Unlocking the Awesome Power of Refactoring at Work

SB Simply Business

Insuring Small Businesses. **Enabling Big Dreams**

Three things to keep in mind

1. Start anywhere
2. Start small
3. Strategic DDD can have a big pay off

It can be hard to know where to begin when confronted by legacy code. I encourage you start anywhere and start small - the Simple Design Dynamo can be one way to let the code flow. Once you bring accidental complexity as far down as possible, essential complexity is the next step.

Agenda

01 Context

02 Timeline

03 Technical Refactoring (some highlights)

04 Strategic Design Refactoring

05 Testing in Production with `scientist`

06 Outroduction

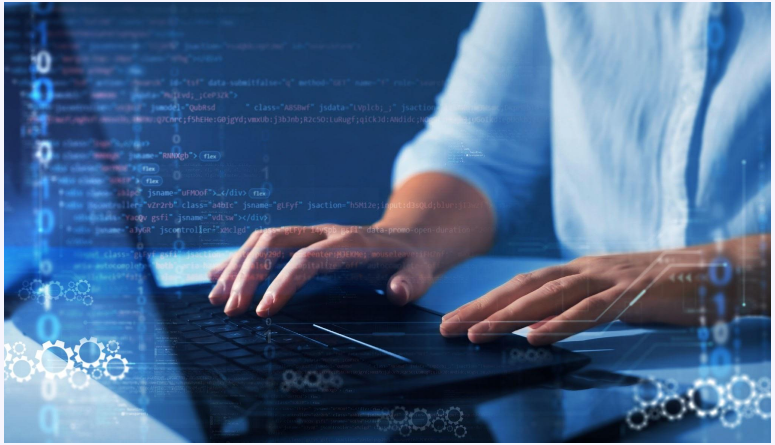
Simply Business

The beginning

In 2020, under pressure to deliver, a new lead management component was shipped with concerning levels of accidental complexity.

Memorable events of that year include:

- reviewed a 71 file PR meticulously;
- gnarly refactorings recommended;
- use of the Specification tactical pattern
- there were the beginnings of Domain Partitioning (c.f. Mark Richards, Software Architecture Monday)

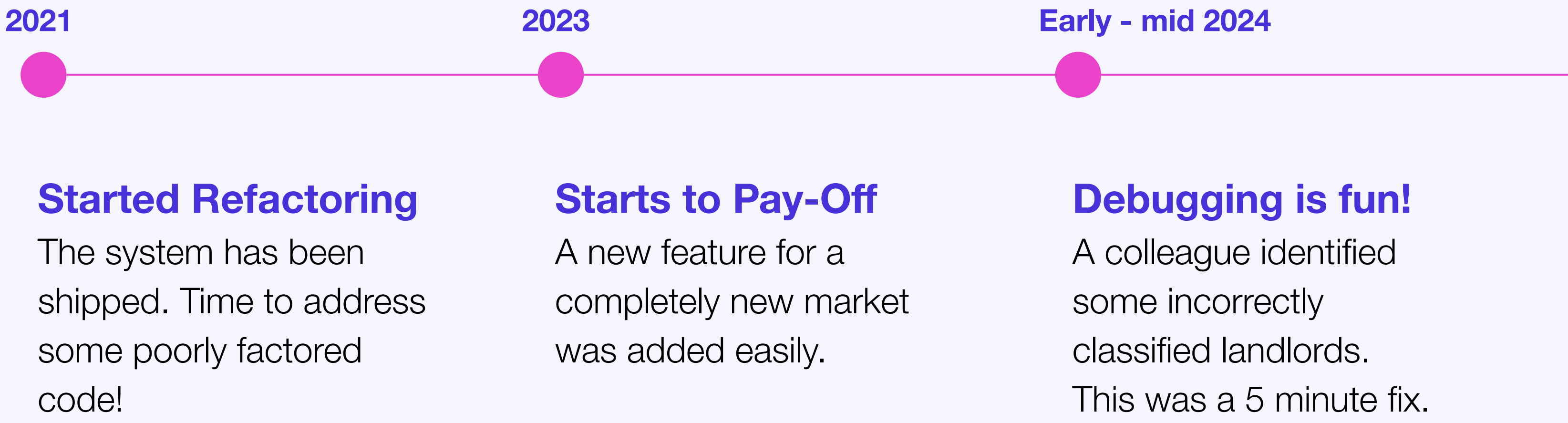


Simply Business

Timeline slide

Refactoring

Refactoring is hard, not only knowing when to start, but the uncertainty that it’ll pay-off.



2021

Started Refactoring

The system has been shipped. Time to address some poorly factored code!

2023

Starts to Pay-Off

A new feature for a completely new market was added easily.

Early - mid 2024

Debugging is fun!

A colleague identified some incorrectly classified landlords. This was a 5 minute fix.

Timeline slide

Evolving the Architecture

Enforcing boundaries around a ubiquitous language, making architecture obvious and keep enhancing its agility.

Early -Mid 2024

DDD

With accidental complexity minimised, the next step was to enforce linguistic boundaries.

Mid 2024

Ports & Adapters

Make Ports and Adapters obvious in a Rails codebase.

2025

Test Architecture?

How could we architect tests to enhance agility?

Guilty Yet Courageous

Feelings of guilt

I let poorly factored code through, and I felt I could’ve stood my ground better. I chose to atone for my sins.

Have I unlocked the awesome power of refactoring?

I remember seeing a presentation by J B Rainsberger, entitled “Unlocking the Awesome Power of Refactoring” I thought “I think I can rescue this situation.”

