Mark II logbook, Sep 9, 1947

9/9

0800 Andon started
1000 Andon stopped - andon

13°C (0.5% MP - MC)

03% PRO2 2.130476761

Conv 2.130476761

Relays 6-2 in 03% failed special speed test
In relay
Relays changed

1100 Started Cosine Tape (Sine chest)
1225 Started Multi Adder Test.
1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.
1700 Andon restarted.
1700 closed down.
Debugging

UW CSE P 504
Today’s outline

• Recap of coverage- and mutation-based testing
• Debugging basics
• Delta debugging technique
• Demo
• In-class activity: delta debugging

Background Reading:
Simplifying and Isolating Failure-Inducing Input, Zeller and Hildebandt, 2002
Mutation test prompting vs. mutation coverage

- Mutation testing
  - Primary output is new tests.

- Mutation analysis
  - Primary output is adequacy score for existing tests.

How expensive is mutation testing?
Is the mutation score meaningful?
Discussion: Coverage- and mutation-based testing

- Coverage-based testing
  - Any questions?
  - What are the pros and cons?
  - Will you test differently in the future?
- Mutation-based testing: open discussion
  - Any questions?
  - What have you observed and learned?
  - What are the pros and cons?
    - Example challenge: mutant comprehension vs. test creation
  - Do you feel it is worth using?
A Bug’s Life

Defect – mistake committed by a human
Error – incorrect computation
Failure – visible error: program violates its specification

Debugging starts when a failure is observed
  • Unit testing
  • Integration testing
  • In the field
Goal of debugging: go from failure back to defect
Ways to get your code right

• Design & verification
  • Prevent defects from appearing in the first place

• Defensive programming
  • Program with debugging in mind: fail fast

• Testing & validation
  • Uncover problems (even in spec), increase confidence

• Debugging
  • Find out why a program is not functioning as intended

• Testing ≠ debugging
  • test: reveals existence of problem (failure)
  • debug: pinpoint location + cause of problem (defect)
Defense in depth

1. Make errors **impossible**
   A (memory-)managed language prevents type errors, memory corruption

2. Don’t **introduce** defects
   Correctness: get things right the first time

3. Make errors immediately **visible**
   Example: assertions; `checkRep()`
   Converts an error to a failure; reduces distance from defect to failure

4. **Debugging** is the **last resort**
   Work from effect (failure) to cause (defect)
   **Scientific method:** Design experiments to gain information about the defect
   Easiest in a modular program with good specs and test suites
Debugging and the scientific method

Debugging must be **systematic**
- Carefully decide what to do (avoid fruitless avenues)
- **Record** everything that you do (**actions** and **results**)
  - Can replicate previous work
  - Or avoid the need to do so

Iterative scientific process:
- Formulate a **hypothesis**
- Design an **experiment**
- Perform the **experiment**
- Interpret **results**

Should have explanatory power
Not just “Maybe line 3 is wrong”

Should investigate cause
The typical debugging process

- **Identify** – it’s a bug, not a feature
- **Reproduce** – what are the inputs and conditions causing the error
- **Test** – create a (minimal) **regression test** to illustrate the issue
- **Localize** – locate the problematic code
- **Fix** the code
- **Validate** – run the regression test and the original failing scenario
A good bug report (issue)

A **bug report** should contain:
- How to **reproduce**, including context. (What is “context”?)
  - Preferably commands that can be cut-and-pasted into a shell
    - Starting with `git clone`
  - Don’t use vague English; provide an exact command
- All inputs
- Full output
  - No screenshots of textual output
- What you expected; why the behavior is wrong

A **test case** should be as simple as possible ("**minimized**")
- Why?
A test case should be as simple as possible

Why?
A test case should be as simple as possible

Why?

• Helps to localize the defect
  • fewer lines and features are exercised
• Speeds up the edit-compile-test cycle
• You need a small regression test
Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

— Richard Stallman, Using and Porting GNU CC
Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it. This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else.

— Richard Stallman, Using and Porting GNU CC
Binary search (e.g., git bisect)

Continuous integration runs:

v1  v2  v3  v4  v5  v6  v7  v8  v9  v10  v11  v12  v13  v14  v15

Add a new test case:

v1  v2  v3  v4  v5  v6  v7  v8  v9  v10  v11  v12  v13  v14  v15

Binary search is not guaranteed to reproduce the original failure. => You might not fix the defect that caused that failure.

v13 might have failed in a different way, for a different reason.

Still fails!

Did this use the scientific method?
Delta Debugging

A debugging technique to create a minimal test case that fails in the same way.

