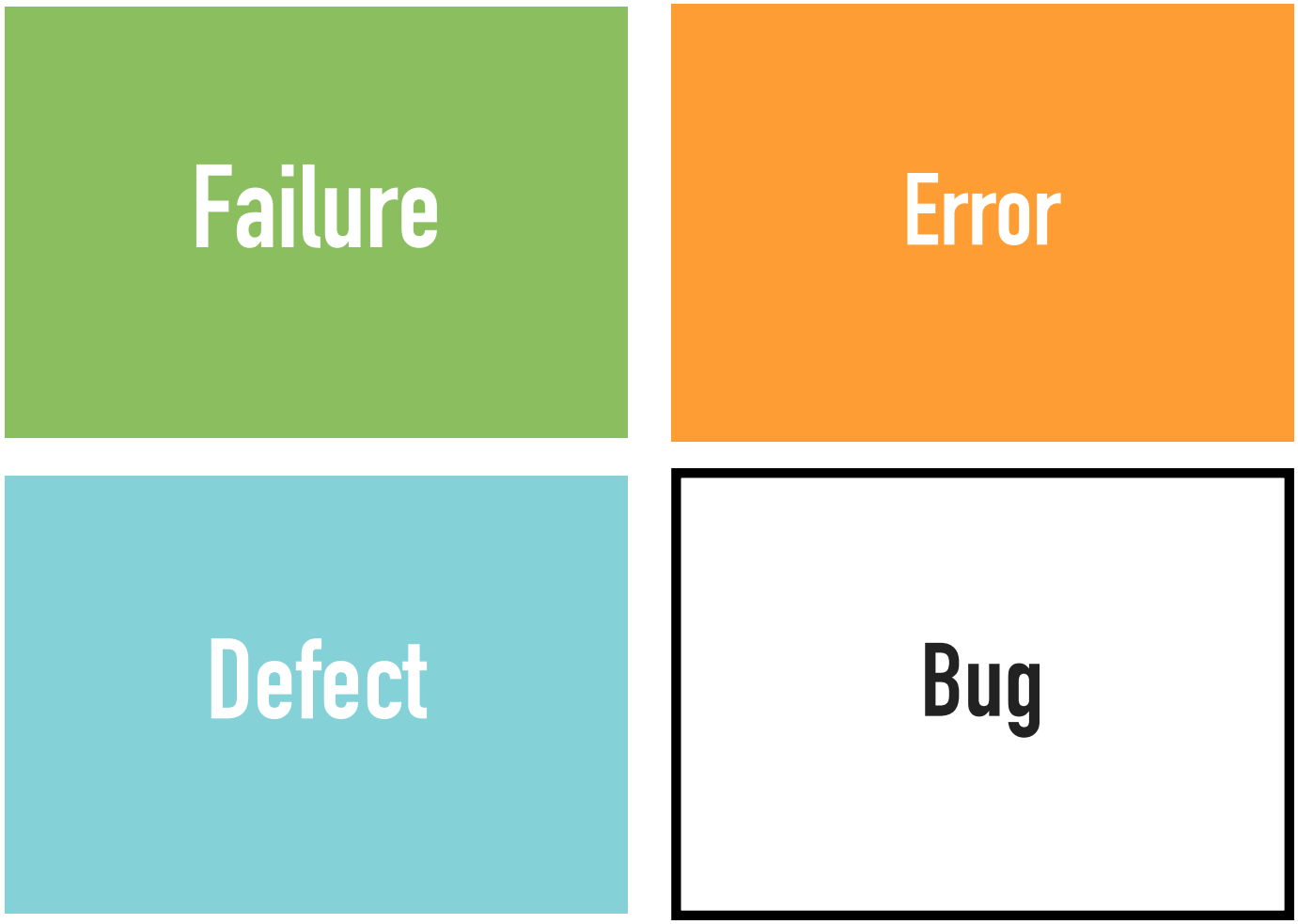
SEG 2105 - LECTURE 10

Testing and Inspecting to Ensure High Quality

Key Terminology



Failure

- Unacceptable behaviour
- Reliability (frequency of failures)
- Design objectives
  - Low failure rate
  - High reliability
- Violation of explicit (or implicit) requirement

DEFECT (OR BUG)

- A flaw of the system
  - Contributes to failures
- Found in requirements, design, code, anywhere!
- Several defects might be *needed* to cause a particular failure

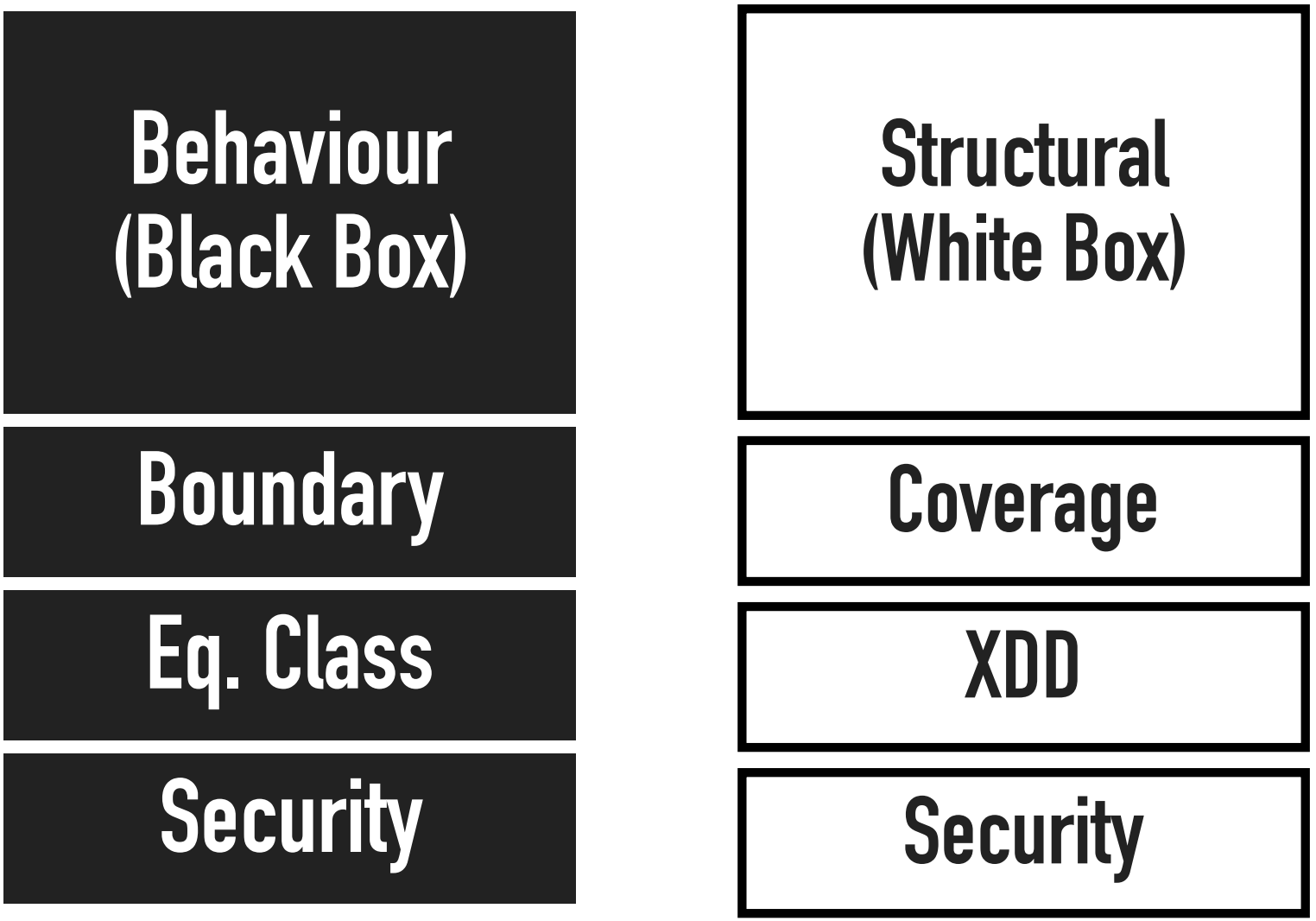
ERROR (IN THIS CONTEXT)

- Inappropriate decision leading to a defect

```
if (age > 18) {
   return "Yes, you can vote";
} else {
   return "No, not yet!";
}
```

Off by 1 error

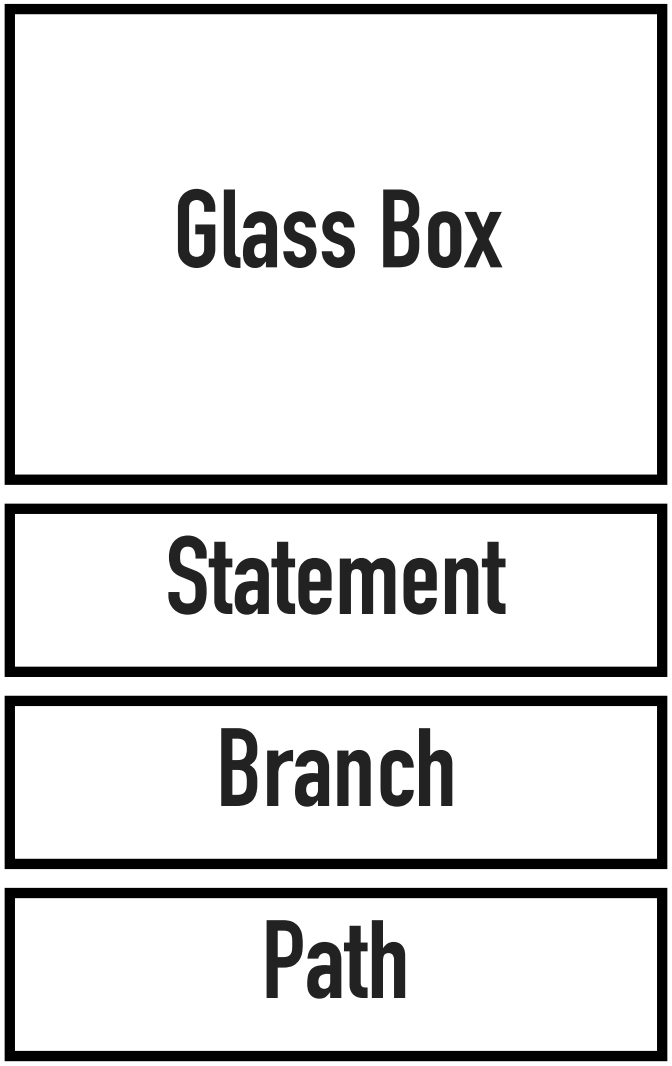
Testing Approaches



GLASS BOX

TESTING

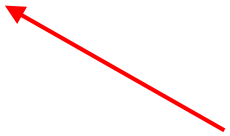
Types of Coverage



```
var displayVote = ((age, didVote) => {
  let canVote = null;
  if (age > 18) {
    canVode = true;
  } else {
    canVote = false;
  }

  let message = null;
  if (!didVote) {
    message = "You should consider voting.";
  }

  if (!canVote) {
    message += "When you are older.";
  }
  return [canVote, message];
});
```

```
var displayVote = ((age, didVote) => {
  let canVote = null;
  if (age > 18) {
    canVode = true;
  } else {
    canVote = false;
  }

  let message = null;
  if (!didVote) {
    message = "You should consider voting.";
  }

  if (!canVote) {
    message += "When you are older.";
  }
  return [canVote, message];
});
```

