Dispelling the Myth of Code Coverage

How to get tests you can *really* rely on

Jeremy Cook

About me

About me

Canadian with English accent/Englishman with Canadian passport

About me

Canadian with English accent/Englishman with Canadian passport

Recovering musician, current software developer

About me

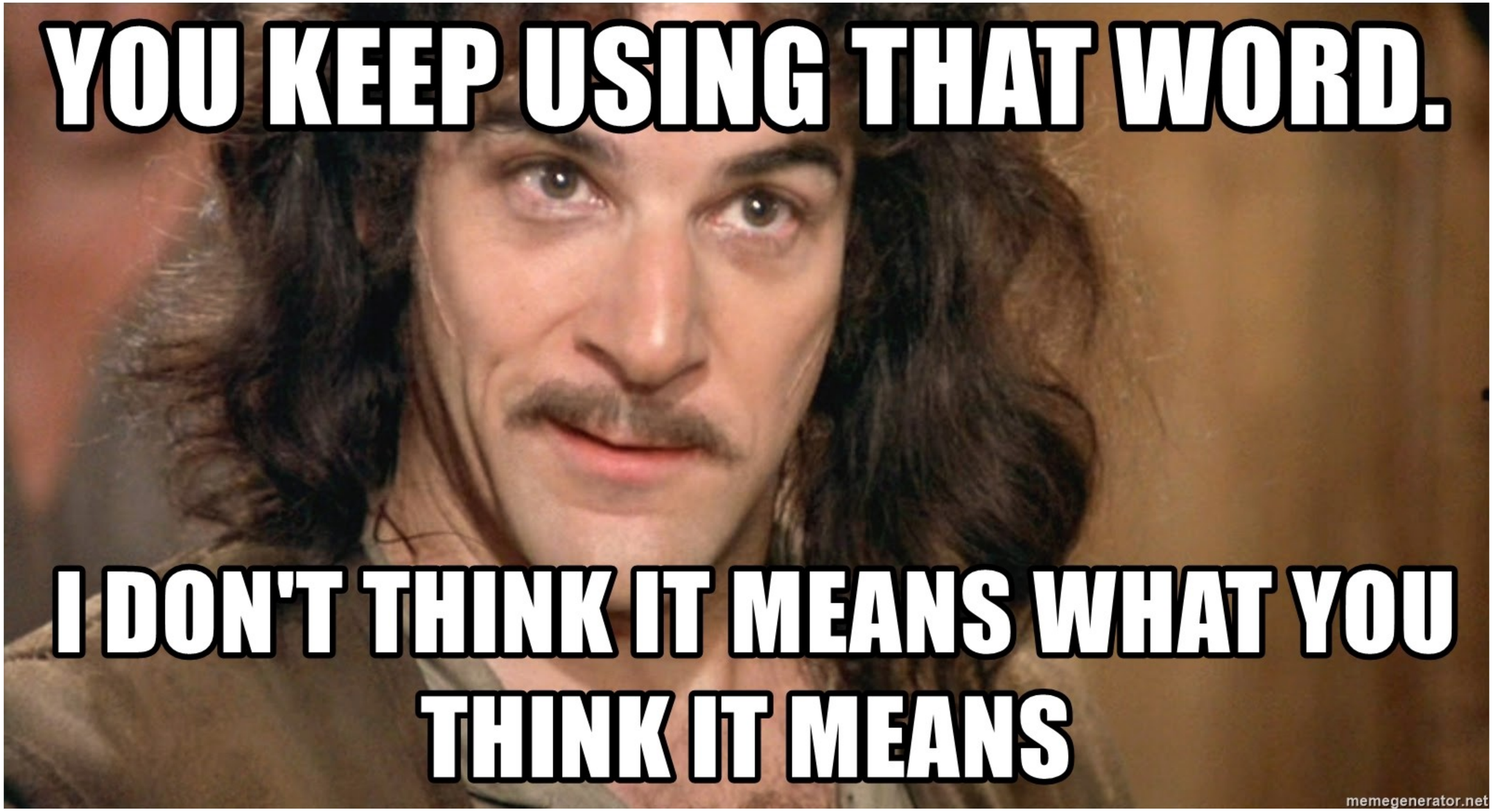
Canadian with English accent/Englishman with Canadian passport

Recovering musician, current software developer

Senior Principal Developer at Oracle

The Myth of Code Coverage…

I ❤️ code coverage


YOU KEEP USING THAT WORD.
I DON'T THINK IT MEANS WHAT YOU THINK IT MEANS
memegenerator.net

High Code Coverage Guarantees Valuable Tests*

High Code Coverage Guarantees Valuable Tests*

* this is a myth

What do we want from our tests?

If the observable behaviour of the code changes, a test should fail

Code coverage only tells us what lines are (not) executed by our tests

It’s laughably easy to get high coverage with tests that don’t fail when behaviour changes*

It’s laughably easy to get high coverage with tests that don’t fail when behaviour changes*

*This is a fact

High Coverage, Bad Tests

High Coverage, Bad Tests

```
function greaterOrEqualToTen(number: number): boolean {
    return number >= 10;
}

export default greaterOrEqualToTen;
```

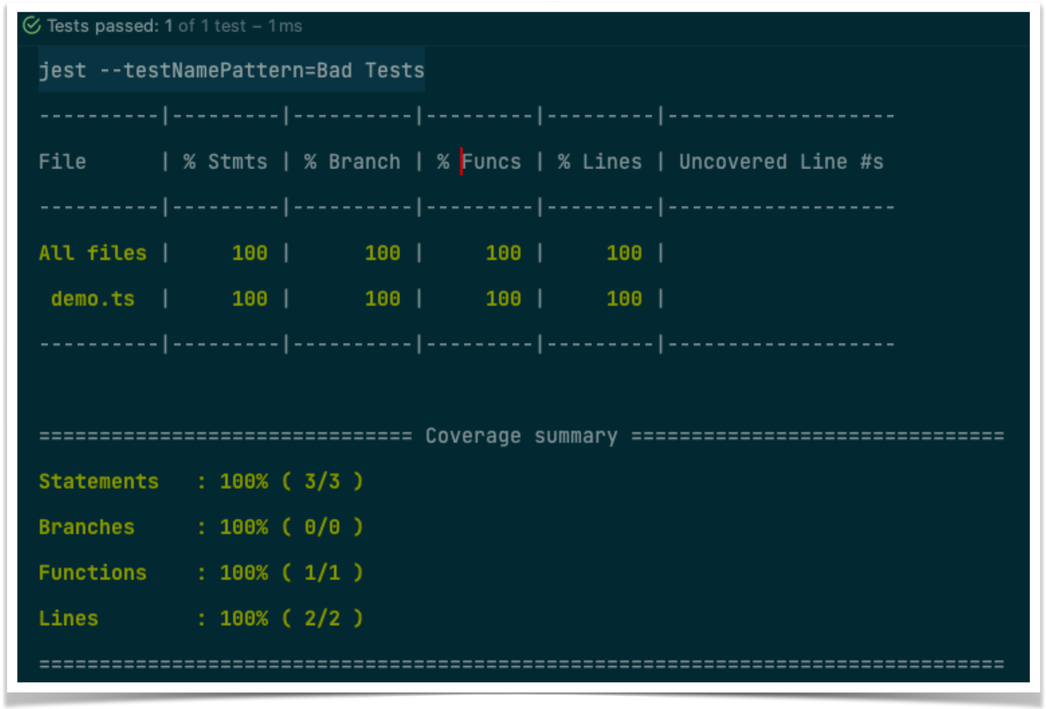
High Coverage, Bad Tests

High Coverage, Bad Tests

```
import greaterOrEqualToTen from "../src/demo";

describe('Bad Tests', () => {
    it('will never fail', () => {
        greaterOrEqualToTen(11);

        expect(true).toBe(true);
    });
});
```

High Coverage, Bad Tests

```
import greaterOrEqualToTen from "../src/demo";

describe('Bad Tests', () => {
    it('will never fail', () => {
        greaterOrEqualToTen(11);

        expect(true).toBe(true);
    });
});
```



High Coverage, Bad Tests

```
import greaterOrEqualToTen from "../src/demo";

describe('Bad Tests', () => {
    it('will never fail', () => {
        greaterOrEqualToTen(11);
    });
});
```

High Coverage, Bad Tests

```
import greaterOrEqualToTen from "../src/demo";

describe('Bad Tests', () => {
    it('will never fail', () => {
        greaterOrEqualToTen(11);
    });
});
```

```
jest --testNamePattern=More bad tests
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 demo.ts  |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------


============================== Coverage summary ==============================
Statements   : 100% ( 2/2 )
Branches     : 100% ( 0/0 )
Functions    : 100% ( 1/1 )
Lines        : 100% ( 2/2 )
==============================================================================
```

