Today

• Part 1 (Lecture)(~80 mins)
  • Debugging
• Break!

• Part 3 (Discussion)(45 mins)
  • Discussion of readings
Steps in fixing bugs

• Reproduce the problem
• Find cause of defect
• Investigate fix
• Implement fix
• Test fix

• Will focus on finding cause of defect today
Edit / Debug Cycle

Circle size: % of time
Edge thickness: % of transitions observed

For tasks in code in your own codebase that you haven’t seen recently

LaToza and Myers. Developers ask reachability questions. ICSE 2010.
Debugging process model

Bug reports
Source code comprehension
Documentation

Gather context

Hypothesis to test

Set break points
Insert log statements
Delete statements

Information

Instrument hypothesis

Modified system

Test hypothesis

Step through breakpoints
Execute the program/system
Log execution data

Formulate & test hypotheses

- Use knowledge & data so far to formulate hypothesis about why bug happened
cognition, meditation, observation, inspection,
contemplation, hand-simulation,
gestation, rumination, dedication, inspiration,
articulation

- Recognize cliche
  seen a similar bug before

- Controlled experiments - test hypotheses by gathering data

## Resources for testing hypotheses

<table>
<thead>
<tr>
<th># subjects</th>
<th>Hypothesis instrumentation methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Inserting breakpoints and watch variables</td>
</tr>
<tr>
<td>4</td>
<td>Inserting log statements</td>
</tr>
<tr>
<td>2</td>
<td>Removing irrelevant code</td>
</tr>
<tr>
<td>2</td>
<td>Tweaking - modifying existing code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># subjects</th>
<th>Hypothesis testing and comparison methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Stepping in the debugger</td>
</tr>
<tr>
<td>4</td>
<td>Comparing against examples</td>
</tr>
<tr>
<td>2</td>
<td>Comparing against an oracle</td>
</tr>
<tr>
<td>1</td>
<td>Analyzing network packets</td>
</tr>
<tr>
<td>1</td>
<td>Backtracking</td>
</tr>
<tr>
<td>1</td>
<td>Printing out hard copies of code</td>
</tr>
</tbody>
</table>

Definitions

• Error - discrepancy between actual behavior of system and intended behavior

• Failure - incorrect output value, exception, etc.; an error that has become observable

• Fault - lines in code which are incorrect

• Debugging: determining the cause of a failure
  • May involve finding location (fault localization) as well as explanation.
## Resources used in debugging

<table>
<thead>
<tr>
<th># subjects</th>
<th>Resources used in debugging</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Debugger tools</td>
</tr>
<tr>
<td>14</td>
<td>Bug information</td>
</tr>
<tr>
<td>12</td>
<td>Communication with others</td>
</tr>
<tr>
<td>9</td>
<td>Internet resources</td>
</tr>
<tr>
<td>7</td>
<td>Custom code/manual debugging data</td>
</tr>
<tr>
<td>6</td>
<td>System state information (variables, packets)</td>
</tr>
<tr>
<td>5</td>
<td>Searching the source repository</td>
</tr>
<tr>
<td>4</td>
<td>Code browsers</td>
</tr>
<tr>
<td>3</td>
<td>Printed publications</td>
</tr>
<tr>
<td>2</td>
<td>Production health/status/monitoring systems</td>
</tr>
<tr>
<td>2</td>
<td>Build information</td>
</tr>
<tr>
<td>1</td>
<td>Personal library of technical tidbits</td>
</tr>
<tr>
<td>1</td>
<td>Shared internal development team resources</td>
</tr>
<tr>
<td>1</td>
<td>Product documentation</td>
</tr>
</tbody>
</table>

Information needs in debugging

- How did this runtime state occur? (12)
  data, memory corruption, race conditions, hangs, crashes, failed API calls, test failures, null pointers

- Where was this variable last changed? (1)

- Why didn’t this happen? (3)

omniscient debuggers

Record execution history
Provide interactions for browsing or searching

WhyLine

directly supports all 3 questions in some situations

How do I debug this bug in this environment? (3)

In what circumstances does this bug occur? (3)

statistical debugging [1]

- Sample execution traces on user computers
- Find correlations between crashes and predicates

No need to reproduce environment on developer computer

Examine correlations between crashes and predicates

How is this object **different** from that object? (1)

What runtime state **changed** when this executed? (2)

Which **team’s** component caused this bug? (1)
Which team should I assign this bug to?
Information needs in debugging

- What code could have caused this behavior?
- What's statically related to this code?
- What code cause this program state?

A: Why did I get gibberish? Storing field, given PPack, what is an MPField? I have no idea what this data structure contains. SPSField? I suspect SPS is just busted.
Activity

• What's the hardest debugging bug you've ever debugged?
• What made it hard?
What makes debugging hard?

