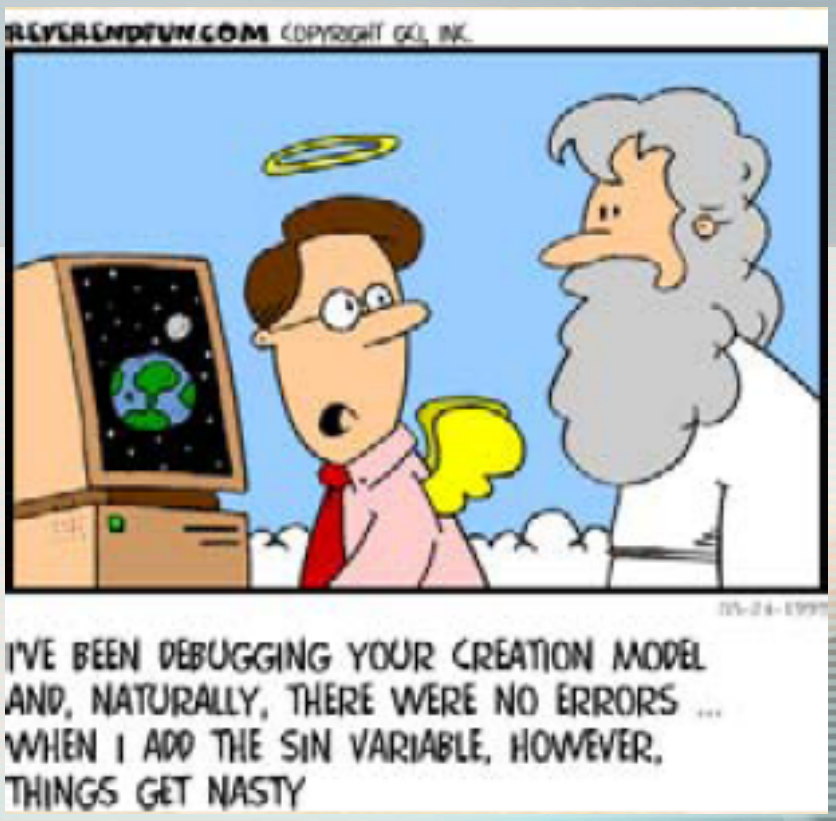
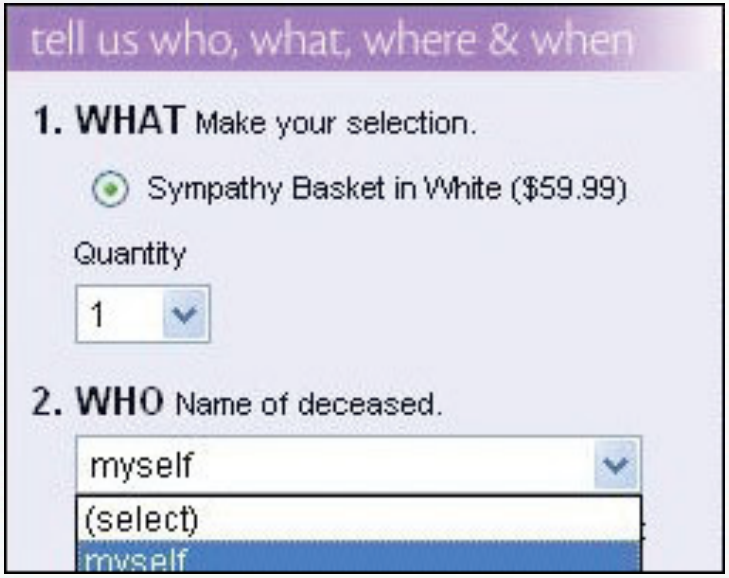
Debugging


I'VE BEEN DEBUGGING YOUR CREATION MODEL AND, NATURALLY, THERE WERE NO ERRORS ... WHEN I ADD THE SIN VARIABLE, HOWEVER, THINGS GET NASTY