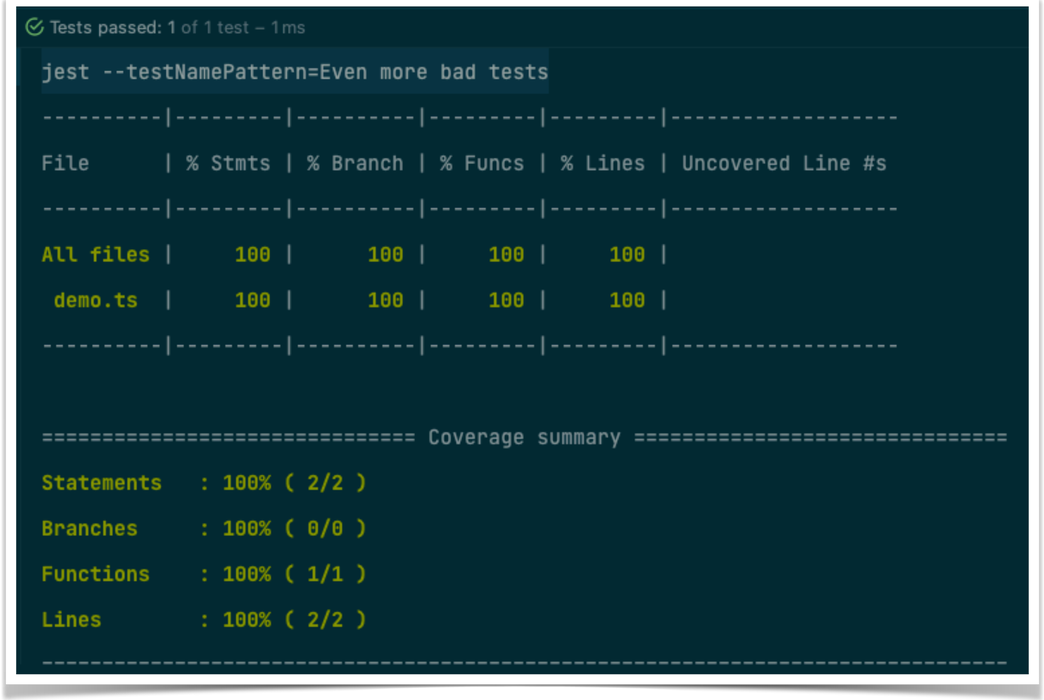
High Coverage, Bad Tests

High Coverage, Bad Tests

```
import greaterOrEqualToTen from "../src/demo";

describe('Even more bad tests', () => {
    it('has 100% coverage when missing a boundary condition', () => {
        const lessThanTenResult = greaterOrEqualToTen(9);
        const greaterThanTenResult = greaterOrEqualToTen(11);

        expect(lessThanTenResult).toBe(false);
        expect(greaterThanTenResult).toBe(true);
    });
});
```

High Coverage, Bad Tests

```
import greaterOrEqualToTen from "../src/demo";

describe('Even more bad tests', () => {
    it('has 100% coverage when missing a boundary condition', () => {
        const lessThanTenResult = greaterOrEqualToTen(9);
        const greaterThanTenResult = greaterOrEqualToTen(11);

        expect(lessThanTenResult).toBe(false);
        expect(greaterThanTenResult).toBe(true);
    });
});
```



High Coverage, Bad Tests

High Coverage, Bad Tests

```
public static String foo(boolean b) {
  if (b) {
    performVitallyImportantBusinessFunction();
    return "OK";
  }

  return "FAIL";
}
```

High Coverage, Bad Tests

```
public static String foo(boolean b) {
  if (b) {
    performVitallyImportantBusinessFunction();
    return "OK";
  }

  return "FAIL";
}

@Test
public void shouldFailWhenGivenFalse() {
  assertEquals("FAIL", foo(false));
}

@Test
public void shouldBeOkWhenGivenTrue() {
  assertEquals("OK", foo(true));
}
```

High Coverage, Bad Tests

High Coverage, Bad Tests

```
public static String foo(Collaborator c, boolean b) {
  if (b) {
      return c.performAction();
  }

  return "FOO";
}
```

High Coverage, Bad Tests

```
public static String foo(Collaborator c, boolean b) {
  if (b) {
      return c.performAction();
  }

  return "FOO";
}

@Test
public void shouldPerformActionWhenGivenTrue() {
  foo(mockCollaborator,true);
  verify(mockCollaborator).performAction();
}

@Test
public void shouldNotPerformActionWhenGivenFalse() {
  foo(mockCollaborator,false);
  verify(never(),mockCollaborator).performAction();
}
```

Why is this so bad?

Why is this so bad?

What happens when someone comes to edit the code?

Why is this so bad?

What happens when someone comes to edit the code?

Regressions that are hard to detect

Why is this so bad?

What happens when someone comes to edit the code?

Regressions that are hard to detect

Expensive to fix

Why is this so bad?

What happens when someone comes to edit the code?

Regressions that are hard to detect

Expensive to fix

More code executed, less meaningful code coverage is

Great Tests Enable Fearless Change

Great Tests Enable Fearless Change

Essential Complexity vs. **Accidental Complexity**

Great Tests Enable Fearless Change

Essential Complexity vs. **Accidental Complexity**

Technical debt: not removing accidental complexity

Great Tests Enable Fearless Change

Essential Complexity vs. **Accidental Complexity**

Technical debt: not removing accidental complexity

Need fast, reliable tests to remove accidental complexity

“The true test of good code is how easy it is to change it”

Martin Fowler, *Refactoring*

Techniques to get Valuable Tests

Mutation Testing

Have you ever changed some code to make sure a test can fail?

Mutation Testing Basics

Mutation Testing Basics

Seed changes, run tests

Mutation Testing Basics

Seed changes, run tests

Verify tests will fail

Mutation Testing Basics

Seed changes, run tests

Verify tests will fail

Test failures are good!

Mutation Testing Frameworks

Pit	Java	http://pitest.org
Stryker	JavaScript/TypeScript, C#, Scala	https://stryker-mutator.io
MutPy	Python	https://github.com/mutpy/mutpy
Mutant	Ruby	https://github.com/mbj/mutant
Mull	C/C++	https://github.com/mull-project/mull
Infection	PHP	https://infection.github.io

Mutation Testing Workflow

Mutation Testing Workflow

Configure mutation testing

Mutation Testing Workflow

Configure mutation testing

Generate mutants, run tests:

Test fails -> **mutant killed**

Test passes -> **mutant survived**

Mutation Testing Workflow

Configure mutation testing

Generate mutants, run tests:

