Joe McMahon (mcmahon@cpan.org)

Intro to Automated Testing

Philosophy, tools, and practice

CAUTION CAUTION CAUTION

Opinion zone!

CAUTION CAUTION CAUTION

Toyota on-board software

- Source: http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-“spaghetti”-code

- September 2007 unintended acceleration event
- Exiting highway
  - Brakes would not stop the car
  - Parking brake did not stop the car (150-foot skid mark)
  - Car collided with an embankment at full speed
  - Passenger killed; driver spent 5 months recovering

"There are a large number of functions that are overly complex…some of them are untestable, meaning that it is so complicated a recipe that there is no way to develop a reliable test suite or test methodology to test all the possible things that can happen in it. Some of them are even so complex that they are what is called unmaintainable…It is possible for a large percentage of the failsafes to be disabled at the same time that the throttle control is lost."

–Michael Barr, embedded software specialist

Other problems in the Toyota code

- Coding standards violations
  - 81,514 violations
- Global variables
  - 10,000 global variables
- No code reviews
- Failure to detect task death
- “Kitchen-sink” task (big ball of mud)
- Discarding exceptions
- Watchdog supervisor doesn’t monitor tasks, monitors CPU load
  - And it does that wrong

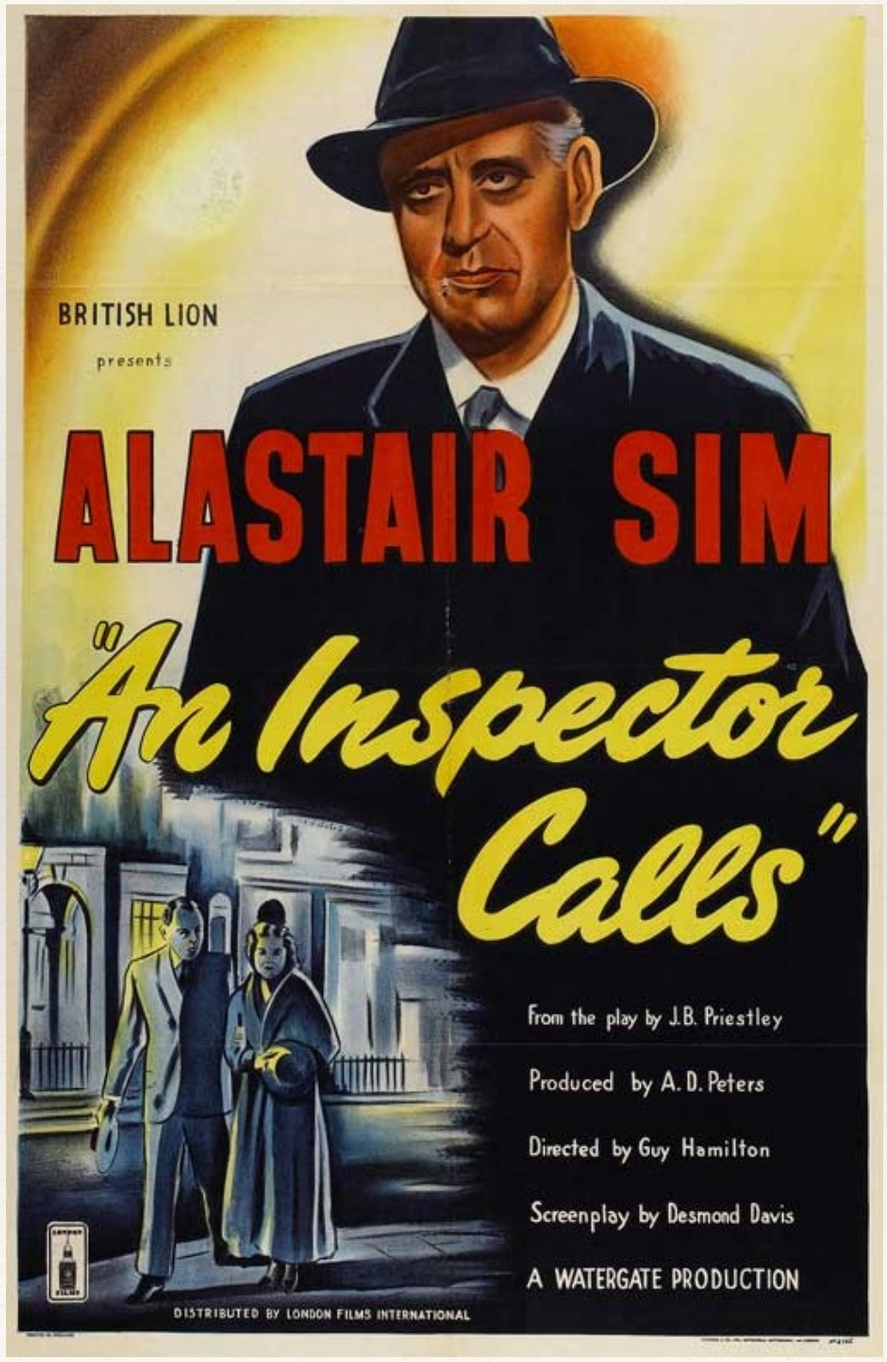
CAUTION CAUTION CAUTION

Many of these problems could have been caught by better testing and tighter standards

CAUTION CAUTION CAUTION

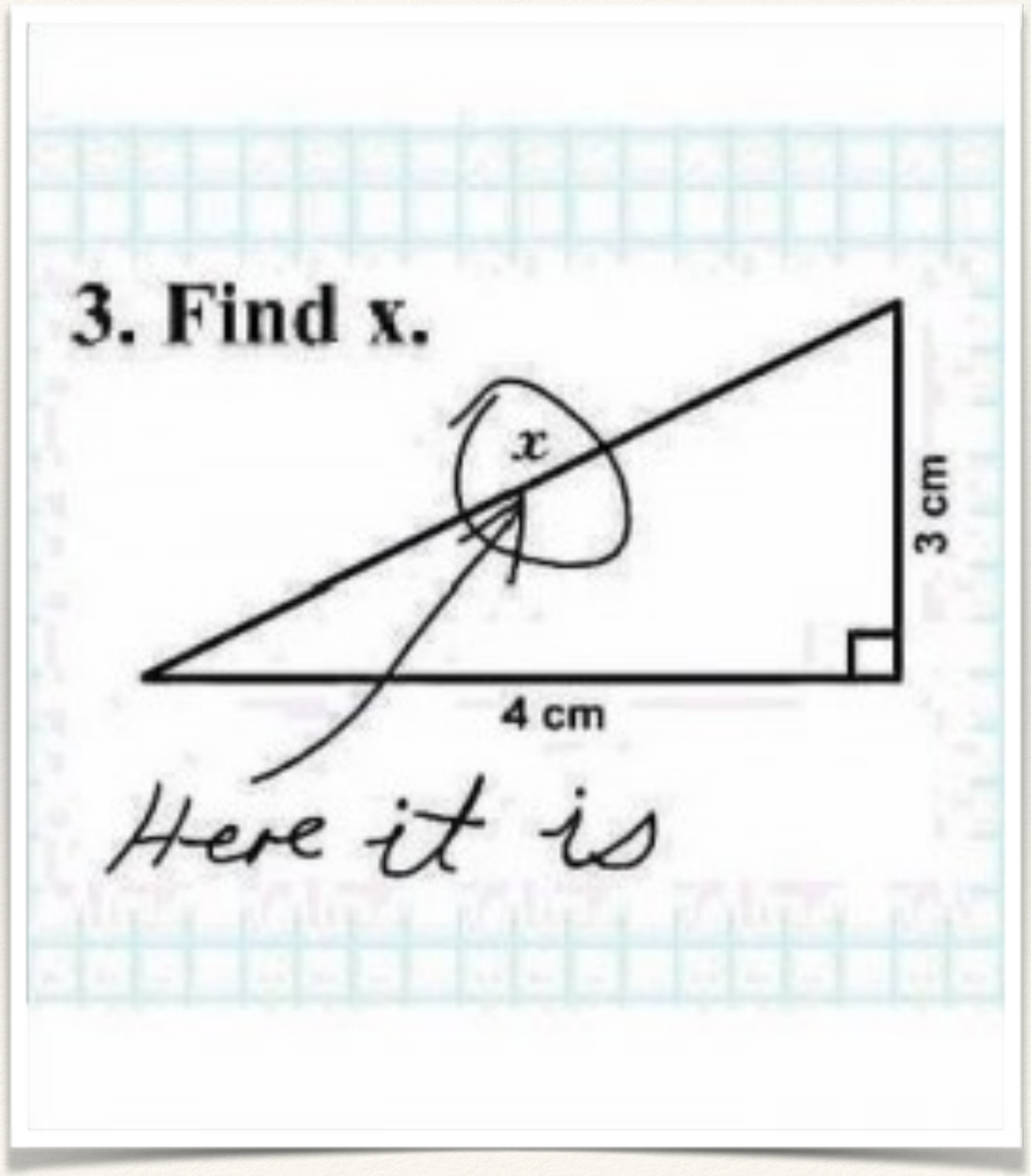
Static tests

- Look at the code itself and point out possible problems
- Textual analysis, possibly simulated execution
- Generally only require source access and are stateless
- No dependencies
- Running them has no effect since they only look and don't touch