"We are working feverishly on writing and documenting the Tango-PCB error messages. In the meantime, please try not to make any mistakes." —*Preliminary documentation for Tango, a printed circuit board design software program by Accel Technologies.*

Administration

- **Assignment 2b due Thursday**

Programming is great stuff

- It is the biggest, best, and most flexible "set of building blocks", "modeling clay", & "Erector set" the world has ever known
- It's magic
  - You can create immensely useful things pretty much out of thin air simply by writing down the proper incantations

But programming is hard (really hard)

- You have to get a lot of things exactly right
  - not just mostly right
- There are more details (which have to be exactly right) than you can humanly deal with

→all programmers make mistakes (lots of them)

→Programming tends to be a humbling experience (it makes you feel stupid on a regular basis)

The hard part of programming

- Creating the code is tedious, but after some practice not the hard part
- Hard part starts when the program doesn't do what you think it should
  - Finding those mistakes → debugging

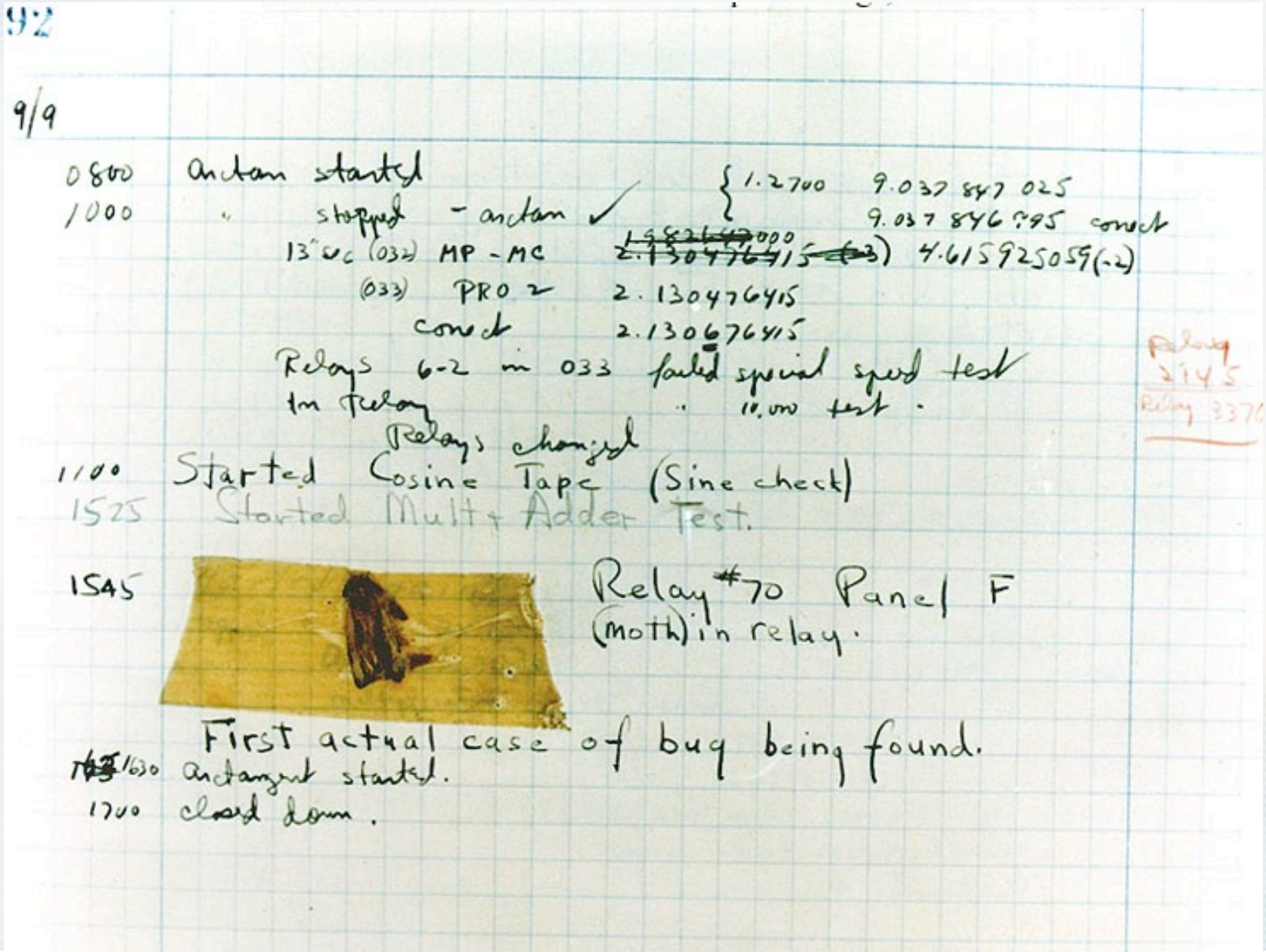
Bug Origins

- **Shakespeare**
  - **Henry VI, King Edward: "So, lie thou there. Die thou; and die our fear; For Warwick was a *bug* that fear'd us all."**
- **Edison**
  - **Denotes "flaw in a mechanism"**

Origin of "debugging"

- The "first computer bug"
  - Grace Hopper, working on one of the first electronic computers (Harvard Mark II, 1945)
  - Machine failed and operator found a moth caught in a relay
  - Taped into log book with note: "First actual case of bug found"

Origin of "debugging"



