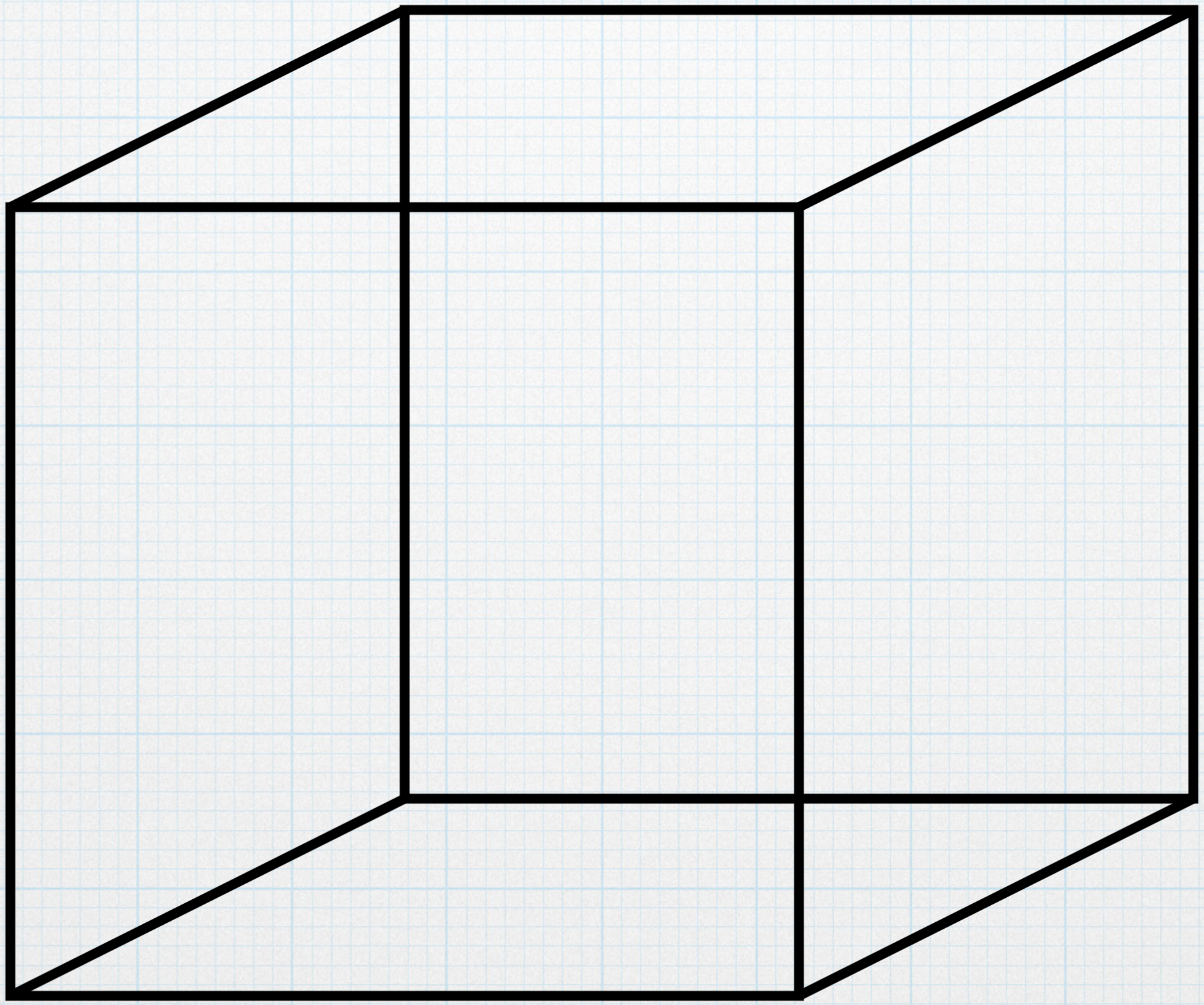
The Three Rules of TDD are Useful and Incomplete

Burk Hufnagel - Solution Architect
Daugherty Business Solutions

Is this talk for you?

