# CS 5142
# Scripting Languages

11/25/2013

Debugging for
Scripting Languages

**Reference**

# Literature

- Why Programs Fail: A Guide to Systematic Debugging. Andreas Zeller. 2nd edition, Morgan Kaufmann, 2009.
  - Highly recommended. Today's class is based on 1st ed.
- From Automated Testing to Automated Debugging. Andreas Zeller, 2000. Available at http://www.infosun.fim.uni-passau.de/st/papers/computer2000/
- Working Effectively with Legacy Code. Michael Feathers. Prentice Hall, 2005.
  - Covers software vise and dependency breaking.
- How Debuggers Work. Jonathan B. Rosenberg. Wiley, 1996.
  - Not how to debug, but how to write a debugger.

**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.

# Outline

- Systematic Debugging
- Debugging Tools
- Testing for Debugging

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
```

| | avg(1,2,0) | avg(0,0,2) |
|---|---|---|
| Expected | 1 | 0.6667 |
| Observed | 0.75 | 0.5 |

Concepts

# Log Book (Example Debugging Session)

| | Hypothesis | Experiment | Observation | Conclusion |
|---|---|---|---|---|
| Round 1 | Result(i) wrong before Line 6 | Breakpoint at 6, inspect Result(i) | Result(i)==3 | Hypothesis is wrong , correct numerator |
| 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**

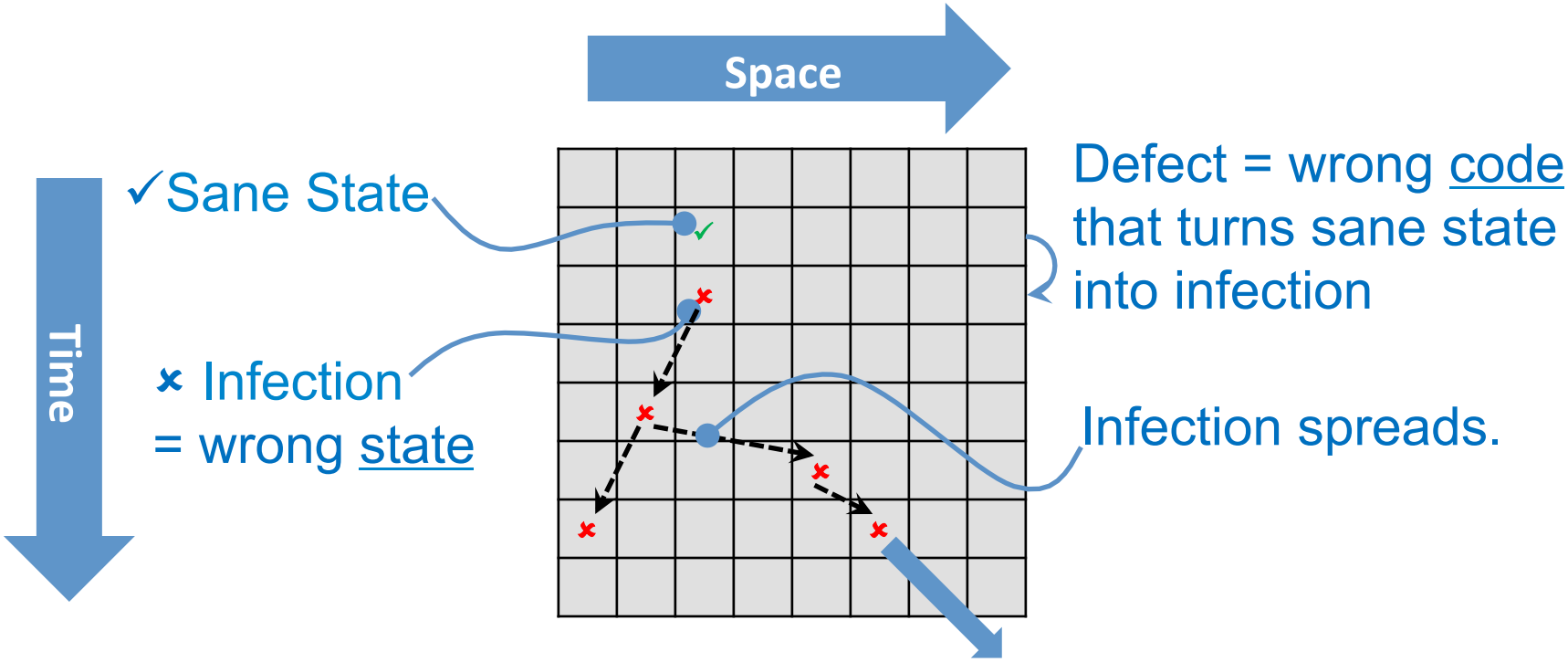
# Space+Time Search


| Line | x(0) | x(1) | x(2) | y | Result | Rows |
|---|---|---|---|---|---|---|
| 10 | - | - | - | - | - | - |
| 11 | E:E:E:E | E:E:E:E | E:E:E:E | - | - | - |
| 13 | 1:2:0:E | 0:0:2:E | E:E:E:E | - | - | - |
| 14 | 1:2:0:E | 0:0:2:E | E:E:E:E | E :E :E | - | - |
| 2 | 1:2:0:E | 0:0:2:E | E:E:E:E | E :E :E | alias(y) | alias(x) |
| 6 | 1:2:0:E | 0:0:2:E | E:E:E:E | 3 :E :E | alias(y) | alias(x) |
| 7 | 1:2:0:E | 0:0:2:E | E:E:E:E | 0.75:E :E | alias(y) | alias(x) |
