



We're Coming for Your Flaky Tests!

Ole Friis Østergaard

June 15, 2022

@olefriis



Agenda

The problem with flaky tests

Flaky tests at GitHub

Strategy

Burning down flakes to 0

Examples

How to stay at 0 flakes

Questions

Me Me Me

Ole Friis Østergaard

Engineer at **GitHub**

[@olefriis](#)

github.com/olefriis

Blog: olefriis.github.io

Play Stunt Car Racer in your browser: olefriis.github.io/play



Based on Team Effort

Things I say **may** imply that I did all of this myself.

This is a lie.

I work in a wonderful team, and we worked on all of this together.



Two Caveats!

You may have entirely different kinds of flaky tests than we do!

Despite being GitHub-centric, no actual GitHub code will be shown.

The Problem with Flaky Tests

What's a Flaky Test?

Seemingly regular test that switches from success to failure for no apparent reason.

In practice it's hard to reproduce locally and fix. (Otherwise somebody would have already done it.)



Classic Unit Test

```
time = Time.at("2021-06-15 13:59:00")
customer_repository = mock_customer_repository(customers:
  ["Dave", "Bryan"]
)

report = ReportGenerator.generate(time, customer_repository)

assert_equals(2, report.number_of_customers)
assert_equals(["Dave", "Bryan"], report.customer_names)
```



Rails Unit Test

```
create_customer("Dave")  
create_customer("Bryan")  
  
report = ReportGenerator.generate  
  
assert_equals(2, report.number_of_customers)  
assert_equals(["Dave", "Bryan"], report.customer_names)
```



Example (involving math!)

10_000 tests in your suite.

1% are flaky.

=> 100 flaky tests.

Each flaky test will fail 1% of the time.

Your builds will fail ~~100%~~ of the time!!!

63

Debug
CI
Failures

Rerun
Failed
Jobs



Flaky Test Detection

Various ways of detecting flakes.

Azure DevOps:

- Run the whole test suite from a known “good state” 500 times. (Videos available on YouTube.)

GitHub:

- Recording test failures and successes, marking tests as flaky on too many errors.



Strategy

Accelerate, Chapter 4!

“When the automated tests pass, teams are confident that their software is releasable.”

“it’s worth investing ongoing effort into a suite that is reliable.”

[...flakes...] “you could just delete them. If they’re version-controlled (as they should be), you can always get them back.”

Nicole Forsgren, Jez Humble, Gene Kim



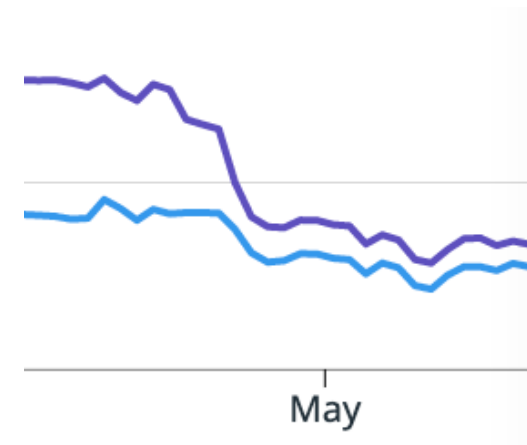
Delete or Fix!

Basic plan: Create pull requests for deletion of every flake (one pull request per code-owner), let them “simmer” for 2 weeks, then merge those whose flakes hadn’t been fixed.

Ignore flakes that didn’t occur within the last week.

Metrics, graphs.

Various ideas to keep the flakes at 0.



How Much Effort?

Thousands of flakes to start with.

A pretty big codebase.

Team of 4 people.

About 3 months.



How did it go?

After 2 rounds of pull requests: Still around 50 flakes.

After a few weeks of letting teams take care of their own tests: 16 flakes.

New flakes are still being introduced, and existing flakes pop up again.

Now wrapping up “actual work” and entering the “long-term mode”.



Psychology

Large Surface Area

Our team is small

GitHub is large

Many developers are introverts



People's Reactions to Deleting Their Tests?

“Sounds sensible!”
- Colleague X

“What a great idea!”
- Colleague Y



Deleting a Test is Hard!

Not **technically**, but **mentally**.

You keep the production code, but remove the safeguards.

This is **much** harder than just deleting production code.

“Sounds sensible! But what if we...”