92

9/9

0800 Arctan started
1000 " stopped - arctan ✓
13" wc (032) MP - MC
(033) PRO 2
correct
Relays 6-2 in 033 failed special speed test
In relay " " 11.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1700 closed down.

1.2700 9.037 847 025
9.037 846 795 correct
4.615925059(-2)
2.13047641 5
2.13047641 5
2.13067641 5

Relay 3145
Relay 3370

Debugging

- **Confirming things you know should be true until you find one that is not**

Why Debug?

- **Hard, but critically important**
- **Any bug not detected in the design phase will cost ten times more to detect at the coding phase**
  and an additional ten times more at the debugging phase
- **Almost nothing kills usability faster than bugs**

Debugging: Famous Last Words

- "if debugging is defined as the art of taking bugs out of a program, programming must be the art of putting them in
  - Dijkstra
- "It's not a bug, it's a feature"
  - Microsoft

Thoughts on Debugging

- **"Programming is an art form that fights back."**

Thoughts on Debugging

- **"My software never has bugs. It just develops features."**

Why is Debugging Hard?

- Cause and effect may be hard to connect (remote in time/space)
- Symptoms may seem random (result of 2 bugs interacting)
- Complex interactions
- Psychological issues: frustration, pressure, guilt

The Five P's

Prior
Planning
Prevents
Poor
Performance

- Design your application before you write any code

Expertise

- No substitute for practice and experience
- Best and most experienced programmers can be 20x more productive than least experienced
  - Very unusual to see that much performance spread in a professional activity!
  - A lot to be gained by practice

