



Debugging

Shin-Jie Lee (李信杰)

Assistant Professor

Computer and Network Center

Department of CSIE

National Cheng Kung University



What is BUG?

- ❑ Things the software does that it is not supposed to do, [or] something the software doesn't do that it is supposed to. [Telles and Hsieh]
- ❑ A **software bug** is an **error**, **flaw**, **mistake**, **failure**, or **fault** in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways. [From Wikipedia]
- ❑ 1. Synonym of *defect*. 2. Synonym of *failure*. 3. Synonym of *problem*. 4. Synonym of *infection*. [Andreas Zeller]

[Telles and Hsieh] Telles, Matt and Yuan Hsieh. *The Science of Debugging*. Scottsdale: Coriolis, 2001.

[Andreas Zeller] Andreas Zeller. *Why Programs Fail*, Second Edition: A Guide to Systematic Debugging, 2009



The First "Computer Bug" Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1947.

9/9


0800 Antan started
 1000 " stopped - antan ✓

1300 (032) MP - MC $\begin{cases} 1.2700 & 9.037847025 \\ 2.130476415 & 9.037846995 \end{cases}$ correct
 (033) PRO 2 $\begin{cases} 2.130476415 \\ 2.130676415 \end{cases}$
 correct

Relays 6-2 in 033 failed special speed test
 in relay " 10.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)
 1525 Started Multi-Adder Test.

1545  Relay #70 Panel F
 (moth) in relay.

First actual case of bug being found.

1630 antan started.
 1700 closed down.

Relay 3345
 Relay 3376





Bug-Free Software?





Ineffective Approaches to Debugging₁

❑ Find the defect by guessing

- Scatter print statements randomly throughout a program
- If you can't find the defect with print statements, try changing things in the program until something seems to work
- Don't back up the original version
- Programming is more exciting when you're not quite sure what the program is doing



Ineffective Approaches to Debugging₂

❑ Don't waste time trying to understand the problem

- It's likely that the problem is trivial, and you don't need to understand it completely to fix it
- Simply finding it is enough



Ineffective Approaches to Debugging₃

❑ Fix the error with the most obvious fix

- Fix the specific problem you see, rather than wasting a lot of time making some big, ambitious correction that's going to affect the whole program.

- An example:

```
x = compute(y)
If(y==17)
    x=25.15    -- compute() doesn't work for y=17, so fix
it
```



Ineffective Approaches to Debugging₄

❑ Debugging by Superstition (**The attitude in debugging**)

- If you have a problem with a program you've written, it's your fault. It's not computer's fault, and it's not the compiler's fault.
- Even if an error at first appears not to be your fault, it's strongly in your interest to assume that it is
- **It's hard enough to find a defect when you assume your code is error-free**



The Scientific Method of Debugging

1. Stabilize the error (Refine the test cases that produce the error)
2. Locate the source of the error
 - a. Gather the data that produces the defect
 - b. Analyze the data that has been gathered, and form a hypothesis about the defect
 - c. Determine how to prove or disprove the hypothesis, either by testing the program or by examining the code
 - d. Prove or disprove the hypothesis by using the procedure identified in 2(c)
3. Fix the defect
4. Test the fix
5. Look for similar errors



Stabilize the Error

- ☐ The defect is easier to diagnose if you can stabilize it
 - that is, make it occur reliably
- ☐ To find test cases that produces the error



An Example–Debugging

- ☐ Assume that you have an employee database program that has an intermittent error.
- ☐ The program is supposed to print a list of employees and their income-tax withholdings in alphabetical order.



Stabilize the Error–An Example

□ Test case 1:

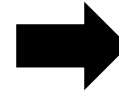
Formatting, Fred Freeform	\$5,877
Global, Gary	\$1,666
Modula, Mildred	\$10,788
Many-Loop, Mavis	\$8,889
Statement, Sue Switch	\$4,000
Whileloop, Wendy	\$7,860

The first run

□ Test case 2:

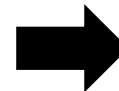
Fruit-Loop, Frita	\$5,771
Formatting, Fred Freeform	\$5,877
Global, Gary	\$1,666
Many-Loop, Mavis	\$8,889
Modula, Mildred	\$10,788
Statement, Sue Switch	\$4,000
Whileloop, Wendy	\$7,860