Start Anywhere: My First Microstep

I started at the **entry point** of the programme, where:

1. I aimed to maximise clarity: some method names weren’t clear enough, so I “improved” them; then
2. I inverted dependencies: a function was given an object but only needed one attribute from it, so why not just give it that attribute directly?;
3. I rigorously followed the definition of refactoring, making sure that the tests were **always green**. This way I could ship the **refactoring at a moment’s notice**.

Start Small: Maximise Clarity

```diff
-if (extranet_sales_platform = eligibility_for_extranets(rfq,
distribution_channel_name))

+if (extranet_sales_platform = find_extranet(rfq,
distribution_channel_name))
```

Start Small: Invert Dependencies

```
DeterminedSalesPlatform.new(
  sales_platform: DeterminedSalesPlatform::NONE,
 -rules: rules
 +reason: rules.reason
)
```

If the function really wants a reason, just give it that. Client-server coupling is loosened, modules are more context independent and thus reusable.

Before Refactoring

```
def determine_sales_platform(rfq:, distribution_channel_name:)
  if (extranet_sales_platform = eligibility_for_extranets(rfq,
distribution_channel_name))
    extranet_sales_platform
  elsif (rules = lead_should_be_suppressed(rfq))
    DeterminedSalesPlatform.new(...rules: rules)
  else
    DeterminedSalesPlatform.new({ … })
  end
end
```

After Refactoring

```
def determine_sales_platform(rfq:, distribution_channel_name:)
  if (extranet_sales_platform =find_extranet(rfq, distribution_channel_name))
    extranet_sales_platform
  elsif (rules = lead_should_be_suppressed(rfq))
    DeterminedSalesPlatform.new(..reason: rules.reason)
  else
    DeterminedSalesPlatform.new(..reason: rules.reason)
  end
end
```

Start Small: Minimise Ifs

The code still felt “inappropriate”. I knew that Ruby allows you to write code like so:

```
x || y || z
```

I wished I had something like this - it would make the code (slightly) more unconditional which could then lead to something like the following:

```
specifications.detect { |spec| spec.satisfied_by?(...) }
```

Before Refactoring

```ruby
def determine_sales_platform(rfq:, distribution_channel_name:)
  if (extranet_sales_platform = find_extranet(rfq, distribution_channel_name))
    extranet_sales_platform
  elsif (rules = lead_should_be_suppressed(rfq))
    DeterminedSalesPlatform.new(...reason: rules.reason)
  else
    DeterminedSalesPlatform.new(...reason: rules.reason)
  end
end
```

Slightly More Unconditional Code

```
def determine_sales_platform(rfq:, distribution_channel_name:)
 find_extranet(rfq, distribution_channel_name) ||
   lead_should_be_suppressed(rfq) ||
   backoffice
end
```

So I maximised clarity, inverted, but what heuristic am I using?

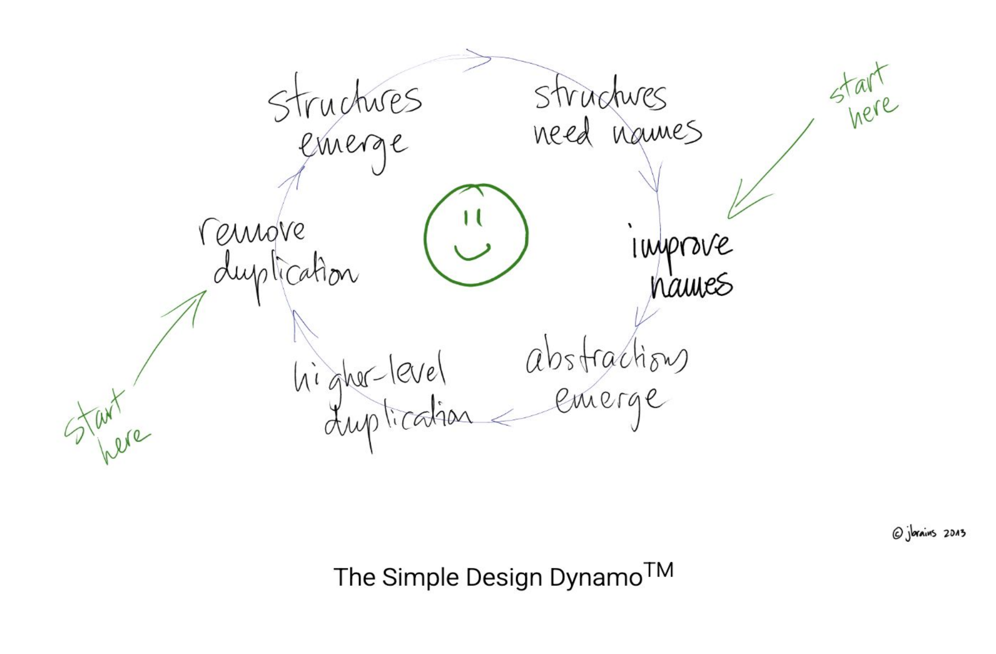
Four Rules of Simple Design

1. Passes the tests
2. Reveals intention
3. No duplication
4. Fewest elements

Rules 2 and 3 can form a tight feedback loop...

Beck Design Rules, Fowler (2015)

Simple Design Dynamo



The Simple Design Dynamo^{TM}

Putting an Age Old Battle to Rest, J. B. Rainsberger (2013)

Class Methods to Instance Methods

Next I focused on the SPECIFICATION objects where I noticed a lot of hard-coded details too deep in the code, which looked like they could become configurable. This may be a natural consequence of adopting class methods. By **inverting dependencies**, replacing class methods with instance ones, the **hard coded data could be turned into state variables that clients could pass in through the constructor**.