Statement Coverage
age = 28
didVote = false



Accidental coverage

```
var displayVote = ((age, didVote) => {
  let canVote = null;
  if (age > 18) {
    canVote = true;
  } else {
    canVote = false;
  }

  let message = null;
  if (!didVote) {
    message = "You should consider voting.";
  }

  if (!canVote) {
    message += "When you are older.";
  }
  return [canVote, message];
});
```

Statement Coverage
age = 28
didVote = false

canVote not canVode

```
var displayVote = ((age, didVote) => {
  let canVote = null;
  if (age > 18) {
    canVote = true;
  } else {
    canVote = false;
  }

  let message = null;
  if (!didVote) {
    message = "You should consider voting.";
  }

  if (!canVote) {
    message += "When you are older.";
  }
  return [canVote, message];
});
```

Statement Coverage
age = 18
didVote = false

Accidental coverage

```
var displayVote = ((age, didVote) => {
  let canVote = null;
  if (age >= 18) {
    canVote = true;
  } else {
    canVote = false;
  }

  let message = null;
  if (!didVote) {
    message = "You should consider voting.";
  }

  if (!canVote) {
    message += "When you are older.";
  }
  return [canVote, message];
});
```

Statement Coverage
age = 17
didVote = false

18 or older
(not older than 18)

```
var displayVote = ((age, didVote) => {
  let canVote = null;
  if (age >= 18) {
    canVote = true;
  } else {
    canVote = false;
  }

  let message = null;
  if (!didVote) {
    message = "You should consider voting.";
  }

  if (!canVote) {
    message += "When you are older.";
  }
  return [canVote, message];
});
```

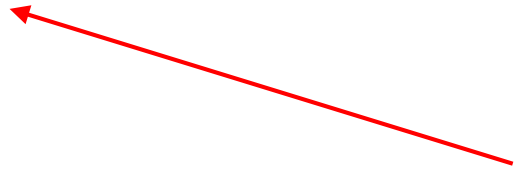
Branch Coverage
age = 18
didVote = true

Branching wanting all paths (including the "else" that does nothing)

But in this case, it still would not have caught this bug.

```
var displayVote = ((age, didVote) => {
  let canVote = null;
  if (age >= 18) {
    canVote = true;
  } else {
    canVote = false;
  }

  let message = null;
  if (!didVote) {
    message = "You should consider voting.";
  }

  if (!canVote) {
    message += "When you are older.";
  }
  return [canVote, message];
});
```

Path Coverage
age = 17
didVote = true



In this case “message” was not properly initialized

THE BEAUTY OF TESTS . . .

```
assert(
  [true, "Thanks for voting!”],
  displayVote(18, true))

assert(
  [true, "You should consider voting."],
  displayVote(18, false))

assert(
  [false, "How did you vote?"],
  displayVote(17, true))

assert(
  [false, "You should consider voting.When you are older."],
  displayVote(17, false))
```

More clearly document your intent. And, allow those *other developers* that can’t stand your code to not break it.

```
var displayVote = ((age, didVote) => {
  let canVote = null;
  if (age >= 18) {
    canVote = true;
  } else {
    canVote = false;
  }

  let message = "";
  if (!didVote) {
    message = "You should consider voting.";
    if (!canVote) {
      message += "When you are older.";
    }
  } else if (!canVote) {
    message = "How did you vote?";
  } else {
    message = "Thanks for voting!";
  }

 return [canVote, message];
});
```

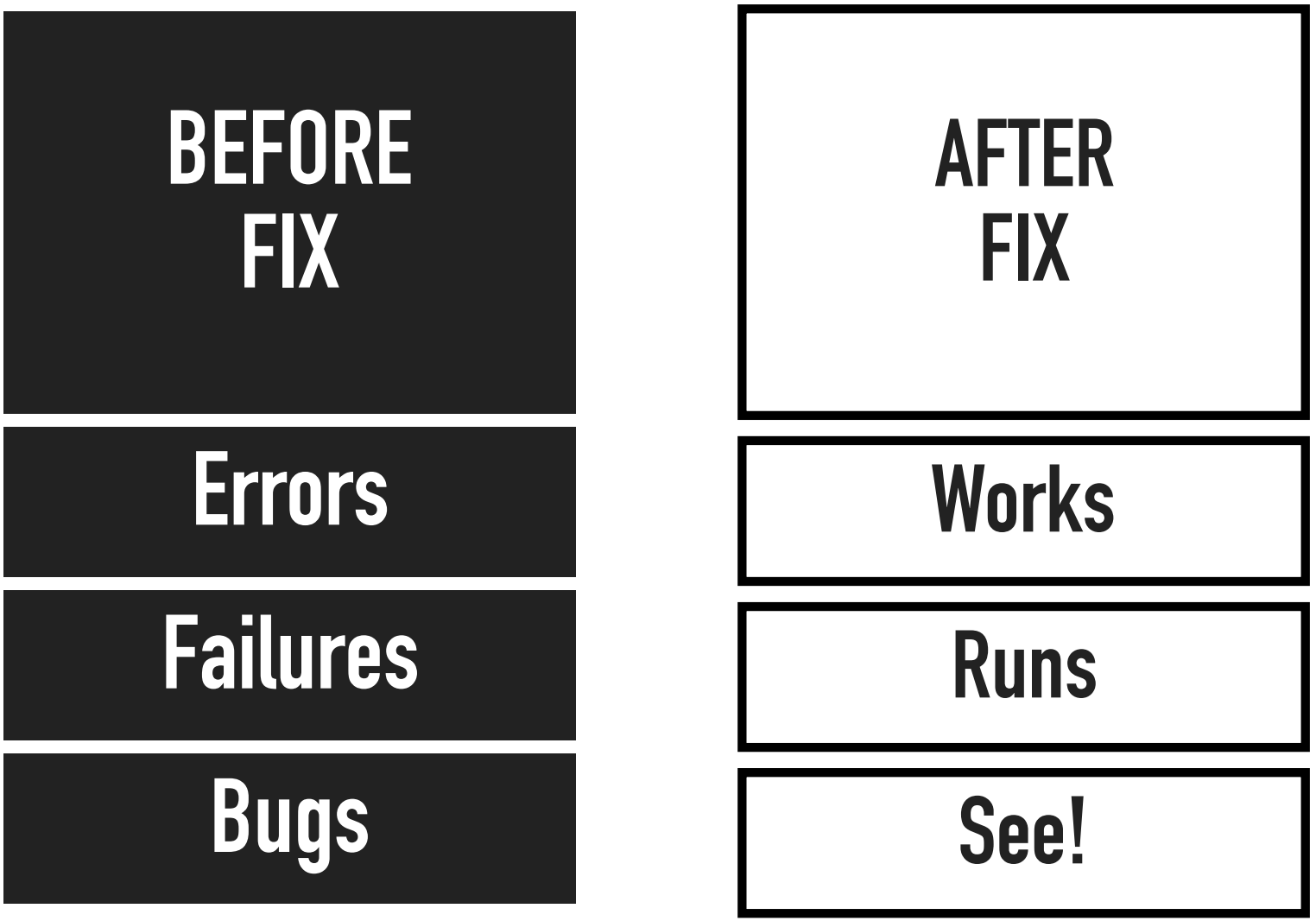
Working code, but tried to be clever to save a few lines, and now it's quite confusing.

GLASS-BOX TESTERS HAVE ACCESS TO .... EVERYTHING!

- Run a debugger
- Generate a trace
- Inject new code to see what happens

Consistently Replicating a bug is the biggest challenge

THE "LIGHT SWITCH" TO (INCREASE CONFIDENCE) IN YOUR FIX

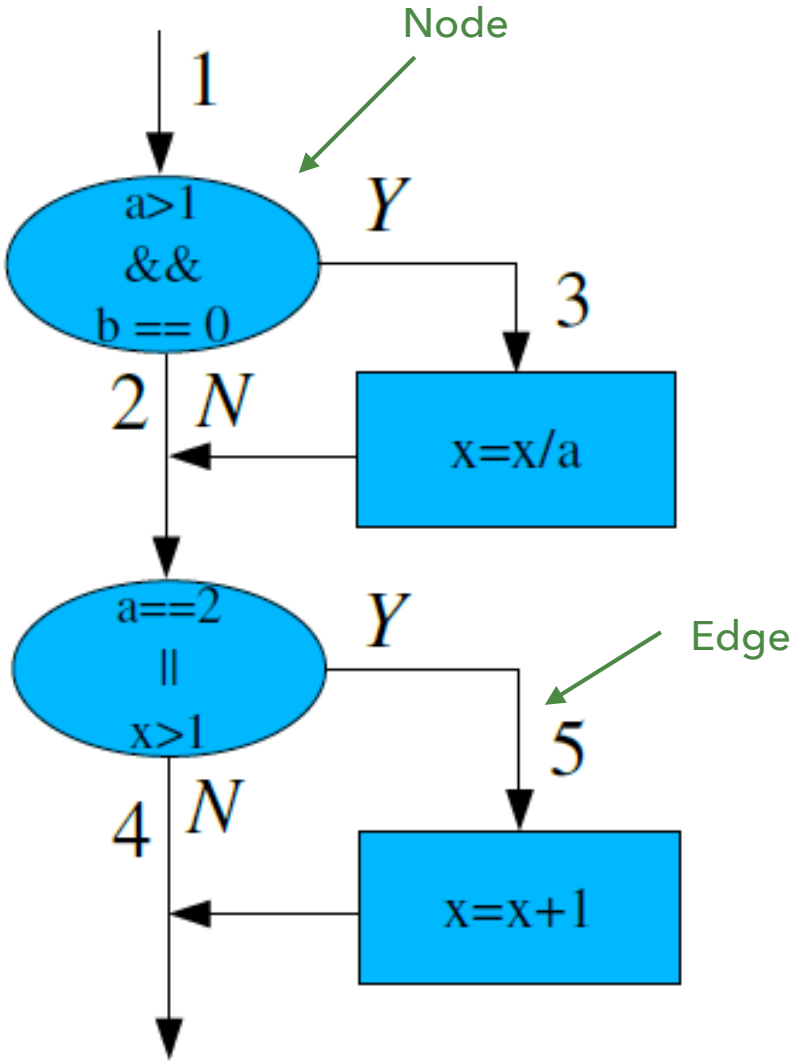


FLOW GRAPH FOR GLASS-BOX TESTING

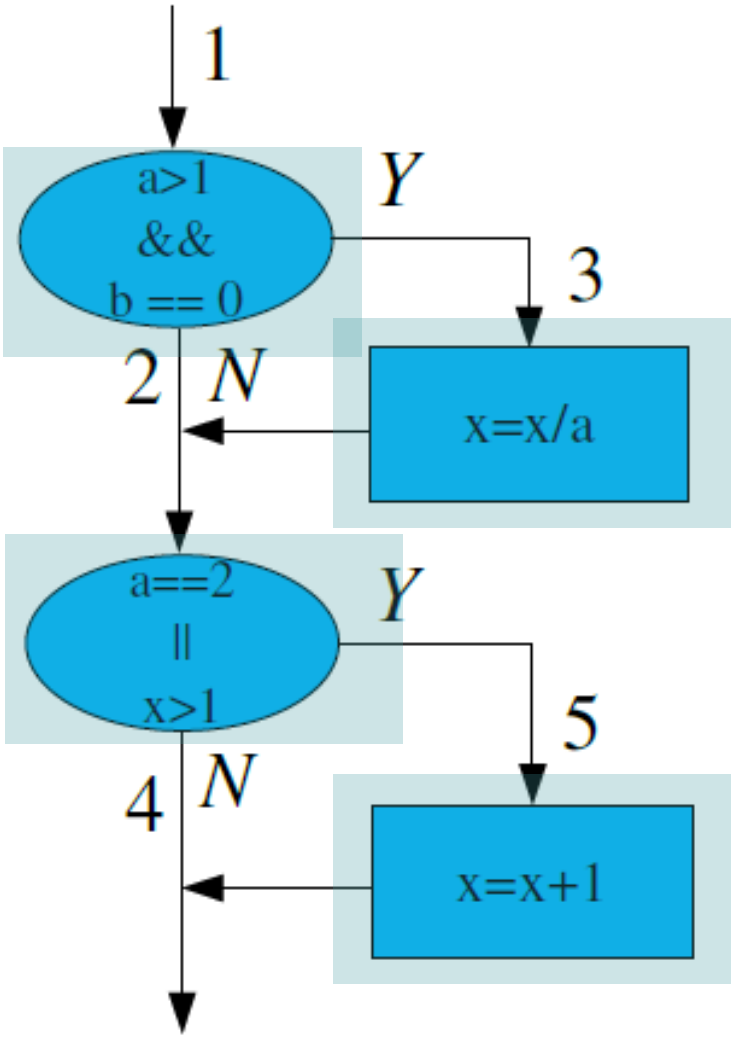
- Each branch in the code (such as if and while statements) creates a node in the graph
- The testing strategy has to reach a targeted coverage of statements and branches; the objective can be to:
  - Cover all possible **paths** (often infeasible)
  - Cover all possible **edges** (most efficient)
  - Cover all possible **nodes** (simpler)