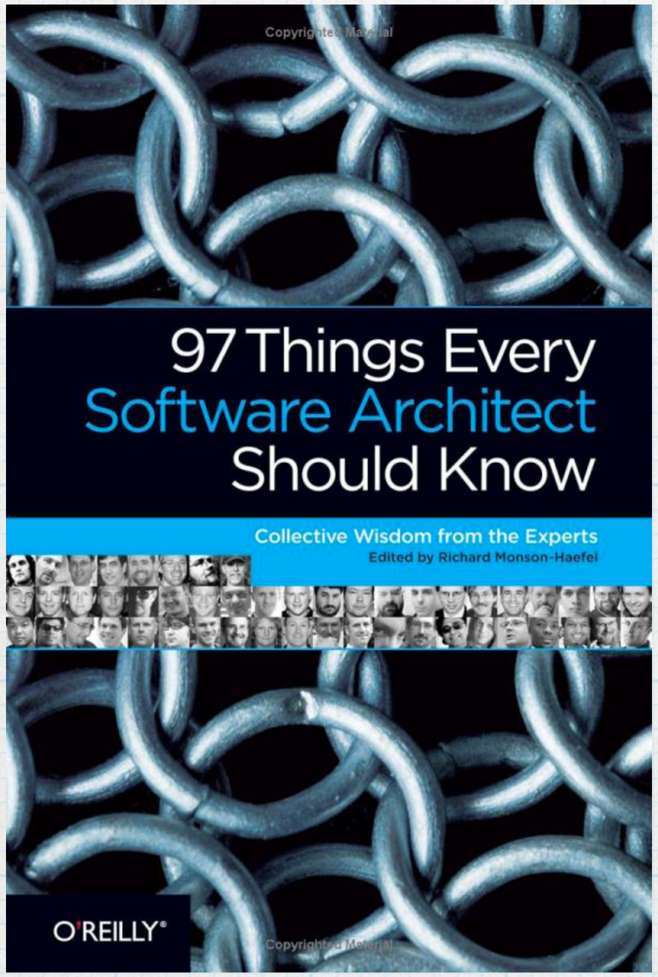
1. Are you a developer?
2. Do you want to deliver better code faster than you do now?
3. Are you willing to change your development process?



Who is Burk Hufnagel?