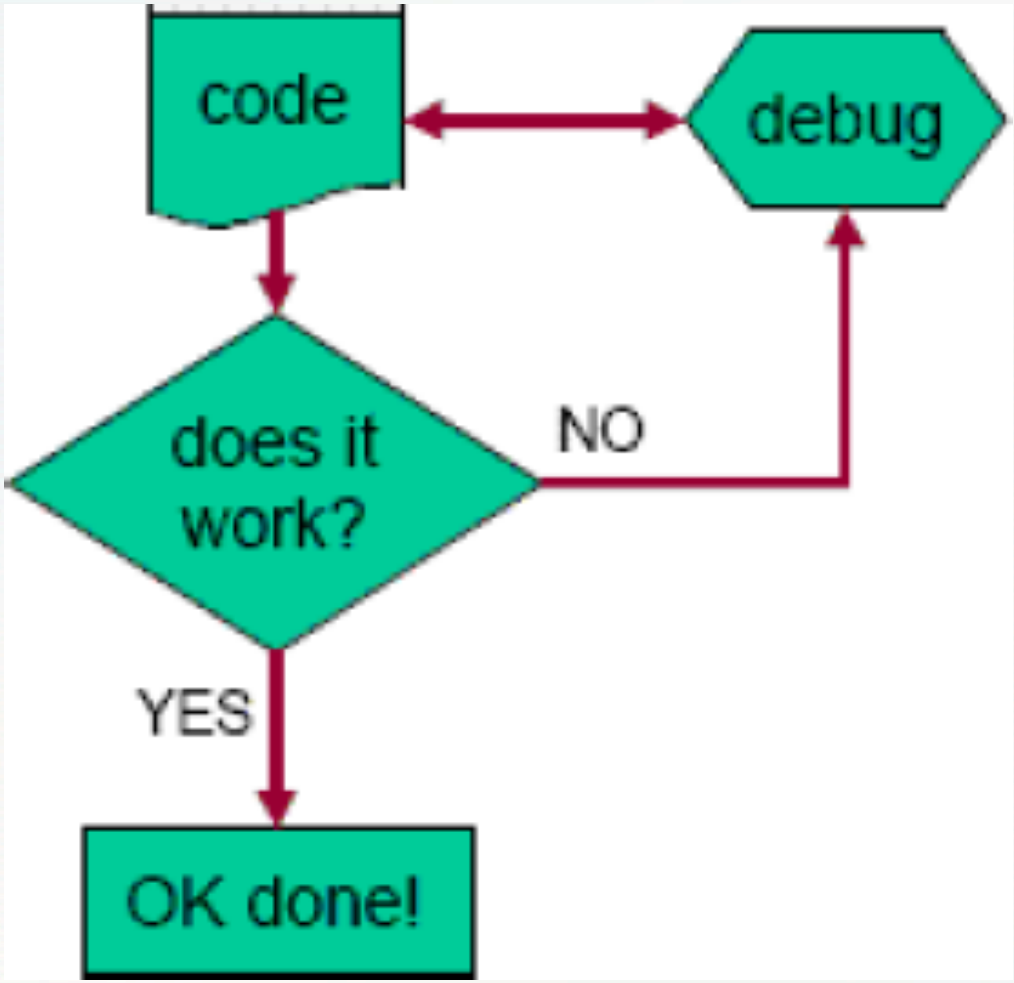
  →Your work can pay off

  →There is something to look forward to

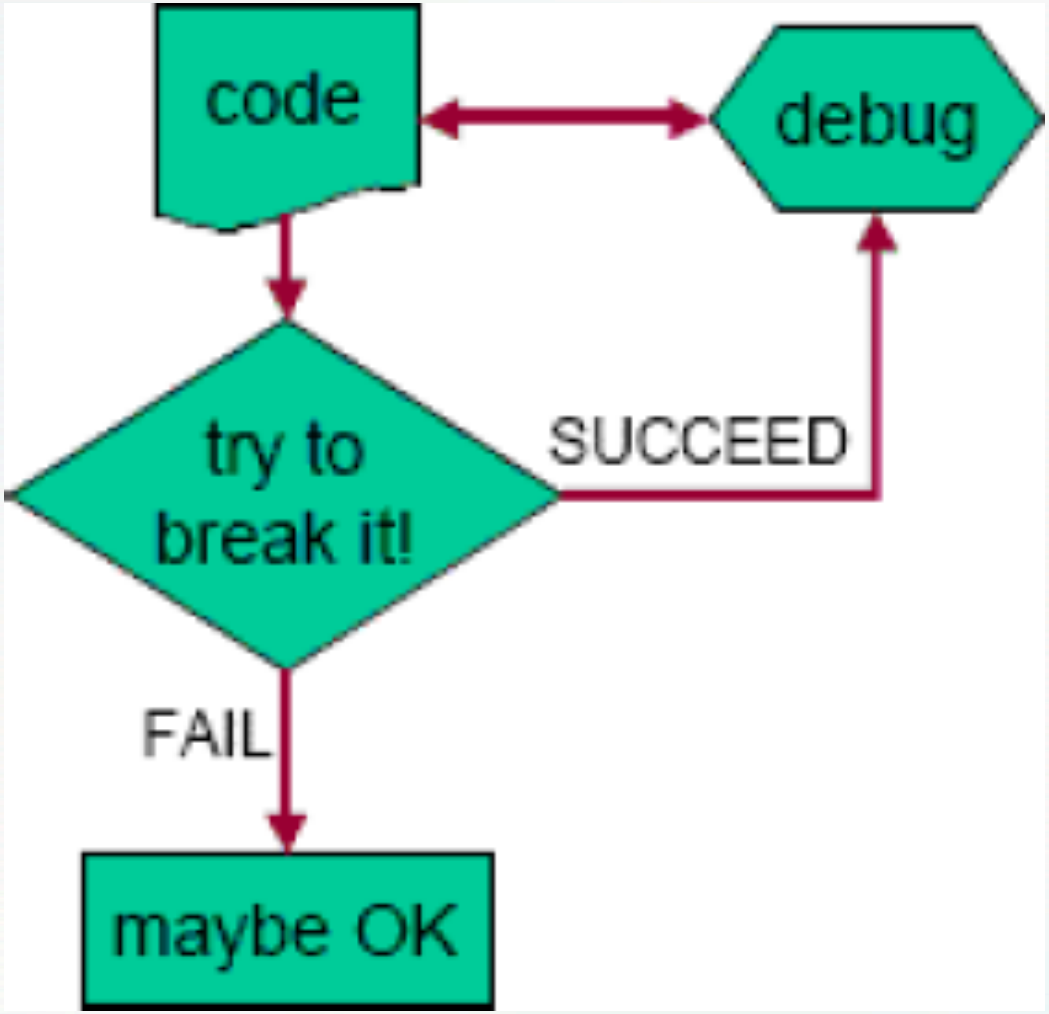
Debugging principles

- Bugs caught early (right after you make them) are pretty easy to fix
  - Details are in mind
  - You know where to look
- Once bugs "escape" (not part of what you just worked on) they get much harder to find
  - Details not in mind
  - Much wider search space

Debug Cycle



Bug-Seeking Cycle



Debugging tips

#1: Catch your bugs early

Debugging tips

#1: Catch your bugs early

#2: Only have one bug at a time!

Debugging tips

#1: Catch your bugs early

#2: Only have one bug at a time!

change 1 thing at a time

Debugging tips

#1: Catch your bugs early

#2: Only have one bug at a time!

change 1 thing at a time

⇨Tip #0: "Test early, test often, test, test, test"

Debugging tip #3

Where's the bug?
It's probably in the code you were just working on!

- **Look there first (but not just there…)**
- **If you can keep this true most of the time you will be a happy programmer**

→ **Test, test, test**

Testing is "preemptive debugging"

Debugging tip #4

- Work in small chunks (then test)
  - Make "code you just worked on" manageably
  - After a small chunk verify that it is doing what you think it is supposed to
    - Note: you need to know very clearly what it is supposed to be doing
    - Devise a list of things that should be true (e.g., this var should have this value here) and show yourself that they really are
  - Start with chunks of maybe 10-15 lines of code (later increase as the rate of errors in your code drops)

Debugging tip #5

- “Throw away a lot of your code”