FLOW GRAPH FOR GLASS-BOX TESTING

```
int proc(int a, int b, int x) {
  if ((a>1) && (b==0))
    x = x/a;
  if ((a==2)||(x>1))
    x = x+1;
  return x;
}
```

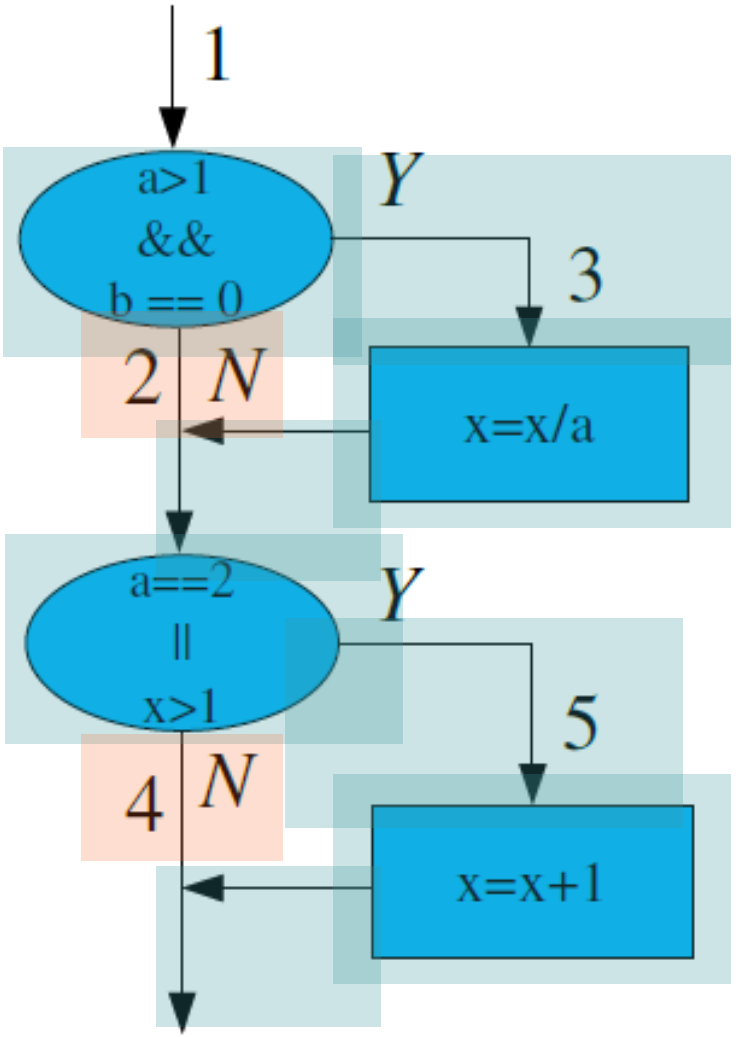


Node Coverage



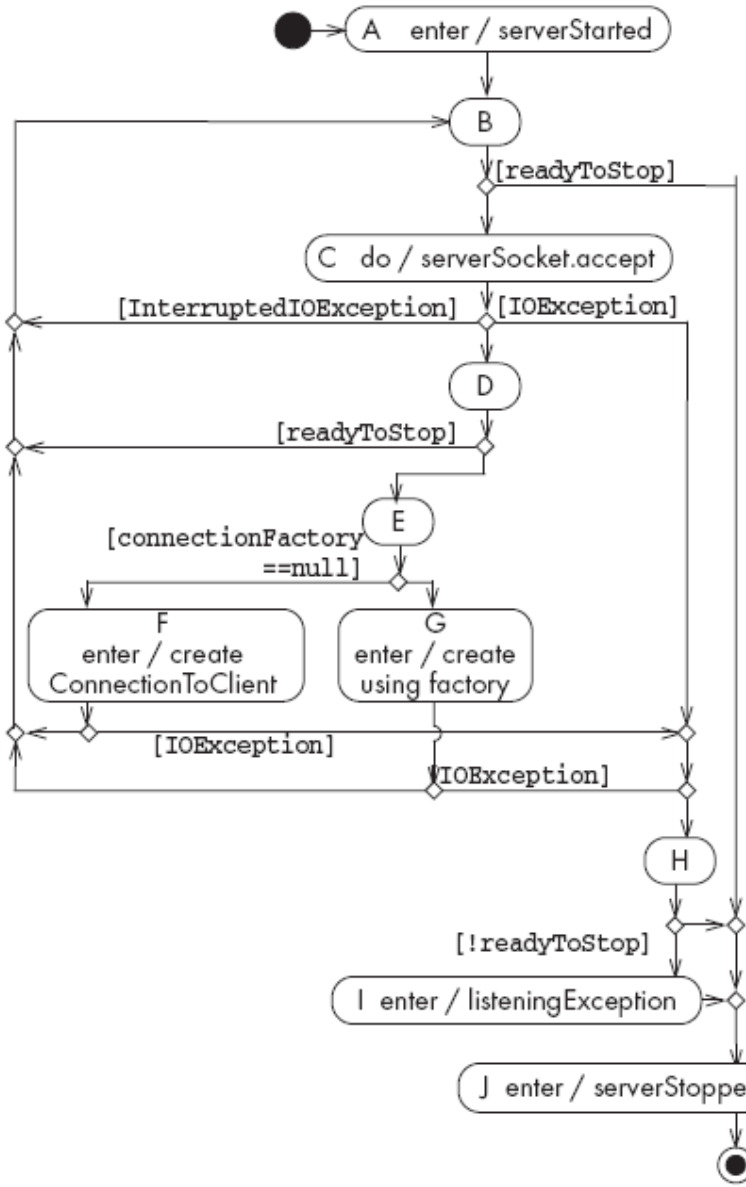
Test Case 1
a = 2
b = 0
x = 6

Edge Coverage



Test Case 1
a = 2
b = 0
x = 6

Test Case 2
a = 3
b = 1
x = 0




```
readyToStop= false;
serverStarted();     // A
try
{
  while(!readyToStop)  // B
  {
    try
    {
      Socket clientSocket = serverSocket.accept();  // C
      synchronized(this)
      {
        if (!readyToStop) // D
        {
          if (connectionFactory == null) {  // E
            new ConnectionToClient(         // F
              this.clientThreadGroup, clientSocket, this);
          } else {
            connectionFactory.createConnection(    // G
              this.clientThreadGroup, clientSocket, this);
          }
        }
      }
    }
    catch (InterruptedIOException exception) { }
  }
}
catch (IOException exception)
{
  if (!readyToStop)   // H
  {
    listeningException(exception);  // I
  }
}
finally
{
  readyToStop = true;  // J
  connectionListener = null;
  serverStopped();
}
```

BLACK BOX

TESTING

BLACK-BOX TESTERS HAVE ACCESS TO . . . THE SYSTEM

- Provide the system with inputs
  - Observer outputs
- No access to
  - Source code
  - Internal data
  - Documentation relating to the systems internals

EQUIVALENCE CLASSES

- Impossible to test by brute force using all input values
  - e.g. every integer
- Instead, divide the possible inputs into groups
  - Should be treated similarly
  - Known as **equivalence classes**
- A tester needs only to run 1-3 tests per equivalence class
  - You must understand the required input
  - Appreciate how software *may* have been designed

EXAMPLE, KNOWING THE VOTING AGE

```
assert(
  [true, "Thanks for voting!”],
  displayVote(18, true))

assert(
  [true, "You should consider voting."],
  displayVote(18, false))

assert(
  [false, "How did you vote?"],
  displayVote(17, true))

assert(
  [false, "You should consider voting.When you are older."],
  displayVote(17, false))
```

Voters are anyone 18 or older (one equivalent class)

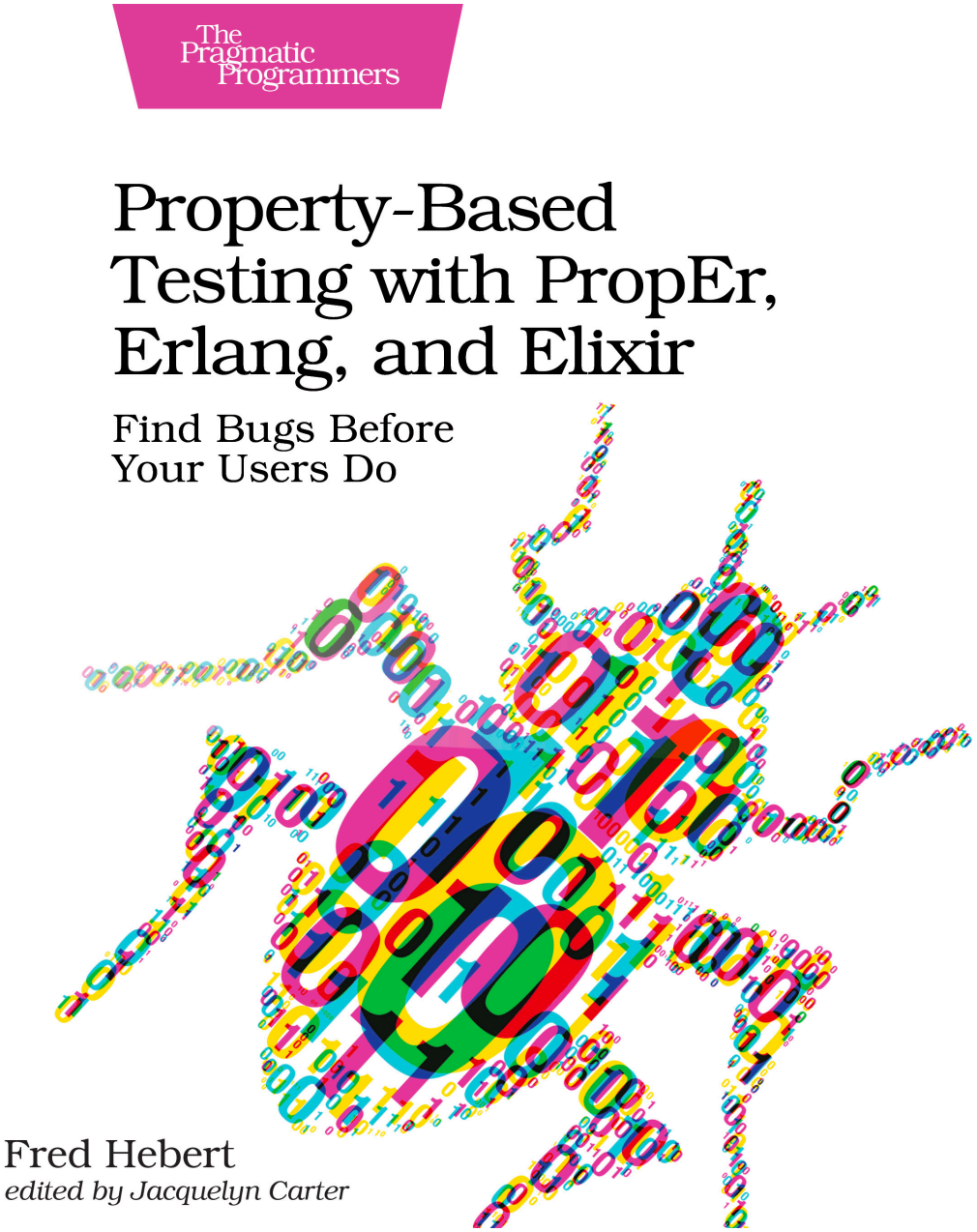
Under 18, you cannot yet vote. But, what about “negative” numbers

