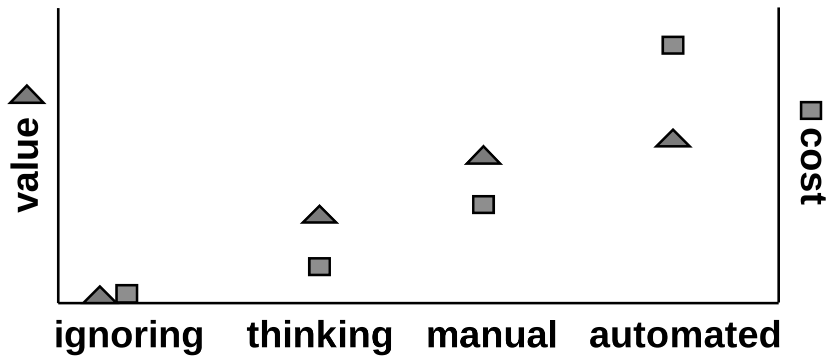
There is one way that I can get a rise out of people, though, and that's to talk about test automation. I do it by saying that many people try too hard to automate, and that a completely automated set of tests is probably a waste of effort.

If you bristled at that, please withhold judgement for a bit. Thanks.

(And please note that I did **not** say that "automated tests are a waste of effort". Every testing effort should include some automated tests.)

You can find a more detailed discussion of many of the points in the next two sections in my paper "When Should a Test Be Automated?", http://www.testing.com/writings/automate.pdf. That discussion is targeted to "black box" testers who are testing through the product's external interfaces, but the basic argument is more broadly applicable.

Tests Are Economic Entities



Automation value is often overestimated

Automation cost is often underestimated

A given test has a cost and provides some value.

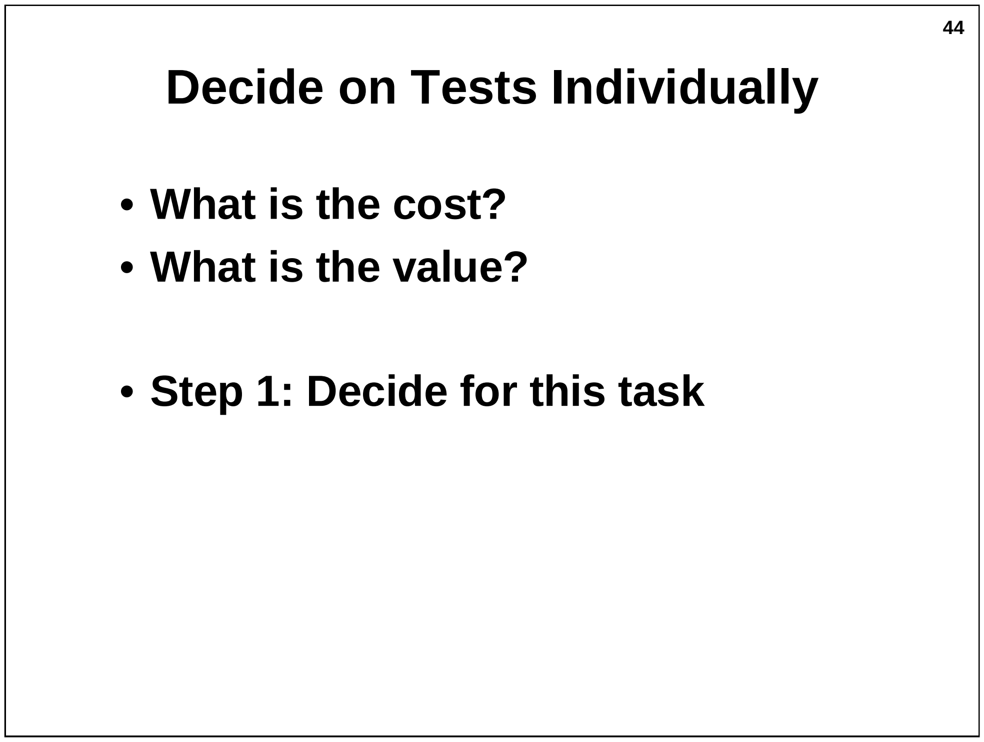
Sometimes a test costs more to automate than it would to run it once. For example, if you have a GUI available, it may be easier just to poke values into the GUI than to write a chunk of Java code to call methods with the same values. Other times, automating a test is cheaper than running it manually. Load testing is the canonical example. How do you want to test your web site? Rent a stadium, fill it with people, then use the public address system to tell them all to click at the same time? Or use a load testing tool?

The value of an automated test is always more than the value of a manual test. (Well, there are exceptions, but those are, I think, not relevant to programmer testing.)

But it has to be enough more to exceed the additional cost of automation, if any. (If there’s no additional cost to automation, by all means, automate.)

(Note that it can be the case that both manual and automated testing are too expensive - in that case, you just think. Perhaps you walk through the code by hand.)

I think people often overestimate the incremental value of automated tests. They also underestimate the cost. Thus, they over-automate.



44

Decide on Tests Individually

- **What is the cost?**
- **What is the value?**

- **Step 1: Decide for this task**

Okay, that’s all very well. Some tests should be automated and some should not. But which? How do you make the decision for a specific test?

Here’s how I’ve started to think about it. I’ve changed my approach somewhat these days, under the influence of Extreme Programming and testing colleagues like James Bach (www.satisfice.com) and Cem Kaner (www.kaner.com). I certainly hope my thinking continues to evolve. Until then, I offer this for your consideration.

First, I want to break the decision down into subdecisions. The first is to decide whether automation will help me *in this task*, with what I’m doing now. Will it help me finish implementing today’s feature?

The second subdecision involves worrying about how automation will be useful in future tasks.

Each subdecision is a balancing of cost and value.

(Note: if you read the “When Should a Test Be Automated” paper, you’ll find a somewhat different approach. I still think that one is valid, mainly because the context - GUI test automation - is different.)

Cost Differential of Automation?

- **How much more effort will it be to automate than to run manually?**
  - **manipulating linked structures?**
  - **popping up browsers?**

Present cost is pretty easy to estimate.

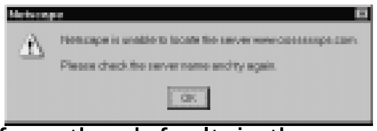
If, as in the running example, the code being tested manipulates simple linked structures, there's no reason not to automate. Even if you're working in a programming environment with an interpreter (like Lisp or Smalltalk), it is usually only negligibly harder to type test statements into a jUnit format than it is to type them directly into the interpreter and look at the printed result.

On the other hand, suppose that your feature pops up a browser. There are various things to check:

• if there's an old browser window running, does it use that instead of creating a new one?

• if the old window is obscured, is it raised to the top?

• if the browser is displaying a modal dialog, does the code behave correctly? (What *is* correct in this case?)



Netscape

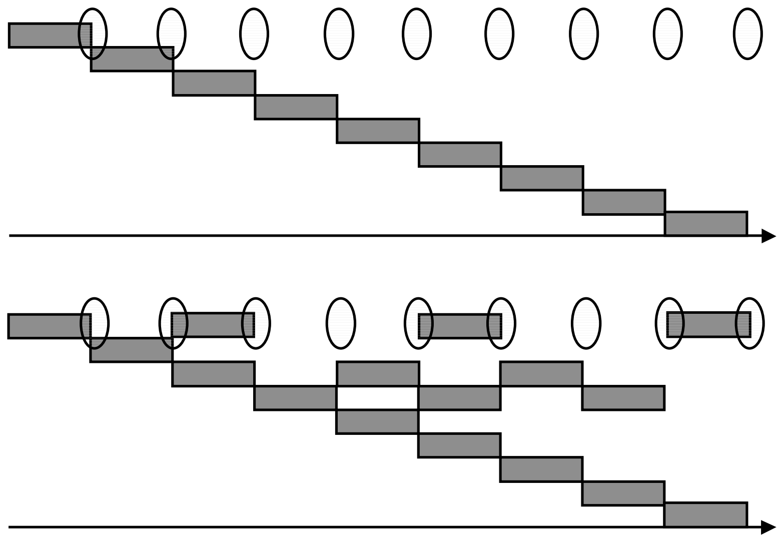
Netscape is unable to locate the server [illegible].

Please check the server name and try again.

• if the preferred browser has been changed from the default, is the correct browser invoked?

These are all pretty trivial to test manually, but much harder with an automated test. (I suppose… I don't know anything about invoking browsers, just that I've seen products botch some of these things.)

Expected Value of Rerunning the Test Often?



What will be the value of rerunning a newly-created test while you work on your current task? It depends on how much the code will change during the task. Let's look at two extreme cases.

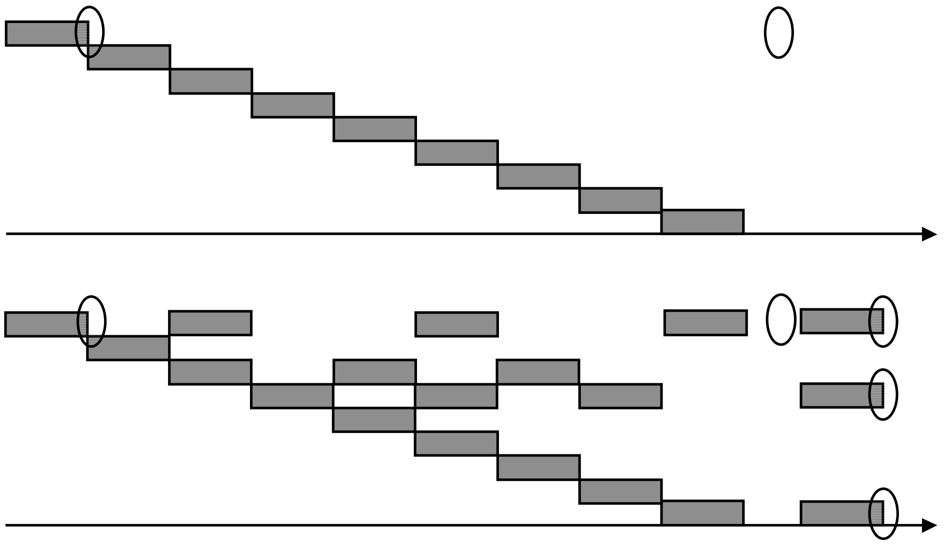
The first timeline shows a very discrete coding process. A first chunk of code is written, as are its tests. Then another chunk is written, and another, and another. During all this time, the first chunk's tests are run again and again and again. **If** that chunk of code doesn't change, nor does any of the code it depends on, running those tests adds no value. It can't. The same inputs exercise exactly the same code, so they will always give exactly the same result. That result must always be a "pass" (since you don't move on from the first chunk of code until all the tests pass).

In the second timeline, that first chunk of code changes several times. The changes are required by the writing of other chunks of code. Since the tests for the first chunk are now executing changed code, they may have value - they may find bugs.

So predicting the value of a test boils down to predicting how much the code being tested, and the code it calls, is going to change during this task.

47

What Do You Lose by Running the Manual Test Late?



The alternative to an automated test is a manual test. (I include in “manual test” all forms of semi-automated tests, ones that are partially manual.)

A manual test for a chunk of code will be executed much less often. It’s quite likely it will be run only twice.

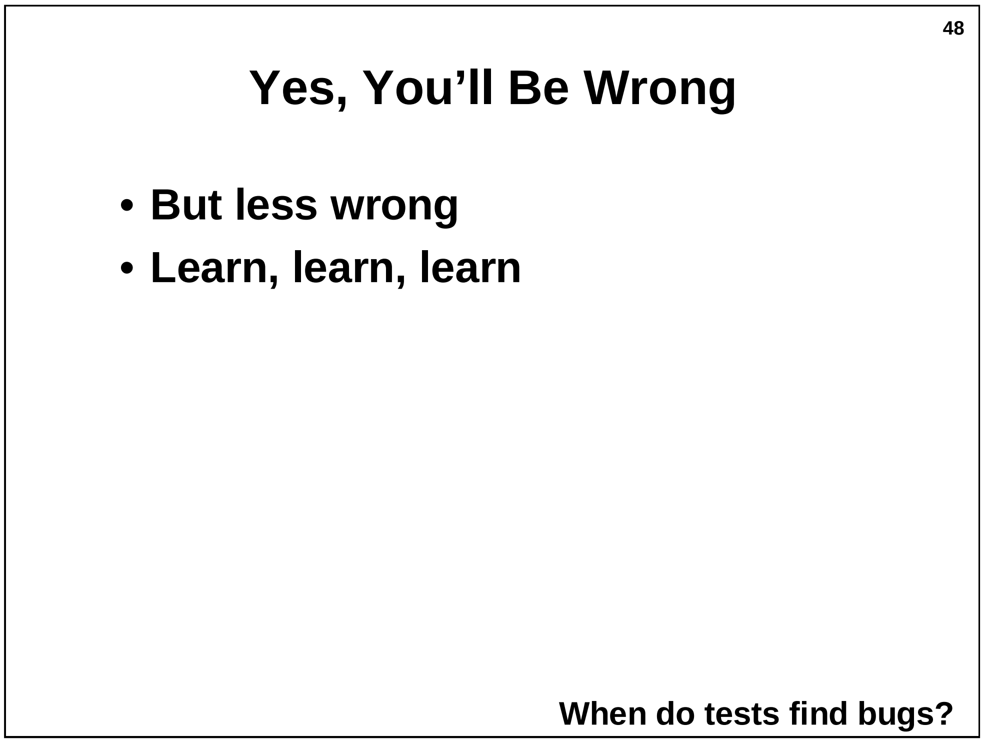
• The first time is when the chunk of code is originally written.

• The second time is when the task is finished and the code is being integrated. All the manual tests are run one more time, just to see if anything’s been broken since the first time they ran.

In the first scenario, the second run of the manual test won’t discover any bugs. The chunk of code hasn’t changed, nor has any code it depends upon. You lose nothing by not running tests often.

In the second scenario, there’s a much larger chance that the second run will find a bug. Then you have to fix the bug, which will be harder than if you’d caught it earlier. And you have to run the test against the fixed code. Further, it might be prudent to rerun all tests for all code that depends on your fixed chunk. Certainly you’d rerun those tests if your fix required other code to change to match.

Not having automated tests is costing a lot of time. It’s looking now like paying the extra cost for automation would have been well worth it.



Yes, You’ll Be Wrong

- But less wrong
- Learn, learn, learn

When do tests find bugs?

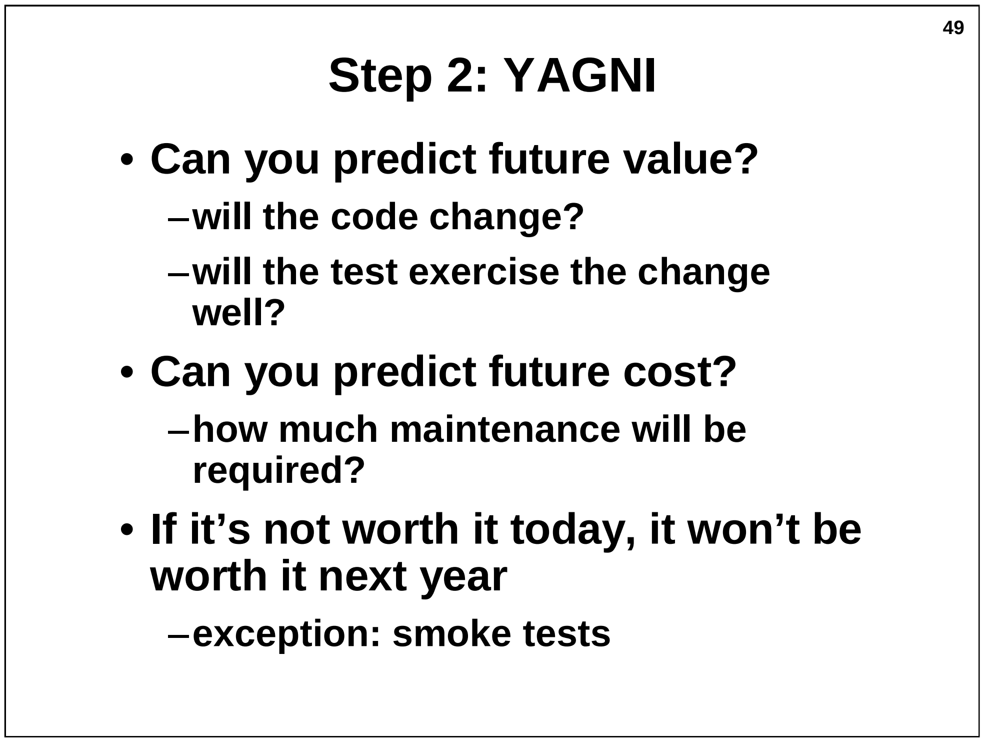
It’s hard to predict what will happen for a particular task. Will you really know whether some chunk of code will change a lot or not at all?

Perhaps not. But it’s my faith that you’ll make better decisions than if you don’t think about the tradeoff, that if you adopt the default strategy of “automate everything, except when it’s too hard to stomach”.

Mistakes won’t be fatal, after all. If the code starts to change a lot, now you can automate the tests. If you automated when you didn’t need to… oh well, the tests aren’t doing any harm.

Over time, I suspect you’ll get better at making the decision as you watch and see which tests actually prove useful.

Just a note here. Most of the value of automated tests is gotten the first time they’re run. That is, a test is much more likely to find a bug that first time than in any automated repetition. Even in projects with a large automated test suite, rerunning automated tests finds the minority of bugs. Most are found by new tests. I describe the results of a survey I took at http://www.testingcraft.com/regression-test-bugs.html



Step 2: YAGNI

- **Can you predict future value?**
  - **will the code change?**
  - **will the test exercise the change well?**
- **Can you predict future cost?**
  - **how much maintenance will be required?**
- **If it’s not worth it today, it won’t be worth it next year**
  - **exception: smoke tests**

What about future value?. The future is hard for me to predict, so I won’t try.

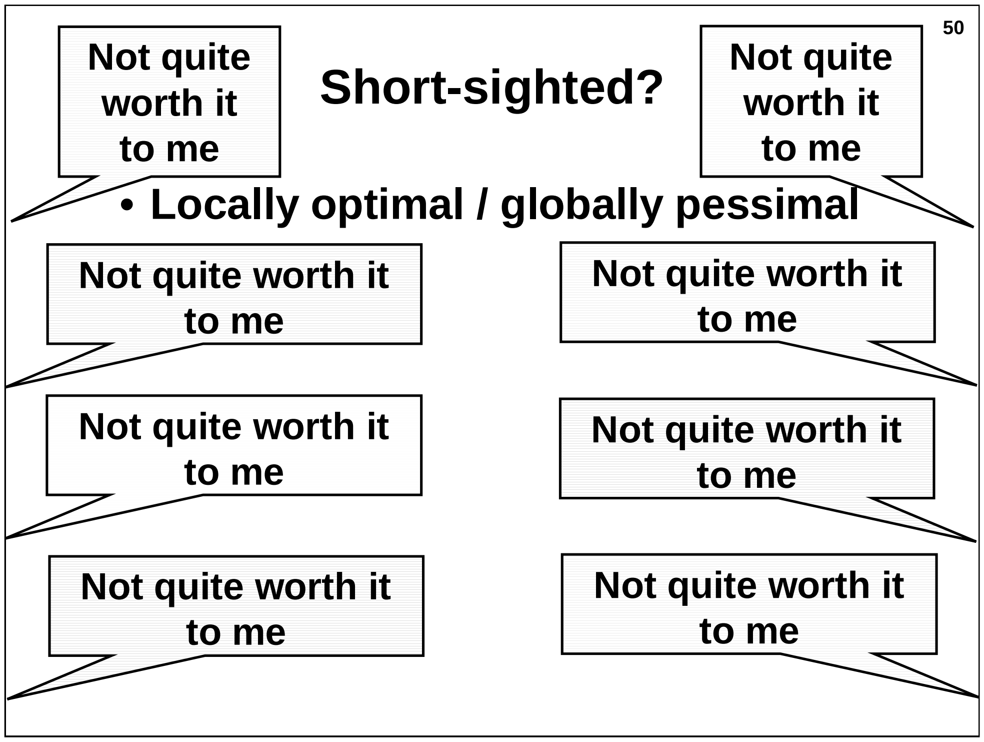
There’s a principle in Extreme Programming called YAGNI (“You Aren’t Going to Need It”).
http://www.xprogramming.com/Practices/PracNotNeed.html

It advocates *not* writing code because you think you’ll need someday. When someday comes, you’ll probably need something different. Instead, write only the code you know you need for *this* task.

Let’s extend the principle. Suppose you have a test that wouldn’t be worth automating for your current task, but which, well, might find some bugs someday. If you don’t need it now, you aren’t going to need it then. Tests have value if they exercise changed code - but you can’t predict what changes are coming.

The dominant long-term cost of an automated test is maintenance. Should you try to predict the cost of future maintenance? To do that, you must predict how the code will change next year, and how those changes will affect tests. You’ll be wrong often enough that it’s not worth trying.

I buy this XP analogy, mostly. However, I think it’s always useful to have some broad but shallow tests for basic function. These are commonly called “smoke tests”. (By analogy to electronics: if you plug it in and smoke pours out, no need to test further.) These simple smoke tests I think will mostly suffice for refactoring. (Refactoring is preparatory maintenance; restructuring the code to support changes you’re about to make to it.)



50

Short-sighted?