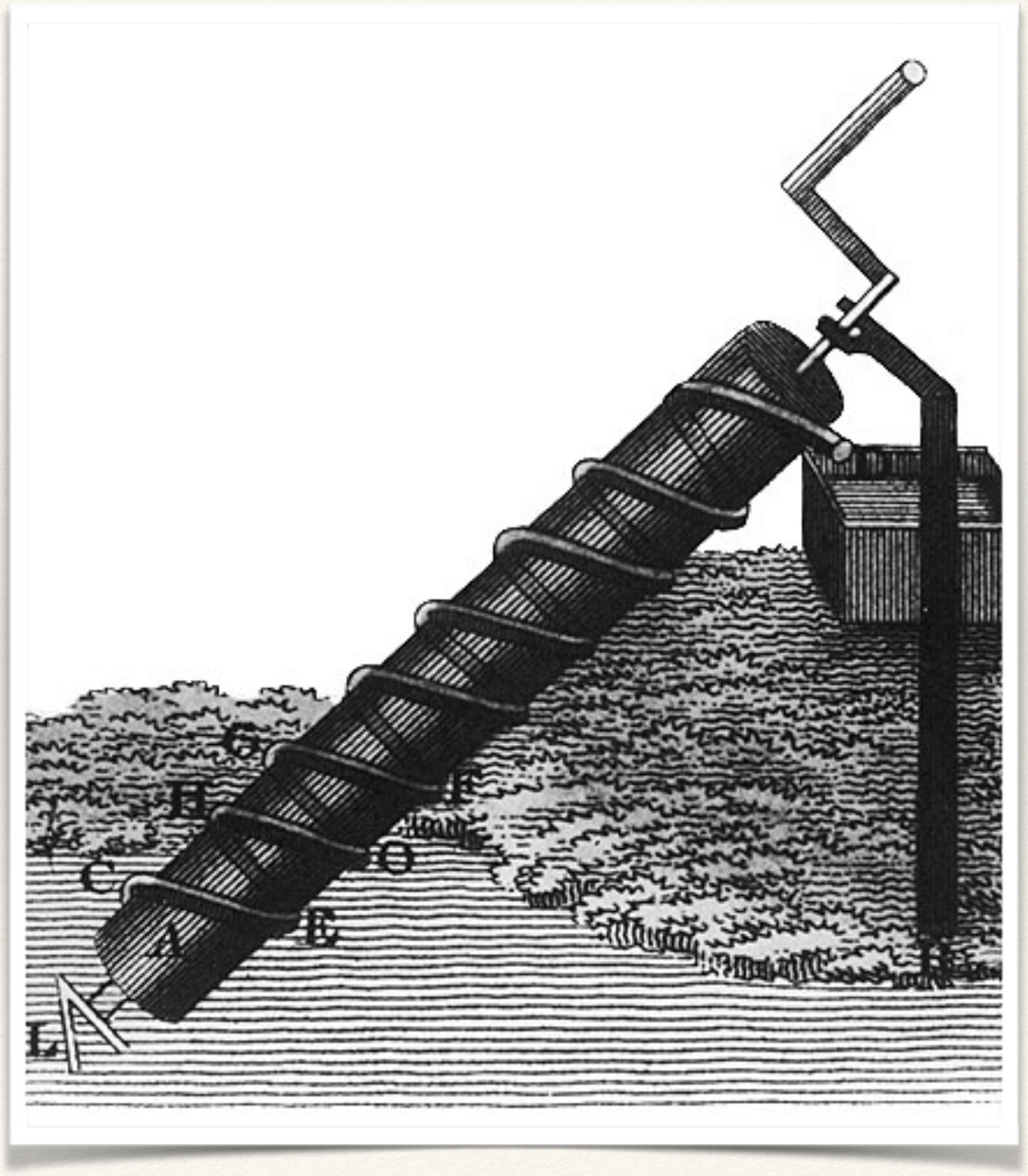
Unit tests

- Generally simplest
- Test one piece of code
- No real external dependencies
- Generally need very little setup
- Usually are stateless, or state is sufficiently hidden inside an object as to be unshared