More examples, if valid input is... Eq. Classes are

- **A month number (1-12)**
  - [-∞..0]
  - [1..12]
  - [13.. ∞]
- **One of ten strings representing a type of fuel (enum)**
  - 10 classes, one for each string
  - A class representing all other strings

COMBINATIONS OF EQUIVALENCE CLASSES

- You cannot realistically test every possible combination of equivalence classes (**combinatorial explosion**)
- Consider **4 inputs, 5 possible values** each would result in **625** (4^5) possible combinations of eq. classes
- Instead, test…
  - Each Eq. Class separately
  - Combinations likely to affect interpretation
  - A few other random combinations


The Pragmatic Programmers
Property-Based Testing with PropEr, Erlang, and Elixir
Find Bugs Before Your Users Do
Fred Hebert
edited by Jacquelyn Carter

EXAMPLE EQUIVALENCE CLASS COMBINATIONS

- One valid input is either '**Metric**' or '**US/Imperial**, Equivalence classes are:
  - Metric,
  - US/Imperial,
  - Other
- Another valid input is maximum speed: **1 to 750 km/h** or **1 to 500 mph**
  - Validity depends on whether metric or US/imperial
  - [-∞..0]
  - [1..500]
  - [501..750]
  - [751.. ∞]

SOME TEST COMBINATIONS

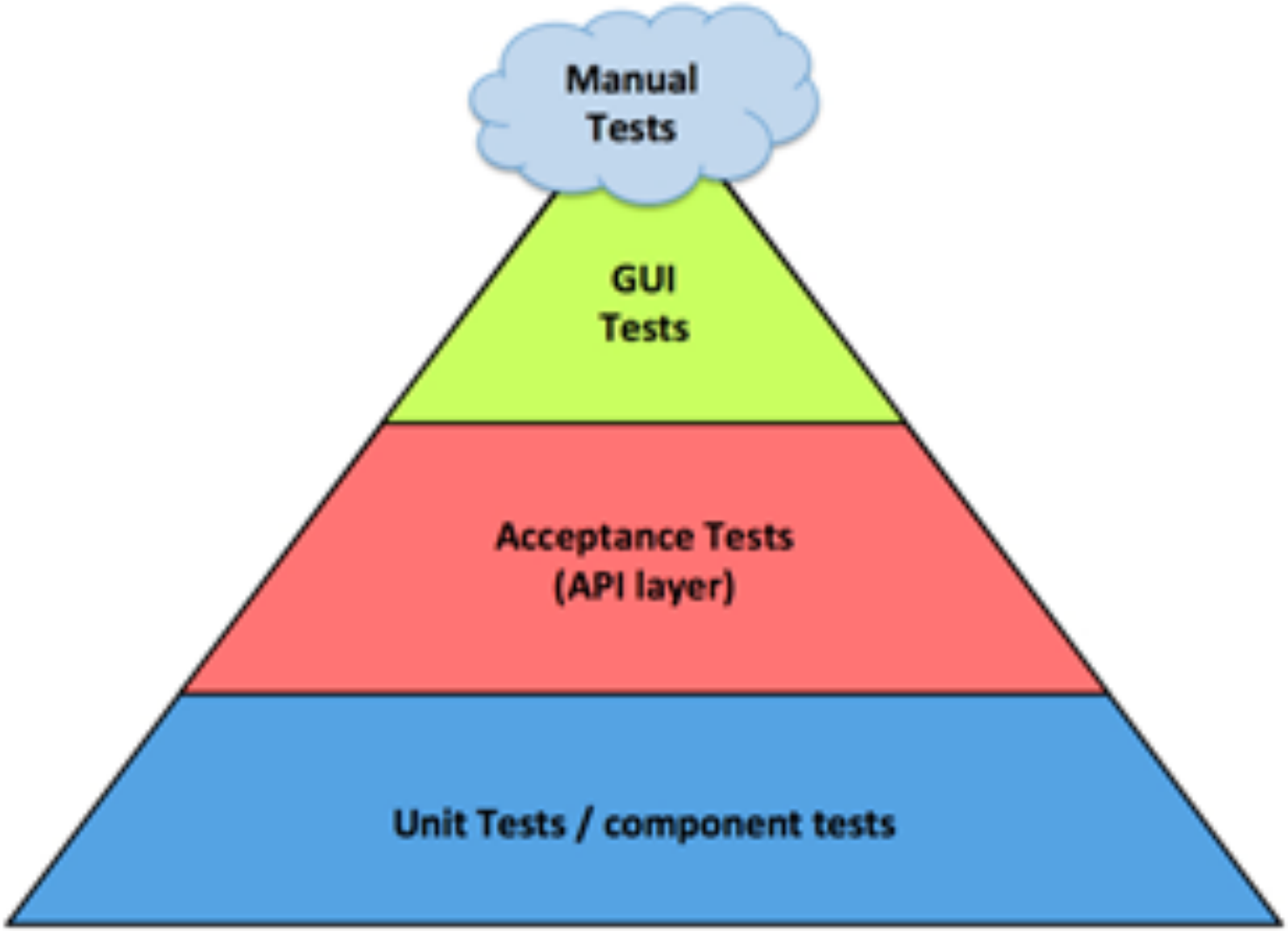
- Eq Class 1
  - Metric,
  - US/Imperial,
  - Other
- Eq Class 2
  - [-∞..0]
  - [1..500]
  - [501..750]
  - [751.. ∞]