| 15 | 1:2:0:E | 0:0:2:E | E:E:E:E | 0.75:0.5:E | - | - |


- 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

Concepts

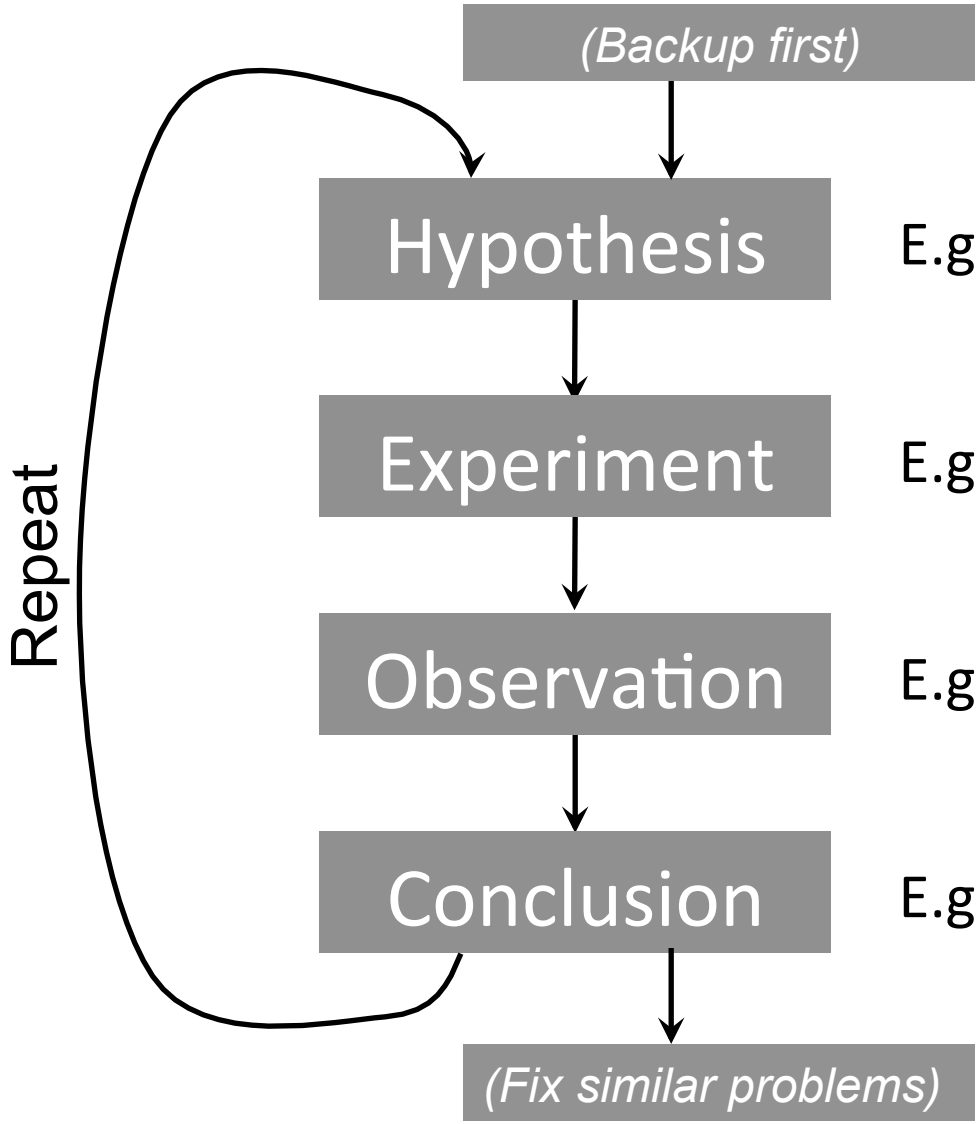
# Defects, Infections, Failures




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

Concepts

# The Scientific Method




E.g., from looking at code

E.g., run with certain input

E.g., using print statement

E.g., step of infection chain

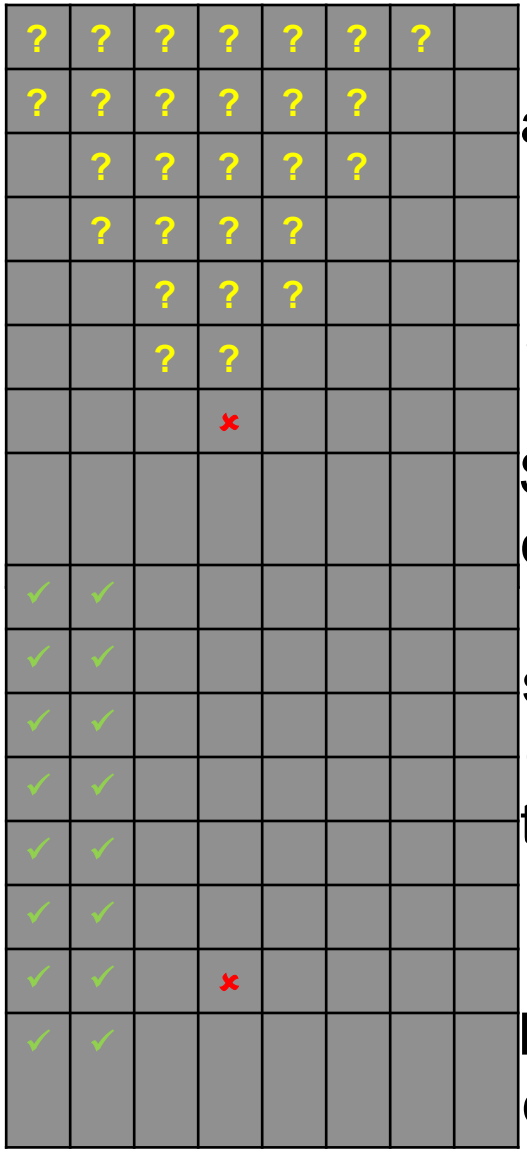
Concepts

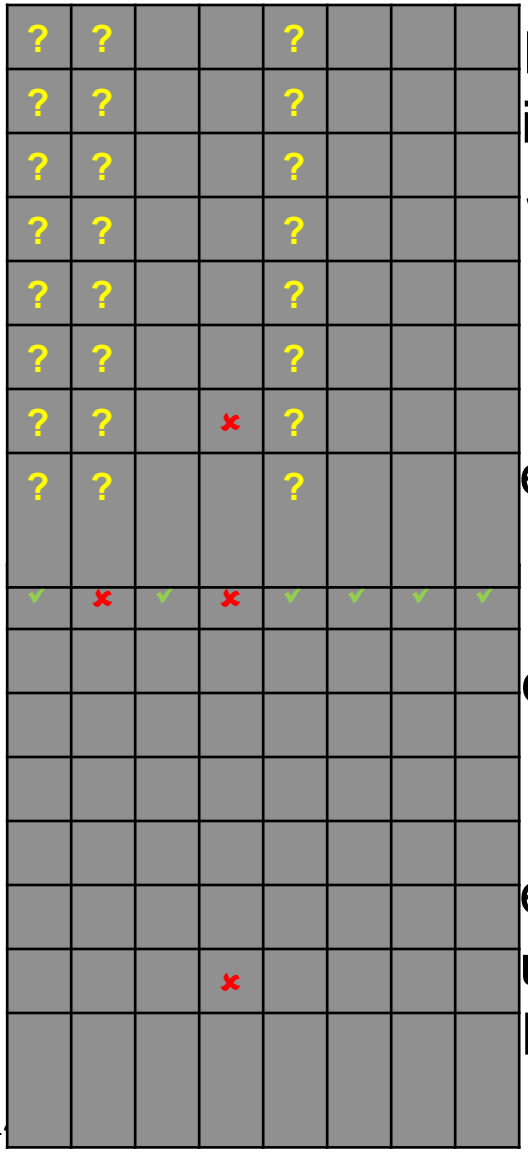
# Reasoning Techniques

| | | | |
|---|---|---|---|
| Deduction | General → Specific | 0 runs (look at code) | Finding hypotheses by "eye-balling" the code |
| Observation | | 1 run (and sensors) | Finding needle (infection) in hay stack (space+time) |
| Induction | Specific → General | Many similar runs | Finding hypotheses by brute force |
| Experiment | | ≥1 syste-matic runs | Confirming or rejecting hypotheses |

