



Debugging

The why, where && WTF
behind your error messages...

Topics Covered

- History
- Reading Error messages
- Debugging Workflows
- Tools
- Cleanup some code (maybe)
- Ten most common errors with examples

Why you care

- 80% of your time as a Dev is spent READING CODE
 - 60% of that time is DEBUGGING
 - Good Debugging skill is one of the things that will keep you sane in this job...
- * All times estimated... ☺



History

Photo # NH 96566-KN (Color) First Computer "Bug", 1947

92

9/9

0800 Antran started
1000 .. stopped - antran ✓
13" uc (032) MP - MC { 1.2700 9.037 847 025
 9.037 846 995 correct
(033) PRO 2 2.130476415
 2.130676415

Relays 6-2 in 033 failed special speed test
in relay .. 10.00 test.

Relay
2145
Relay 3371

1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.

1545



Relay #70 Panel F
(moth) in relay.

1630 Antran started.
1700 closed down.

First actual case of bug being found.

Definitions:



BUG: Whenever a program/system is not behaving the way we expect.

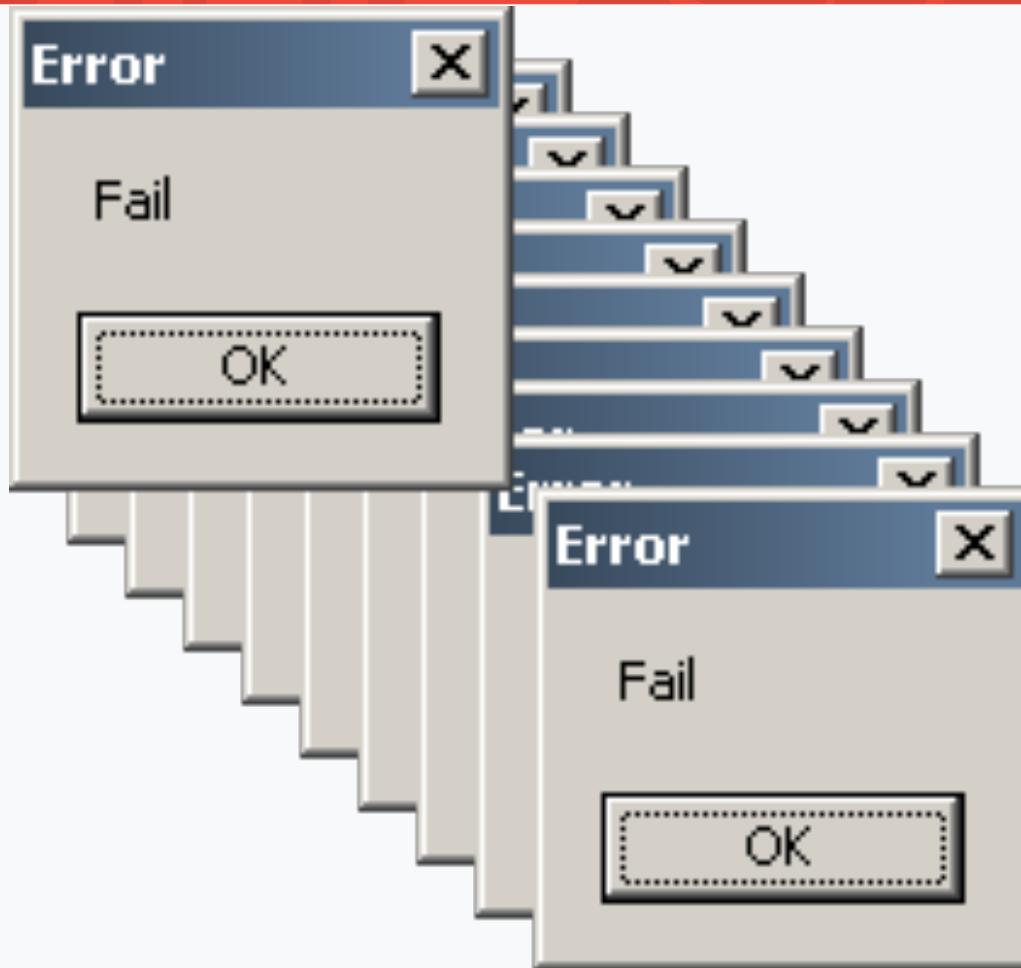
- **DEBUGGING**: Any systematic approach undertaken in the understanding and elimination of a BUG.



