

# **PHPUnit and Drupal 8: 'No Unit Left Behind'**

Paul Mitchum  
Mile23 on Drupal.org  
paul-m on Github  
from Seattle

# **G3T C0D3Z**

Support page on my site:

<http://mile23.com/phpunit-talk>

# Keep In Mind:

The ability to give an hour-long presentation is not the same as being an expert on the subject.

Also: Please hold questions until afterwards.

# Two Goals:

Give some practical knowledge in how to run unit tests and interpret what they mean.

Even if you never write unit tests, you can run and interpret them.

Introduce some concepts about testing in general, and unit testing in particular.

To get you started writing tests.

# I Assume That You Understand:

- Basic PHP OOP.

Classes, Interfaces, Methods

- Basics of Composer, /vendor, etc.
- \*nix flavored command line. No Windows.

# Three Sections:

## 1. How to use the tools.

- Run Drupal 8 PHPUnit tests.
- Generate a PHPUnit coverage report.
- Basics of interpreting coverage reports.

## 2. Which tests are which?

- Definitions: Unit, Functional, Behavioral
- Mainly differences between Functional and Unit testing.
- SimpleTest vs. PHPUnit

## 3. Important unit testing concepts.

- Isolation, DataProviders, Test doubles

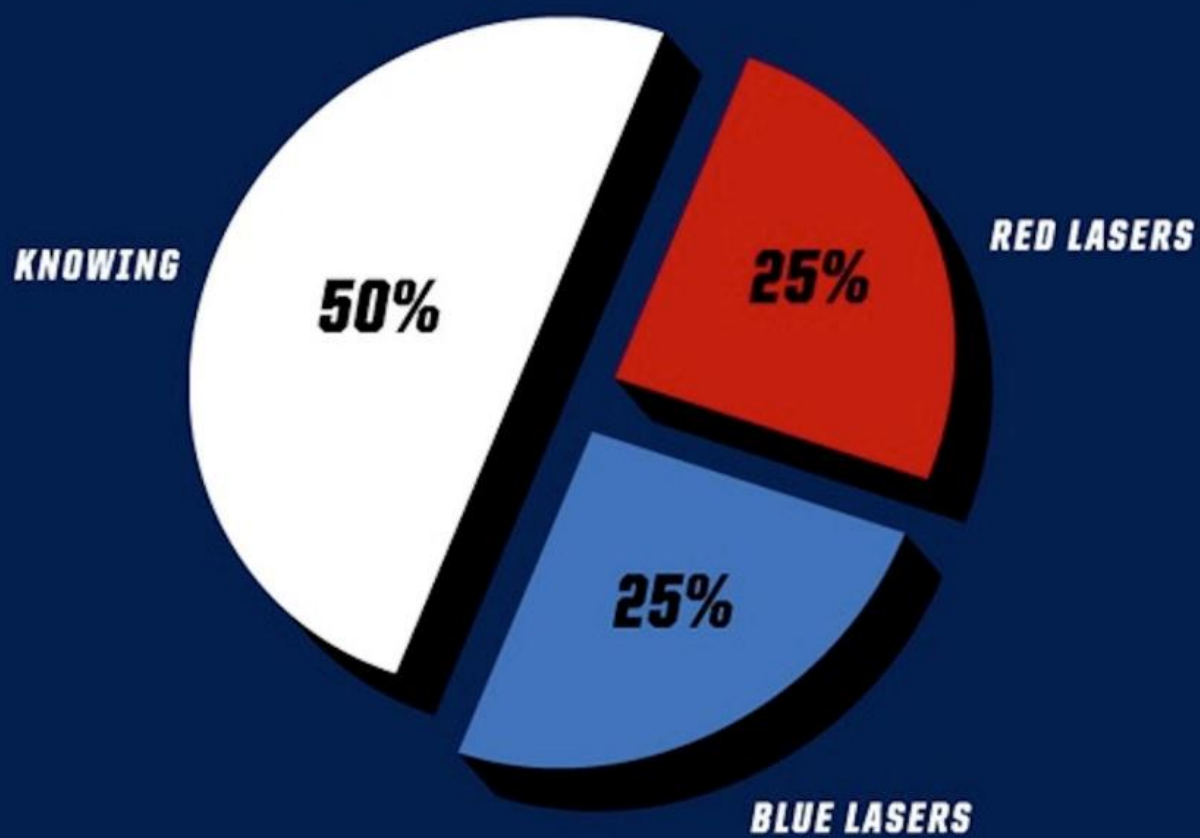
**What Do We Want?**

**To Know.**

**Stuff.**

**For Sure.**

# ***THE BATTLE***





# What Do We Want To Know?

## That Our Code Works.

Development Process

Regression Testing

We want to know what we know.