Java™ Champions

JavaOne Rock Star

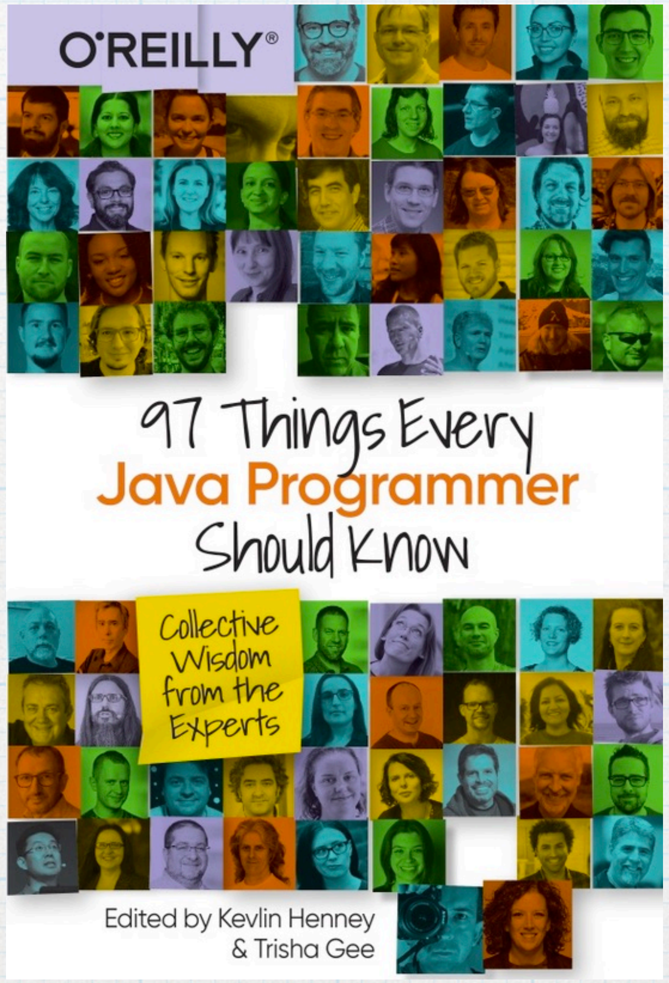


97 Things Every Software Architect Should Know

Collective Wisdom from the Experts

Edited by Richard Monson-Haefel

O'REILLY®


O'REILLY®

97 Things Every Java Programmer Should Know

Collective Wisdom from the Experts

Edited by Kevlin Henney & Trisha Gee

97

Collective Wisdom from the Experts

97 Things Every Programmer Should Know

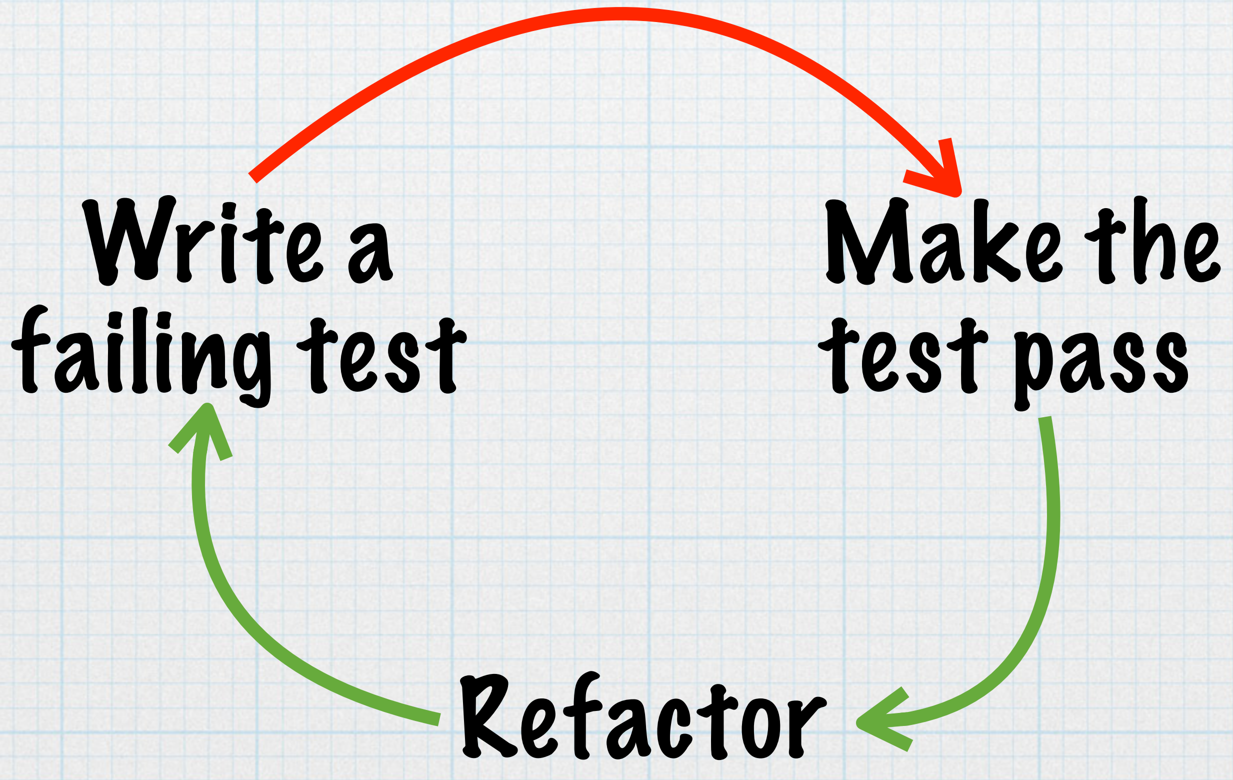
O'REILLY®

Edited by Kevlin Henney

Test-Driven Development

TDD is an efficient, three step process

* Write a failing automated test
* Write production code to make it pass
* Refactor to clean up any mess



