CSE 331
Software Design & Implementation

James Wilcox

Autumn 2021

Debugging

A Bug's Life

defect – mistake committed by a human

error – incorrect computation

failure – visible error: program violates its specification

Debugging starts when a failure is observed

- Unit testing
- Integration testing
- In the field

Goal of debugging is to go *from failure back to defect*

Bug Removal

Tools, Inspection, & Testing

- increase confidence that the code is correct
- uncover problems by running the code

Defensive programming

- lay traps as you code to catch bugs when run

Debugging

- find out why a program is not functioning as intended

Testing ≠ debugging

- *test*: reveals existence of problem
- *debug*: pinpoint location & cause of problem
- debugging is often painful, testing is not

Defense in depth

Levels of defense:

1. Make errors *impossible*
   - examples: Java prevents type errors, memory corruption
     Python prevents key mutation

2. Don't introduce defects
   - "get things right the first time" (by reasoning & unit testing)

3. Make errors *immediately visible* (often by defensive programming)
   - examples: assertions, **`checkRep`**
   - reduce *distance* from error to failure

(subtle bugs like key mutations are hard to find
because of the distance between error and failure)

First defense: Impossible by design

In the language

- Java prevents type mismatches, memory overwrite bugs; guaranteed sizes of numeric types, …

In the protocols/libraries/modules

- TCP/IP guarantees data is not reordered
- **`BigInteger`** guarantees there is no overflow

In self-imposed conventions

- immutable data structure guarantees behavioral equality
- **`finally`** block can prevent a resource leak

Caution: You must maintain discipline

Second defense: Correctness

Get things right the first time

- think before you code (don't code before you think!)
- if you're making **lots** of easy-to-find defects,
  you're probably **also** making hard-to-find defects

Especially important when debugging is going to be hard

- concurrency, real-time environment, no access to customer environment, etc.
- (formalism gets more important as the problems get harder)

The key techniques are everything we have been learning:

- forward & backward reasoning
- clear and complete specs
- these techniques lead to *simpler software*

Strive for simplicity

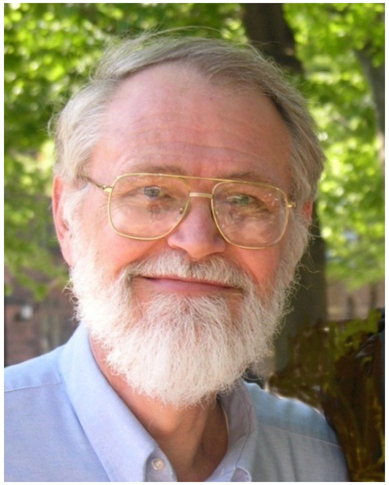
"There are two ways of constructing a software design:

> One way is to make it so simple that there are obviously no deficiencies, and
>
> the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."

Sir Anthony Hoare

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."



Brian Kernighan

Second defense: Correctness

Find errors by testing before you check in the code:

Unit testing: when you test a module in isolation, any failure is due to a defect in that unit (or the test driver)

Regression testing: run tests as often as possible when changing code. If there is a failure, chances are there's a mistake in the code you just changed (or the new code is triggering a bug that hadn't been observed before)

Test early and often. More tests is almost never a bad thing.

Third defense: Immediate visibility

If we can't prevent errors, we can try to spot them early

asserts (e.g., in checkRep): check at runtime that the program is in the state that we are expecting.

Assertions make more sense for server code

- Google policy (10+ years ago) on servers:
  - assertions always enabled
  - if any failure is detected, **crash** the program
- Microsoft policy (20+ years ago) on clients:
  - try to recover from failures. hide them from the user

Benefits of immediate visibility

Failure is likely to be closer to the defect
- failure can occur far from the mistake that caused it
- immediate visibility reduces the search time to find the defect

Defect is less likely to have infected other parts of the program
- the longer we wait, the more code we'll likely have to change

Don't program in ways that hide errors
- this lengthens distance between defect and failure

Don't hide errors

```
// requires: x must be present in A
int i = 0;
while (true) {
  if (A[i] == x) break;
  i++;
}
```

This code fragment searches an array **A** for a value **x**

- value is guaranteed to be in the array
- what if that guarantee is broken (by a defect)?