## That We Can Maintain The Code.

Refactoring

Coverage

Complexity

We want to know what we don't know.

# What's a Unit?

## Code-level

- Class Unit
- Method/Function Unit
- Individual Line of Code Unit

# Tools

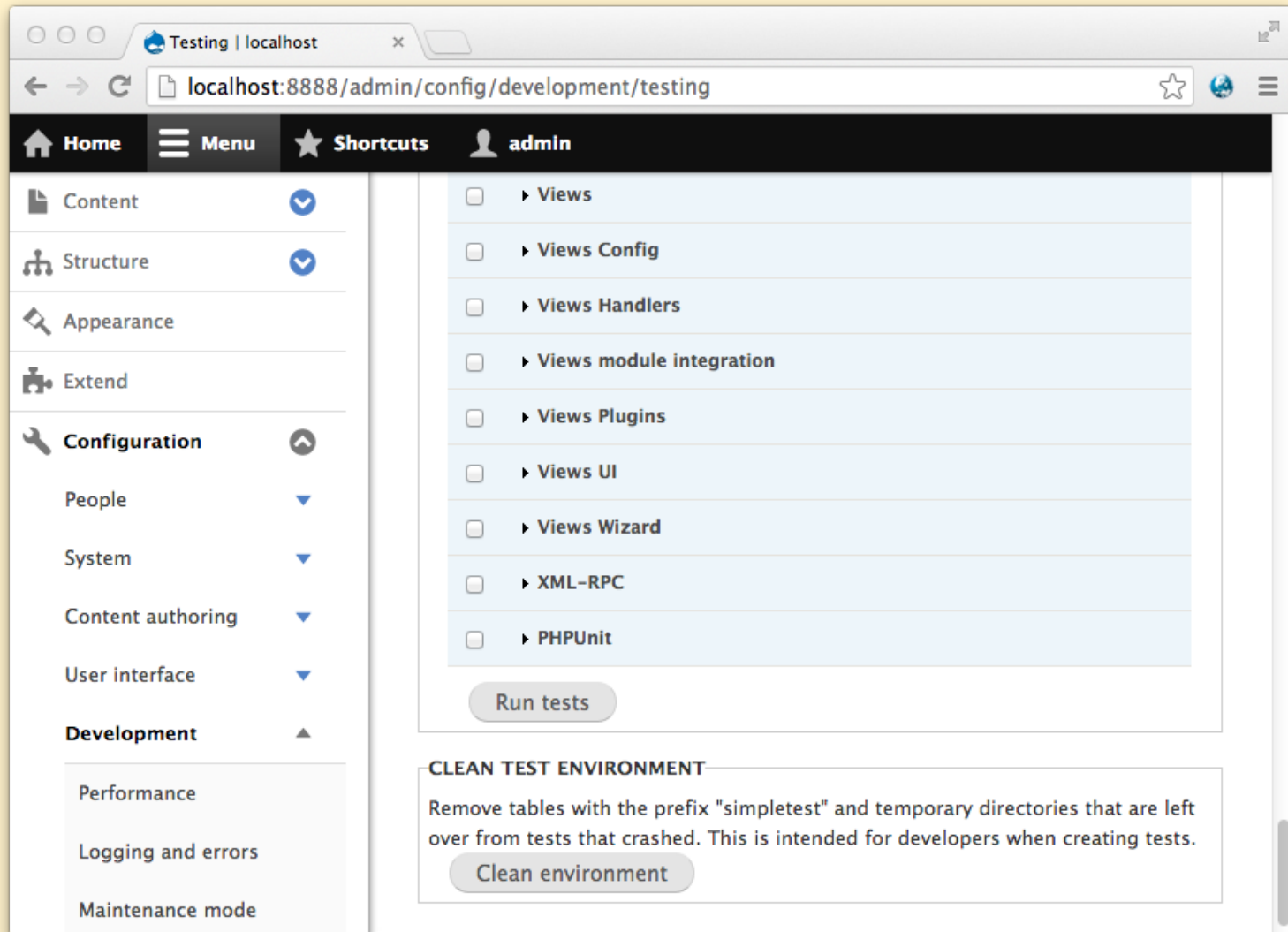
How To Run Stuff

# Testing Under Drupal 8

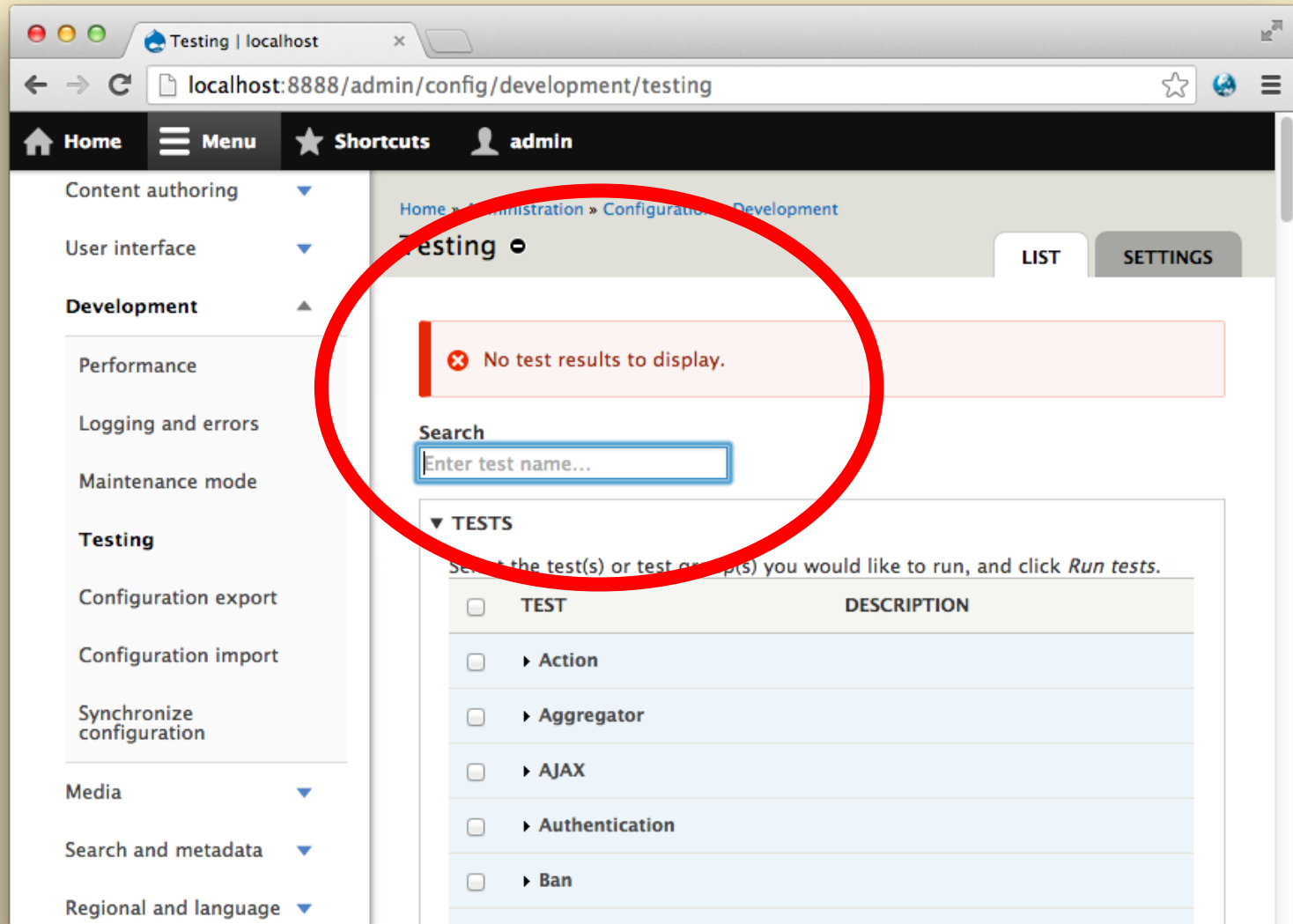
Drupal 8 has two built-in testing systems:

- SimpleTest - Functional testing, from D7.
- PHPUnit - Unit testing.

