Debugging for Scripting Languages
Literature

  - Highly recommended. Today’s class is based on 1st ed.
  - Covers software vise and dependency breaking.
  - Not how to debug, but how to write a debugger.
Why Debugging in this Class?

Scripts are easier to debug
- Less code.
- Higher-level code.
- Read-eval-print loop.
- Easier to change.

Scripts are harder to debug
- No static type checks.
- More “hacks”, code is less readable.
- Web applications are hard to test.

Scripts help debug other applications
- Scripting-as-glue makes it easy to run programs and check outputs.
- Scripting as application extension can automate GUI tests.
Outline

• Systematic Debugging
• Debugging Tools
• Testing for Debugging
Example Bug

1 Sub AverageRows(Result(), Rows())
2 For I = 0 To UBound(Rows, 1)
3 For J = 0 To UBound(Rows, 2)
4 Result(I) = Result(I) + Rows(I, J)
5 Next J
6 Result(I) = Result(I) / (1 + UBound(Rows, 2))
7 Next I
8 End Sub
9 Sub Main()
10 Dim x(2, 3)
11 x(0, 0) = 1: x(0, 1) = 2: x(0, 2) = 0
12 x(1, 0) = 0: x(1, 1) = 0: x(1, 2) = 2
13 Dim y(2)
14 Call AverageRows(y, x)
15 Debug.Print y(0) & ", " & y(1)
16 End Sub

<table>
<thead>
<tr>
<th></th>
<th>avg(1,2,0)</th>
<th>avg(0,0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>1</td>
<td>0.6667</td>
</tr>
<tr>
<td>Observed</td>
<td>0.75</td>
<td>0.5</td>
</tr>
<tr>
<td>Concepts</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Log Book**

(Example Debugging Session)

<table>
<thead>
<tr>
<th>Round 1</th>
<th>Hypothesis</th>
<th>Experiment</th>
<th>Observation</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Result(i) wrong before Line 6</td>
<td>Breakpoint at 6, inspect Result(i)</td>
<td>Result(i)==3</td>
<td>Hypothesis is wrong, correct numerator</td>
</tr>
</tbody>
</table>

| Round 2 | Ubound(Rows,2) wrong before Line 6 | Breakpoint at 6, inspect Rows(0) bounds | Rows(0) indices go from 0 to 3 | Ubound(Rows,2) is 3: array too large, should go from 0 to 2 |

| Round 3 | Ubound(x,2)==3 | Breakpoint at 11, inspect x(0) bounds | x(0) indices go from 0 to 3 | Array x (0) is too large, should go from 0 to 2 |

| Round 4 | Upper bounds in Dim are wrong | Dim x(1,2) and Dim y(1) | Output 1 and 0.6667 | Bug is fixed |
Concepts

Space+Time Search

<table>
<thead>
<tr>
<th>Line</th>
<th>$x(0)$</th>
<th>$x(1)$</th>
<th>$x(2)$</th>
<th>y</th>
<th>Result</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>E:E:E:E</td>
<td>E:E:E:E</td>
<td>E:E:E:E</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>1:2:0:E</td>
<td>0:0:2:E</td>
<td>E:E:E:E</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>1:2:0:E</td>
<td>0:0:2:E</td>
<td>E:E:E:E</td>
<td>E</td>
<td>:E</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>1:2:0:E</td>
<td>0:0:2:E</td>
<td>E:E:E:E</td>
<td>E</td>
<td>:E</td>
<td>alias(y) alias(x)</td>
</tr>
<tr>
<td>6</td>
<td>1:2:0:E</td>
<td>0:0:2:E</td>
<td>E:E:E:E</td>
<td>3</td>
<td>:E</td>
<td>alias(y) alias(x)</td>
</tr>
<tr>
<td>7</td>
<td>1:2:0:E</td>
<td>0:0:2:E</td>
<td>E:E:E:E</td>
<td>0.75:E</td>
<td>:E</td>
<td>alias(y) alias(x)</td>
</tr>
<tr>
<td>15</td>
<td>1:2:0:E</td>
<td>0:0:2:E</td>
<td>E:E:E:E</td>
<td>0.75:0.5:E</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Each line (time step) is program state (memory space)
- This diagram shows only a few selected states (time = many more steps, even for our simple program)
- Most programs have larger state (space = thousands of variables)
- Debugging is a search in time and space
Defects, Infections, Failures

- Sane State
- Infection = wrong state
- Defect = wrong code that turns sane state into infection
- Failure = wrong output observed by user

Zeller avoids the word “bug”, since it could mean any of the above.