- Locally optimal / globally pessimal

Not quite worth it to me
Not quite worth it to me
Not quite worth it to me
Not quite worth it to me
Not quite worth it to me
Not quite worth it to me
Not quite worth it to me
Not quite worth it to me

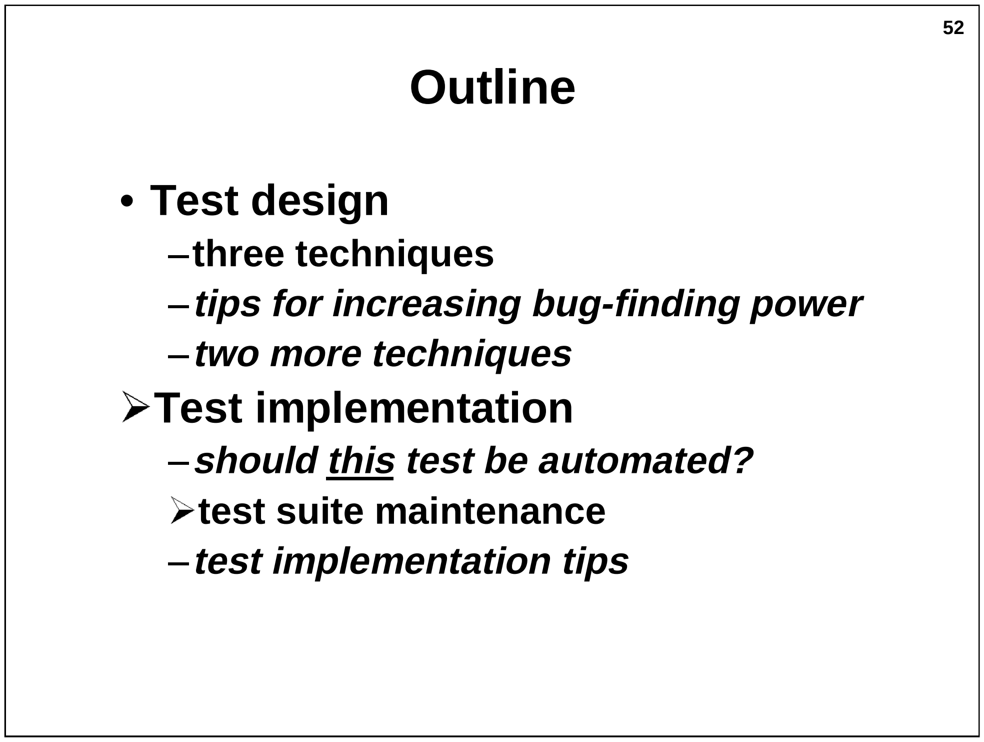
If something is difficult to automate, that’s because automation requires substantial support code. That support code might be useful to someone else automating *their* tests. Is focusing only on value to yourself too short-sighted?

Put another way: a problem with this approach is that it alone would never have led to the jUnit framework. No one person would be justified in inventing a really good framework just for her own tests. But, without the framework, many fewer tests would be justified. (This is the “if everyone acted as if their vote didn’t count…” problem.)

No argument. However, building a framework that’s not justified for this set of tests, but that would be worthwhile for other programmers or later sets of tests, is properly a feature request. When you’re working on your feature, concentrate on it. Don’t spend time on infrastructure tasks that won’t pay off now. Get them scheduled as separate development tasks, maybe one you do before starting this task.

Mammal Stories? Questions?

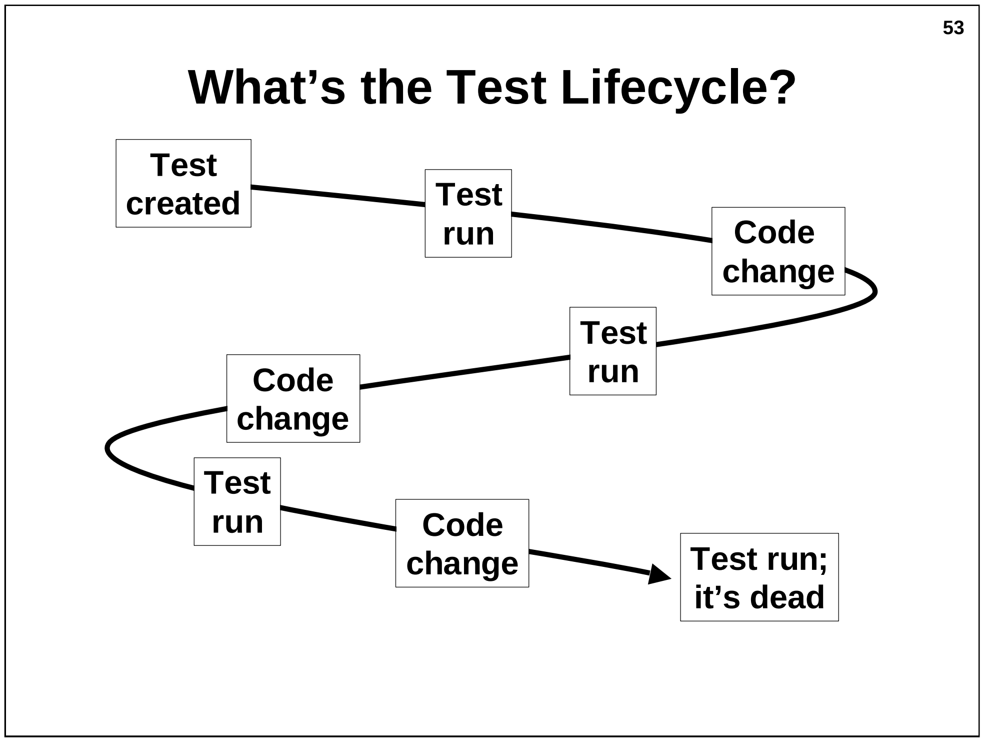
Let’s talk about people’s experience with test automation.


52

Outline

- **Test design**
  - **three techniques**
  - ***tips for increasing bug-finding power***
  - ***two more techniques***
- ➢ **Test implementation**
  - ***should <u>this</u> test be automated?***
  - ➢ **test suite maintenance**
  - ***test implementation tips***

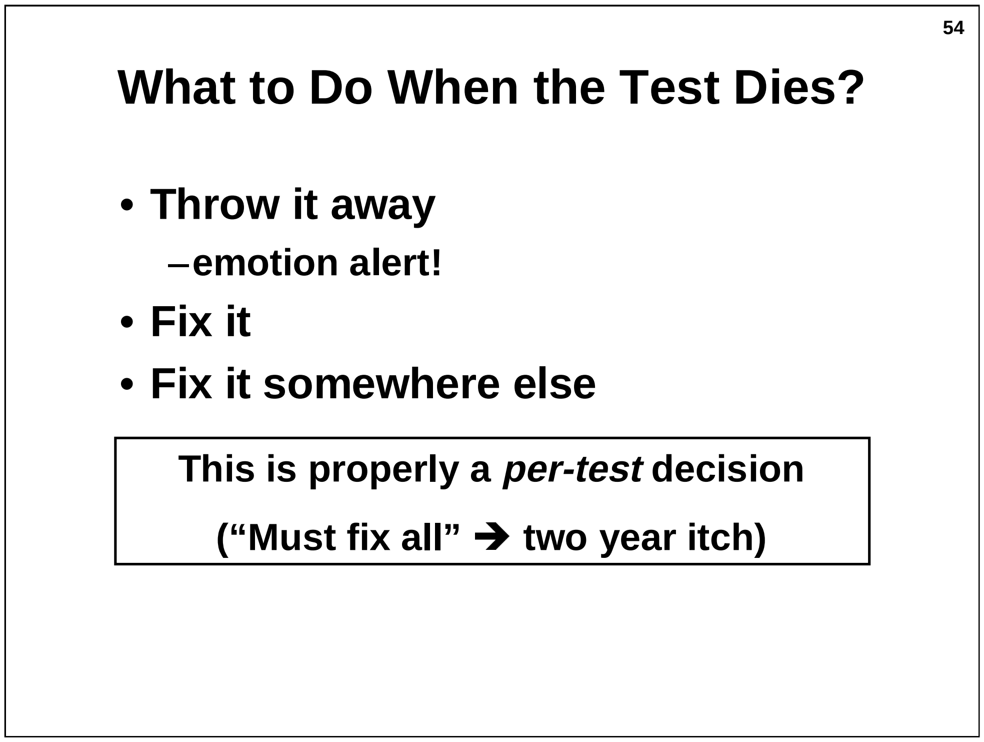
So you've written some automated tests. What happens to them now?



Tests have the opportunity to justify their existence only when the code they execute changes. That's when they can find a new bug. The code they execute can be the code that you intended to test, or code that it calls, or code that *that* code calls, etc.

At some point, one of the code changes could break the test. That is, the code behaves differently than it used to, but the different behavior is intentional, not a bug. Since the test expected the old behavior, it flags a failure. To fix the test, you'd have to change it to expect the new behavior.

At that point, I call the test "dead". The question is what to do with the corpse.



What to Do When the Test Dies?

- **Throw it away**
  - **emotion alert!**
- **Fix it**
- **Fix it somewhere else**

This is properly a *per-test* decision

(“Must fix all” ➔ two year itch)

There are three options.

• You can throw the test away. My observation is that many people are overly reluctant to do that. (I certainly am.) That is, a person who would not choose to spend the time to write the test in the first place will, as a matter of course, spend the *same* amount of time to fix it. The effort to create the test is, in economic terms, a “sunk cost” (*) and should have no effect on future decisions.

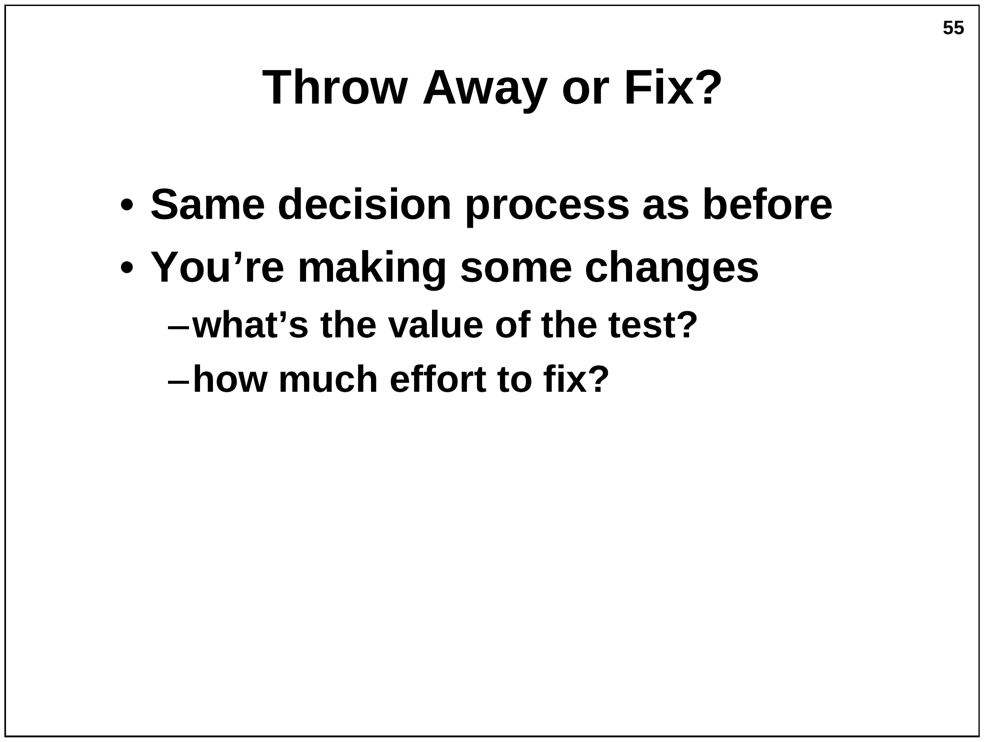
• You can fix the test.

• You can move it somewhere else. (I’ll explain what this means later.)

It’s important that you made the decision separately for each test. I speculate that the feeling that you either need to fix *all* of the tests or give up on *all* of them leads to what I call “the two year itch”, by analogy to the “seven year itch” that is supposed to afflict married couples. After some time, all the effort of maintaining a relationship seems like too much. So there comes a day when the programmer gets a bunch of broken tests, looks at them, contemplates the work needed to fix them, sighs, and sets the whole suite aside to “fix when I have time”. That, my friends, is all she wrote. Those tests are gone for good. I’ve seen this a lot.

(*) A definition of sunk costs: http://www.microeconomics.com/essays/cost_def/cost_def.htm#Sunk

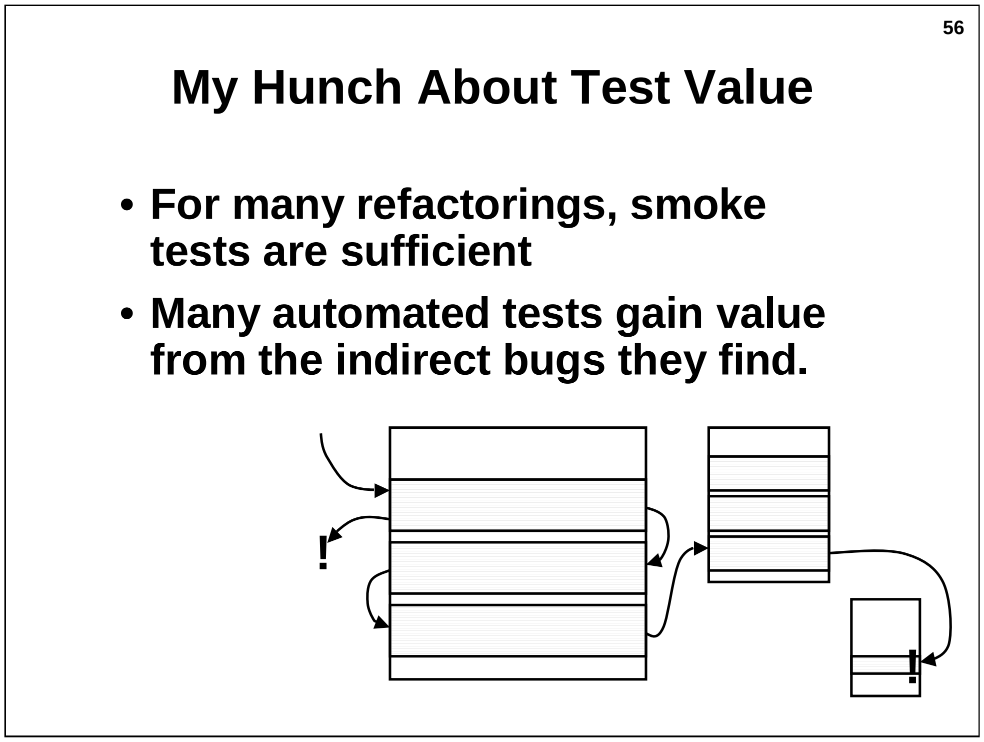
Some discussion: http://www.best.com/~ddfr/Academic/Price_Theory/PThy_Chapter_13/PThy_Chapter_13.html


55

Throw Away or Fix?

- Same decision process as before
- You're making some changes
  - –what's the value of the test?
  - –how much effort to fix?

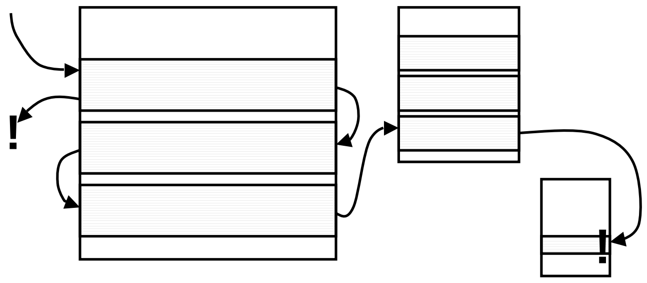
You should reason about whether to throw the test away or fix it in the same way as before. You're making some changes to some code. Most likely it's the code the test was written for. Or it could be deeply buried code, and the changed behavior happens to show through to the interface the test exercises. In either case, you're performing some task. Will the test provide value for that task that exceeds its cost?



56

My Hunch About Test Value

- **For many refactorings, smoke tests are sufficient**
- **Many automated tests gain value from the indirect bugs they find.**



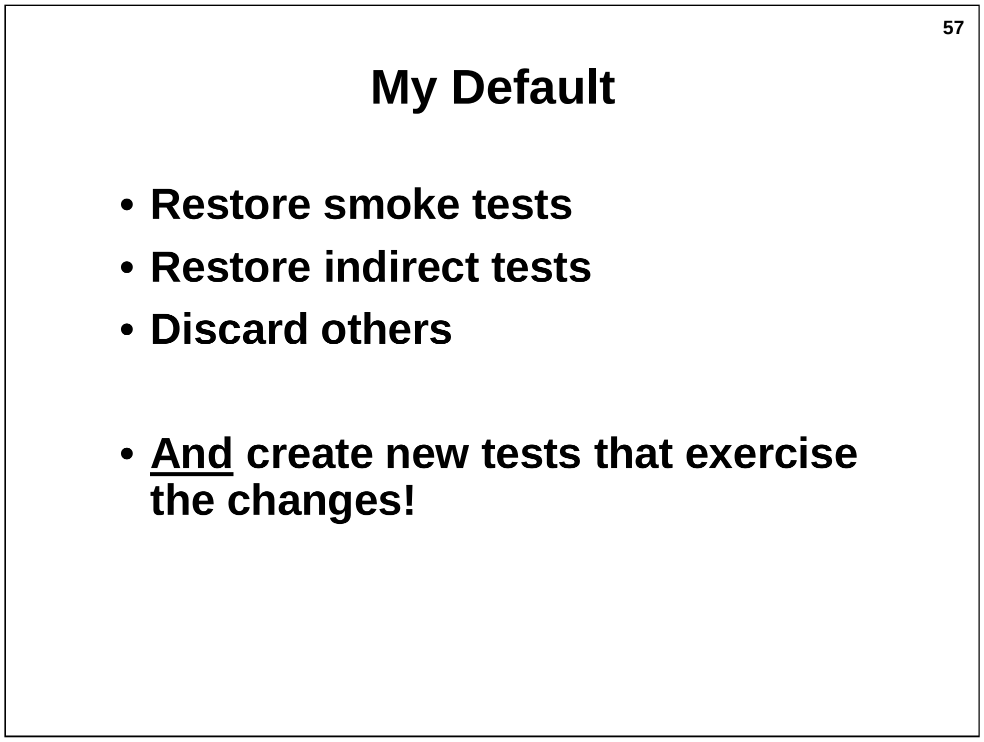
Remember that smoke tests are the broad but shallow tests that are basically sanity checks: does it work at all?

My hunch is that for refactorings - changes of code that are intended to leave behavior alone - the smoke tests will catch most of the bugs. (I’d welcome discussion and experience reports from those who are expert in refactoring.)

I further believe that much value from automated tests comes indirectly. It seems that many automated tests find bugs that don’t have anything to do with the reason they were written. That is, the test was written to satisfy a test idea (“will it work with an empty collection?”), but the bug is due to some chance combination of values passed to called methods that provoke a problem deep in some support code.

Chriss Agruss of Autodesk analysed a large number of bug reports. He concluded that perhaps up to half the bugs (from tests run through the user interface) were in support code, not in the code directly tested. (Personal communication)

Other than that, I don’t have any strong evidence for my claim.



My Default

- **Restore smoke tests**
- **Restore indirect tests**
- **Discard others**

- **<u>And</u> create new tests that exercise the changes!**

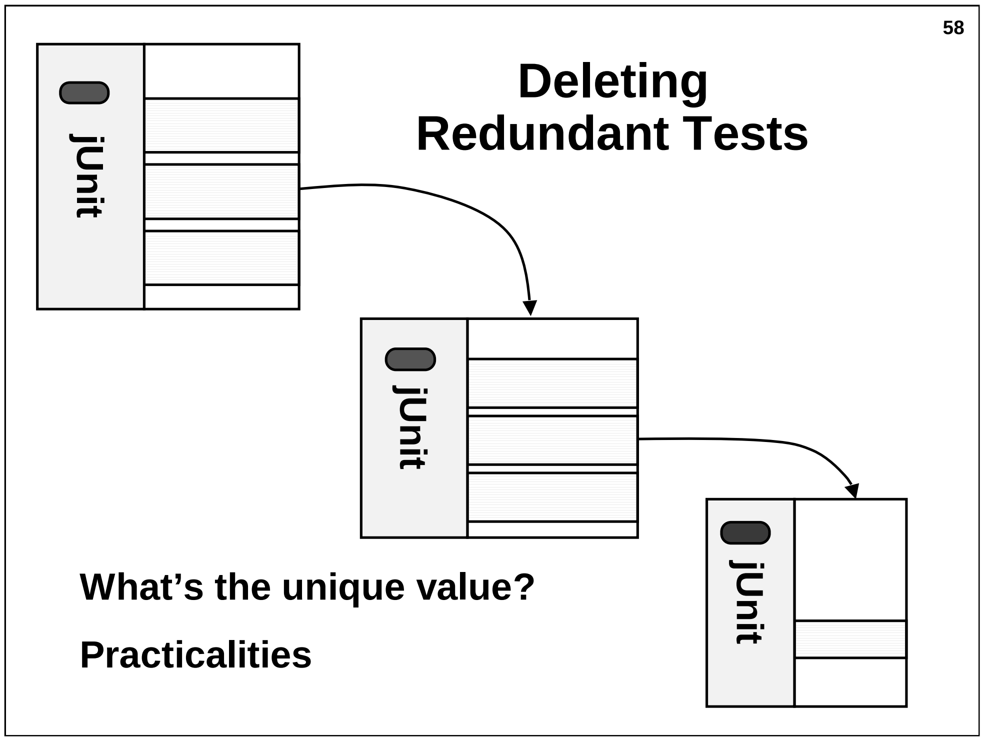
So my default inclination is to fix up smoke tests.

And I’d be much more likely to fix a test that exercises support code than one that checks whether the sqrt method correctly rejects numbers less than zero (assuming the error check calls no support code). My justification is that I know the error-handling code works now. If I ever change it for some reason (maybe to change the error message), I’ll quickly create a new test for that change. Until then, it’s unlikely that other changes would inadvertently break that error check.

Which tests exercise support code in new and interesting ways? They’re ones that have complexity and variety. That’s a later topic: see the “tips for increasing bug finding power” subsection of Test Design Reference.

As for the rest of the tests… I fight my pack-rat behavior (you should see my basement) and throw them away. It’s hard.

But please remember that all changes to behavior come with their own tests. So my test suite doesn’t eventually shrink down to near-nothing. It’s continually replenished with tests that I know are important. I’d rather spend time on those tests than on refurbishing tests that I think might come in handy someday.



Deleting Redundant Tests

What's the unique value?

Practicalities

There's another reason you might delete a test. The class with the broken tests is perhaps used by another class. That class also has tests. Some of its tests also exercise the class with the broken tests.

It's even possible that tests from several classes away exercise the class under test.