Test fails -> **mutant killed**

Test passes -> **mutant survived**

Test strength measured on mutants killed

Configuring Pit

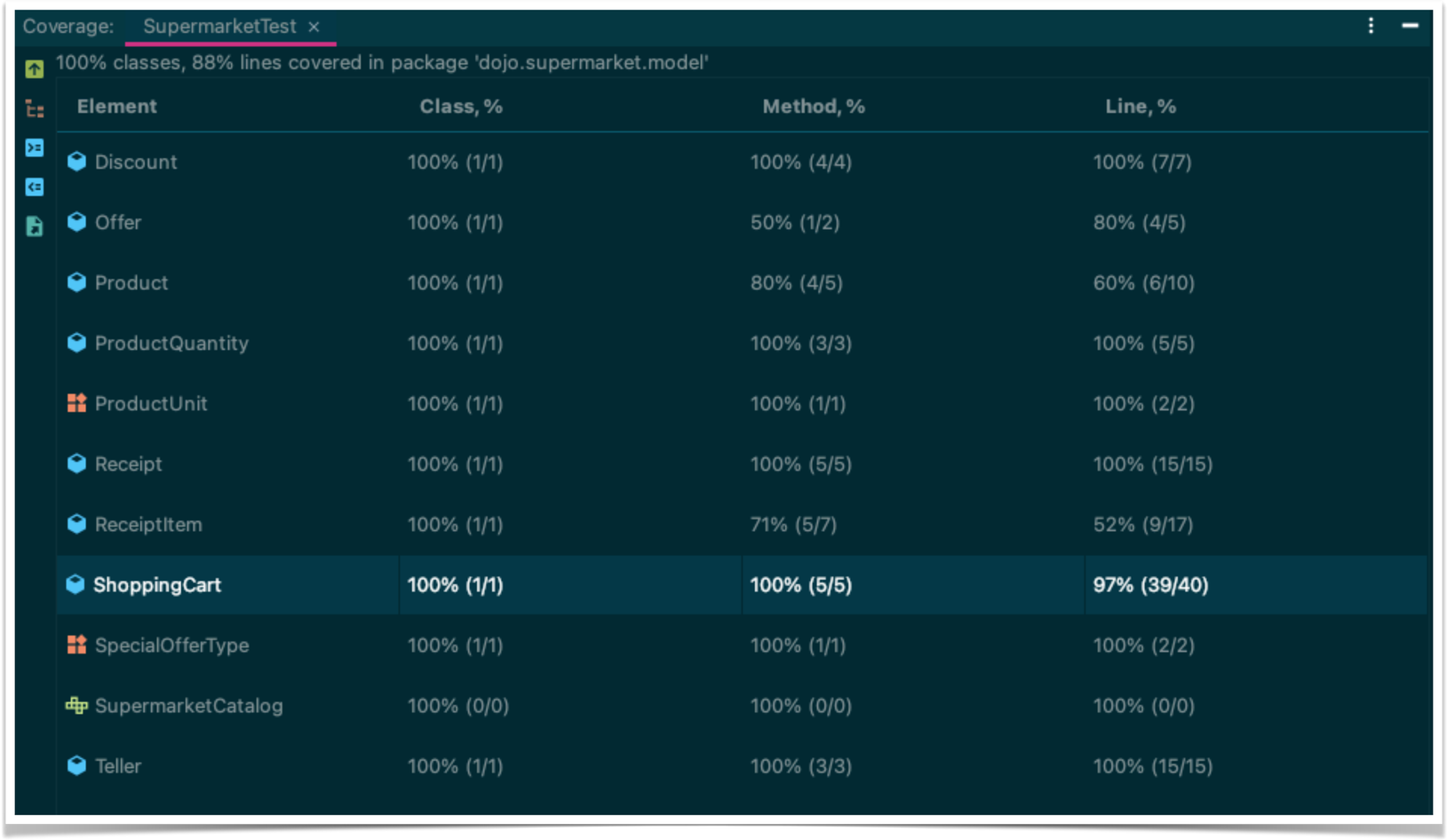
Configuring Pit

```
pitest {
    threads = 4
    outputFormats = ['XML', 'HTML']
    jvmArgs = ['-Xmx1024m']
    fileExtensionsToFilter.addAll('xml', 'orbit')
    junit5PluginVersion = '0.15'
    mutators = ['STRONGER', 'REMOVE_CONDITIONALS',
'INLINE_CONSTS','CONSTRUCTOR_CALLS']
    timestampedReports = false
}
```

Configuring Pit

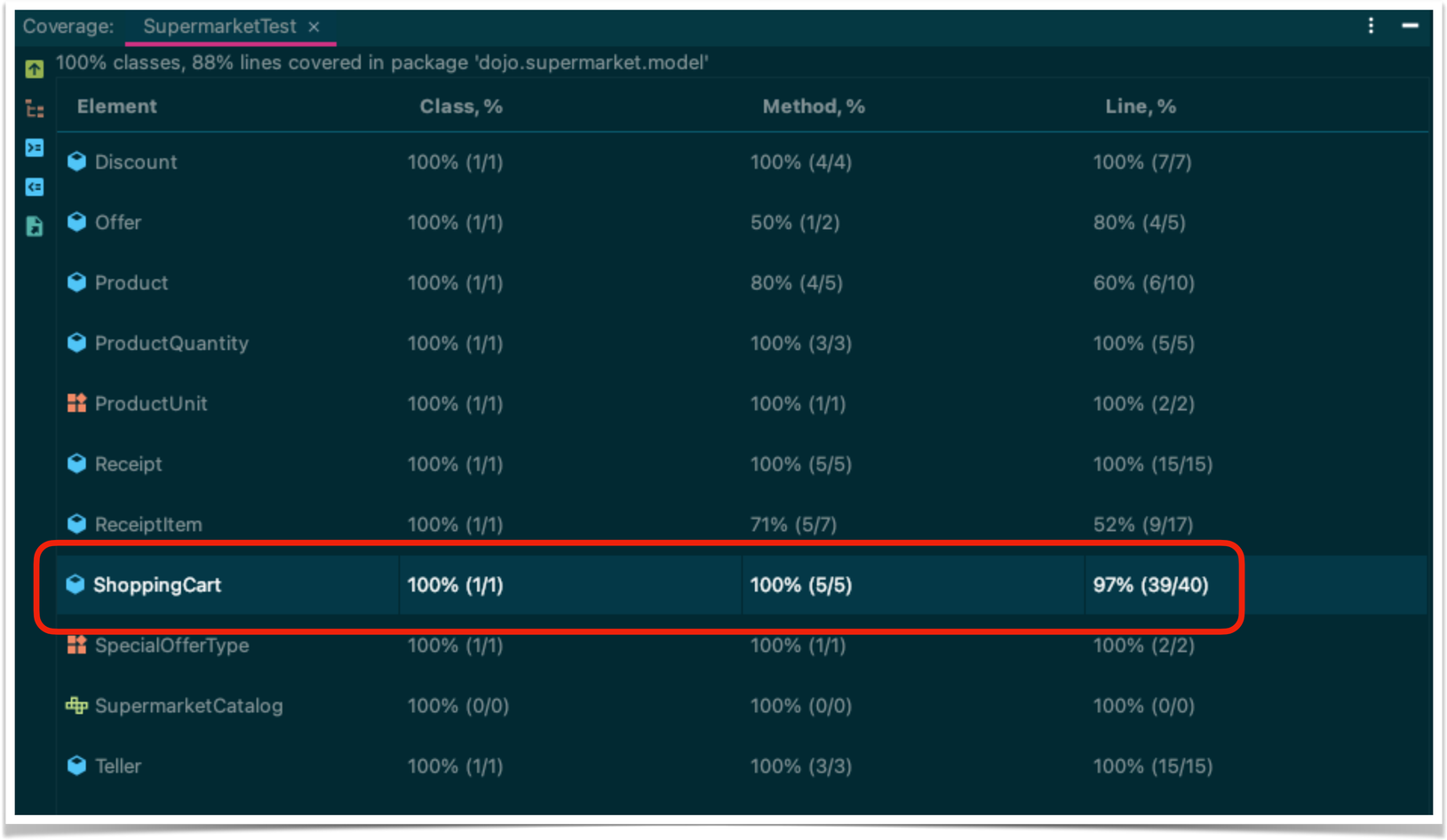
```
pitest {
    threads = 4
    outputFormats = ['XML', 'HTML']
    jvmArgs = ['-Xmx1024m']
    fileExtensionsToFilter.addAll('xml', 'orbit')
    junit5PluginVersion = '0.15'
    mutators = ['STRONGER', 'REMOVE_CONDITIONALS',
'INLINE_CONSTS','CONSTRUCTOR_CALLS']
    timestampedReports = false
}
```

Running Pit



Element	Class, %	Method, %	Line, %
Discount	100% (1/1)	100% (4/4)	100% (7/7)
Offer	100% (1/1)	50% (1/2)	80% (4/5)
Product	100% (1/1)	80% (4/5)	60% (6/10)
ProductQuantity	100% (1/1)	100% (3/3)	100% (5/5)
ProductUnit	100% (1/1)	100% (1/1)	100% (2/2)
Receipt	100% (1/1)	100% (5/5)	100% (15/15)
ReceiptItem	100% (1/1)	71% (5/7)	52% (9/17)
ShoppingCart	**100% (1/1)**	**100% (5/5)**	**97% (39/40)**
SpecialOfferType	100% (1/1)	100% (1/1)	100% (2/2)
SupermarketCatalog	100% (0/0)	100% (0/0)	100% (0/0)
Teller	100% (1/1)	100% (3/3)	100% (15/15)

Running Pit


Coverage: SupermarketTest ×
100% classes, 88% lines covered in package 'dojo.supermarket.model'
Element
Class, %
Method, %
Line, %
Discount 100% (1/1) 100% (4/4) 100% (7/7)
Offer 100% (1/1) 50% (1/2) 80% (4/5)
Product 100% (1/1) 80% (4/5) 60% (6/10)
ProductQuantity 100% (1/1) 100% (3/3) 100% (5/5)
ProductUnit 100% (1/1) 100% (1/1) 100% (2/2)
Receipt 100% (1/1) 100% (5/5) 100% (15/15)
ReceiptItem 100% (1/1) 71% (5/7) 52% (9/17)
ShoppingCart 100% (1/1) 100% (5/5) 97% (39/40)
SpecialOfferType 100% (1/1) 100% (1/1) 100% (2/2)
SupermarketCatalog 100% (0/0) 100% (0/0) 100% (0/0)
Teller 100% (1/1) 100% (3/3) 100% (15/15)

