

Behaviour Driven Development and thinking about testing

Malcolm Tredinnick
malcolm.tredinnick@gmail.com

Why have I asked you here today?

- 1) Testing landscape
- 2) A different way to think about this
- 3) Code

Chapter I: The Testing Landscape

Isolated

Integrated



Isolated

Integrated



Testing unit
size?

External service
availability

Mocks

Running time?

Mocks

- Substitute for external systems & components
- Need to be injected into the tests
 - “Monkey patching”
 - Dependency injection

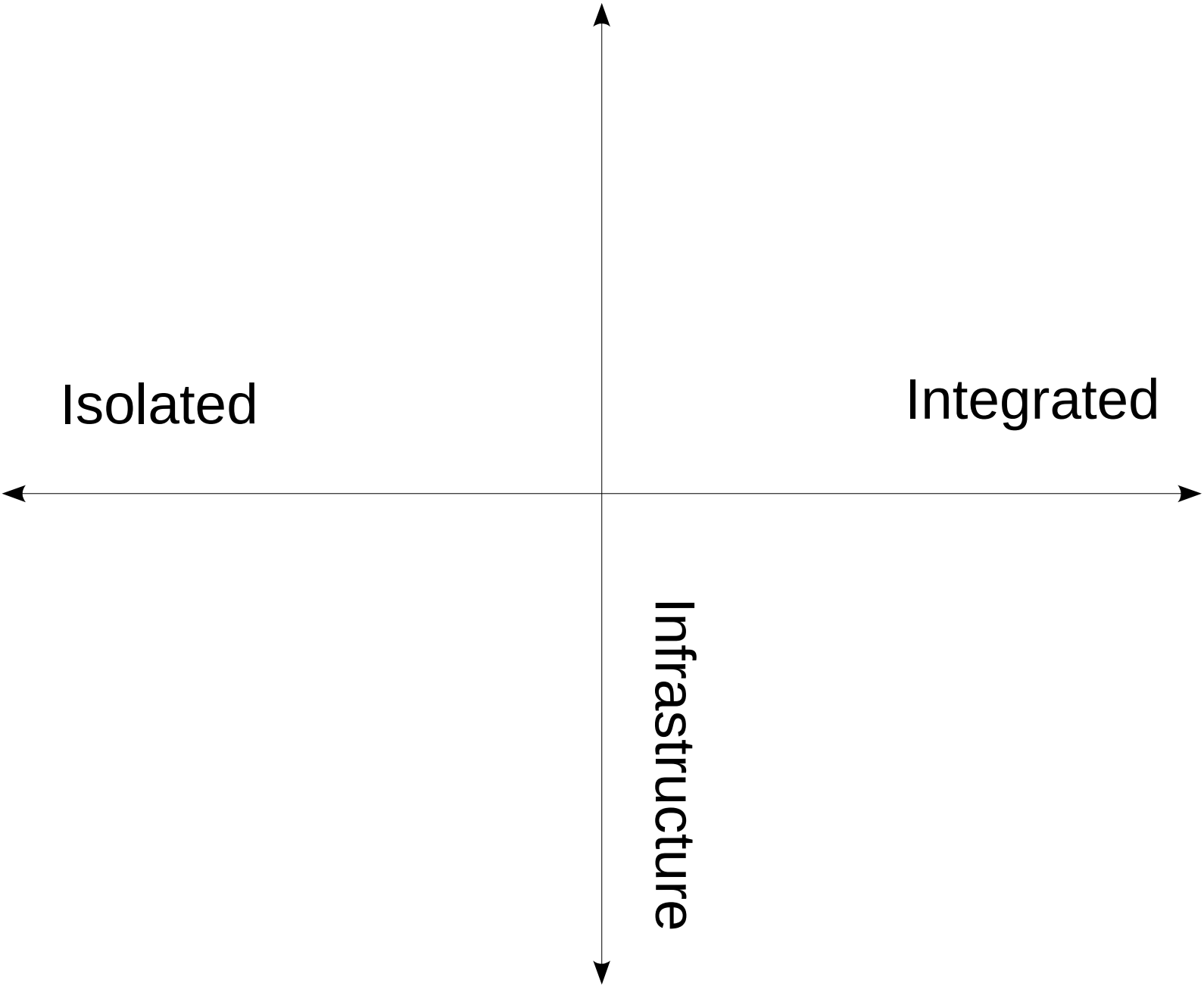
Monkey patching

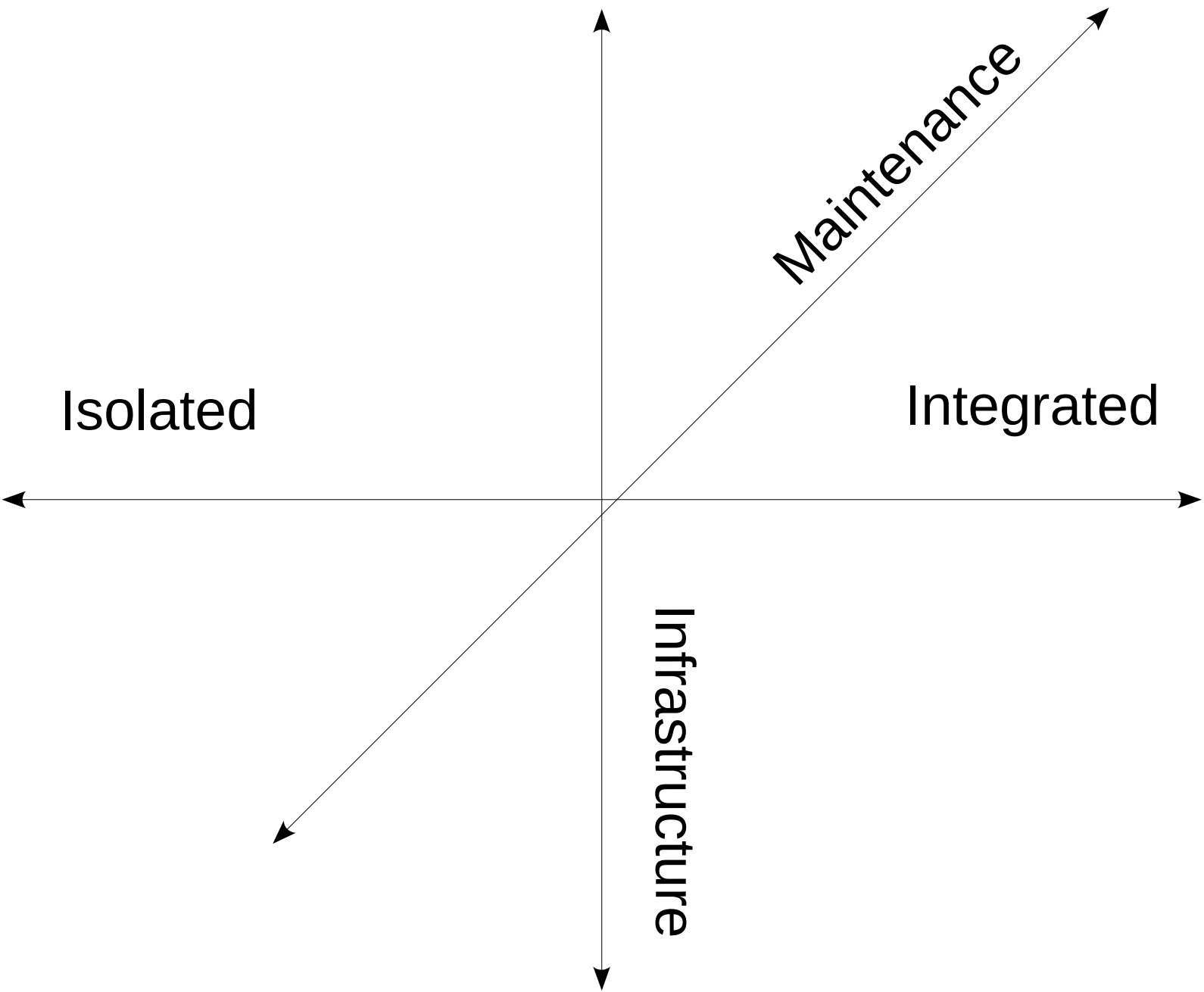
```
import socket
```

```
def setUp(...):  
    socket.socket = DummySocketObject  
    ...
```

Mocks

- Substitute for external systems & components
- Need to be injected into the tests
 - “Monkey patching”
 - Dependency injection
- Need to be kept in synch with reality!