So perhaps the broken tests are redundant. Should you fix a test that provides only redundant value?

My tendency is to delete broken tests that I know are redundant.

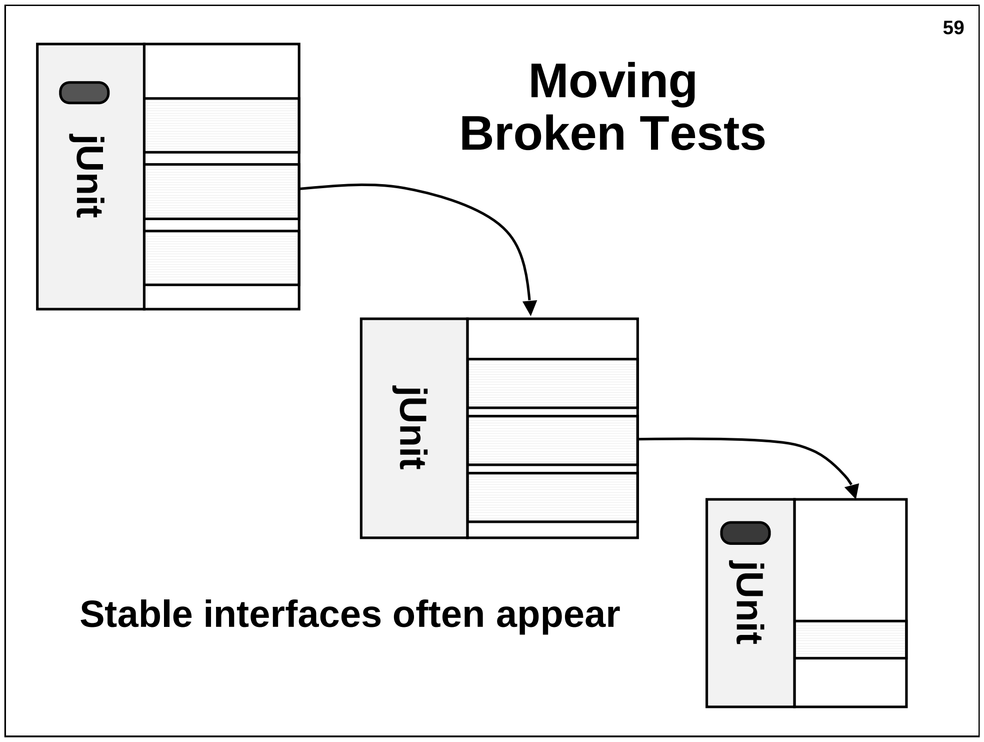
Now, there are some practicalities here.

• Do you *really* know the test is redundant? Maybe that test on the top left class looks as if it ought to exercise the lower-right class, but it really doesn't. By not fixing the broken lower-right test, you're increasing the chance that it's not well tested.

• Even if the top-left class's test exercises the lower-right class today, changes to it - or to any class along the way - could mean that it no longer does.

• Debugging is harder when the tests find a bug in the lower-right class. Faults are not as isolated.

These are risks you have to decide whether to take. I've taken them in the past and not had horrible shocks, so I intend to keep taking them. (Note that the systems with which I took the risks didn't have a huge amount of internal change. The greater the change, the greater the risk of bullet two mattering.)



Moving Broken Tests

Stable interfaces often appear

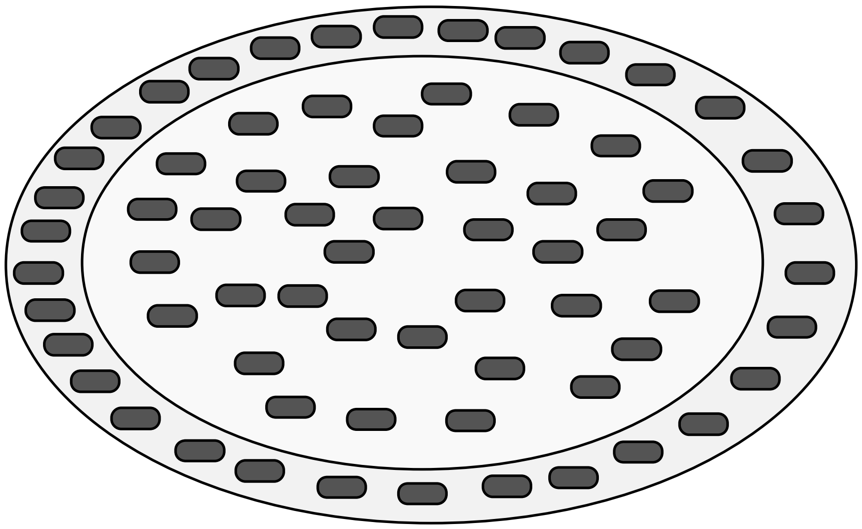
Systems often either start with or grow stable interfaces. The classes in those interfaces have public methods whose signatures don't change and whose behavior stays the same (or grows by addition, which should not break existing tests).

This is especially true when those interfaces become published APIs that programmers outside your organization use.

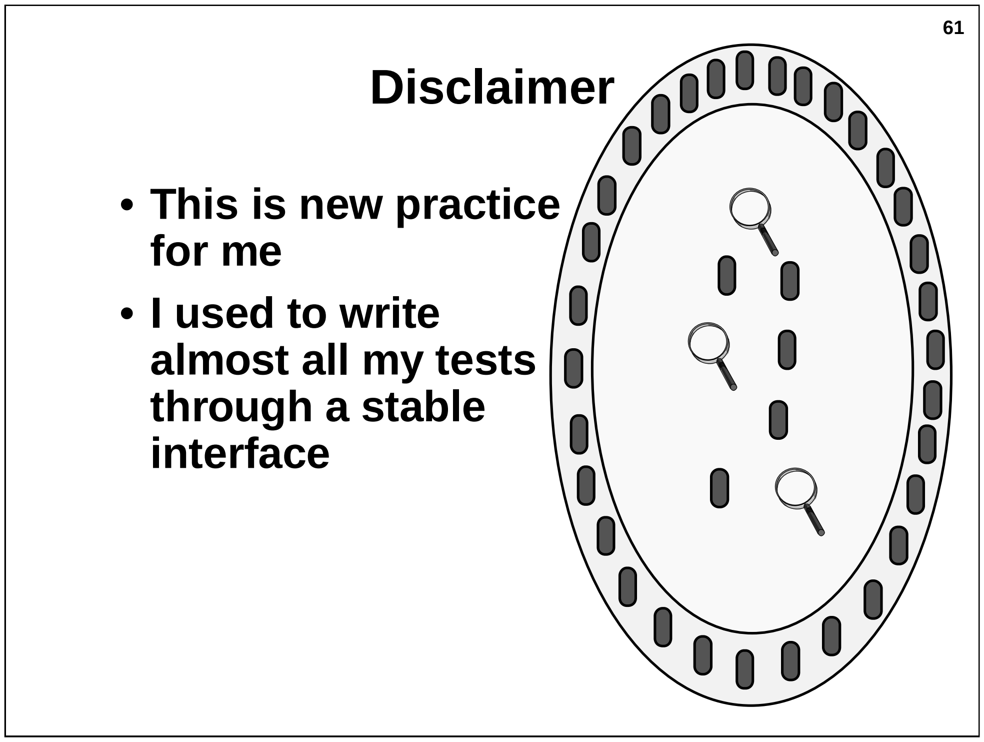
Up to now, I've been XP-ish in my reluctance to make predictions about the future. But I will predict that a stable interface will stay stable. More precisely, its tests will break less often than tests for classes behind the interface.

Therefore, if I can "promote" a broken test so that it talks to methods in the stable interface, I'll do that. It reduces the chance that I'll be fixing that test again… and again… and again...

The Steady State



So my vision of the long-term history of a product test suite is that old tests will stay where they are if the classes they test never change and break them. As tests break, they may migrate out to a stable interface. As those tests migrate out, new tests are created to take their place. Those tests are ones specifically written for the changes being made.



Disclaimer

- This is new practice for me
- I used to write almost all my tests through a stable interface

Now I should confess that I have not personally lived with a system that did this. In the past, I was preemptive. I first identified (and built) a stable interface. Then I implemented my features, always writing my tests against that stable interface.

I tried to have as few tests as I could inside the interface. I restricted internal tests only to code that was really hard to exercise through the stable interface.

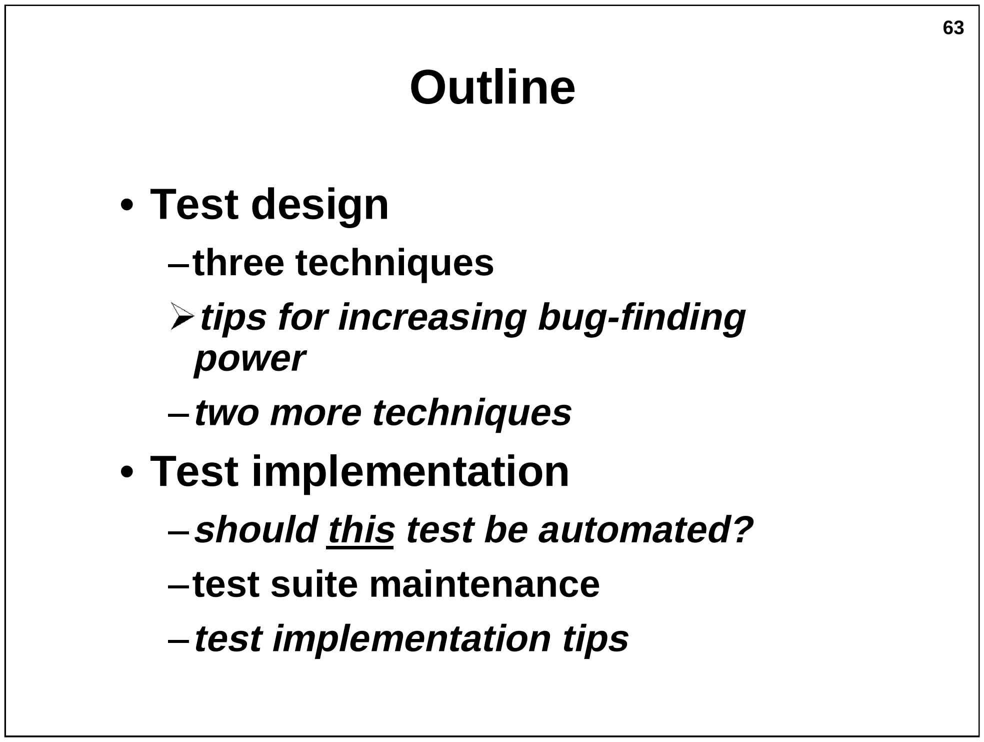
Some code is easy to execute from the external interface, but it’s hard to judge from there whether the code behaved correctly or not. Typically, it makes some change to a big internal data structure whose state is not readily visible.

Rather than write internal tests, I preferred to deal with that by providing “test points” or “viewports” into the code that allow the tests to query the state of those structures. (The impolite term for this is “breaking encapsulation”.) That made the tests somewhat vulnerable to changes in the interior, but not as vulnerable as purely interior tests.

There’s a bit more on breaking encapsulation later in the course.

Why have I switched? I realized I was afraid of change. I hated the thought of the “wasted” work of updating tests. And I was too intent on saving all my tests - so I updated them, even if I should have thrown them away. I’m more comfortable these days with the need to redo things, less worried about getting anything right the first time.

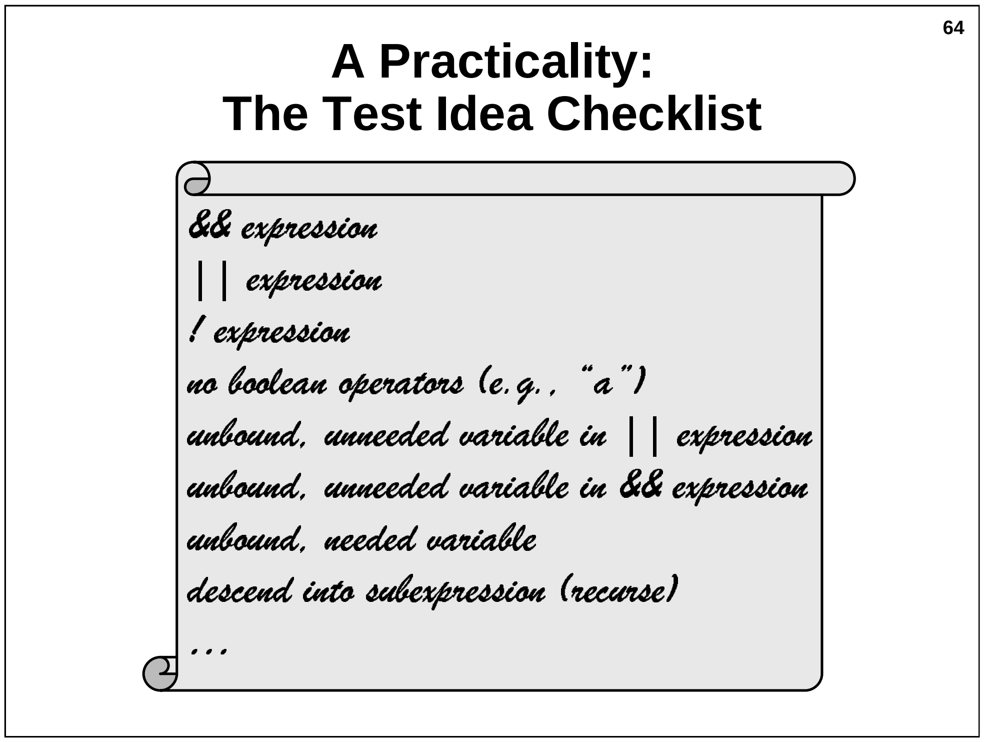
Mammal Stories? Questions?



Outline

- **Test design**
  - –**three techniques**
  - ➢***tips for increasing bug-finding power***
  - –***two more techniques***
- **Test implementation**
  - –***should <u>this</u> test be automated?***
  - –**test suite maintenance**
  - –***test implementation tips***

We're now in the reference portion of the tutorial notes.



A Practicality: The Test Idea Checklist

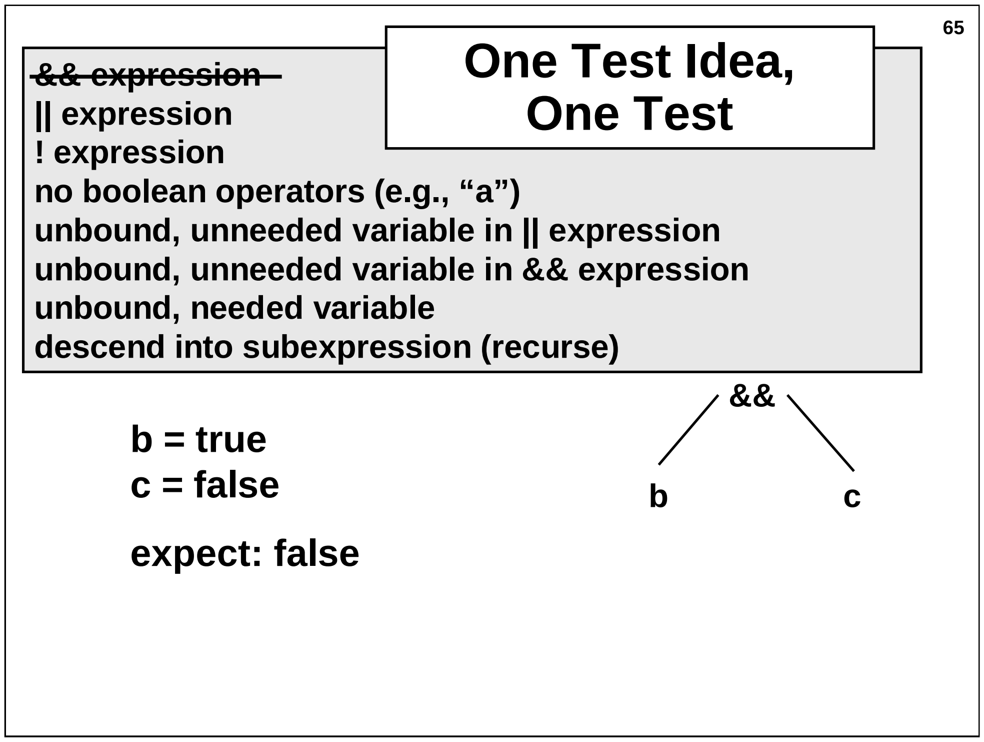
&& expression
|| expression
! expression
no boolean operators (e.g., "a")
unbound, unneeded variable in || expression
unbound, unneeded variable in && expression
unbound, needed variable
descend into subexpression (recurse)
...

In the main body of the tutorial, I have made it seem that as soon as I get an idea for a test (e.g., “eval an && expression”), I sit down and write the test.

Actually, I’m more likely to add it to a checklist of test ideas, as shown on the slide. I’ve made that checklist be handwritten, on what’s supposed to represent a piece of paper. That’s to emphasize that the checklist is not supposed to be any big deal, nothing formal, nothing preserved, nothing checked into the CM system, nothing reviewed by independent auditors. It’s supposed to be lightweight.

In practice, I sometimes handwrite it, sometimes type it in. I do whatever seems more convenient at the time. (On airplanes, I handwrite it. If it contains pictures or tables, I handwrite it. Etc.)

(If you’ve read my book, *The Craft of Software Testing*, or seen some of my other writings, you may know that I’ve also referred to “test ideas” as “test requirements”. In that book, I talk about the “test requirement checklist”. This checklist is the same thing, used in the same way.)

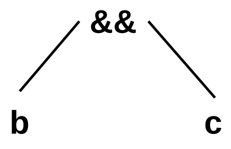


One Test Idea, One Test

~~&& expression~~
|| expression
! expression
no boolean operators (e.g., “a”)
unbound, unneeded variable in || expression
unbound, unneeded variable in && expression
unbound, needed variable
descend into subexpression (recurse)

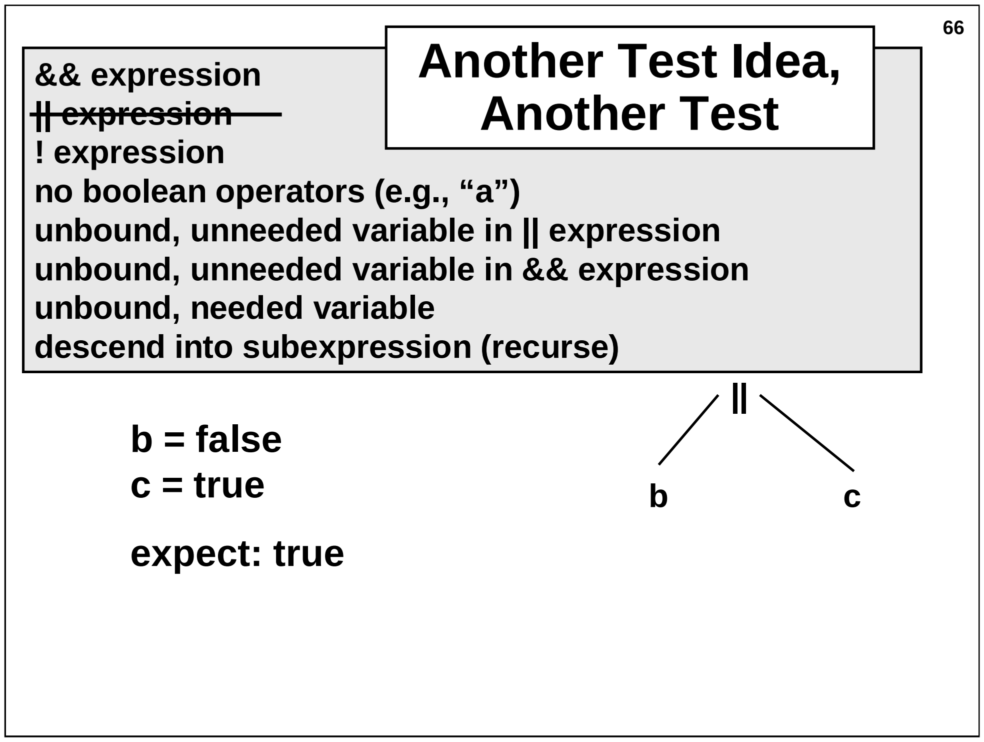
b = true
c = false

expect: false



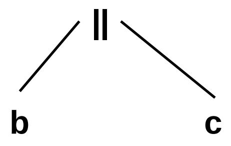
What can you do with a checklist?

Here’s one thing: you can take each test idea and write a test for it. The slide shows a test for the first test idea.



Another Test Idea, Another Test

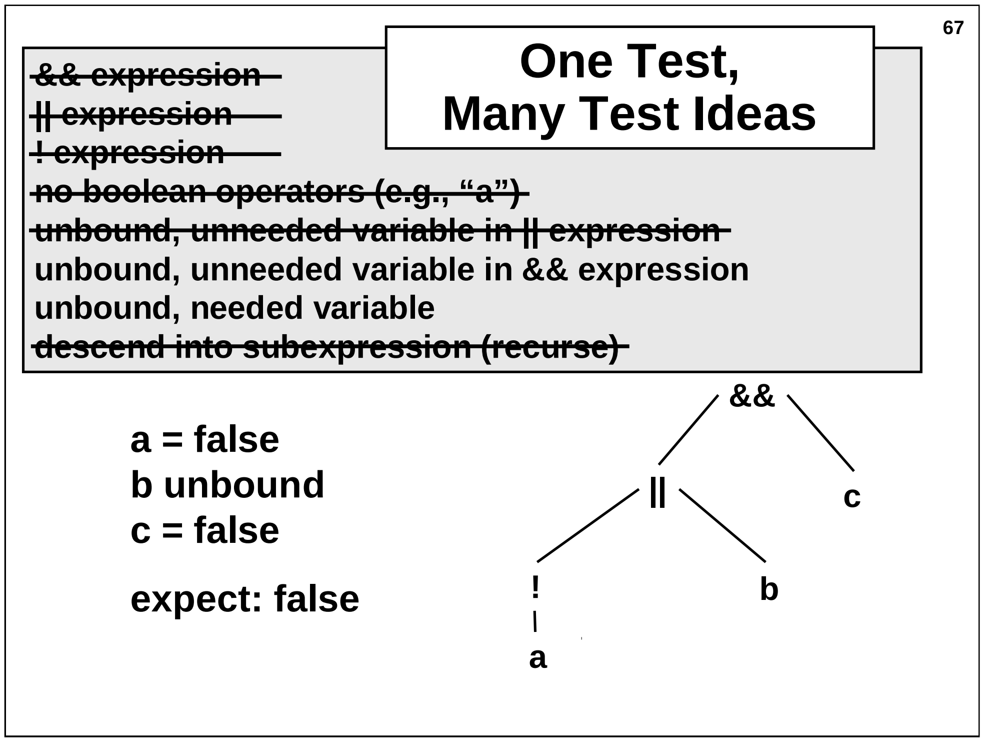
&& expression
~~|| expression~~
! expression
no boolean operators (e.g., “a”)
unbound, unneeded variable in || expression
unbound, unneeded variable in && expression
unbound, needed variable
descend into subexpression (recurse)



b = false
c = true

expect: true

Here’s a second test, for the second test idea.