Class Methods to Instance Methods

```ruby
def self.satisfied_by?(rfq)
  new.satisfied_by?(rfq)
end


def satisfied_by?(rfq)
  AllOf[
    Specification.new(...),
    Specification.new(...)
  ]
end
```

Simply Business

A Sign: Increased Configurability

As I transformed the methods to instance ones, dependencies could be inverted and that meant the system was becoming increasingly **configurable**, to the point where a business or product person could tweak the rules for their use case. At one point, the **rules could indeed be configured using YAML**.

However, as the code continued to be changed, there were **subtle issues that suggested moving away from YAML** (in particular, an innocuous `array.flatten`). The best course of action was to move back to ruby configuration, half an hour's work for a developer. The code was **becoming increasingly agile**.

A Sign: A Global Platform

It wasn't just in the SPECIFICATION objects where dependencies were inverted, soon enough, we could configure polymorphic objects based on the business unit the code was executing in, meaning the system was becoming **a global platform**.

Finally, in August 2022, I informed our US colleagues that if they wanted, they could configure the system for their needs when they felt ready, and in January 2023, they indeed needed to add a new feature! **It took my colleague about an hour and half to introduce it and would definitely have been harder had this refactoring not happened.**

We were months ahead of the wider business.

Evolutionary Architecture

Using the dynamo got us going at a low level. Others supported me, either by refactoring as they worked on their cards or by reviewing my or their team's PRs. I found myself playing the role of architect, guiding teams to apply the dynamo themselves with rigorous TDD and feedback to me.

I established a feedback loop with Peter Vandenberk to help guide the higher level architecture as we go, in particular we debated over lead routing, lead classification or lead categorisation, finally settling on the latter.

I found myself in a flow, intrigued by where the design and architecture were going, and I just couldn't stop!

Domain Partitioning and Cohesion

A colleague had written an ADR to introduce Domain Partitioning, and the lead categorisation code was in multiple places (the domain partition and in Rails folders). So from January 2022, I began moving said code to a domain partition especially for the purpose. To do this, I appealed to a technique known as Parallel Change (also known as Expand and Contract), popularised by J B Rainsberger and others:

1. add the new thing;
2. migrate clients;
3. remove the old thing

Here, I would copy a module to the partition, run the tests, migrate clients of the module until there were none for the 'old' code, and then deleted that old code. This work inadvertently led to an evolution towards Ports and Adapters (although I didn't know that at the time).

Safely Evolving a Constructor

Say we have a constructor like so:

```
def initialize(x:, y:, direction:)

  @x = x

  @y = y

end
```

We now realise that (x, y) represents the concept of a Coordinate or Point and that’s what we really wanted to pass in all along.

Question: how could we change the signature above without breaking clients?

Add the New Thing

```
def initialize(x:, y:, direction:, starting_point: Point.new(x, y))

  @x = x

  @y = y

  @current_position = starting_point

end
```

The default setting of starting_point means clients (and therefore tests) won’t break.

Migrate Clients

```
MarsRover

  .new(x: 5, y: 5, direction: :north, starting_point: Point.new(5,
5))

…

MarsRover

  .new(x: -1, y: -1, direction: :north, starting_point: Point.new(-1,
-1))
```

Remove the old thing

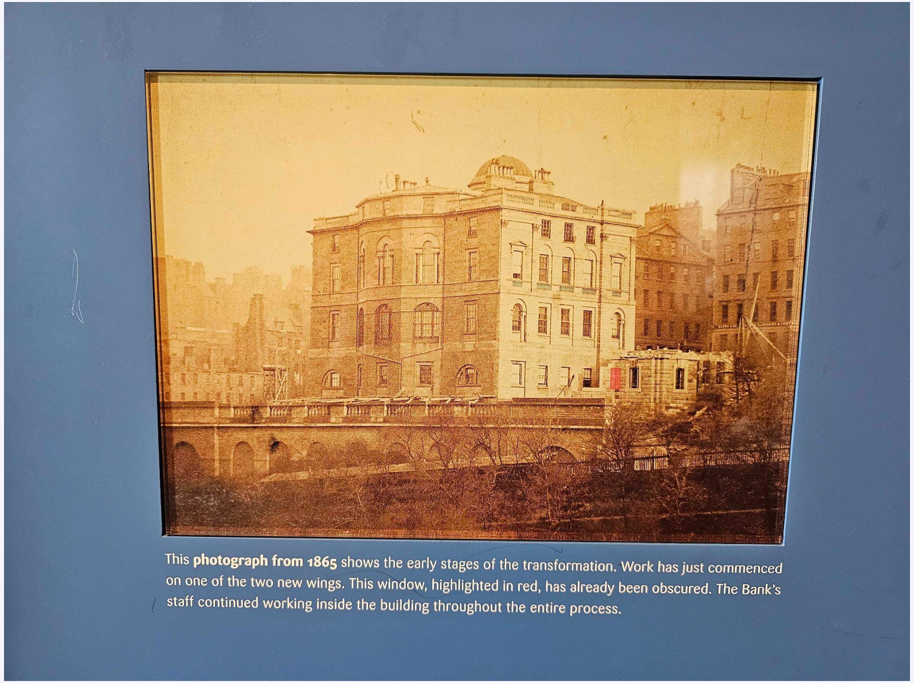
```
def initialize(direction: :north, starting_point:)

  @current_position = starting_point

end
```

Eventually, we may remove the old arguments (it involves a few more steps e.g. default setting the old arguments in the constructor so that clients no longer pass them in).