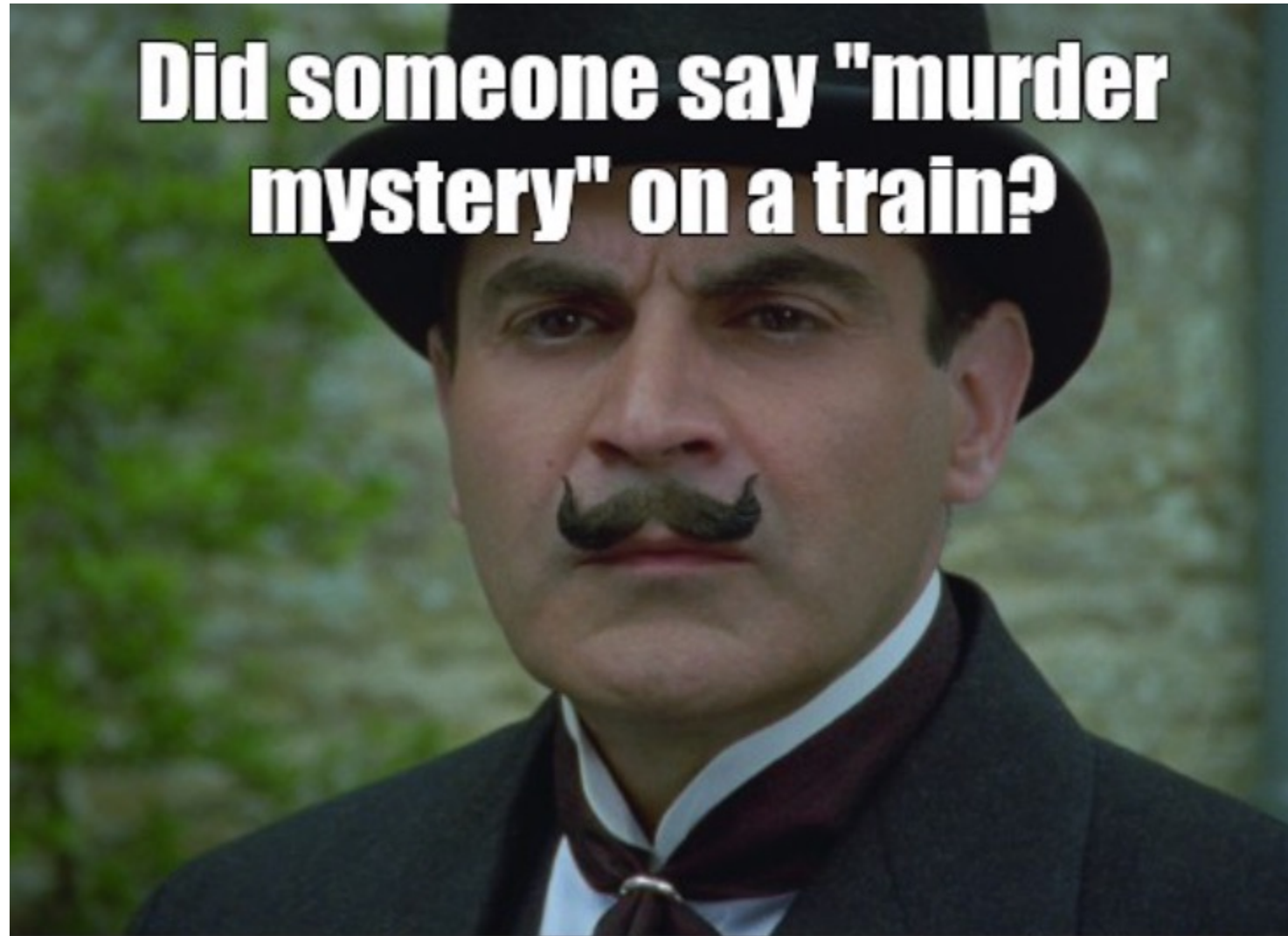
- Colleague X

“What a great idea! But let’s...”

- Colleague Y



Flakes Drag You In!