Debugging tip #5

- "Throw away a lot of your code"
- A bunch of the code you write should be just for testing
  - To verify that your code does what you think it should
    - E.g., debugging print statements
      (`debug.writeln(`*<string>*`)`)
  - Removed once you're reasonably sure and can move on
    Note: do remove this code, otherwise output gets unmanageable

Types of bugs

- One way to classify
  - Coding
  - Logical
  - Architectural or Design

Types of bugs

- Coding bugs
  - Most common, easiest to debug
  - "Slip": code is written incorrectly
    - 2+3*5 when you meant (2+3)*5
  - Often result of not fully understanding programming language constructs

Types of bugs

- Logical bugs
  - Steps undertaken don't solve the problem or carry out the task correctly
    - E.g., loop ends too late (bug manifests as array index error)
  - Harder to find and fix

Types of bugs

- **Design bugs**
  - **Program does what it was designed to do, but that's not the right thing**
  - **Much harder to find and fix**
  - **Typically have to "start over"**
- **Recall that almost all "usability bugs" are "design bugs" in traditional sense**
  - **→ programmers will naturally tend to "hate you" (so do your own prototyping)**

Types of bugs

- Another way to classify
  - Syntax errors
  - Run-time errors
  - Logical errors

Types of bugs – "syntax" errors

- Caught by compiler (typically won't compile)
- Mistyped or incorrect usage
  - E.g., wrong number of parameters
- IDEs provides nice environment for dealing with these
  - Underlines what it doesn't understand on the fly
  - Task list shows list of compile problems

Fixing syntax errors

- **Read the message carefully**

  → understand it
  perhaps look up message in help
- **Start from the top of the list**
  - **Often one error "cascades" causing others later (fix of first fixes others)**
- **Look for easy errors to correct (e.g., spelling mistakes)**

Types of bugs – run-time errors

- Application tries to perform operation that is not allowed
  - Divide by zero
  - Array index out of bounds
  - Add a string to an integer

Fixing run-time errors

- IDEs do a good job of showing where bad operation occurs
  - But this spot may not be where real error (real cause) is
  - Start here and mentally/visually trace backwards in the program
    - E.g., How did the value involved in this RT-error get into this variable?

Types of bugs – "logical" errors

- Application compiles and executes without error, but doesn't produce expected results
- Most difficult to track down
- Most debugging efforts are focused on tracking down logic errors introduced by the programmer

Fixing "logic bugs"

- Key is clear idea of what is supposed to happen
  - First verify that this is really happening
  - Assuming it is, figure out what about it what's happening is incorrect (not producing the desired result in this case)

Fixing "logic bugs"

- Print statements used to make flow and key values visible
  - Can also use debug tools in environment
- Create test input that exhibits error and pour over print trace
  - Figure out what you think is supposed to happen with this input
  - Verify that this is happening
    - May take several tries inserting prints
  - Rethink whether details of current logic has flaw that leads to error in this case