Do Not Disturb

Techniques like the ones listed above meant that teams could continue to deliver features without interruption.

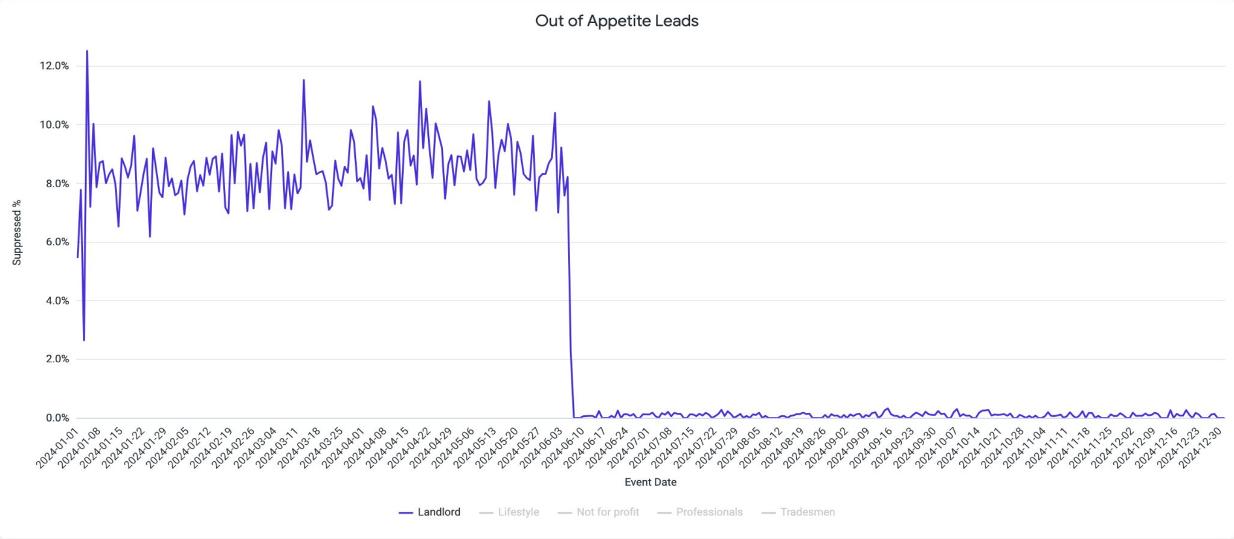
This is important, because often teams feel that they have to stop shipping features in order to refactor.



This **photograph from 1865** shows the early stages of the transformation. Work has just commenced on one of the two new wings. This window, highlighted in red, has already been obscured. The Bank's staff continued working inside the building throughout the entire process.

A Sign: Fixing Bugs Was Quick (and Fun)

A business person raised a concern to me that some landlords were incorrectly out of appetite. I wrote a test with him to reproduce the bug, and it was literally a five-minute fix.

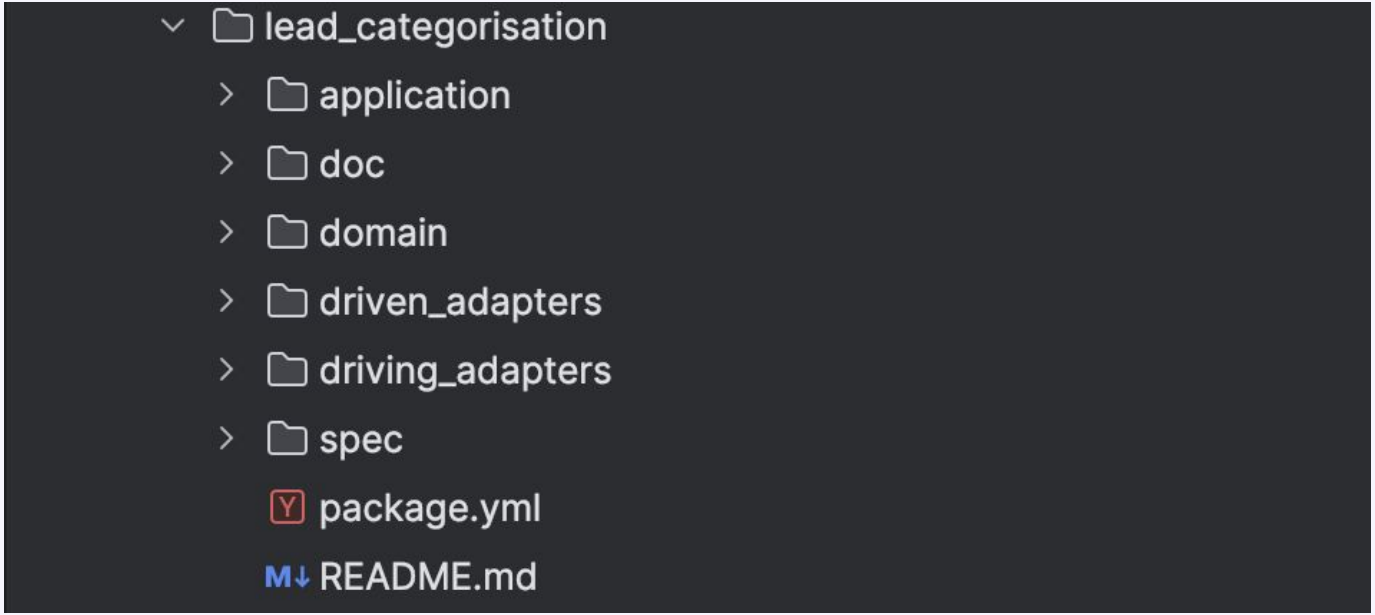


Ports and Adapters

I knew I'd separated the domain logic from frameworks (Rails, Sidekiq), but, quite fortuitously, in April last year (2024), I met Alistair Cockburn at Extreme Tuesday Club in Hackney, London.