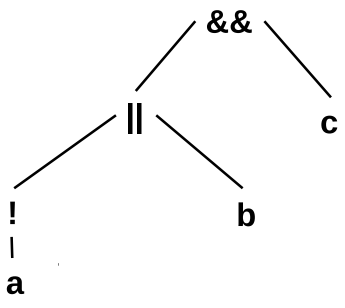


One Test, Many Test Ideas

~~&& expression~~
~~|| expression~~
~~! expression~~
~~no boolean operators (e.g., “a”)~~
~~unbound, unneeded variable in || expression~~
unbound, unneeded variable in && expression
unbound, needed variable
~~descend into subexpression (recurse)~~

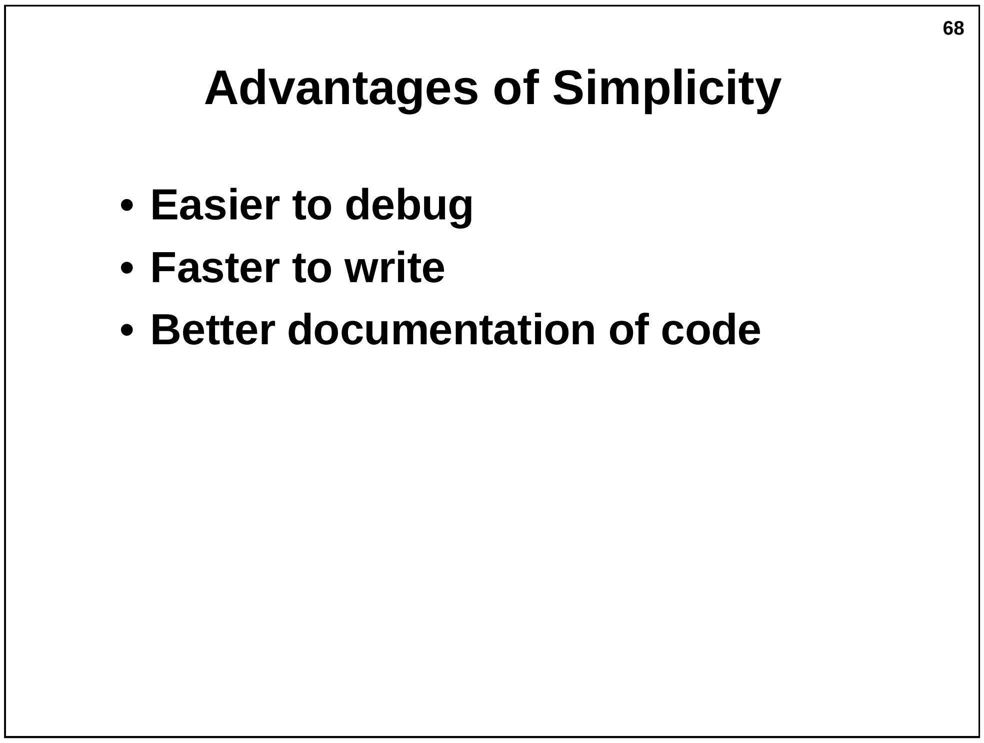
a = false
b unbound
c = false

expect: false



However, you could also implement one test that “satisfies” many test ideas. The single test on the slide satisfies six. I need only write that one test for those. I’ll need one or more additional tests for the remaining requirements.

The total number of tests will be smaller, but each test is likely to be more complex.



Advantages of Simplicity

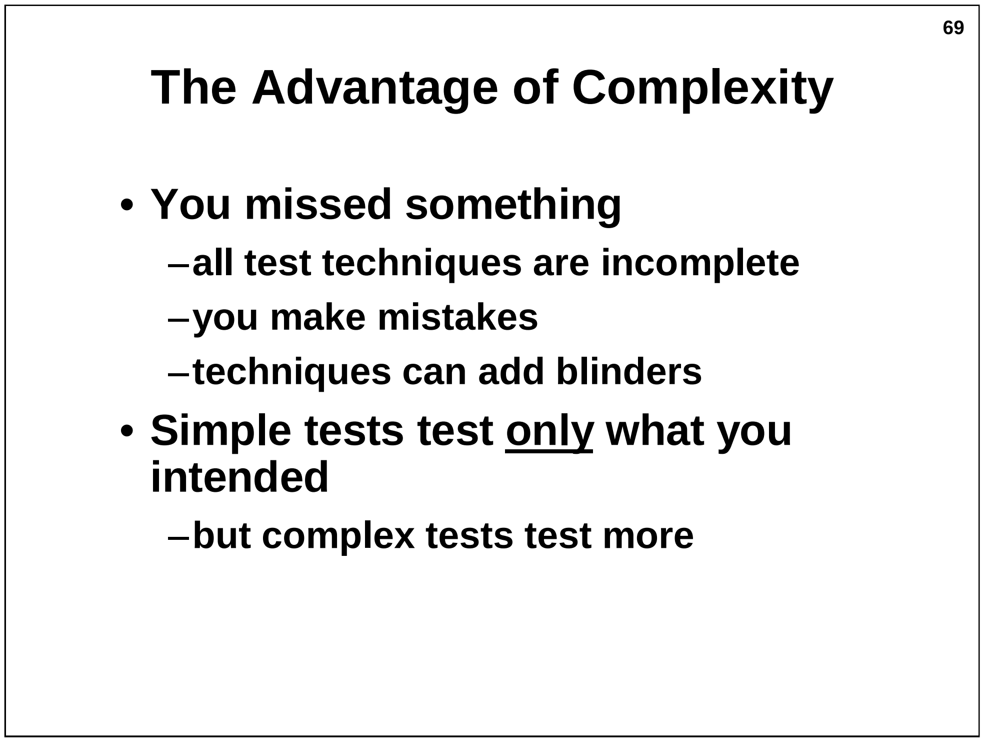
- **Easier to debug**
- **Faster to write**
- **Better documentation of code**

Which should you do? What are the tradeoffs?

• Debugging is easier with simpler tests. Suppose you find a bug with the previous slide’s test. Which of the 6 things the test did caused the bug?

• It might seem that more simple tests, being more tests, would take longer to write. In fact, the reverse seems to be true. The time spent thinking about which test ideas to satisfy in a complex test outweighs the savings of less typing of test code.

• Some (XP people, for instance) rely on tests as concrete examples of what the code does. Complex tests aren’t useful for that.



The Advantage of Complexity

- **You missed something**
  - **all test techniques are incomplete**
  - **you make mistakes**
  - **techniques can add blinders**
- **Simple tests test <u>only</u> what you intended**
  - **but complex tests test more**

The reason for complexity is that you missed something while creating test ideas. The techniques you use - the ones taught here, and whatever others you might use - are incomplete. Inevitably, they'll miss bugs. Moreover, you might have made mistakes using those techniques and missed some test ideas that might have led to bugs.

Further - and this worries me a fair amount - using methodical techniques like the ones taught here can blind you to other test ideas. You might do *only* what the techniques tell you.

Simple tests only test what you intended. If you blend ideas into a more complex test, you stand a chance of inadvertently implementing a test idea that finds a bug. You test <u>more</u> than you intended. And sometimes, the act of combining will itself lead to new test ideas. You smack your head and say, "Aha! There's something else this code should do!"

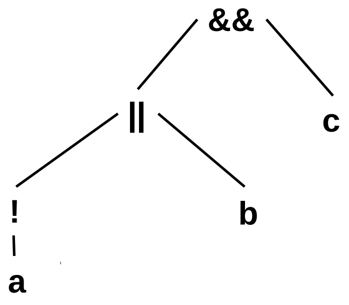
That's what, to me, justifies the additional work of producing complex tests. They find more bugs. Yes, they find the additional ones through sheer dumb luck. But I'm not proud. I'll take bugs from anywhere.

Further Disadvantages of Complexity (1)

- **Failures are more likely to be masked**

a = false
b = false
c = false

expect: false



Complexity brings with it some further disadvantages.

The more complicated the test, the more likely failures are to be masked. Take the above test. Suppose ! is mis-implemented, so that !X has the same value as X. In that case, when the test evaluates !a, the value is false when it should be true. The value of the ||-expression is also, incorrectly, false. And the value of the whole &&-expression is false.

But that’s the same value as the &&-expression would have had if ! had been implemented correctly. C being false makes the value of (!A||B) irrelevant. We say that the error is **masked**.

Error masking can happen in simple tests, but it’s rather more likely in complex ones.

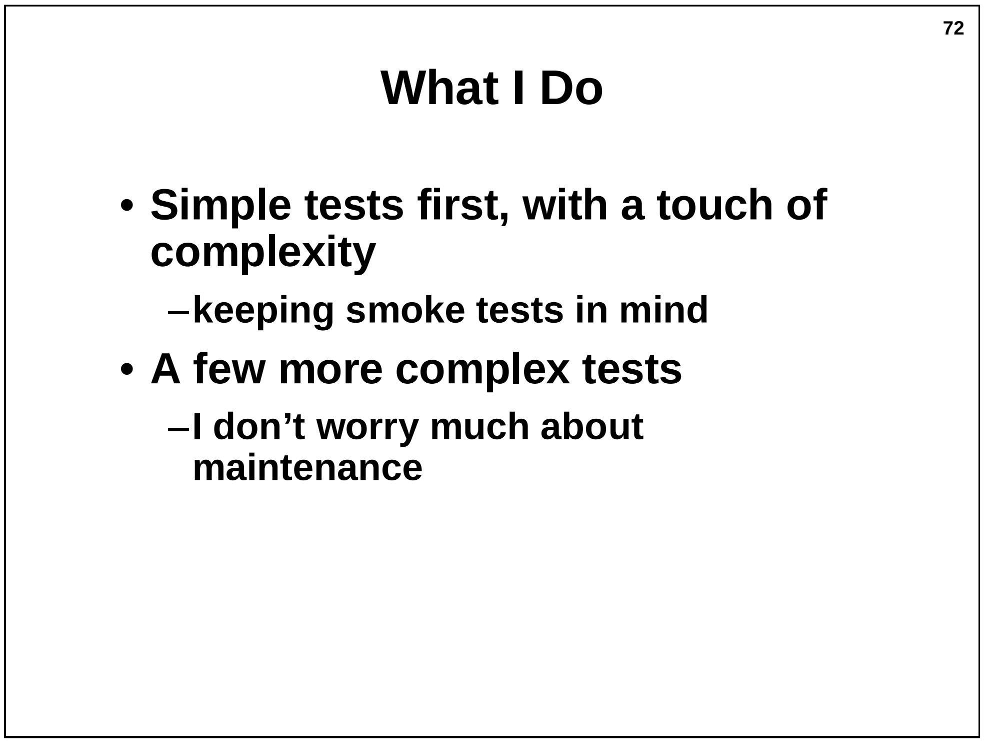
Further Disadvantages of Complexity (2)

- **You’re more likely to make a mistake**
  - **–not test what you think you’re testing**
- **Tests are harder to maintain**

There’s another way to incorrectly construct a complex test. You might not really satisfy the test ideas you thought you were. Often, this happens because you construct tests by successively adding new test ideas. So, for example, you might start with a simple expression A&&B, then add an OR - A&&(C||D) - without realizing that invalidates the reason you created the original expression.

Taking some care can avoid this problem, but not eliminate it.

Finally, complex tests are more difficult to maintain. In particular, it can be difficult to maintain their bug-finding power. The maintenance changes you make later are likely to invalidate some of the original test ideas. (Note that it’s unlikely that you retained the original checklist, so it can be hard to guess what the test was all about.)



What I Do

- **Simple tests first, with a touch of complexity**
  - **keeping smoke tests in mind**
- **A few more complex tests**
  - **I don’t worry much about maintenance**

So what do I do?

I write a relatively simple set of tests to run first. Those should catch the obvious bugs in a way that’s easy to debug. Many of them will also make useful smoke tests that I’ll try to retain for a long time.

I often add a little bit of complexity to those tests by satisfying more than one test idea, but not at the expense of comprehensibility.

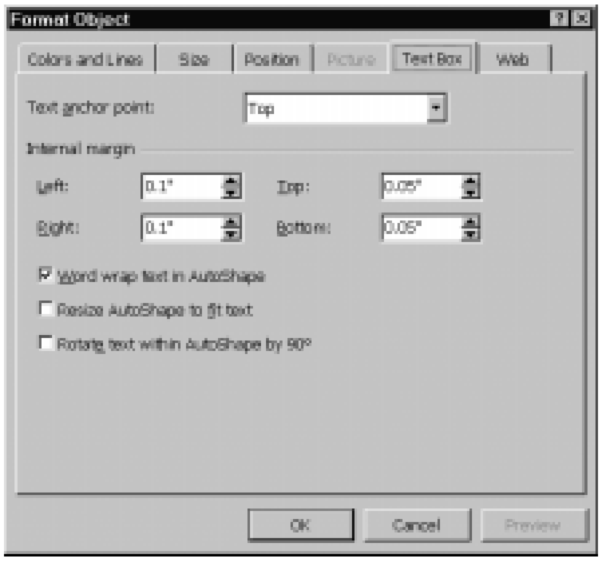
Then I will add a few complex tests. These “soak up” test requirements I don’t think justify a test of their own but are easy to merge into tests I’m writing anyway for other reasons. I may also satisfy again some ideas I already satisfied.

So, for the expr() tests, I’d have tests for simple expressions and also a few tests for complex, deeply nested expressions.

The complex tests will be difficult to maintain. Yet I want to maintain them because they’ll tend to find bugs in support code. But since these tests are largely *supplementary* to my smoke tests, test updates that invalidate the test ideas originally satisfied by the complex tests may not be such a bad thing. It may in fact be *preferable* to throw away the original complex test and write a new, rather different, complex test. Remember that the point of complex tests is to exercise the code in ways you didn’t think of. Remember also that most tests lose most of their bug-finding potential after they’ve been run once. So a broken complex test that’s replaced rather than fixed may be the best thing - and it probably costs about the same to replace the test as to fix it.

73

Further Complexity



Text anchor point: Top

Internal margin

Left: 0.1"
Top: 0.05"
Right: 0.1"
Bottom: 0.05"

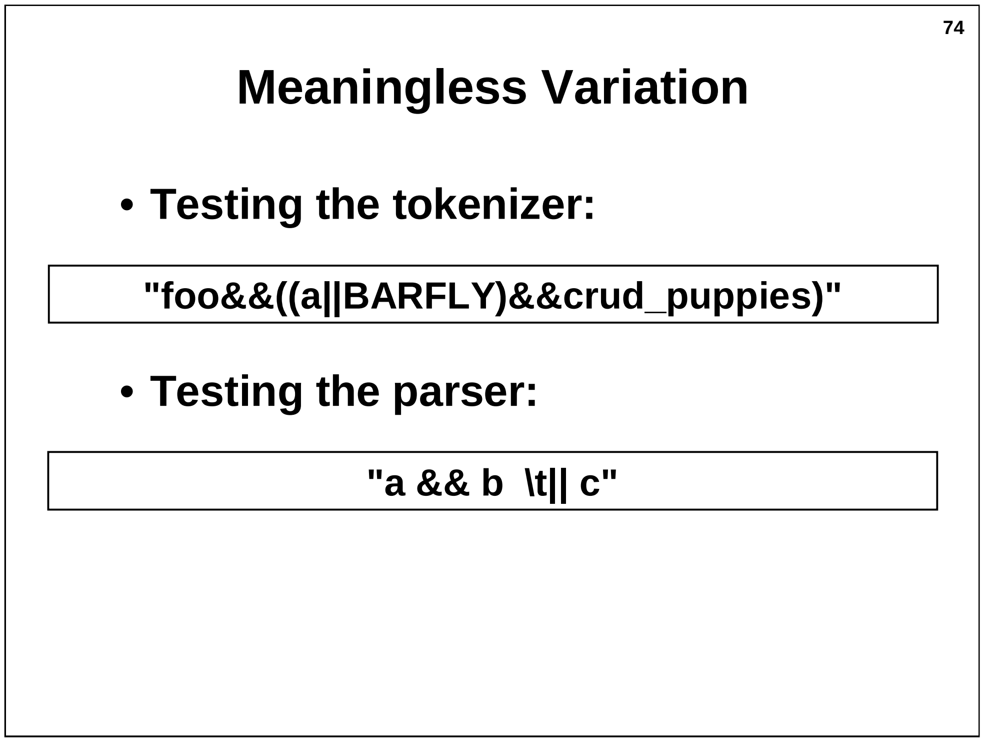
[✓] Word wrap text in AutoShape
[ ] Resize AutoShape to fit text
[ ] Rotate text within AutoShape by 90°

I often take complexity one step further. Consider this dialog, which is one used in Powerpoint.

It controls how text is laid out in objects like this one.

Some of the options might appear to be independent. For example, does the Text Anchor have anything to do with Word Wrapping or Rotate Text? Suppose I thought they didn't - and so had implemented them as supposedly-independent code in three separate tasks. I might still write a single test that incorporated ideas from all three checklists. Perhaps what I thought was independent really interacted through some shared support code. Or perhaps figuring out what the expected results of the test should be would make me realize my design for the features didn't make sense - they *should* interact because that's what a user would expect.

Unnecessary complexity finds bugs.



74

Meaningless Variation

- **Testing the tokenizer:**

```
"foo&&((a||BARFLY)&&crud_puppies)"
```

- **Testing the parser:**

```
"a && b  \t|| c"
```

There’s something else I do that’s also inspired by my certain knowledge that I’ve overlooked test ideas. I add meaningless variation to my tests. Here are two examples.

I have some code that parses boolean expressions. As you might expect, there’s a tokenizer that splits a string up into lexical units. There’s then a parser that builds trees from those units.

In the tokenizer, I am absolutely positively certain that it doesn’t matter if I use upper or lower-case letters. I’m completely sure that ‘_’ is treated the same as ‘a’. I believe with all my heart that it doesn’t matter if a token is a single character or twelve characters.

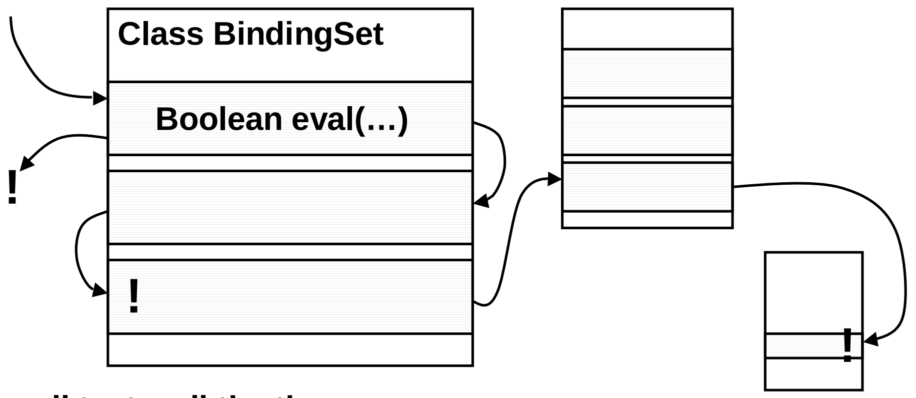
Nevertheless, I try all those variants because, well, I’ve been certain but wrong before, and it’s easy to do.

Similarly, in the parser tests, I could assume that the tokenizer works. White space should be completely irrelevant to parser tests. However, it doesn’t hurt to toss in a few tabs occasionally.

Note that, in the parser tests, I always used one-character lowercase names. I don’t insist on endless variation; that’s too much of a strain. But I toss in more variation than I think I’ll need. Occasionally, it finds me a bug.

Where Are Bugs Found?

- **Many bugs that tests find are not in the code directly under test**



run all tests, all the time

Here’s another reason for meaningless variation.

Tests find bugs when you feed inputs to a method and it produces the wrong results. According to the *IEEE Standard Glossary of Software Engineering Terminology*, that wrong result is technically called a “failure”.

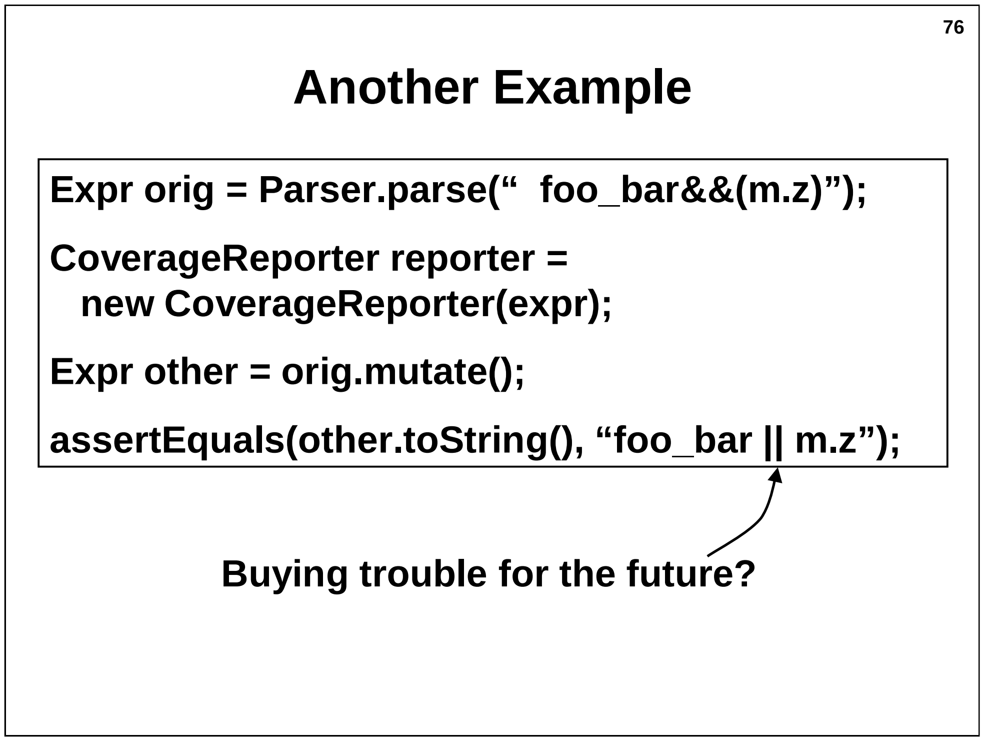
But that failure only happens because there’s some underlying “fault” - some wrong code. That code may live in the method you called. Or it may lie in another method in the same class (a method invoked by the method you called). Or it may live many method calls away in a completely different class.