# PHPUnit under Drupal: Testing Page



# PHPUnit under Drupal: Results Page



The screenshot shows the Drupal administration interface for the 'Testing' module. The breadcrumb trail is 'Home » Administration » Configuration » Development » Testing'. The page has two tabs: 'LIST' (active) and 'SETTINGS'. A red circle highlights a message box that says 'No test results to display.' and a search input field with the placeholder text 'Enter test name...'. Below the search field is a section titled 'TESTS' with a dropdown arrow. The text below the title says 'Select the test(s) or test group(s) you would like to run, and click *Run tests*.' Below this is a table with two columns: 'TEST' and 'DESCRIPTION'. The table contains several rows, each with a checkbox and a description:

TEST	DESCRIPTION
<input type="checkbox"/>	► Action
<input type="checkbox"/>	► Aggregator
<input type="checkbox"/>	► AJAX
<input type="checkbox"/>	► Authentication
<input type="checkbox"/>	► Ban

# How To Run PHPUnit Tests (the right way)

0. At the command line...
1. Obtain Drupal 8.
2. cd to core/
3. ./vendor/bin/phpunit
4. Wait 8 seconds.
5. Know.....(all/all)

(demo)

# How To Run PHPUnit Tests: Fail

0. At the command line...

1. Obtain Drupal 8.

2. cd to core/

3. ./vendor/bin/phpunit

4. Wait 8 seconds.

5. Know.....(all/all)

(demo)



# Coverage Report

*How do you know if you can trust the tests?*

Coverage Things You Can Know:

- 1) Coverage by line numbers.
- 2) How the complexity of your code relates to its coverage. (aka C.R.A.P.: Change Risk Anti-Pattern)

# How To Generate A Coverage Report

Same as running the test, except for two things:

Requires XDebug.

```
./vendor/bin/phpunit --coverage-html [path]
```

Will generate a static HTML site.

Or a minimal text report: `--coverage-text`

# Where Is SimpleTest?

PHPUnit coverage reports do not reflect SimpleTest coverage.

SimpleTest 'coverage' isn't really line-based anyway. They test different things. So it's apples and kumquats.

# E-Z Ways To Interpret A Coverage Report

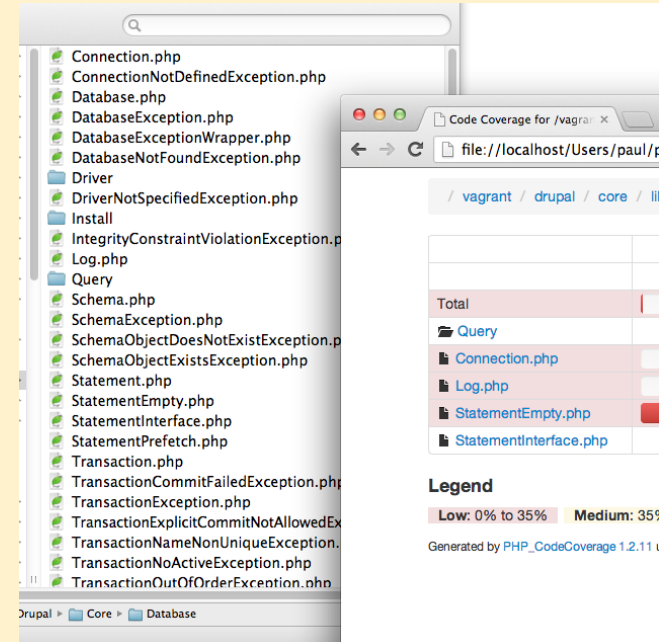
Goals:

1) **Quick Scary Info:**  
Dashboard.

## Top Project Risks

- [DisplayPluginBase](#) (183646)
- [ViewExecutable](#) (59734)
- [FilterPluginBase](#) (50400)
- [FieldPluginBase](#) (34149)
- [ViewUI](#) (21711)

2) **Missing Coverage:**  
Compare Directory  
w/ Coverage  
Coverage report won't reflect  
classes untouched by tests.

