Fixing the Bug

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Exam (updated)

on Tuesday, 2003-02-18, 14:00 in lecture room 45/001 (here)

Written examination, duration: 90 minutes

Tools: course material, books, papers; no electronic devices

Final grade will be

- 20% exercises, 80% examination or
- 100% examination (whatever is best)

Q & A lab on Friday, 2003-02-14

Register by e-mail to Holger Cleve ⟨*cleve@cs.uni-sb.de*⟩ until Friday, 2003-02-14

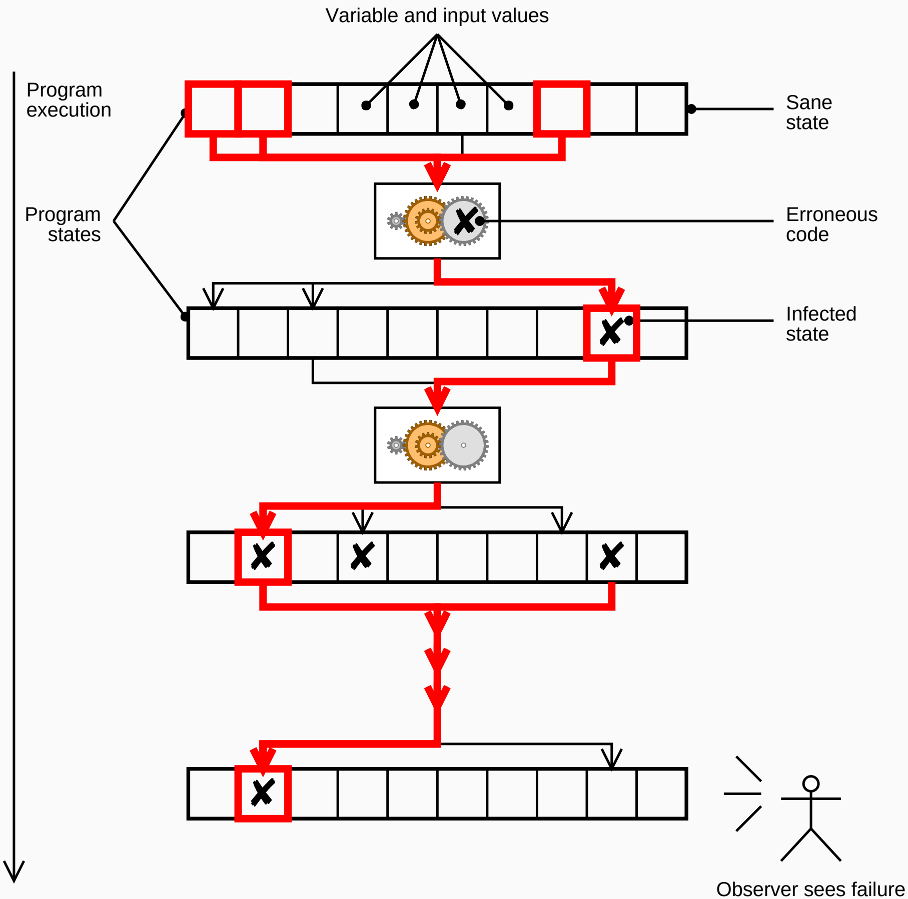
Where are we?

Reproduce Problem. Make sure the problem can be reproduced at will.

Scientific Method. Isolate the cause-effect chain from the root cause to the failure.

Fix Program. Ensure the failure no longer occurs.

Breaking the Cause-Effect Chain



Problems with Fixing

The *hard* part is finding the defect.

Fixing a defect is the easy part.

However, fixing a defect is so easy that it is *likely to induce new defects.*

Defect corrections have more than 50% chance of being wrong the first time.

Fixing Guidelines

Summary:

- Understand the problem
- Understand the program
- Prove causality
- Relax
- Fix the problem, not the symptom
- Change one thing at a time
- Check the fix
- Check for side effects

Understand the Problem

Before fixing, be sure to understand the problem.

