Demeanor Documentation

Release 0.5

Christopher Davis

September 07, 2014

Contents

Demeanor is PHP testing framework that supports several different test formats.

1. Specification Tests (SpecBDD)
2. Unit Tests
3. PHPT Tests

CHAPTER 1

Contents

1.1 Configuration

Demeanor use JSON for configuration and looks for the configuration files `demeanor.json` and `demeanor.dist.json` by default. A custom configuration file can be used by using the `-c` or `--config` command line options.

```
./vendor/bin/demeanor --config a_custom_config.json
```

1.1.1 Test Suites

The centerpiece of Demeanor's configuration is the `testsuites` argument. If no suites are defined, the command line test runner will error. Similarly if `testsuites` isn't a JSON object, the runner will error.

Each test suite can have it's own bootstrap file(s) as well as define it's own test locations.

Here's a complete example:

```
{
    "testsuites": {
        "A Test Suite": {
            "type": "unit",
            "bootstrap": [
                "test/bootstrap.php"
            ],
            "directories": [
                "test/classes",
                "test/another_directory"
            ],
            "files": [
                "test/path/to/a/file.php"
            ],
            "glob": [
                "test/files/*Test.php"
            ],
            "exclude": {
                "directories": [
                    "test/classes/not_this_one"
                ]
                , "files": [
                    "test/path/to/exclude.php"
                ]
```

```
                , "glob": [
                    "test/exclude/*.php"
                ]
            }
        }
    }
}
```

- `type` is the type of test suite. This just tells demeanor what to do with the suite itself. Valid values are `unit`, `spec`, and (eventually) `story`.
- `bootstrap` is a list of files that will be `require_once`'d before of the suites tests are run. Use these files to do any setup for the test suite.
- `directories` tells demeanor to look for files in a directory. What files it looks for depends on the suite type.
  - `unit` test suites look for files that end in `Test.php`
  - `spec` test suites look for files that end in `spec.php`
- `files` is a list of files that will be treated as if they contain test cases.
- `glob` is a list of glob patterns that will be used to locate test files.
- `exclude` is used to blacklist files from your test suite. It's an object that looks very similar to the test suite itself. `directories`, `files`, and `glob` work exactly as they do in the test suite itself.

1.1.2 Default Test Suites

There's a good chance you won't want to run all your test suites all the time. For instance, acceptance tests often take a long time – they'll test your complete system end to end.

That's where the `default-suites` configuration option comes in. When defined only the test suites defined in it's array (or string) will be run with the naked `demeanor` command.

`default-suites` may be an array.

```
{
    "default-suites": ["a_suite"],
    "testsuites": {
        "a_suite": {
            "type": "spec",
            "directories": [
                "test/spec"
            ]
        }
    }
}
```

Or it can just be a string.

```
{
    "default-suites": "a_suite",
    "testsuites": {
        "a_suite": {
            "type": "spec",
            "directories": [
                "test/spec"
            ]
        }
```

```
    }
}
```

If a suite that doesn't exist is supplied, the `demeanor` CLI will fail.

```
{
    "default-suites": "this-will-not-work",
    "testsuites": {
        "a_suite": {
            "type": "spec",
            "directories": [
                "test/spec"
            ]
        }
    }
}
```

How can I run other test suites then?

Use the `--testuite` (or `-s`) command line option.

```
./vendor/bin/demeanor --testsuite a_suite
```

Or use a few of them.

```
./vendor/bin/demeanor -s a_suite -s another_suite
```

Or use the `--all` (or `-a`) option to run all test suites.

```
./vendor/bin/demeanor --all
```

1.1.3 Subscribers

`subscribers` can be defined in `demeanor.json` to add event subscribers to that hook in and change how the test runs work.

`subscribers` should be a list of class names that implement `Demeanor\Event\Subscriber`.

```
{
    "subscribers": [
        "Acme\\Example\\TestSubscriber"
    ],
    "testsuites": {
        ...
    }
}
```

These subscribers should have a argumentless constructor. Demeanor uses the event subscriber API itself, look in the `src/Subscriber` directory of the demeanor repo for examples.

1.1.4 Code Coverage

Demeanor's code coverage uses a whitelist of files to generate reports. Unless `coverage` is defined in `demeanor.json` no coverage will be reported on.

`coverage` looks very close to a test suite.

```
{
    "coverage": {
        "reports": {
            "text": "coverage/coverage.txt",
            "html": "coverage/html_dir",
            "diff": "coverage/diff_dir"
        },
        "directories": [
            "src/"
        ],
        "files": [
            "path/to/a/file.php"
        ],
        "glob": [
            "files/*.php"
        ],
        "exclude": {
            "directories": [
                "src/NotThisOne"
            ],
            "files": [
                "path/to/a/file/excluded.php"
            ],
            "glob": [
                "files/nothere/*.php"
            ]
        }
    }
}
```

- `directories` is a list of directories that will be search for all files ending with `.php`
- `files` and `glob` work as described in the *test suites*
- `exclude` can be used to leave files out of the coverage report. All of its keys (`directories`, `files`, and `glob`) work the same as described in the two points above.
- `reports` is really the only coverage specific part of the configuration. It defines a set of coverage reports with the report type as the key and an output path as the value. Reports is optional, report options can be specified from the CLI.

If no directories, files, or glob keys are provided, Demeanor will no generate any coverage reports or may generate an empty `index.html`.

Please see *Code Coverage* for a more complete guide to report types.

Command Line Coverage Configuration

- The `--no-coverage` option will completely disable collection and rendering of coverage reports.
- `--coverage-html` tells Demeanor to use a the supplied directory to output a html report.
- `--coverage-text` tells Demeanor use the supplied file path to write a text report.
- `--coverage-diff` tells Demeanor to use the supplied directory to output a diff report.