Running Pit

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
9	92%	153/167	67%	197/294	89%	197/221

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
dojo.supermarket	1	100%	41/41	100%	79/79	100%	79/79
dojo.supermarket.model	8	89%	112/126	55%	118/215	83%	118/142

Report generated by PIT 1.7.0

Running Pit

ShoppingCart.java

```
1   package dojo.supermarket.model;
2
3   import java.util.ArrayList;
4   import java.util.HashMap;
5   import java.util.List;
6   import java.util.Map;
7
8   public class ShoppingCart {
9
10 1     private final List<ProductQuantity> items = new ArrayList<>();
11 1     Map<Product, Double> productQuantities = new HashMap<>();
12
13
14      List<ProductQuantity> getItems() {
15 2         return new ArrayList<>(items);
16      }
17
18      void addItem(Product product) {
19 2         this.addItemQuantity(product, 1.0);
20      }
21
22      Map<Product, Double> productQuantities() {
23 1         return productQuantities;
24      }
25
26
27      public void addItemQuantity(Product product, double quantity) {
28 2         items.add(new ProductQuantity(product, quantity));
29 4         if (productQuantities.containsKey(product)) {
30 5             productQuantities.put(product, productQuantities.get(product) + quantity);
31          } else {
32 2             productQuantities.put(product, quantity);
33          }
34      }
35
36      void handleOffers(Receipt receipt, Map<Product, Offer> offers, SupermarketCatalog catalog) {
37 2         for (Product p: productQuantities().keySet()) {
38 2
39 4
40 1
41 1
42
43
44 1                 int x = 1;
45 3                 if (offer.offerType == SpecialOfferType.ThreeForTwo) {
46 1                     x = 3;
```

1. handleOffers : removed conditional - replaced equality check with true → SURVIVED
2. handleOffers : negated conditional → KILLED
3. handleOffers : removed call to java/util/Map::containsKey → KILLED
4. handleOffers : removed conditional - replaced equality check with false → KILLED

Running Pit

ShoppingCart.java

```
1   package dojo.supermarket.model;
2
3   import java.util.ArrayList;
4   import java.util.HashMap;
5   import java.util.List;
6   import java.util.Map;
7
8   public class ShoppingCart {
9
10 1     private final List<ProductQuantity> items = new ArrayList<>();
11 1     Map<Product, Double> productQuantities = new HashMap<>();
12
13
14      List<ProductQuantity> getItems() {
15 2         return new ArrayList<>(items);
16      }
17
18      void addItem(Product product) {
19 2         this.addItemQuantity(product, 1.0);
20      }
21
22      Map<Product, Double> productQuantities() {
23 1         return productQuantities;
24      }
25
26
27      public void addItemQuantity(Product product, double quantity) {
28 2         items.add(new ProductQuantity(product, quantity));
29 4         if (productQuantities.containsKey(product)) {
30 5             productQuantities.put(product, productQuantities.get(product) + quantity);
31          } else {
32 2             productQuantities.put(product, quantity);
33          }
34      }
35
36      void handleOffers(Receipt receipt, Map<Product, Offer> offers, SupermarketCatalog catalog) {
37 2         for (Product p: productQuantities().keySet()) {
38 2             double quantity = productQuantities.get(p);
39 4             if (offers.containsKey(p)) {
40 1                 Offer offer = offers.get(p);
41 1                 double unitPrice = catalog.getUnitPrice(p);
42                  int quantityAsInt = (int) quantity;
43                  Discount discount = null;
44 1                 int x = 1;
45 3                 if (offer.offerType == SpecialOfferType.ThreeForTwo) {
46 1                     x = 3;
```

Running Pit

Product.java

```
1   package dojo.supermarket.model;
2
3   import java.util.Objects;
4
5   public class Product {
6       private final String name;
7       private final ProductUnit unit;
8
9       public Product(String name, ProductUnit unit) {
10          this.name = name;
11          this.unit = unit;
12      }
13
14      public String getName() {
15 1        return name;
16      }
17
18
19      public ProductUnit getUnit() {
20 1        return unit;
21      }
22
23      @Override
24
25 4
26 10
27
28 10
29                  unit == product.unit;
30      }
31
32      @Override
33      public int hashCode() {
34
35 5        return Objects.hash(name, unit);
36      }
37  }
```

1. equals : Substituted 1 with 0 → NO_COVERAGE
2. equals : negated conditional → NO_COVERAGE
3. equals : removed conditional - replaced equality check with true → NO_COVERAGE
4. equals : replaced boolean return with false for dojo/supermarket/model/Product::equals → NO_COVERAGE

Mutations

15 1. replaced return value with "" for dojo/supermarket/model/Product::getName → KILLED

20 1. replaced return value with null for dojo/supermarket/model/Product::getUnit → KILLED

25
1. Substituted 1 with 0 → NO_COVERAGE
2. negated conditional → NO_COVERAGE
3. removed conditional - replaced equality check with true → NO_COVERAGE
4. replaced boolean return with false for dojo/supermarket/model/Product::equals → NO_COVERAGE

Day to Day Mutation Testing

Day to Day Mutation Testing

Mutation testing can be slow and computationally expensive

Day to Day Mutation Testing

Mutation testing can be slow and computationally expensive

Locally: only changed code

Day to Day Mutation Testing

Mutation testing can be slow and computationally expensive

Locally: only changed code

CI: complete suite

Day to Day Mutation Testing

Mutation testing can be slow and computationally expensive

Locally: only changed code

CI: complete suite

Best used with unit tests

Test Driven Development

“Isn’t that where you write all of your tests first?”

“How can you test something that you haven’t written yet?”

“I tried that once for five minutes and I hated it.”

If TDD is a best practice in our industry why aren’t we all doing it?

Test driven development seems counter intuitive

Three Rules of TDD

Three Rules of TDD

1. You cannot write any production code until you have first written a failing unit test.

Three Rules of TDD

1. You cannot write any production code until you have first written a failing unit test.

2. You cannot write more of a unit test than is sufficient to fail, and not compiling is failing.

Three Rules of TDD

1. You cannot write any production code until you have first written a failing unit test.

2. You cannot write more of a unit test than is sufficient to fail, and not compiling is failing.

3. You cannot write more production code than is sufficient to pass the currently failing unit test.

“Clean code that works”

