Debugging

CSE 331
University of Washington

Michael Ernst

Mark II logbook, Sep 9, 1947

9/9

0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995 correct
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ ~~(-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 11,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 arctangent started.
1700 closed down.

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: go *from failure back to defect*

Ways to get your code right

- Design & verification
  - Prevent defects from appearing in the first place
- Defensive programming
  - Programming debugging in mind: fail fast
- Testing & validation
  - Uncover problems (even in spec), increase confidence
- Debugging
  - Find out why a program is not functioning as intended
- Testing ≠ debugging
  - test: reveals existence of problem (failure)
  - debug: pinpoint location + cause of problem (defect)

Defense in depth

1. Make errors impossible
   Java prevents type errors, memory corruption
2. Don't introduce defects
   Correctness: get things right the first time
3. Make errors immediately visible
   Example: assertions; **`checkRep()`**
   Reduce distance from error to failure
4. Debugging is the last resort
   Work from effect (failure) to cause (defect)
   Scientific method: Design experiments to gain information about the defect
   Easiest in a modular program with good specs and test suites

First defense: Impossible by design

Use the language

Java prevents type mismatch, memory overwrite errors

Use protocols/libraries/modules

TCP/IP guarantees that data is not reordered

`BigInteger` guarantees that there is no arithmetic overflow

Use self-imposed conventions

- Immutable data structure guarantees behavioral equality
- **`try`**-with-resources (or **`finally`**) prevents resource leak
- Avoid recursion to prevent stack overflow

Caution: You must maintain the discipline

Second defense: Correctness

Get things right the first time

- **Think** before you code. Don't code before you think!
- Don't use the compiler as crutch – does not find all defects
  - If it is finding defects, you are making defects it does not catch

Especially true, when debugging is going to be hard

- Concurrency, real-time environment, no access to customer environment, etc.

Simplicity is key

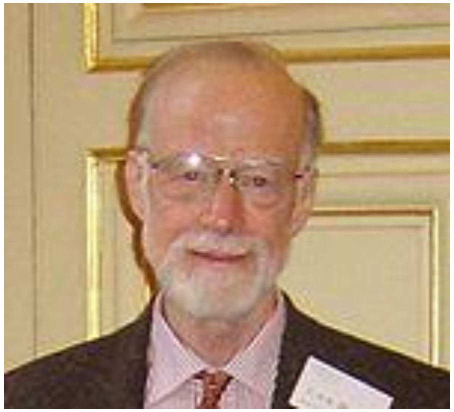
- Modularity
  - Divide program into chunks that are easy to understand
  - Use abstract data types with well-defined interfaces
  - Use defensive programming; avoid rep exposure
  - Test early, often, and compehensively
- Specification for all modules
  - Explicit, well-defined contract between each module and its clients

Strive for simplicity

"There are two ways of constructing a software design:

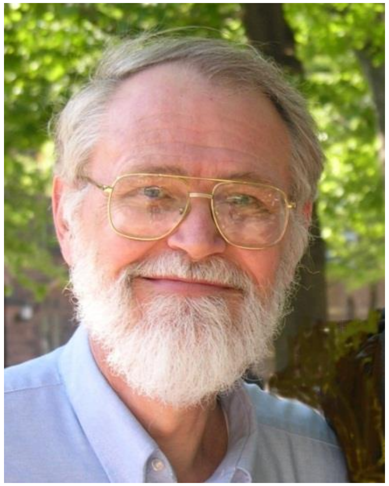
One way is to make it so simple that there are obviously no deficiencies, and
the other way is to make it so complicated that there are no obvious deficiencies.

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

Third defense: Immediate visibility

If we can't prevent errors, we can try to localize them

- Assertions: catch errors early, before they contaminate and are perhaps masked by further computation
- 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, the code you just changed is wrong or is triggering a latent defect

If you can localize problems to a single method or small module, you can often find defects simply by studying the program text

Benefits of immediate visibility

The key difficulty of debugging is to find the defect: the code fragment responsible for an observed problem

- A correct method may return an erroneous result, if there is prior corruption of representation

The earlier a problem is observed, the easier it is to fix

- Fail fast: check invariants and assertions frequently
- Don't (usually) try to recover from errors – it may just mask them

Don't hide errors

```
// precondition: k is present in a
int i = 0;
while (true) {
  if (a[i] == k) {
    break;
  }
  i++;
}
```

This code fragment searches an array **a** for a value **k**

- The value **k** is guaranteed to be in the array
- What if that guarantee is broken (by a defect)?