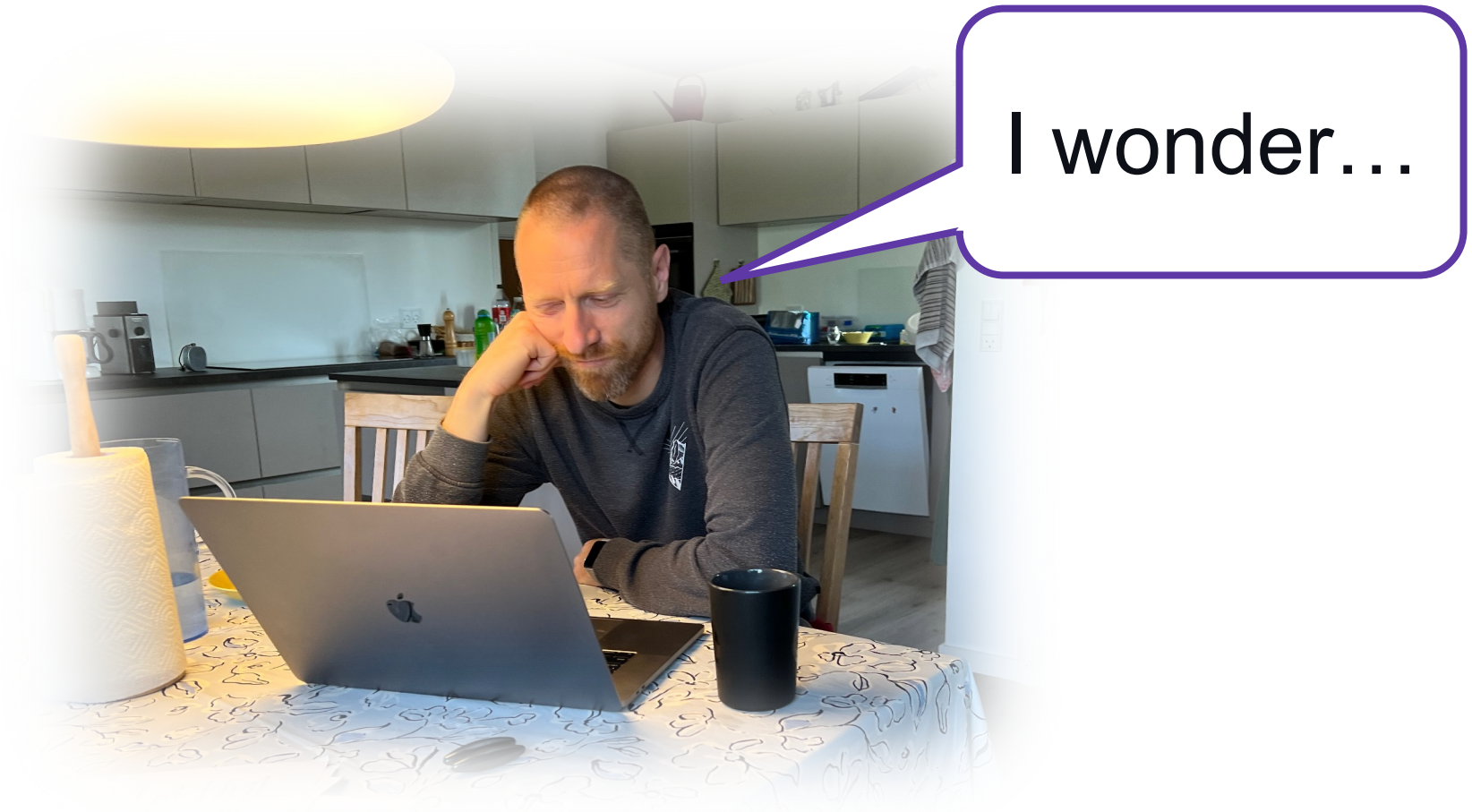
Completion Principle

Completeness **generates** energy
and
incompleteness **drains** energy

Types of Flakes

(For us, at least)

Results of Hours of Debugging



Our Categories

Hard-coded database IDs

Implicit ordering of database results

Timing

Lack of test isolation

Time bombs

Other

Hard-coded Database IDs

Make tests easy to read.

Seems like good testing advice: Keep tests simple.

Hard-coded Database IDs

```
class ReportTest
  setup do
    @paul = create(:user, id: 123, name: "Paul")
    @ringo = create(:user, id: 124, name: "Ringo")
  end

  test "can create a report with two users" do
    assert_equal generate_report, {
      user_ids: [123, 124],
      # ...
    }
  end
end
```



Hard-coded Database IDs

```
class ReportTest
  setup do
    @paul = create(:user, id: 123, name: "Paul")
    @ringo = create(:user, id: 124, name: "Ringo")
  end

  test "can create a report with two users" do
    assert_equal generate_report, {
      user_ids: [123, 124],
      # ...
    }
  end
end
```

```
test "can create report with more users" do
  create_list(:user, 5)
  assert_equal 7, generate_report[:user_ids].length
end
end
```

users table	DB sequence
	120
id name	121
123 Paul	122
124 Ringo	
121 John	
122 George	



Hard-coded Database IDs

How to avoid?