Some examples:

```
# disable coverage completely
./vendor/bin/demeanor --no-coverage
```

```
# output an HTML report to the coverage directory
./vendor/bin/demeanor --coverage-html=coverage

# output a diff reprot to the coverage directory
./vendor/bin/demeanor --coverage-diff=coverage

# output a text report to the file coverage.txt
./vendor/bin/demeanor --coverage-text=coverage.txt
```

1.1.5 Filtering Test Cases

Filters allow you to include (and in some cases, exclude) certain tests. Filters require a consensus: all filters must be met for a test case to run.

- The `--filter-name` option will allow only test cases that contain the provided string in their name(s).

Examples:

```
# run tests with 'SomeTest' in their name(s)
./vendor/bin/demeanor --filter-name SomeTest

# run tests with 'SomeTest' and 'config' in their names
./vendor/bin/demeanor --filter-name SomeTest --filter-name config
```

Including and Excluding Groups

- The `--include-group` will exclude all tests but those in a given group.
- The `--exclude-group` will exclude any tests in a given group

Some examples:

```
# include a single group
./vendor/bin/demeanor --include-group aGroup

# include multiple groups
./vendor/bin/demeanor --include-group aGroup --include-group anotherGroup

# exclude a single group
./vendor/bin/demeanor --exclude-group aGroup

# exclude multiple groups
./vendor/bin/demeanor --exclude-group aGroup --exclude-group anotherGroup
```

Including only Certain Files and Directories

Passing paths as arguments to the demeanor command line tool will include only those paths in a run.

```
# include only tests in the test/unit/Config directory
./vendor/bin/demeanor test/unit/Config

# include only tests in a single file
./vendor/bin/demeanor test/unit/Config/ConsoleConfigurationTest.php

# include tests in either the directory test/unit/Config or test/unit/Spec
```

```
./vendor/bin/demeanor test/unit/Config/ test/unit/Spec/

# include tests in either the directory test/unit/Config or in the file
# test test/unit/Spec/SpecTestCaseTest.php
./vendor/bin/demeanor test/unit/Config test/unit/Spec/SpecTestCaseTest.php
```

1.2 The Test Context

Demeanor has a concept of a test context that's defined by the `Demeanor\TestContext` interface. This context object is **always** the first argument to any test method or callback.

The test context simply lets you pass messages back to the test case – telling it to fail, skip, expect and exception, or log a message. Additionally the context implements `ArrayAccess` so you can use it to store values for the duration of the test run.

Here's an example for a unit test.

```php
<?php
// SomeTest.php

use Demeanor\TestContext;

class SomeTest
{
    public function testStuff(TestContext $ctx)
    {
        $ctx->skip('some reason why the test is to be skipped');
    }

    public function testThisWillBeFailed(TestContext $ctx)
    {
        $ctx->fail('The reason for the failure here');
    }

    public function testLogs(TestContext $ctx)
    {
        $ctx->log('some message here');
    }

    public function testExpectException(TestContext $ctx)
    {
        $ctx->expectException('Exception');
        throw new \Exception('this does not cause the test to fail');
    }
}
```

Of these `log` is probably the more interesting. These message are not shown to the user unless they've cranked up the verbosity on the command line test runner (with the `-v|-vv|-vvv` options) or the test fails.

The same test context object that's passed to the actually test method/callback is also passed to the before and after callbacks. This is very useful for spec-style tests.

```php
<?php
// SomeClass.spec.php

use Demeanor\TestContext;
```

```
$this->before(function (TestContext $ctx) {
    $ctx['something'] = createSomeObjectHere();
});

$this->after(function (TestContext $ctx) {
    $ctx['something']->cleanup();
});

$this->it('should do something', function (TestContext $ctx) {
    // use $ctx['something'] here
});
```

Or using *Annotations* with unit tests.

```
<?php
// AnotherTest.php

use Demeanor\TestContext;

/**
 * @Before(method="setUp")
 * @After(method="tearDown")
 */
class AnotherTest
{
    public function setUp(TestContext $ctx)
    {
        $ctx['something'] = createSomeObjectHere();
    }

    public function tearDown(TestContext $ctx)
    {
        $ctx['something']->tearDown();
    }

    public function testSomething(TestContext $ctx)
    {
        // use $ctx['something']
    }
}
```

1.3 Writing Spec Tests

Demeanor's spec test API is heavily inspired by tools like RSpec and Jasmine.

1.3.1 Spec Test Basics

A spec test is a file that is `included` inside an instance of `Demeanor\Spec\Specification`. Demeanor, by default, looks for files that end in `.spec.php`, the part of the file before that suffix is used as the initial test name.

```
<?php
// SomeTest.spec.php

// by default Demeanor does this to set up the initial test name/context
// It's simply the file name minus the '.spec.php'
```

```
$this->describe('SomeTest');

// of course, if you don't like that, it can be changed.
$this->describe('Some Cool Feature');

// generally spec files contain one or more calls to `it`. Whenever a call
// to `it` happens, a test case is created. The first argument of `it` is
// used as part of the name of the test
$this->it('should be true', function () {
    Assert::assertTrue(true, 'true is not true, something is horribly wrong');
});

$this->it('should be false', function () {
    Assert::assertFalse(false, 'false is not false, something is horribly wrong');
});
```

1.3.2 Grouping Tests with Describe

As explained above, when demeanor includes a file, it calls `describe` with part of the file name. This create a group of tests. This is really nothing more than prefixing the name of the test. The example above would generate two test cases with the names:

- [Some Cool Feature] should be true
- [Some Cool Feature] should be false

Demeanor does this for you automatically, but sometimes futher grouping is required. For instance, a given feature or behavior might have several sub-features or behaviors that should be grouped. This can be done by nesting calls to `describe`.

```
<?php
// SomeOtherTest.spec.php

use Counterpart\Assert;
use Demeanor\TestContext;

// When describe is given a second argument, a closure, it creates an entirely
// new instance of `Demeanor\Spec\Specification` to use
$this->describe('#SomeSubFeature', function () {
    $this->it('will throw an exception', function (TestContext $ctx) {
        $ctx->expectException('Exception');
        throw new \Exception('broken');
    });

    $this->it('will equal zero', function () {
        Assert::assertEquals(0, 0);
    });
});
```

This will generate two test cases with the following names:

1. [SomeOtherTest#SomeSubFeature] will throw an exception
2. [SomeOtherTest#SomeSubFeature] will equal zero

Describe can be nested as many times as you wish.