1. Metric, [1..500] valid
2. US/Imperial, [1..500] valid
3. Metric, [501..750] valid
4. US/Imperial, [501..750] invalid

TESTING AND TOOLS

xUnit

Everyone Tests



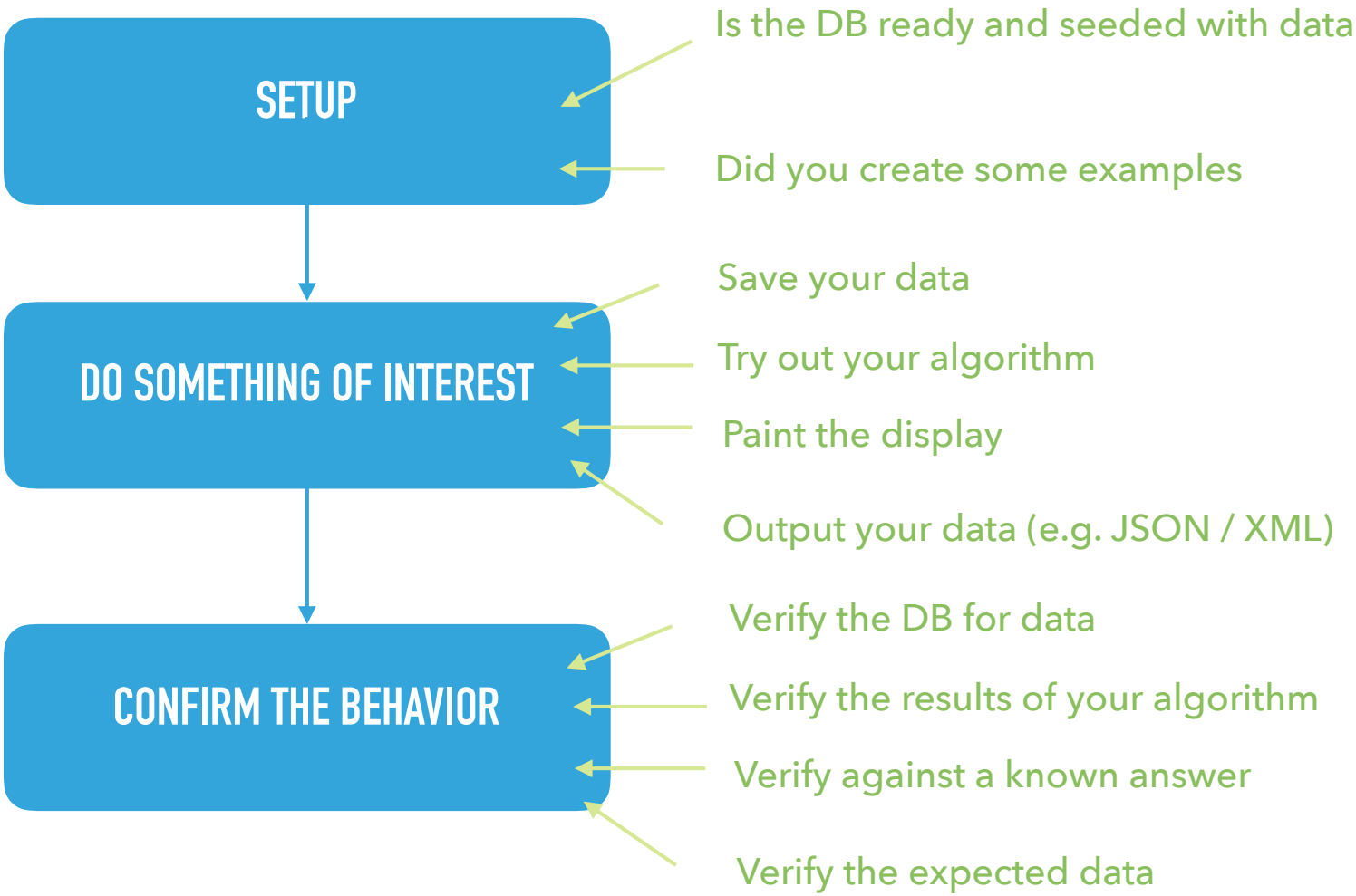
TESTING IS IT'S OWN SKILL

- Most (all?) junior developers at one time or another have said

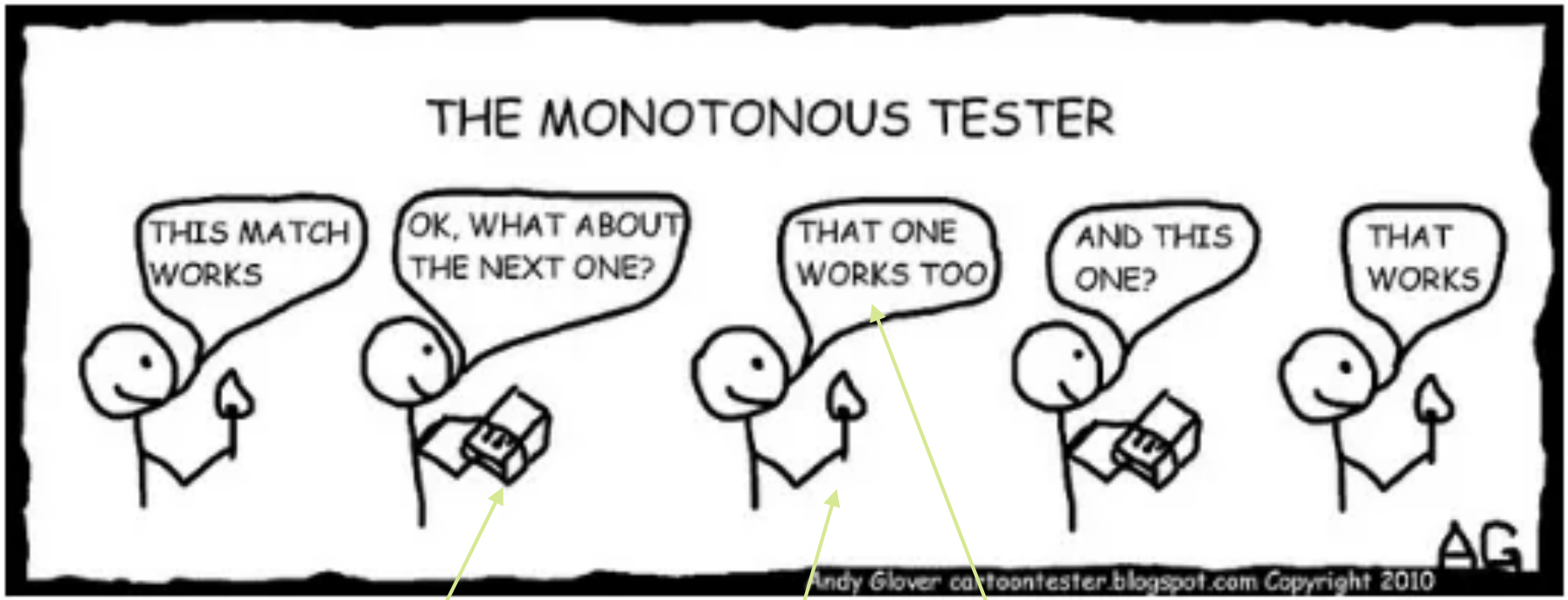
  > "Testing is so frustrating and hard and not worth the effort.
  > I don't want to do it"

- Think about how much you practice programming, versus practicing (automated testing) testing
  - Yes it is hard
  - Yes it is frustrating
  - And, yes you will be terrible at it
- That's not a valid reason and an excuse you should stop telling yourself

WHAT IS A TEST REALLY?



It's same when you are you manually testing



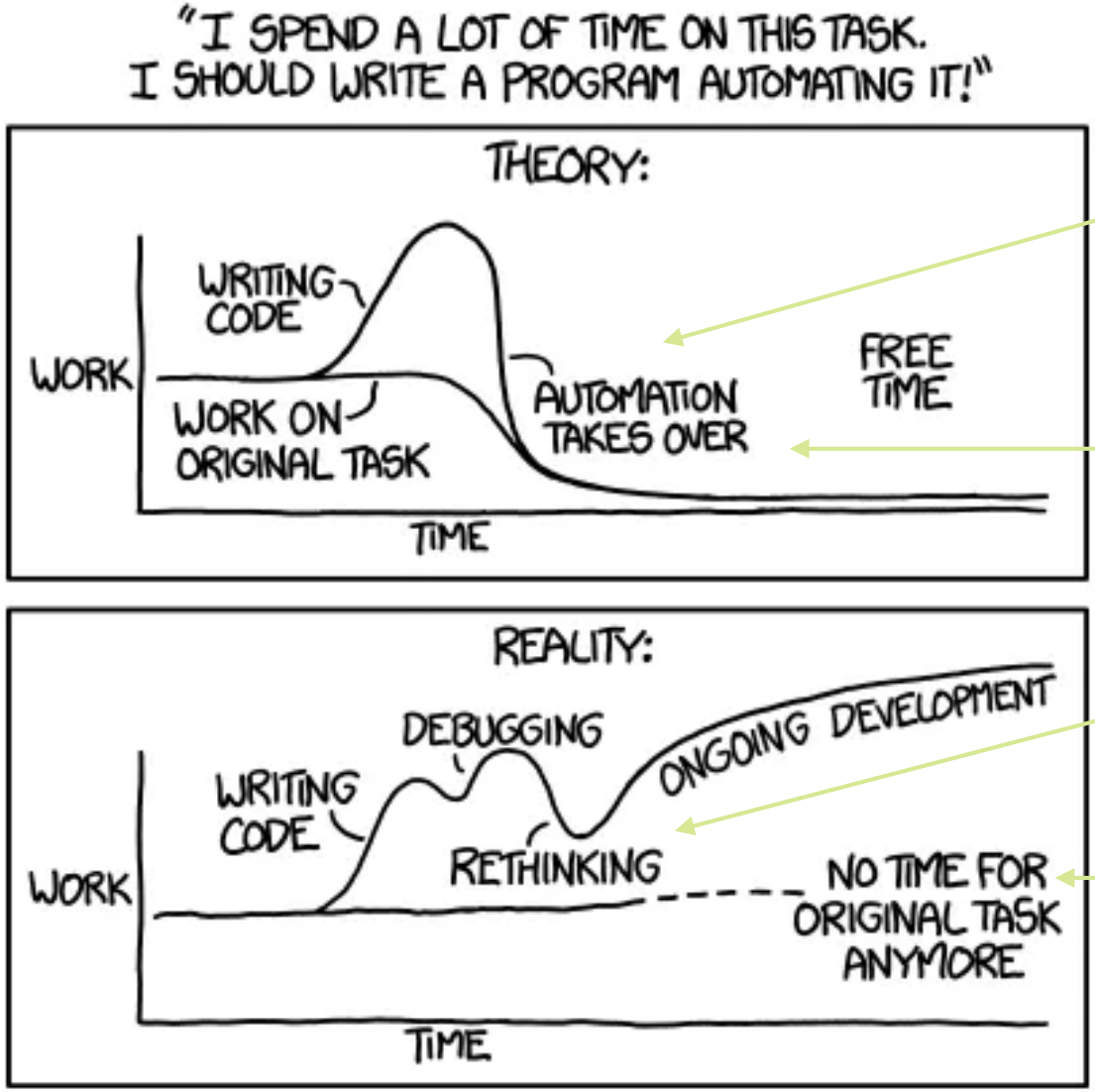
Setup stuff

Run Stuff

Confirm Stuff

TESTING AT BOUNDARIES OF EQUIVALENCE CLASSES

- More errors in software occur at the boundaries of equivalence classes
- So test values at the extremes of each equivalence class
  - E.g. The number 0 often causes problems
- If the valid input is a month number (1-12)
  - Test 0, 1, 12 and 13
  - Test very large positive and
  - Test very large negative values