Don't hide errors

```
// requires: x must be present in a
int i = 0;
while (i < A.length) {
  if (A[i] == x) break;
  i++;
}
```

Fixes the bug so the loop always exits

- but no longer guaranteed that `A[i] == x`
- if other code relies on this, then problems arise later
  - this just hides the bug!

Don't hide errors

```
// requires: x must be present in a
int i = 0;
while (i < A.length) {
  if (A[i] == x) break;
  i++;
}
assert i != A.length : "key not found";
```

- Assertions let us document and check invariants at run time
- Abort/debug program as soon as problem is detected
  - turn an error into a failure
- Unfortunately, we may still be a long distance from the *defect*
  - the defect is what caused **x** not to be in the array

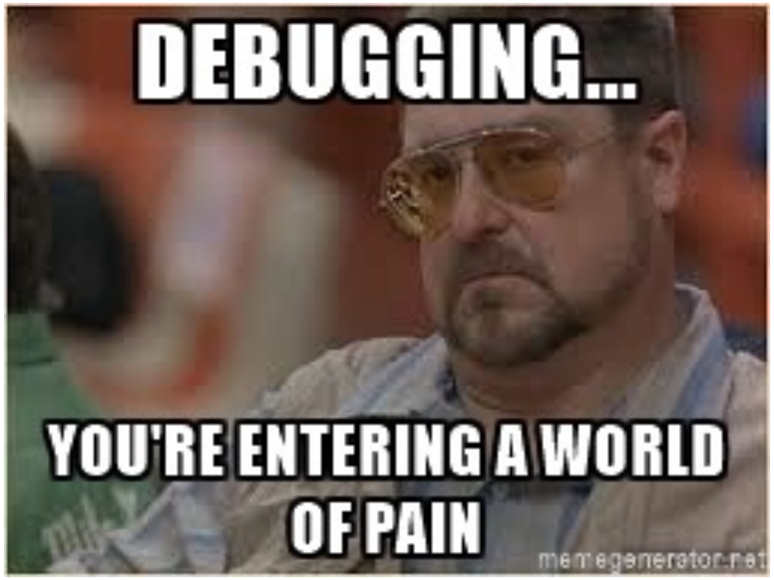
Last resort: debugging

Defects happen

- people are imperfect
- industry average (?): 10 defects per 1000 lines of code

Defects are sometimes not immediately clear from the failure

That means…



Basic Bug Removal

Work through the following steps:

step 1 – Clarify symptom (simplify input), create "minimal" test

step 2 – Localize and understand cause

step 3 – Fix the defect

step 4 – Rerun *all* tests, old and new

The bug removal process

step 1: *find (small) repeatable test case that produces the failure*

- smaller test case will make step 2 easier
- do *not* start step 2 until you have a repeatable test

step 2: *narrow down location and cause*

- *loop*: (a) study the data (b) hypothesize (c) experiment
- experiments often involve changing the code
- do *not* start step 3 until you understand the cause

step 3: *fix the defect*

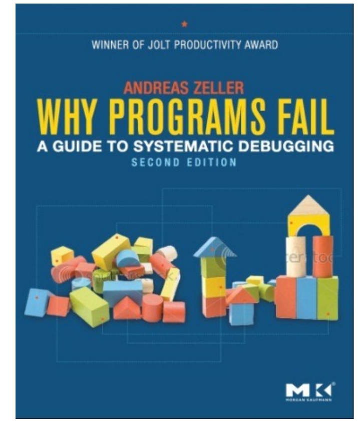
- is it a simple typo or a design flaw?
- does it occur **elsewhere** in the code?

step 4: *run all the tests (including the new one)*

- is this failure fixed? are any other new failures introduced?