Isolated

Integrated

Infrastructure

Maintenance

Testing Strategies

Some of many...

Testing Strategies

- Formal verification

Testing Strategies

- Formal verification
- Test everything you can think of

Testing Strategies

- Formal verification
- Test everything you can think of
- **Anti-regression suite**

No bug fix without a test

Testing Strategies

- Formal verification
- Test everything you can think of
- Anti-regression suite
- Test driven development (TDD)

TDD

1. Write a (failing) test for next method or class
2. Write minimal code to make test pass
3. Rinse, wash, repeat.

Testing Strategies

- Formal verification
- Test everything you can think of
- Anti-regression suite
- Test driven development (TDD)
“Specification, not verification”

Testing Strategies

- Formal verification
- Test everything you can think of
- Anti-regression suite
- Test driven development (TDD)
- Documentation driven development (DDD?)

Testing Strategies

- Formal verification
- Test everything you can think of
- Anti-regression suite
- Test driven development (TDD)
- Documentation driven development (DDD?)
- Behaviour driven development (BDD)

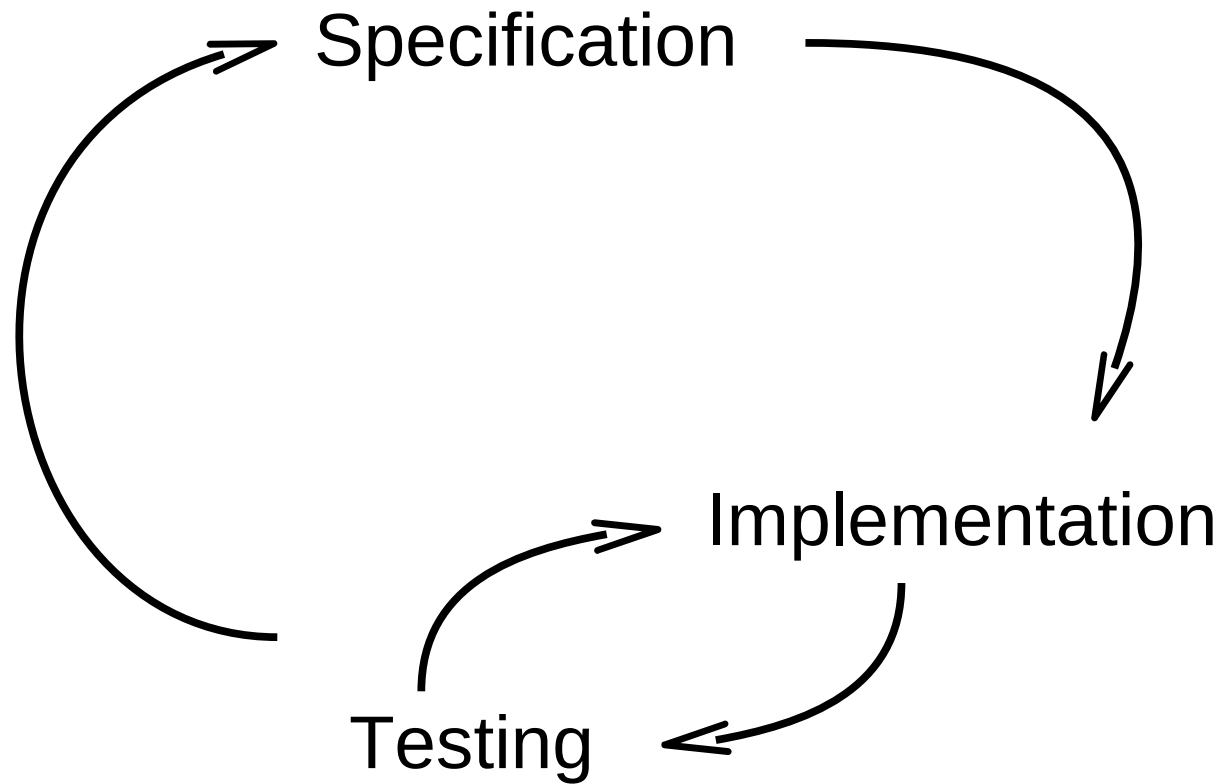
Chapter II: A different way of thinking

Interlude: Creating systems...

Interlude: Creating systems...

Specification

Interlude: Creating systems...



Specification

- On (e-)paper?
 - quick sketch
 - formal requirements document
 - mailing list discussion
- In your head?

Specification

- Capturing the idea is probably good
- “What was I ~~thinking~~ smoking when...?”

Specification

As a *[user or role]*,

I want *[feature]*

so that *[something happens]*

Specification

As a maze creator,

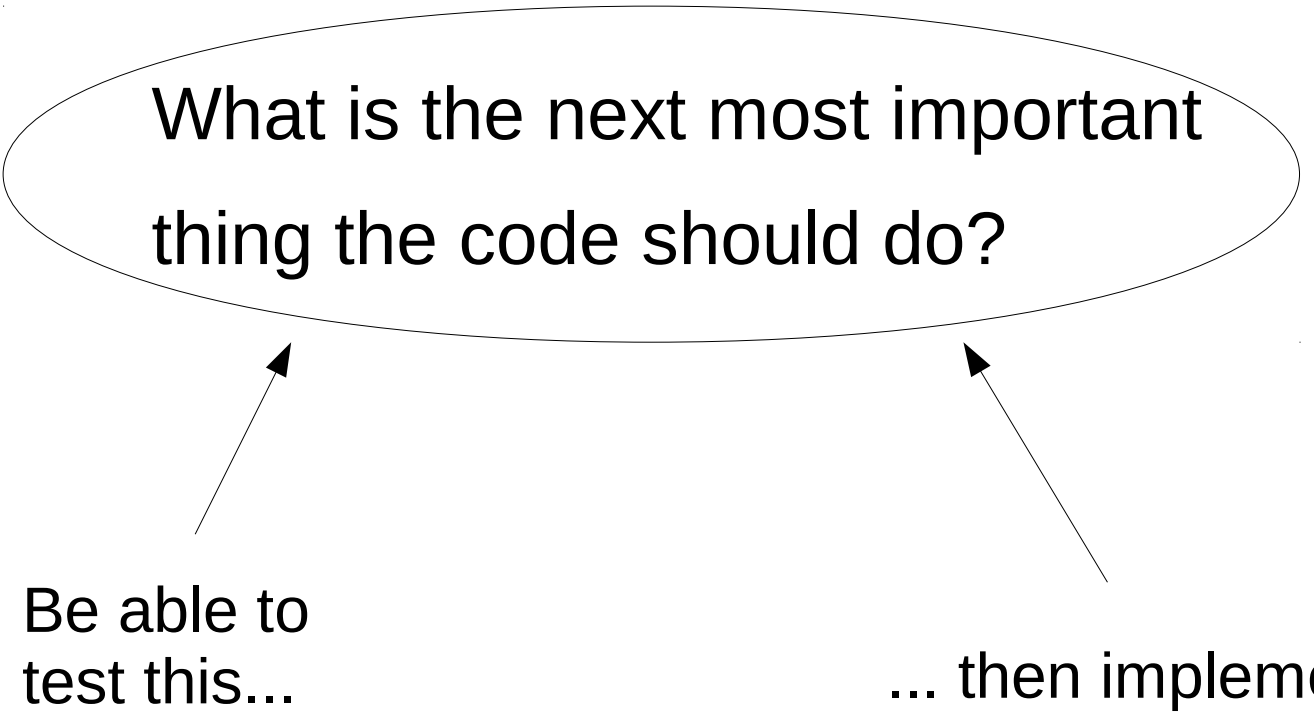
I want to be able to specify dimensions

so that a new maze of the given size is created.