Ron Jeffries, quoted in *Test Driven Development by Example* by Kent Beck

Three Rules of Example Guided Development

Three Rules of Example Guided Development

1. You cannot write any production code until you have first written a non-working example of how the code works.

Three Rules of Example Guided Development

1. You cannot write any production code until you have first written a non-working example of how the code works.

2. You cannot write more of an example than is sufficient to not work, and not compiling is not working.

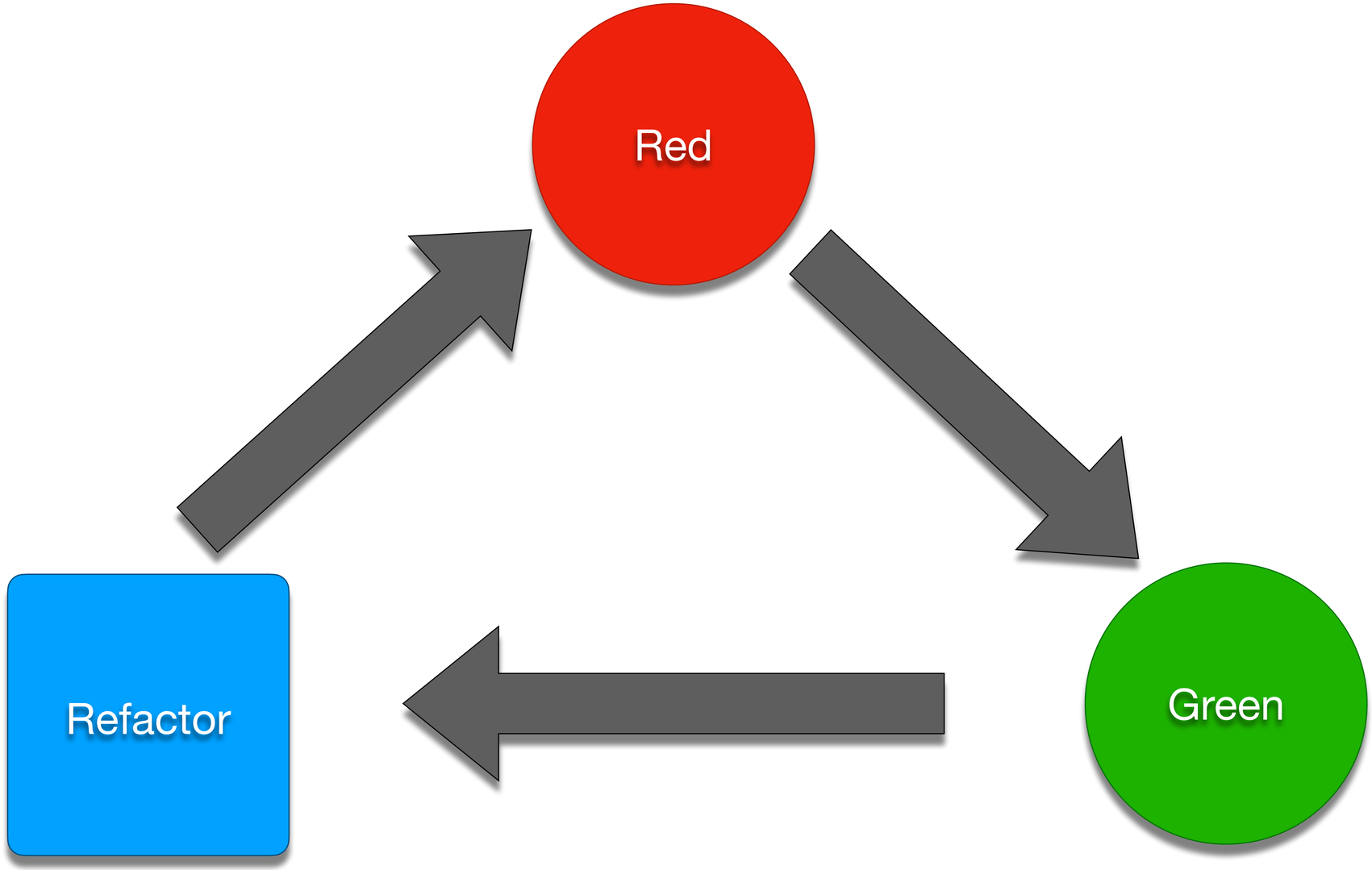
Three Rules of Example Guided Development

1. You cannot write any production code until you have first written a non-working example of how the code works.

2. You cannot write more of an example than is sufficient to not work, and not compiling is not working.

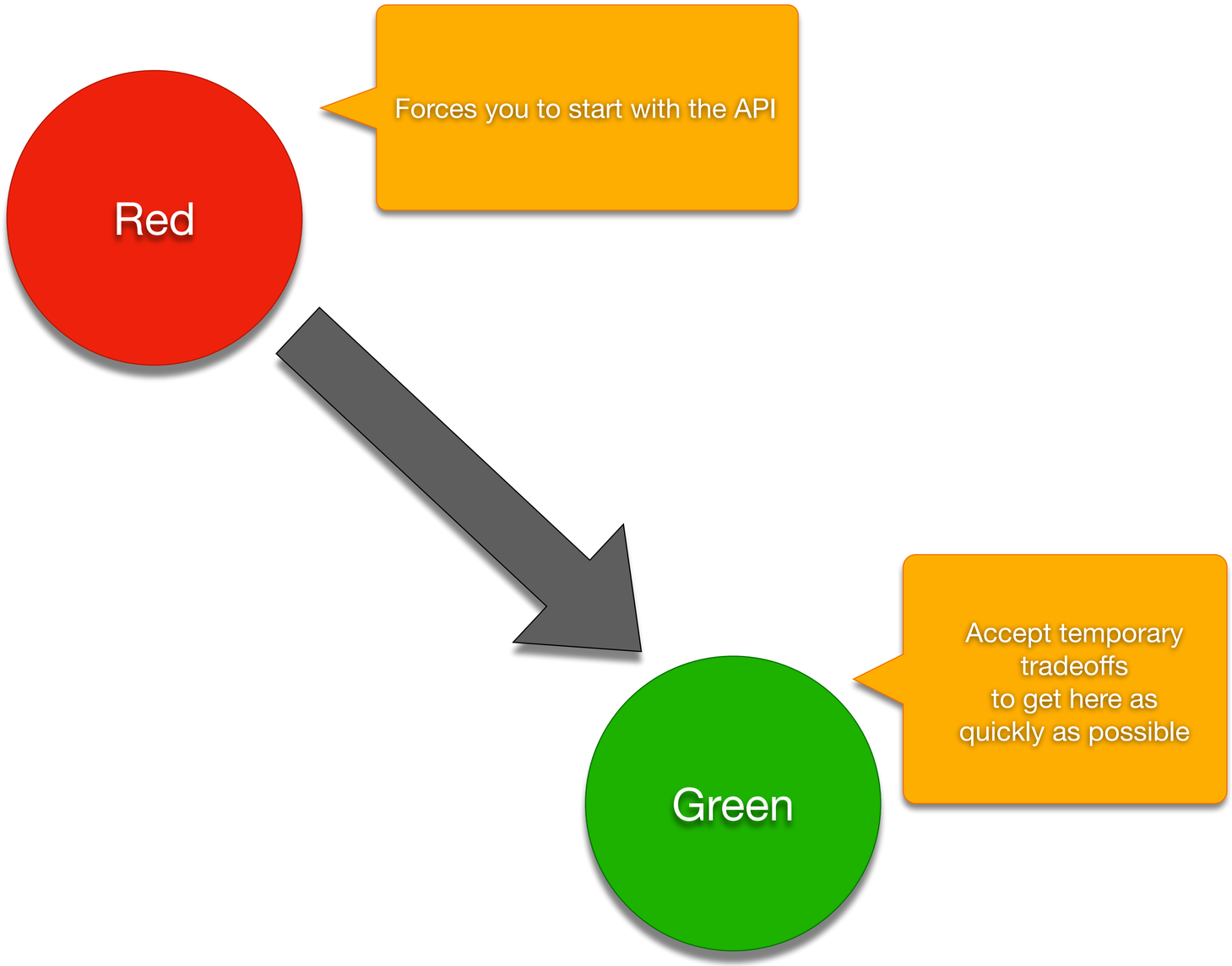
3. You cannot write more production code than is sufficient to make the current example work.