Linting rule against “create(:user, id: 123)” in tests?

Resetting database sequences between test suites?

Hard-coded Database IDs

```
class ReportTest
  setup do
    @paul = create(:user, id: 123, name: "Paul")
    @ringo = create(:user, id: 124, name: "Ringo")
  end

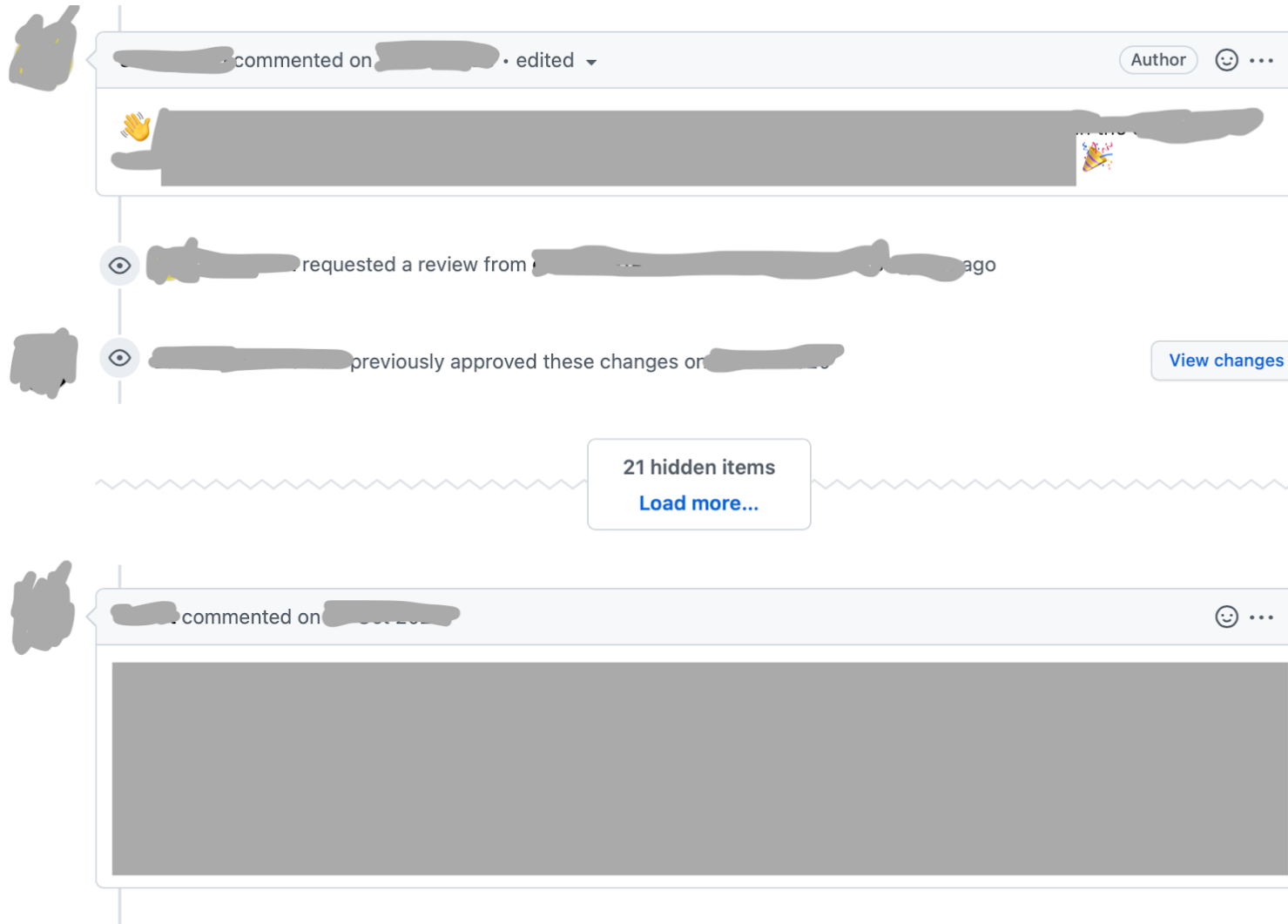
  test "can create a report with two users"
    assert_equal generate_report, {
      user_ids: [123, 124],
      # ...
    }
  end
end
```

```
class ReportTest
  setup do
    @paul = create(:user, name: "Paul")
    @ringo = create(:user, name: "Ringo")
  end

  test "can create a report with two users" do
    assert_equal generate_report, {
      user_ids: [@paul.id, @ringo.id],
      # ...
    }
  end
end
```



