Systematic Debugging

Software Engineering
Andreas Zeller • Saarland University
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
$ ls
bug.c
$ gcc-2.95.2 -O bug.c
gcc: Internal error: program cc1 got fatal signal 11
Segmentation fault
$
How to Debug
(Sommerville 2004)

Locate error → Design error repair → Repair error → Re-test program
The Process

T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect
### Tracking Problems

#### Time Tracking

<table>
<thead>
<tr>
<th>Ticket</th>
<th>Planned</th>
<th>Spent</th>
<th>Remaining</th>
<th>Accuracy</th>
<th>Customer</th>
<th>Summary</th>
<th>Component</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>#6</td>
<td>10h</td>
<td></td>
<td>10h</td>
<td>0.0</td>
<td>milestone1</td>
<td>asdf</td>
<td>component1</td>
<td>new</td>
</tr>
<tr>
<td>#5</td>
<td>2h</td>
<td>4h</td>
<td>0h</td>
<td>2.0</td>
<td>milestone1</td>
<td>234</td>
<td>component1</td>
<td>new</td>
</tr>
<tr>
<td>#4</td>
<td>4h</td>
<td>4h</td>
<td></td>
<td>0.0</td>
<td>milestone1</td>
<td>yxcv</td>
<td>component1</td>
<td>new</td>
</tr>
<tr>
<td>#3</td>
<td>4h</td>
<td>4h</td>
<td></td>
<td>0.0</td>
<td>milestone1</td>
<td>test3</td>
<td>component1</td>
<td>closed</td>
</tr>
<tr>
<td>#2</td>
<td>4h</td>
<td>2h</td>
<td>2h</td>
<td>0.0</td>
<td>milestone1</td>
<td>test2</td>
<td>component1</td>
<td>new</td>
</tr>
<tr>
<td>#1</td>
<td>8h</td>
<td>7.0h</td>
<td>3.0h</td>
<td>2.0</td>
<td>milestone1</td>
<td>test1</td>
<td>component1</td>
<td>new</td>
</tr>
<tr>
<td>#7</td>
<td>1h</td>
<td></td>
<td></td>
<td>-1.0</td>
<td>milestone2</td>
<td>3452345</td>
<td>component1</td>
<td>new</td>
</tr>
</tbody>
</table>

**Note:** See [TracReports](https://trac.edgewall.com/) for help on using and creating reports.
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

- **UNCONFIRMED**
- **NEW**
- **ASSIGNED**
- **REOPENED**
- **VERIFIED**
- **CLOSED**

States:
- **INVALID**
- **DUPLICATE**
- **FIXED**
- **WONTFIX**
- **WORKSFORME**

Flow:
- From **UNCONFIRMED** to **NEW**
- From **NEW** to **ASSIGNED**
- From **ASSIGNED** to **RESOLVED**
- From **RESOLVED** to **VERIFIED**
- From **VERIFIED** to **CLOSED**
- From **REOPENED** to **ASSIGNED** if resolution is **FIXED**
Reproduce

Randomness → Operating System

Communication

Interaction

Data

Concurrency

Physics

Debugger
```java
// 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.
Finding Origins
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
delete list;
A Program State
Search

WHERE'S WALDO?
Focus

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

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

        return true;
    }
}
## Assertions

<table>
<thead>
<tr>
<th>T</th>
<th>R</th>
<th>A</th>
<th>F</th>
<th>I</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- **Correct Assertions:** 21
- **Incorrect Assertions:** 1
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. Test the predictions by experiments or observations and modify the hypothesis.

5. Repeat 3 and 4 to refine the hypothesis.
Scientific Method

Hypothesis

Prediction

Experiment

Observation + Conclusion

Hypothesis is supported:
refine hypothesis

Hypothesis is rejected:
create new hypothesis

Problem Report

Code

Run

More Runs

Diagnosis
Mastermind

The Challenging Game of Logic and Deduction

Break the hidden code! New sleek design with improved built-in storage tray and code shield!
The execution causes $a[0] = 0$.

At Line 37, $a[0] = 0$ should hold.

Observe $a[0]$ at Line 37.

$a[0] = 0$ holds as predicted.

Hypothesis is confirmed.

<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>Keeping everything in memory is like playing mastermind blind!</td>
</tr>
<tr>
<td>Observations</td>
<td>$a[0] = 0$ holds as predicted.</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Hypothesis is confirmed.</td>
</tr>
</tbody>
</table>
Explicit Hypotheses
Rubberducking
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
The Devil’s Guide to Debugging

Don’t waste time understanding the problem.

• Most problems are trivial, anyway.
The Devil’s Guide to Debugging

Use the most obvious fix.

• Just fix what you see:

```java
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

T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect
Automated Debugging
(Udacity)

Which hypotheses are consistent with our observations so far?

- Double quotes are stripped from targeted input

<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>foo</td>
</tr>
<tr>
<td>&quot;bar&quot;</td>
<td>&quot;bar&quot;</td>
<td>bar</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>(empty)</td>
</tr>
</tbody>
</table>

The error is due to tag being set.
The Process

- Track the problem
- Reproduce
- Automate
- Find Origins
- Focus
- Isolate
- Correct

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.

Summary

Scientific Method

- Problem Report
- Hypothesis: Prediction → Experiment
  - Observation + Conclusion
  - Hypothesis is supported: refine hypothesis
  - Hypothesis is rejected: create new hypothesis
  - Diagnosis

Online Course on Debugging

Which hypotheses are consistent with our observations so far?

```
In: "foo" "bar" "foo"
Out: "foo"
(early)
```

The error is due to tag being set.