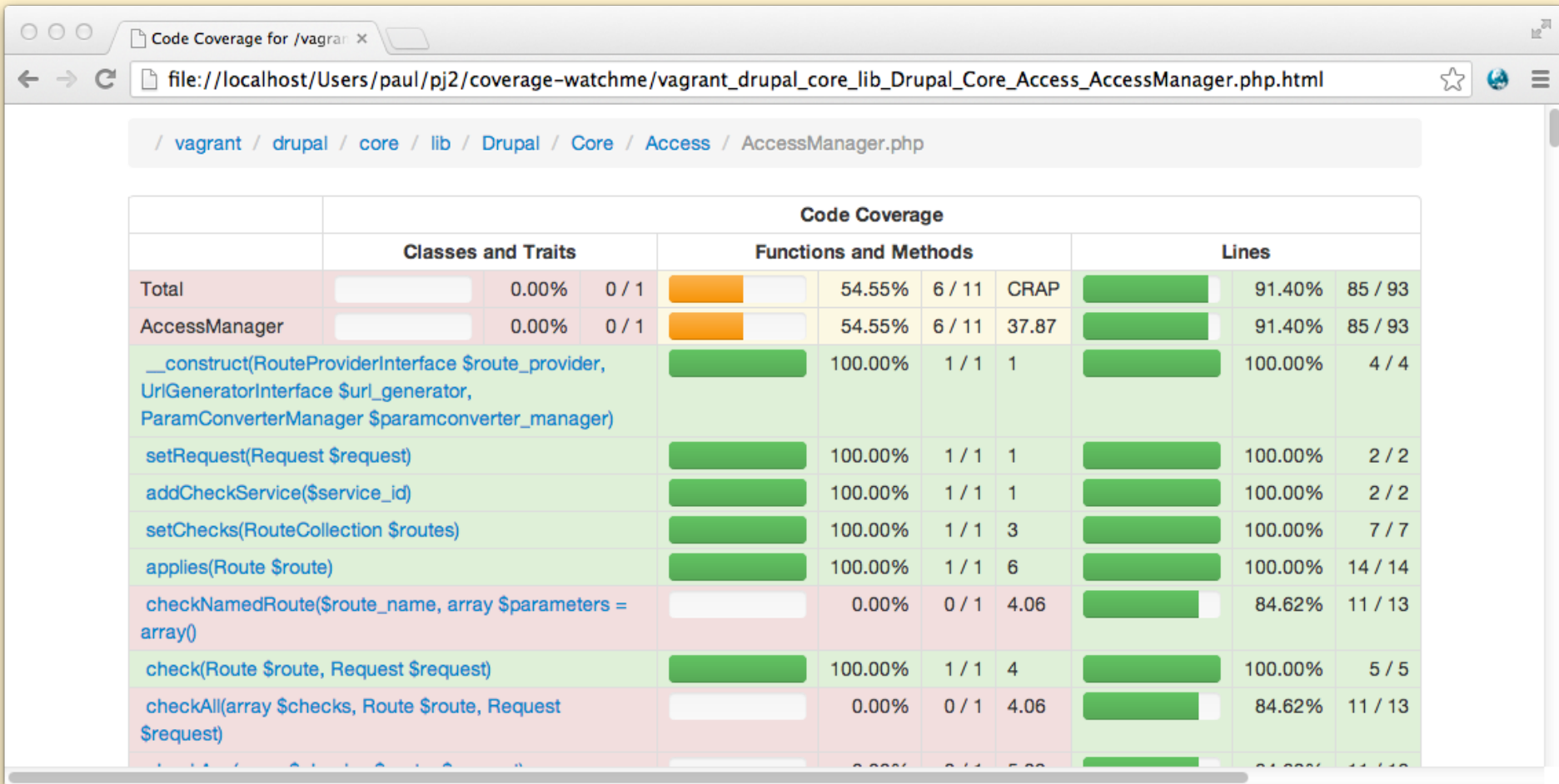


# Demo of coverage report

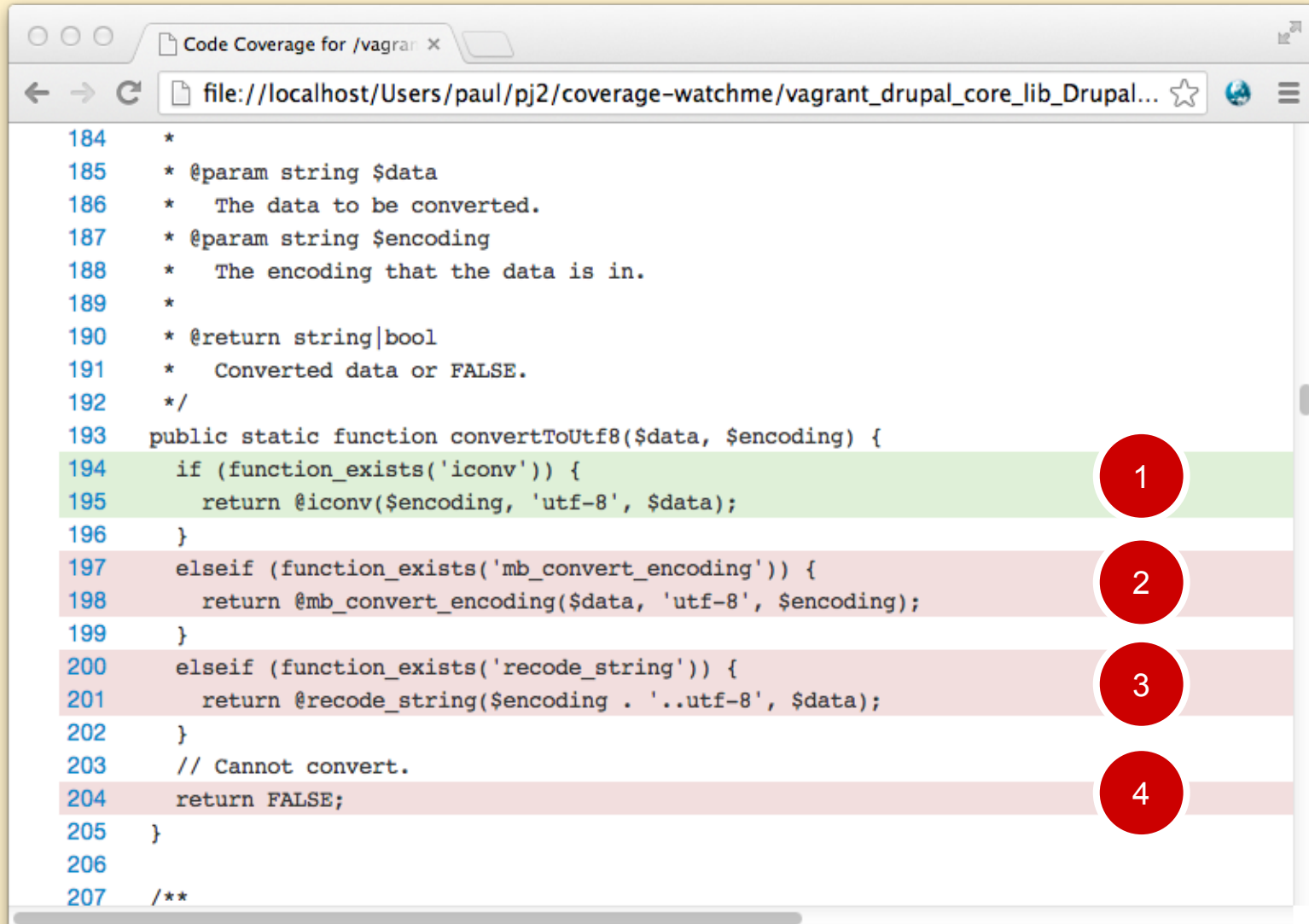
# Coverage Report 1: Overview

[illegible]

# Coverage Report 2: Single Class



# CRAP: Change Risk Anti-Pattern



The screenshot shows a code editor window titled "Code Coverage for /vagrant". The address bar shows the file path: `file:///localhost/Users/paul/pj2/coverage-watchme/vagrant_drupal_core_lib_Drupal...`. The code is a PHP function `convertToUtf8` with the following structure:

```
184  *
185  * @param string $data
186  *   The data to be converted.
187  * @param string $encoding
188  *   The encoding that the data is in.
189  *
190  * @return string|bool
191  *   Converted data or FALSE.
192  */
193  public static function convertToUtf8($data, $encoding) {
194      if (function_exists('iconv')) {
195          return @iconv($encoding, 'utf-8', $data);
196      }
197      elseif (function_exists('mb_convert_encoding')) {
198          return @mb_convert_encoding($data, 'utf-8', $encoding);
199      }
200      elseif (function_exists('recode_string')) {
201          return @recode_string($encoding . '..utf-8', $data);
202      }
203      // Cannot convert.
204      return FALSE;
205  }
206
207  /**
```

Four red circles with white numbers are overlaid on the right side of the code, indicating different execution paths or risk levels:

- 1: Line 194 (Start of the `if` block)
- 2: Line 197 (Start of the `elseif` block)
- 3: Line 200 (Start of the `elseif` block)
- 4: Line 204 (Return `FALSE` statement)



# Recap

- Run tests in order to *know* things.
- Analyze coverage to *know* about tests.
- Line coverage is good.
- CRAP reveals maintainability risks.  
(Not a judgement of the code under test.)

# **Testing Categories**

**Unit - PHPUnit**

**Functional - SimpleTest**

**Behavioral - Behat**

# Behavioral Testing



Behat

# Functional Testing (SimpleTest)

What Are You Testing?

*How (sub)systems interact.*

Does my module's `hook_schema()` install a table?

Does my module implement access control?

Is my form rendered properly?

Uses: SimpleTest tests, fixture database, server

# Unit Testing (PHPUnit)

What are you testing?

*Subatomic teensy weensy nanoparticulate code.*

Very few dependencies.