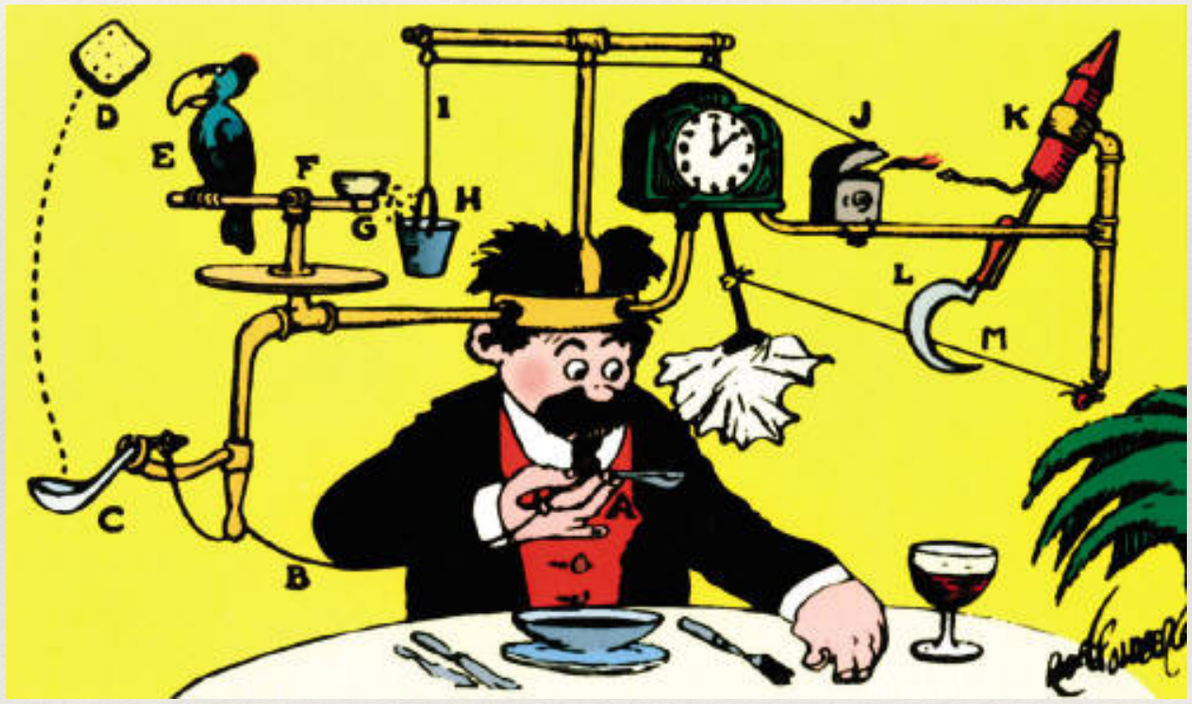
Functional tests

- Less simple
- Test a blob of code made up of units
- Parts are interdependent but whole thing stands alone
- Parts share a common state; may share external state
- Setup may or may not be needed



Integration tests

- Most complex
- Tests interactions of a bunch of functions
- Depends on externals (network servers, databases, etc.)
- Generally are stateful
- Can’t run simultaneously because of shared external state
- Often slower because of external setup time



CAUTION CAUTION CAUTION

Always be aware of which kind of test you're using

CAUTION CAUTION CAUTION

Important safety tip

Don’t cross the streams

Only one kind of testing per test program!

How does one test well?

The elementary particles of testing

- Truth
- Equality

Representing test results

- How many tests did we run?
- Did this test pass or fail?
- Is there anything else we should know about this?

