Fixing the Defect
Andreas Zeller
1. The programmer creates a *defect* – an error in the code.

2. When executed, the defect creates an *infection* – an error in the state.

3. The infection *propagates*.

4. The infection causes a *failure*.

This infection chain must be traced back – and broken.
Techniques

Infections
- e.g. a failed assertion

Causes
- e.g. a[2] = 0 causes the failure

Anomalies
- e.g. f() executed only in failing run

Dependences
- e.g. a[2] comes from a[0]

How do we integrate these techniques?
All Techniques

Grid representation
Dependencies
Observation
Observation
Assertion
Assertion
Cause Transition
The Defect
Validating the Defect

Any element of the infection chain must be

- *infected* – i.e., have an incorrect value
- *a failure cause* – i.e., changing it causes the failure to no longer occur

Demonstrate by experiments and observation
Is the Error a Cause?

```c
a = compute_value();
printf("a = %d\n", a);
```

\[
a = 0
\]
Is the Cause an Error?

```c
balance[account] = 0.0;
for (int i = 0; i < n; i++)
    balance[account] += deposit[i]

// account 123 is wrong - fix it
if (account == 123)
    balance[123] += 45.67
```
static void shell_sort(int a[], int size) {
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i += h)
            { int v = a[i];
                for (j = i; j >= h && a[j - h] > v; j -= h)
                    a[j] = a[j - h];
                if (i != j)
                    a[j] = v;
            }
    } while (h != 1);
}
Validating Causality

• In principle, we must show causality for each element of the infection chain

• However, a successful correction retrospectively validates causality:
  • Since the failure has gone, we have proven that the defect caused the failure
  • Yet, we must not fall into ignorant surgery
Think before you code

Before applying a fix, you must understand

• how your code change will break the infection chain, and

• how this will make the failure (as well as other failures) no longer occur

In fact, you have a theory about 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:

```java
x = compute(y)
// compute(17) is wrong – fix it
if (y == 17)
  x = 25.15

Why bother going into compute()?```
Correcting the Defect
Homework

Does the failure no longer occur?

• If the failure is still there, this should
  • leave you astonished
  • cause self-doubt + deep soul-searching
  • happen rarely

• Note that there may be a second cause
Homework (2)

Did the correction introduce new problems?

- Have corrections peer-reviewed
- Have a regression test to detect unintended changes in behavior
- Check each correction individually
Homework (3)

Was the same mistake made elsewhere?

• Check for other defects caused by the same mistake
• Other code of the same developer
• Code involving the same APIs
Homework (4)

Did I commit the change?

- Be sure to commit your change to
  - the version control system
  - the bug tracking system
Workarounds

Correcting the defect may be impossible:

• Unable to change
• Risks
• Design flaw

A workaround solves the problem at hand – but mark it as a temporary solution