1.3.3 Before & After Callbacks

Sometimes you need to set up or tear down state before and after a test. This can done by calling `before` and `after` in your spec file.

```
<?php
// BeforeAfter.spec.php

use Counterpart\Assert;
use Demeanor\TestContext;

$this->before(function (TestContext $ctx) {
    // use the test context to pass values
    $ctx['one'] = true;
});

$this->after(function (TestContext $ctx) {
    unset($ctx['one']);
});

$this->it('should have values set for before and after', function (TestContext $ctx) {
    Assert::assertTrue($ctx['one']);
});
```

Unfortunately `before` and `after` come with some limitations. They are dependent on their position within the file related to the test cases. For instance: if a call to `after` is later in the file than a call to `it` that after callback will not be run on earlier test cases.

```
<?php
// BeforeAfter.spec.php

use Demeanor\TestContext;

$this->before(function (TestContext $ctx) {
    // ...
});

$this->it('should be a test', function (TestContext $ctx) {
    // test code here
});

$this->after(function (TestContext $ctx) {
    // this will never be run
});
```

1.3.4 Test Groups/Tags

Apart from grouping tests with `describe`, test groups are ways to *tag* test so they can be easily run or excluded later. See *Test Groups* for more information.

1.3.5 Describe with Before, After, and Group

When `describe` is called inside a spec file, the new `Demeanor\Spec\Specification` object that's created for it will inherit all of the before and after callbacks as well as any groups from the outer scope.

```php
<?php
// BeforeAfterDescribe.spec.php

use Counterpart\Assert;
use Demeanor\TestContext;

$this->group('aGroup');

$this->before(function (TestContext $ctx) {
    // this will run before EVERY test case in this file, even the
    // ones inside another `describe` call
    $ctx['one'] = true;
});

// this test will be placed in `aGroup`
$this->it('has a before callback and group', function (TestContext $ctx) {
    Assert::assertTrue($ctx['one']);
});

$this->describe('#NestedDescribe', function () {
    // this test will also be placed in `aGroup`
    $this->it('also has a before callback and group', function (TestContext $ctx) {
        Assert::assertTrue($ctx['one']);
    });
});
```

The relationship doesn't work the other way, however. Before and after callbacks inside a `describe` are jailed there.

```php
<?php
// BeforeAfterDescribe.spec.php

use Counterpart\Assert;
use Demeanor\TestContext;

$this->describe('#NestedDescribe', function () {
    $this->group('inner group');

    $this->before(function (TestContext $ctx) {
        $ctx['one'] = true;
    });

    $this->it('has a before callback and group', function (TestContext $ctx) {
        Assert::assertTrue($ctx['one']);
    });
});

$this->it(
    'does not share a the same before and group with the inner spec',
    function (TestContext $ctx) {
        Assert::assertArrayDoesNotHaveKey('one', $ctx);
    }
);
```

1.4 Writing XUnit-Style Tests

XUnit-style tests are methods inside a class.

1.4.1 Convention over Configuration & Inheritance

Rather than forcing you to extend a `TestCase` base class demeanor favors a naming convention for test classes: they must end with the suffix `Test`.

```
<?php

// will be treated as a container for test cases
class SomeTest
{
    // ...
}

// demeanor will ignore this class
class Something
{
    // ...
}
```

Additionally all test methods must start with `test` and be public.

```
<?php

class SomeTest
{
    // will be turned into a test case
    public function testSomeObjectDoesStuff()
    {
        // ...
    }

    // not a test
    public function someObjectDoesOtherStuff()
    {

    }

    // also not a test
    private function testPrivateMethodsAreIgnored()
    {

    }
}
```

1.4.2 Using Counterpart Assertions

Starting with Counterpart 1.4, `Counterpart\Assert` and `Counterpart\Matchers` are traits. You can embed them in your test classes.

```
<?php

class SomeOtherTest
{
    use \Counterpart\Assert;
    use \Counterpart\Matchers;

    public function testSomething()
```

```
    {
        $this->assertTrue(true);
        // instead of Assert::assertTrue(true)

        $this->assertThat($this->arrayHasKey('one'), ['one' => true]);
        // instead of Assert::assertThat(Matchers::arrayHasKey('one'), ['one' => true]);
    }
}
```

1.4.3 Annotations

The *Annotations* documentation has a ton of information about using annotations to modify and change the behavior of unit tests.

1.5 Writing PHPT Tests

`phpt` is a special file layout that the php core uses for its tests.

1.5.1 PHPT Basics

A `.phpt` file is separated into sections by headers that look like `--([A-Z]+)--` where `([A-Z]+)` is replaced with some sequents of one or more uppercase characters.

A bare minimum of the follow sections are required.

- `--TEST--`: Describes the test
- `--FILE--`: The actual php code to run (including the open `<?php` tag)
- `--EXPECT--` or `--EXPECTF--`: Something to match the output of the `--FILE--` section againts.

Here is a valid `.php` file

```
--TEST--
This describe what the file is meant to test: outputing hello world.
--FILE--
<?php
// You need the open php tag here
echo 'Hello, World';
--EXPECT--
Hello, World
```

This is a test that would pass. Why? Because Demeanor will execute the code in the `--FILE--` section in a separate PHP process and compare it with the `--EXPECT--` section. If they match: test passes.

`--EXPECTF--` can also be used to match output. This is a details of `--EXPECTF--`'s format can be found [here](http://qa.php.net/phpt_details.php), but here's quick overview.