Concepts

# Search Space Reduction




Separate relevant from irrelevant

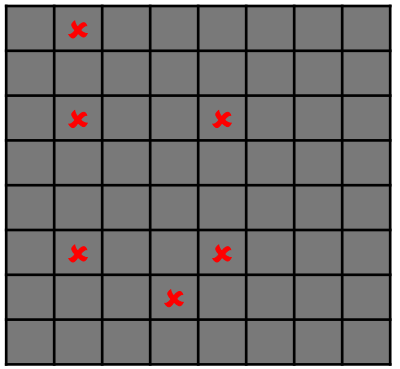
Separate sane from infected

# Outline

- Systematic Debugging
- Debugging Tools
- Testing for Debugging

Concepts

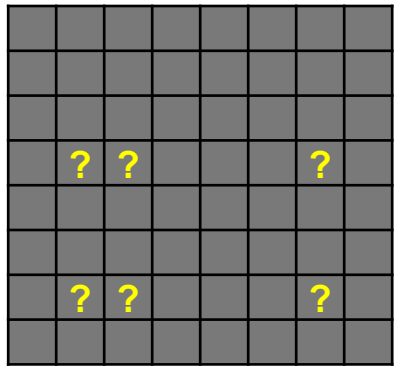
# Static Checking



- 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

Concepts

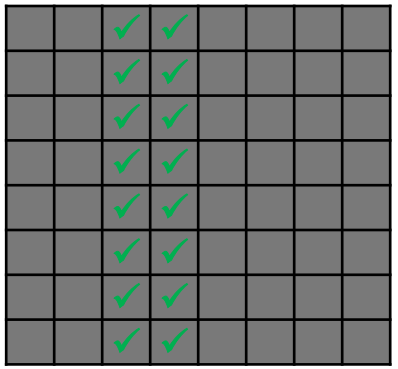
# Print Statements




- What
  - Observation to check hypothesis in experiment
  - Useful to automate printing source location
- How
  - VBA: **`Debug.print`** *expr*
  - Perl: **`print __FILE__,':',__LINE__,":\n";`**
  - PHP: **`echo __FILE__.':'.__LINE__.':';`**
    **`var_dump(`***expr***`);`**
  - JS: **`try{throw Error();}catch(e){alert(e.stack);}`**
    (Mozilla Firefox only)

Concepts

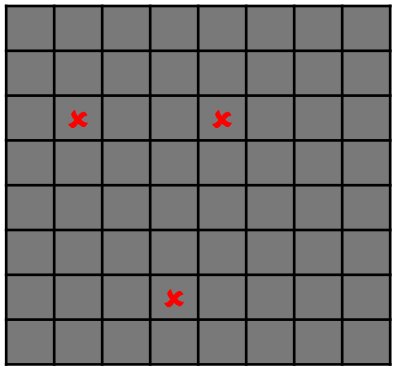
# Assertions



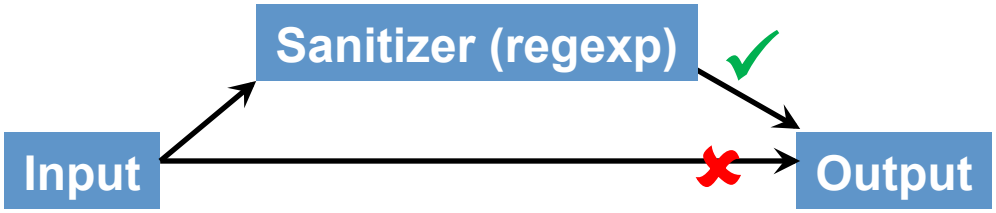
