CS301 Software Development

Instructor: Peter Kemper

Debugging

Overview

- Debugging and its Issues
- Finding a Defect
- Fixing a Defect
- Psychological Considerations in Debugging
- Debugging Tools

Photo: flickr, creative commons: [phil h]

- For a tutorial on the Eclipse Debugger
  - see Mark Dexter's video lectures, at least session 1 & 2.

Bugs vs Errors or Faults

- A bug in software means a programmer made a mistake!
- This isn't cute and could lead to this:

Figure: Steve McConnel, Code Complete 2nd Ed

- Why is Debugging worth a look?

	Fastest Three Programmers	Slowest Three Programmers
Average debug time (minutes)	5.0	14.1
Average number of defects not found	0.7	1.7
Average number of defects made correcting defects	3.0	7.7

Source: "Some Psychological Evidence on How People Debug Computer Programs" (Gould 1975)

Sweet Memories?

- Do you remember a programming mistake you made?
- One that took you a long time to figure out?
- One you swore to yourself: never again?

Photo: flickr, creativecommons
Michael Sarver

Sweet Memories?

- Do you remember a programming mistake you made?
- One that took you a long time to figure out?
- One you swore to yourself: never again?

Errors are an opportunity to grow!

Defects as Opportunities

Photo: flickr creative commons gbohne

Learn about

- the program you are working on
- the kinds of mistakes you make
- the quality of your code from the point of view of someone who has read it
- learn about how you solve problems
- how you fix defects

FINAL WARNING DON'T DO THIS

(Incomplete) list of really stupid strategies to apply

- Find the defect by guessing
  - unsystematic print statements and wild unorganized changes with no backup of original version
- Don't waste time trying to understand the problem
  - most likely the problem is simple, anything deeper wastes time
- Fix the error with the most obvious fix
  - just fix it and call it a day

```
x = Compute( y )
if ( y = 17 )
   x = $25.15       -- Compute() doesn't work for y = 17, so fix it
```

- Debugging by superstition
  - claim that root causes are elsewhere: unstable OS, mysterious compiler defects, hidden language defects

Finding a Defect

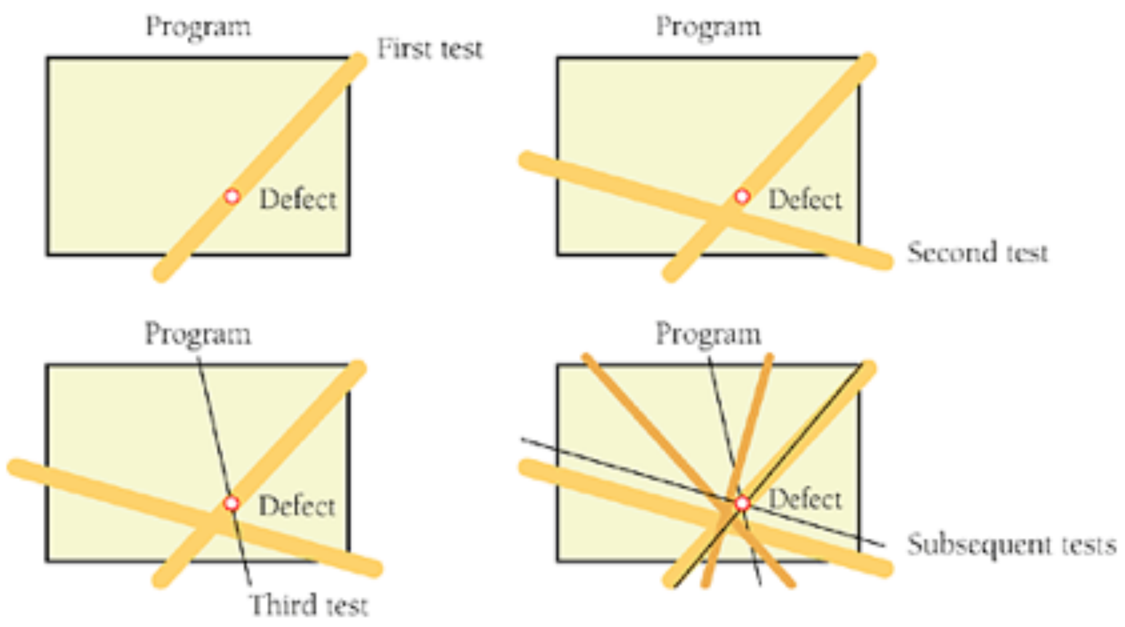
- From the classic scientific method
  1. Gather data through repeatable experiments
  2. Form a hypothesis that accounts for the data
  3. Design an experiment to prove/disprove hypothesis
  4. Prove/disprove hypothesis
  5. Repeat as needed
- The "Scientific Method of Debugging"
  1. Stabilize the error
  2. Locate the source of the error ("the fault")
     a. Gather data that produces the defect
     b. Analyze the data and form a hypothesis about the defect
     c. Determine how to prove/disprove the hypotheses, either by testing the program or by examining the code
     d. Prove/disprove the hypothesis by using procedure 2.c
  3. Fix the defect
  4. Test the fix
  5. Look for similar errors