- %e: A directory separator (DIRECTORY_SEPARATOR)
- %s: One or more of anything (character, whitespace, etc) except the end of line character. [^rn]+
- %S: Zero or more of anything (character, whatespace, etc) except the end of line character. [^rn]*
- %a: One or more of anything (character, whitespace, etc) including the end of line character. .+
- %A: Zero or more of anything, including the end of line character. .*

- %w: Zero or more whitespace characters. s*
- %i: A signed integer value (+123, -123). [+-]?d+
- %d: An unsigned integer value. d+
- %x: One or more hexadecimal character. [0-9a-fA-F]+
- %f: A floating point number. [+-]?.?d+.?d*(?:[Ee][+-]?d+)?
- %c: A single character. .
- %unicodelstring% or %stringlunicode%: Matches 'string'
- %binary_string_optional% and %unicode_string_optional%: Matches 'string'
- %ulb% or %blu%: replaced with nothing

We could rewrite the example above to use `--EXPECTF--` so it doesn't care whether it sees "World" or anything else

```
--TEST--
This describe what the file is meant to test: outputing hello world.
--FILE--
<?php
// You need the open php tag here
echo 'Hello, World';
--EXPECT--
Hello, %s
```

1.5.2 Skipping PHPT Tests

Use the `--SKIPIF--` section. This is a bit of code that will be pased to a separate PHP process. If the output from it start with *skip* the test will be skipped.

```
--TEST--
Only Runs on php less than 5.4
--SKIPIF--
<?php if (version_compare(phpversion(), '5.4', '<')) {
    echo 'skip on php less than 5.4';
}
--FILE--
<?php
// test code here
--EXPECT--
some sort of output
```

1.5.3 Cleaning Up

Use the `--CLEAN--` section to clean up after yourself. Please note that the `--CLEAN--` section is **not** passed to the same PHP process as the `--FILE--`, so you can expect it to have the same variables available

1.5.4 Sharing Environment

If a file has the optional `--ENV--` section, it will parsed into an associative array and passed to all PHP processes as environment variables.

```
--TEST--
Test with environment
--ENV--
FROM_PHPT_ENV=1
--FILE--
<?php
var_dump(getenv('FROM_PHT_ENV'));
--EXPECTF--
%string|unicode%(%d) "1"
```

The `FROM_PHPT_ENV` will be available (via `getenv` or `$_ENV`, depending on your php settings) in `--SKIPIF--`, `--FILE--`, and `--CLEAN--`.

1.5.5 Does Demeanor Support All PHPT Features?

Definitely not. The PHP core's `run-tests.php` is still much, much more complete. Demeanor just barely does an impression of the phpt functionality found there.

1.6 Code Coverage

The *code coverage* configuration section has some detailed documentation on how to configure code coverage.

1.6.1 A Warning

Demeanor does code coverage *only* based on a whitelist as defined in the *configuragion*.

If no `directories`, `files`, or `glob` patterns are defined, no real coverage reporting will be generated. These things *must* be set up in `demanor.json`.

1.6.2 Types of Coverage Reports

Demeanor can generate several types of coverage reports. There are three valid types that may be used in the `reports` configuration key or specified via the *CLI*.

1. html
2. diff
3. text

Unrecognized types are simply ignored.

HTML Coverage Reports

These are HTML files that show covered lines in green along with the filename and a percent covered number.

Coverage Index:

Code Coverage Report

Filename	Covered
/Users/chris/Code/demeanor/src/AbstractTestCase.php	18.421%
/Users/chris/Code/demeanor/src/AbstractTestSuite.php	0.000%
/Users/chris/Code/demeanor/src/Annotation/After.php	10.000%
/Users/chris/Code/demeanor/src/Annotation/Annotation.php	13.235%
/Users/chris/Code/demeanor/src/Annotation/Before.php	10.000%
/Users/chris/Code/demeanor/src/Annotation/Callback.php	19.672%
/Users/chris/Code/demeanor/src/Annotation/DataProvider.php	25.806%
/Users/chris/Code/demeanor/src/Annotation/Expect.php	28.125%
/Users/chris/Code/demeanor/src/Annotation/Requirement.php	23.729%
/Users/chris/Code/demeanor/src/Command.php	42.157%
/Users/chris/Code/demeanor/src/Config/Configuration.php	0.000%
/Users/chris/Code/demeanor/src/Config/ConsoleConfiguration.php	29.825%
/Users/chris/Code/demeanor/src/Config/JsonConfiguration.php	42.740%
/Users/chris/Code/demeanor/src/Console.php	0.000%
/Users/chris/Code/demeanor/src/Coverage/Collector.php	20.611%
/Users/chris/Code/demeanor/src/Coverage/Driver/Driver.php	0.000%
/Users/chris/Code/demeanor/src/Coverage/Driver/DriverFactory.php	0.000%
/Users/chris/Code/demeanor/src/Coverage/Driver/NullDriver.php	0.000%
/Users/chris/Code/demeanor/src/Coverage/Driver/XdebugDriver.php	4.167%
/Users/chris/Code/demeanor/src/Coverage/Report/DiffReport.php	29.032%
/Users/chris/Code/demeanor/src/Coverage/Report/FileBasedReport.php	20.930%
/Users/chris/Code/demeanor/src/Coverage/Report/HtmlReport.php	42.105%
/Users/chris/Code/demeanor/src/Coverage/Report/html_templates/index.php	24.324%
/Users/chris/Code/demeanor/src/Coverage/Report/html_templates/single.php	55.263%
/Users/chris/Code/demeanor/src/Coverage/Report/NullReport.php	0.000%
/Users/chris/Code/demeanor/src/Coverage/Report/Report.php	5.000%
/Users/chris/Code/demeanor/src/Coverage/Report/ReportFactory.php	0.000%