Infection spreads.
The Scientific Method

Concepts

Hypothesis
E.g., from looking at code

Experiment
E.g., run with certain input

Observation
E.g., using print statement

Conclusion
E.g., step of infection chain

(Backup first)

(Fix similar problems)

Repeat
## Concepts

### Reasoning Techniques

<table>
<thead>
<tr>
<th>Concept</th>
<th>General → Specific</th>
<th>General → Specific</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deduction</td>
<td>0 runs (look at code)</td>
<td>Finding hypotheses by “eye-balling” the code</td>
<td></td>
</tr>
<tr>
<td>Observation</td>
<td>1 run (and sensors)</td>
<td>Finding needle (infection) in hay stack (space+time)</td>
<td></td>
</tr>
<tr>
<td>Induction</td>
<td>Many similar runs</td>
<td>Finding hypotheses by brute force</td>
<td></td>
</tr>
<tr>
<td>Experiment</td>
<td>≥1 systematic runs</td>
<td>Confirming or rejecting hypotheses</td>
<td></td>
</tr>
</tbody>
</table>
## Search Space Reduction

<table>
<thead>
<tr>
<th>B</th>
<th>acker</th>
<th>ack</th>
<th>w</th>
<th>ar</th>
<th>d</th>
<th>Slice</th>
<th>A</th>
<th>ss</th>
<th>er</th>
<th>A</th>
<th>tion</th>
<th>n</th>
<th>he</th>
<th>ck</th>
</tr>
</thead>
</table>

Separate relevant from irrelevant

Separate sane from infected
Outline

• Systematic Debugging
• Debugging Tools
• Testing for Debugging
Static Checking

• What
  – Automatic deduction of common defects
  – Do this habitually before you need to debug

• How
  – VBA:  continuous compilation; option explicit
  – Perl:  use strict; use warnings; perl -w
  – PHP:  php -l
  – JS:  http://www.jslint.com
Print Statements

• What
  – Observation to check hypothesis in experiment
  – Useful to automate printing source location

• How
  – VBA: `Debug.print expr`
  – Perl: `print __FILE__,'
',__LINE__,'
';`
  – PHP: `echo __FILE__,'
',__LINE__,'
';`
    `var_dump(expr);`
  – JS: `try{throw Error();}catch(e){alert(e.stack);}`
    (Mozilla Firefox only)
Assertions

• What
  – Reduces search space by categorically ruling out some infections in some of the data
  – May be disabled for production run

• How
  – VBA: `If !cond Then Error 1`
  – Perl: `doSomething or die $!;`
  – PHP: `assert(cond);`
  – JS: `if(!cond) alert("message");`
Dynamic Checking

• What
  – Turn silent infection into user-visible fault
  – System assertion as opposed to user assertion

• How
  – C: Valgrind asserts absence of common memory errors, e.g., using value before first assignment
  – Perl: Taint mode (`perl -T`) asserts that inputs are sanitized, e.g., to avoid SQL injection
REPL: Read-Eval-Print Loops

• What
  – Fast turn-around for ad-hoc experiments
  – Call one function at a time, without harness

• How
  – VBA: Visual Basic Editor→Tools→Immediate Window
  – Perl: `perl -wde1`
  – PHP: `php -a`, if it is compiled that way
  – JS: Firefox→Tools→Web Developer→Error Console
Interactive Debuggers

• What
  – Experiments such as “break at 9, look at x”
  – REPL + break points + stack inspection

• How
  – VBA: integrated with editor
  – Perl: `perl -wd file.pl`
  – PHP: [http://www.php.net/debugger](http://www.php.net/debugger)
  – JS: Firefox Venkman add-on; debug closure trick; Firebug
<html><head><script>
function breakpoint(evalFunc, msg) {
    var expr = "arguments.callee";
    var result;
    while (true) {
        var line = "\n----------------------\n";
        expr = prompt("BREAKPOINT: " + msg + "\n" + (result ? "eval('" + expr + ") -> " + line + result + line : "\n") + "Enter an expression:"; expr);
        if (expr == null || expr == "") return;
        try {
            result = evalFunc(expr);
        } catch (e) {
            result = e;
        }
    }
}

function foo(x, y) {
    breakpoint(function(expr){return eval(expr);}, "bar");
}
foo(2, 4);
</script></head><body><script>
</script></body></html>
sub sort_ref_to_array {  # buggy!
  my @sorted = sort @_;
  return $sorted[0];
}

sub test_sort {
  my $arrayref = sort_ref_to_array($_[0]);
  for (my $i=0; $i+1 < @$arrayref; $i++) {
    if ($arrayref->[$i] > $arrayref->[$i+1]) {
      return 'fail';  # ✗
    }
  }
  return 'pass';  # ✓
}

our $min = ddmin([1,3,5,2,4,6], \&test_sort);
print "minimized to ", @$min, "\n";

sub dadmin { 
    my ($inputs, $test) = @_; 
    $test->[[]] eq 'pass' && $test->($inputs) eq 'fail' or die; 
    my $splits = 2; 
    outer: while (2 <= @$inputs) { 
        for my $subset (subsets($inputs, $splits)) { 
            my $complement = list_minus($inputs, $subset); 
            if ('fail' eq $test->($complement)) { 
                $inputs = $complement; 
                $splits-- if $splits > 2; 
                next outer; 
            } 
        } 
    } 
    last outer if $splits == @$inputs; 
    $splits = 2 * $splits < @$inputs ? 2 * $splits : @$inputs; 
    return $inputs; 
}
Perl