Ideally, all those other methods should be so well tested that you’ll never find bugs (faults) in them. Well, they’re not, not in this imperfect world.

Variation that’s meaningless to the method you intend to test might well be meaningful to methods you’re indirectly testing. For that reason, it’s good. It helps you find bugs.

Stripping unneeded variation from your tests might mean that all methods indirectly exercised are exercised the same way each time. That reduces the chance of stumbling over bugs.

Because tests often find bugs in far-distant code, it’s wise to run all tests that could exercise code as you’re changing it. Perhaps not as often as you run the tests specific to your code, but certainly before you declare your task done. (In XP, all unit tests for all code must run and pass before new code can be integrated.)



76

Another Example

```
Expr orig = Parser.parse("  foo_bar&&(m.z)");

CoverageReporter reporter =
  new CoverageReporter(expr);

Expr other = orig.mutate();

assertEquals(other.toString(), "foo_bar || m.z");
```

Buying trouble for the future?

Here’s another example of variation and some complexity. I’m testing the Expr.mutate() method. That transforms one Expr into another, in this case by replacing the && with an ||. (That isn’t really the way the mutate() method works, but I didn’t want to complicate this example with detail.)

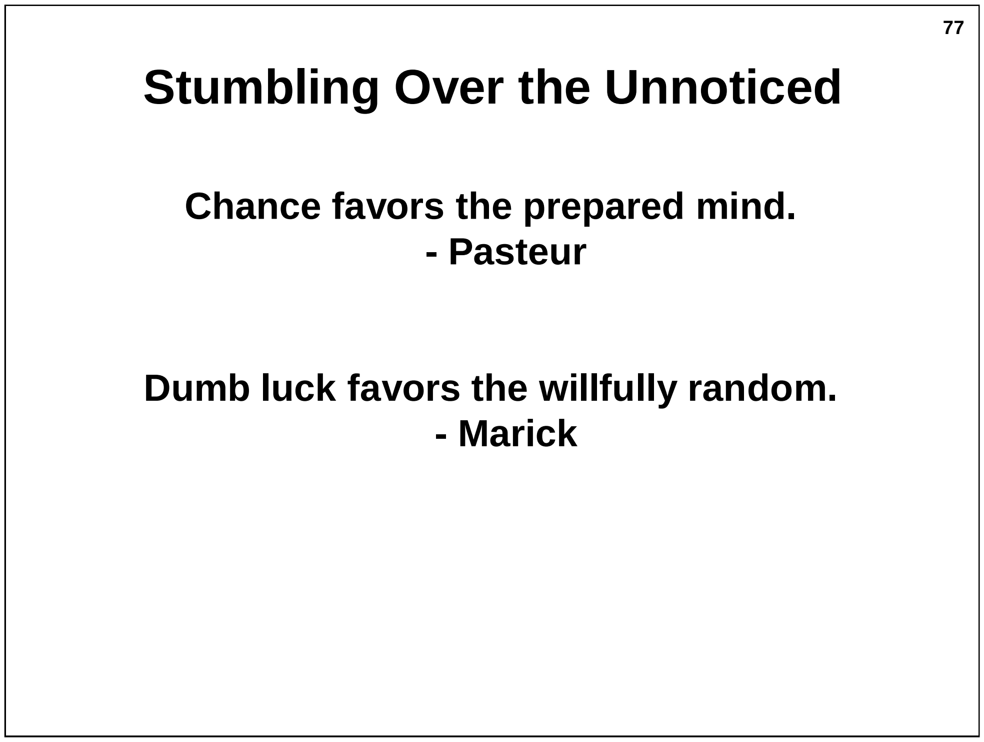
I used the Parser.parse() method to construct the tree, because it’s easier than building a tree by hand. It also tests the parser. I added some whitespace to the string because - what the heck - I’ll test the tokenizer too.

Next, I included a *completely irrelevant* line. A CoverageReporter does something mysterious to the innards of Exprs that should not affect anything about mutating them… unless there’s a bug. So I tossed in this “no op”. (By the way, this isn’t what CoverageReporter really does, either.)

Then I tested in the usual way and checked results by converting the Expr into a string and using string comparison. That’s easier than walking the tree with some code. Note that it does expose me to a risk: what if Expr.toString() changes the way it formats expressions? My test will now break. But the present savings is probably worth that future risk.

There’s another risk. Errors in Parser.parse() or Expr.toString() might mask errors in Expr.mutate(). In the application this example was drawn from, one error was indeed so masked. I found it not too much later, though.

Richard Schooler has an interesting article about introducing variation into tests in the “Front Line” department of Vol 1. No 1 of *Software Testing and Quality Engineering*. I hope to put it on www.testingcraft.com/techniques.html soon.


77

Stumbling Over the Unnoticed

Chance favors the prepared mind.
- Pasteur

Dumb luck favors the willfully random.
- Marick

You might choose to read Robert Binders' *Testing Object-Oriented Systems*. It's over 1000 pages. Or Boris Beizer's *Software Testing Techniques* (over 500). You might do <u>everything</u> in those books to test your code.

You'd still find bugs through sheer dumb luck, by accidentally doing things you hadn't intended. That's the way testing is.

Given the importance of dumb luck, it's good to organize your work so that luck works on your side. That's what complexity and variation do.

A Problem: Deadening Your Mind

- **Techniques should increase creativity, not prevent it**
- **Catalog as idea-generator**
  - **the less obvious the match, the better**
- **Brainstorming and other creativity techniques**
- **User focus**

Applying techniques by rote will lead to too many missed ideas. So you need ways to get more ideas.

• Suppose you're looking at the catalog and you say, "I don't really see a collection in the feature I'm implementing. Well, if you look at it in this screwy way, you could *interpret* it as kind of having a collection"… go for it. See where the interpretation leads you. Sometimes these screwy interpretations make you realize you left something out.

• Brainstorming techniques and other techniques have been developed to unleash creativity. I haven't studied them enough to know how to apply them well to testing, but they must be useful, especially for those who pair program. (For brainstorming, see *How to Make Meetings Work*, by Doyle and Straus, and *Facilitator's Guide to Participatory Decision-Making*, by Kaner et. al. Both are worth getting for other reasons.)

• It's easy to concentrate on code in isolation. Try to relate it to the eventual user. What will that user do that will cause your code to get called? Try to avoid the "who would have ever guessed they'd want to do *that*?" syndrome.

Luck Can Work Against You

```
…
if (a || b) {
   bomb.detonate(2 * time);
} else {
   bomb.detonate(time * time);
…
```

What if time=2?

variety

I've been talking about getting luck on your side. But it can also work against you.

We've been concentrating on inputs to the code. For example, in the section on boolean expression errors, I showed a program that will make a boolean expression containing one of a set of likely errors evaluate to the wrong value. In the above code snippet, it would give you a test that would force the program to take the wrong branch of the IF statement.

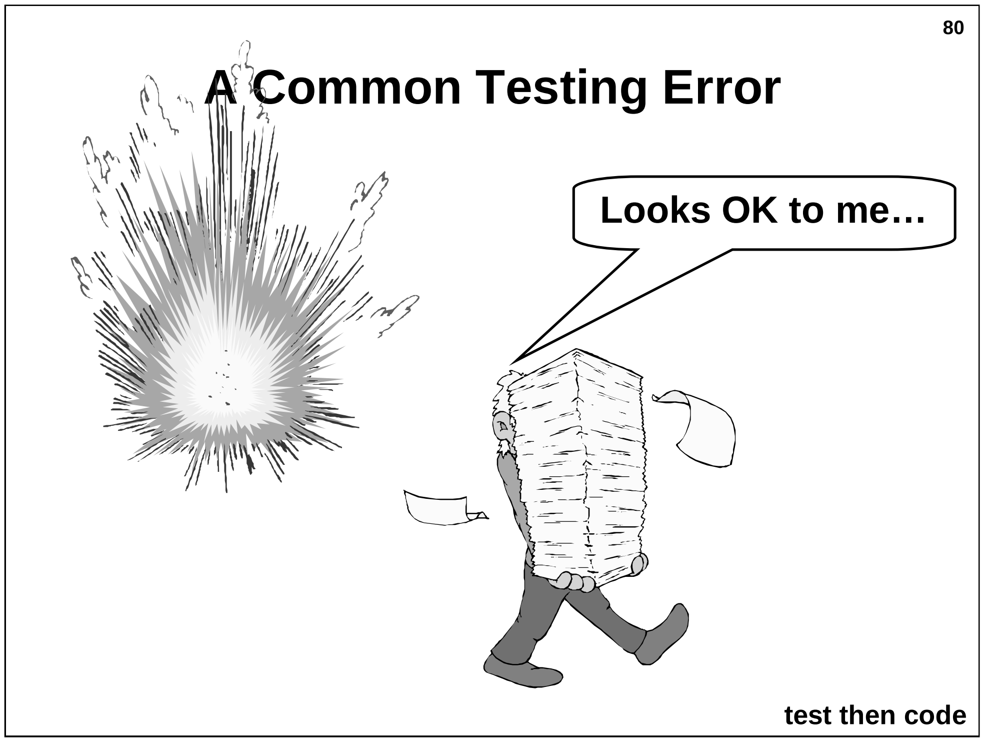
Unfortunately, in that test, TIME might have the value 2, through sheer bad luck. The wrong branch will do the same thing as the right one.

Rats. Bug missed. Right answer, wrong reason.

You can't completely avoid this problem (not affordably*). But, as you write your tests, it's worth asking yourself "If the code does the wrong thing, will I be able to tell?"

Adding variety to your tests (e.g., not having all the tests use 2 as the value of time) can help. (Not with this particular case, but with more complicated ones.)

* The way to avoid this problem is to use a technique called "strong mutation", first proposed in Demillo, Lipton, and Sayward, "Hints on test data selection: help for the practicing programmer", *IEEE Computer*, April, 1978.) It's not affordable, though.



A Common Testing Error

Looks OK to me…

test then code

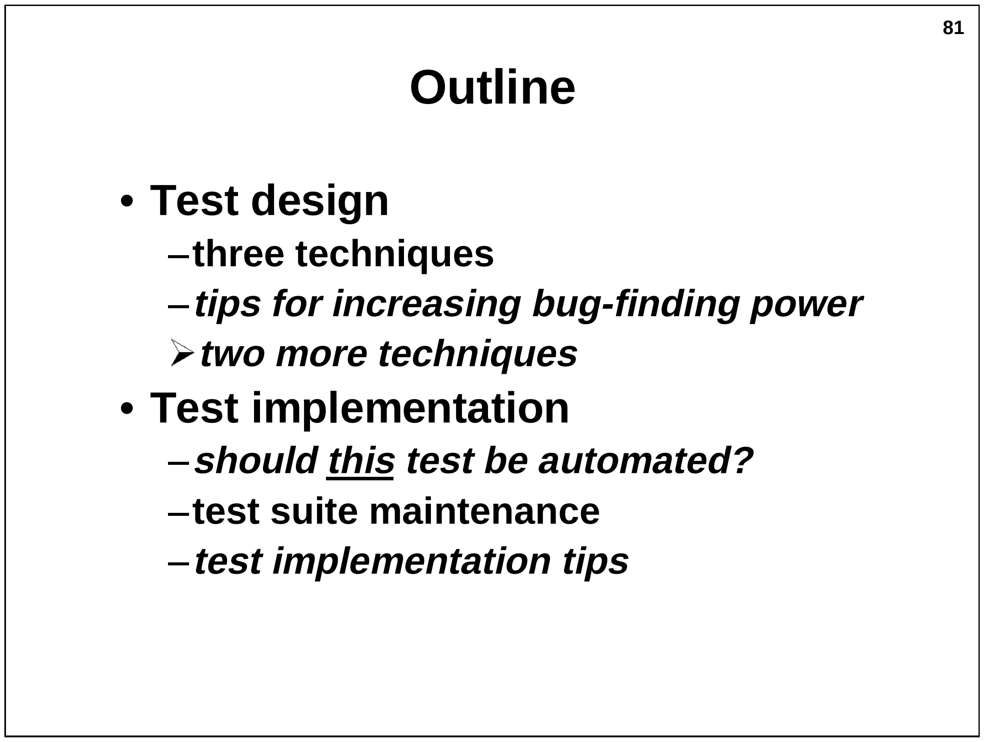
Even if the program *does* produce the wrong result, you might not notice. People are fairly good at looking at incorrect program output, shrugging, and saying “Yeah, looks OK”.

(See these papers for experimental evidence:

V. Basili and R.W. Selby. ‘Comparing the Effectiveness of Software Testing Strategies.’ IEEE Transactions on Software Engineering, vol. SE-13, No. 12, December, 1987.

G.J. Myers. ‘A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections.’ Communications of the ACM, Vol. 21, No. 9, September, 1978.)

The way to best avoid this problem is to decide what’s correct *in advance*, by writing your tests before you write your code. If you run a few test cases in a debugger or interpreter, check to see what the answers are, think they’re OK, then write the tests to check that those answers keep coming back, there’s a chance you’ve just “immortalized” a bug - the tests will keep passing the buggy output.



Outline

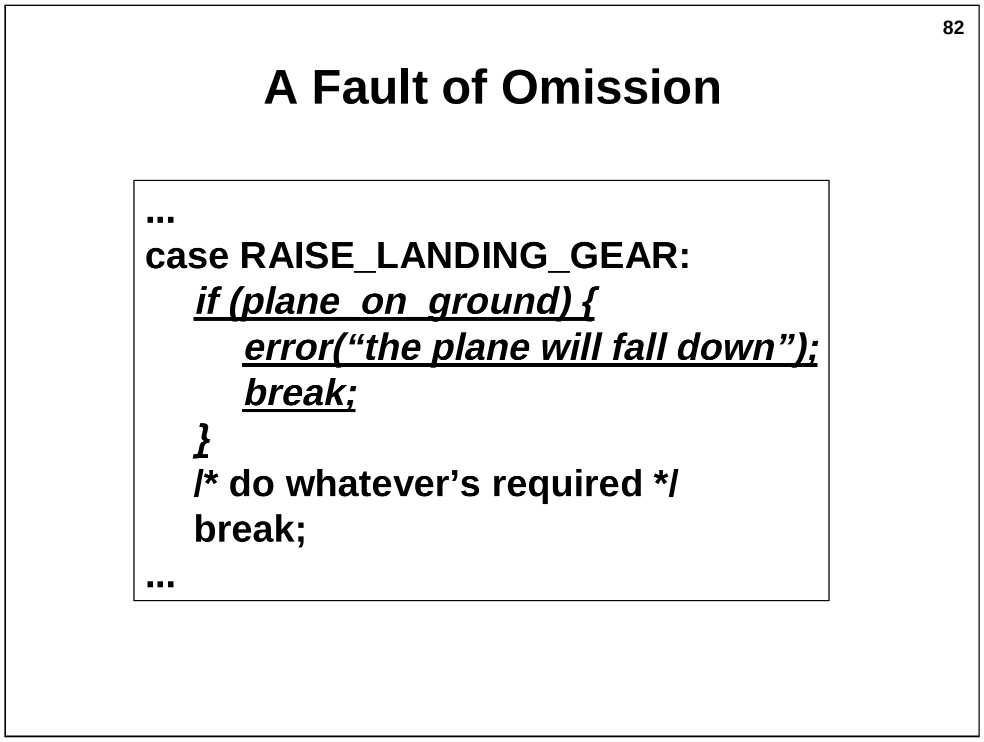
- **Test design**
  - **three techniques**
  - ***tips for increasing bug-finding power***
  - ➢ ***two more techniques***
- **Test implementation**
  - ***should this test be automated?***
  - **test suite maintenance**
  - ***test implementation tips***

Here are two more test design techniques.

One looks for mistakes handling results of method calls.

One looks for overlooked variables.

Both target an important class of fault, the fault of omission.



A Fault of Omission

```
...
case RAISE_LANDING_GEAR:
    if (plane_on_ground) {
        error("the plane will fall down");
        break;
    }
    /* do whatever's required */
    break;
...
```

Here's a story that I don't know is true, but that I've heard from two separate sources.

Once there was a jet fighter. A test pilot got in it, turned it on, and lifted the landing gear - while the plane was on the ground. It broke.

What was the cause? I don't know, but it could well have been code like the one in the slide. There's a big event loop (or whatever) taking inputs from the outside. One of those inputs is the RAISE_LANDING_GEAR command. In the slide, the italicized and underlined code shows the fault. It is the code that *isn't* in the buggy version. It is the code that would have prevented the command from taking effect.

Such faults - ones that are fixed by adding code - are called faults of omission. They are typically faults where someone - some programmer or designer - forgot about a case or requirement. They're what Bob Glass memorably called "code not complex enough for the problem". (Robert L. Glass, "Persistent Software Errors," *IEEE Transactions on Software Engineering,* March 1981.)

Faults of Omission: Some Numbers

- **47% of USENET patches (Marick)**
- **30% of bugs in a shipped Microsoft product (Sherman)**
- **22%-54% of shipped product bugs (five other studies)**

Faults of omission are an important class of fault in shipped products. They are bugs that escape programmer testing and independent testing.

For citations to all the studies, and more discussion of faults of omission, see “Faults of Omission” on my web site:

http://www.testing.com/writings/omissions.html

It was originally published in *Software Testing and Quality Engineering,* Volume 2, Number 1, January 2000.

Faults of Omission and Testing

- **Test, then code**
  - **before the code is written, you might avoid them**
  - **after, you won't detect them**
- **Test design techniques should help**
  - **catalog does, in part**
  - **here are two more**

How can you find these faults?

• Once you've written code and overlooked something, you're in some sense "locked in" to a particular implementation. It's now harder to realize that the user (or the environment, or some other source of inputs) is going to do something you didn't anticipate.

So writing tests after code will cause weaker tests and thus weaker code. Write the tests first.

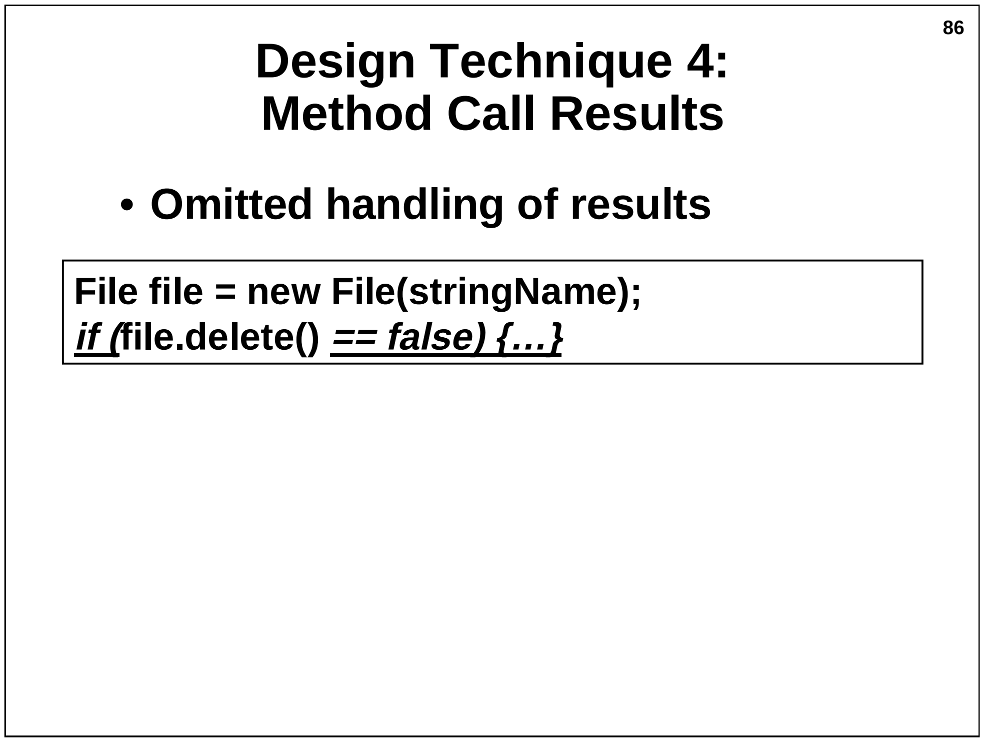
• The catalog helps you find certain faults of omission: failure to handle null pointers, for example. As you build up your own catalog of test ideas (probably a mental one), you'll find that it contains ideas for catching faults of omission.

• The rest of this section describes two techniques that target faults of omission.

Any Good Stories?

But first, have any good stories about faults of omission?

I am one of the technical editors for *Software Testing and Quality Engineering* magazine. We have a department called Bug Report. In it, authors describe an interesting bug and recount what they learned from it. If you have a good bug and would like to write an article, contact me at marick@testing.com.