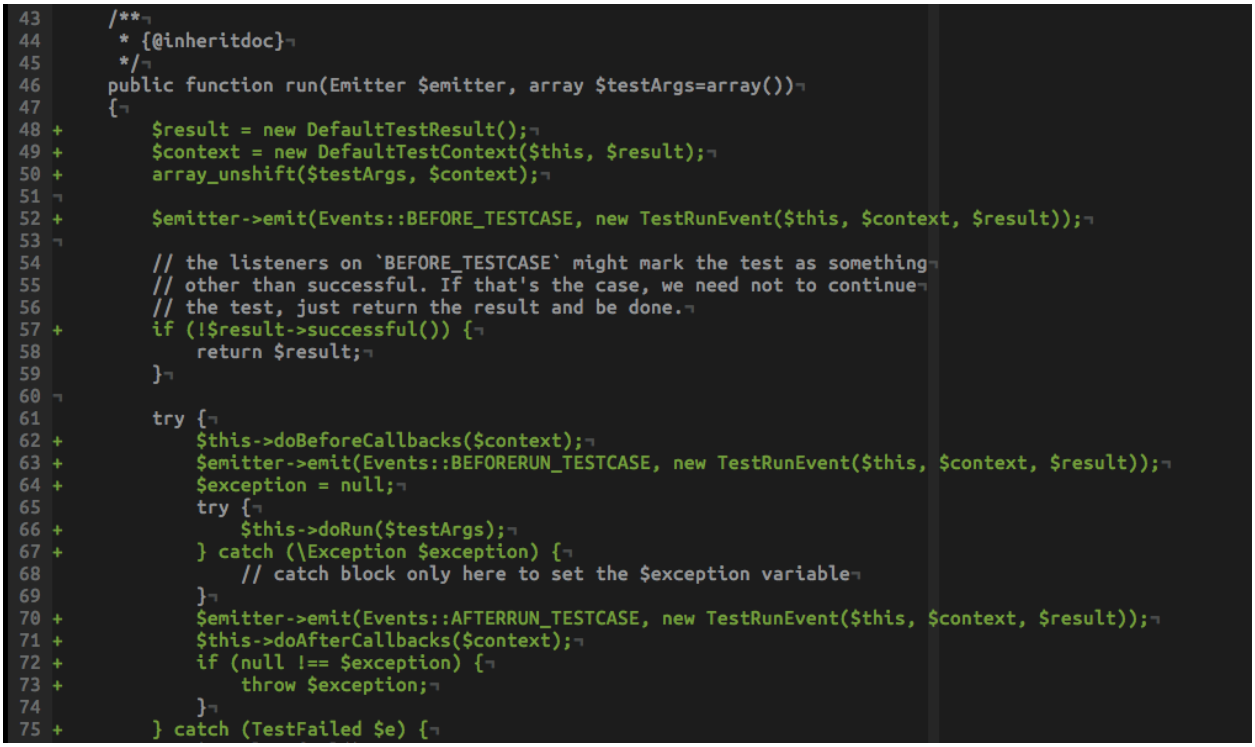
Coverage for a single file:

```
    protected $dataProvider = null;

    /**
     * {@inheritdoc}
     */
    public function run(Emitter $emitter, array $testArgs=array())
    {
        $result = new DefaultTestResult();
        $context = new DefaultTestContext($this, $result);
        array_unshift($testArgs, $context);

        $emitter->emit(Events::BEFORERUN_TESTCASE, new TestRunEvent($this, $context, $result));
        $exception = null;
        try {
            $this->doRun($testArgs);
        } catch (\Exception $exception) {
            // catch block only here to set the $exception variable
        }
        $emitter->emit(Events::AFTERRUN_TESTCASE, new TestRunEvent($this, $context, $result));
```

```
[Demeanor\Phpt\PhptTestCaseTest] Phpt With Successful Skip Code Marks Test As Skipped
    /Users/chris/Code/demeanor/test/unit/Phpt/PhptTestCaseTest.php:38
[Demeanor\Phpt\PhptTestCaseTest] Phpt With Umatching Expectf Marks Test As Failed
    /Users/chris/Code/demeanor/test/unit/Phpt/PhptTestCaseTest.php:53
[Demeanor\Phpt\PhptTestCaseTest] Phpt With Unmatching Expect Marks Test As Failed
    /Users/chris/Code/demeanor/test/unit/Phpt/PhptTestCaseTest.php:67
[Demeanor\Phpt\PhptTestCaseTest] Phpt With Clean Calls Executor More Than Once
    /Users/chris/Code/demeanor/test/unit/Phpt/PhptTestCaseTest.php:81
[Demeanor\Phpt\PhptTestCaseTest] Phpt With Ini Section Passes Ini To Executor
    /Users/chris/Code/demeanor/test/unit/Phpt/PhptTestCaseTest.php:98
```

Diff Coverage Reports

Diff coverage reports are a series of "diff" files. Each generated file has a + in front of any line that was covered. While HTML reports include information about what test covered what line, diff reports don't.



```
43      /**
44       * {@inheritdoc}
45       */
46      public function run(Emitter $emitter, array $testArgs=array())
47      {
48 +        $result = new DefaultTestResult();
49 +        $context = new DefaultTestContext($this, $result);
50 +        array_unshift($testArgs, $context);
51
52 +        $emitter->emit(Events::BEFORE_TESTCASE, new TestRunEvent($this, $context, $result));
53
54          // the listeners on `BEFORE_TESTCASE` might mark the test as something
55          // other than successful. If that's the case, we need not to continue
56          // the test, just return the result and be done.
57 +        if (!$result->successful()) {
58              return $result;
59          }
60
61          try {
62 +            $this->doBeforeCallbacks($context);
63 +            $emitter->emit(Events::BEFORERUN_TESTCASE, new TestRunEvent($this, $context, $result));
64 +            $exception = null;
65              try {
66 +                $this->doRun($testArgs);
67 +            } catch (\Exception $exception) {
68                  // catch block only here to set the $exception variable
69              }
70 +            $emitter->emit(Events::AFTERRUN_TESTCASE, new TestRunEvent($this, $context, $result));
71 +            $this->doAfterCallbacks($context);
72 +            if (null !== $exception) {
73 +                throw $exception;
74              }
75 +        } catch (TestFailed $e) {
```