The first run



Formatting, Fred Freeform	\$5,877
Global, Gary	\$1,666
Many-Loop, Mavis	\$8,889
Modula, Mildred	\$10,788
Statement, Sue Switch	\$4,000
Whileloop, Wendy	\$7,860

The second run



Formatting, Fred Freeform	\$5,877
Fruit-Loop, Frita	\$5,771
Global, Gary	\$1,666
Many-Loop, Mavis	\$8,889
Modula, Mildred	\$10,788
Statement, Sue Switch	\$4,000
Whileloop, Wendy	\$7,860

The second run



Locate the Source of the Error₁

- ❑ In the example, you may hypothesize: *the problem has something to do with entering a single new employee*
- ❑ Then, you test the hypothesis with a new test case:

Formatting, Fred Freeform	\$5,877
Fruit-Loop, Frita	\$5,771
Global, Gary	\$1,666
Hardcase, Henry	\$493
Many-Loop, Mavis	\$8,889
Modula, Mildred	\$10,788
Statement, Sue Switch	\$4,000
Whileloop, Wendy	\$7,860

However, it is fine in the first run, which means that the hypothesis is false¹³



Locate the Source of the Error₂

- ❑ Then, you look at the code and find the problem:
 - Two different sorting routines are used. One is used when an employee is entered, and another is used when the data is saved.
 - The data is printed before it's sorted
- ❑ You later confirm this hypothesis with additional test cases



Tips for Finding Defects₁

❑ Use all the data available to make your hypothesis

- When creating a hypothesis about the source of a defect, account for as much of the data as you can in your hypothesis

❑ Refine the test cases that produce the error

- You might be able to vary one parameter more than you had assumed, and focusing on one of the parameters might provide the crucial breakthrough



Tips for Finding Defects₂

❑ Exercise (**test**) the code in your unit test suite

- Defects tend to be easier to find in small fragments of code than in large integrated programs. Use your unit tests to test the code in isolation.

❑ Use available tools

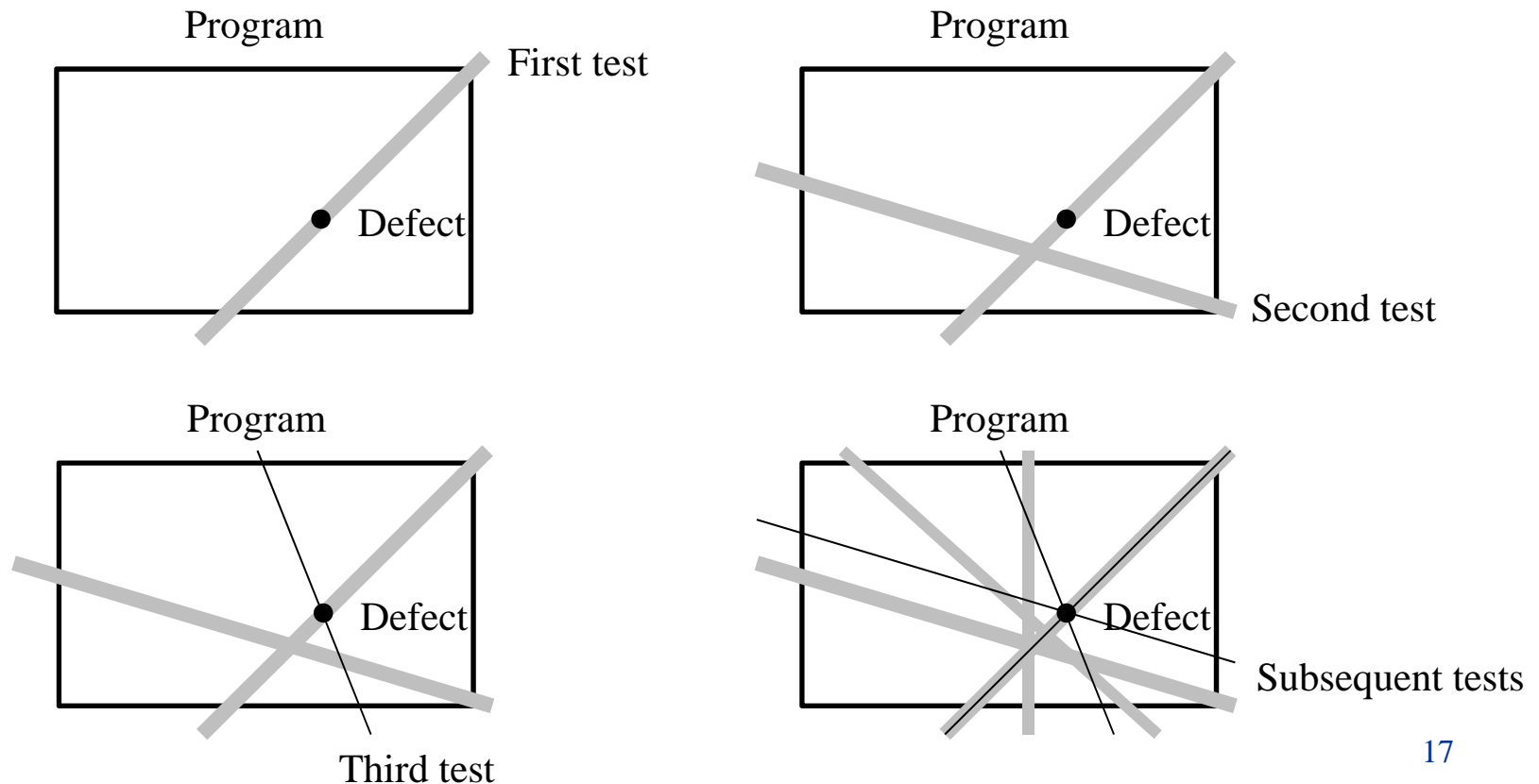
- With one tough-to-find error, for example, one part of the program was overwriting another part's memory.
 - This error was difficult to diagnose using conventional debugging practices.
- To use tool, for example, **Eclipse Java Debugger**.