Implicit Ordering of Database Results



Implicit Ordering of Database Results

```
class TimelineTest
  test "hides middle of very long thread" do
    issue = create(:issue)
    comments = create_list(:comment, 5, issue: issue)

    assert_shown comments[0]
    assert_shown comments[1]
    assert_hidden comments[2]
    assert_shown comments[3]
    assert_shown comments[4]
  end
end
```



Implicit Ordering of Database Results

```
comments = issue.comments.order(:created_at)
if comments.length > 4
  result.hidden = comments.length - 4
  result.first_block += comments[0..1]
  result.last_block += comments[-2..-1]
else
  result.comments = comments
end
```



Implicit Ordering of Database Results

How to avoid?

Two ways to go:

- Make tests more likely to fail.
- Make tests more deterministic.

Increase Likelihood of Test Failure

Code like this:

```
comments = Issue.comments.order(:created_at)
```

is essentially the same as this:

```
comments = Issue.comments.order(:created_at, "RAND()")
```

So why not “fix” your ORM during test runs?

Increase Likelihood of Test Failure

```
module Shuffler
  def visit_Arel_Nodes_SelectStatement(o, collector)
    o.orders += Arel.sql("RAND()")
    super
  end
end
Arel::Visitors::ToSql.prepend(Shuffler)
```



Decrease Likelihood of Test Failure

When people write code like this:

```
comments = Issue.comments.order(:created_at)
```

they normally mean something like this:

```
comments = Issue.comments.order(:created_at, :id)
```

So why not “fix” your ORM?

Decrease Likelihood of Test Failure

```
module Shuffler
  def visit_Arel_Nodes_SelectStatement(o, collector)
    o.orders += Arel.sql(:id)
    super
  end
end
Arel::Visitors::ToSql.prepend(Shuffler)
```



Philosophy Segway: Decrease or Increase Likelihood of Test Failure?

Set your database sequences to specific values before test, or make them “more evil”?

Try to stabilize your external dependencies, or make them really unstable?

Make time go faster/slower, or just lock it down?

...and whatever else affects your tests...

Implicit Ordering of Database Results

```
class TimelineTest
  test "hides middle of very long thread" do
    issue = create(:issue)
```

```
    comments = create_list(:comment, 5, issue: issue)
```

```
class TimelineTest
  test "hides middle of very long thread" do
    issue = create(:issue)
    comments = 5.downto(1) { |i| create(:comment, issue: issue, created_at: i.minutes.ago) }

    assert_shown comments[0]
    assert_shown comments[1]
    assert_hidden comments[2]
    assert_shown comments[3]
    assert_shown comments[4]


  end
end
```



Timing

```
class ReportTest
  test "provides timestamp in footer" do
    expected_footer = "Generated at #{Time.now.strftime("%d/%m/%Y %H:%M %Z")}"

    assert_equal expected_footer, generate_report.footer
  end
end
```



Time passes!



Timing

How to avoid?

Same discussion as before:

- `Timecop.freeze`
- `Timecop.scale(3600)`

Timing

```
class ReportTest
  test "provides timestamp in footer" do
    expected_footer = "Generated at #{Time.now.strftime("%d/%m/%Y %H:%M %Z")}"
```

```
  as
end
end

class ReportTest
  test "provides timestamp in footer" do
    Timecop.freeze do
      expected_footer = "Generated at #{Time.now.strftime("%d/%m/%Y %H:%M %Z")}"

      assert_equal expected_footer, generate_report.footer
    end
  end
end
```



Lack of Test Isolation

One test alters global state, another assumes pristine state.

We know nowadays that tests should be run in a database transaction. But there are many other kinds of state that should be reset.

- Other databases (ElasticSearch, Redis, ...)?
- Session information.
- Time zone settings.
- Rate limiting.
- ...

Lack of Test Isolation

```
class GlobalizationTest
  test "handles other time zones" do
    Time.zone = "Australia/Sydney"
    report
    assert_equal Time_well report
  end
end
```

Succeeds

```
class BillingTest
  test "bills for the right day" do
    perform_bill
    assert_equal Date.today
  end
end
```

Fails



Lack of Test Isolation

How to avoid?

