G22.3033-002 NYU 7/17/2008 Debugging Scripts

**Literature**
- From Automated Testing to Automated Debugging. Andreas Zeller, 2000. Available at http://www.infouni-passau.de/atpapers/computer2000/ This was the recommended reading for today’s class.

**Outline**
- Systematic Debugging
- Debugging Tools
- Testing for Debugging
- At 7:30pm: Quiz 3

**Soap-Box**
**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.

**Reference**

**VBA**

```vba
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
```

```plaintext
<table>
<thead>
<tr>
<th>Expected</th>
<th>Observed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.75</td>
</tr>
<tr>
<td>2.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>
```

**Log Book**

(Example Debugging Session)

<table>
<thead>
<tr>
<th>Round</th>
<th>Hypothesis</th>
<th>Experiment</th>
<th>Observation</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Result(I) wrong before Line 6</td>
<td>Breakpoint at 6</td>
<td>Result(I) == 2</td>
<td>Hypothesis is wrong, correct numerator</td>
</tr>
<tr>
<td>2</td>
<td>UBound(Rows, 2) wrong before Line 6</td>
<td>Breakpoint at 6</td>
<td>Rows(0) indices go from 0 to 3</td>
<td>UBound(Rows, 2) is 3; array too large, should go from 0 to 2</td>
</tr>
<tr>
<td>3</td>
<td>UBound(x, 2) == 3</td>
<td>Breakpoint at 11, inspect x(0) bounds</td>
<td>x(0) indices go from 0 to 3</td>
<td>Array x(0) is too large, should go from 0 to 2</td>
</tr>
<tr>
<td>4</td>
<td>Upper bounds in Dim are wrong</td>
<td>Dim x(1, 2) and Dim y(1)</td>
<td>Output 1</td>
<td>Bug is fixed</td>
</tr>
</tbody>
</table>

**Concepts**

- Hypothesis
- Experiment
- Observation
- Conclusion
Space-Time Search

- Each line (time step) is a program state (memory space)
- This diagram shows only a few selected states
- Most programs have larger state (space = thousands of variables)
- Debugging is a search in time and space

<table>
<thead>
<tr>
<th>Line</th>
<th>x(0)</th>
<th>x(1)</th>
<th>x(2)</th>
<th>Result</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Defects, Infections, Failures

- Defect = wrong code that turns sane state into infection
- Infection = wrong state
- Failure = wrong output observed by user
- Zeller avoids the word “bug”, since it could mean any of the above.

The Scientific Method

- 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

Reasoning Techniques

- Deduction: General → Specific
  - 0 runs (look at code)
  - Finding hypotheses by "eye-balling" the code.
- Observation: Specific → General
  - 1 run (and sensors)
  - Finding needle (infection) in haystack (space+time).
- Induction: Specific → General
  - Many similar runs
  - Finding hypotheses by brute force
- Experiment: >1 systematic runs
  - Confirming or rejecting hypotheses

Search Space Reduction

- Separate relevant from irrelevant
- Separate sane from infected

Outline

- Systematic Debugging
- Debugging Tools
- Testing for Debugging
- At 7:30pm: Quiz 3
Static Checking

- What
  - Automatic deduction of common defects
  - Do this at start of debugging session
- 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:
  - JS: try{throw Error();}catch(e){alert(e.stack);}(Mozilla Firefox only)

Concepts

- 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

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
  - JS: Firefox Venkman add-on; debug closure trick
### JavaScript

**Debug Closure Trick**


```javascript
function breakpoint(evalFunc, msg) {
  var expr = "arguments.callee"
  var result;
  while (true) {
    var line = "----------------------

    expr = prompt("BREAKPOINT: " + msg + "

    if (expr == null || expr == "") return;
    try {
      result = eval(evalFunc(expr));
    } catch (e) {
      result = e;
    }
    
    expr = "Enter an expression:

    if (expr == null || expr == "") return;
  }
}
```

```html
<body>
<script>
  function foo(x, y) {
    breakpoint(function(expr){return eval(expr);}, "bar");
  }
  foo(2, 4);
</script>
</body>
```

By Steve Yen:

[http://trimpath.com/project/wiki/TrimBreakpoint](http://trimpath.com/project/wiki/TrimBreakpoint)

### Delta Debugging Example

```perl
sub sort_ref_to_array {  # buggy!
  my @sorted = sort @_;  # return $sorted[0];
  return $sorted[0];
}
sub test_sort {
  my $arrayref = sort_ref_to_array($_[0]);
  for (my $i=0; $i < @$arrayref; $i++) {
    if ($arrayref->[$i] > $arrayref->[$i+1]) {
      return 'fail'; #
    } else {
      return 'pass'; #
    }
  }
  our $min = ddmin([1,3,5,2,4,6], &test_sort);
  print "minimized to ", @$min,
```

By Andreas Zeller:

[http://www.whyprogramsfail.com/resources.php](http://www.whyprogramsfail.com/resources.php)

### Perl

#### Delta Debugging Algorithm

By Andreas Zeller: [http://www.whyprogramsfail.com/resources.php](http://www.whyprogramsfail.com/resources.php)

```perl
sub ddmin {
  my ($inputs, $test) = @_;  # buggy!
  $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;
        last outer;
      }
    }
    last outer if $splits == @$inputs;
    $splits = 2 * $splits < @$inputs ? 2 * $splits : @$inputs;
  }
  return $inputs;
}
```

By Andreas Zeller: [http://www.whyprogramsfail.com/resources.php](http://www.whyprogramsfail.com/resources.php)

#### Delta Debugging Helper Functions

```perl
sub subsets {
  my ($fullset, $splits) = @_;  # buggy!
  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, $fullset->[$j]; }
    push @result, [ @subset ];
  }
  return @result;
}
```

### Concepts

#### Delta Debugging Algorithm

By Andreas Zeller: [http://www.whyprogramsfail.com/resources.php](http://www.whyprogramsfail.com/resources.php)

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

### Outline

- Systematic Debugging
- Debugging Tools
- Testing for Debugging
- At 7:30pm: Quiz 3
Bug Tracking
Life Cycle of a Problem in Bugzilla

Bug Jargon

Sources of Input

Software Vise

System Tests vs. Unit Tests

Dependency Breaking
Seams

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

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 (see reading for today)

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 → use unit tests

Last Slide

- hw08 was assigned late → due 7/25
- hw09 is smaller (15 points) → also due 7/25
- Today’s lecture
  - Scientific method
  - Tools for scripting language debugging
  - TRAFFIC
- Next lecture
  - Bash
  - Python
  - Ruby
  - …

Outline

- Systematic Debugging
- Debugging Tools
- Testing for Debugging
- At 7:30pm: Quiz 3