Temptation: make code more “robust” by not failing

Don't hide errors

```
// precondition: k is present in a
int i = 0;
while (i < a.length) {
  if (a[i] == k) {
    break;
  }
  i++;
}
```

Now the loop always terminates

- But it is no longer guaranteed that **`a[i]==k`**
- Code that relies on this will fail later
- *This makes it harder to see the link between the defect and the failure*

Don't hide errors

```
// precondition: k is present in a
int i = 0;
while (i < a.length) {
  if (a[i] == k) {
    break;
  }
  i++;
}
assert i != a.length : "key not found";
```

Assertions document and check invariants

Abort/debug program as soon as problem is detected

- Turn an error into a failure

Failure occurs only when assertion is checked

- May still be a long time after the earlier error
- "Why isn't the key in the array?"
  - Answer: due to some yet-undiscovered defect

Defect-specific checks

Defect is manifested as a failure: 1234 is in the list

Check for that specific condition

```
static void check(Integer[] a, List<Integer> index) {
  for (int i = 0; i < a.length; i++) {
    assert a[i] != 1234 : "Bad data at index " + i;
  }
}
```

A dirty trick, but it works

You can do this as a conditional breakpoint in a debugger

Checks in production code

Should you include assertions and checks in production code?

- Yes: stop program if check fails — don't want to take chance program will do something wrong
- No: may need program to keep going, maybe defect does not have such bad consequences (the failure is acceptable)
- Correct answer depends on context!

Ariane 5: overflow in unused value, exception thrown but not handled until top level, rocket crashes…

- [full story is more complicated]

Regression testing

- Whenever you find and fix a defect
  - Add a test for it
  - Re-run all your tests
- Why is this a good idea?
  - Often reintroduce old defects while fixing new ones
  - Helps to populate test suite with good tests
  - If a defect happened once, it could well happen again
- Run regression tests as frequently as you can afford to
  - Automate the process
  - Make concise test suites, with few superfluous tests

Inevitable phase: debugging

Defects happen – people are imperfect

- Industry average: ~10 defects per 1000 lines of code ("kloc")

Defects happen that are not immediately localizable

- Found during integration testing
- Or reported by user

Cost of an error increases by an order of magnitude for each lifecycle phase it passes through

1. Clarify symptom (simplify input), create test
2. Find and understand cause, create better test
3. Fix
4. Rerun all tests

The debugging process

1. Find a small, repeatable test case that produces the failure
   - Hard but worth it: clarifies the defect, gives a regression test
   - Don't proceed until you have a repeatable test
2. Narrow down location and cause
   - Loop: { study the data; hypothesize; experiment; localize; }
   - You may change the code to get more information
   - Don't proceed until you understand the root cause
3. Fix the defect
   - Is it a simple typo, or design flaw?
   - Does it occur elsewhere?
4. Add test case to regression suite
   - Is this failure fixed? Are any other new failures introduced?

Debugging and the scientific method

Debugging must be systematic

- Carefully decide what to do (avoid fruitless avenues)
- Record everything that you do (actions and results)
  - Can replicate previous work
  - Or avoid the need to do so

Iterative scientific process:

Should have explanatory power
Not just “Maybe line 3 is wrong”