Debugging and the scientific method

- Debugging should be *systematic*
  - carefully *decide* what to do
    - don’t flail about randomly!
  - may help to keep a *record* of what you tried
  - don’t get sucked into fruitless avenues



- Use an iterative scientific process:

Formulate a hypothesis

Interpret results

Design an experiment

Perform an experiment

Example

```
// returns true iff sub is a substring of full
// (i.e. iff there exists A,B such that full=A+sub+B)
boolean contains(String full, String sub);
```

User bug report:

It can't find the string **"very happy"** within:

"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."

Poor responses:

- See accented characters, panic about not knowing about Unicode, begin unorganized web searches and inserting poorly understood library calls, …
- Start tracing the execution of this example

Better response: simplify/clarify the symptom…

Reducing input size

Find a simple test case by divide-and-conquer

Pare test down:

Can not find **`"very happy"`** within

> **`"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."`**
>
> **`"I am very very happy to see you all."`**
>
> **`"very very happy"`**

Can find **`"very happy"`** within

> **`"very happy"`**

Can not find **`"ab"`** within **`"aab"`**

General strategy: simplify

Find simplest input that will provoke failure

- usually not the input that revealed existence of the defect

Start with data that revealed the defect

- keep paring it down (binary search can help!)
- sometimes leads directly to an understanding of the cause

When not dealing with just one method call:

- “test input” is the set of steps that reliably trigger the failure
- same basic idea

Localizing a defect

Sometimes you can take advantage of modularity

- start with everything, take away pieces until failure goes away
- start with nothing, add pieces back in until failure appears

Binary search speeds up this process too

- error happens somewhere between first and last statement
- do binary search on that ordered set of statements
  - is the state correct after the middle statement?

Binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```

no problem yet

Check intermediate result at half-way point

problem exists

Binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10)
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```

no problem yet

Check intermediate result at half-way point

problem exists

Detecting Bugs in the Real World

Real systems

- large and complex
- collection of modules, written by multiple people
- complex input
- many external interactions
- non-deterministic

Replication can be an issue

- infrequent failure (the worst)
- instrumentation eliminates the failure (the worst of the worst)

Defects cross abstraction barriers

Large time lag from corruption (error) to detection (failure)

Heisenbugs

In a sequential, deterministic program, failure is repeatable

But the real world is not that nice…

- continuous input/environment changes
- concurrency and parallelism
- failure occurs randomly
  - literally depends on results of random-number generation

Common for debugging because...

- these are most likely bugs to **sneak past** reasoning & testing

Bugs hard to reproduce when:

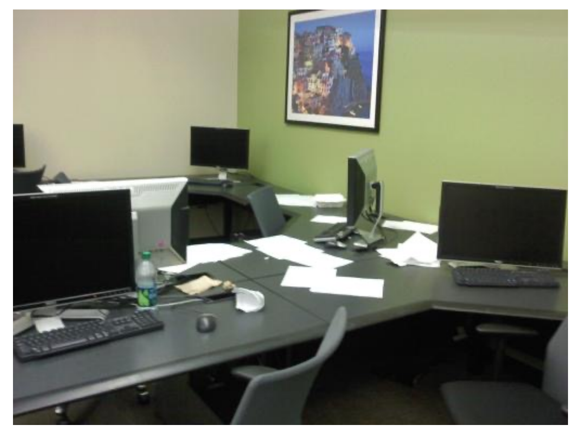
- use of debugger or assertions makes failure goes away
  - due to timing or assertions having side-effects
- only happens when under heavy load
- only happens once in a while

Debugging In Harsh Environments

Failure is non-deterministic, difficult to reproduce

Can’t print or use debugger

Can’t change timing of program (or defect/failure depends on timing)



Such bugs are more common when users are your testers!