TAP

1..4
ok 1 - module loads
ok 2 - new() produces an object
not ok 3 - object is expected type
Expected ‘ARRAY’, got ‘HASH’.
ok 4 - all expected methods present

TAP

1..4
ok 1 - module loads
ok 2 - new() produces an object
not ok 3 - object is expected type
Expected ‘ARRAY’, got ‘HASH’.
ok 4 - all expected methods present

TAP

1..4
ok 1 - module loads
ok 2 - new() produces an object
not ok 3 - object is expected type
Expected ‘ARRAY’, got ‘HASH’.
ok 4 - all expected methods present

TAP

1..4
ok 1 - module loads
ok 2 - new() produces an object
not ok 3 - object is expected type
Expected ‘ARRAY’, got ‘HASH’.
ok 4 - all expected methods present

TAP

1..4
ok 1 - module loads
ok 2 - new() produces an object
not ok 3 - object is expected type
Expected ‘ARRAY’, got ‘HASH’.
ok 4 - all expected methods present

TAP producers and consumers

- Test classes provide testing subs/methods
- Testing subs/methods actually test things and produce TAP
- Test harness monitors tests, consumes and summarizes TAP

Test::Simple

- Provides ok()
- Can only test for true or false, nothing else

ok() - things to remember

- Any value other than undef, ‘’, 0, or ‘0’ is true (passes).
- Tests run in scalar context
  - ok(@array, ‘succeeds if there are elements’);

Test::Simple demo

Test::More

is() and isnt()

- compares two things in scalar context
- uses string comparison
- passes if they are equal, fails if they are not

```
is( 1, 1, “1 is the same as itself”);
isnt(“foo”, “bar”, “metasyntax not the same”);

ok 1 - 1 is the same as itself
ok 2 - metsyntax not the same
```

like() and unlike()

- regex match true (like) or false (unlike)

```
like($line_num, qr/\d+/,
     “$line_num is a number”);

unlike($line, qr/\s+/, “is nonblank”);
```

cmp_ok()

- Compares using whatever operator you want
- `cmp_ok($foo, '==', $bar, "foo numerically equals bar");`
- same as
  - `ok($foo == $bar, "foo numerically equals bar");`
    - but shows you values if the comparison fails

```
cmp_ok($foo, ‘==‘, $bar, “same counts”);


not ok 1
# Failed test in blech.t at line 232.
#      ’27’
#             ==
#      ’25’
```

Deep structure checks

```
is_deeply($ref1, $ref2, “identical structures”);
```

OO-related tests

```
can_ok($module_or_object, @methods);

isa_ok($module_or_object, $class);
```

Test control

```
pass(“this is always ‘ok’”);

fail(“this is always ‘not ok’”);

BAIL_OUT(‘this stops the test immediately’);
```

note() and diag()

- tell us something without affecting the test

```
diag(“arbitrary message, always shows”);

note(“only shows with verbose on”);
```

- Prints:

```
# arbitrary message, always shows

# only shows with verbose on
```

- Use `note()` for debug, `diag()` for error diagnosis

SKIP

```
SKIP: {
    skip “because of reasons”, $number of tests
        if $some_condition_is_met;
    ok($blah, ‘some test’);
    …
}
```

TODO

```
TODO: {
   local $TODO = “these tests fail for this reason”;
   ok($will_fail,
      “fails but doesn’t cause test fail”);
   …
}
```

The plan

- The numbers at the top (or the bottom) of your tests

```
1..20
```

- How many tests you *plan* to run
- Test fails unless exactly this many run

Planning first

```
use Test::More tests => 20;
```

```
use Test::More;

plan tests => keys %tests * 4;
```

```
use Test::More skip_all => “unconditionally skip”;


use Test::More;
if (environment_check()) {
   plan skip_all => ‘bad environment’;
}
else {
   plan tests => keys %check + 8;
}
```

done_testing()

- Prints a test plan at the end of the test
- Useful if you aren't sure of your test count
- Plans as many tests as you actually ran
- Pro:
  - makes writing tests simpler and faster
  - you don't fail because you counted wrong
- Con:
  - if the number of tests changes, you don't find out

Test::More demo

prove

- Runs the tests and summarizes them
- Can run tests in a stateful manner

`prove --state=save t`

- Runs all the tests in `t/`
- Saves whether or not they passed or failed
- Prints a summary

Rerunning just the failed tests

```
prove -r --state=failed,save t
```

Run most-recently-failing tests first

```
prove -r --state=hot,new,save t
```

Parallel testing, slowest first

```
prove -r -j 9 --state=slow,save t
```

prove -j will fail for anything that depends on shared state.