Tips for Finding Defects₃

❑ Reproduce the error several different ways

- If you can get a fix on it from one point and a fix on it from another, you can better determine exactly where it is.





Tips for Finding Defects₄

❑ Generate more data to generate more hypotheses

- Choose test cases that are different from the test cases you already know to be erroneous or correct.
- Run them to generate more data, and use the new data to add to your list of possible hypotheses.

❑ Use the results of negative tests

- Suppose that a test case disproves your hypothesis, so you still don't know the source of error.
- However, you do know that the defect is not in the area you thought it was. That narrows your search field and the set of remaining possible hypotheses.



Tips for Finding Defects₅

❑ Brainstorm for possible hypotheses

- Rather than limiting yourself to the first hypothesis you think of, try to come up with several

❑ Keep a notepad by your desk, and make a list of things to try

- One reason programmers get stuck during debugging sessions is that they go too far down dead-end paths.
- Make a list of things to try, and if one approach isn't working, move on to the next approach



Tips for Finding Defects₆

❑ Narrow the suspicious region of the code

- Rather than removing regions haphazardly, divide and conquer
- Use a binary search algorithm to focus your search

❑ Be suspicious of classes and routines that have had defects before

- Classes that have had defects before are likely to continue to have defects



Tips for Finding Defects₇

❑ Check code that's changed recently

- If you can't find a defect, run an old version of the program to see whether the error occurs
- Check the version control log to see what code has changed recently

❑ Expand the suspicious region of the code

- If you don't find the defect in a focused small section of code, consider the possibility that the defect isn't in the section



Tips for Finding Defects₈

☐ Integrate incrementally

- If you add a piece to a system and encounter a new error, remove the piece and test it separately



Tips for Finding Defects₉

❑ Check for common defects

- Use checklists of the common problems in your environment to stimulate your thinking about possible defects

CHECKLIST (Example)

Initializing Variables

- Does the code initialize variables as they're declared, if possible?
- Are counters and accumulators initialized properly and, if necessary, reinitialized each time they are used?

Using Conditionals

- Is the *else* clause correct?
- Are the *if* and *else* clauses used correctly-not reversed?

Loops

- Does the loop end under all possible conditions?
- If *break* or *continue* are used, are they correct?

Recursion

- Does the recursive routine include code to stop the recursion?
- Is the routine's depth of recursion within the limits imposed by the size of the program's stack?



Tips for Finding Defects₁₀

❑ Talk to someone else about the problem (confessional debugging)

- You often discover your own defect in the act of explaining it to another person

❑ Take a break from the problem

- Sometimes you concentrate so hard you can't think
- The auxiliary benefit of giving up temporarily is that it reduces the anxiety associated with debugging