Debugging

is like science
in a slightly wacky universe

Image: Pill bug, Pacific Northwest National Laboratory

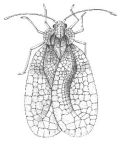
Debugging

Testing is for revealing bugs

So you succeeded: Your program failed.

Now what?

Debugging is for understanding and repairing bugs



First things first

Start with effective approach and discipline

» A skilled, disciplined debugger with no tools will beat a haphazard debugger with the best tools available

Then: supporting tools

» Observability, controllability, and automation make debugging tasks more efficient

» Differences are sometimes dramatic

Form an argument

Debugging starts from a *correctness argument*:
How your program was supposed to work, and why it *should have worked correctly*

You are looking for the flaw in the argument.

If you don't have a correctness argument ...

- Congratulations, you've found the bug. The bug is that you don't have a correctness argument.

Image: Living bug drawing, Giantbomb

The theory of your program

The correctness argument is a theory of how your program works

- Devise experiments (tests) as you would for a scientific theory: Predict something that a test can disconfirm
  - » (Dis)confirm specific parts of the correctness argument. The structure of the argument guides debugging

Possible bugs are alternative theories

- If you suspect bug X, your goal is not to fix it, but to distinguish between two theories: "My program has bug X" or "that part of my correctness argument is sound"

If you ask for help debugging ...

I will ask you ...

What is your program doing?

What should it be doing?

How is it supposed to work?

I am asking for your correctness argument, and the deviation you have observed.

Observe

We make observations of three kinds:

Assumption: I know *x* (but let's just check)

Conjecture: It seems like *y* (but is it really?)

Hypothesis: If it works like *m*,
then we should see *z*

We start with a model of how the program should work ... what should we see?

Anomaly

We expected to see something

something consistent with our mental model of how the program was supposed to work

We saw something else

an anomaly ... something that shouldn't happen if the program worked as we thought it should

- » To see the anomaly, you must know what you expected to see. The anomaly is valuable because it breaks part of your theory, and demands an explanation.

Conjecture

That was weird ...

... but it could happen if the flozzle was rerouted through the bargistator

Physicians, auto mechanics, and expert debuggers conjecture explanations based on their understanding of how systems (bodies, cars, programs) work. Scientists too.

Image: Mike Libby, Insect Lab Studio (http://insectlabstudio.com)

Hypothesis Testing

The new windshield was brilliant, but within a few days it was foggy again. The haze is on the inside, ...

TOM: The other possibility is that your heater core is leaking. **If the heater core has a hole in it, the haze on your windshield could be a thin film of coolant.**

RAY: If it's coolant, it would have certain characteristics. It would be greasy to the touch. It would smell sweet. And it would likely be thicker at the bottom of the windshield, near where the vents are.

TOM: So try a couple of experiments. Try removing the seat covers for a week, and see what happens. If that fixes it, your son may need to go commando -- without his seat covers.

Car Talk, May 2009

When my bicycle makes a noise ...

I listen ...

- Does it happen only when I'm pedaling?
- Does it happen only when I'm moving?
- Only in hills? Only in the big ring?
- Only when I'm seated?
- Faster and slower depending on my speed?
- Faster and slower depending on cadence?

These are observations that can confirm some causes and and eliminate others.