Design Technique 4: Method Call Results

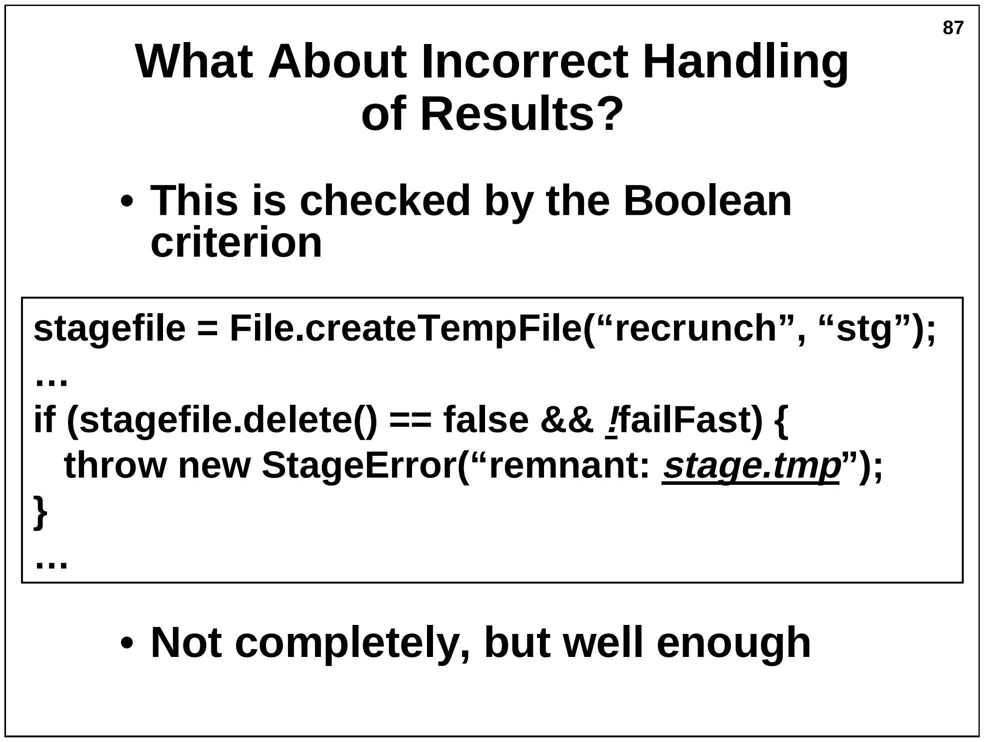
- **Omitted handling of results**

File file = new File(stringName);
if (**file.delete()** ***== false) {…}***

One common type of fault of omission is failing to handle some result of a method call.

The first example in the slide shows a call to File.delete(). The bug is that File.delete() returns a boolean to indicate whether the deletion is successful. The underlined and italicized code is omitted: the programmer *assumed* that deletion always works. Well, it doesn't.

This next test technique checks for such overlooked results.



87

What About Incorrect Handling of Results?

- **This is checked by the Boolean criterion**

```
stagefile = File.createTempFile(“recrunch”, “stg”);
…
if (stagefile.delete() == false && !failFast) {
  throw new StageError(“remnant: stage.tmp”);
}
…
```

- **Not completely, but well enough**

What about the types of results that *haven’t* been overlooked. Wouldn’t it be good to check not only that they’ve been remembered, but also that they’ve been handled correctly?

Recall that we’re going to be exercising all the code. Among other reasons, the boolean test design technique guarantees it. So we’ll have tests that trigger both bugs in the code on the slide:

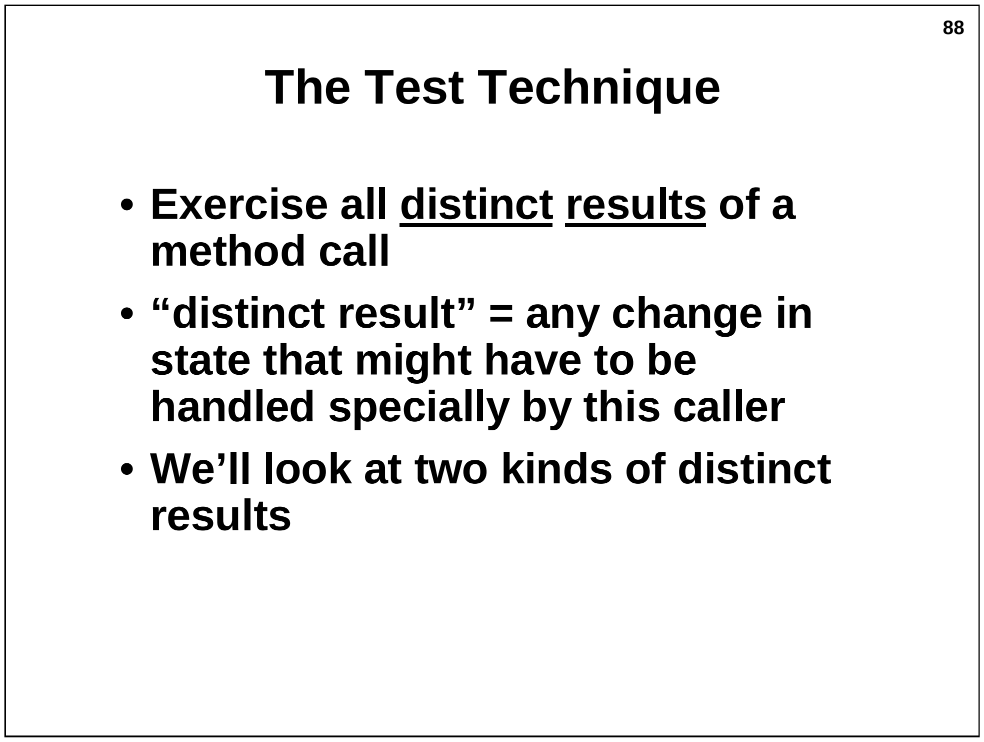
• the incorrect ! in front of failFast.

• the fact that the error mentions a file, “stage.tmp”, that is the wrong name. Apparently, the name was changed, but this mention of the old name wasn’t fixed.

Merely exercising a line of code won’t find all the bugs in it. Here’s a bug that won’t necessarily be found:

```
a = b/c;        // what if c is 0? - a fault of omission.
```

However, I believe the five criterion presented in this tutorial are good enough for programmer testing.



88

The Test Technique

- **Exercise all <u>distinct</u> <u>results</u> of a method call**
- **“distinct result” = any change in state that might have to be handled specially by this caller**
- **We’ll look at two kinds of distinct results**

So, a good set of tests will exercise all the distinct results of any given method call.

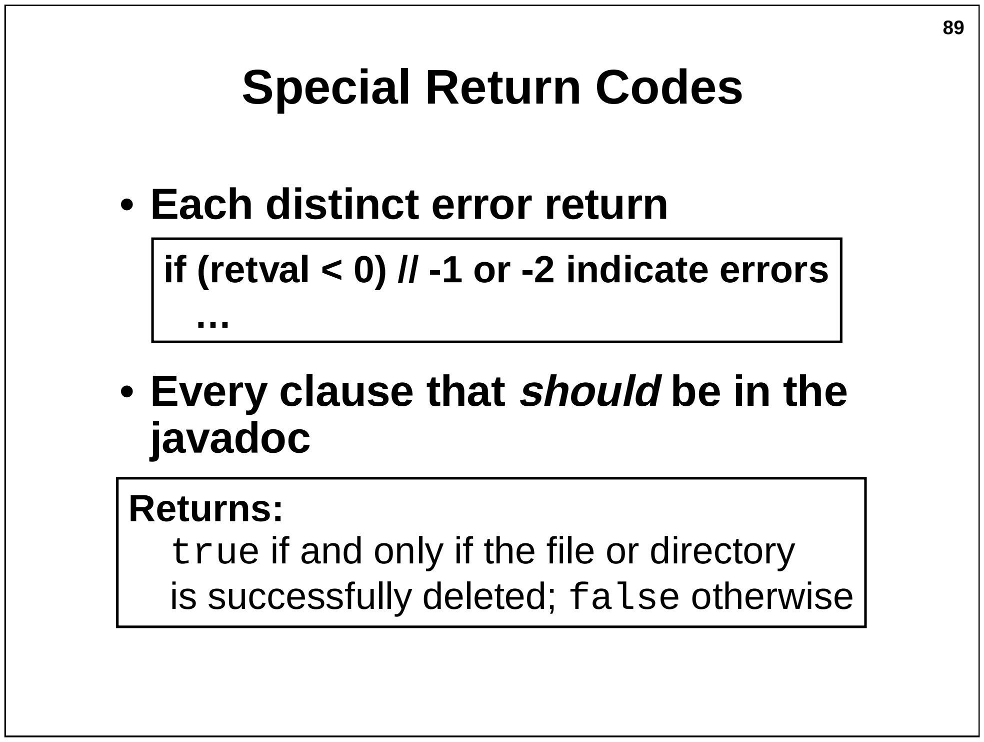
What’s a distinct result? (Bear with me - this is hard to explain.)

When the method executes, it changes the state of the world. It might allocate new objects. It might push return values on the stack. It might change a global variable. It might stuff values in a database. It might send data over a network. Whatever.

The continued execution of the caller might have to take account of some of those changes. For example, it might need to check the return value and do something special if that’s FALSE. On the other hand, even though a message printed to standard output changes the state of the world, that change isn’t something the method needs to take account of. On the third hand, something written to a database might be, if the method might then read the database and be affected by the change.

Of these changes or results, some are different enough that checking one of them doesn’t give you any particular confidence that the other works right. So you need to check them separately.

Let’s see some specific examples.



89

Special Return Codes

- **Each distinct error return**

```
if (retval < 0) // -1 or -2 indicate errors
   …
```

- **Every clause that *should* be in the javadoc**

Returns:
`true` if and only if the file or directory is successfully deleted; `false` otherwise

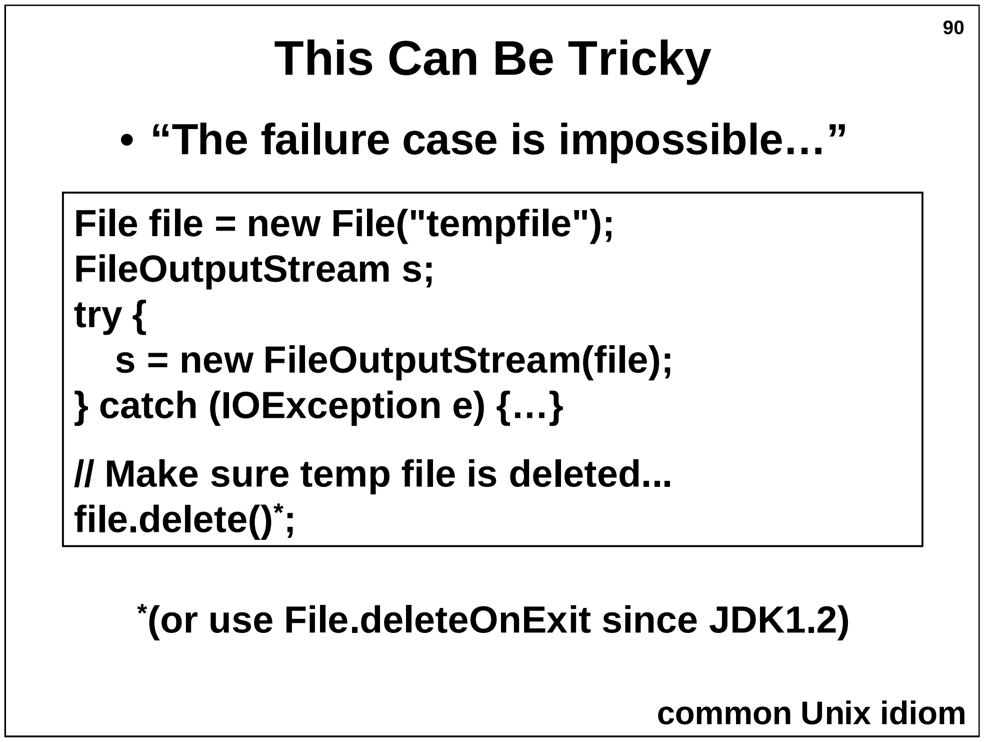
The most apparent case is that each error return from a method should be checked. Suppose the method returns two error codes, but the code handles them both together (as in the slide, or by having them both be cases of a switch statement like this one:

```
case ERROR1:
case ERROR2:
   …
```

or whatever). Despite the fact that the code treats both values the same, write tests that return each. Maybe the code *shouldn't* treat them both the same and a surprisingly different result will alert you to that.

More generally, you want to check each return code case that really complete documentation would spell out. Of course, it's often the case that return cases - especially error cases - aren't described. For example, I know of one commercial operating system whose documentation describes only the error returns immediately generated by routines in the system API - it does **not** describe the return codes that those routines pass on from deeper kernel routines.

In such a case, do the best you can.



This Can Be Tricky

- “The failure case is impossible…”

```
File file = new File("tempfile");
FileOutputStream s;
try {
   s = new FileOutputStream(file);
} catch (IOException e) {…}

// Make sure temp file is deleted...
file.delete()*;
```

*(or use File.deleteOnExit since JDK1.2)

common Unix idiom

Often, it will appear that error returns are impossible. Sometimes they aren’t.

The slide shows a Java implementation of a common Unix idiom for handling temporary files. The goal is to make sure that a temporary file is always deleted, no matter how the program exits. You do this by creating the temporary file, then immediately deleting it. On Unix, you can continue to work with the deleted file. The operating system takes care of cleaning up when the process exits. It’s quite cute.

It doesn’t work on Windows. The deletion will fail.

Nothing in the Java documentation mentions this as of August 2000. The documentation does not enumerate the situations in which delete() fails. A not-painstaking Unix programmer might not write the code to check for a failed deletion - in her world, it can’t happen. Having not done that, she certainly wouldn’t write a test case for that “impossible” (but not impossible) case.

What can be done about this?

In general, nothing. It’s good to know a lot of trivia, like differences between Unix and Windows. It’s important to try hard to surface and disprove your assumptions, difficult though that is.

And, after JDK 1.2, use deleteOnExit, which I expect was put in exactly because of this idiom.

91

This Can Seem Pointless

- **“I know that case doesn’t matter…”**

```
void foo(Comparator c) {
  …
  if (c.compare(o1, o2) <= 0) {
    …
  } else {
    …
  }
```

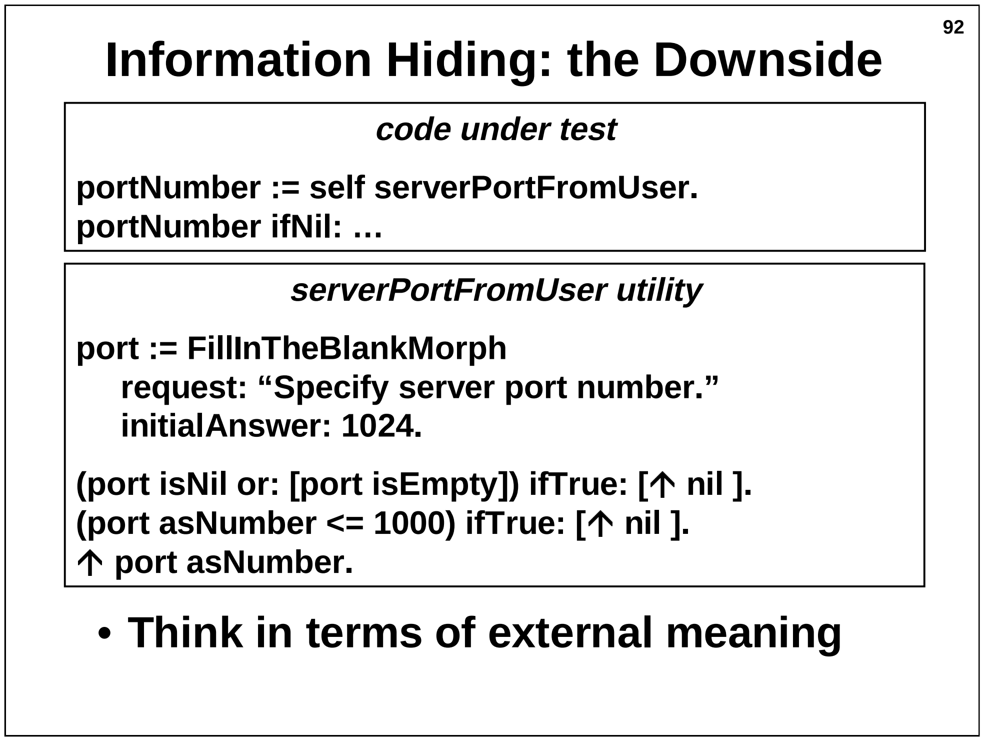
Three distinct results: <0, =0, >0

multiple executions

Another force against just bucking down and trying a case is being already convinced that it doesn’t matter. A java Comparator’s compare() method returns either a number <0, 0, or a number >0. Those are three distinct cases that might be tried.

The code lumps two of them together. But that might be wrong. The way to discover whether it is or not is to try the two cases separately, even if you really believe it will make no difference. (Your beliefs are really what you’re testing.)

Note that you might be executing the THEN case of the IF statement more than once for other reasons. Why not try one of them with the result less than 0 and one with the result exactly equal to zero?



Information Hiding: the Downside

code under test

```
portNumber := self serverPortFromUser.
portNumber ifNil: …
```

serverPortFromUser utility

```
port := FillInTheBlankMorph
    request: “Specify server port number.”
    initialAnswer: 1024.

(port isNil or: [port isEmpty]) ifTrue: [↑ nil ].
(port asNumber <= 1000) ifTrue: [↑ nil ].
↑ port asNumber.
```

- **Think in terms of external meaning**

There’s a danger that comes when you’re coding - and testing - method-by-method. Here’s an example. It’s in Smalltalk, because that’s where I saw it. The code should still be easy enough to understand.

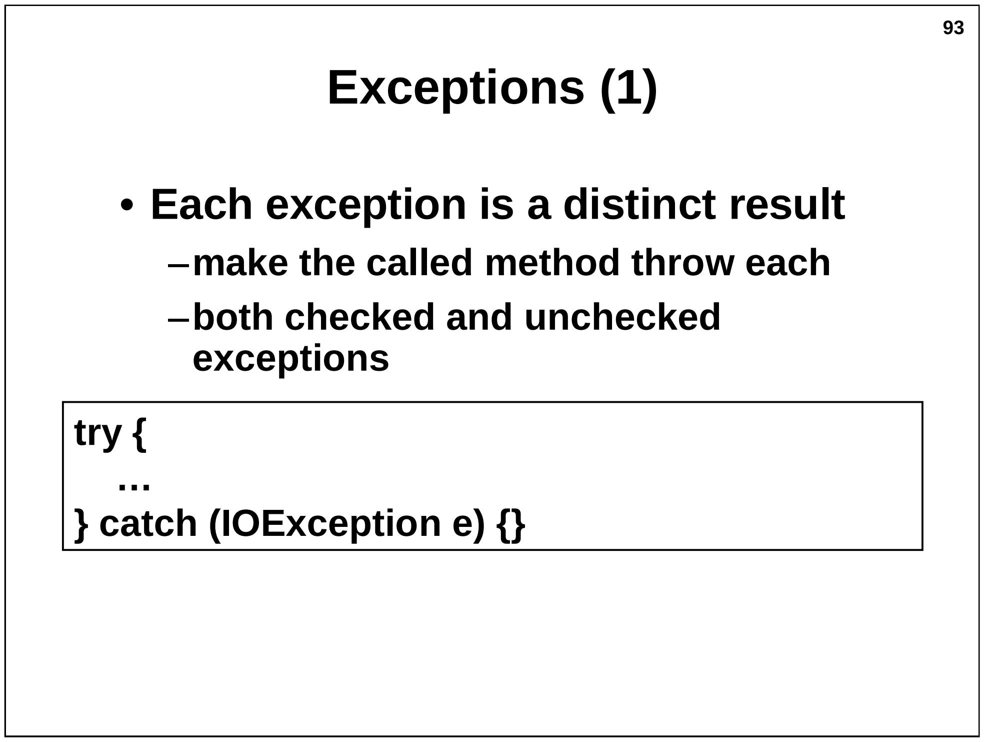
There are two methods. The first, the *code under test*, wants to establish a network connection. It calls serverPortFromUser to get a port number. That method returns two values. It returns a port number chosen by the user if the number chosen is valid (greater than 1000). Otherwise, it returns NIL. If NIL is returned, the code under test pops up an error message and bails out.

I believe that the code was tested with two cases: a valid port number and an invalid one. That’s what the interface to serverPortFromUser naturally suggests.

Unfortunately, serverPortFromUser collapses two cases into one. Here’s what it does. It pops up a window (a FillInTheBlankMorph) that takes a string and has the standard OK and CANCEL buttons. CANCEL returns an empty string, which serverPortFromUser relays to its caller as NIL. serverPortFromUser also checks the value from the Morph (if there is one). If the number is too small (less than 1000), it relays this erroneous input to the caller as… NIL. I expect that the programmers also tested this, and it indeed behaves as intended.

The combination of the two chunks of code, though, has a bad consequence: the user hits Cancel and gets a message about a bad port number. All the code works as intended, but it’s still wrong. It was tested in a reasonable way, but a bug was missed.

The trick to catching the bug is, when testing the using method, to ask what a return value of NIL *means*, and thus recover the two special cases (and find the design flaw).