- **BREAK – 15 minutes**

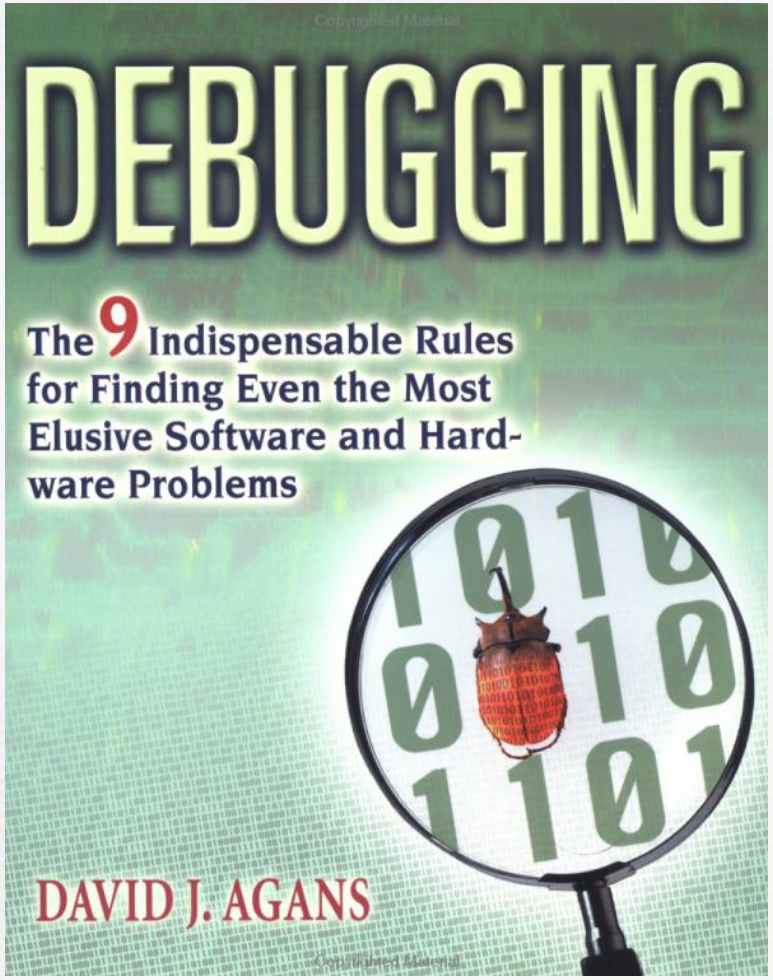
Debugging Steps

- Plan your attack
- Back up files
- Isolate code and modules
- Find the error
- Fix – don’t patch
- Test the fix
- Look for similar bugs
- Document the bug

Finding the Bug

- **Characterize the bug:**
  How do you know it fails?
- **Localize the bug:**
  Where does it fail?
- **Isolate the bug:**
  What circumstances cause the bug to appear?

Debugging Rules (www.debugging



Debugging steps and strategies

- Reproducibility
  - First step in fixing a bug is being able to reproduce it
  - Can’t fix it if you can’t make it happen on demand
    - Find conditions where it occurs and produce a data set or test framework that exhibits the bug
    - Need to understand cause and effect before you start changing your code

Debugging steps and strategies

- Reduction
  - Find the smallest / simplest dataset or test that exhibits the bug
    - Reduce the problem to its essence
    - Bugs are not random, there are caused by something (somewhere) that you need to find
      - Reduce search space by reducing complexity of test case

Debugging steps and strategies

- Deduction – a primary weapon
  - What components are involved
  - What path is program taking
  - What is the difference between working input and non-working
  - Reduce scope of possibilities by forming new hypotheses and eliminating them
    - finding evidence against them or verifying them → finding the bug

Debugging steps and strategies

- **Isolation**
  - **Often useful to think about finding *where* the bug is**
    - **If you can't find the bug where you are, you're in the wrong place**
    - **Remember it's probably in the code you just wrote, so look there first**

Debugging steps and strategies

- **Isolation – one strategy: "cutting the code in half"**
  - **Find point where problem has manifested (e.g., variable has bad value)**
  - **Find a point before that and look at conditions there**
    - **If manifested there, bug is before that point**
    - **If not, bug is after that point**
    - **Repeat on the "half" the bug is in**

Debugging steps and strategies

- Isolation – Related strategy: "commenting out code"
  - "cut in half" by commenting out a section of code or a call
    - If bug still happens it wasn't in the commented out area
    - If bug stops, it was
  - But note that commenting out arbitrary code can break other things

Debugging steps and strategies

- Isolation – another strategy: Traceback
  - Find point bug is manifested
  - Trace steps backwards
    - Figure out (possibly working on paper): "At this point, this should be true"
      - Verify that it is true
      - Possibly with aid of additional info (e.g., prints)

Another strategy

- Often helps to debug with someone else
  - They don't necessarily need to understand your program
  - Process of explaining it to them is often very helpful
  - "Outsider" may be able be able to see wrong assumptions you've made