Stabilize the Error

- If difficult to reproduce
  - check list of prime suspects for random values during execution
    - variable used without initialization
    - variable refers to freed memory that is used elsewhere
    - multithreading: race conditions
    - multithreading: timing issues, order of execution
    - execution of program not memory less, state in persistent storage
- Find a test case that reproduces the error each time executed
- Vary the test case to recognize which factor influences the error and which ones do not.
- Simplify the test case to the smallest amount of factors and data and functionality executed to produce the error

Locate the Source of the Error

- Brainstorm to produce a set of hypothesis
- Create experiments, additional tests to obtain data to prove/disprove hypothesis and to locate the error



- Note:
  - If you're having a hard time finding a defect, it could be because the code isn't well written.

Tips for Finding Defects (an incomplete series, part I)

- Keep a notepad by your desk and make a list of things to try
- Brainstorm for possible hypotheses
- Exercise the code in your unit test suite
- Use available tools (interactive debuggers, memory checkers, picky compilers, static code checkers)
- Refine the test cases that produce the error
- Reproduce the error in several different ways
- Generate more data to generate more hypothesis
- Use the results of negative tests

Tips for Finding Defects (an incomplete series, part II)

- Narrow the suspicious region of the code
- Be suspicious of classes and routines that have had defects before
- Check code that's changed recently
- Expand the suspicious region of code
- Integrate incrementally
- Check for common defects
- Talk to someone else about the problem ("Confessional debugging")

Photo: flickr, creativecommons: anyjazz65

Brute Force Debugging (an incomplete series, part I)

- Design/Review/Recode
  - Perform a full design and/or code review on broken code
  - Throw away the whole program or the suspected section of code and redesign/recode from scratch
- Compile code
  - with full debugging information
  - at pickiest warning level and fix all the picky compiler warnings
- Testing
  - Strap on a unit test harness and test suspected code in isolation
  - Create an automated test suite and run it all night
- Debugging
  - Step through a big loop in the debugger manually until you get to the error
  - Instrument code with print/display/logging information

Brute Force Debugging (an incomplete series, part II)

- Environment and tools
  - Compile code with different compiler
  - Compile and run program in a different environment
  - Link/run code against special libraries or execution environments that produce warnings if code is used incorrectly.
  - Replicate the end-users full machine configuration
- Bottom up
  - Integrate new code in small pieces, full testing each piece as it's integrated

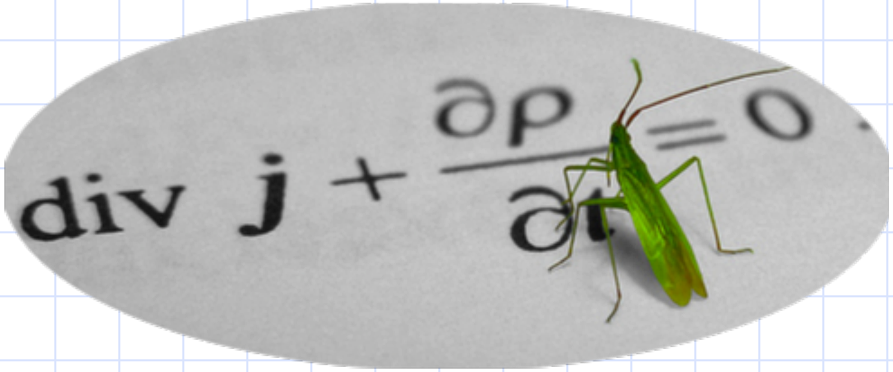


Photo: flickr, creativecommons vojtisek

Quick & Dirty Debugging vs Brute Force

- Quick & Dirty Debugging
  - Build hypothesis and check this
  - Building a hypothesis is not systematic, we perform an unsystematic search for a needle in the haystack.
- Brute force technique
  - More work but guaranteed to succeed
- Recommendation
  - Identify a list of possible brute-force techniques before you start debugging
  - Set a maximum time limit for quick & dirty debugging
  - Once that is passed, go for the brute force approach.