- What
  - Reduces search space by categorically ruling out some infections in some of the data
  - May be disabled for production run
- How
  - VBA: **`If !`***cond* **`Then Error 1`**
  - Perl: *doSomething* **`or die $!;`**
  - PHP: **`assert(`***cond***`);`**
  - JS: **`if(!`***cond***`) alert("`***message***`");`**

Perl

# 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



Concepts

# 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

Concepts

# 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; Firebug

JavaScript

# Debug Closure Trick

By Steve Yen: https://code.google.com/p/trimpath/wiki/TrimBreakpoint

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

Perl

# Delta Debugging Example

```
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";
```

```
135246  ✘
  246   ✓
135     ✓
  5246  ✘
    46  ✓
  52    ✘
   2    ✓
  5     ✓
  52
```

Concepts

# Delta Debugging Algorithm

By Andreas Zeller: http://www.whyprogramsfail.com/resources.php

```
sub ddmin {
  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;
}
```

# Delta Debugging Helper Functions

```
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, $fullset->[$j]; }
    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 ];
}
```

# Outline

- Systematic Debugging
- Debugging Tools
- Testing for Debugging

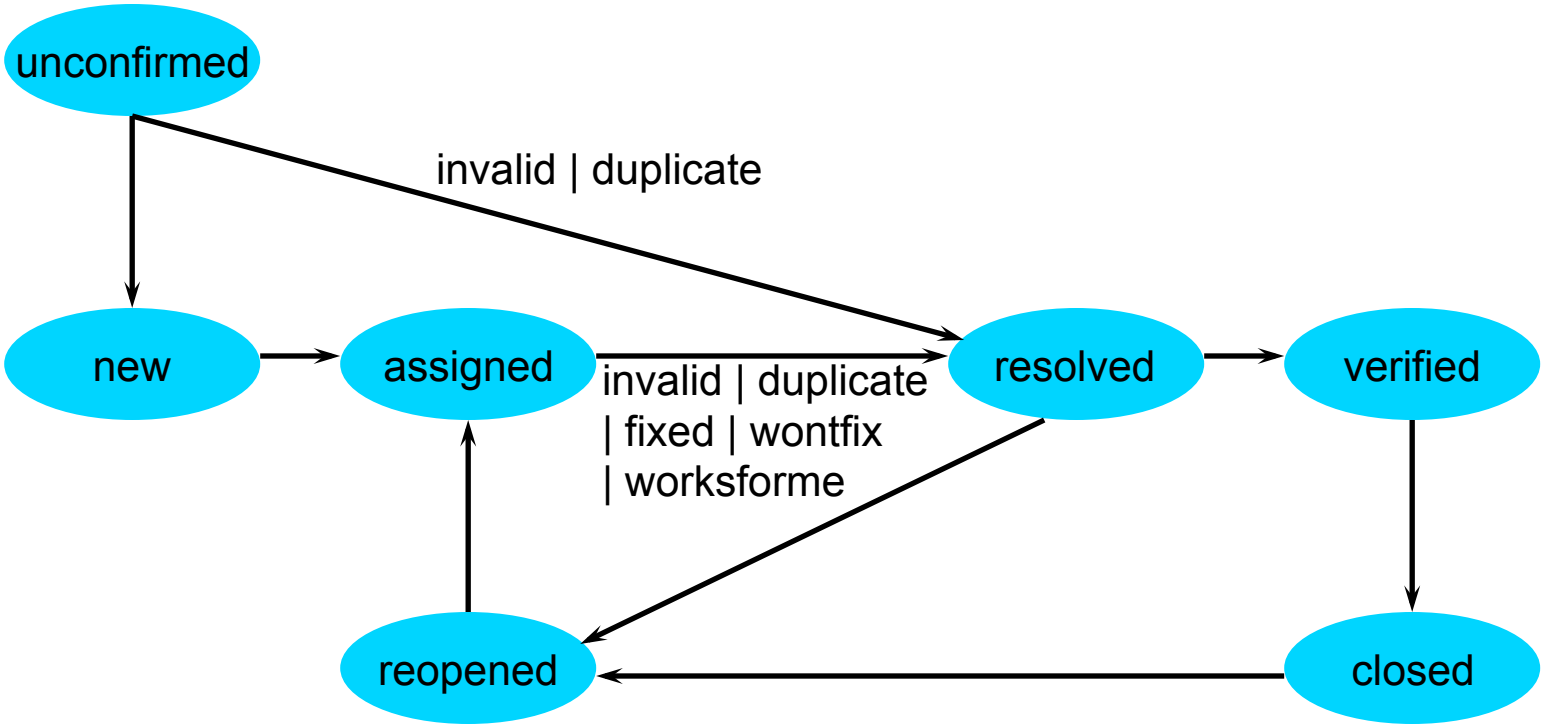
# TRAFFIC

| | |
|---|---|
| T rack | Enter in bug database |
| R eproduce | Get all the inputs |
| A utomate | Create test harness |
| F ind origin | Use scientific method to trace back infection chain |
| F ocus | |
| I solate | |
| C orrect | Remove defect |

**Concepts**

# Bug Tracking
## Life Cycle of a Problem in Bugzilla

T rack
R eproduce
A utomate
F ind origin
F ocus
I solate
C orrect



T rack
R eproduce
A utomate