Three Stages of TDD

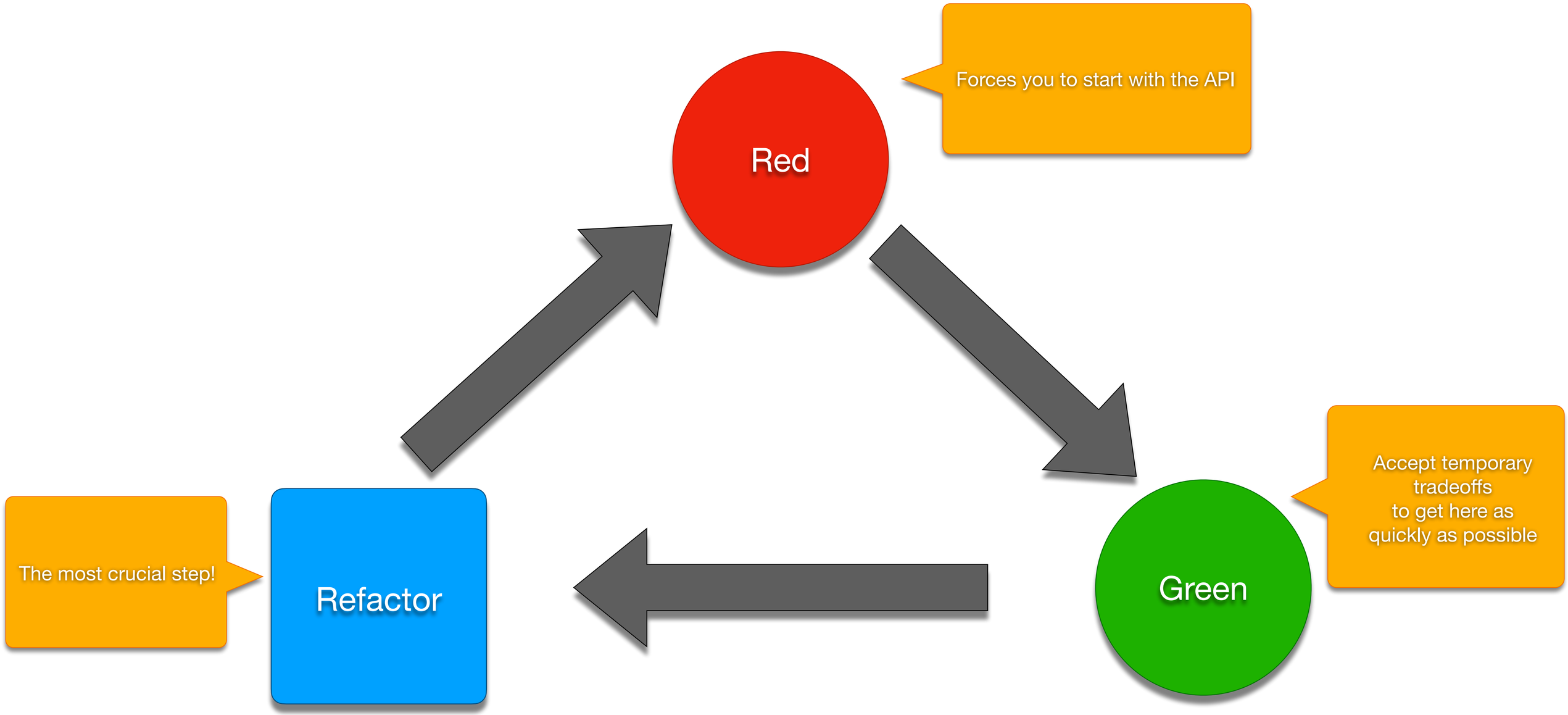


Three Stages of TDD

Three Stages of TDD



Three Stages of TDD



Doesn’t this slow me down?*

Doesn’t this slow me down?*

*No, it actually makes you go faster!

TDD makes you go faster

TDD makes you go faster

Small upfront investment -> almost instant dividends

TDD makes you go faster

Small upfront investment -> almost instant dividends

Fewer bugs when putting code together

TDD makes you go faster

Small upfront investment -> almost instant dividends

Fewer bugs when putting code together

Code is simple to change later

Tips for Successful TDD

Make a list

Think about ZOMBIES

Make a List

Make a List

Make a list of test cases:

- Give each a name
- List the inputs and expected output

Make a List

Make a list of test cases:

- Give each a name
- List the inputs and expected output

Doesn’t need to be comprehensive

Make a List

Make a list of test cases:

- Give each a name
- List the inputs and expected output

Doesn’t need to be comprehensive

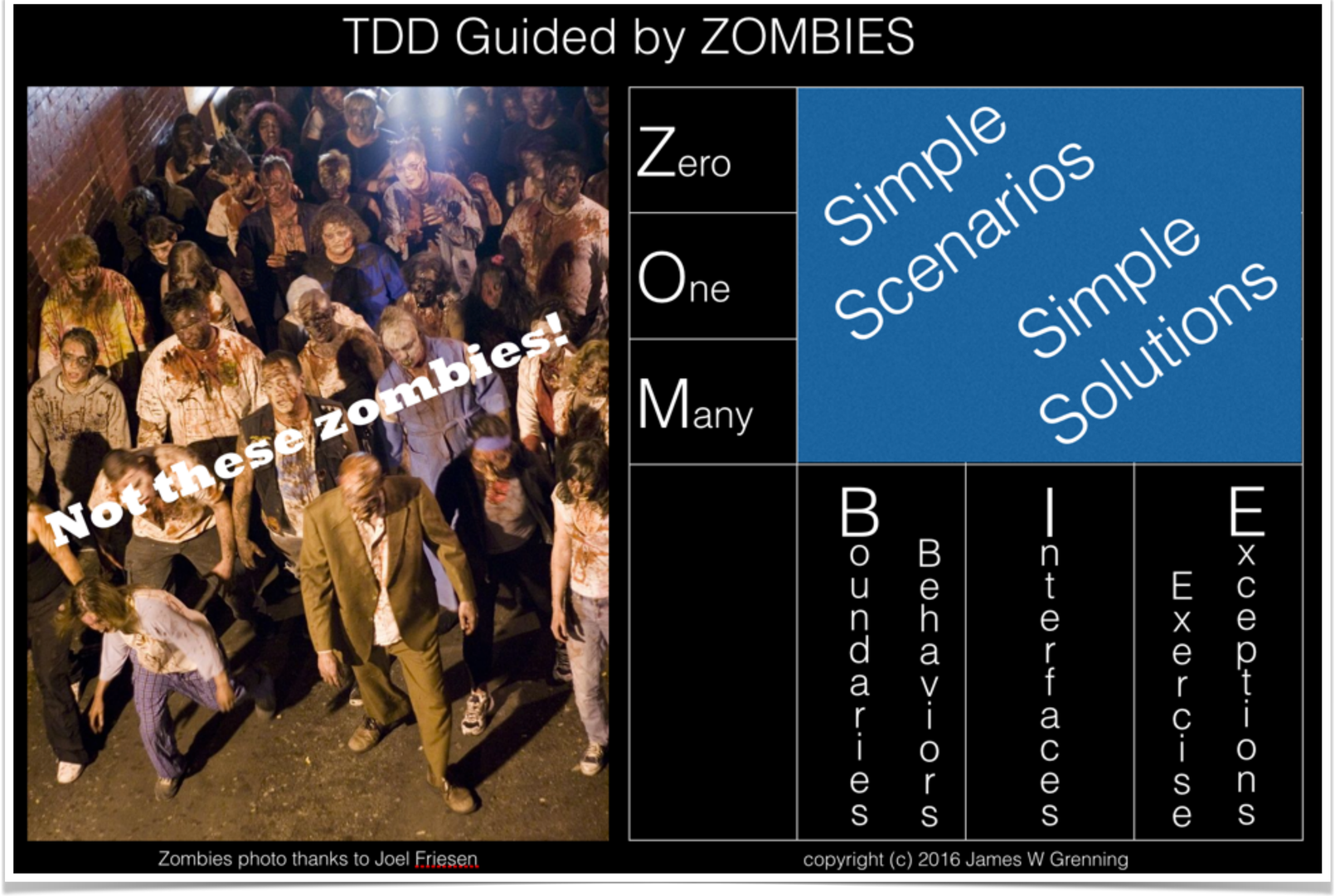
Gets you thinking about the API and design

Think about ZOMBIES

https://blog.wingman-sw.com/tdd-guided-by-zombies

Think about ZOMBIES

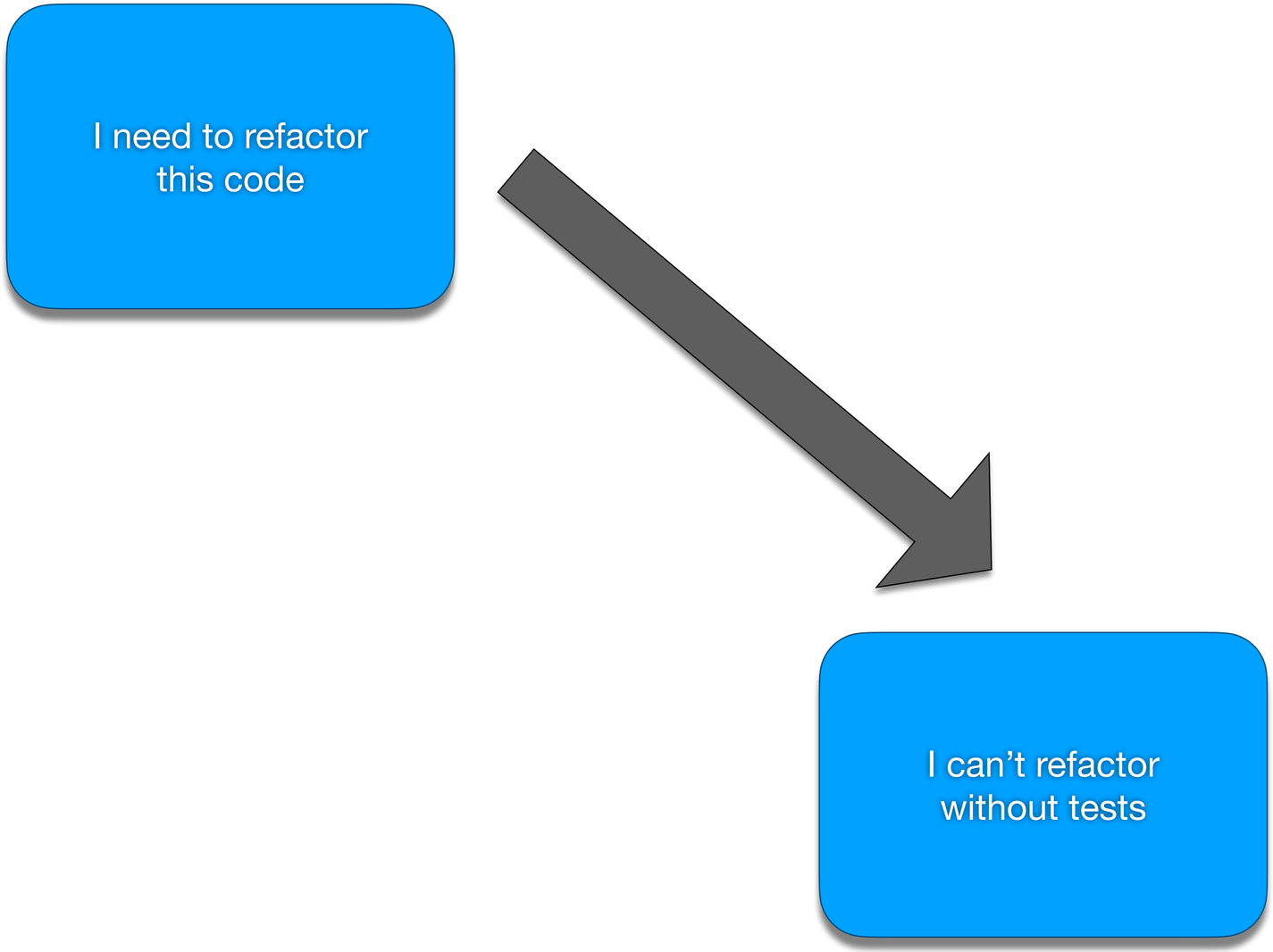
https://blog.wingman-sw.com/tdd-guided-by-zombies