TDD Benefits

Deliver Better Code, Faster

TDD Benefits

- Deliver
  - Writing code doesn't create business value
- Better Code
  - Solving the business problem is not enough
- Faster
  - Less time debugging, passing QA and UAT

The Three Rules of TDD

1. Only write production code when a test fails due to the lack of that code.
2. Write just enough test code to fail — and not compiling counts as a failure.
3. Write just enough production code to pass the failing test.

from "Clean Agile"

Why are They Useful?

1. Constantly switching between writing test and production code helps you break your current habit and learn the TDD way instead.
2. Encourages better API design, so your code is easier to maintain and use.

Why are They Incomplete?

1. They only cover thee first two steps of TDD. No mention of refactoring!
2. Doesn't help you know what tests to write or when you're done.

The New Rules of TDD

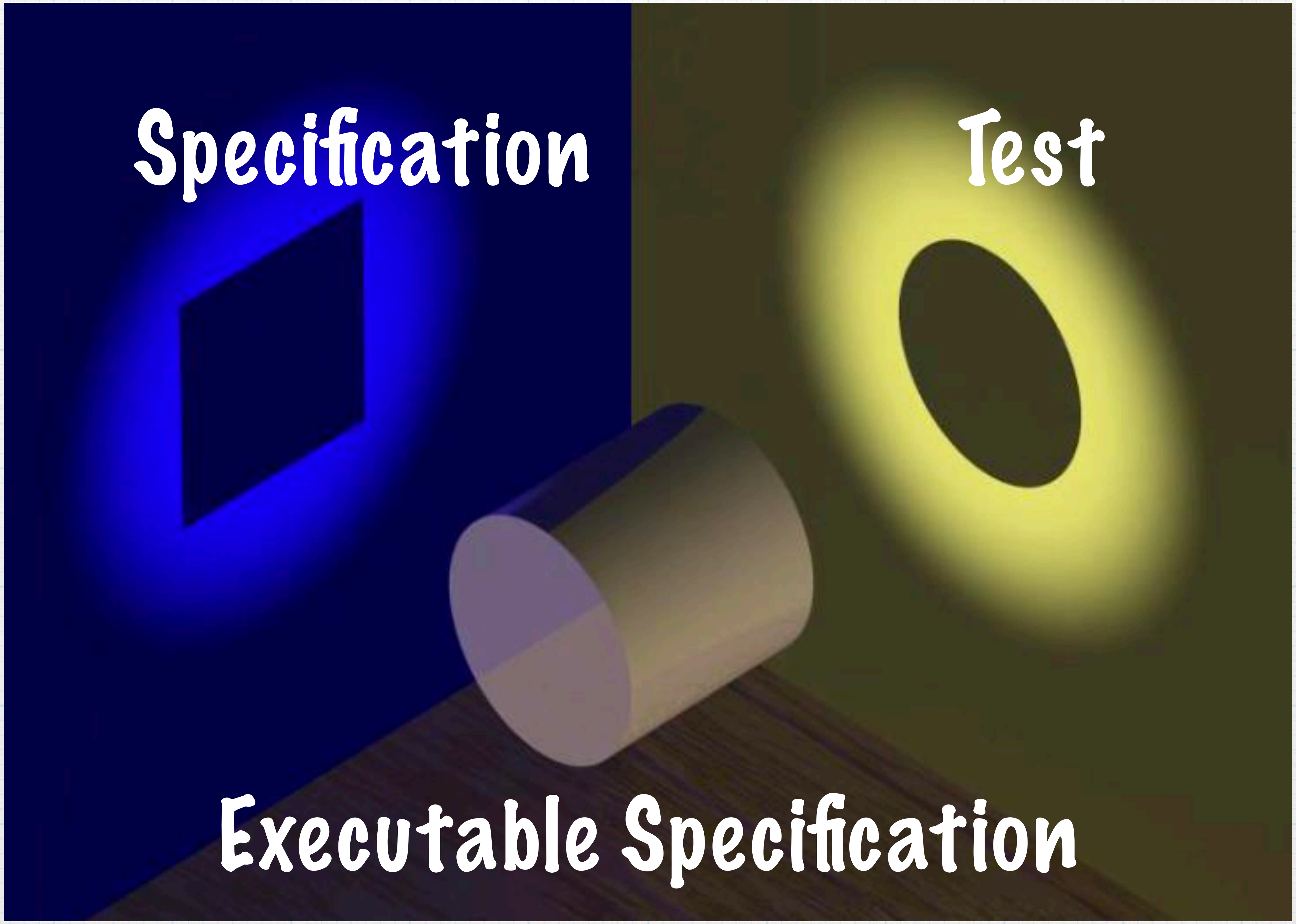
0) Only write a spec when there is an unmet requirement.