Delta Debugging Helper Functions

```perl
sub subsets {
    my ($fullset, $splits) = @_;  
    my @result;
    my $bin_size = int((@$fullset + $splits - 1) / $splits);
    for (my $i=0; $i<$splits; $i++) {
        my ($start, $end) = ($i * $bin_size, ($i + 1) * $bin_size);
        if ($end > @$fullset) { $end = @$fullset; }
        my @subset;
        for (my $j=$start; $j<$end; $j++) { push @subset, $fullset->[0]; }
        push @result, [ @subset ];
    }
    return @result;
}

sub list_minus {
    my ($fullset, $subtract) = @_;  
    my (%subtract, @result);
    for (@$subtract) { $subtract{$_} = 1; }
    for (@$fullset) { push(@result, $_) unless $subtract{$_}; }
    return [ @result ];
}
```

© Martin Hirzel
Outline

• Systematic Debugging
• Debugging Tools
• Testing for Debugging
## Concepts

<table>
<thead>
<tr>
<th>Track</th>
<th>Enter in bug database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reproduce</td>
<td>Get all the inputs</td>
</tr>
<tr>
<td>Automate</td>
<td>Create test harness</td>
</tr>
<tr>
<td>Find origin</td>
<td>Use scientific method to trace back infection chain</td>
</tr>
<tr>
<td>Focus</td>
<td></td>
</tr>
<tr>
<td>Isolate</td>
<td></td>
</tr>
<tr>
<td>Correct</td>
<td>Remove defect</td>
</tr>
</tbody>
</table>
Bug Tracking
Life Cycle of a Problem in Bugzilla

Concepts

Track
Reproduce
Automate
Find origin
Focus
Isolate
Correct
## Bug Jargon

<table>
<thead>
<tr>
<th>Bug Jargon</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bohr bug</td>
<td>Quantum physics</td>
</tr>
<tr>
<td></td>
<td>Repeatable, manifests reliably</td>
</tr>
<tr>
<td>Heisenbug</td>
<td>Uncertainty principle</td>
</tr>
<tr>
<td></td>
<td>Disappears due to observation probe (e.g., time dependent)</td>
</tr>
<tr>
<td>Mandelbug</td>
<td>Mandelbrot set</td>
</tr>
<tr>
<td></td>
<td>Causes are complex, appears nondeterministic (but is Bohr bug)</td>
</tr>
<tr>
<td>Schroedinbug</td>
<td>Schrödinger’s cat, thought experiment</td>
</tr>
<tr>
<td></td>
<td>Hidden until first person notices it, then becomes show-stopper</td>
</tr>
</tbody>
</table>
Sources of Input

Difficult to reproduce problem if
- User’s input ≠ developer’s input
- Input is large and/or time sensitive
Concepts

Software Vise

- Vise = holds an artifact firm for working on it
- Perl makes it easy to build vise for batch application
- VBA allows you to build vise for GUI application
- How to build vise for just one unit of a program?
System Tests vs. Unit Tests

- System test = test entire application
- Unit test = test part of system in isolation
- Why use unit tests in debugging?
  - Focus: less code = smaller hay stack
  - Speed: faster to run experiment
  - Prevent side effects: e.g., to database
  - Verify fix: make sure the defect is gone
Dependency Breaking

- To test a unit, must break its dependency on the rest of the system
Seams

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Seams</th>
</tr>
</thead>
</table>
| **Goal** | Replace dependency by mock object  
Do not modify code of unit under test |
| **Solution** | Use virtual method dispatch as “seam” |
| **Challenge** | Dependency may not be on method call  
Refactor to object-oriented style first |
Minimal Tests

- Using delta debugging, either automatically or by hand
- The test to keep is the minimal end result
- If you submit a bug report to a project, it will get fixed faster if you minimize it first
- Gecco BugAThon
Regression Testing

- Regression
  - Shift towards less perfect state
  - In software: when old bugs appear again
- Regression testing
  - Check that fixed bugs are still fixed
- Recommended practice
  - Keep the tests you use during debugging
  - Run them frequently (at least daily)
  - To run many tests often, each individual test must be fast $\Rightarrow$ use unit tests
• Today’s lecture
  – Scientific method
  – Tools for scripting language debugging
  – TRAFFIC