He published a book with the late Juan Manuel Garrido de Paz called "Hexagonal Architecture Explained". One thing I remember in particular Alistair telling me was a folder for driven and driving adapters. As I pondered this, read his book, and it got me thinking about making Ports & Adapters obvious as the architectural style for my domain partition.

Ports and Adapters



```
lead_categorisation
  application
  doc
  domain
  driven_adapters
  driving_adapters
  spec
  package.yml
  README.md
```

Domain-Driven Design

Around that time, we also wanted to move away from a key concept in Simply Business' domain model - the Request for Quote (RFQ). This concept was used all over Lead Categorisation. I had been refactoring and soon there will be leads who did not make an RFQ. To start with, there wanted to be a bounded context, where the ubiquitous language was crisp and the domain model could be managed autonomously.

So my coachee and I introduced that context boundary, replace references to Request for Quote with a new concept, **Lead**. The first step was to **define an interface for it**, keep the methods closer together to present a coherent group responsible to a single actor, and have RFQ temporarily implement that interface, effectively have play the **role** of Lead. Afterwards, we were able to introduce Lead **safely**.

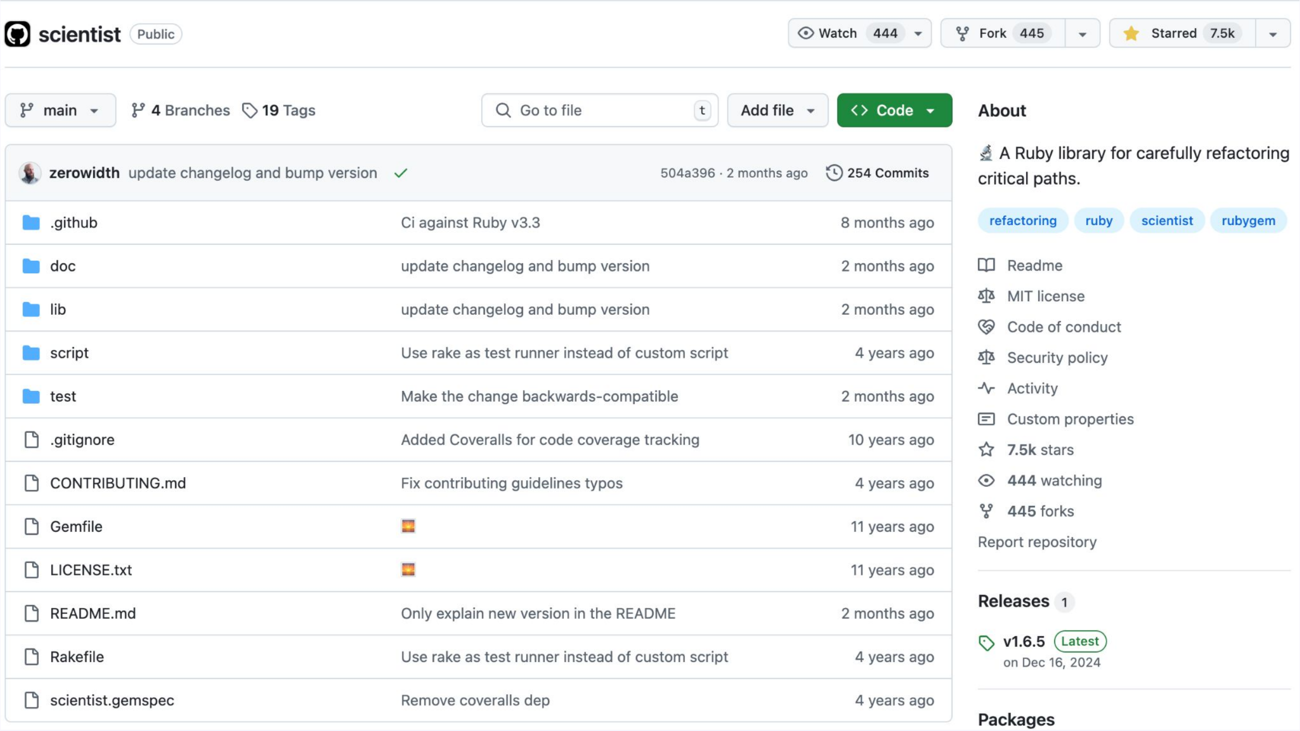
Lost in Translation Layers and Deprecated APIs

I was told by our architects that one of our internal APIs was being deprecated, and advised to move away from it. We discussed what other APIs could be used instead and was assured that it provided the same data as the deprecated one. One of our architects and I began by doing some conceptual mapping of concepts from other bounded contexts into ours (Lead Management), which helped me figure out how to incrementally ship the translation and keep the system working. However, there was a nagging question. My coachee actually designed the translation layer's public interface.

How can I design a **safe to fail experiment** to see that translating with the new API would yield the same results as the current way?

Enter…

Testing in Production With scientist



zerowidth update changelog and bump version ✓ 504a396 · 2 months ago 254 Commits

.github	Ci against Ruby v3.3	8 months ago
doc	update changelog and bump version	2 months ago
lib	update changelog and bump version	2 months ago
script	Use rake as test runner instead of custom script	4 years ago
test	Make the change backwards-compatible	2 months ago
.gitignore	Added Coveralls for code coverage tracking	10 years ago
CONTRIBUTING.md	Fix contributing guidelines typos	4 years ago
Gemfile	🌅	11 years ago
LICENSE.txt	🌅	11 years ago
README.md	Only explain new version in the README	2 months ago
Rakefile	Use rake as test runner instead of custom script	4 years ago
scientist.gemspec	Remove coveralls dep	4 years ago

About

🔬 A Ruby library for carefully refactoring critical paths.