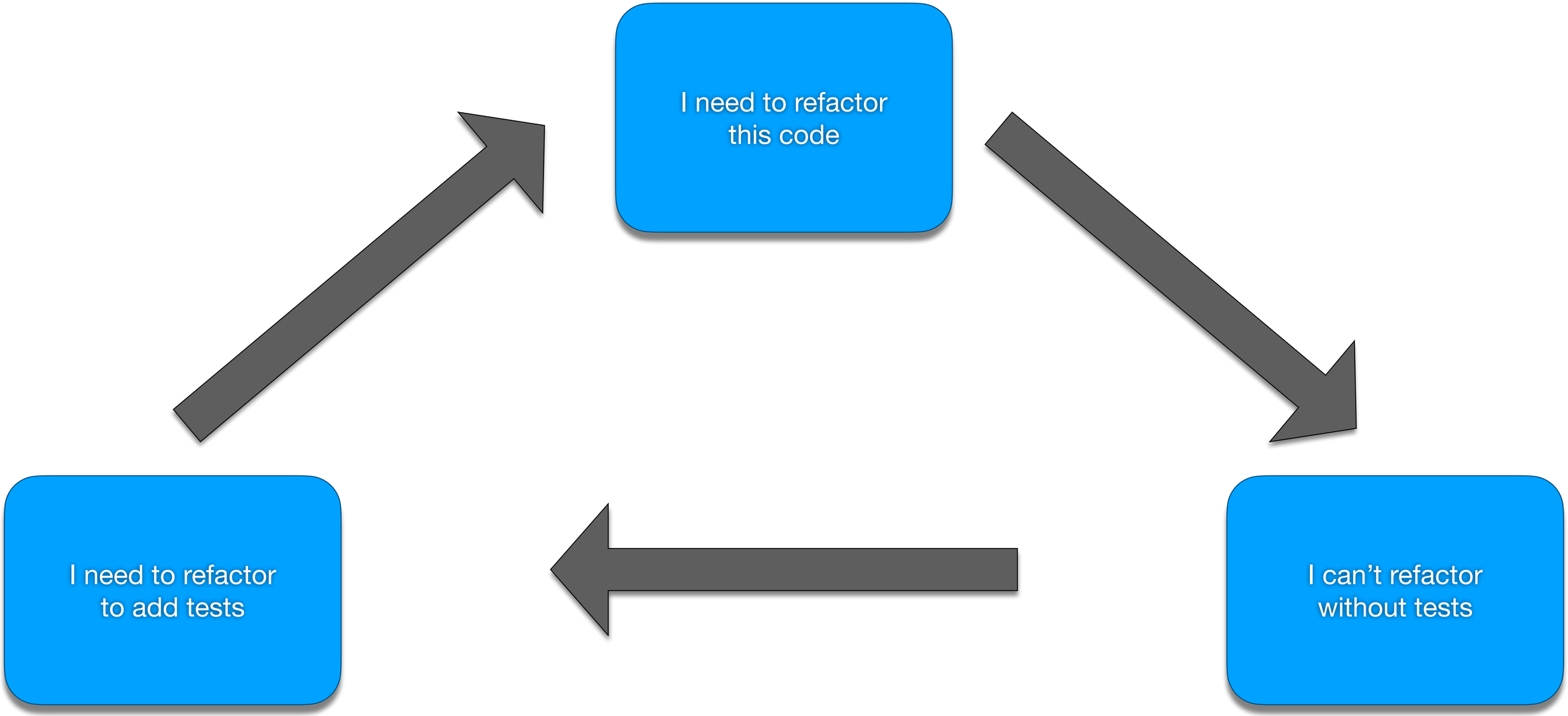
Dealing with Legacy Code

The Legacy Code Dilemma

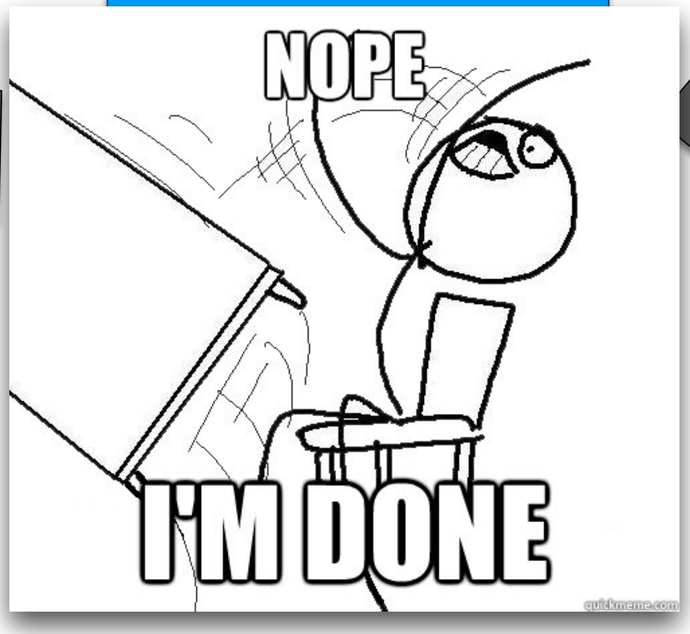
The Legacy Code Dilemma



The Legacy Code Dilemma

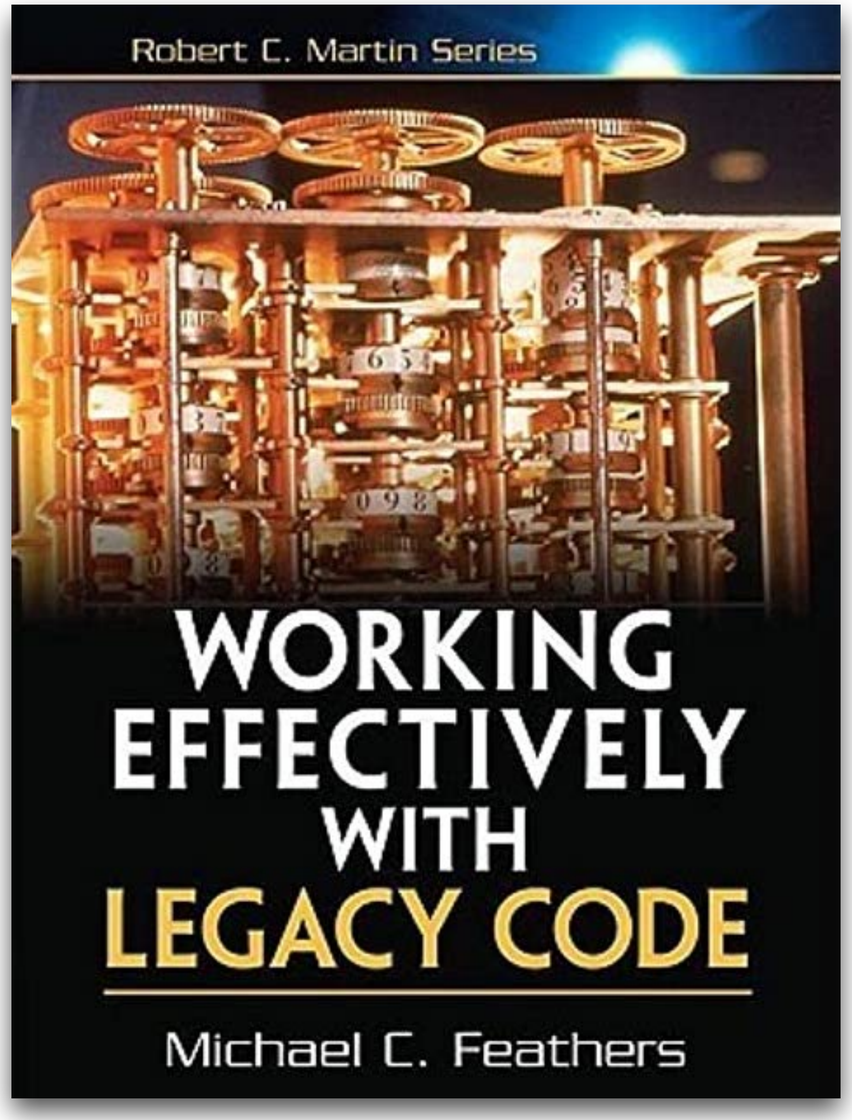


The Legacy Code Dilemma



I need to refactor to add tests

I can't refactor without tests


Robert C. Martin Series
WORKING EFFECTIVELY WITH LEGACY CODE
Michael C. Feathers

The Legacy Code Change Algorithm

Identify change points

Find test points

Break dependencies

Write tests

Make changes and refactor

The Legacy Code Change Algorithm

Identify change points

Find test points

Break dependencies

Write tests

Make changes and refactor

“A *pinch point* is a narrowing in an effect sketch, a place where tests against a couple of methods can detect changes in many methods.”

Michael Feathers, *Working Effectively with Legacy Code*

Break Dependencies

Break Dependencies

Just enough changes to add tests

Break Dependencies

Just enough changes to add tests

Accept temporary ugliness

Break Dependencies

Just enough changes to add tests

Accept temporary ugliness

Safe changes -> **lean on your tools**!

I can’t add tests to this code until I understand it*

I can’t add tests to this code until I understand it*

*This is false

Write Tests

Write Tests

Different to testing new code

Write Tests

Different to testing new code

Let code tell you its behaviour

Write Tests

Different to testing new code

Let code tell you its behaviour

Focus on characterization tests with mutation testing

Approval Testing

Approval Testing

Captures output and serializes it to a file

Approval Testing

Captures output and serializes it to a file

Future test runs compare output to that stored

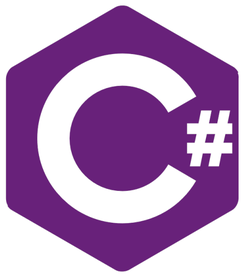
Approval Testing

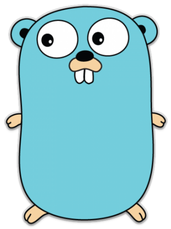
Captures output and serializes it to a file

Future test runs compare output to that stored

Many uses but **very powerful with legacy code**

Approval Testing





Approval Testing

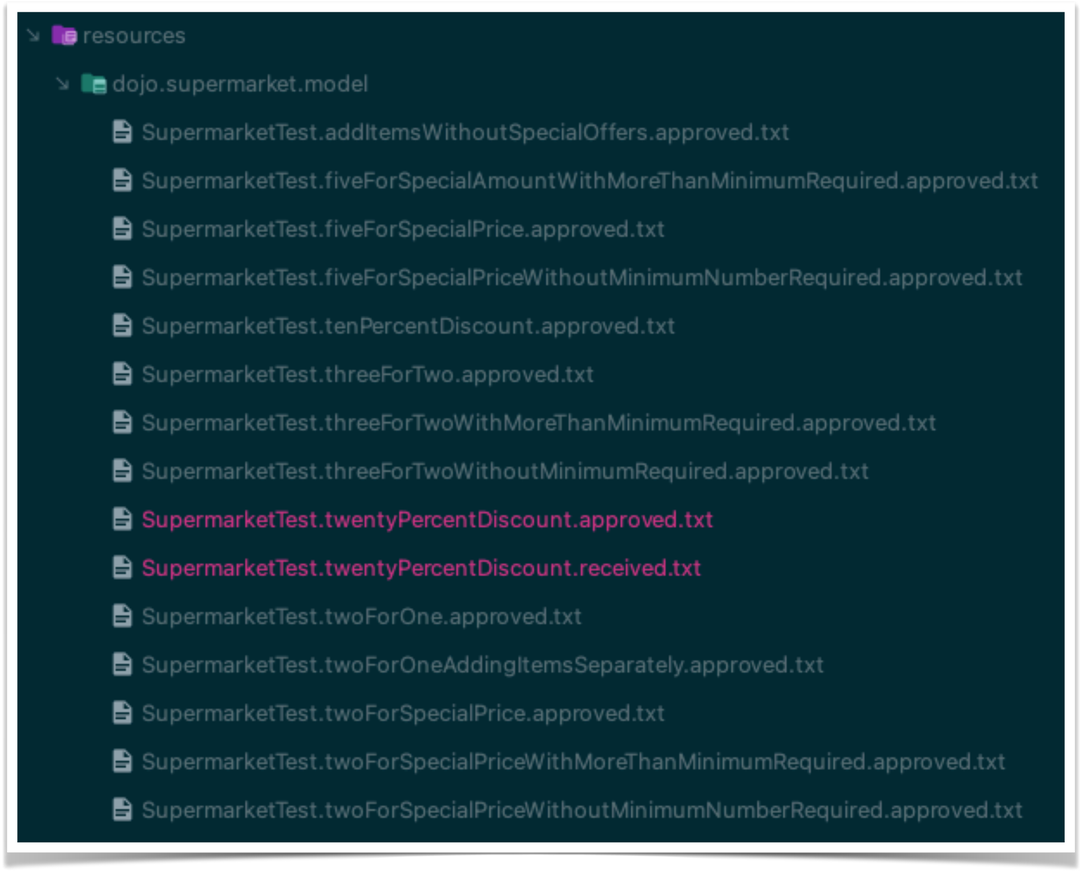
Approval Testing

```
@Test
void twentyPercentDiscount() {