Input:
- Program
- Failing test case

Output:
- Failing test case that is as small as possible
This is a crashing test case

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Priority</th>
<th>Bug Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>P1</td>
<td>blocker</td>
</tr>
<tr>
<td>Windows 3.1</td>
<td>P2</td>
<td>critical</td>
</tr>
<tr>
<td>Windows 95</td>
<td>P3</td>
<td>major</td>
</tr>
<tr>
<td>Windows 98</td>
<td>P4</td>
<td>normal</td>
</tr>
<tr>
<td>Windows ME</td>
<td>P5</td>
<td>trivial</td>
</tr>
<tr>
<td>Windows 2000</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Windows NT</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 7</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 7.5</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 7.6.1</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 8</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 8.5</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 8.6</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 9.x</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>MacOS X</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Linux</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>FreeBSD</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>NetBSD</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>OpenBSD</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>AIX</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>BeOS</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>HP-UX</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>IRIX</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Neutrino</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>OpenVMS</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Solaris</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>SunOS</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>other</td>
<td></td>
<td>enhancement</td>
</tr>
</tbody>
</table>

- **Crashed Mozilla**
- **Consider 370 of these being filed!**
- **How would you debug the problem?**
This is a crashing test case

<table>
<thead>
<tr>
<th>Crashed Mozilla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consider 370 of these being filed!</td>
</tr>
<tr>
<td>How would you debug the problem?</td>
</tr>
<tr>
<td>What content is sufficient to reproduce the failure?</td>
</tr>
</tbody>
</table>
This is a crashing test case

- **Crashed Mozilla**
- **A minimal test case is:**

```html
<SELECT NAME="priority" MULTIPLE SIZE=7>

<SELECT NAME="bug severity" MULTIPLE SIZE=7>
  <OPTION VALUE="blocker">blocker</OPTION VALUE="critical">critical</OPTION VALUE="major">major</OPTION VALUE="normal">normal</OPTION VALUE="minor">minor</OPTION VALUE="trivial">trivial</SELECT>
```
This is a crashing test case

- **Crashed Mozilla**
- **A minimal test case is:**
  
  - Can we automate the process of minimizing test cases?
  - Idea: use binary search
Try the first half of the input

- Crashed Mozilla
- A minimal test case is: <SELECT>

- Can we automate the process of minimizing test cases?
- Idea: use binary search
- What is the result of the test?
Delta Debugging

A debugging technique to create a minimal test case that fails *in the same way*.

Input:
- Program
- Failing test case

Output:
- Failing test case that is as small as possible
A debugging technique to create a minimal test case that fails in the same way.

Input:
- Program
- Failing test case
- Predicate on executions: did the execution fail in the same way?

Output:
- Failing test case that is as small as possible

Test passes => false
Test fails in the same way => true
Test fails in some other way => false
Minimizing test cases

Goal: minimize the failing test case (remove some lines)
The happy path: binary search

Failing test with 16 lines
The minimal failing test has 2 lines: 3 and 4
The happy path: binary search
The happy path: binary search
The happy path: binary search
The happy path: binary search
The happy path: binary search

Successfully minimized the failing test to 2 lines
The not so happy path...

Failing test with 16 lines
The minimal failing test has 3 lines: 3, 4, and 9
The not so happy path...
The not so happy path...

Binary search is no help
Delta Debugging = binary search
+ handle subsets
+ account for multiple types of test outcomes

See paper:
Simplifying and Isolating Failure-Inducing Input
Zeller and Hildebrandt, 2002
The Delta Debugging algorithm

Three basic phases (in a loop):
1. Test each subset* (= binary subdivision)
2. Test each complement*
3. Increase granularity (double the # of subsets)
*On success, reduce the # of subsets (don’t continue testing)

Complement example:
Input = 1, 2, 3, 4
A subset is \{ 1 \}
Its complement is \{ 2, 3, 4 \}

Minimizing Delta Debugging Algorithm

Let \(\text{test}\) and \(c_x\) be given such that \(\text{test}(\emptyset) = \checkmark \land \text{test}(c_x) = \times\) hold.

The goal is to find \(c'_x = \text{ddmin}(c_x)\) such that \(c'_x \subseteq c_x, \text{test}(c'_x) = \times\), and \(c'_x\) is 1-minimal.

The minimizing Delta Debugging algorithm \(\text{ddmin}(c)\) is

\[
\text{ddmin}(c_x) = \text{ddmin}_2(c_x, 2)
\]

where

\[
\text{ddmin}_2(c'_x, n) =
\begin{cases}
\text{ddmin}_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \land \text{test}(\Delta_i) = \times \text{ (“reduce to subset”)}

\text{ddmin}_2(\nabla_i, \max(n - 1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \land \text{test}(\nabla_i) = \times \text{ (“reduce to complement”)}

\text{ddmin}_2(c'_x, \min(|c'_x|, 2n)) & \text{else if } n < |c'_x| \text{ (“increase granularity”)}

c'_x & \text{otherwise ("done")}
\end{cases}
\]

where \(\nabla_i = c'_x - \Delta_i, c'_x = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n, \) all \(\Delta_i\) are pairwise disjoint, and \(\forall \Delta_i \cdot |\Delta_i| \approx |c'_x|/n\) holds.