It's totally possible

But testing is it's own competence

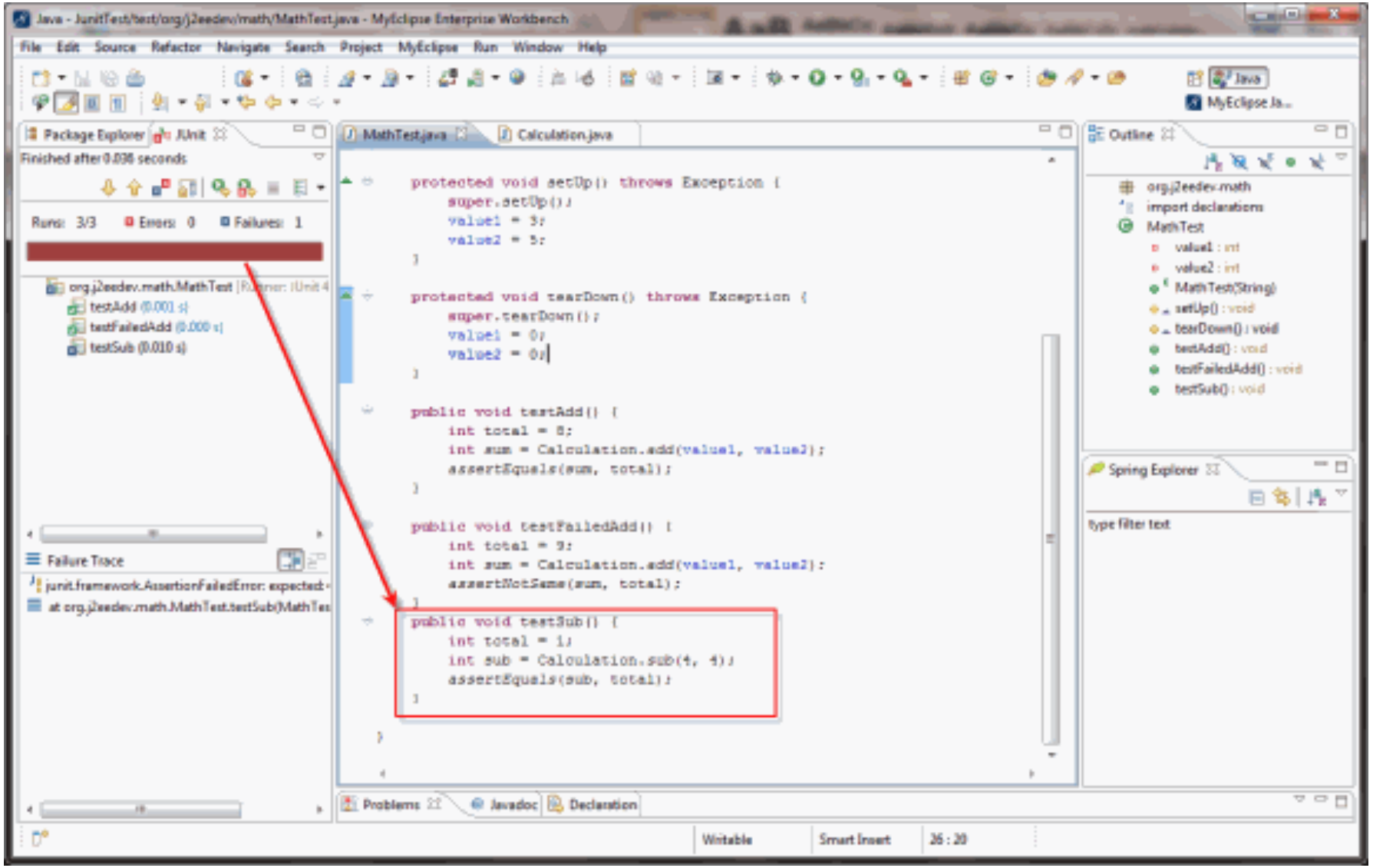
So it will hurt at the start

Don't abandon all hope, it will get better

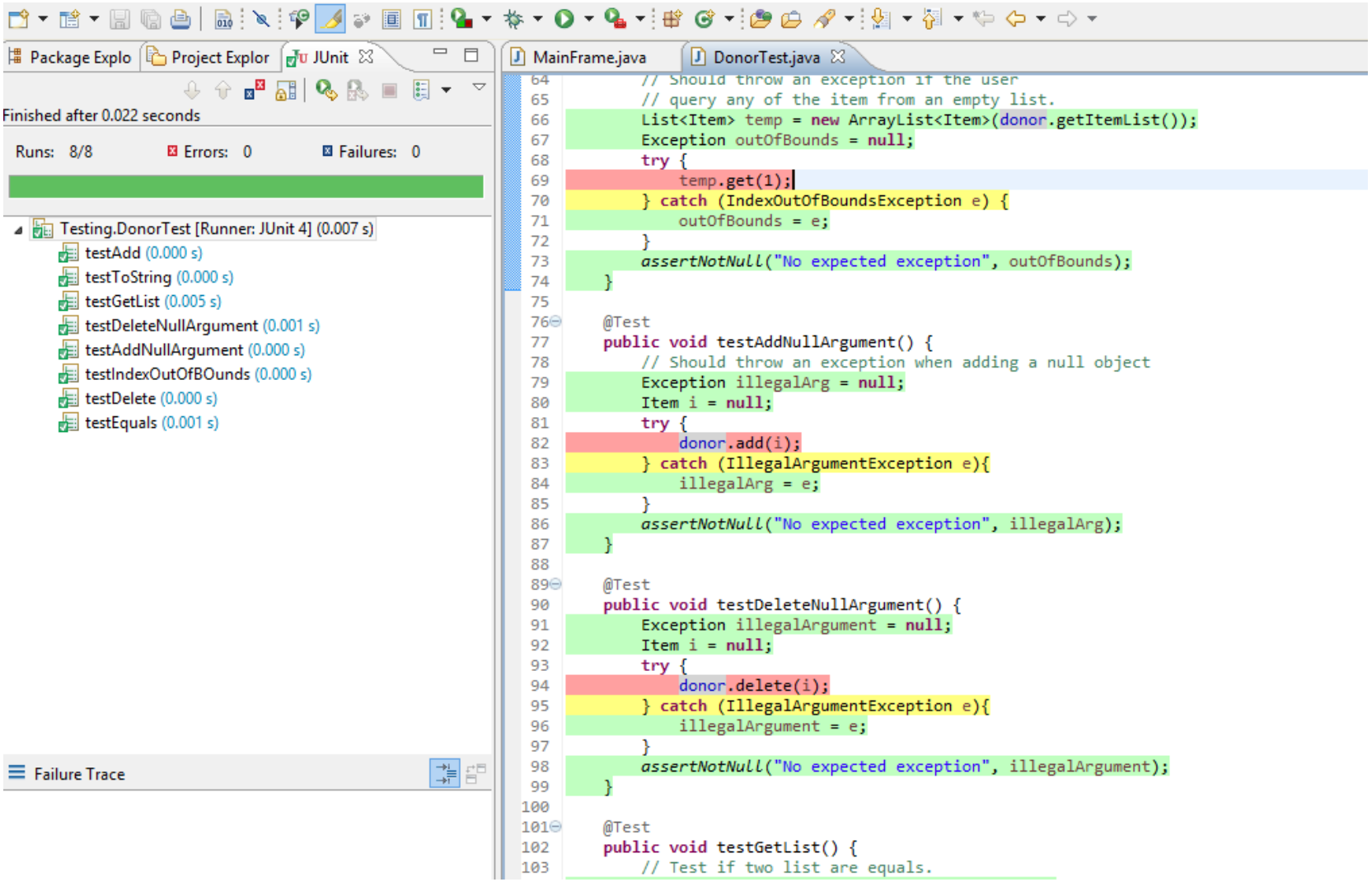
SEVERAL TESTING LIBRARIES

Not just for “unit” testing

JUNIT – A AUTOMATED TESTING FRAMEWORK (NOT JUST FOR UNIT TESTS)



JUnit + EclEmma



TESTING PHP

WITH PHPUNIT

PHPUnit Example

```
[10:56 ~/sin/projects/current/samplephp (master)$ phpunit --bootstrap tests/bootstrap.php tests/
PHPUnit 6.0.6 by Sebastian Bergmann and contributors.

...                                                                 3 / 3 (100%)

Time: 56 ms, Memory: 8.00MB

OK (3 tests, 3 assertions)
```

Command line tool

Run everything under "test" dir

. (pass)
F (fail)
E (exception)
S (skip)

OUR CLASS UNDER TEST

```php
<?php
namespace Samplephp;

class Message
{
    public static function hello($name = null)
    {
        $name = $name ?: "World";
        return "Hello {$name}";
    }
}
```

BOOTSTRAPPING

```php
<?php

function __autoload($fullName)
{
    $parts = explode("\\", $fullName);
    $len = count($parts);
    $className = $parts[$len - 1];
    if (file_exists("app/models/{$className}.php"))
    {
      require_once "app/models/{$className}.php";
    }
}
```

Boilerplate Test Case

```php
<?php
namespace Samplephp\Test;

use Samplephp\Message;
use PHPUnit\Framework\TestCase;

class MessageTest extends TestCase
{
    // see next slide

}
```

Actual Tests

TESTING RUBY

WITH RSPEC

RSpec Example

```
[08:33 ~/sin/projects/current/samplesinatra (master)$ rspec spec/
...

Finished in 0.00108 seconds (files took 0.09459 seconds to load)
3 examples, 0 failures
```

. (pass)
F (fail)
E (exception)
S (skip)

Command line tool

Run everything under "spec" dir

BOOTSTRAP


```
rspec --init
```

OUR CLASS UNDER TEST

```
class Message

  def self.hello(name = nil)
    name ||= "World"
    return "Hello #{name}"
  end

end
```

Boilerplate Test Case

```
require "message"

RSpec.describe Message, "#hello" do

  // See next slide

end
```

Actual Tests

Behaviour driven, so use "desribe"

Do something

Confirm behavior

Edge case

Success case

```
require "message"

RSpec.describe Message, "#hello" do

  it "should support no input" do
    expect(Message.hello).to eq "Hello World"
  end

  it "should convert nil to default" do
    expect(Message.hello(nil)).to eq "Hello World"
  end

  it "should say your name" do
    expect(Message.hello("Andrew")).to eq "Hello Andrew"
  end

end
```

expect(<actual>).to eq <expected>

TESTING ELIXIR

WITH EXUNIT

ExUnit Example

```
[13:09 ~/sin/projects/current/sampleplug (master)$ mix test
Compiling 1 file (.ex)
....

Finished in 0.03 seconds
4 tests, 0 failures
```

. (pass)
F (fail)
E (exception)
S (skip)

Command line tool
(for many things)

Run the “tests”

BOOTSTRAP

None really

OUR CLASS UNDER TEST

```
defmodule Sampleplug.Message do

  def hello, do: hello(nil)
  def hello(nil), do: "Hello World"
  def hello(name), do: "Hello #{name}"

end
```

Boilerplate Test Case

```
defmodule Sampleplug.MessageTest do
  use ExUnit.Case
  alias Sampleplug.Message
  doctest Sampleplug.Message

  // see next slide

end
```

Actual Tests

One of the best features ...
DocTest

```
defmodule Sampleplug.MessageTest do
  use ExUnit.Case
  alias Sampleplug.Message
  doctest Sampleplug.Message

  test "should support no input" do
    assert Message.hello == "Hello World"
  end

  test "should convert nil to default" do
    assert Message.hello(nil) == "Hello World"
  end

  test "should say your name" do
    assert Message.hello("Andrew") == "Hello Andrew"
  end

end
```

test block

Confirm behavior

Do something

Edge case

Success case

assert <actual> == <expected>

DOCUMENT ELIXIR

WITH DOCTEST

https://github.com/elixir-lang/ex_doc

EXAMPLE DRIVEN TESTING

Documentation works best

With examples

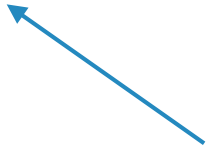
```
defmodule Sampleplug.Message do


  @doc ~S"""
  A welcome message, you can provide an optional name otherwise
  get a generic Hello World.

  ## Examples

      iex> Sampleplug.Message.hello
      "Hello World"

      iex> Sampleplug.Message.hello("James")
      "Hello James"
  """
  def hello, do: hello(nil)
  def hello(nil), do: "Hello World"
  def hello(name), do: "Hello #{name}"

end
```

THOSE ARE TESTS TOO!

```
[13:17 ~/sin/projects/current/sampleplug (master)$ mix test
Compiling 1 file (.ex)
..

  1) test doc at Sampleplug.Message.hello/0 (1) (Sampleplug.MessageTest)
     test/message_test.exs:4
     Doctest failed
     code: Sampleplug.Message.hello === "Bonjour Monde"
     lhs:  "Hello World"
     stacktrace:
       lib/sampleplug/message.ex:10: Sampleplug.Message (module)

...

Finished in 0.05 seconds
6 tests, 1 failure
```

AND THEY ARE DOCUMENTATION

```
13:23 ~/sin/projects/current/sampleplug (master)$ mix docs
Docs successfully generated.
View them at "doc/index.html".
```



Like JavaDocs, we can output our documentation to HTML

BUT, WE ARE CONFIDENT THE EXAMPLES ARE VALID!

Functions

> **hello()**

A welcome message, you can provide an optional name otherwise get a generic Hello World.

Examples

```
iex> Sampleplug.Message.hello
"Hello World"

iex> Sampleplug.Message.hello("James")
"Hello James"
```

> **hello(name)**

CONTINUOUS TESTING . . .

Run your tests whenever a file changes

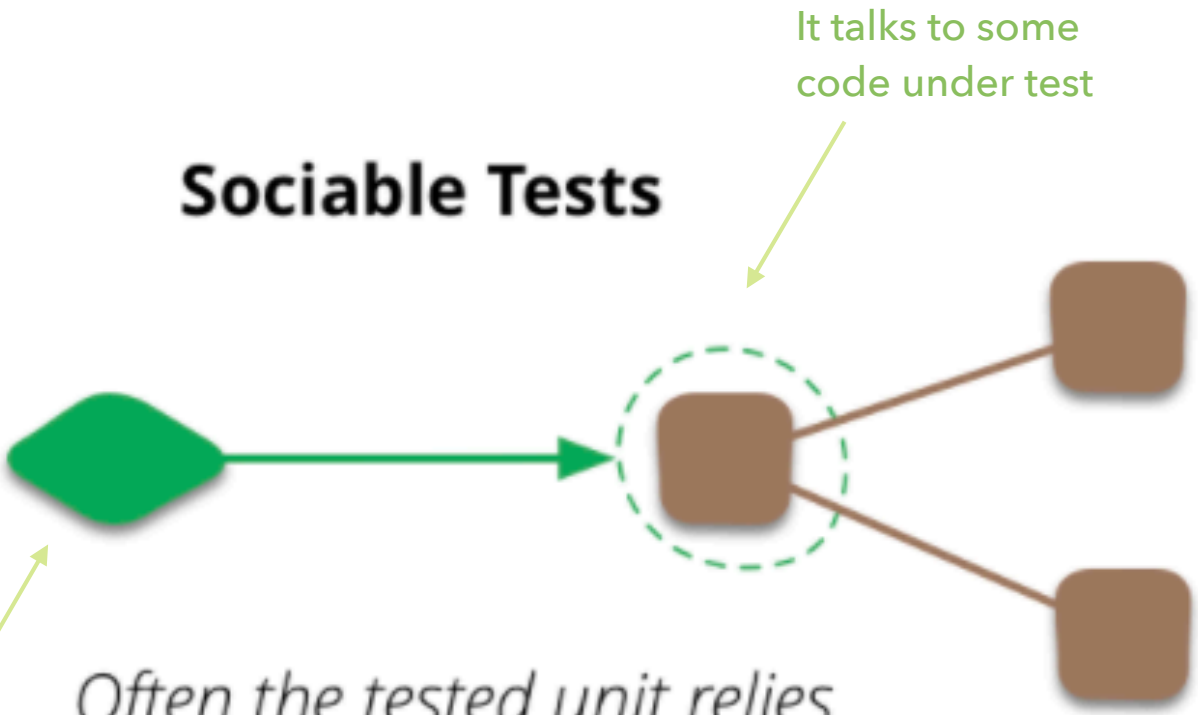
```
[^C13:50 ~/sin/projects/current/sampleplug (master)$ mix test.watch

Running tests...
......

Finished in 0.05 seconds
6 tests, 0 failures

Randomized with seed 27561

Running tests...
....

  1) test should should handle spaces (Sampleplug.MessageTest)
     test/message_test.exs:18
     Assertion with == failed
     code: Message.hello("Andrew Forward") == "Hello Andrew XXX"
     lhs:  "Hello Andrew Forward"
     rhs:  "Hello Andrew XXX"
     stacktrace:
       test/message_test.exs:19: (test)

..

Finished in 0.05 seconds
7 tests, 1 failure

Randomized with seed 852494

Running tests...
.......

Finished in 0.06 seconds
7 tests, 0 failures
```

