DEBUGGING
RULES!

Understand the system
Make it fail
Quit thinking and look
Divide and conquer
Change one thing at a time
Keep an audit trail
Check the plug
Get a fresh view
If you didn't fix it, it ain't fixed

From Debugging © 2002 by David Agans
To get the book or download this free poster, go to
www.debuggingrules.com
1. Understand the system (summary)

- The foundational assumption in programming: the computer is only doing what it's told.

The most important debugging tool is your mind.

Bugs aren't a waste of time. They're an opportunity to grow your understanding towards mastery.

Don't thrash. Be the tortoise. Slow and steady wins the race. Be methodical. Going slow is the fastest way to go.

Go backwards if you need to. Review the foundations. Read a manual or book.
2. Make it fail

The idea

- When you have a bug, figure out how to trigger it reliably.

Why?

1. So you can look at it.
2. So you can focus on the cause.
3. So you can tell when you've fixed it.

this is especially important if the bug is intermittent: if you can't trigger the bug reliably, it's difficult to focus on finding the cause
2. Make it fail

Story time

Story #1: suspicious car noise
2. Make it fail

Practicalities

- A test case is a great bug trigger!

Unit tests and relatively isolated tests already have a narrow scope.

Many bugs uncovered by such test cases don't need a particularly methodical approach for this reason.

Failing that, find a set of inputs that make it fail, and run the program

You can use the command-line or the REPL for this

Trigger it once, then trigger it again, to be sure the first trigger didn't change the environment in a significant way.
3. Quit thinking and look

The idea

- You already see the *result* of the failure. But you need to find the *cause*.

Looking means cracking open the black box and tracing what's actually happening.

Looking is *hard*, and sometimes tedious. There's some emotional resistance. But we have to look to figure it out.

Story: the sleepy server
3. Quit thinking and look

Instrumentation in daily life

Are you feeling sick? Use a thermometer.

Temperature and pressure gauges in mechanical systems

"To find air leaks in a house, we dangle a ribbon around the window frames and electrical outlets and look for drafts."

"When you have a leak in a bicycle tire, you swab soapy water on it and look for the bubbles."

"Natural gas has a rotten egg odor added to it for the sole purpose of making it detectable when it's leaking."
4. Divide and conquer

The idea

This is the only rule that involves actually finding the problem.

"All of the [other rules] are just to help you follow this one. This rule is at the heart of debugging."

Example: find an item X in a sorted array with binary search. Divide the problem in half at each step. You can search a very big array quite quickly.

Start with the bad result, and follow the bad result. There are plenty of working parts of the system; don't debug those.
4. Divide and conquer

D&C in Clojure (1)

If there's a bug somewhere in:

```
(wr ite-tree (make-tree-object ".")
```

Then the bug is either in:

. `make-tree-object`, or

1. `write-tree`
4. Divide and conquer

D&C in Clojure (2)

If there's a bug somewhere in:

```
(make-commit arg1 arg2 arg3)
```

Then the bug is either in:

. `make-commit`, or

. How you're calling it
4. Divide and conquer

D&C in Clojure (3)

If there's a bug somewhere in:

```
(-> value fn1 fn2)
```

Then the bug is either in:

1. \texttt{fn1} (or how you're calling it), or
2. \texttt{fn2} (or how you're calling it)
4. Divide and conquer

D&C in Clojure (4)

If there's a bug somewhere in:

```
(if (test value)
  (then value)
  (else value))
```

Then the bug is either in:

1. **test** (or how you're calling it),
2. **then** (or how you're calling it), or
3. **else** (or how you're calling it).
4. Divide and conquer

git bisect

Git has a binary search feature called 'bisect'.

Give it a good commit (when things worked) and a bad commit (when things didn't).

It finds a commit in between and asks whether it's bad or good, and recurs from there, binary search style.
5. Change one thing at a time

The idea

Change one thing, and get feedback. Did you still trigger the bug?

This is why automation in the "Make it fail" step is helpful: you might have to make 100 tiny changes before isolating the cause.

This is applied science. Control your variables.
6. Keep an audit trail

The idea

Sometimes it's the most insignificant-seeming thing that's actually the key to making a bug happen.

What seems obvious to one person might be missed by another, so include plenty of detail.

The person debugging needs data to correlate events with the bug.

You think you'll remember those details, but you won't. :-)

Those audit trails might be useful months later, when you encounter a similar bug.

To be honest, this is a big gun that I don't bring out very often. But it's good to have in
6. Keep an audit trail

Story time

Story: the failing video streaming system
6. Keep an audit trail

Practicalities

The way I do this is to categorize each thing I write down in my audit log.

**FACTs** are observations about preconditions or events, as precisely and thoroughly as I can capture them.

**THEORYs** are an idea that I have about the cause, or about what part of the code it might be hiding in.

**TESTs** are a test I design to prove or disprove the theory. The I run the test and note the **FACT** of the result, which might lead me to a new **THEORY**.

If I'm stuck, I can review my log for inspiration for new theories, or to see if anything may have fallen through the cracks. If a test doesn't do what I thought it would, then I may need to test it differently.
7. Check the plug

The idea

*Convictions are more dangerous enemies of truth than lies.* - Friedrich Nietzsche

The key idea: if you're still stuck, question your assumptions.

Some part of the system you were relying on to work isn't actually working correctly.

When you start debugging, you should question your own code before the library code that you depend on.

But if you can't find a bug in your code, it might be the library, which, after all, was written by another fallible human.

Maybe the tool you're using isn't working, or isn't telling you what you thought it was.
8. Get a fresh view

The idea

If you can find an expert, they might be able to point you in the right direction and save you hours of debugging time, or even longer reading a manual or book. (Story: brake light short)
9. If you didn't fix it, it ain't fixed

The idea

Be suspicious if the problem just goes away. It'll probably come back later.

Maybe some aspect of the environment changed at the same time you tried a fix. So undo your fix, see it fail, redo your fix, and see it work, and then you can be confident it was really your fix that fixed it.
Understand the system
Make it fail
Quit thinking and look
Divide and conquer
Change one thing at a time
Keep an audit trail
Check the plug
Get a fresh view
If you didn’t fix it, it ain’t fixed
Live demo of debugging principles
Reflection poll (go to pollev.com/jeffterrell)

DEBUGGING
RULES!

Understand the system
Make it fail
Quit thinking and look
Divide and conquer
Change one thing at a time
Keep an audit trail
Check the plug
Get a fresh view
If you didn’t fix it, it ain’t fixed

From Debugging © 2002 by David Agans