BDD

What is the next most important
thing the code should do?

BDD



What is the next most important
thing the code should do?

Be able to
test this...

... then implement it.

Scenario

Given that *[initial context]*,

if and when *[some event occurs]*

then *[ensure some outcomes]*

Use case versus scenario

As a *[user or role]*,
I want *[feature]*
so that *[something happens]*

Given that *[initial context]*,
if and when *[some event occurs]*
then *[ensure some outcomes]*

Chapter III: Concrete code

Interlude: Naming things

Language helps guide our brain.

Focuses the mind.

Perhaps too exclusively.

Interlude: Naming things

The Sapir-Whorf Hypothesis:

“[...] the structure of a language affects the ways in which its speakers are able to conceptualise their world [view].”

Example #1: Python's unittest module

Test titles

```
class BasicFieldTests (TestCase):  
    def test_field_name (self):  
        ...  
  
    def test_show_hidden_initial (self):  
        ...
```

Test titles

```
class Fields(TestCase):  
    def should_allow_name_override(self):  
        ...
```

```
class ChoiceFields(TestCase):  
    def should_permit_initial_values_in_hidden_widgets(self):  
        ...
```

Run new test titles

Write a `load_test()` method to pull out all the “should” methods.

(Easy enough, but a few lines of code.)

Matching to scenarios

- Typically, end up creating more classes than unittests (not a bad thing; classes are “free”).
 - Thinking about test names helps with class names and roles.
- Context setup is done in setUp() method.
- Normal unittest assertions work fairly cleanly.

Example #2: Doctests

Narrative style testing

Jane created a new maze with a small (5 x 5) size.

```
>>> from maze import Maze  
>>> maze = Maze(5, 5)
```

Spot, her dog, did not think the new maze was the right size, but Jane was able to convince him like this:

```
>>> maze.x_size  
5  
>>> maze.y_size  
5
```

They found the entry and exit locations in their new maze:

```
>>> maze.entry_cell  
(0, 0, "left")
```

(etc)

Narrative style testing

- Excellent if you are truly writing a narrative.
- Can be difficult to maintain (but not always).
- Avoid if large amounts of setup required.
- Use with caution. Do use. Do not abuse.

Example #3: Domain specific languages

DSL packages

- Lettuce
- Freshen

Both available through PyPI

Freshen example

Scenario: Divide regular numbers

Given I have entered 3 into the calculator

And I have entered 2 into the calculator

When I press divide

Then the result should be 1.5 on the screen

(Code samples from Freshen documentation)

Backing code

```
from freshen import *
```

```
import calculator
```

```
@Before
```

```
def before(unused):  
    scc.calc = calculator.Calculator()  
    scc.result = None
```

```
@Given("I have entered (\d+) into the calculator")
```

```
def enter(num):  
    scc.calc.push(int(num))
```

(Code samples from Freshen documentation)

Backing code

```
@When("I press (\w+)")  
def press(button):  
    op = getattr(scc.calc, button)  
    scc.result = op()
```

```
@Then("the result should be (.*?) on the screen")  
def check_result(value):  
    assert_equal(str(scc.result), value)
```

(Code samples from Freshen documentation)

Freshen templated example

Scenario Outline: Add two numbers

Given I have entered <input_1> into the calculator

And I have entered <input_2> into the calculator

When I press <button>

Then the result should be <output> on the screen

Examples:

input_1	input_2	button	output	
20	30	add	50	
2	5	add	7	
0	40	add	40	

(Code samples from Freshen documentation)

Conclusions(?)

- Behavioural tests are worth trying
- Question the way you think from time to time
- Are your tests' purpose clear to future you?

<https://github.com/malcolmt/bdd-and-testing-talk>

malcolm.tredinnick@gmail.com
@malcolmt