Gen Knowledge

- Debugging is the process of figuring out the source of the error and fixing it.
- Think of it as the disconnect between your assumptions and what the code is actually doing.
- It's a skill, so you'll need to practice it.
- Pro Tip: Helping peers is a super great if not THE way to hone this skill.

Error Messages



Stack Trace

Stack Trace (aka stack backtrace or stack traceback) is a report of the active stack frames upon error condition during a program's execution.

We follow this path back to find the offending code segment and correct it.

Ruby Exception Classes

Exception

- fatal *used internally by Ruby*
- NoMemoryError
- ScriptError
 - LoadError
 - NotImplementedError
 - SyntaxError
- SecurityError
- SignalException
 - Interrupt
- StandardError
 - ArgumentError
 - FiberError
 - IndexError
 - KeyError
 - StopIteration
 - IOError
 - EOFError
 - LocalJumpError
 - NameError
 - NoMethodError
 - RangeError
 - FloatDomainError
 - RegexpError
 - RuntimeError
 - SystemCallError
 - *system-dependent exceptions (Errno::xxx)*
 - ThreadError
 - TypeError
 - ZeroDivisionError
- SystemExit
- SystemStackError

Rules For Debugging

The First rule of debugging:

Rules For Debugging

The First rule of debugging:

- Read the error message!

Rules For Debugging

The First rule of debugging:

- Read the error message!

The Second rule of debugging:

Rules For Debugging

The First rule of debugging:

- Read the error message!

The Second rule of debugging:

- Read the error message!

Rules For Debugging

The Third rule of debugging:

- No matter how fucked you think you are....

BE COOL

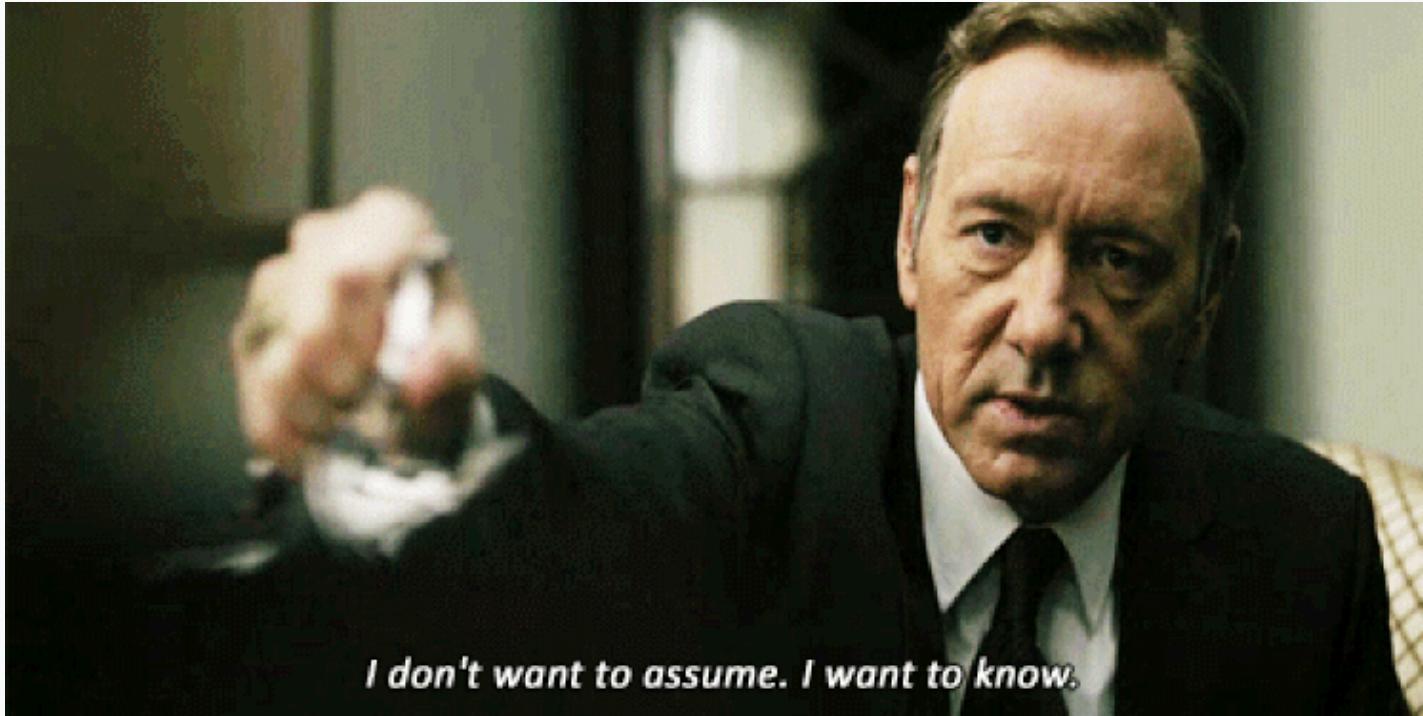
&&

Keep Working
The Problem



Rules For Debugging

The Fourth rule of debugging:



SHITCAN YOUR ASSUMPTIONS

Errors are your friends!