Text Coverage Reports

A test coverage report is simply a list of files with their percent covered numbers

```
/Users/chris/Code/demeanor/src/AbstractTestCase.php 18.421%
/Users/chris/Code/demeanor/src/AbstractTestSuite.php 0.000%
/Users/chris/Code/demeanor/src/Annotation/After.php 10.000%
/Users/chris/Code/demeanor/src/Annotation/Annotation.php 13.235%
/Users/chris/Code/demeanor/src/Annotation/Before.php 10.000%
/Users/chris/Code/demeanor/src/Annotation/Callback.php 19.672%
/Users/chris/Code/demeanor/src/Annotation/DataProvider.php 25.806%
/Users/chris/Code/demeanor/src/Annotation/Expect.php 28.125%
/Users/chris/Code/demeanor/src/Annotation/Requirement.php 23.729%
/Users/chris/Code/demeanor/src/Command.php 42.157%
/Users/chris/Code/demeanor/src/Config/Configuration.php 0.000%
/Users/chris/Code/demeanor/src/Config/ConsoleConfiguration.php 29.825%
/Users/chris/Code/demeanor/src/Config/JsonConfiguration.php 42.740%
```

A few special values can be used for the text report output path.

`STDERR` will write the report to, as one might expect, stderr.

```
# write a coverage report to stderr
./vendor/bin/demeanor --coverage-text STDERR
```

`STDOUT` will write the report to stdout.

```
# write a coverage report to stdout
./vendor/bin/demeanor --coverage-text STDOUT
```

1.7 Annotations

Demeanor uses a simple annotation library to make it a bit easier to configure unit and spec tests.

Annotations are case sensitive.

1.7.1 Where to Put Annotations

Annotations can be defined on the test class or test method. Annotations on the class will be applied to all test methods in the class. A great use case for this is running a method before every test.

```
<?php
// AnnotationTest.php

/**
 * @Before(method="setUp")
 */
class AnnotationTest
{
    public function setUp()
    {

    }
}
```

Additionally, annotations can be used with *specification tests*. The annotation *must* be in a docblock directly before a call to `it`.

```
<?php

/**
 * @Excpect("LogicException")
 */
$this->it('will throw a logic exception', function () {
    throw new \LogicException();
});
```

1.7.2 Adding Before/After Callbacks

Only supported in :doc:‘unit test cases <types/unit-tests>‘.

The `Before` and `After` annotations provide ways to call methods on the test class or some function before and after each test case.

```
<?php
// BeforeTest.php

function run_before_example()
{

}

class BeforeTest
{
    public function setUp()
    {
```

```
    }

    /**
     * These two are the same:
     * @Before(method="setUp")
     * @Before("setUp")
     *
     * @Before(function="run_before_example")
     */
    public function testStuff()
    {
        // the `setUp` method is run before the `testStuff` method
        // the `run_before_example` function is also run before the
        // `testStuff method
    }
}
```

Both `Before` and `After` can take a `method` OR `function` argument. As you might expect, `method` calls a method on the test class before the test run and `function` calls a function.

Nothing will be added if the `method` doesn't exist or isn't public or if the function doesn't exist.

1.7.3 Expecting Exceptions

The `Expect` annotation can be used instead of calling `TestContext::expectException` in the test method. `Expect` requires the `exception` argument to work.

```
<?php
// ExpectTest.php

use Demeanor\TestContext;

class ExpectTest
{
    /**
     * These two are the same:
     * @Expect("InvalidArgumentException")
     * @Expect(exception="InvalidArgumentException")
     */
    public function testDoingSomethingThrowsException(TestContext $ctx)
    {
        // same as calling $ctx->expectException('InvalidArgumentException');
    }
}
```

If the class name in the `exception` argument doesn't exist, the test will be errored and will show an error message saying that the exception class wasn't found.

1.7.4 Specifying Requirements

See the *Requirements* documentation for information about using annotations to specify requirements.

1.7.5 Data Providers

Data providers can also be specified with annotations. Details on them can be found on the *data providers* page.

1.7.6 Groups

Test Groups must be specified on unit tests using annotations. See the *group* documentation for more information.

1.8 Data Providers

Data providers let you run the same test with multiple sets of arguments. Any `TestCase` implementation can have a provider, but currently they are only really first class citizens in the unit test world on Demeanor.

1.8.1 Annotations

Data providers are set on unit tests using the `@Provider` annotation.

Data providers can be one of three types:

1. A static method on the test class – `@Provider(method="someMethod")` or `@Provider("someMethod")`
2. A function – `@Provider(function="a_provider_function")`
3. Inline
   - `@Provider(data=["one", "two"])`
   - `@Provider(data={aKey: ["data", "set"], anotherKey: "dataset"})`

1.8.2 Static Method Data Provider

This can only be done for *unit tests*.

```
<?php
use Demeanor\TestContext;
use Counterpart\Assert;

class DataProviderMethodTest
{
    public static function aProvider()
    {
        return [
            'one',
            'two',
        ];
    }

    /**
     * These two are the same:
     * @Provider("aProvider")
     * @Provider(method="aProvider")
     */
    public function testWithMethodProvider(TestContext $ctx, $arg)
    {
        Assert::assertType('string', $arg);
    }
}
```

1.8.3 Function Data Provider

```
<?php
use Demeanor\TestContext;
use Counterpart\Assert;

function acceptance_dataprovider_function()
{
    return [
        'one',
        'two',
    ];
}

// DataProviderFunctionTest.php
class DataProviderFunctionTest
{
    /**
     * @Provider(function="acceptance_dataprovider_function")
     */
    public function testWithFunctionProvider(TestContext $ctx, $arg)
    {
        Assert::assertType('string', $arg);
    }
}

// some.spec.php
/**
 * @Provider("acceptance_dataprovider_function")
 */
$this->it('has a data provider', function (TestContext $ctx, $arg) {
    Assert::assertType('string', $arg);
});
```