You must know

- that the failure is a failure
- the cause-effect chain from root cause to failure
- the infection site (i.e. the moment when the infection occurs)

Your *hypothesis* about the problem cause must become a *theory*—a theory that allows you to *predict* problem occurrences.

Understand the Program

Understand the program, not just the problem.

A change to the code may induce new effects in other parts of the program.

Protect against such effects

- by understanding the vicinity of the fix
- by understanding possible effects of the fix (e.g. a static forward slice)
- by running tests that protect against undesired effects
- by having your change reviewed by others

Prove Causality

Make sure a cause is a cause.

If you diagnosed a potential failure cause, *prove it*—by showing that the failure does not occur if the cause is altered.

- Start with finding causes in the program input,
- proceed with program data,
- end up with program code as the very last step.

Before you make a change to the code,
be confident that it will work!

Save Original Code

Never start changing code before saving the original.

If you don't keep track of old versions, you won't be able to reproduce the original problem—and you won't be able to return to the original code.

Always use version control.

Relax

Relax long enough to make sure your solution is right.

Don’t rush into solving a problem. You need

- well-qualified judgments
- complete understanding of how the failure came to be
- a proof that your solution actually fixes the problem
- confidence that your solution does not induce new problems.

Wishful thinking doesn’t fix bugs!

Fix the Problem, not the Symptom

Use the most general fix available.

Breaking the cause-effect chain for a particular failure is easy—simply check for the infectuous value and correct it.

The issue, though, is to break the cause-effect chain in such a way that *as many failures as possible* are prevented.

Fix the Problem, not the Symptom (2)


```
for (claim = 1; claim < numClients; claim++)
{
    sum[client] += claimAmount[claim];
}

if (client == 45)
    sum[45] += 3.45;
else if (client == 37 && numClaims[37] == 0)
    sum[37] = 0.0;
```

Where's the problem with such fixes?

Always use the most general fix!

Change one Thing at a Time

Do not attempt to fix multiple defects at the same time.

Rationale: multiple fixes can interfere with each other and create failures that look like the original one.

Then you don't know whether

- you actually fixed the defect,
- you fixed the defect, but introduced a new, similar, defect
- you did not fix the defect, but introduced a similar one

Check your Fix

After fixing, make sure the fix solves the problem.

This is achieved by reproducing the original failure.

Remember: you should be confident about your fixes!

Being wrong about a fix should

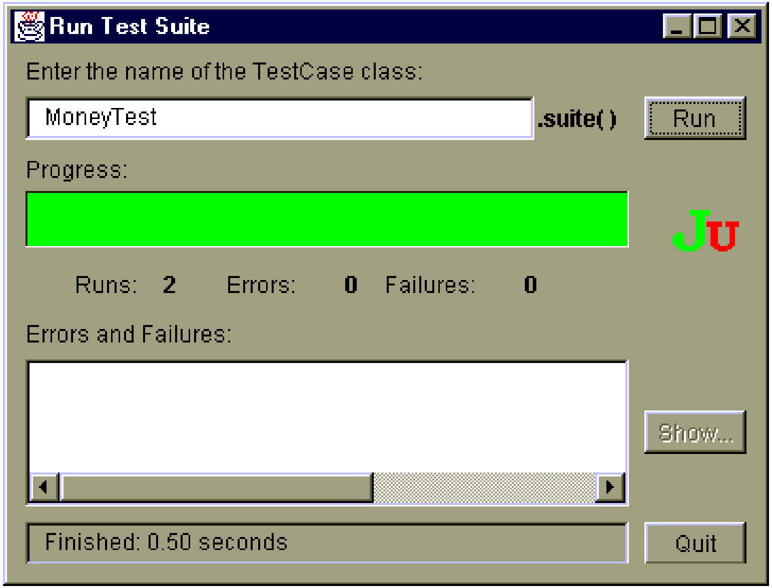
- leave you astonished
- cause self-doubt, personal re-evaluation, and deep soul-searching
- happen rarely.

Again: Before you make a change to the code, be confident that it will work!

Check for Side Effects

After fixing, make sure no new defects are introduced.