1) Write just enough of the spec to fail; not running counts as failing.

2) Write just enough production code to make the spec pass.

3) Refactor to clean up any mess.

Executable Specification



A Poor Automated Test

```
class StackTest {

   @Test
   void testPushPop() {
         Stack stack = new Stack();
         stack.push(42);
         stack.push(23);

         int x = stack.pop();

Critical! →  assertEquals(23, x);
   }
}
```

A Better Automated Test

```
class StackTest {

  @Test
  void callingPopShouldReturnTheLastValuePushed() {
      // Given 42 and 23 are pushed onto an empty Stack
      Stack stack = new Stack();
      stack.push(42);
      int expectedValue = 23;
      stack.push(expectedValue);

      // When pop() is called
      int actualValue = stack.pop();

      // Then it should return 23
      assertEquals(expectedValue, actualValue);
  }
}
```

An Executable Specification

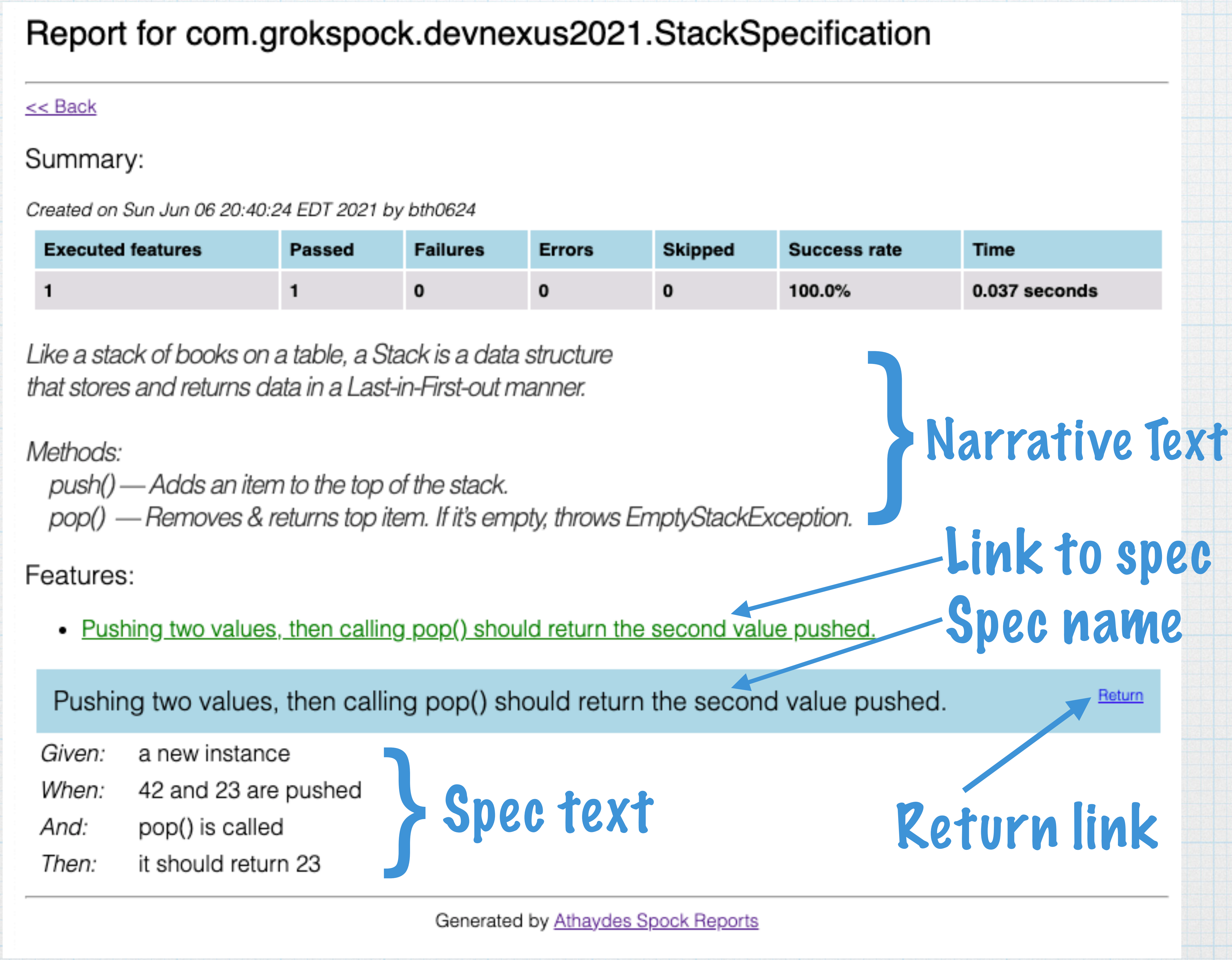
```
class StackSpec extends Specification {
    def "Pushing two values, then calling pop() should return" +
        "the second value pushed."() {
        given: "a new instance"
            def stack = new Stack();
        when: "42 and 23 are pushed"
            stack.push(42);
            stack.push(23);
        and: "pop() is called"
            def returnedValue = stack.pop();
        then: "it should return 23"
            returnedValue == 23;
    }
}
```

An Executable Specification

```
@Narrative("""
Like a stack of books on a table, a Stack is a data structure
that stores and returns data in a Last-in-First-out manner.

Methods:
  push() – Adds an item to the top of the stack.