<table>
<thead>
<tr>
<th># subjects</th>
<th>Debugging challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Environmental challenges</td>
</tr>
<tr>
<td>7</td>
<td>Multithreaded/multicore</td>
</tr>
<tr>
<td>6</td>
<td>Information quality</td>
</tr>
<tr>
<td>6</td>
<td>Communication challenges</td>
</tr>
<tr>
<td>6</td>
<td>Unable to reproduce failures consistently</td>
</tr>
<tr>
<td>4</td>
<td>Debugging process challenges</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># subjects</th>
<th>Debugging challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Capture and replay of production events</td>
</tr>
<tr>
<td>3</td>
<td>More contextual information in runtime logs/stack traces</td>
</tr>
<tr>
<td>3</td>
<td>Integrating data from different sources</td>
</tr>
<tr>
<td>3</td>
<td>Bi-directional debugger</td>
</tr>
<tr>
<td>3</td>
<td>Debugging tool training</td>
</tr>
<tr>
<td>3</td>
<td>Multithreaded support</td>
</tr>
<tr>
<td>2</td>
<td>Automatic breakpoints upon entry into a class</td>
</tr>
<tr>
<td>2</td>
<td>Automated log analysis</td>
</tr>
<tr>
<td>2</td>
<td>Program context</td>
</tr>
<tr>
<td>2</td>
<td>Visually showing the execution trace</td>
</tr>
</tbody>
</table>

What makes hard bugs hard to debug?

• Cause / effect chasm - symptom far removed from the root cause (15 instances)
  timing / synchronization problems
  intermittent / inconsistent / infrequent bugs
  materialize many iterations after root cause
  uncertain connection to hardware / compiler / configuration

• Inapplicable tools (12 instances)
  Heisenbugs - bug disappears when using debugging tool
  long run to replicate - debugging tool slows down long run even more
  stealth bug - bug consumes evidence to detect bug
  context - configuration / memory makes it impossible to use tool

• What you see if probably illusory (7 instances)
  misreads something in code or in runtime observations

• Faulty assumption (6)

• Spaghetti code (3)

What makes hard bugs hard to debug?

Table 6 Root cause of the hardest bug (number of answers given).

<table>
<thead>
<tr>
<th></th>
<th>memory</th>
<th>parallel</th>
<th>vendor</th>
<th>design</th>
<th>init</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexical</td>
<td>42</td>
<td>53</td>
<td>41</td>
<td>82</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>ambiguous</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>29</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

Table 7 Most useful technique to find the hardest bug (number of answers given).

<table>
<thead>
<tr>
<th></th>
<th>stepping</th>
<th>wrapper</th>
<th>printf</th>
<th>log diff</th>
<th>breakpoints</th>
<th>tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>reading</td>
<td>54</td>
<td>5</td>
<td>33</td>
<td>12</td>
<td>38</td>
<td>15</td>
</tr>
<tr>
<td>expert</td>
<td>41</td>
<td>4</td>
<td>58</td>
<td>31</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>experiments</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not fixed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8 Main difficulty source for hardest bug (number of answers given).

<table>
<thead>
<tr>
<th></th>
<th>distance</th>
<th>tools</th>
<th>output</th>
<th>assumption</th>
<th>bad code</th>
<th>unknown</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>87</td>
<td>47</td>
<td>1</td>
<td>33</td>
<td>38</td>
<td>35</td>
<td>62</td>
</tr>
</tbody>
</table>
Activity

• What is a strategy you've used to debug a defect?
Some debugging strategies

• Backwards: Find statement that generated incorrect output, follow data and control dependencies backwards to find incorrect line of code
• Forwards: Find event that triggered incorrect behavior, follow control flow forward until incorrect state reached
• Input manipulation: Edit inputs, observe differences in output
• Blackbox debugging: Find documentation, code examples to understand correct use of API
Traditional debugging techniques

• Stepping in debugger
• Logging - insert print statements or wrap particular suspect functions
• Dump & diff - use diff tool to compare logging data between executions
• Conditional breakpoints
• Profiling tool - detect memory leaks, illegal memory references

Debugging tools

• Make breakpoint debuggers **better**
  • Support stepping backwards (omniscient debuggers)
  • Support finding statement that generated incorrect output
• Find **part** of program that generated incorrect output (slicing)
  • Output: subset of program
• Help developers **follow** data and control dependencies backwards
  • Support for navigating control and data dependencies (WhyLine)
• **Compare** execution across different runs to guess locations that might be related (automatic debugging)
  • Output: list of potential fault locations
• **Simplify** input to find a simpler input that still generates failure (delta debugging)
  • Output: simplified input
• Enable developers to directly answer questions about **causality**
  • Support searching across control and data flow (Reacher)
• Help developers understand execution **state**
  • Show visualizations which depict the current contents of memory
• Let developers **experiment**, editing program till they find the right one (live programming)[editing code]
Program analysis building blocks

• Many tools rely on gathering an execution trace
  • Record the value of every expression as it executes (or sometimes at function boundaries)
  • Challenge: scalability

• Other tools use log data
  • Gives developer control over what is being logged
  • More easily scalable, requires developer to control what is logged