This is typically done by running an automated regression test.



Check for Side Effects (2)

More ideas:

Have someone review the fix. Many industrial environments have *formal procedures* for adding new production code. Typically, this involves a review of the code.

Reviewing is also common in open source programming—try to convince Linus Thorvalds to add this piece of code to the Linux kernel.

Select regression test cases. In case of a regression test suite that takes too long, you can use *forward slicing* to include only those tests that are actually affected by a change.

Preventing Bugs

After the problem has been fixed, you might want to *learn* from it.

Look for similar defects. Can we identify similar problems in code?

Keep debugging code. Make sure similar problems are detected early.

Keep bug metrics. Learn from mistakes.

Look for Similar Defects

After fixing, check whether the same mistake was made elsewhere.

Example:

```
char *s = malloc(sizeof(char) * 13);
strcpy(s, "Hello, world!");
```

Fix:

```
char *s = malloc(sizeof(char) * 14);
strcpy(s, "Hello, world!");
```

Now check for all other `malloc` calls! (written by the same person?)

Look for Similar Defects (2)

Looking for similar defects is a great opportunity to *refactor* the code and prevent similar mistakes.

Example:

```
char *s = malloc(sizeof(char) * 14);
strcpy(s, "Hello, world!");
```

Much better approach:

```
char *s = strdup("Hello, world!");
```

where `strdup(arg)` calculates the amount of required memory—using `strlen(arg) + 1` or similar—and, by the way, handles the case that `malloc()` returns NULL.

Keep Debugging Code

Make finding the infection easier next time.

If you inserted assertions to narrow down the infection, *keep them in the code.*

If you inserted statements to examine state, *turn them into assertions, logging macros or similar.*

Keep Bug Metrics

Understand why defects occur.

Idea: gather data not only about *problems* (as in problem tracking systems), but also about the causing *defects*:

What was the defect? This is a description of the defect—typically with categories like “use of non-initialized variable”, “bad control flow”, “heap misuse” etc.

Where was the defect located? Was the data flow or the control flow wrong? Which component was affected?

When was the defect introduced? Did it originate in the requirements / design / coding phase?

Why was the defect introduced? Check version control logs for the original motivation.

Keep Bug Metrics (2)

Once a database of defects exists (typically as part of the problem tracking system), it can be used to answer the following questions:

- Which defects occur at *which production stage?*
  - You may want to prevent such defects
- Which modules have had the *most defects?*
  - error-prone modules are likely infection sources
  - error-prone modules may be subject to reengineering
- If we see a failure, which other defects have caused *similar failures* so far?
  - search for a specific defect category—“heap misuse” might be found using a Valgrind-like tool

The Devil's Guide to Debugging

Find the defect by guessing. This includes:

- Scatter debugging statements throughout the program.
- 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);
if (y == 25)
    x = 25.15;
```

Why bother going all the way through `compute()`?

Debugging by Superstition

The computer does not like me, so I'm lost. (There is no such thing as a computer vendetta.)

I'm re-running it in case the computer made a mistake. (Just waste your time.)

The computer is wrong. (The chance that you uncover a defect in the computer is infinitesimal.)

The program got corrupted on disk. (The whole computer would crash, then.)

The computer lost my program. (Only yours? You probably deleted it.)

Somebody hacked my account and changed my program. (Come on! Who would care for your program files?)

Concepts

- Before fixing, be sure to understand the problem.
- Understand the program, not just the problem.
- Never start changing code before saving the original.
- Relax long enough to make sure your solution is right.
- *Use the most general fix available.*
- Do not attempt to fix multiple defects at the same time.

Concepts (2)

- After fixing,
  - make sure the fix solves the problem.
  - make sure no new defects are introduced.
  - check whether the same mistake was made elsewhere.
- Keep debugging code to make finding the infection easier next time.
- Keep bug metrics to understand why defects occur.

References

- Steve McConnell, *Code Complete*, Microsoft Press, Chapter 26 “Debugging” (end especially 26.2 “Fixing an Error”). http://www.stevemcconnell.com/cc.htm