Logging Events

Log (record) events during execution as program runs (at full speed)

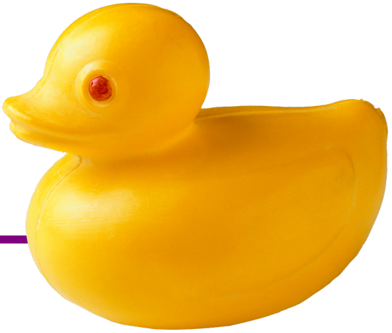
Examine logs to help reconstruct the past

- particularly on failing runs
- and/or compare failing and non-failing runs

The log may be all you know about a customer’s environment

- (some amount of logging is fairly standard)
- needs to tell you enough to reproduce the failure

More Tricks for Hard Bugs



Rebuild system from scratch

– bug could be in your build system or persistent data structures

Make sure that you have correct source code

– check out fresh copy from repository; recompile everything

Explain the problem to a friend (or to a rubber duck)

– The Pragmatic Programmer calls this “r

Make sure it is a bug

– program may be working correctly!

More Tricks for Hard Bugs

Rebuild system from scratch
- bug could be in your build system or persistent data structures

Explain the problem to a friend (or to a rubber duck)
- The Pragmatic Programmer calls this “rubber ducking”

Make sure that you have correct source code
- check out fresh copy from repository; recompile everything

Make sure it is a bug
- program may be working correctly!

And things we already know:

- minimize input required to exercise bug (exhibit failure)
- add more checks to the program
- add more logging

Where is the defect?

The defect is **not** where you think it is (or else you'd have found it)

- ask yourself where it can not be; explain why

Look for simple easy-to-overlook mistakes first, e.g.,

- reversed order of arguments:
  `Collections.copy(src, dest);`
- spelling of identifiers: **`int hashcode()`**
  `@Override` can help catch method name typos
- same object vs. equal: **`a == b`** versus **`a.equals(b)`**
- deep vs. shallow copy

When the going gets tough

Reconsider assumptions

- e.g., has the OS changed? Is there room on the hard drive? Is it a leap year? 2 full moons in the month?
- debug the code, *not* the comments
  - ensure that comments and specs describe the code

Start documenting your system

- gives a fresh angle, and highlights area of confusion

Get help

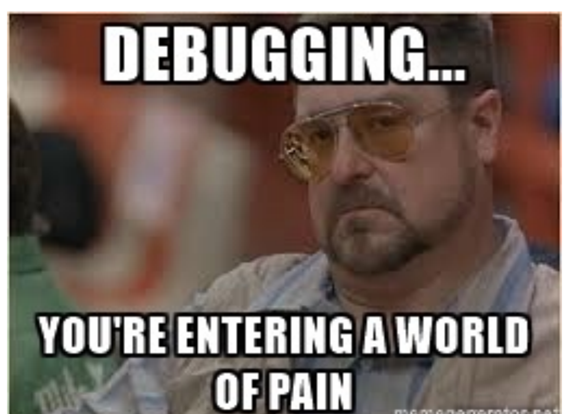
- we all develop blind spots
- explaining the problem often helps (even to rubber duck)

Walk away

- sleep! often you can't see the problem because you're too tired
- one good reason to start early

Summary

- Debugging occurs when tools & inspection fail to catch a bug
- Debugging is the **search** from a **failure** back to the **defect**
  - defect = the actual bug, somewhere in the code
  - failure = bad effects from the bug becoming visible to users
    (crash, error message, incorrect result, etc.)
- Debugging can be *hours* (or even *days!*) of *frustrating* work
  - bugs that get past tools & inspection are usually the **most subtle**



Advice

Use assertions (& checkRep) to make failures **visible** ASAP

- reduces the work in debugging
- shortens the search from failure to defect

Debugging should be a systematic process

- use the *scientific method*

Understand the source of defects

- find similar ones and prevent them in the future