Exceptions (1)

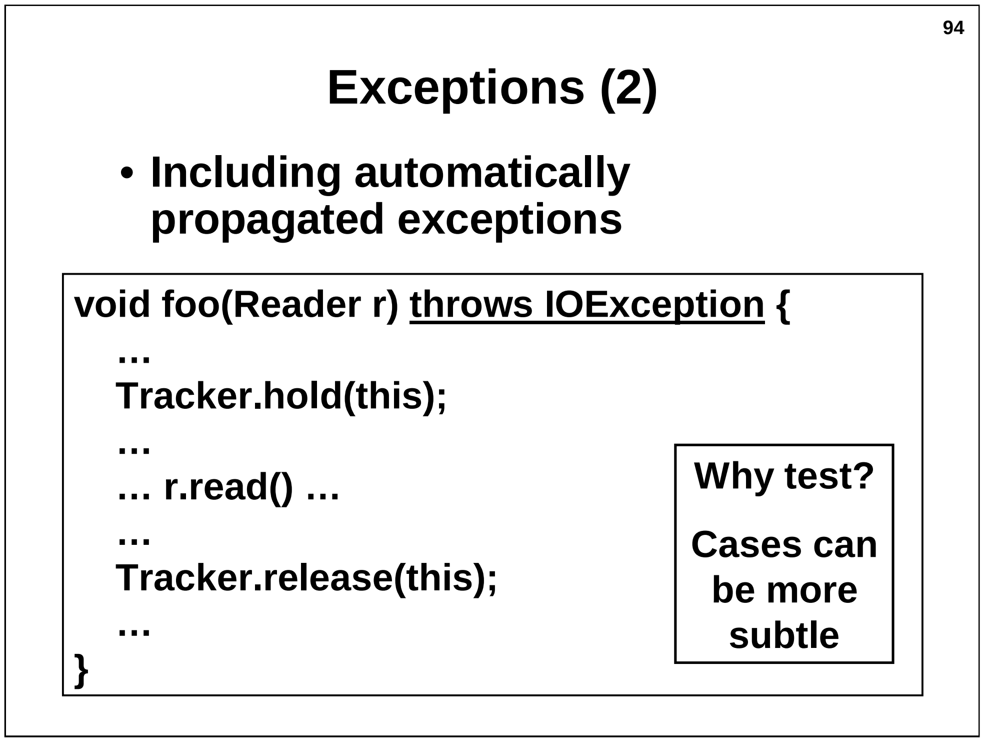
- **Each exception is a distinct result**
  - **make the called method throw each**
  - **both checked and unchecked exceptions**

```
try {
   ...
} catch (IOException e) {}
```

Exceptions are another type of result. You should make a called method throw each kind of exception. That will exercise every catch block and perhaps also reveal the need for an omitted catch block.

Exercising catch blocks is a good idea. Error-handling code is notoriously error-prone. I've indicated that on the slide by showing an extreme case. Java forces you to catch some exceptions. Sometimes, when programmers don't know what to do with them, they catch them and do nothing - which is usually the wrong thing to do. How to find out? Try it.

Note: for simplicity's sake, I lied when I said earlier that the boolean criterion forces every line of code to be exercised. It doesn't necessarily exercise catch blocks. This does - and the boolean criterion forces all lines within them to be completely exercised.



Exceptions (2)

- **Including automatically propagated exceptions**

```
void foo(Reader r) throws IOException {
   …
   Tracker.hold(this);
   …
   … r.read() …
   …
   Tracker.release(this);
   …
}
```

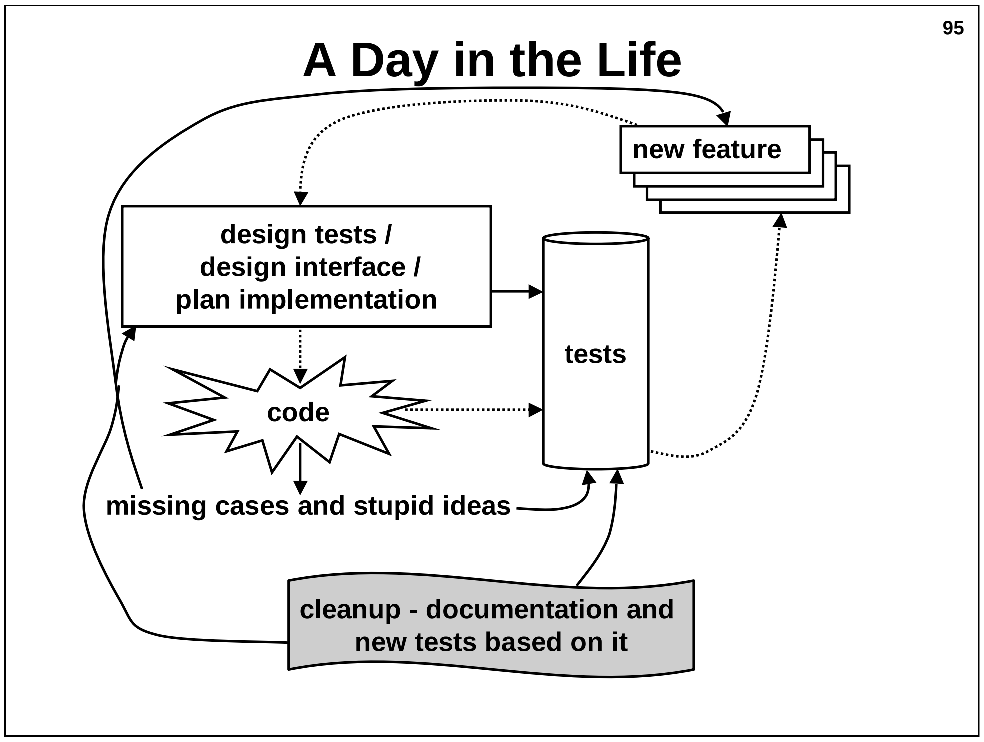
Why test?

Cases can be more subtle

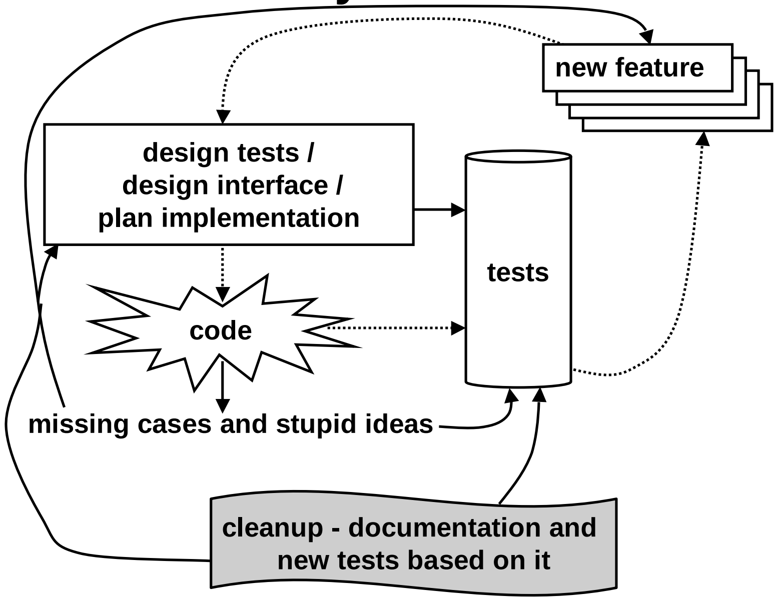
It's important to test an exception even if all the method does is to automatically re-throw the exception. The reason? The method might set up state that needs to be torn down before it returns.

Note that to find the bug, you'll have to call the method again. But that's free, since you'll have multiple tests for other reasons.

For an interesting article on testing for these sorts of bugs, see Keith Stobie's Bug Report, "Testing for Exceptions" in Volume 2, Number 4, of *Software Testing and Quality Engineering* magazine.



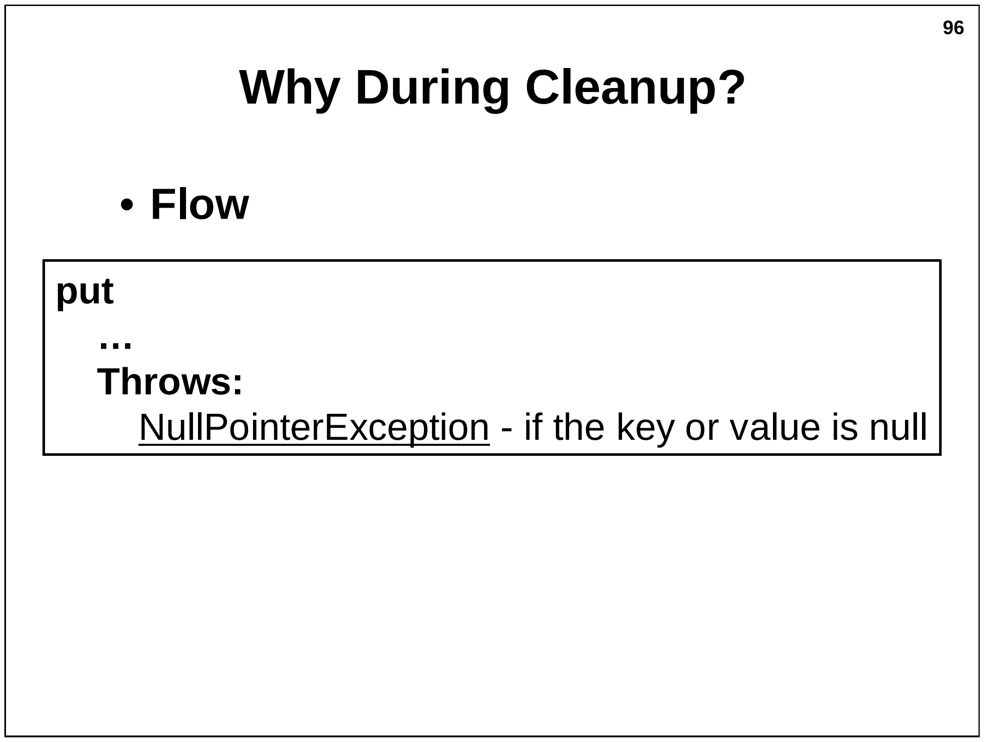
A Day in the Life


new feature
design tests / design interface / plan implementation
tests
code
missing cases and stupid ideas
cleanup - documentation and new tests based on it

When do I do all this?

I do some of it when coding. When I know I'll use File.delete(), I create the test case for a deletion failure. (Deletion success will probably be checked in tests I was writing anyway, for other reasons.)

During coding, though, I only create tests for the results I know about. Later, in cleanup, I check for results I might have overlooked. See the next slide.



Why During Cleanup?

- **Flow**

put
…
Throws:
NullPointerException - if the key or value is null

Here’s an example. Suppose I’m using a Hashtable. My standard use of the Java documentation is mainly to look at the summaries that show the method names, return values, and argument types. So, if I’ve forgotten how to put an item in a Hashtable, I’ll look there to see that the method is named “put”.

I will rarely look at the full description of put(), which is where it says that the method can throw a NullPointerException. To be constantly reading about all the methods I use would destroy the flow experience. Instead, it works for me to do that check in my documentation post-pass.

Your mileage may vary.

Implementation

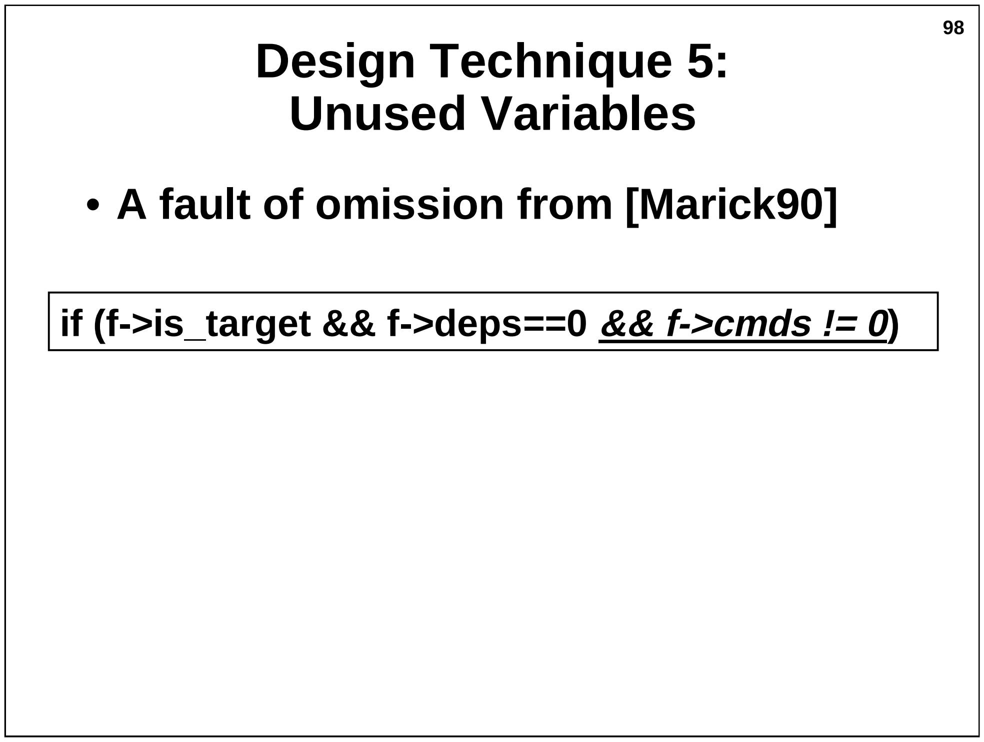
- **These cases can be hard to implement**
- **Thinking about them is justification enough**
- **Let your experience be the guide**

The method result criterion can be hard to implement. It can be hard to force methods to throw their odder exceptions.

You have to make a cost-benefit tradeoff: what's it worth to me to check that this case really works? In some cases, it might not be worth enough.

Even if you don't write and execute the test, you may get enough benefit by just thinking through the case. It may be obvious that your code won't work if the put() method throws an exception.Or the problem might be subtler, one that requires that you trace through the execution.

Let your own experience tell you how much testing is enough.



Design Technique 5: Unused Variables

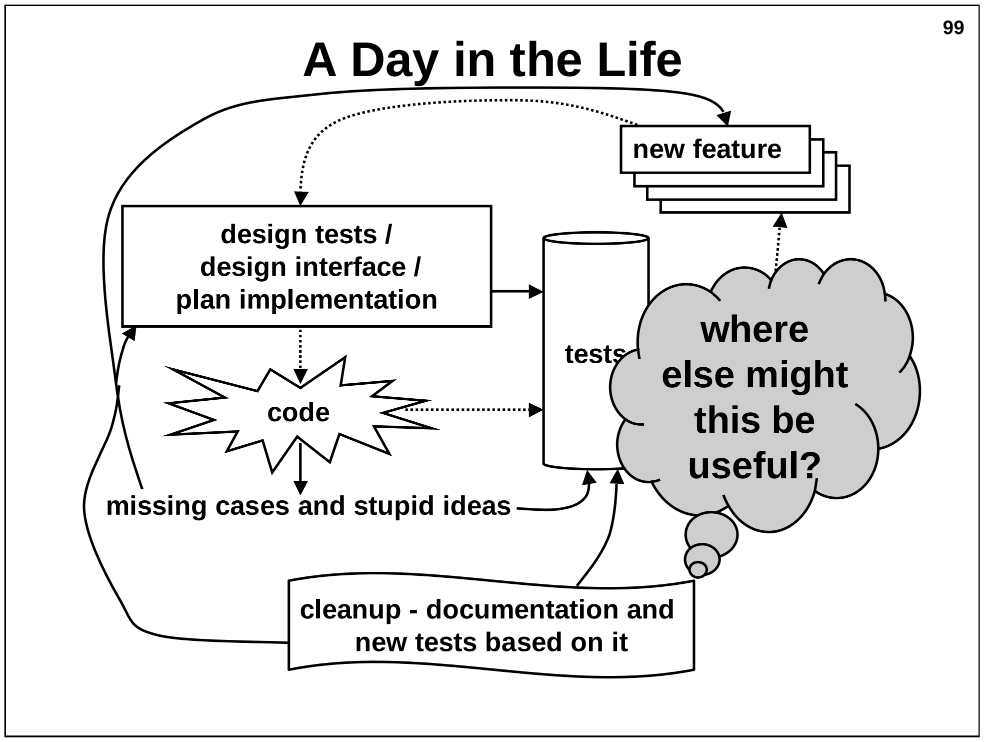
- **A fault of omission from [Marick90]**

if (f->is_target && f->deps==0 *&& f->cmds != 0*)

I did a survey of USENET patches in 1990. Here’s one of the bugs. It’s a fault of omission in some C code: the underlined, italicized code was added by the patch.

In particular, a field in a structure was overlooked. That seems to be a reasonably common variant type of omission fault: related variables that haven’t been considered.

My Usenet survey was published in “Two Experiments in Software Testing”, Brian Marick, University of Illinois Computer Science department report UIUCDCS-R-90-1644, November 1990. Sorry, the electronic version is gone.

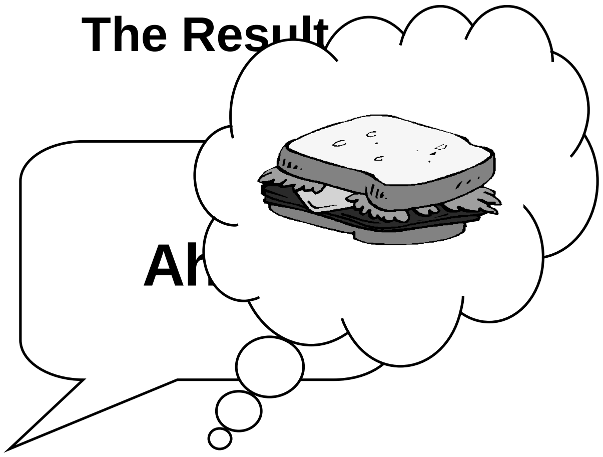


99
A Day in the Life
new feature
design tests / design interface / plan implementation
tests
code
where else might this be useful?
missing cases and stupid ideas
cleanup - documentation and new tests based on it

So, during my cleanup phase, I read through methods in a class. When a class uses a variable, I ask “where else should this variable have an effect?” Sometimes (not often enough) I think of a place. I say, “Aha! I overlooked something!”

An alternative would be to look at the variable declarations and ask, “where could this variable be relevant?” For me, it works better to be prompted by actual uses of the variable in methods. (But your brain may work differently.) Moreover, scanning through methods for variables doesn’t confine my thinking to variables declared in the class I’m working on.

The Result



Complexity and variety

That’s what should happen. Sad to say, I’m not very good at this. I don’t get as many “Aha!” moments as I should. I can’t concentrate well enough, or think broadly enough about where a variable might be relevant. I’m too stuck in what I did to think well about what I should have been.

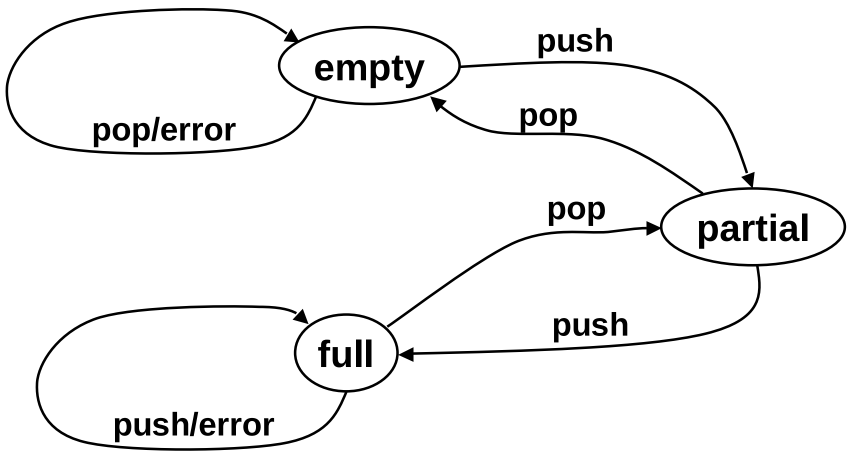
Test complexity partially compensates for my failings: it leads to variables being set to unexpected values that turn out to have an effect.

Variety helps: I try not to set “irrelevant” variables to the same value all the time.

I *could* insist on testing every method against every distinct value of every variable, but I don’t have time. So I rely on thought, supplemented by dumb luck.

101

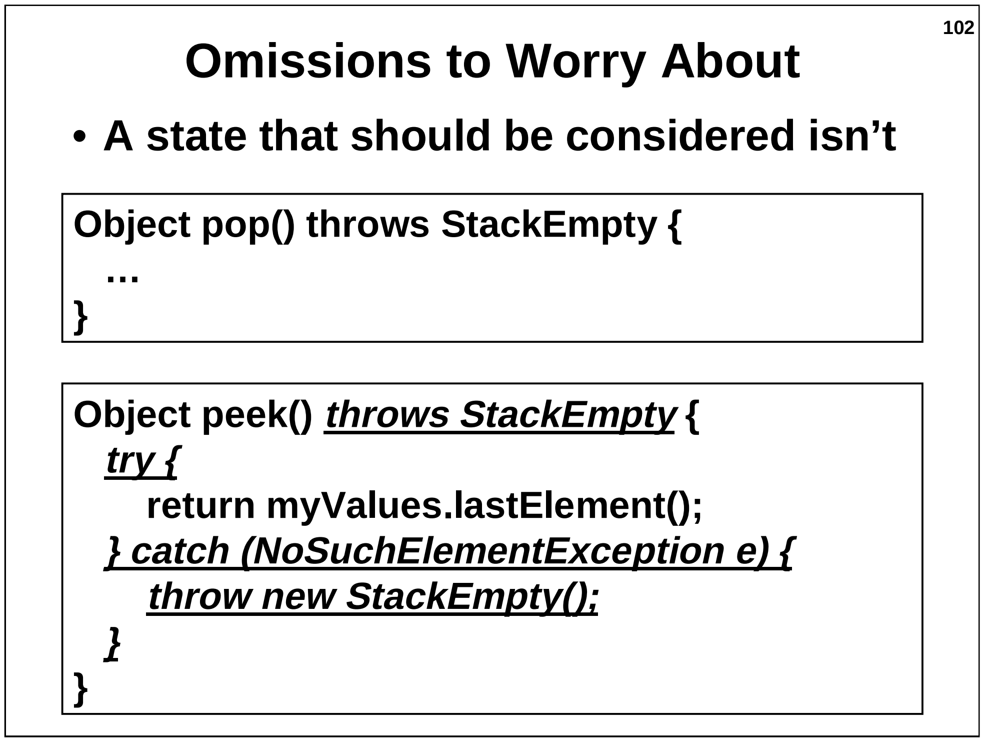
State Machines



three elements for simplicity

Here’s a related application of the same idea.

Some people sometimes model classes or systems as state machines. State diagrams like this one can make the system easier to understand.



Omissions to Worry About

- **A state that should be considered isn't**

```
Object pop() throws StackEmpty {
   …
}
```

```
Object peek() throws StackEmpty {
   try {
      return myValues.lastElement();
   } catch (NoSuchElementException e) {
      throw new StackEmpty();
   }
}
```

A problem that sometimes happens is that a particular method that should take account of the state does not.