Don't freak when you see an error

Analyze the error:

- What are the line number(s)?
- What type of error is it?
- What specific message did you get?

Debugging Workflow

- Read your error messages!
- SPECIFICS: type, line #, error msg & filename
- Know what your expected behavior is.
- Shitcan your assumptions.
- Verify and understand the input(s).
- Verify the program / variable state.
- Make small incremental changes.
- Roll back the code little by little until you get back to a ‘functional state’.

Strategies

- Debug “inline”
- Use the REPL (irb | | pry)
- Reverse Engineer
- Guess && Check

Debugging “inline”

- Use p statements to quickly show variable's value

p “some_variable: #{@some_variable}”
- Quickly determine if you are reaching a method
- def some_method

p “HIT: some_method”

end

Debugging “inline”

- Terrific way to determine flow control around conditional statements

```
if num = 1
    p "inside if"
else
    p "inside else"
end
```

Debugging “inline”

- Use “signaling code” to easily flag your spot in conjunction with a variable check

```
p “~” * 80
```

```
p “var: #{var}”
```

```
p “~” * 80
```

Tip: Use different signal characters when looking at multiple points in your code.

- This is very useful in larger applications or when p-ing to a busy server console.

Reverse Engineer

Working the problem backwards

First step here is restoring functionality

Tip: Use different signal characters when looking at multiple points in your code.

Debug via BINARY SEARCH

Comment whole swaths of code when you are dealing with a frustrating error.

Did that make the problem go away?

Great!

Uncomment half the code, continue until the error returns.

You should be able to narrow down the exact line that was causing your problem.

Tools



awesome_print

```
gem install awesome_print
```

```
require 'awesome_print'  
ap some_array  
ap some_hash
```

PRY – The IRB Alternative

Pry is a REPL (Read-Eval-Print-Loop) much like IRB but with some additional key features:

- Syntax Highlighting
- Built in methods
- A Debugger
- Tabbed completion

PRY – Install

gem install pry

gem install pry-doc

gem install pry-byebug

rbenv rehash

PRY – terminal commands

ls (list methods)

_ (the last output)

? (show-doc)

. (send command to bash)

cat filename (displays the given file)

wtf? (wtf.....)

pry-byebug

```
gem install pry-byebug
```

```
require "pry-byebug" # in the top of the file
```

To stop execution and enter the REPL insert a
binding.pry

This triggers a breakpoint at the spot you wish to
examine with everything currently in scope.

pry-byebug commands

!!!: Quit byebug session and resume, use continue if this doesn't work

step: Step execution into the next line or method. Takes an optional numeric argument to step multiple times.

next: Step over to the next line within the same frame. Also takes an optional numeric argument to step multiple lines.

finish: Execute until current stack frame returns.

continue: Continue program execution and end the Pry session.

up: Moves the stack frame up. Takes an optional numeric argument to move multiple frames.

down: Moves the stack frame down. Takes an optional numeric argument to move multiple frames.

PRY#show-doc

```
[7] pry(main)> show-doc Array#each\_with\_index
```

From: enum.c (C Method):

Owner: Enumerable

Visibility: public

Signature: each_with_index(*arg1)

Number of lines: 11

Calls **block** with two arguments, the item and its index, for each item in **enum**. Given arguments are passed through to **#each()**.

If no block is given, an enumerator is returned instead.

```
hash = Hash.new
%w(cat dog wombat).each_with_index { |item, index|
  hash[item] = index
}
hash  #=> {"cat"=>0, "dog"=>1, "wombat"=>2}
[8] pry(main)> []
```

Exceptions

- An instance of the Exception class
- A raised exception will propagate through each method in the call stack until it is stopped or reaches the point where the program started
- Exceptions can be **Raised && Rescued**

Debugging Wrapup

Questions?

Sweet Links

Pry Usage (youtube)

Replace IRB with PRY

Final Thought

Don't like debugging...

A Strong test suite
will greatly reduce the
amount of debugging
you do boo be do.

What now...