Releases 1

v1.6.5 Latest
on Dec 16, 2024

Packages

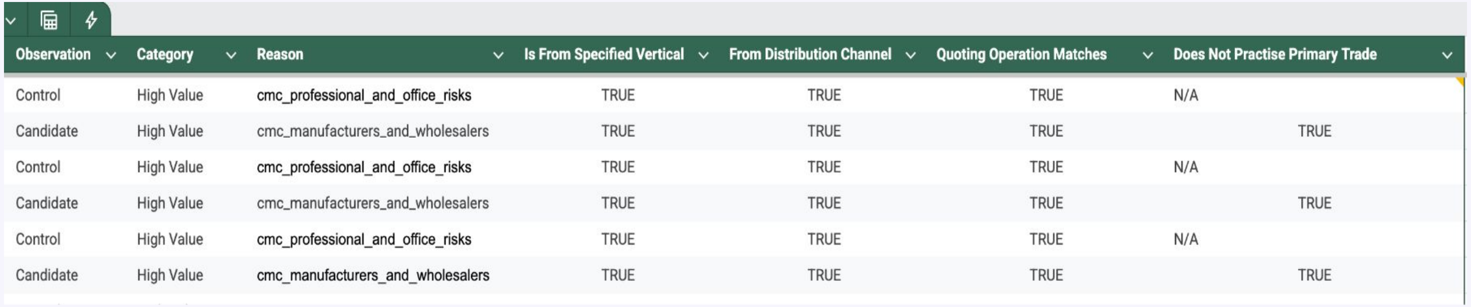
Integrating Scientist Was Easy

I hit upon the idea of using scientist on a quiet Sunday evening, and by Wednesday, I had the experiment (just the control) running in production. The next step was to test-drive the experiment I wanted to actually run, translating concepts using the new API (the candidate), and capturing the results to New Relic. All the refactoring from earlier made this surprisingly easy.

I decided to assert that the control and candidate experiments computed the same domain event in production. The candidate experiment would compute the domain event but it wasn't necessary to announce it downstream.

Bug Fixing Felt Like an Detective Story

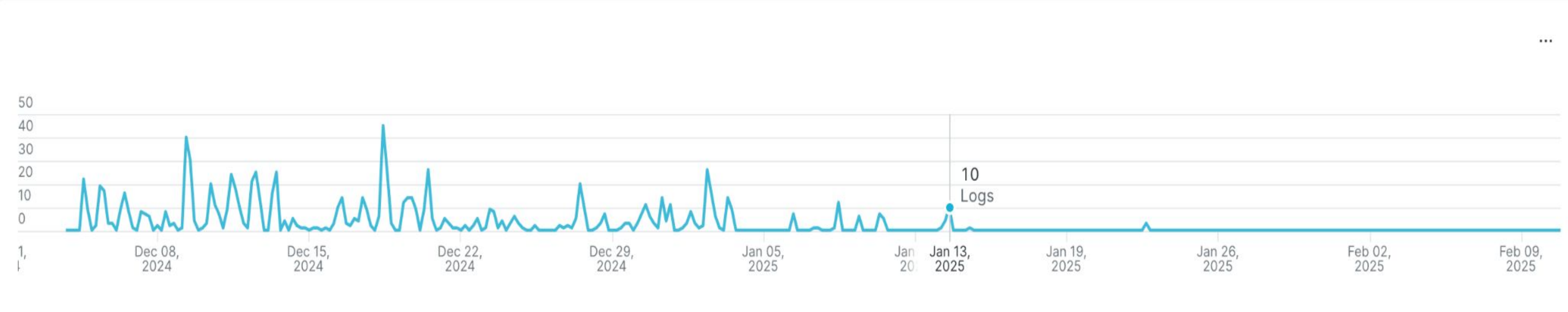
Owing to less stress, I could be scientific about my bug fixing (even using spreadsheets!). Here, the control represents what the outcome should have been. I could even use proof by contradiction to determine whether the SPECIFICATIONs were satisfied or not. The conclusion I came to was that I was not correctly translating the trade for certain types of leads.



Observation	Category	Reason	Is From Specified Vertical	From Distribution Channel	Quoting Operation Matches	Does Not Practise Primary Trade
Control	High Value	cmc_professional_and_office_risks	TRUE	TRUE	TRUE	N/A
Candidate	High Value	cmc_manufacturers_and_wholesalers	TRUE	TRUE	TRUE	TRUE
Control	High Value	cmc_professional_and_office_risks	TRUE	TRUE	TRUE	N/A
Candidate	High Value	cmc_manufacturers_and_wholesalers	TRUE	TRUE	TRUE	TRUE
Control	High Value	cmc_professional_and_office_risks	TRUE	TRUE	TRUE	N/A
Candidate	High Value	cmc_manufacturers_and_wholesalers	TRUE	TRUE	TRUE	TRUE

Testing in Production

The peaks represent the number of mismatched events per day so, thankfully, I didn't "just" switch over to the new API. It was time to find out the cause of the mismatches, and because there was no business impact I didn't need to panic.