1.8.4 Inline Data Provider

```
<?php

// DataProviderInlineTest.php

use Demeanor\TestContext;
use Counterpart\Assert;

class DataProviderInlineTest
{
    /**
     * @Provider(data=["one", "two"])
     */
    public function testWithDataProviderAsIndexedArray(TestContext $ctx, $arg)
    {
        Assert::assertType('string', $arg);
    }

    /**
     * @Provider(data={aSet: "one", anotherSet: "two"})
     */
    public function testWithDataProviderAsAssociativeArray(TestContext $ctx, $arg)
```

```
    {
        Assert::assertType('string', $arg);
    }
}

// some.spec.php
/**
 * @Provider(["one", "two"]);
 */
$this->it('has a data provider', function (TestContext $ctx, $arg) {
    Assert::assertType('string', $arg);
});
```

1.8.5 The Test Context

Notice the the *test context* is *always* the first argument to test methods. In Demeanor the context object is important, and any data provider arguments will come after it.

1.9 Test Groups

Groups provide a way to mark test with something like a tag. Some use cases for this:

1. You might have a regression test suite and mark each test with an issue number.
2. Components that use external libraries might have their tests tagged with the library name.
3. Some tests are slow, so you might tag them with *slow*.

In general groups make it easy to run only certain tests from the *CLI*. Demeanor doesn't support group definition in the configuration file(s), only in test code itself.

1.9.1 Unit Test Groups

Unit tests can be grouped with the `@Group` annotation.

```
<?php

/**
 * Mark all test methods in this class with the 'slow' group:
 * @Group("slow")
 */
class SomeTest
{
    /**
     * Multiple groups can be added at once:
     * @Group("oneGroup", 'anotherGroup')
     *
     * This is the same as:
     * @Group('oneGroup')
     * @Group('anotherGroup')
     */
    public function testSomething()
    {
```

```
    }
}
```

1.9.2 Spec Test Groups

Spec tests can be grouped by call `$this->group()` inside a spec test file. Just like before and after callbacks, child tests inherit their parent's groups.

```
<?php
// some_test.spec.php

$this->group('aGroup');

$this->it('should have a group', function () {
    // this test will be in `aGroup`
});

$this->describe('something else', function () {
    $this->group('anotherGroup');

    $this->it('should have two groups', function () {
        // this test will be in `aGroup` and `anotherGroup`
    });
});
```

The `@Group` annotation may also be used, but does not cause the inheritance that `$this->group()` does.

```
<?php
// some.spec.php

/**
 * @Group("AGroup")
 */
$this->it('has a group', function () {

});
```

1.10 Requirements

Requirements conditions that have to be met for a test to run. Things like PHP version, a required extension, or a specific OS.

1.10.1 Setting Requirements via the Test Context

The `TestContext` object for a test case will have the key `requirements` that can be used used to add new requirements to a test. This should be done in a before callback.

Unit Test Example

```
<?php
// SomeTest.php
```

```
use Demeanor\TestContext;
use Demeanor\Extension\Requirement\VersionRequirement;
use Demeanor\Extension\Requirement\RegexRequirement;
use Demeanor\Extension\Requirement\ExtensionRequirement;

class SomeTest
{
    public function beforeTest(TestContext $ctx)
    {
        // require PHP 5.4
        $ctx['requirements']->add(new VersionRequirement('5.4'));

        // requires a specific verison of some other software
        $ctx['requirements']->add(new VersionRequirement('1.0', getTheVersion(), 'Software Name'));

        // require a specific OS
        $ctx['requirements']->add(new RegexRequirement('/darwin/u', PHP_OS, 'operating system'));

        // require an extension
        $ctx['requirements']->add(new ExtensionRequirement('apc'));
    }

    /**
     * @Before(method="beforeTest")
     */
    public function testSomeStuff()
    {
        // requirements are checked before this is run
    }
}
```

Unit test requirements can also be set with an annotation. See the *Annotations* documentation for examples.

Spec Test Example

```
<?php
// Some.spec.php

use Demeanor\TestContext;
use Demeanor\Extension\Requirement\VersionRequirement;
use Demeanor\Extension\Requirement\RegexRequirement;
use Demeanor\Extension\Requirement\ExtensionRequirement;

$this->before(function (TestContext $ctx) {
    // require PHP 5.4
    $ctx['requirements']->add(new VersionRequirement('5.4'));

    // requires a specific verison of some other software
    $ctx['requirements']->add(new VersionRequirement('1.0', getTheVersion(), 'Software Name'));

    // require a specific OS
    $ctx['requirements']->add(new RegexRequirement('/darwin/u', PHP_OS, 'operating system'));

    // require an extension
    $ctx['requirements']->add(new ExtensionRequirement('apc'));
});
```

```
$this->it('should do something', function (TestContext $ctx) {
    // requirements are checked before this is run
});
```

1.10.2 Setting Requirements via Annotations

Unit test requirements can be set with an annotation. These are limited to PHP version, OS, and extension requirements.

```
<?php
// SomeOtherTest.php

use Demeanor\TestContext;

class SomeOtherTest
{
    /**
     * @Require(php="5.4", os="/darwin/u", extension="apc")
     */
    public function testSomeStuff()
    {
        // requirements are checked before this is run
    }

    /**
     * Or each Require annotation can be separate
     *
     * @Require(php="5.4")
     * @Require(os="/darwin/u")
     * @Require(extension="apc")
     */
    public function testSomeOtherStuff()
    {
        // requirements are checked before this is run
    }
}
```

Requirement annotations can also be done on spec tests.

```
<?php
// some.spec.php

/**
 * @Require(php="5.4")
 */
$this->it('requires php 5.4+', function () {

});
```

1.11 Mock Objects

Use Mockery. Demeanor takes care of verifying mock expectations for you.