At the implementation level.

Do methods behave as expected?

What happens if I change code within a method?

# Restated

What problem domain do you care about?

**Implementation:** Unit tests.

**Overall operation:** Functional tests.

**Project-Level Outcome:** Behavioral tests.

Caveat: Very loose definitions.

# Talk About Writing Unit Tests Already!



# **PHPUnit Drupal 8**

## **Rules of the Game**



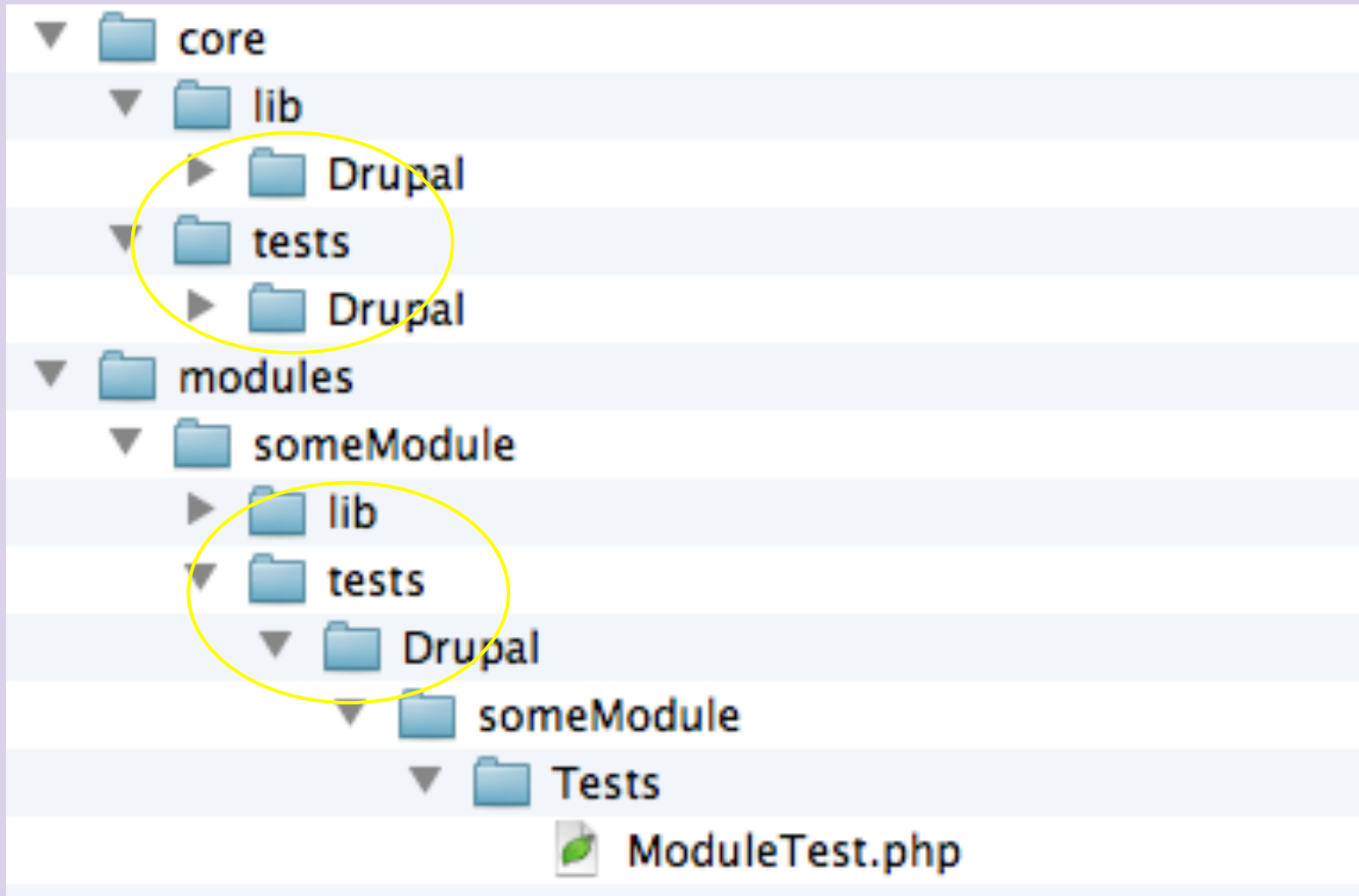
# Generic Drupal 8 PHPUnit Test Case

```
// @file [PSR_path]/ModuleTest.php
namespace \Drupal\[module]\Tests;
// UnitTestCase extends \PHPUnit_Framework_TestCase
use Drupal\Tests\UnitTestCase;

class ModuleTest extends UnitTestCase {
    public function testSomething() {
        $this->assertTrue(TRUE);
    }
}
```

# Drupal 8 PHPUnit File Setup

PSR-0 (4?) for both /lib and /tests



# PHPUnit Concepts

Things to keep in mind if you are writing tests.