A few notes on syntax errors (and if editor is not smart)

- Don’t trust line numbers in compiler messages
  - also check the lines around it, in particular the ones before
- Don’t trust compiler messages
  - there is a problem, but the wording is sometimes misleading
- Don’t trust the compiler’s second message
  - once something is wrong (first message)
  - more messages could just come from that, fix first error first
- Divide and conquer
  - partition program into sections (use comments) and compile individual sections
- Find misplaced comments and quotation marks
  - For C,C++,Java, insert the following `/*"/**/` which will terminate either a comment or a string.

Fixing a Defect

- Finding an error is hard, fixing it is often easy

  => pitfall: being easy makes us careless

  => fixing an error introduces new error(s)

- Recommendations:
  - Understand the problem before you fix it
  - Understand the program, not just the problem
  - Confirm the defect diagnosis
  - Relax
  - Save the original source code
  - Fix the problem, not the symptom
  - Change the code only for good reason
  - Make one change at a time
  - Check your fix
  - Add a unit test that exposes the defect
  - Look for similar defects

Psychological Considerations

- Debugging Blindness thanks to "Psychological Set"
- Brain makes shortcuts based on expectations
- Here:
  - Students expect "while" condition to be checked all the times
  - Programmer uses 2 variables SYSTSTS and SYSSTSTS instead of 1
  - Programmer read this

```
if ( x < y )
    swap = x;
    x = y;
    y = swap;
```

for this

```
if ( x < y ){
    swap = x;
    x = y;
    y = swap;
}
```

Psychological Considerations

- Once understood, more clear why good programming practices are necessary.
  - Formatting
  - Commenting
  - Variable names
  - Routine names
  - such that likely defects appear as variations and stick out
- "Psychological Distance" can help

First Variable	Second Variable	Psychological Distance
stoppt	stcppt	Almost invisible
shiftrn	shiftrm	Almost none
dcount	bcount	Small
claims1	claims2	Small
product	sum	Large

Debugging Tools

Most prominently

- Interactive Debuggers
  - step by step execution
  - break points
  - examination of data

Others

- Source code comparators, like Diff
- Compiler Warning Messages
- Static Code Checker, like Findbugs, PMD
- Execution Profiler
- Test Frameworks

Photo: flickr, creativecommons cmpalmer

Debugging Tools

Interactive Debuggers are subject to a variety of opinions:

"An interactive debugger is an outstanding example of what is not needed—it encourages trial-and-error hacking rather than systematic design, and also hides marginal people barely qualified for precision programming." -- Harlan Mills

I respect Harlan Mills for his contribution to SWE and his clean room approach. But I clearly disagree on this, errors happen and we need to use all available tools at hand and in addition to our brains to find and fix them.

Eclipse Debugger

- Tutorials:
  - Check Mark Dexter's Tutorials on the Eclipse debugger
- Basics:
  - Step by step execution, break points
  - Exploration of variables, data structures and stack frames
- Advanced
  - Expressions
  - More on breakpoints
    - Exception breakpoints
    - Conditional breakpoints, hit counts
    - Watch points (Field breakpoints)
  - On-the-fly: change variables, hot code replacement

Summary

- Debugging is a make-or-break aspect of SW development.
  - Best approach: use techniques to avoid defects in the first place.
  - Debugging skills are worth developing: can save order of mag in time.
- Critical to success: be systematic.
  - Focus your debugging so that each test moves you a step forward.
  - Use the Scientific Method of Debugging.
- Understand the root problem before you fix the program.
  - Random guesses about the sources of errors and random corrections will leave the program in worse condition than when you started.
- Set your compiler warning to the pickiest level possible, and fix the errors it reports.
  - It's hard to fix subtle errors if you ignore the obvious ones.
- Debugging tools are powerful aids to software development.
  - But no substitute for your own brain!