Add “state resets” to the global setup/teardown for your test suite.

Lack of Test Isolation

```
class BillingTest
  test "bills for the right day" do
    perform_billing
    assert_billed_for Date.today
  end
end
```

```
class BillingTest
  setup do
    Time.zone = "UTC"
  end

  test "bills for the right day" do
    perform_billing
    assert_billed_for Date.today
  end
end
```

```
class BillingTest
  test "bills for the right day" do
    perform_billing
    assert_billed_for Date.today
  end
end

# Global setup
testcase.setup do
  Time.zone = "UTC"
end
```



Time Bombs

Does anybody remember the year 2000?

Static test fixtures can become outdated and start to cause test failures.

Tests around midnight, summer/winter time changes, ...

Time Bombs

```
class SubscriptionTest
  test "knows when subscription is valid" do
    subscription = load_fixture("valid_subscription.json")
    assert subscription.valid?
  end
end
```

File: valid_subscription.json

```
{
  "username": "paul",
  "subscription_start": "2021-08-25",
  "subscription_end": "2025-08-24"
}
```



Time Bombs – Investigation!



Time Bombs – Investigation!

Whoa, a test started failing during the build! 🚨

Look up test file and associated production code.

“git log” all the things.

Something changed in the last 24 hours!

“git revert” those changes

You just reverted the fix!



Time Bombs

How to avoid?

Occasionally run your tests time-shifted to a future date.

Fix the date for your test.

Lint against hard-coded dates in the future.

Time Bombs

```
class SubscriptionTest
  test "knows when subscription is valid" do
    subscription = load_fixture("valid_subscription.json")
    assert subscription.valid?
  end
end
```

```
class SubscriptionTest
  test "knows when subscription is valid" do
    Timecop.travel("2022-03-04") do
      subscription = load_fixture("valid_subscription.json")
      assert subscription.valid?
    end
  end
end
```



“Other”

This is the biggest bucket!

Various inconsistencies between the tests and the code under test.

Floating-point rounding.

Flaky external dependency.

Actual production code issues.

...

Our Categories - Recap

Hard-coded database IDs

Implicit ordering of database results

Timing

Lack of test isolation

Time bombs

Other

Are We at 0 Flakes Now?

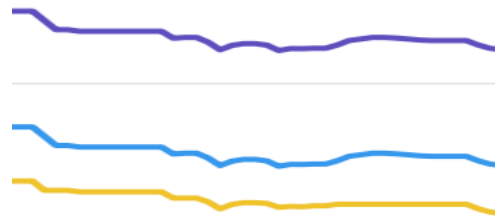
No, But...

The flakes we monitor are getting fewer and fewer.

Teams are generally very helpful in fixing “their” tests.

We’ve adjusted our monitoring a few times.

We’ll “squeeze” the remaining flakes to gradually cover more and more “low-impact flakes”.



Staying at ~0 Flakes

Stress New Tests

In CI, check which tests have been added and modified in a pull request. (GitHub has an API for that!)

Run these tests 100 times each, fail the build if any run fails.

This should make it harder to introduce new flakes.

Lots of ramp-up challenges, though.

Stress New Tests

Future ideas:

- Time-shift at random.
- Time-shift to before/after midnight.
- Time-shift to summer/winter time changes.
- Make time go slower/faster.
- Freeze time.
- Introduce randomness in the database results.

Updated Testing Documentation

We kept a “learning document” up to date when working on flaky tests, recording and categorizing the flakes.

It would be a shame not to incorporate this into the documentation on how to write tests.



Linters

Avoid “create(:user, id: 123)”.

In Rails: “Date.today” vs. “Date.current”.

In Rails: “find_by” -> “find_sole_by”.

...and anything that fits **your** code base.

Automatic “Delete Test” Pull Request Creation

As a new flake is detected, automatically create a pull request that deletes the test.

Follow up after a week or two and merge the PR if the owning team hasn't fixed it themselves.

This should be manageable by a single first responder.

No Size Fits All

Hopefully some food for thought.

...and we don't even know if our flake strategy in GitHub will be successful in the long run...



Summary

Now You Know...

What's a flaky test, why it's a problem.

Suggestions for strategies for fixing flaky tests.

Some psychology involved.

Why fixing flaky tests is hard!

Categories of flakes.

Ideas on how to stay at 0 flakes.

Questions?