F ind origin
F ocus
I solate
C orrect

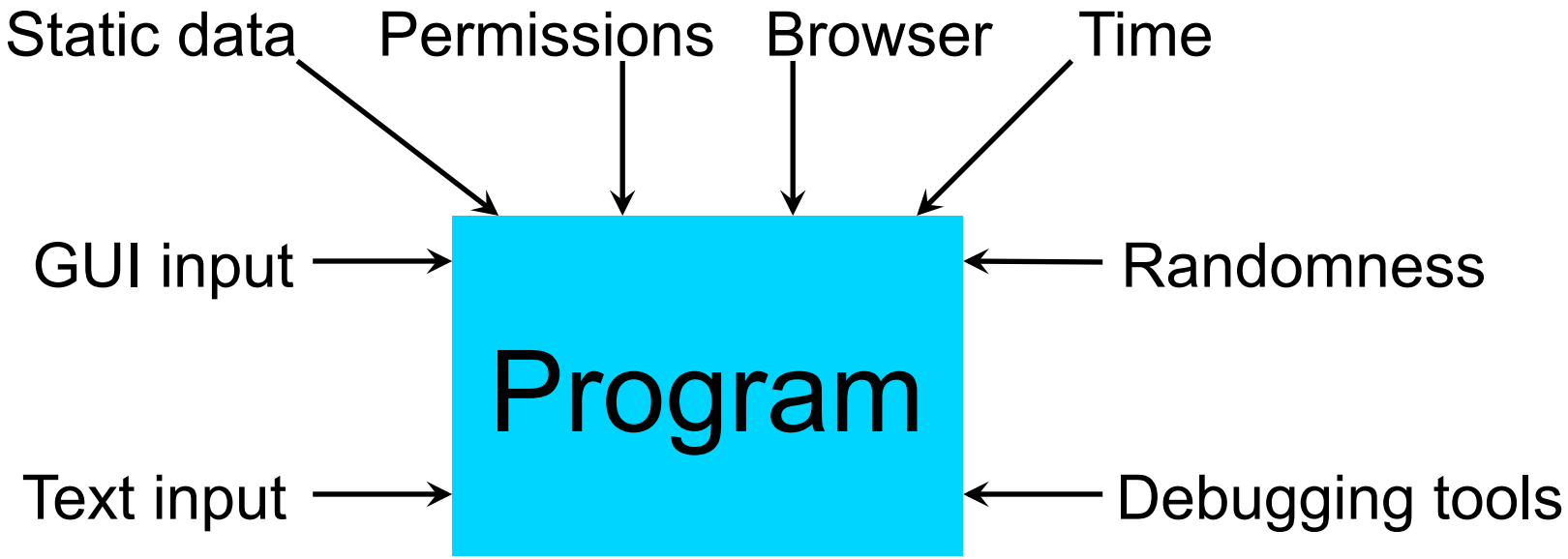
# Bug Jargon

| | | |
|---|---|---|
| Bohr bug | Quantum physics | Repeatable, manifests reliably |
| Heisenbug | Uncertainty principle | Disappears due to observation probe (e.g., time dependent) |
| Mandelbug | Mandelbrot set | Causes are complex, appears nondetermi-nistic (but is Bohr bug) |
| Schroedinbug | Schrödinger's cat, thought experiment | Hidden until first person notices it, then becomes show-stopper |

**Concepts**

T rack
R eproduce
A utomate
F ind origin
F ocus
I solate
C orrect

# Sources of Input



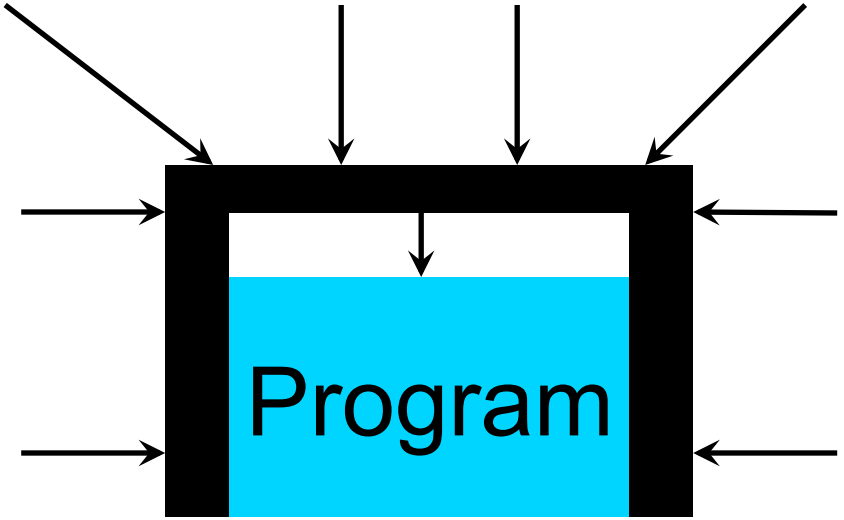

Difficult to reproduce problem if

- User's input ≠ developer's input
- Input is large and/or time sensitive

**Concepts**

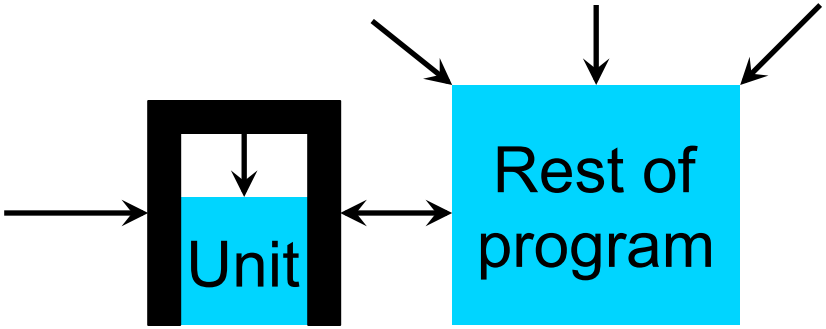
T rack
R eproduce
A utomate
F ind origin
F ocus
I solate
C orrect