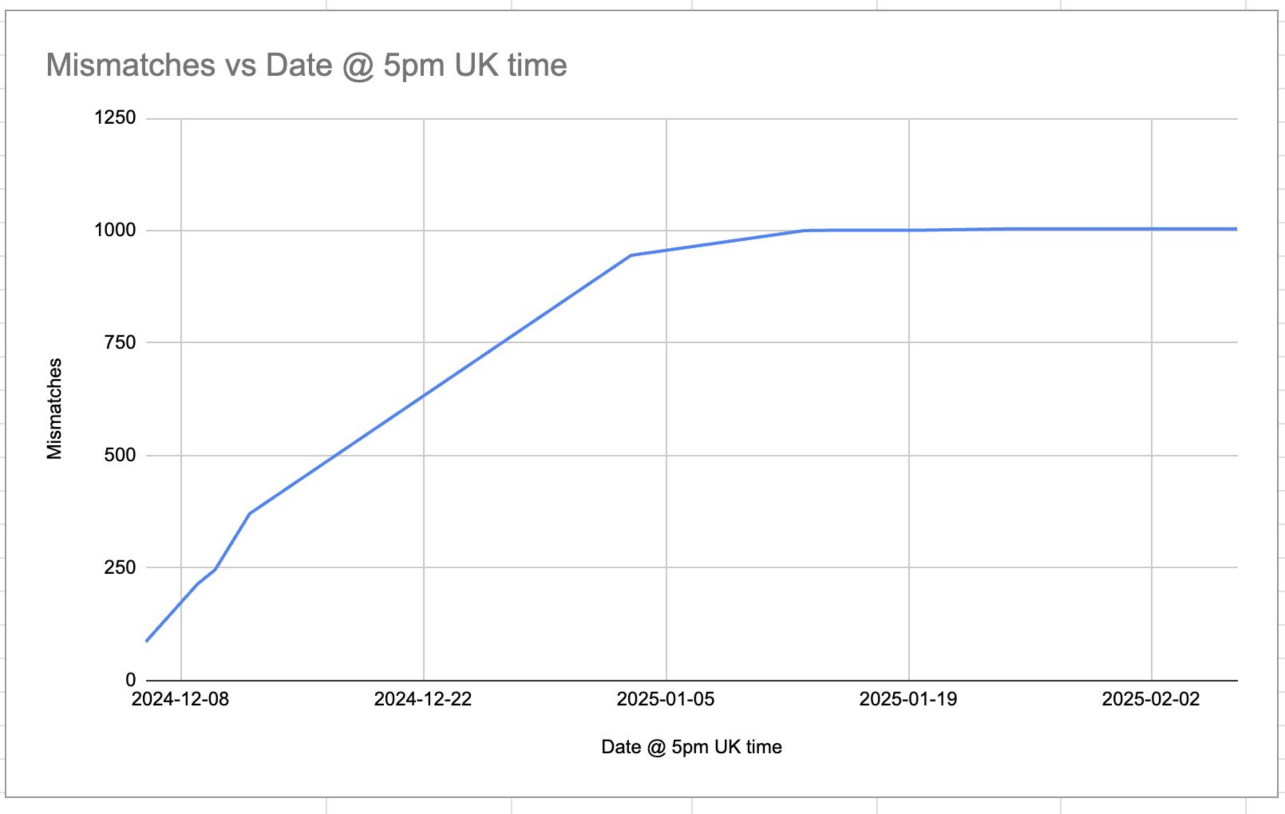
Simply Business

And Yet Still More Incorrect Assumptions

I fixed one translation error, but it wasn't enough: the number of mismatches was still increasing, albeit at a reduced rate. I casually spoke to one of my coachees about my issues and he explained the scenario to me, relating to particular kinds of leads, and how to translate the trade for them, so off I went and test-drove the fix.

Incrementally I shipped a fix until I could be confident enough there were no mismatches…

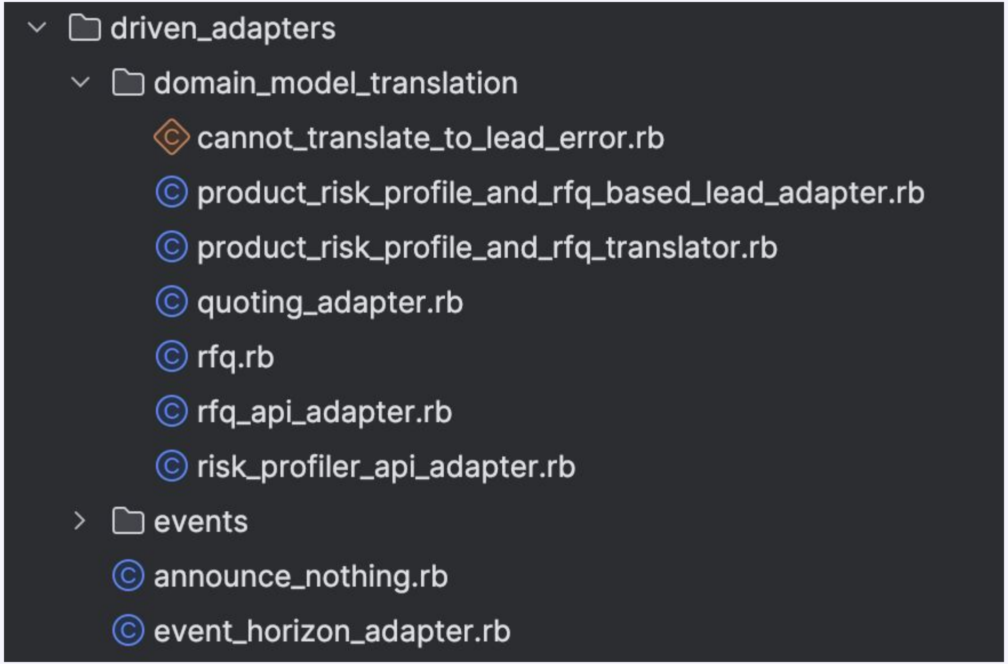
The Big Picture



Lost in Translation Layers Again

As I was fixing the mismatches, testing was proving very painful: I had set up I had to include to drive the public function, but it was irrelevant to what I was testing, namely, conceptual translation, so I realised that a separate object responsible purely for that could help. I used the Extract Class refactoring & testing was pure joy.