Reduce the feedback loop during testing

TYPES OF (UNIT)

TESTING

Social Tests (Great if fast and isolated)

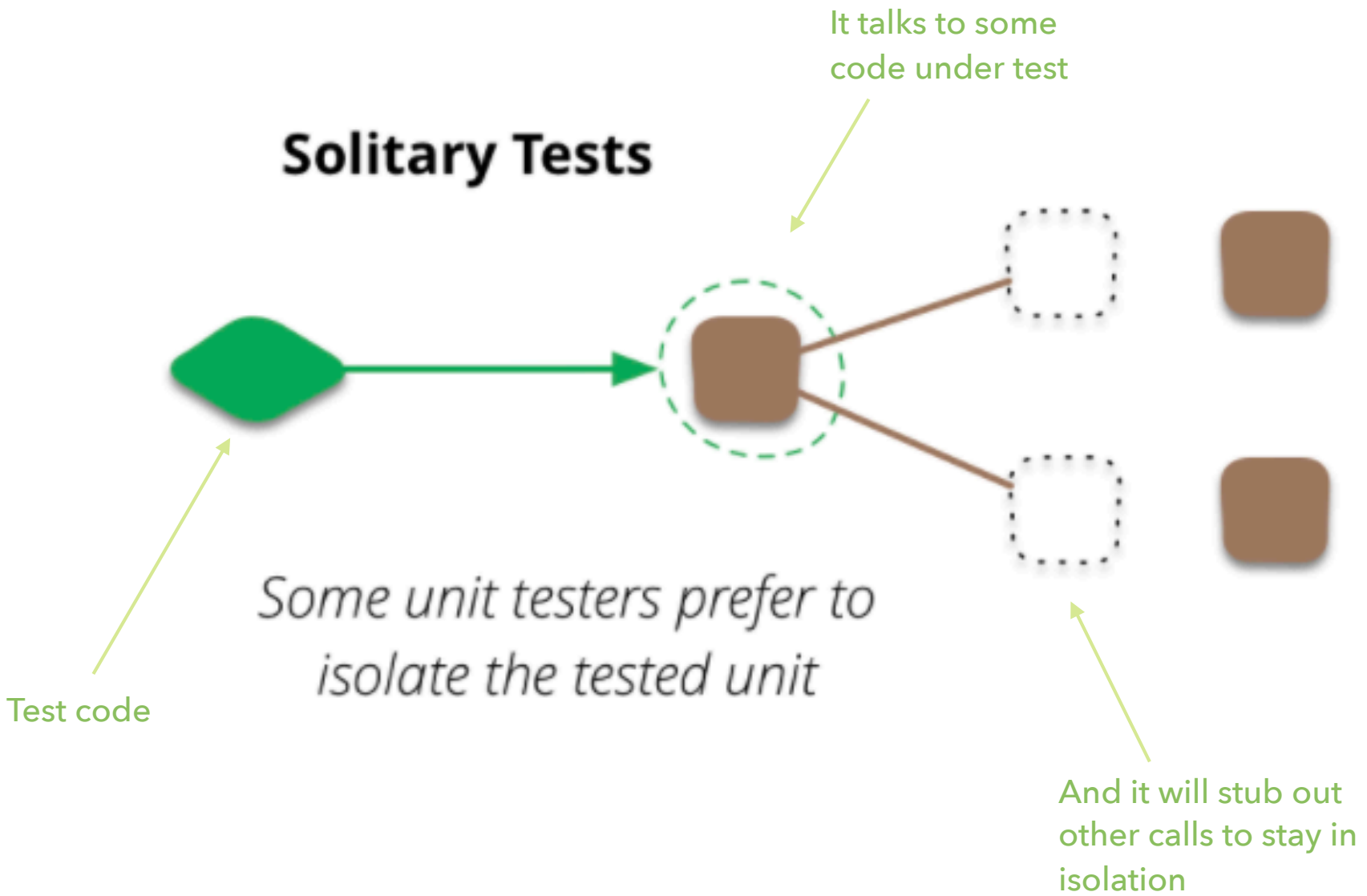


It talks to some code under test

Test code

But, it will also talk to other code

SOLITARY TESTS (GREAT IF OTHERWISE SLOW AND BRITTLE)

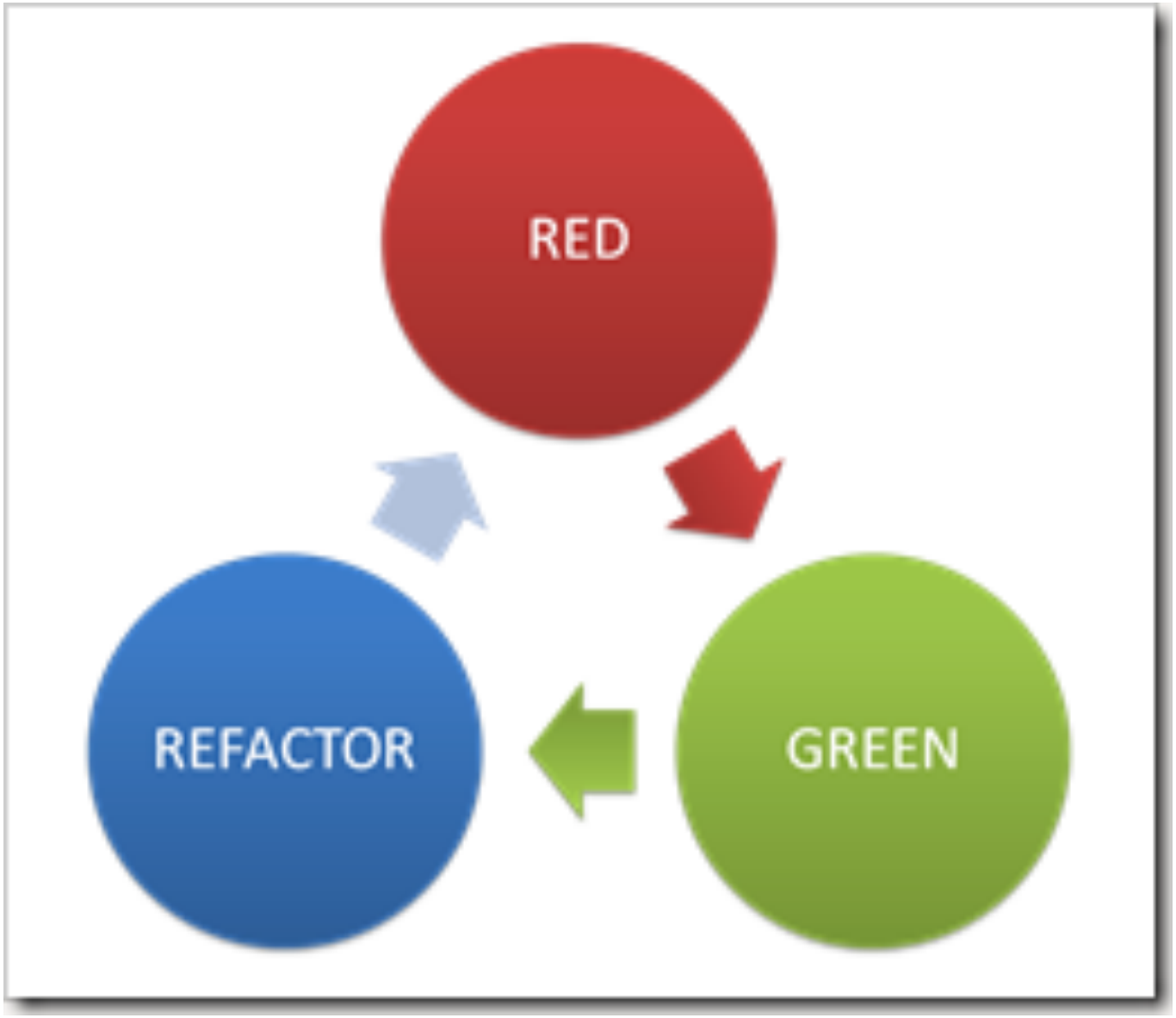


TEST DRIVEN DEVELOPMENT

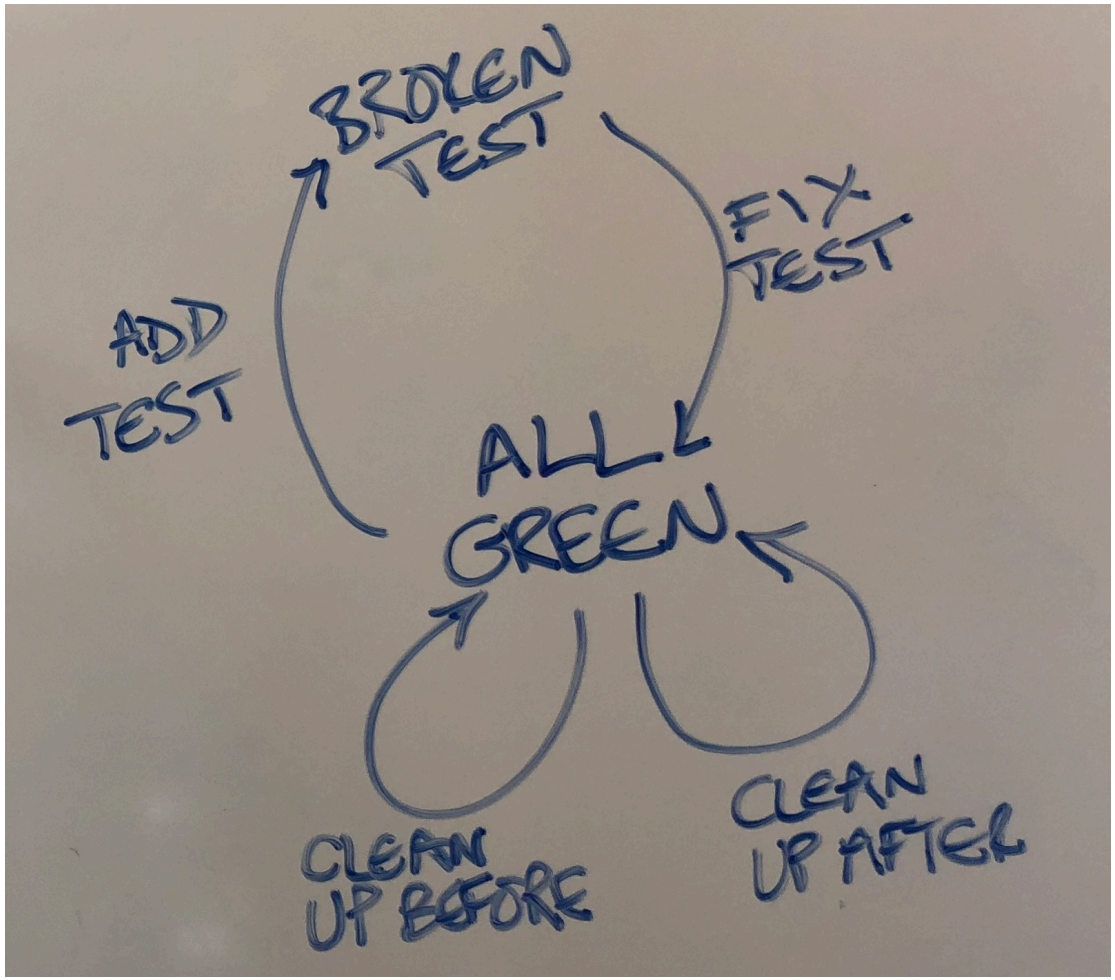
THE ORIGINAL "XDD"