Isolation

Isolation  
opposite of  
Dependency

# Concept: Isolation

Systems under test in unit tests should have as few dependencies as possible.

As few moving parts as possible.

PHPUnit wants *nothing* and should remain lean.

SimpleTest depends on the database and server: Isolation PHAIL. (Which is OK.)

# Concept: Isolation

Types of isolation:

- **System** isolation:

No database, no server

- **Language** isolation:

Pick out extensions and libraries

- **Code** isolation:

Dataproviders, test doubles, reflection, hard.

# **U R DOING IT RONG!1!!**

## **IF U R....**

- Requiring a database
- Making a module
- Subclassing anything
- Writing complex support code

Isolation Anti-Patterns



# Patterns For Isolation

- Data Providers
- `@expectedException`
- Test Doubles (mock, stub)
- Dependency Injection
- Interfaces



In code under test.

The diagram consists of a light yellow oval with a dark gray border located in the bottom right. Two dark gray arrows originate from the left side of this oval. One arrow points diagonally upwards and to the left, ending at the text 'Dependency Injection'. The other arrow points diagonally upwards and to the left, ending at the text 'Interfaces'.

# Pattern: Data Providers

**Data provider is a method that returns an array of data which PHPUnit then iterates to the test method's parameters.**

```
public function providerTestSomething() {  
    return array( array('expected', 'data'), );  
}  
  
/**  
 * @dataProvider providerTestSomething  
 */  
public function testSomething($expected, $data)  
{ // Your Logic Here }
```

# Pattern: DataProvider Isolation

Once a unit test is written, it becomes:

The Test™

Test methods should not be altered.

Data providers give us a way to change test data without changing test logic.

ALWAYS write a data provider, for any data-based test you write. The test method should not depend on specific data.

This is my sneaky way to teach you about dependency injection.

# Anti-Pattern: Exception Handling

Example without @expectedException annotation.

```
/**
 * @dataProvider providerTestException
 */
public function testException($boom)
{
    try {
        $item = new \Some\Class();
        $item->badDataMakesMeGo($boom);
    }
    catch (\InvalidArgumentException $e) {
        $this->assertTrue(TRUE); // PASS
        return;
    }
    catch ($e) {
    }
    $this->assertTrue(FALSE); // FAIL
}
```

- Cumbersome

- 12 lines

- Not immediately clear

# Pattern: ExpectedException

Test passes if an exception is thrown. Isolates test from code.

```
/**
 * @expectedException \InvalidArgumentException
 * @dataProvider providerTestException
 */
public function testException($boom) {
    $item = new \Some\Class();
    $item->badDataMakesMeGo($boom);
}
```

# Pattern: Test Doubles

PHPUnit can provide an imposter object which you can program to do stuff.

This is a 'test double.'

Test doubles perform stubbing or mocking of items needed by the system under test.

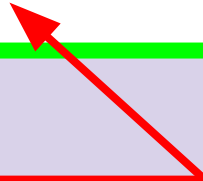
Isolate behavior of SUT from other implementations.

# Pattern: Test Doubles

Under Test



```
Class_A::  
fooMethod(\Interface_B $b);
```



```
$mock = $this->getMock  
( '\Interface_b' );
```

# Pattern: Test Doubles

```
Class_A::fooMethod(\Interface_B $b);
```

```
$mock = $this->getMock('Interface_b');  
$mock->expects($this->any())  
    ->method('stubThisMethod')  
    ->will($this->returnValue('expected'));  
$this->assertEquals(  
    'expected',  
    $a->fooMethod($mock)  
);
```

Dependency Injection  
of Mocked Object  
(Modern PHP)



# Winding Up: Recap

PHPUnit: `./vendor/bin/phpunit`

Coverage Report: `--coverage-html [path]`

CRAP: Reflects maintainability

Functional vs. Unit Testing: Systems vs. Code

Patterns: Isolation, Data providers, Mocking

# Thank you!

Paul Mitchum

Mile23 on Drupal.org

@PaulMitchum on Twitter

paul-m on GitHub

<http://mile23.com/phpunit-talk>