  pop()  – Removes & returns top item. If it’s empty, throws EmptyStackException.
""")

class StackSpec extends Specification {
    def "Pushing two values, then calling pop() should return" +
        "the second value pushed."() {
        given: "a new instance"
            def stack = new Stack();
        when: "42 and 23 are pushed"
            stack.push(42);
            stack.push(23);
        and: "pop() is called"
            def returnedValue = stack.pop();
        then: "it should return 23"
            returnedValue == 23;
    }
}
```



Report for com.grokspock.devnexus2021.StackSpecification

<< Back

Summary:

Created on Sun Jun 06 20:40:24 EDT 2021 by bth0624

Executed features	Passed	Failures	Errors	Skipped	Success rate	Time
1	1	0	0	0	100.0%	0.037 seconds

Like a stack of books on a table, a Stack is a data structure
that stores and returns data in a Last-in-First-out manner.

Methods:
push() — Adds an item to the top of the stack.
pop() — Removes & returns top item. If it's empty, throws EmptyStackException.

Features:

- Pushing two values, then calling pop() should return the second value pushed.

Pushing two values, then calling pop() should return the second value pushed. Return

Given:	a new instance
When:	42 and 23 are pushed
And:	pop() is called
Then:	it should return 23

Generated by Athaydes Spock Reports

Live Long and Prosper

Resources

“Clean Code: A Handbook of Agile Software Craftsmanship” by Robert C. Martin

“Growing Object-Oriented Software, Guided by Tests” by Steve Freeman and Nat Pryce

“Specification By Example” by Gojko Adzic

“BDD in Action” by John Ferguson Smart