Formulate a hypothesis

Design an experiment

Should investigate cause

Perform the experiment

Interpret results

Example bug report

```
// returns true iff sub is a substring of full
// (i.e. iff there exists A and 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:

1. Notice accented characters, panic about not knowing Unicode, begin unorganized web searches and inserting poorly understood library calls, …
2. Try to trace the execution of this example

Better response: simplify/clarify the symptom

Reducing *absolute* input size

Find a simple test case by divide-and-conquer

Can't 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't find "**ab**" within "**aab**"

(We saw what might cause this failure earlier in the quarter!)

Reducing *relative* input size

Find two almost-identical test inputs where one gives the correct answer and the other does not

Can't find "**`very happy`**" within

"**`I am very very happy to see you all.`**"

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

"**`I am very happy to see you all.`**"

General strategy: simplify

In general: 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)
- Often leads directly to an understanding of the cause

When not dealing with simple method calls

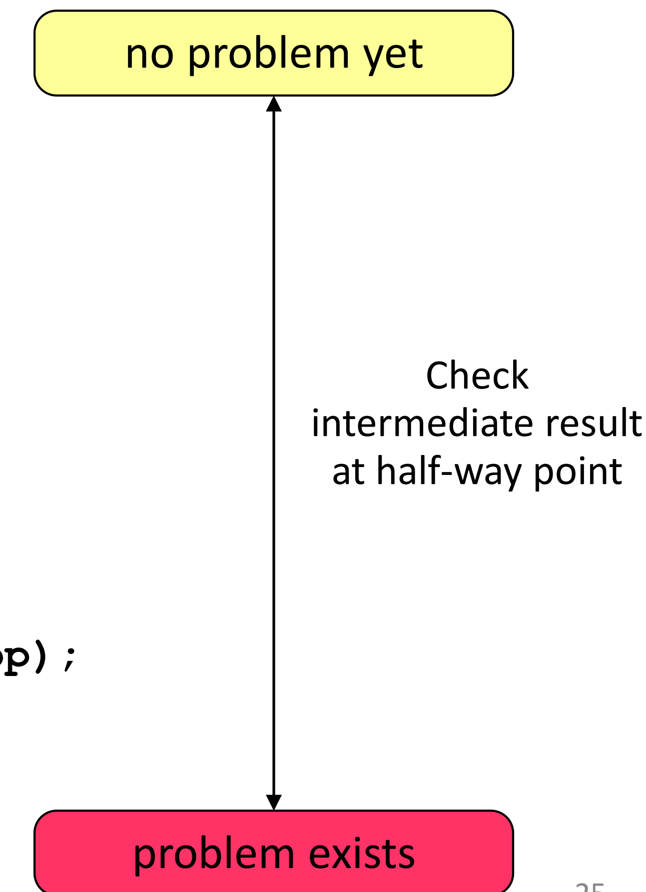
- The "test input" is the set of steps that reliably trigger the failure
- Same basic idea

Localizing a defect

- Take advantage of modularity
  - Start with everything, take away pieces until failure goes away
  - Start with nothing, add pieces back in until failure appears
- Take advantage of modular reasoning
  - Trace through program, viewing intermediate results
- Binary search speeds up the process
  - Error happens somewhere between first and last statement
  - Do binary search on that ordered set of statements

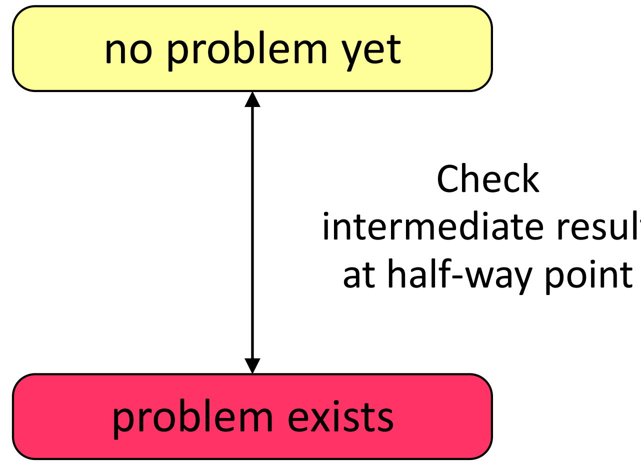
binary search on buggy code

```
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;
    }
}
```



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;
    }
}
```



Quickly home in on defect in O(log n) time by repeated subdivision

Logging

Log (record) events during execution

  Logging = tracing = printf debugging

  An alternative to using an interactive debugger

Advantages of using a debugger:

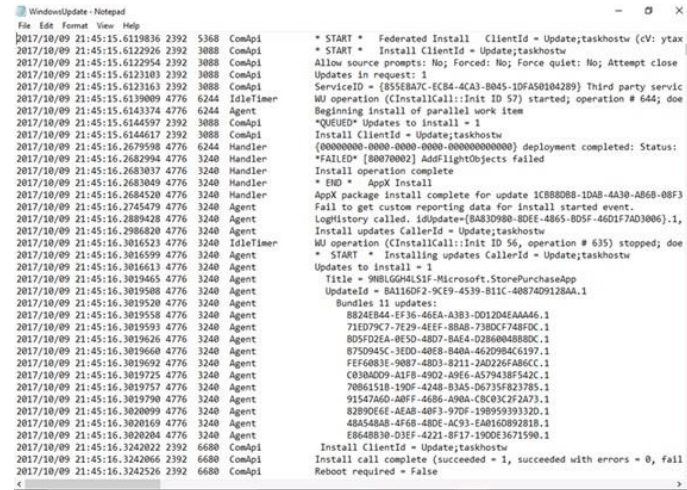
- Can examine arbitrary values
- Can change values to experiment
- Faster turnaround (vs. edit/compile/run)
- Requires no setup