A translation layer appeared =>

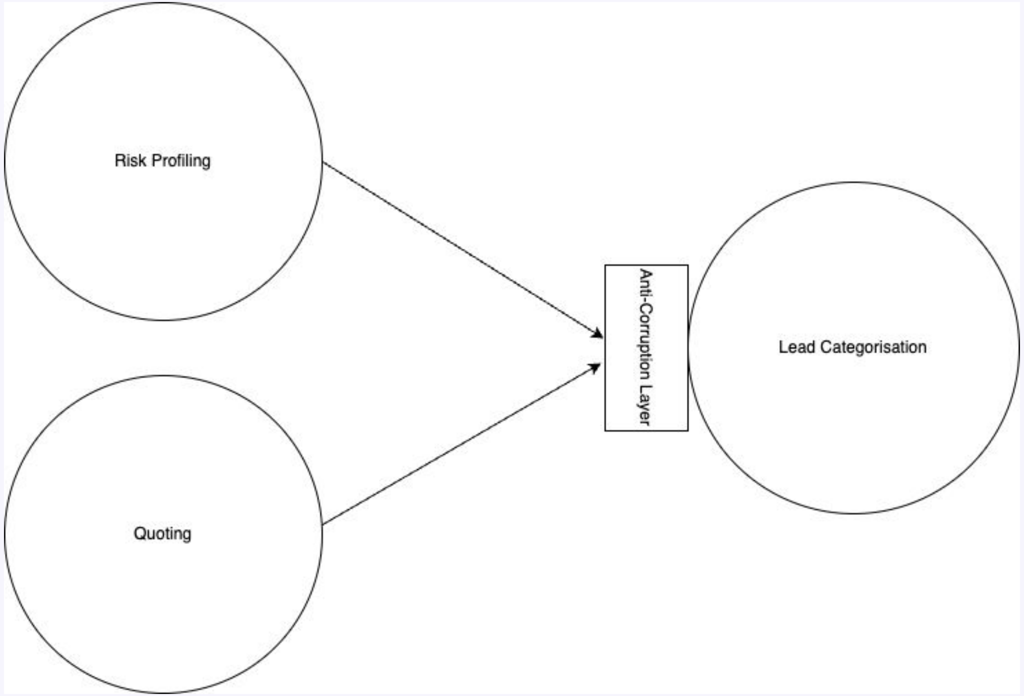


```
driven_adapters
    domain_model_translation
        cannot_translate_to_lead_error.rb
        product_risk_profile_and_rfq_based_lead_adapter.rb
        product_risk_profile_and_rfq_translator.rb
        quoting_adapter.rb
        rfq.rb
        rfq_api_adapter.rb
        risk_profiler_api_adapter.rb
    events
    announce_nothing.rb
    event_horizon_adapter.rb
```

Context Map

The effort of translation is somewhere between conformist and anti-corruption layer, but feels closer to the former.

What I found whilst lamenting the design decisions of Risk Profiling around how trade was stored, the consequences were contained inside the translation layer, and the Lead Categorisation domain model artefacts remained clean.



“

Contact points with other BOUNDED CONTEXTS are particularly important to test. Tests help compensate for the subtleties of translations and the lower level communication that typically exist at boundaries

Eric Evans

Domain-Driven Design

Retrospective

In Retrospect

Looking back what did I learn?

1 Little steps can make a big difference

2 Ensuring alignment, but also being guided by stories

3 Sponsorship

4 DDD can offer the next big pay-off in refactoring

5 Ports and Adapters

6 Safe to Fail Experiments and Testing in Production

Thank you

To the following:

1. Peter Vandenberk;
2. Colum O’Donovan;
3. Ben Johnson;
4. Cengizan Ziyaeddin;
5. Amar Shah;
6. Nitish Rathi;
7. Eric Evans;
8. Adam Scott;
9. Daniel Barlow;
10. Alistair Cockburn;
11. Tim Mackinnon

Image: UK Brand Campaign 2024

Insuring Small Businesses. **Enabling Big Dreams.**

SB Simply Business

Introducing

Hemal Varambhia

Technical Coach, Simply Business
Pronouns: He/Him/His
Github: @hemalvarambhia

About us

Simply Business is one of the UK's largest Business and Landlord insurance providers.

Since we started life in 2005, we've helped over three million small businesses and self-employed people find the protection that's right for them, from builders to bakers and personal trainers.

Simply Business

References

1. Unlocking the Awesome Power of Refactoring, J. B. Rainsberger (2021)
2. Domain-Driven Design, Tackling Complexity in the Heart of Software, Eric Evans (2003)
3. Implementing Domain-Driven Design, Vaughn Vernon (2013)
4. Putting an Age Old Battle to Rest, J. B. Rainsberger (2013)
5. Hexagonal Architecture Explained, Juan Manuel Garrido de Paz and Alistair Cockburn (2024)
6. Decoupling from Rails, Jim Weirich (2013)
7. Tidy First? A Personal Exercise in Empirical Software Design

Simply Business