Divide your tests up so you can use -j

prove demo

Test::Exception

```
throws_ok { code block } qr/something in \$@/, ‘dies
with expected message’;


dies_ok { code block } “dies, we don’t care why”;

like $@, qr/but we can check/, ‘validate reason’;


lives_ok { code block } “lives”;


lives_and { is $foo->thing, ‘value’ }, “lives and passes”;
```

Test::Differences

```
use Test::Differences;

eq_or_diff $string1, $string2, ‘string diff’;

eq_or_diff $structure1, $structure2;
```

Diff-able structures

- scalars composed of record-per-line (e.g. read a file)
- arrays of scalars
- arrays of arrays of scalars
- arrays of hashes containing only scalars

- else compares Data::Dumper versions of structures

Test::Exception & Test::Differences demo

Trapping output

- We often see stuff like this:

```
prove x.t

1..2

ok 1 - module loads

(Many lines of blithering by the module...)

Subroutine “do_something_expensive” redefined at
line 17 of x.t.

ok 2 - methods work
```

CAUTION CAUTION CAUTION

Unexpected output is a test failure

CAUTION CAUTION CAUTION

Why?

- It’s not related to the task at hand - the test
- It can hide a real error message or warning you wanted to see
- We don’t know if the right output or the wrong output was created because we never test it
- We don’t know if
  - …that was supposed to happen but we were sloppy
  - …that shouldn’t have happened at all

Assume unpredicted output means there's a bug

Trapping output

- Warnings we didn't need to see at all (artifacts from testing)
- Warnings from the code under test
- Output from the code under test

Eliminating redefinition warnings

```
{
    no warnings ‘redefine’;
    *Code::Under::Test::replaced = sub {
        ...
    };
}
```

- Makes this block not issue the warning
- OK because
  - scope is limited
  - you’re explicitly calling out the decision for no warning

Uninitialized variable warnings are not warnings – they are errors.

Causes

- Uninitialized variable warnings mean either:
  - You were sloppy about ensuring proper default values - sloppy is wrong
  - You did something that resulted in an undef, but didn't check that case - not checking returns is wrong
  - Code *calling* your code ("uninitialized value in subroutine invocation") was sloppy in the same way - not checking arguments is wrong
  - By implication, something was not thought out completely or well enough, or the implementation did not check for the possible acceptable values - not thinking through your code is *really* wrong

Fixing uninitialized variable warnings

- Verify that the undefined value is all right in this context
  - E.g., pattern match was done, but no matches at all is a perfectly acceptable result in this context
  - If you might be passed or generate an undef, checking for it and bypassing the code that would throw the warning is acceptable – as long as the bypass doesn't generate *more* undefs.
- Use defined-or to fix up the `undef`:
  - `$complainer //= $good_default_value;`
- If an undefined value is acceptable in the expression that's throwing the warning and *you know this for sure,* then a `{ no warnings 'uninitialized' }` block *and a comment saying why* is OK.

Catching and checking warnings

```
use Test::Warnings qw(:all);


sub warns {
    warn “incoherent screaming”;
}


my @warnings = warnings { warns() };
is @warnings, 1, ‘got one warning as expected’;
like $warnings[0], qr/screaming/, ‘the one we wanted’;
```

Catching STDOUT or STDERR

```
use Test::Output;

stdout_is(
   { print “yes” }, ‘yes’,
   ‘expected output’);

stdout_like { call_something },
   qr/$expected_output_matcher/;
```

Cleanup demo

Static Analysis

- What you use depends on your language
  - Ruby: rubocop
  - Python: pylint
  - Perl: Perl::Critic and Perl::Tidy

Perl::Critic

- Textual analysis of your Perl code
- Finds common problematic constructs
- Pluggable architecture, so new “criticisms” can be added
- No simulated execution (Perl is hard)

Forcing people to use it

- Test::Perl::Critic
  - Runs Perl::Critic as a part of your test suite
  - Configure it to stop what you want, allow what you want
  - Forces programmers to address known-bad, known-unsafe usage
  - Adding your own plugins can check other stuff (e.g., don't use Apache classes in the DBI layer)

Test::Perl::Critic demo

Questions?

Thank you!

BONUS: Cucumber

Translating tests to code is hard

- You need to
  - Figure out the requirement
  - Figure out the inputs
  - Figure out the outputs
  - Validate both

Translating code to desired behavior is hard

- Read the code
- Actually understand it
- “Head-check” it’s testing the right things

What if we could write natural(ish) language instead?

```
Feature: Adding

Scenario Outline: Add two numbers
    Given the input "<input>"
    When the calculator is run
    Then the output should be "<output>"

Examples:
    | input | output |
    | 2+2   | 4      |
    | 98+1  | 99     |
```

Cucumber

- Behavior-driven testing
- We write test descriptions in a restricted language (Gherkin)
- We execute these with Cucumber

Cucumber test-writing demo