Introduction to Debugging

The Problem

Facts on Debugging

- Software bugs cost ~60 bln US$/yr in US
- Improvements could reduce cost by 30%
- Validation (including debugging) can easily take up to 50-75% of the development time
- When debugging, some people are three times as efficient than others
How to Debug
(Sommerville 2004)

The Process

rack the problem
eproduce
utomate
ind Origins
ocus
olate
orrect
Tracking Problems

- Every problem gets entered into a *problem database*
- The *priority* determines which problem is handled next
- The product is ready when all problems are resolved

Problem Life Cycle

![Problem Life Cycle Diagram]
Reproduce

Randomness  Operating System
Communication  Concurrency
Interaction  Physics
Data  Debugger

Automate

// Test for host
public void testHost() {
    int noPort = -1;
    assertEquals(askigor_url.getHost(), "www.askigor.org");
    assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
    assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
    assertEquals(askigor_url.getQuery(), "id=sample");
}

Automate

- Every problem should be reproducible automatically
- Achieved via appropriate (unit) tests
- After each change, we re-run the tests
Finding Origins

1. The programmer creates a defect in the code.
2. When executed, the defect creates an infection.
3. The infection propagates.
4. The infection causes a failure.

This infection chain must be traced back – and broken.

Not every defect creates an infection – not every infection results in a failure.

The Defect
A Program State
Finding Origins

1. We start with a known infection (say, at the failure)
2. We search the infection in the previous state

A Program State
Search

During our search for infection, we focus upon locations that

• are possibly wrong
  (e.g., because they were buggy before)
• are explicitly wrong
  (e.g., because they violate an assertion)

Assertions are the best way to find infections!
Finding Infections

class Time {
public:
    int hour(); // 0..23
    int minutes(); // 0..59
    int seconds(); // 0..60 (incl. leap seconds)

    void set_hour(int h);
    ...
}

Every time between 00:00:00 and 23:59:60 is valid

Finding Origins

bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane()); // Precondition
    ...
    assert (sane()); // Postcondition
}

Finding Origins

bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

sane() is the invariant of a Time object:

• valid before every public method
• valid after every public method
Finding Origins

- Precondition fails = Infection before method
- Postcondition fails = Infection after method
- All assertions pass = no infection

```cpp
void Time::set_hour(int h)
{
    assert (sane()); // Precondition
    ...
    assert (sane()); // Postcondition
}
```

Complex Invariants

class RedBlackTree {
    ...
    boolean sane() {
        assert (rootHasNoParent());
        assert (rootIsBlack());
        assert (redNodesHaveOnlyBlackChildren());
        assert (equalNumberOfBlackNodesOnSubtrees());
        assert (treeIsAcyclic());
        assert (parentsAreConsistent());
        return true;
    }
}

Assertions

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✘</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Focusing

- All possible influences must be checked
- Focusing on most likely candidates
- Assertions help in finding infections fast

Isolation

- Failure causes should be narrowed down systematically
- Use observation and experiments

Scientific Method

1. Observe some aspect of the universe.
2. Invent a hypothesis that is consistent with the observation.
3. Use the hypothesis to make predictions.
4. Tests the predictions by experiments or observations and modify the hypothesis.
5. Repeat 3 and 4 to refine the hypothesis.
Scientific Method

Explicit Hypotheses

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>The execution causes a[0] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>At Line 37, a[0] should hold.</td>
</tr>
<tr>
<td>Experiment</td>
<td>Line 37.</td>
</tr>
<tr>
<td>Observation</td>
<td>a[0] holds as predicted.</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Hypothesis is confirmed.</td>
</tr>
</tbody>
</table>

Keeping everything in memory is like playing Mastermind blind!
Explicit Hypotheses

Isolate

- We repeat the search for infection origins until we found the defect
- We proceed systematically along the scientific method
- *Explicit steps* guide the search — and make it repeatable at any time

Correction

Before correcting the defect, we must check whether the defect
- actually is an *error* and
- *causes* the failure

Only when we understood both, can we correct the defect
The Devil’s Guide to Debugging

Find the defect by guessing:

• Scatter debugging statements everywhere
• Try changing code until something works
• Don’t back up old versions of the code
• Don’t bother understanding what the program should do

Don’t waste time understanding the problem.

• Most problems are trivial, anyway.

Use the most obvious fix.

• Just fix what you see:
  
x = compute(y)
  // compute(17) is wrong - fix it
  if (y == 17)
    x = 25.15

  Why bother going into compute()?
Successful Correction

Homework

- Does the failure no longer occur?  
  (If it does still occur, this should come as a big surprise)
- Did the correction introduce new problems?
- Was the same mistake made elsewhere?
- Did I commit the change to version control and problem tracking?

The Process

- Trace the problem
- Reproduce
- Automate
- Indent Origins
- Focus
- Isolate
- Correct
Online Course on Debugging

Which hypotheses are consistent with our observations so far?

<table>
<thead>
<tr>
<th>Input</th>
<th>Expected</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
</tr>
<tr>
<td>&quot;bar&quot;</td>
<td>&quot;bar&quot;</td>
<td>&quot;bar&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(empty)</td>
</tr>
</tbody>
</table>

The error is due to `tag` being set.

What's in this Course

How Debuggers Work

Hypothesis - Experiment - Observation
What's in this Course

How debuggers work
Asserting expectations
Simplifying failures

assert tree.hasNoCycles()
What's in this Course

1. How debuggers work
2. Asserting expectations
3. Simplifying failures
4. Tracking origins
5. Reproducing failures
6. Learning from mistakes
Summary