In this example, the Stack.pop() method behaves correctly when the stack is empty. Whoever coded the Stack.peek() method forgot about that case (that state), and left out the underlined italicized code. An exception is thrown (by accident), but not the right one.

103

Check the Complete State Table

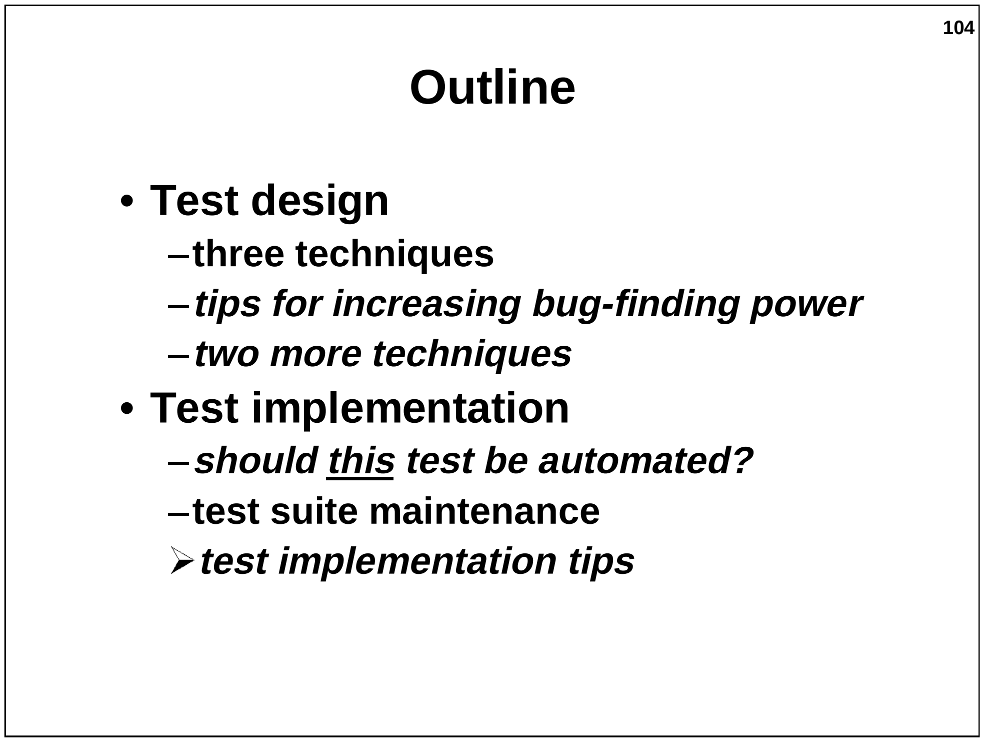
state event	state1	state2	state3	state4	state5
method1					
method2					
method3					
method4					
method5					

In theory, a state table forces you to consider what should happen in every combination of state and event. That makes it better (for finding problems) than a state diagram, which allows you to inconspicuously leave out events.

In practice, it’s hard not to have your mind deadened as you’re going through the exercise. For example, it’s easy to just transcribe the diagram into the table and then more-or-less mindlessly fill in all the empty cells with some error action, without really thinking about what should go there.

For this reason, some recommend that every state/event transition be explicitly tested. I think that’s another cost/value tradeoff, like all the rest in this course.

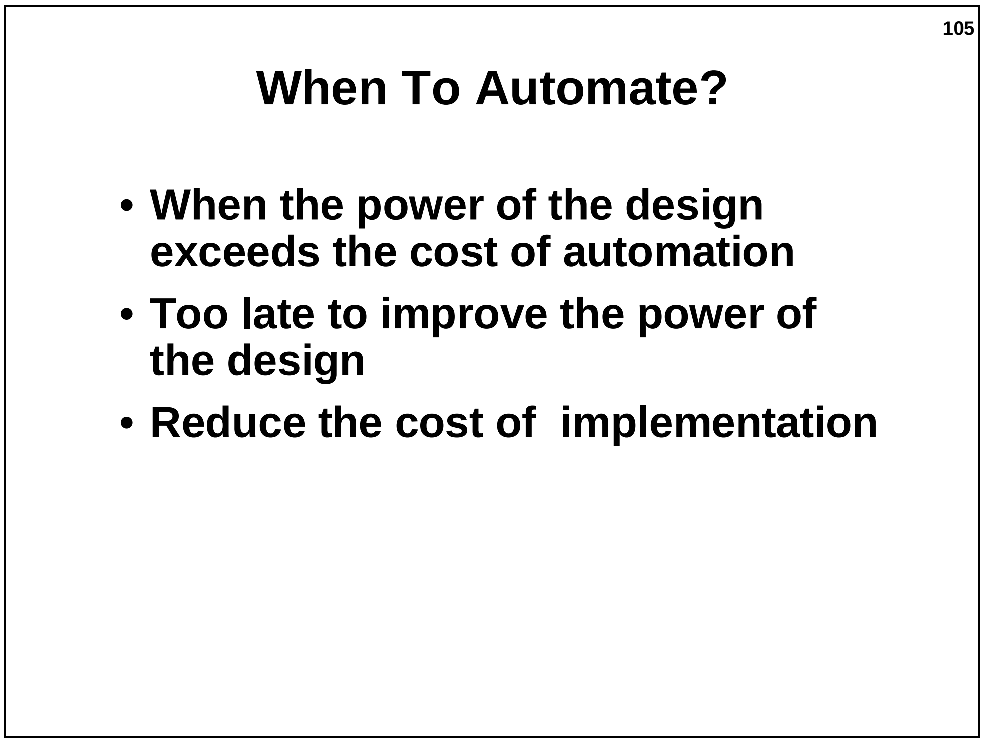
Indeed, testing is all about balancing cost and value under conditions of extreme uncertainty.



Outline

- **Test design**
  - –**three techniques**
  - –***tips for increasing bug-finding power***
  - –***two more techniques***
- **Test implementation**
  - –***should <u>this</u> test be automated?***
  - –**test suite maintenance**
  - ➢***test implementation tips***

We'll now switch gears back to test implementation.



When To Automate?

- When the power of the design exceeds the cost of automation
- Too late to improve the power of the design
- Reduce the cost of implementation

Recall that you should automate a test when its ability to find likely bugs exceeds the cost of automation. Once you’ve designed a test, it’s too late to improve its power. But you can reduce its cost.

Simplifying the Tests (1): Add Test Support Code

```
expr = …
assert(expr instanceof ExprOr);
assert(((ExprOr)expr).leftChild instanceOf ExprNot)
…
```

```
expr = …
assert(expr.toString().equals("!a || b"));
```

- **You'll want a toString() anyway**

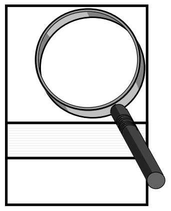
You've seen this example before. Rather than writing the code to check each element of an Expr, I use the toString() method. I wrote the toString() method specifically for testing, but I was hardly surprised to find that it's useful for debugging and also came in handy for printing results back to the user.

It is also often possible to take debugging support code and, with only a tiny amount of effort, make it useful for testing. (The effort is usually to do things like suppress irrelevant and changeable information. For example, C programmers will often write debugging routines that print pointer addresses. Those change from run to run, so they can't be used as is in tests. Debug logs with timestamps present similar problems.)

Simplifying Tests (2): Violate Encapsulation

```
// should be in Backtrack state. If so,
// an accum() should cause a relay()
// to return FALSE.
```

```
assert(machine.state == machine.BACKTRACK);
```



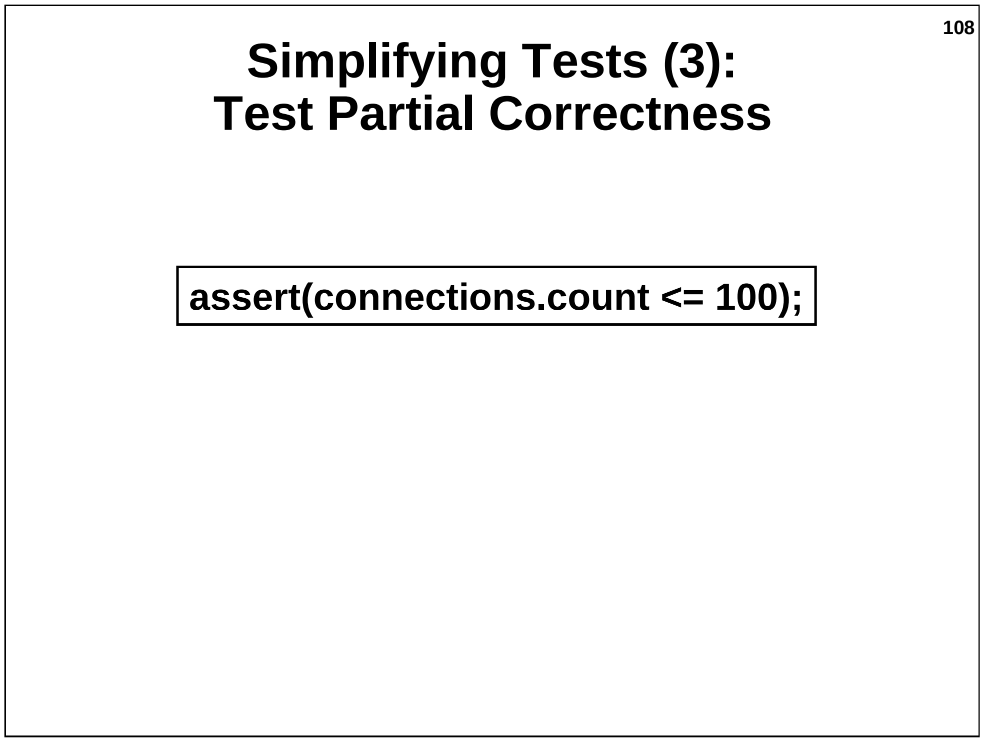
Classes serve as encapsulation boundaries, hiding interior detail from the outside world.

That outside world is usually taken to be composed of other programmers who want to incorrectly couple their code to the details of your implementation, making their code fragile in the face of changes (or making it impossible for you to change your code, practically speaking).

Unfortunately, your tests are part of that outside world. Internal state can be invisible to them (even if they're in the same package as the class under test, which I recommend).

In the testing literature, you often see descriptions of elaborate ways to infer what the state must be. Suppose the class implements a state machine. Your test wants to check that a particular action caused a transition into the BACKTRACK state. You could do that by issuing a sequence of further actions that would *only* work if the state machine is really in the BACKTRACK state.

Or, you could just rip back the covers and look. That's what I recommend. Expose (to the package) internal state that the test needs.



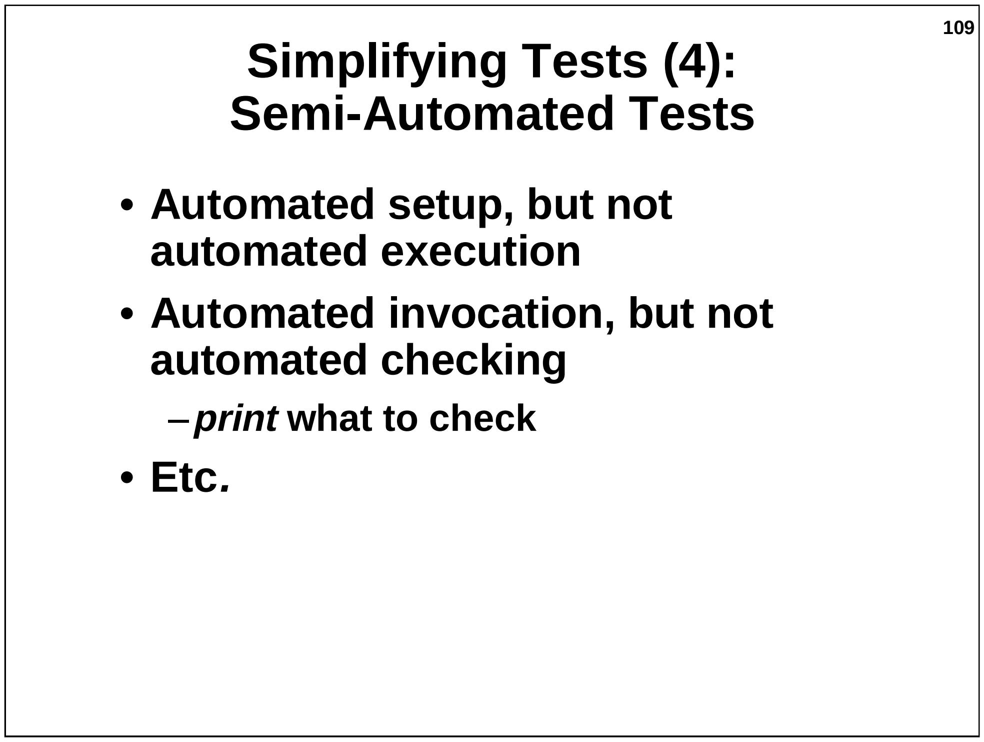
Simplifying Tests (3): Test Partial Correctness

```
assert(connections.count <= 100);
```

Sometimes doing a complete check of correctness isn’t possible. Suppose your tests set up some number of connections (whatever a connection is). For some reason, it’s difficult to know exactly how many connections is correct, but more than one hundred is certainly **in**correct.

So only check what you affordably can.

This principle applies elsewhere. If there are several things you could check, some of which are hard, check only the few that are easy.



Simplifying Tests (4): Semi-Automated Tests

- **Automated setup, but not automated execution**
- **Automated invocation, but not automated checking**
  - ***print* what to check**
- **Etc*.***

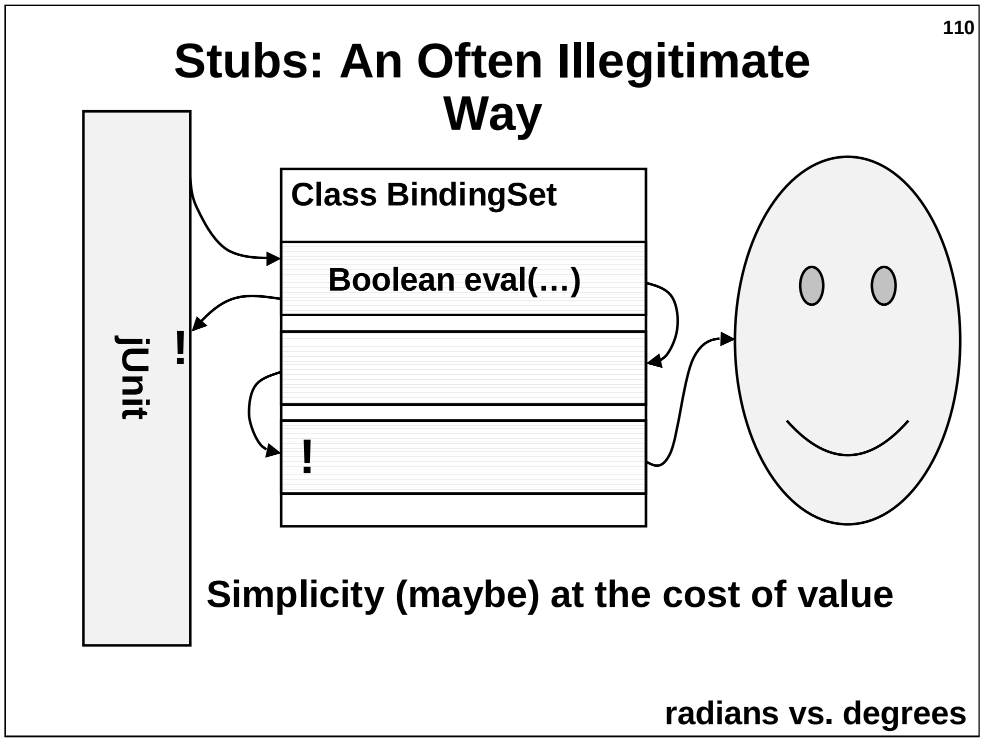
“Manual” and “automated” are two ends of a continuum. You could have many types of semi-automated tests.

For example, you might have a bunch of tests that would be easy to automate, except that each needs a network connection. The network connection is hard to setup up automatedly, easy to set up by hand. In that case, you could set the connection up by hand, then run the tests. Not as nice as full automation, but perhaps the optimal tradeoff.

It seems that my most common semi-automated test is one where the input is automatically generated, but the output is checked by hand. In that case, when I run my test, its output looks something like this:

```
% make run

Check file test1.out for the following:

        __ the frizzbat start message has today’s date

        __ no boojum precedes the first snark

        __ scan to see that there are no wrapped lines
```

The specific list makes me less likely to look at something that’s wrong and think it’s OK because my mind has been dulled by the boredom of looking at test output.



110

Stubs: An Often Illegitimate Way

Simplicity (maybe) at the cost of value

radians vs. degrees

Test setup is often difficult. You may need to put files on disk, set up databases, establish network connections, etc. etc. etc. All this to test your simple little method!

Or, perhaps, your method uses a class that someone else is writing. They're not done yet, but you are. You want to run your tests, show that your code works, and move on.

It is tempting to use **stubs**. Instead of setting up a database to return the correct value, write a fake routine that returns what the database access routine would have.

There are two problems with stubs.

• Any misconceptions you have about the called class will be implemented in the stub. Suppose, for example, that the stubbed-out class is for trigonometric functions that haven't been written yet. You think the functions will operate in radians. The implementer thinks degrees. Your stub will be in radians, and all your tests will pass… until the real code comes along.

• Remember that some fraction of bugs that your tests find will be in the methods they indirectly exercise. They can't exercise those methods if they're stubbed out.

Sometimes stubs are necessary, but use them as little as possible.

Self-Checking

```
Solver solver = new Solver(expr);
System.out.println(… the solution …);
if (!solver.isCorrect(expr))
  System.out.println(solver.describeIncorrectness());
```

It can be useful to have the code check itself on whatever input it happens to get.

For example, I've shown you a program that generates test ideas from boolean expressions. Those ideas (descriptions of BindingSets) are supposed to distinguish the expression that's been given from a set of variant expressions. For example, values of A, B, and C that distinguish A||(B&&C) from these variants:

(A||B)&&C	**A&&B&&C**	**!A||B&&C**	**A||!(B&&C)**
A||B||C	**A||!B&&C**	**A||B&&!C**	
A	**B&&C**	**A||B**	**A||C**

The algorithm to create the BindingSets doesn't explicitly use the variants. However, after calculating the solution, the program takes the time to generate the variants and check that every one of them evaluates differently than the original for at least one of the BindingSets.

The check probably takes quite a bit more time than calculating the result does. But is anyone going to care that the answer that comes in .05 seconds could have come in .01?

Self-checks are not a good substitute for planned tests. The bugs they discover come too late, and you're never sure what cases have been tried. But they're useful when the action they take is less harmful than a wrong answer from the program. They're nice in beta programs.

Assertions

```
class BindingSet {
  …
  Boolean eval(Expr expr) {
     Assertion.test(value != null);
     …
  }
  …
```

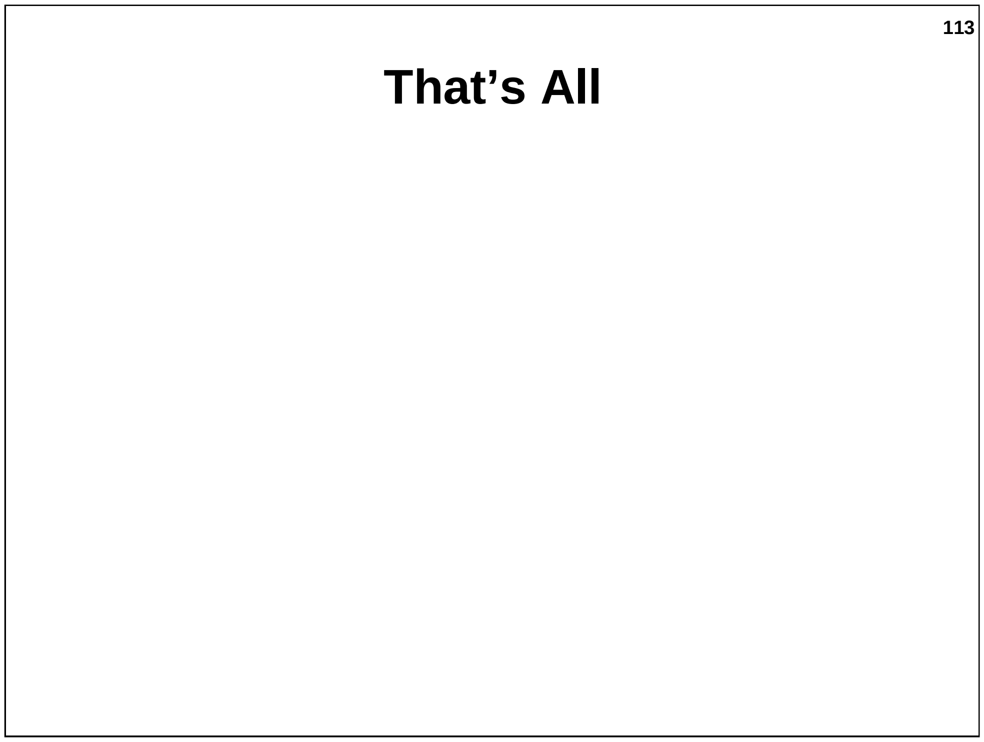
The code in the previous slide checked a **postcondition**, something that must be true of the result. Checking postconditions is good when it can be done affordably, when the code that checks isn’t likely to share errors with the code that creates the result, and when something useful can be done when the postcondition isn’t satisfied.

You can also check a **precondition**, something that the caller is supposed to ensure. In the case on the slide, the caller is not supposed to send in a null. Rather than trusting, I verify. If the test fails, the Assertion.test method throws an AssertionFailed exception.

The code used in the check is an Assertion package I wrote some years ago. You can find it at <http://www.visibleworkings.com/trace>.

(In actual fact, I don’t have an assertion at that point in the program, since the code immediately following will obviously throw its own exception when it gets the null pointer.)

Bertrand Meyer’s *Object-Oriented Software Construction* had a good discussion of preconditions and postconditions in the first edition. I haven’t read the second edition. His Eiffel language has them built in.



That’s All

This is the end of my material. Thank you for reading.

Where’s the Catalog?

- **www.testing.com/writings/short-catalog.pdf**