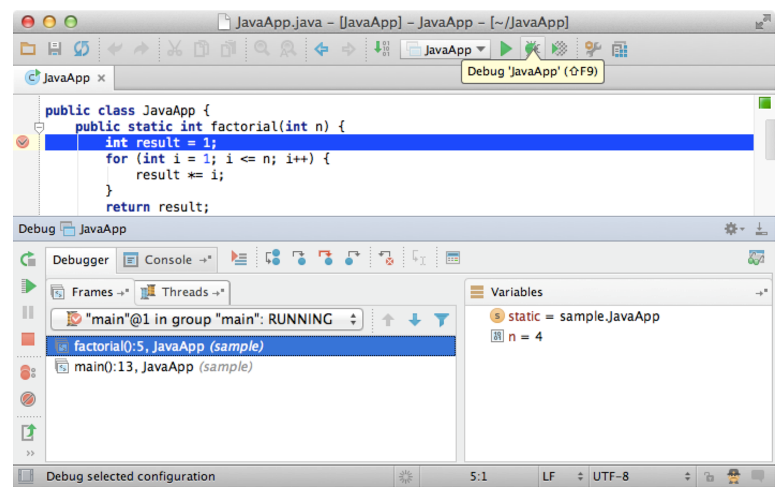
Advantages of using logging:

- Look backward in time
- Compare multiple moments
- Compare multiple executions
- Can be provided by a customer

You should be proficient at both

  Don't choose logging out of laziness





Look inside the machine

Mark Oskin was hacking on a kernel.

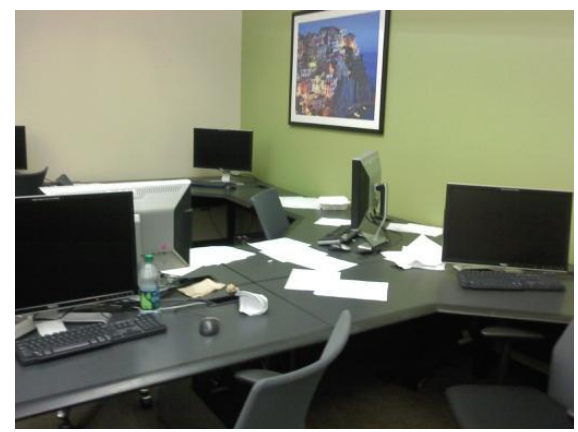
No GDB, no printf, no kprintf, …

But, did have beep from motherboard!

A	·–	J	·–––	S	···	1	·––––
B	–···	K	–·–	T	–	2	··–––
C	–·–·	L	·–··	U	··–	3	···––
D	–··	M	––	V	···–	4	····–
E	·	N	–·	W	·––	5	·····
F	··–·	O	–––	X	–··–	6	–····
G	––·	P	·––·	Y	–·––	7	––···
H	····	Q	––·–	Z	––··	8	–––··
I	··	R	·–·	0	–––––	9	––––·

Detecting bugs in the real world

- Real systems:
  - Large and complex (duh ☺)
  - Collection of modules, written by multiple people
  - Complex input
  - Many external interactions
  - Non-deterministic
- Replication can be difficult
  - No printf or debugger
  - Infrequent failure
  - Instrumentation eliminates the failure
- Errors cross abstraction barriers
- Time lag from corruption (error) to detection (failure)



Heisenbugs

In a deterministic program, failure is repeatable

In the real world, failure seems random

- Continuous input/environment changes
- Timing dependencies
- Concurrency and parallelism
- Nondeterminism
  - Random number generation
  - Hash tables are nondeterministic across runs

Hard to reproduce

- Only happens when under heavy load
- Only happens once in a while
- Use of debugger or assertions → failure goes away

Tricks for hard bugs

Rebuild system from scratch, or restart/reboot

Find the bug in your build system or persistent data structures

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

Make sure it is a bug

Program may be working correctly and you don't realize it!

And things we already know:

- Minimize input required to exhibit failure
- Add checks so the program fails fast
- Use logs to record events

Where is the defect?

- The defect is not where you think it is
  - Ask yourself where it cannot be; explain why
  - Doubt yourself, and look forward to being wrong
- Look for stupid mistakes first, e.g.,
  - Reversed order of arguments:
    `Collections.copy(src, dest);`
  - Spelling of identifiers: `int hashcode()`
    - `@Override` catches method name typos
  - Same object vs. same value: `a == b` versus `a.equals(b)`
  - Failure to set a variable
  - Deep vs. shallow copy
- Make sure that you have correct source code!
  - Obtain a fresh copy and recompile everything
  - Does a syntax error break the build? (It should!)

When the going gets tough

Reconsider assumptions

- Has the OS changed? Is there room on the hard drive? Is it a leap year? 2 full moons in a month?
- Debug the code, *not* the comments
  - Ensure the 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 a rubber duck)

Walk away

- Trade latency for efficiency – **sleep**!
- One good reason to start early

Key Concepts

- Testing and debugging are different
  - Testing reveals existence of failures
  - Debugging pinpoints location of defects
- Debugging must be a systematic process
  - Use the scientific method
- Understand the source of defects
  - To find similar ones and prevent them in the future