The recursion invariant (and thus precondition) for \(\text{ddmin}_2\) is \(\text{test}(c'_x) = \times \land n \leq |c'_x|\).
Delta Debugging is mostly binary search
Delta Debugging: test subsets

\[ \Delta_1 \]

\[ \Delta_2 \]

Notation for subset
Delta Debugging: test complements

Notation for complement of subset 1
Delta Debugging: increase granularity

\[ \Delta_1 \quad \Delta_2 \quad \Delta_3 \quad \Delta_4 \]

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

P P P P

F F
Delta Debugging: complements

Complement of subset 1
Delta Debugging: complements

\[ \nabla_4 \]

\[ \Delta_4 \]

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

F F

* \( \nabla_2 \) also fails tests
Delta Debugging: reduce
Delta Debugging: test subsets

\[ \Delta_1 \quad \Delta_2 \quad \Delta_3 \]

1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16

P  P  P  P

F  F
Delta Debugging: complements
Delta Debugging: complements
Delta Debugging: reduce

\[ \Delta_1 \]

\[ \Delta_2 \]

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

P P P P

F F
Delta Debugging: increase granularity

\[ \Delta_1 \quad \Delta_2 \quad \Delta_3 \quad \Delta_4 \]

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

P P P P

F F
Delta Debugging: complements
Delta Debugging: reduce

And so on...
Failing test cases must be deterministic and monotone
Delta debugging: live example
Delta Debugging: live example

Program and initial test case
- Program P crashes whenever the input contains 1 7 8
- Initial crashing test case is: 1 2 3 4 5 6 7 8

Syntax:
- % ./delta -test=./test.sh -cp_minimal=./min.txt < failing.txt
- test.sh returns 0 if input causes failure, 1 if input passes

Delta debugging approach:
- Test each subset*
- Test each complement*
- Increase granularity
  Reduce on success
Program and initial test case

- Program $P$ takes as input a String of $a$s and $b$s.
- $P$ crashes whenever the input String contains an even number of $a$s and an odd number of $b$s.
- Assume character-level granularity.
- Example crashing test inputs: babab, aaaaabbb.

Determine the following test cases

1. Smallest
2. 1-minimal but not smallest

Give an input such that DD outputs each of these.
Quiz

Program and initial test case
- Program $P$ takes as input a String of $a$s and $b$s.
- $P$ crashes whenever the input String contains an even number of $a$s and an odd number of $b$s.
- Assume character-level granularity.
- Example crashing test inputs: $\text{babab, aaaaabbb}$. 

Determine the following test cases
1. Smallest $b$
2. 1-minimal but not smallest $aab$

Give an input such that DD outputs each of these.
Let’s try one more

Program and initial test case

- Program $P$ takes as input a list of integers $lst$.
- $P$ crashes whenever $lst$ contains 4,2.
- Initial crashing test input is: 2,4,2,4

Complete the following table (add a new row whenever the number of subsets changes)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Number of subsets</th>
<th>input</th>
<th>$\Delta_1, \ldots, \Delta_n$</th>
<th>$\nabla_1, \ldots, \nabla_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2424</td>
<td>$\Delta_1$</td>
<td>$\nabla_1$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>$\Delta_2$</td>
<td>$\nabla_2$</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>$\Delta_3$</td>
<td>$\nabla_3$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\Delta_4$</td>
<td>$\nabla_4$</td>
</tr>
</tbody>
</table>
Let’s try one more

Program and initial test case
- Program $P$ takes as input a list of integers $lst$.
- $P$ crashes whenever $lst$ contains 4,2.
- Initial crashing test input is: 2,4,2,4

Complete the following table (add a new row whenever the number of subsets changes)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>n</th>
<th>input</th>
<th>$\Delta_1$, ..., $\Delta_n$</th>
<th>$\nabla_1$, ..., $\nabla_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2424</td>
<td>24, (24)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2424</td>
<td>2, 4, (2), (4), 424, 224, 244, 242</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>424</td>
<td>(4), (2), (4), (24), 44, 42</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>42</td>
<td>(4), (2)</td>
<td></td>
</tr>
</tbody>
</table>

- Parenthesized values are cached.
- Boldfaced values are success.
- Struck out values were not tested.
- An implementation chooses the order of subsets/complements.
Delta debugging: summary

Discussion

- Non-deterministic programs
- Input structure and granularity
- Monotonicity
- Complexity

- What other structures can you delta debug over?
Delta debugging: in-class exercise
Examples from demo

Oracle test (look for (1 and 2) or 8, 0=test fails, 1=test passes)

```
test.sh
#!/bin/sh
cat $1 | tr -d '
' | grep -q -E "12|8"
```

```
review-files.sh
#!/bin/sh
for file in tmp0/*.c ; do cat $file | tr -d '
' ; echo ; done
```
Sample oracle test if you don’t have one yet

test.sh

#!/bin/sh
#
#   --sh--
if timeout 0.5s perl ..../mysort.pl $1 >/dev/null; then exit 1; fi