COMMON PRACTICE (NOW)

- Write all tests as executable code **before** you write the code do do the actual function (i.e. test-first)
- Use tools such as
  - jUnit (testing framework),
  - Ant (or bash, or make, etc) to automate their execution
  - Github Actions (or Circle) to automate the timing of their execution
- Use the tests as actual specifications of the system
  - When you first write a test, make sure it fails
  - Then write the code for the functionality
  - Then make sure the test passes
  - Then make the code beautiful
- For testing user interfaces consider tools like
  - Selenium
  - Headless browsers

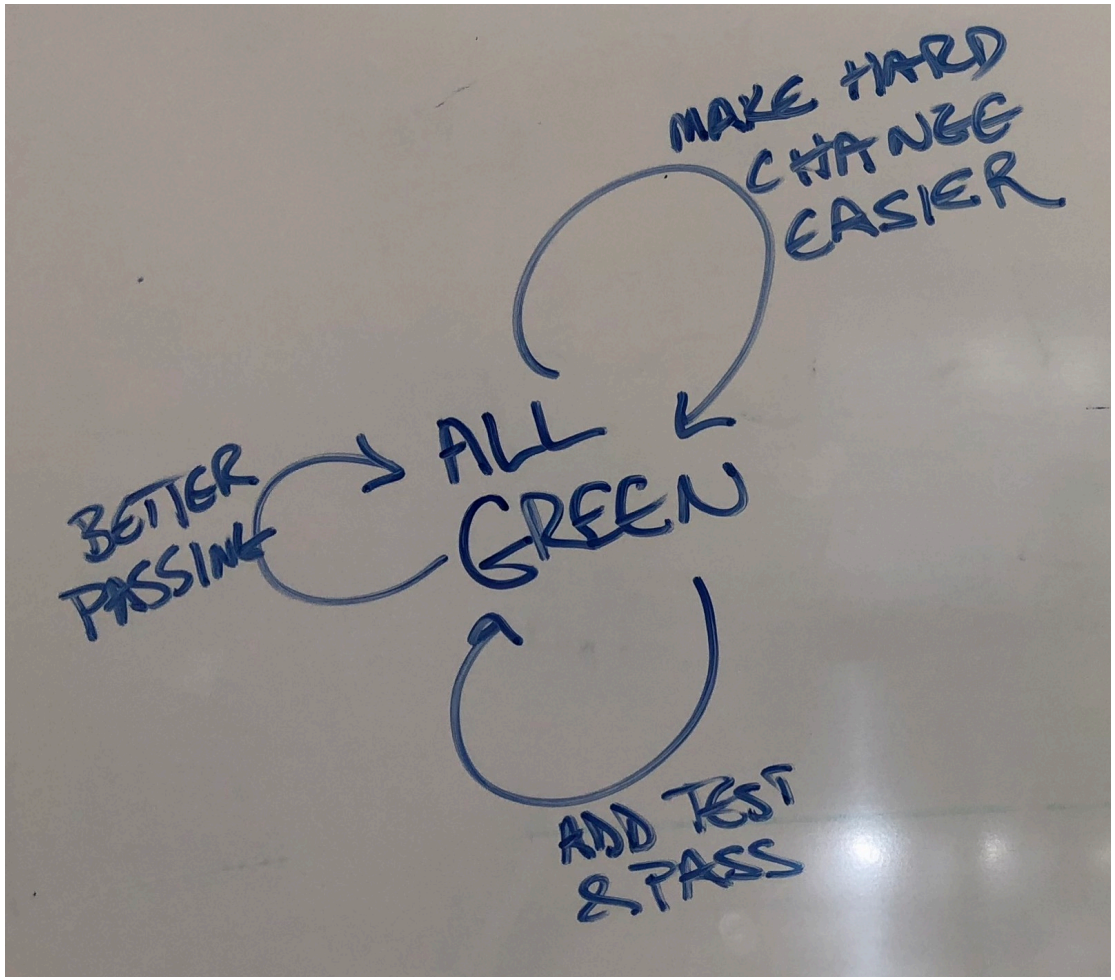

RED
GREEN
REFACTOR

Test Driven Development



https://medium.com/@kentbeck_7670/test-commit-revert-870bbd756864

TEST && COMMIT || REVERT



https://medium.com/@kentbeck_7670/test-commit-revert-870bbd756864

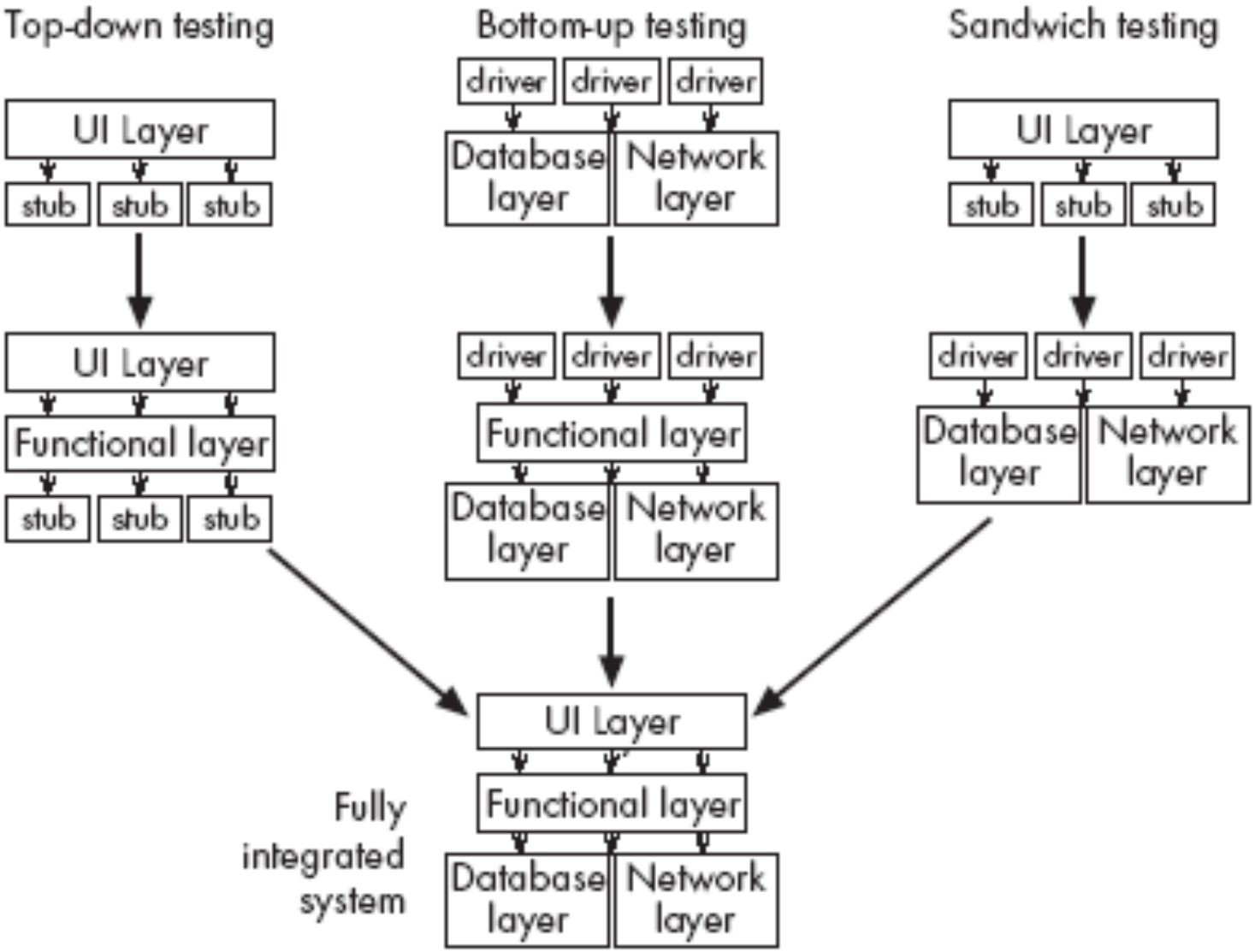
STRATEGIES FOR TESTING

LARGE SYSTEMS

BIG BANG TESTING VERSUS INTEGRATION TESTING

- In big bang testing,
  - you take the entire system and test it as a unit
- A better strategy in most cases is incremental testing:
  - You test each individual subsystem in isolation
  - Continue testing as you add more and more subsystems to the final product
  - Incremental testing can be performed horizontally or vertically, depending on the architecture
  - Horizontal testing can be used when the system is divided into separate sub-applications

VERTICAL STRATEGIES FOR INCREMENTAL INTEGRATION TESTING



Regression testing

- When testing manually
  - Test a well-chosen subset of the previously-successful test cases
- When testing automatically
  - Run all tests, since it is generally fast

THE "LAW OF CONSERVATION OF BUGS"

- The number of bugs remaining in a large system is proportional to the number of bugs already fixed

DIFFICULTIES AND RISKS IN

QUALITY ASSURANCE

MORE TO SOFTWARE THAN CODE

It is very easy to forget to test some aspects of a software system:

- 'running the code a few times' is not enough.
- Forgetting certain types of tests diminishes the system's quality.

Quality versus Delivery

There is a conflict between achieving adequate quality levels, and 'getting the product out of the door'

- Create a separate department to oversee QA.
- Publish statistics about quality.
- Build adequate time for all activities.

KNOWLEDGE VARIES

People have different abilities and knowledge when it comes to quality

- Give people tasks that fit their natural personalities.
- Train people in testing and inspecting techniques.
- Give people feedback about their performance in terms of producing quality software.
- Have developers and maintainers work for several months on a testing team.

SEG 3103 – Software Quality Assurance

- Quality: how to assure it and verify it, and the need for a culture of quality. Avoidance of errors and other quality problems. Inspections and reviews. Testing, verification and validation techniques. Process assurance vs. Product assurance. Quality process standards. Product and process assurance. Problem analysis and reporting.

RÉFÉRENCES

- http://www.agilerecord.com/test-driven-development-good-bad/
- https://medium.com/planet-arkency/testing-is-a-separate-skill-and-thats-why-you-are-frustrated-7239b1500a6c#.34l1nhk7m
- http://www.provartesting.com/blog/what-is-test-automation-in-salesforce-and-why-does-it-matter/
- https://martinfowler.com/bliki/UnitTest.html