    var apples = new Product("apples", ProductUnit.Kilo);
    catalog.addProduct(apples, 1.99);
    teller.addSpecialOffer(SpecialOfferType.TenPercentDiscount, apples, 20.0);

    cart.addItemQuantity(apples, 2.5);
    var result = receiptPrinter.printReceipt(teller.checksOutArticlesFrom(cart));

    Approvals.verify(result);
}
```

Approval Testing

```
@Test
void twentyPercentDiscount() {
    var apples = new Product("apples", ProductUnit.Kilo);
    catalog.addProduct(apples, 1.99);
    teller.addSpecialOffer(SpecialOfferType.TenPercentDiscount, apples, 20.0);

    cart.addItemQuantity(apples, 2.5);
    var result = receiptPrinter.printReceipt(teller.checksOutArticlesFrom(cart));

    Approvals.verify(result);
}
```

Approval Testing



Approval Testing

Approval Testing

Rename received file to approved -> test will pass

Approval Testing

Rename received file to approved -> test will pass

If test fails in future -> new received file will be generated

Approval Testing

Rename received file to approved -> test will pass

If test fails in future -> new received file will be generated

Files automatically handled

Approval Testing Considerations

Approval Testing Considerations

Result must be serializable

Approval Testing Considerations

Result must be serializable

Non-deterministic data needs to be specially handled

Approval Testing Considerations

Result must be serializable

Non-deterministic data needs to be specially handled

Approved files need to be added to VCS

In Conclusion

In Conclusion

Use code coverage but know what it really tells you!

In Conclusion

Use code coverage but know what it really tells you!

Mutation testing -> truer picture of test health

In Conclusion

Use code coverage but know what it really tells you!

Mutation testing -> truer picture of test health

Use TDD + mutation testing in new code

In Conclusion

Use code coverage but know what it really tells you!

Mutation testing -> truer picture of test health

Use TDD + mutation testing in new code

Use characterization tests + mutation testing in legacy code

Any questions?