1.12 Test Result Statuses

Every `Demeanor\TestCase` implementation produces an implementation of `Demeanor\TestResult` when it's run. `TestResult` can have one of many statuses that are explained here.

1.12.1 Successful

The test worked! To demeanor this just means that no exceptions were thrown during the core of the test run.

1.12.2 Skipped

The test was explicitly marked as skipped via `Demeanor\TestContext::skip` or some sort of requirement for the tests execution was not met.

A skipped test does not cause the Demanor CLI to exit with a failure code.

Users can skip tests if some precondition is not met. Need a certain environment variable for a test to work? Didn't get it? Skip the test.

1.12.3 Errored

An unexpected exception or warning occurred during the execution of the test.

1.12.4 Failed

The test was explicitly marked as failed via `Demeanor\TestContext::fail` or an assertion failed.

1.12.5 Filtered

This status is only used internally by demeanor to "skip" tests with out really skipping them. Filtered means some filter condition (like name or otherwise) was not met and the test was simply not executed.

1.12.6 How Test Result Statuses Influence CLI Exit Codes

If one or more tests fail or error, Demeanor will exit unsuccessfully. See *Exit Codes* for more information.

1.13 Exit Codes

When all tests are successful, Demeanor will exit with the status code `0`.

If tests fail or error, the exit code will `1`.

If some sort of error happens (configuration issue, etc) before tests are run, the exit code will be `2`.

If, for some reason, the `demeanor` command line script can't find a composer autoload file, it will exit with the status code `3`.

1.14 Upgrading to 0.4

Filters Now Require a Consensus

Previously if a test case met a single filter, it would run. Now a test case must meet all defined filters to run. For example:

```
./vendor/bin/demeanor --filter-name SomeTest --include-group aGroup
```

Only tests with *SomeTest* in their name that belong to the group *aGroup* will run.

CHAPTER 2

Getting Started

Demeanor can be installed with composer, please read the composer getting started page to learn how to get everything set up.

Once that's done, add `demeanor/demeanor` to your `require-dev` dependencies in `composer.json`.

```
{
    "require-dev": {
        "demeanor/demeanor": "dev-master"
    }
}
```

Then run `composer install` or `composer update` with the `--dev` flag.

2.1 Configuration

Other documentation explains the `demeanor.json` configuration more fully, but, for now, we're going to set up two test suites.

```
{
    "testsuites": {
        "unit": {
            "type": "unit",
            "directories": [
                "test/unit"
            ]
        },
        "spec": {
            "type": "spec",
            "directories": [
                "test/spec"
            ]
        }
    }
}
```

The `testsuites` argument is required and **must** be an object. If it's not the test runner will complain.

The keys of the `testsuites` object are the suite names and the values are their configuration. `type` tells Demeanor what type of test suite its dealing with. Valid values are `unit`, `spec`, or `phpt`. `directories` tells the test run where to look for the test files. How Demeanor finds those files varies by suite type.

2.2 Assertions

Demeanor uses a library called Counterpart to deal with assertions. You'll use the `Counterpart\Assert` class and call one of it's `assert*` methods. The last argument of all `assert*` methods is a message that can be used to describe the business case or importances of the assertion.

Here are some examples:

```
<?php
use Counterpart\Assert;

Assert::assertTrue(true, 'True is somehow false, things are very broken');
Assert::assertFalse(false);
Assert::assertNull(null);
Assert::assertType('string', 'this is a string');
```

2.3 Your First Unit Test

Unit test cases are methods inside of a class. Every time a method is run, a new instance of it's class is created.

Test class names **must** end with `Test` and test method must start with the word `test`. Demeanor will look for all files that end with `Test.php` in the directories defined in the `directories` configuration above.

```
<?php
// test/unit/TruthyTest.php

use Counterpart\Assert;

class TruthyTest
{
    public function testTruthyValuesReturnTrue()
    {
        Assert::assertTrue(filter_var('yes', FILTER_VALIDATE_BOOLEAN));
    }

    public function testFalsyValuesReturnFalse()
    {
        Assert::assertFalse(filter_var('no', FILTER_VALIDATE_BOOLEAN));
    }
}
```

2.4 Your First Spec Test

Spec tests use a `describe` and `it` API to to define a specification for an object. A specification is just a set of expected behaviors.

In demeanor, a spec test looks like this.

```
<?php
// filter_var.spec.php

use Counterpart\Assert;

/** @var Demeanor\Spec\Specification $this */
```

```
$this->describe('#truthyValues', function () {
    $this->it('should return true when given "yes"', function () {
        Assert::assertTrue(filter_var('yes', FILTER_VALIDATE_BOOLEAN));
    });
    $this->it('should return true when given a "1"', function () {
        Assert::assertTrue(filter_var(1, FILTER_VALIDATE_BOOLEAN));
    });
});

$this->describe('#falsyValues', function () {
    $this->it('should return false when given "no"', function () {
        Assert::assertFalse(filter_var('no', FILTER_VALIDATE_BOOLEAN));
    });
    $this->it('should return false when given "0"', function () {
        Assert::assertFalse(filter_var(0, FILTER_VALIDATE_BOOLEAN));
    });
});
```

Each call to `it` creates a new test case. When the `directories` argument is used for a `spec` test suite, all files that end with `.spec.php` are located and compiled to test cases.

See *Writing Spec Tests* for more information.

2.5 Running the Tests

A binary will be installed via composer in your `bin-dir` (`vendor/bin` by default). Once a configuration file and some tests are set up, use the command line to run `php vendor/bin/demeanor` or `./vendor/bin/demeanor` to run the tests.

2.6 Need More Examples?

Demeanor uses itself to test – well, to test itself. Look in the `test` directory of the demeanor repository for a bunch more examples.