Experience

- Get better at previous steps
- Also recognize previously seen bugs
- Leverage experience of others: programmers and Web

Tenacity


Angst Technology
by Barry T. Smith
YOU HAVE DISHONORED ME FOR TOO LONG.
EVEN THOUGH FOR EVERY ONE OF YOU I STRIKE DOWN, ANOTHER TWO MAY TAKE HIS PLACE...
... I WILL KILL YOU ALL!
AAAAAAAAA
AAAAA
DEBUGGING IS A STATE OF MIND.
WWW.INKTANK.COM
© 2000 BARRY T. SMITH

Tenacity

- No real randomness in execution
- Computers doing what you programmed them to do
- By solving bugs, gain more experience and confidence, and become much better programmer
  - Anticipate errors
  - Code in manner less likely to produce bugs

Preventive measures

- Write clean, easy to read code
- Comment your code as you write it
  - Seems like this has nothing to do with debugging, but it does
  - This helps with "figure out (exactly) what it's supposed to be doing here"

Preventive measures

- Make one change at a time
- Test code
  - Unit test: functional blocks
  - Integration test: interactions between those blocks
  - Test with invalid and valid data

Preventive measures

- **Defensive programming**
  - **A gram of prevention is worth a kilo of cure**
  - **Add error checking code and throw exceptions**
    - **If "*this* is supposed to be true *there*" put in code to test it ("assertions")**
  - **Put in "sanity checks"**
  - **Work out what assumptions (e.g., about incoming parameters or state of the system at call) must be true for proper execution and put in a run-time test for them**

Preventive Measures

- Be scientific
  - Formulate hypothesis, predict, run program, provide input, observe behavior and confirm/ refute hypothesis

Debugging issues

- Understanding
  - If you have a solution but don't understand why it works, you can't rely on it
    - May be simply hiding error
    - May be introducing new bug
      - Counteracting bugs is typically not good because they may "break separately" in other circumstances
- Partial solutions
  - If you have a solution but it doesn't solve all the problems
    - May not have found the real solution
    - May have multiple bugs

Other Important Strategies

- Prioritize which features can be omitted
- Incubate: take a break
  - Breathe and stretch
  - Drink water
- Articulate problem
- Brainstorm

Strategies

- Desk-check your code
  - Can do several hundred lines/hour
  - Best software engineers write code 99% correct
  - 1 out of 100 lines is wrong
  - We are not the best software engineers
  - Find several bugs/hour, better use of time than spending hours fixing the bugs

Strategies

- Print statements are your friend, but …
  - Cause you to edit/recompile
  - Often guess incorrectly about what variable to print
  - Often print too much, too hard to review
  - Mix of diagnostics and others hard to deal with
  - Have to eventually disable diagnostics
  - Checkpointing

Tools

- Profiler
  - Tells you where your code is spending time when executing
  - Once bug found, think about what you could have done (process-wise) that would have avoided the bug

Tools

- **Breakpoints**
  - **Stop execution of program at specific points**

Visual Studio

- Command Window: Immediate Mode
  - Enter expressions for evaluation
  - Execute statements
  - Print variable values
  - Change variable values
- Breakpoints
  - Regular, conditional

Visual Basic

- Bookmarks
- Find all references to an object
- Debug class
  - WriteLine
  - Indent
  - Assert(clause, message)

Things not to do

- Ignore errors and hope they go away
  - Can't "let them escape"
  - They won't go away on their own, just multiply when combined
- Make random changes
  - Need a strategy
- Run program over and over hoping that it will start working

- **Need an approach**
- **Need confidence**

Hope for Inexperienced

- Studies of experienced programmers have found that there is a 20-to-1 difference in the time it takes for an experienced programmer compared to an inexperienced programmer to find the same set of errors

→ get better over time

- **Questions?**