# 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?

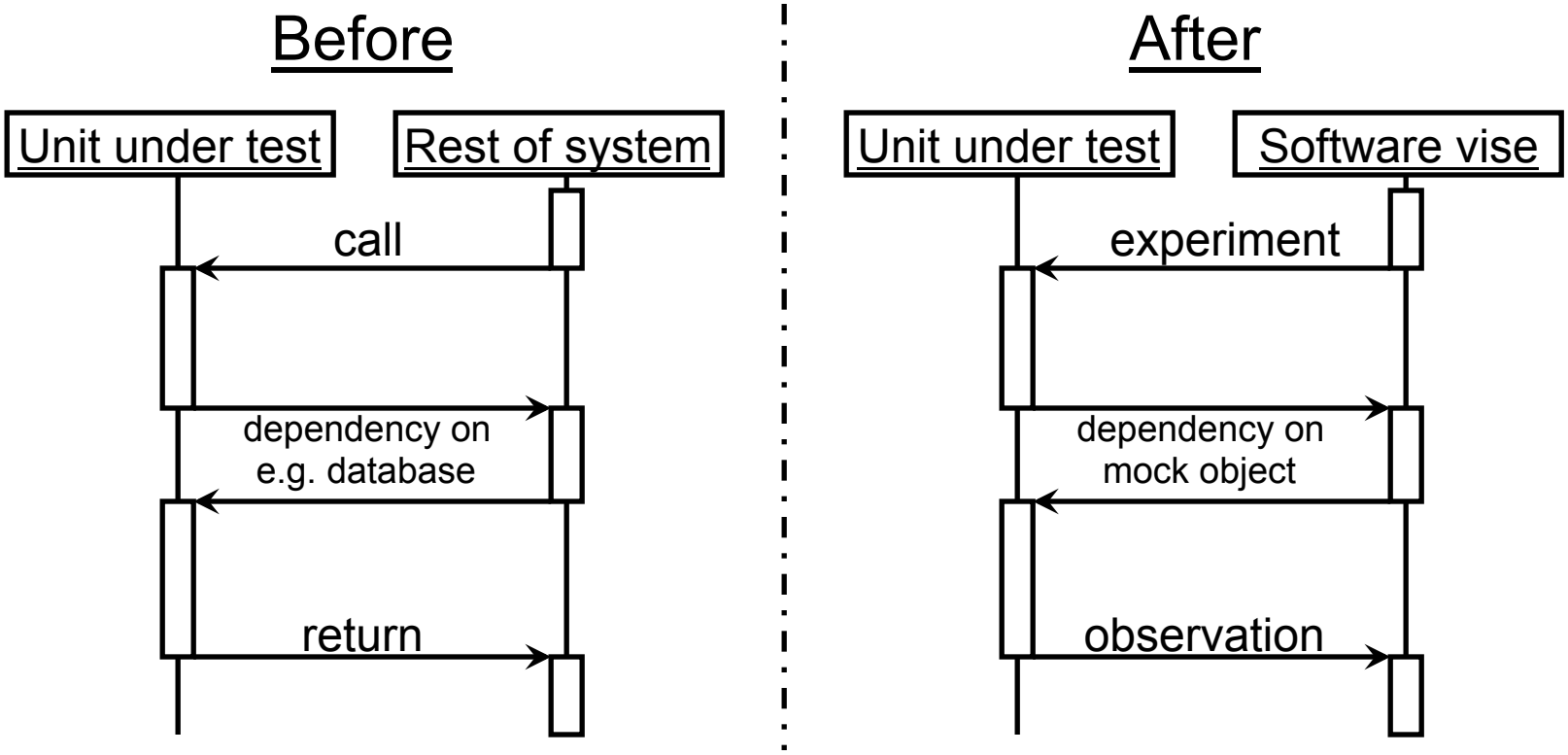
# System Tests vs. Unit Tests




- 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

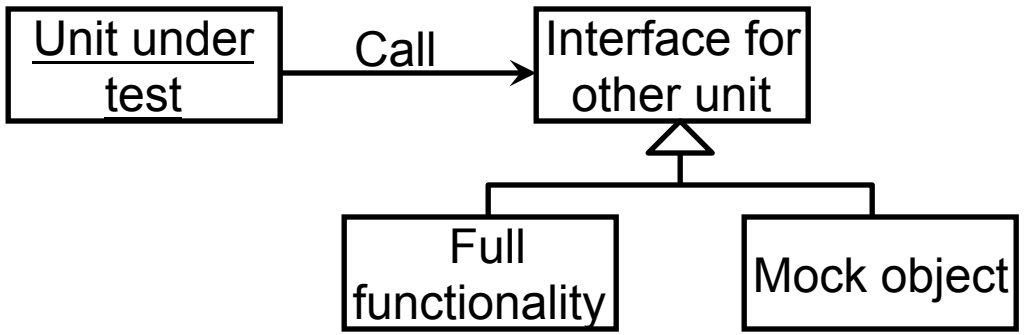
- To test a unit, must break its dependency on the rest of the system



**Concepts**

# Seams

T rack
R eproduce
A utomate
F ind origin
F ocus
I solate
C orrect

| | |
|---|---|
| Goal | Replace dependency by mock object<br>Do not modify code of unit under test |
| Solution | Use virtual method dispatch as "seam"<br>  |
| Challenge | Dependency may not be on method call<br>Refactor to object-oriented style first |

Concepts

T rack
R eproduce
A utomate
F ind origin
F ocus
I solate
C orrect

# 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 ⇒ use unit tests

Administrative

# Last Slide

- Today's lecture
  - Scientific method
  - Tools for scripting language debugging
  - TRAFFIC