• Other tools use test coverage data
  • Which statement executes on each test, test passing or succeeding
Make breakpoint debugging better

- Debugging in a debugger is hard
  - Forces developer to guess which methods to step into
  - Forces developers to guess which values to instrument
  - Changing guess requires reproing failure again
    - Can be time consuming

- What if developers could debug forwards and backwards?
ZStep94

• Forwards / backwards stepping through execution events

• Select graphical output, find code that drew it

• See value of selected variables


Demo: http://web.media.mit.edu/~lieber/Lieberary/ZStep/ZStep.mov
Omniscient debugger

Demo / talk: http://video.google.com/videoplay?docid=3897010229726822034#

Associating incorrect output with responsible code

Figure 2. Susan first uses the Web Inspector to go from the mosaic’s visual output to its DOM elements. Then, she uses Scry to track changes to the mosaic element (a), select different visual states to inspect (b), and see the DOM tree (c) and CSS styles (d) that produced each visual state. To jump to the code that implements interactive behaviors, Susan uses Scry to compare two states and then selects a single style property difference (d). Scry shows the mutation operations indirectly responsible for causing the property difference (e), and Susan can jump to JavaScript code (f) that performed each mutation operation.

Find part of the program that caused incorrect output

• Slice
  • Subset of the program that is responsible for computing the value of a variable at a program point

• Backwards slice
  • Transitive closure of all statements that have a control or data dependency

• Originally formulated as \textit{subset} of program
Early evidence for slicing

- **BEGIN**
  READ(X, Y)
  TOTAL := 0.0
  SUM := 0.0
  IF X <= 1
    THEN SUM := Y
  ELSE BEGIN
    READ(Z)
    TOTAL := X * Y
  END
  WRITE(TOTAL, SUM)
END

- (Static) slice - subset of the program that produces the same variable values at a program point
- Slice on variable Z at 12

Participants performed 3 debugging tasks on short code snippets

Asking to recognize code snippets afterwards

---

Mark Weiser. 1982. [Programmers use slices when debugging](https://doi.org/10.1145/358162.358171), Commun. ACM 25, 7 (July 1982), 446-452.
Slicers debug faster

- Students debugging 100 LOC C++ programs
- Students given
  - Programming environment
  - Hardcopy input, wrong output, correct output
  - Files with program & input
- Compared students instructed to slice against everyone else
  - Excluding students who naturally use slicing strategy
- Slicers debug significantly faster (65.29 minutes vs. 30.16 minutes)

Dynamic slicing

Compare faulty & unfaulty execution traces

User hits bug and program crashes
Program (e.g. Microsoft Watson) logs stack trace
Stack trace sent to developers
Tool classifies trace into bug buckets

Problems

WAY too many bug reports => way too many open bugs
=> can’t spend a lot of time examining all of them
    Mozilla has 35,622 open bugs plus 81,168 duplicates (in 2004)

Stack trace not good bug predictor for some systems (e.g. event based systems)
⇒ bugs may be in multiple buckets or multiple bugs in single bucket

Stack trace may not have enough information to debug
⇒ hard to find the problem to fix

Compare faulty & unfaulty execution traces

- Program runs on user computer
  - Crashes or exhibits bug (failure)
  - Exits without exhibiting bug (success)
- Counters count # times predicates hit
  - Counters sent back to developer for failing and successful runs
- Statistical debugging finds predicates that predict bugs
  - 100,000s to millions of predicates for small applications
  - Finds the best bug predicting predicates amongst these
- Problems to solve
  - Reports shouldn’t overuse network bandwidth (esp ~2003)
  - Logging shouldn’t kill performance
  - Interesting predicates need to be logged (fair sampling)
  - Find good bug predictors from runs
  - Handle multiple bugs in failure runs

Compare faulty & unfaulty execution traces

• Predictor of what statements are related to a bug:
  \[ \text{Fail}(P) - \text{Context}(P) \]
  \[ \text{Pr}(\text{Crash} \mid P \text{ observed to be true}) - \text{Pr}(\text{Crash} \mid P \text{ observed at all}) \]

• Example of a “likelihood ratio test”

• Comparing two hypotheses
  • 1. Null Hypothesis: \( \text{Fail}(P) \leq \text{Context}(P) \)
     \( \text{Alpha} \leq \text{Beta} \)
  • 2. Alternative Hypothesis: \( \text{Fail}(P) > \text{Context}(P) \)
     \( \text{Alpha} > \text{Beta} \)

Simplify failure inducing input

- Long sequence of steps uncovered by tester triggers a bug.
- Which of these steps are causing the bug?
- Complex input - which part of input is responsible for bug?
- Example - 10,700 Mozilla bugs (11/20/2000)

Find shortest repro steps

- dadmin algorithm sketch:
  1. Decompose input into pieces
  2. Run tests on pieces
  3. If there's a piece that still fails, go back to 1 on piece
    Otherwise, found locally minimal smallest input