Step by step

Make observations to confirm correct execution

based on your correctness argument

Observe anomalies

Conjecture possible causes

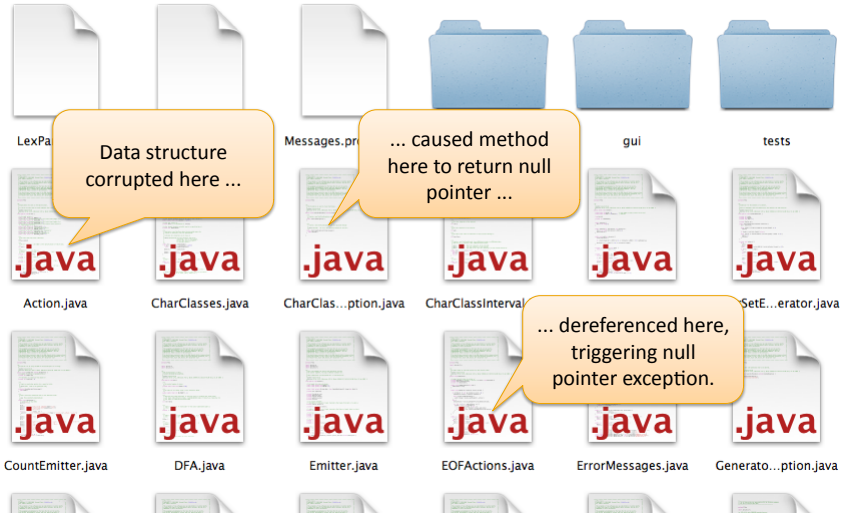
Develop conjectures into hypotheses

observe to (dis)confirm your theory

When stuck, check assumptions

Observability

We see a failure that may be far away (in time and space) from the error that caused it. We need to observe the chain of events.



Observing... low tech approach

Insert print statements

- Make them self-identifying
- Consider what you expect, so anomalies are obvious

Avoid "sipping from the firehose"

- Print just enough to confirm or contradict your conjecture *(you have one, right?)*
- Reduce, reduce, reduce the test case

assert cool && useful: whynot

Many languages have an assertion facility

In java:

```
assert p : m
```

which is like

```
if (! p) { throw new AssertionError(m); }
```

Document your correctness argument with assertions to make mistakes observable

Observing ... high tech approach

Development tools like Eclipse include debugging support

- **Breakpoints** (stop here, look around)
- **Watchpoints** (like temporary print statements)
- **Single-stepping**
- **State inspection** (including activation stack)

Very useful ... with a systematic approach

- A good cook sharpens his knives, but sharp knives won't make you a good cook

Challenges to Observability

Concurrency (threading)

- Schedule decisions (context switches) are both invisible and uncontrolled. Failure can appear random.
- Approaches
  - » Observe invariants, not context switches
  - » Force context switches (to trigger more failures)
  - » High-tech: control context switches

Distribution, host-target development

- The problems of threading, plus real concurrency, plus unexpected differences between development and deployment environments ... hard even with good tools

Controllability

def: Ability to perform the experiment you want

Tools: single-step, "programming at the break", calling methods in isolation, watchpoints ...

Also depends on architectural design of system

- Good interfaces provide control for debugging
- Large systems often invest heavily in test & debug infrastructure as part of design and implementation
- Example: database dump/restore in editable format

Can the debugging process be automated?

Some attempts to narrow the causes of failure:

Systematically reduce test input

Systematically reduce set of recent program changes

... (binary searches to narrow causes)

Not widely used yet ... but maybe soon?

See: Andreas Zeller, *Why Programs Fail*

http://www.whyprogramsfail.com/

Anti-patterns of debugging

Random program changes

- That is "programming by magic" ... and it turns a slightly broken program into a total wreck
- But changes to observe effects can be useful, as well as replacing mystery code by something simpler

Massive data dump

- Useful information is lost in the flood. Better to print selectively, with useful abstractions

Jumping to conclusions

- Don't skip the experiment: Predict a behavior and observe it

Make debugging easier

Build and test incrementally

- Makes bugs smaller and more local

Trim failing test cases

- Make observation and diagnosis easier

Program defensively

- Make your programs crash grandly every time, instead of mostly working most of the time
- Make your program crash as soon as its wrong, not later. assert(...) is an excellent tool for this.

Learn from every bug

What did I do wrong?

How could I have prevented it?

What can I look for to identify this kind of bug?

- Example: Corrupting data structures, failing to lock a shared structure (in threaded code), accepting mal-formed input, ...

Build your mental database of bug patterns and build your systematic debugging skills

Image: Monty Python's Flying Circus Bicycle Repair Person

Debugging is a key skill

As long as programs are written by people, debugging will be part of programming

It's worth learning to do it well.

The basic methods are used across disciplines

- In medicine: Disease diagnosis
- In auto repair: Fault diagnosis
- In science: Theory building
- In teaching: Explanation (broken model diagnosis)

If you learn to debug well, you'll use it in surprising ways.